



UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA MECCANICA INDUSTRIALE

Corso di Laurea
in AUTOMAZIONE INDUSTRIALE

Relazione Finale

CONTROLLO DI UN ROBOT SCARA CON ROS

Relatore: Chiar.mo Prof. Riccardo Adamini

Laureandi:
Francesco Baglioni
Matricola n. 723606

Francesco Campregher
Matricola n. 723547

Anno Accademico 2020/2021

Sommario

1	COSA È UN ROBOT SCARA?	1
1.1	APPLICAZIONI	2
2	ROSBOT, PRIMA E DOPO	3
2.1	OBIETTIVO	4
2.2	ROSBOT STRUTTURA MECCANICA	5
2.4	CINEMATICA E CONTROLLO	6
2.5	INTERCONNESSIONE	6
2.7	SCHEMA A BLOCCHI	7
3	FUNZIONAMENTO	9
3.1	AVVIO INIZIALE	9
3.2	CREAZIONE DI UN CICLO ESEGUITIBILE	10
3.3	CARICAMENTO DI UN CICLO ED ESECUZIONE	12
4	STRUTTURA DEL PROGETTO	15
5	CINEMATICA	17
5.1	CINEMATICA DIRETTA	17
5.2	CINEMATICA INVERSA	17
5.3	LEGGE DI MOTO	19
6	MECCANICA	21
6.1	I RIDUTTORI	22
6.2	I GIUNTI	23
6.2.1	Asse 1	23
6.2.2	Asse 2	24
6.2.3	Asse Z	26
7	ELETTRONICA	29
7.2	ATTUATORI ROTATIVI	30
7.2.1	<i>Motore Brushless</i>	30
7.2.2	<i>Struttura interna del driver Omron</i>	31
7.2.3	<i>PID</i>	32
7.2.4	<i>Regolazione dei parametri</i>	34
7.2.6	<i>Interfaccia del driver</i>	36
7.3	ATTUATORE LINEARE	39
7.3.1	<i>Il driver</i>	39

7.3.2	<i>Configurazione con LinMot Talk</i>	41
7.4	ATTUATORE DI PRESA	44
7.5	MICROCONTROLLORI.....	45
7.5.1	<i>Hercules RM57Lx</i>	45
7.5.1.1	Struttura	46
7.5.1.2	HALCoGen.....	48
7.5.1.4	Code Composer Studio	54
7.5.1.6	Comunicazione	67
7.5.1.8	Cablaggio	68
7.5.3	<i>Arduino Mega</i>	72
7.5.3.1	Programma	72
7.5.3.2	Cablaggio	72
7.5.5	<i>Arduino DUE</i>	74
7.5.5.1	Programma	74
7.5.5.2	Cablaggio	74
7.7	CONDIZIONAMENTO.....	77
7.7.1	<i>Struttura</i>	78
7.7.1.1	Uscite 24 V con amplificatori operazionali	78
7.7.1.2	Uscite a transistor.....	78
7.7.1.3	Uscite 5 V ad alta frequenza	79
7.7.2	<i>Connettori</i>	80
7.7.2.1	Hercules RM57Lx	80
7.7.2.2	Driver Omron	81
7.7.2.3	Driver LinMot.....	82
7.7.2.4	Arduino Mega	82
7.7.4	<i>Alimentazioni</i>	83
9	INTERFACCIA CON IL MONDO ESTERNO: ROS	85
9.1	STRUTTURA DI ROS	86
9.2	MESSAGGI	86
9.3	TOPIC.....	87
9.4	NODI	87
9.4.1	<i>ROS e ROSBOT</i>	87
9.4.2	<i>Installazione</i>	89
9.4.4	<i>ROS Workspace</i>	91
9.4.5	<i>Il messaggio</i>	91
9.4.6	<i>Il nodo server</i>	92
9.4.8	<i>Il nodo seriale</i>	96
9.4.9	<i>Il client</i>	98
10	POSSIBILI MIGLIORAMENTI	103

10.1	ROS2	103
10.2	SISTEMA DI VISIONE	103
10.3	PERIFERICHE AGGIUNTIVE.....	103
10.4	IMPLEMENTAZIONE DI NUOVE MOVIMENTAZIONI	103
12	RINGRAZIAMENTI	105

1 Cosa è un robot SCARA?

Lo SCARA, acronimo di Selective Compliance Assembly Robot Arm, è un tipo di robot industriale, che muove un "braccio" sul piano orizzontale e una presa che può salire e scendere in quello verticale. Il primo e secondo asse sono di rotazione, il terzo e il quarto asse sono generalmente lineari, realizzati con viti a ricircolo di sfere. La presa è montata sulla parte finale dell'asse verticale. Uno SCARA può permettere raggi di azione da 100 mm a 1.200 mm, con capacità di carico che varia da 1 kg a 200 kg.

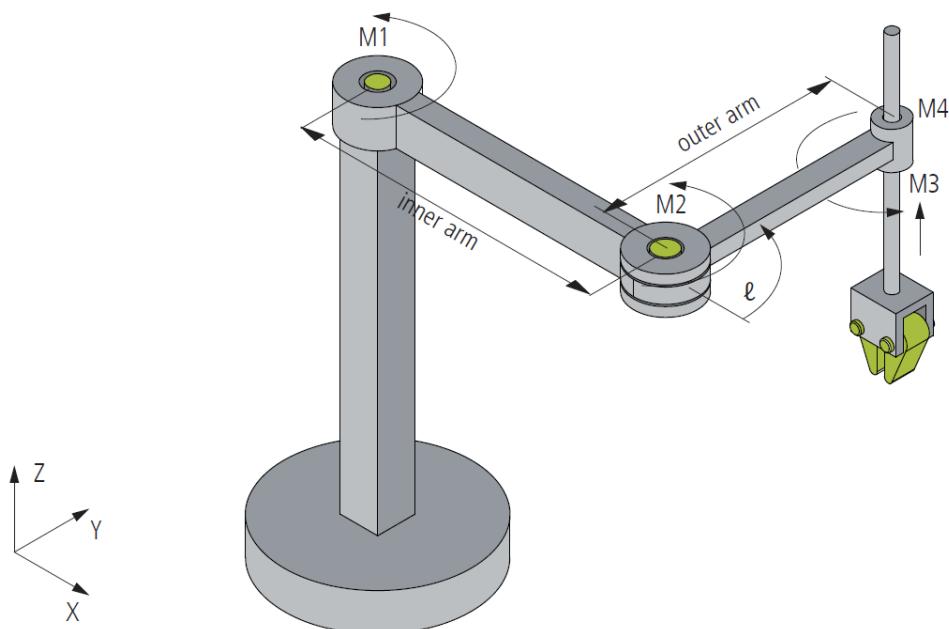


Figura 1: Schema di un robot SCARA

La velocità è un fattore importante per la scelta di un robot: gli SCARA sono spesso fra i prodotti più veloci sul mercato. Essendo dotati di quattro assi, questi robot utilizzano un basso numero di giunti in movimento e sono configurati per fare in modo che M1 e M2 gestiscano il movimento X-Y e M3 e M4 il movimento di rotazione e Z. Ciò consente di semplificare e velocizzare i calcoli della cinematica inversa. Nei casi in cui è fondamentale gestire il tempo del ciclo, è consigliabile prendere in considerazione una soluzione SCARA.

1.1 Applicazioni

Questo tipo di robot venne sviluppato per alte velocità e ripetibilità in montaggi in serie, come il Pick-and-Place da un posto ad un altro. Gran parte delle piccole applicazioni di montaggio, avvitamento e di prelievo e scarico basate sul movimento di una parte dal punto A al punto B sono ideali per i robot SCARA, poiché richiedono di solito lo spostamento di un componente nello spazio con una certa rotazione attorno all'asse Z. Questa architettura si limita però a lavorare nel piano, nel caso in cui servisse una manipolazione a 6 assi generalmente si scelgono robot antropomorfi.

Nell'immagine seguente si vede un esempio di una cella FANUC per realizzare pacchi batteria i cui componenti devono essere spostati dai nastri trasportatori alla tavola rotante con movimenti a 4 gradi di libertà (X, Y, Z, RZ).



Figura 2: Cella con robot SCARA FANUC

2 ROSBOT, prima e dopo

ROSBOT è il robot SCARA che abbiamo realizzato basandoci su un precedente progetto da noi sviluppato presso l'istituto superiore ITIS Benedetto Castelli di Brescia. In origine il progetto era meccanicamente ed elettricamente completo, ma abbiamo deciso di riprogettarlo sotto tutti i punti di vista mantenendo solamente alcuni componenti elettrici tra cui i servomotori e i relativi driver.

Di seguito è riportata un'immagine del progetto originale:



Figura 3: Prima versione di ROSBOT

La versione attuale di ROSBOT è invece riportata nelle seguenti immagini:

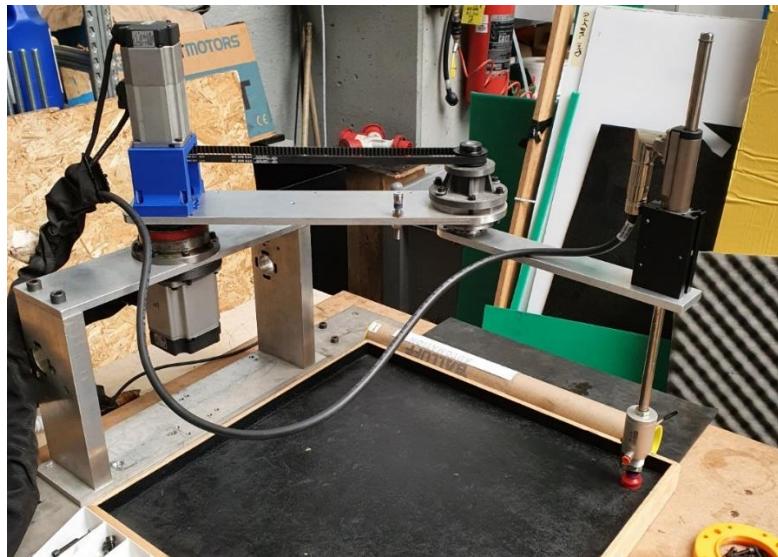


Figura 4: Nuova versione di ROSBOT

2.1 Obiettivo

Il progetto nasce dal nostro desiderio di realizzare un robot verosimilmente industriale, di grandi dimensioni e con componenti professionali. L'obiettivo, scelto con il prof. Riccardo Adamini, è quello di realizzare un prodotto completo predisposto all'interconnessione con il mondo aziendale grazie a ROS. Inoltre un elemento chiave del progetto è la possibilità di espansione delle sue funzioni come ad esempio un sistema di visione.

2.2 ROSBOT Struttura meccanica

ROSBOT è un robot SCARA 3 assi (XYZ) con un raggio di lavoro massimo di 630mm, un'escursione verticale massima di 245mm ed una capacità di carico di 0.3 Kg. Monta due servomotori Omron per gli assi M1 e M2 e un motore magnetico lineare LinMot per l'asse Z.

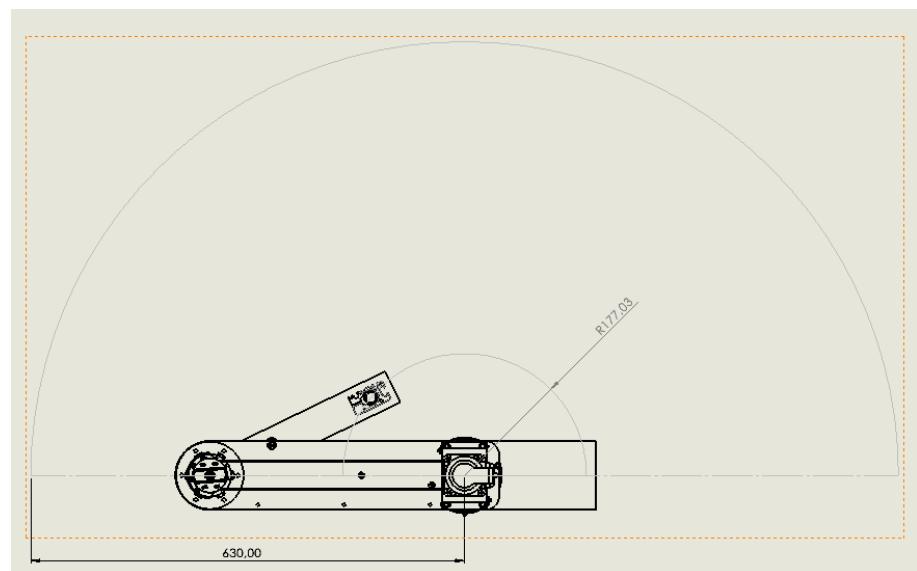


Figura 5: Area di lavoro

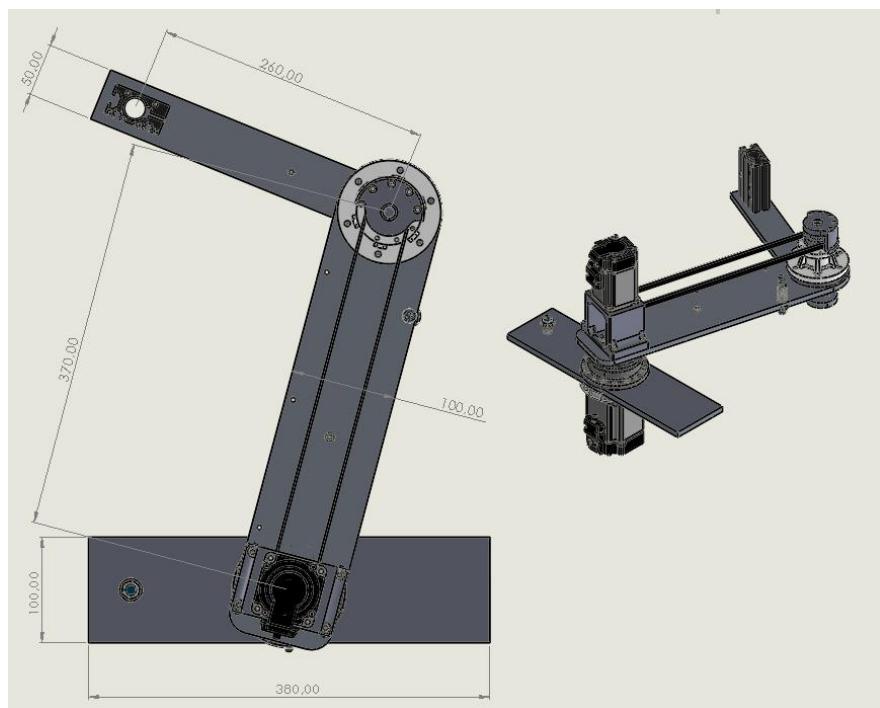


Figura 6: Vista in pianta

2.3 Cinematica e Controllo

Abbiamo sviluppato la cinematica inversa per movimenti degli assi non correlati tra loro, le movimentazioni lineare e circolare sono invece in fase di sviluppo. I calcoli relativi al controllo di ROSBOT sono stati implementati in un microcontrollore Hercules rm57lx della Texas Instruments. Il microcontrollore si interfaccia con tutti i componenti del robot attraverso una scheda di condizionamento di segnale appositamente progettata da noi che permette il corretto controllo dei driver e svolge funzioni di sicurezza.

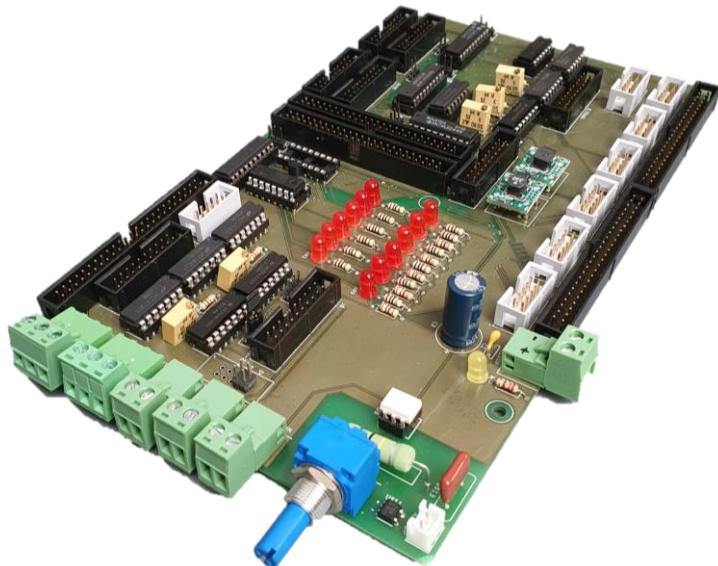


Figura 7: Scheda di condizionamento dei segnali

2.4 Interconnessione

I comandi vengono inviati al microcontrollore via rete attraverso un protocollo di comunicazione TCP-IP e un nodo ROS che permette l'interconnessione di ROSBOT con altri macchinari. Il controllo del robot può quindi avvenire attraverso un programma di comunicazione TCP-IP da un qualsiasi dispositivo connesso alla stessa rete a cui è connesso ROSBOT. Questo prodotto è anche dotato di un software compatibile con sistemi Windows di controllo e simulazione sviluppato da noi che permette di replicare i movimenti dello SCARA e di inviare comandi.

2.5 Schema a blocchi

Il seguente schema a blocchi aiuterà a capire la struttura di ROSBOT e ci guiderà nei prossimi capitoli.

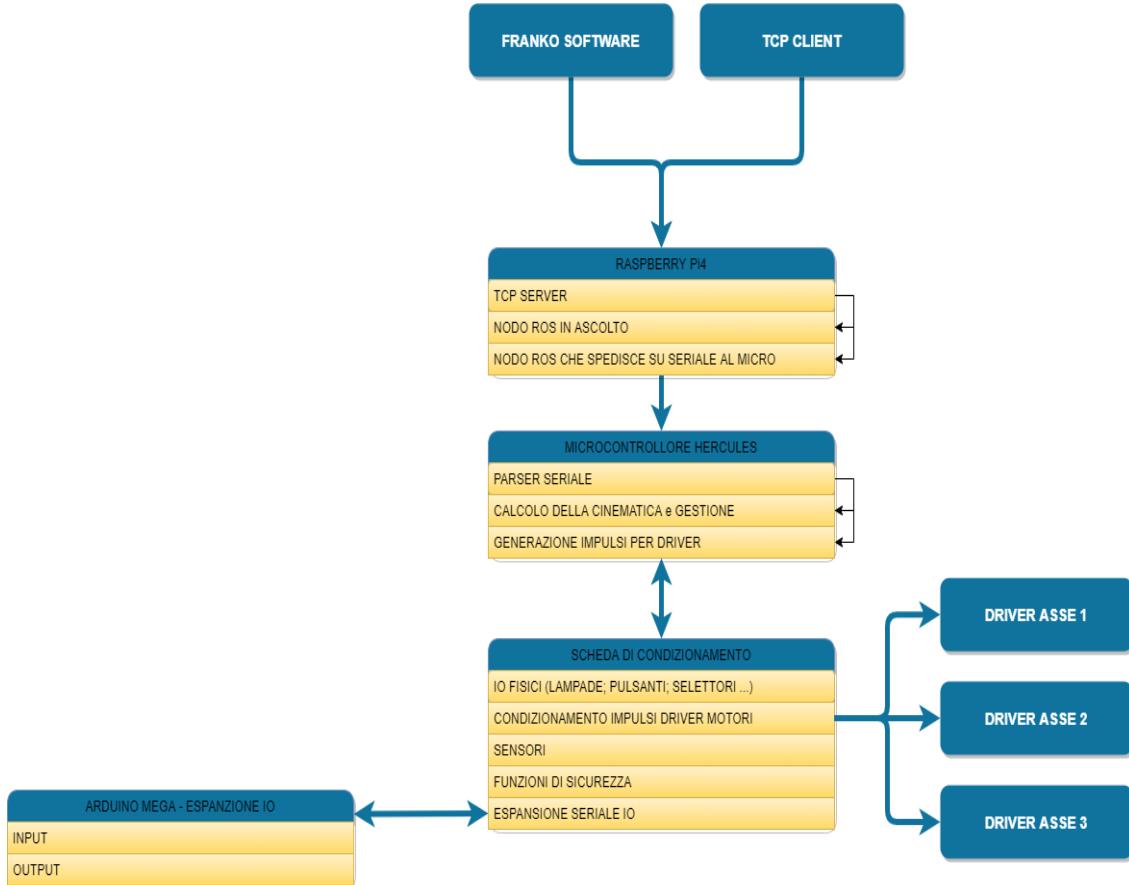


Figura 8: Schema a blocchi del progetto

3 Funzionamento

ROSBOT dispone di un set di istruzioni che si possono utilizzare per comporre programmi che esso è in grado di ripetere ciclicamente. Queste funzioni vengono svolte da un programma che spiegheremo in seguito.

3.1 Avvio iniziale

Per prima cosa bisogna abilitare l'interruttore differenziale per alimentare il sistema. Successivamente collegare il microcontrollore a Raspberry, il quale dovrà poi essere acceso.

Accertarsi che la connessione LAN sia abilitata. In seguito digitare il comando *ifconfig* per determinare l'indirizzo IP di Raspberry, che servirà in seguito.

Accedere al terminale di Raspberry e digitare i seguenti comandi ognuno su un terminale distinto nella sequenza indicata:

```
roscore  
rosrun scara_pkg server_SCARA.py  
rosrun scara_pkg serial_SCARA.py
```

Avviare il programma *ROSBOT-Client.exe* e inserire l'indirizzo IP ricavato in precedenza nell'apposita casella denominata *IP*:

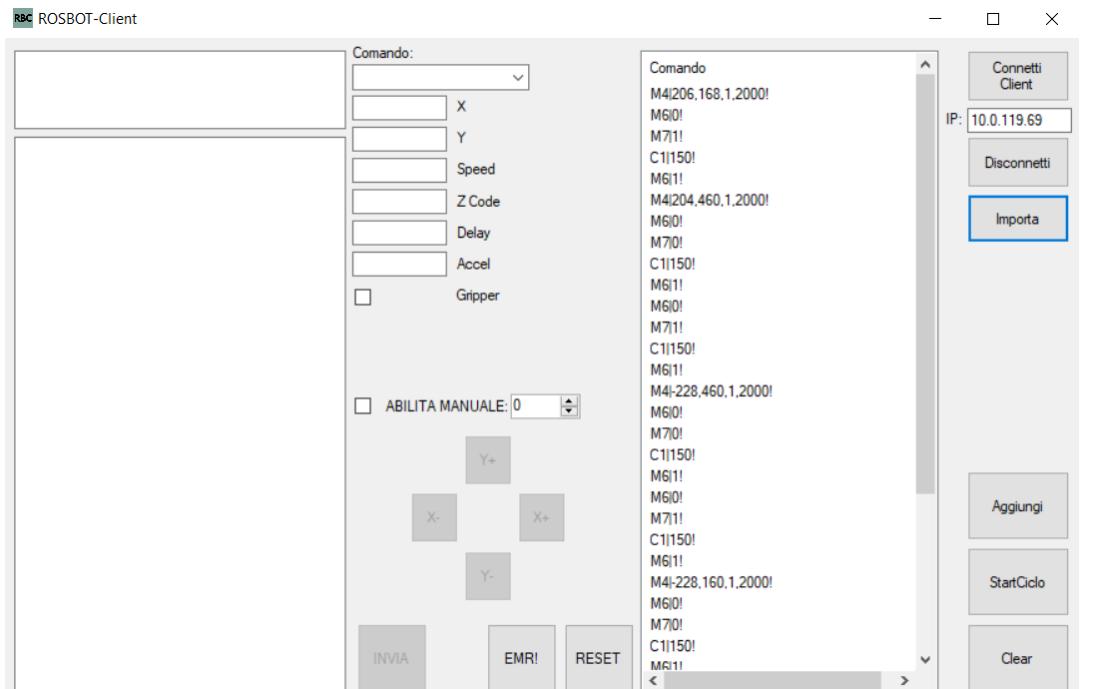


Figura 9: ROSBOT-Client.exe

3.2 Creazione di un ciclo eseguibile

Un ciclo eseguibile da ROSBOT è composto dalle seguenti istruzioni:

Comando	Descrizione
C1,T!	Genera uno stato di attesa nel robot. Al posto di T viene inserito il tempo in millisecondi.
S2!	Resetta lo stato del robot, esso viene utilizzato in seguito a uno stato di emergenza per riavviare i motori.
S3!	Pone il robot in uno stato di emergenza.
S4,A!	Setta il valore dell'accelerazione che viene utilizzata nella movimentazione del robot. Al posto di A viene inserito il valore di accelerazione in mm/s^2.
M4,X,Y,V!	Esegue un movimento dei motori non correlato in cui raggiunge un punto del piano (X,Y) espresso in mm, considerando l'asse del primo motore l'origine del sistema di riferimento. Per conoscere l'orientamento degli assi affidarsi al sistema di riferimento riportato nelle pagine successive.
M5,P,S,V!	Esegue un movimento nello spazio dei giunti anche quando la routine di homing non è stata eseguita. Al posto di P e S bisogna inserire i gradi che si vogliono far compiere al primo e al secondo asse, considerando la convenzione dove gli angoli percorsi in senso antiorario sono positivi, e al posto di V viene inserita la velocità periferica degli assi in [mm/s]. Questa movimentazione è stata pensata per le fasi di set-up del sistema, quando il robot si trova in configurazioni non idonee alla procedura di homing.
M6,S!	Imposta la posizione dell'asse Z, al posto di S bisogna inserire 0 per posizionare l'asse Z nella posizione inferiore e 1 per quella superiore
M7,S!	Imposta lo stato dell'organo di presa, al posto di S bisogna inserire 1 per impostare uno stato di presa mentre 0 per il rilascio
M8!	Esegue la routine di homing del robot

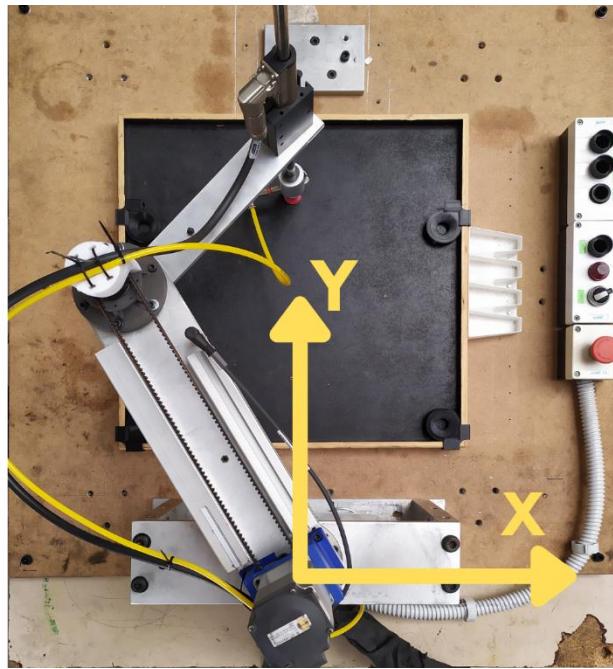


Figura 10: Sistema di riferimento ROSBOT

La lista di comandi deve essere inserita all'interno di un file .txt, che poi sarà caricata all'interno del programma. Qui di seguito è riportato un esempio di programma.

```
M4,-100,340,2000!
M6,0!
M7,1!
C1,150!
M6,1!
M4,72,510,2000!
M6,0!
M7,0!
C1,150!
M6,1!
M4,300,300,2000!
M4,72,510,2000!
M6,0!
M7,1!
C1,150!
M6,1!
M4,-100,340,2000!
M6,0!
M7,0!
C1,150!
```

M6,1!

3.3 Caricamento di un ciclo ed esecuzione

Per caricare un programma compilato bisogna seguire i seguenti passi:

1. Premere il tasto *Importa*

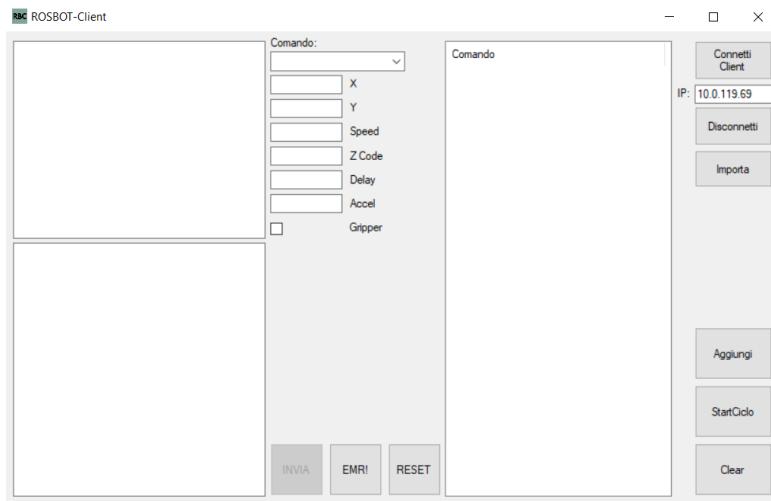


Figura 11: ROSBOT-client importa

2. Selezionare il file .txt del programma che si vuole eseguire

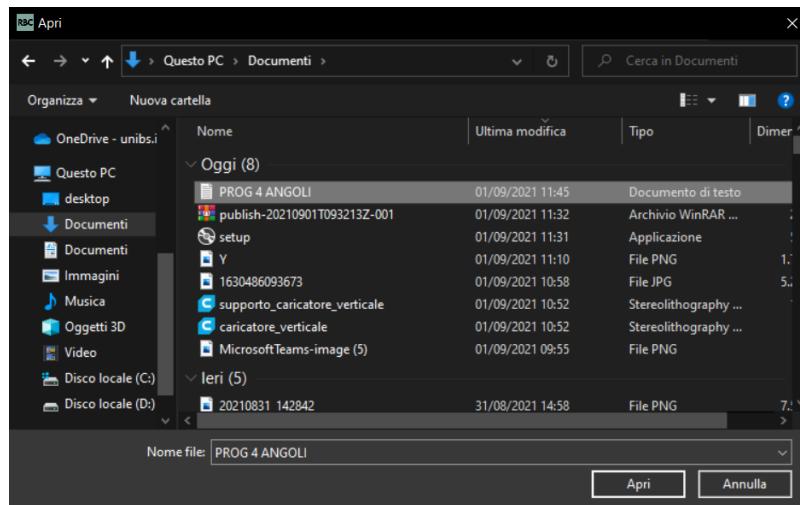


Figura 12: ROSBOT-Client selezione file .txt

Una volta importato si troverà la situazione successivamente illustrata

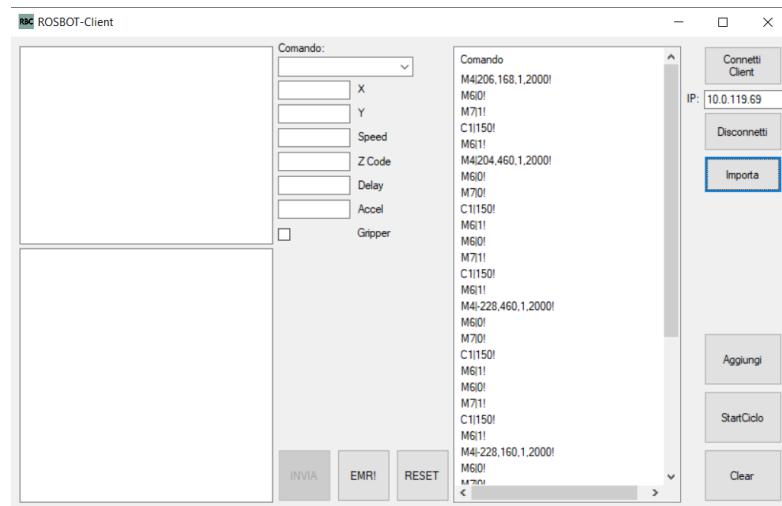


Figura 13: ROSBOT-Client programma caricato

3. Per avviare il programma basta premere il pulsante *StartCiclo*.

4 Struttura del progetto

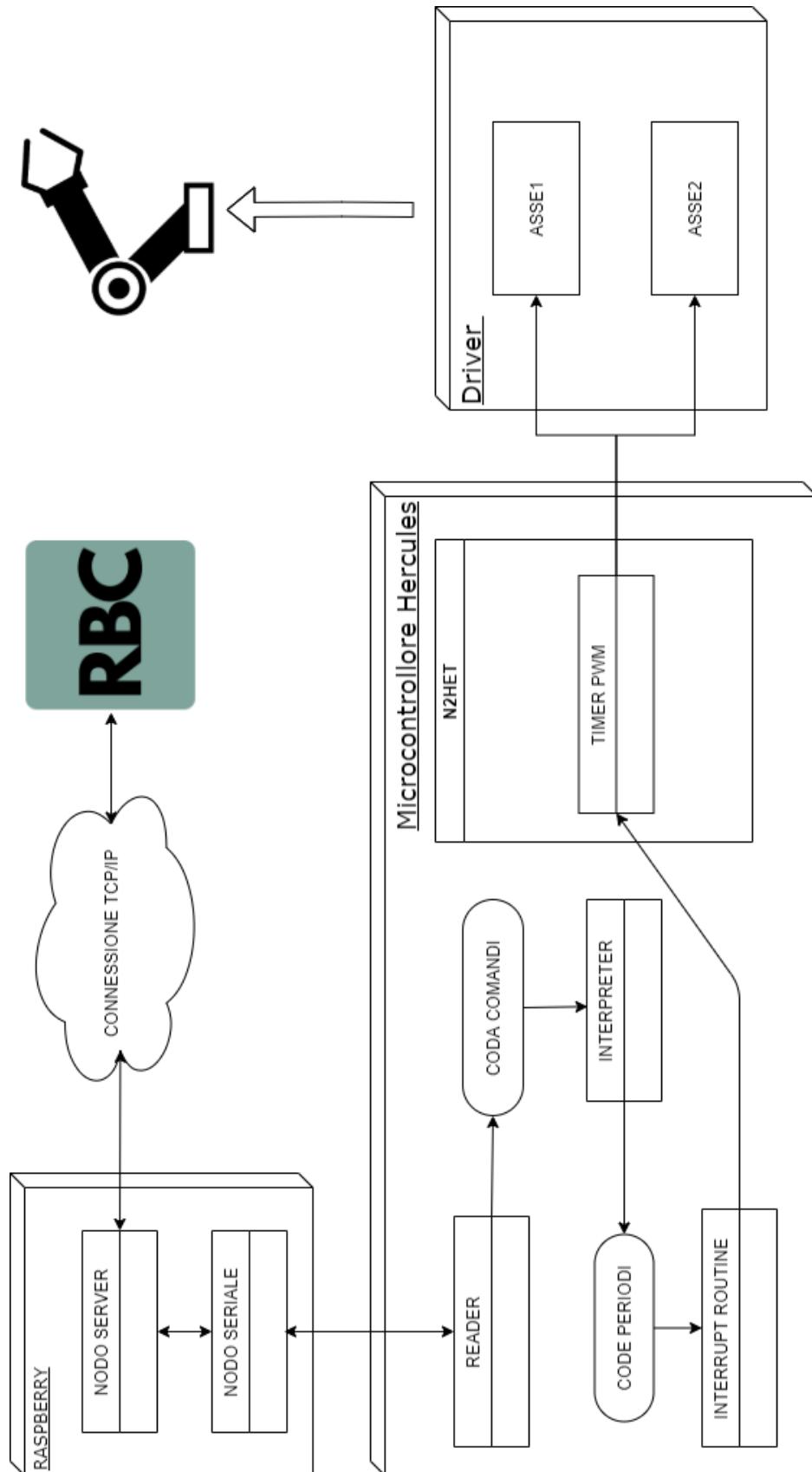


Figura 14: Struttura logica di ROSBOT

5 Cinematica

5.1 Cinematica diretta

Nel problema della cinematica diretta sono noti gli angoli di rotazione assoluti dei giunti e si ricavano le coordinate del punto terminale *TCP*. La risoluzione si basa sulla semplice applicazione delle leggi trigonometriche di base.

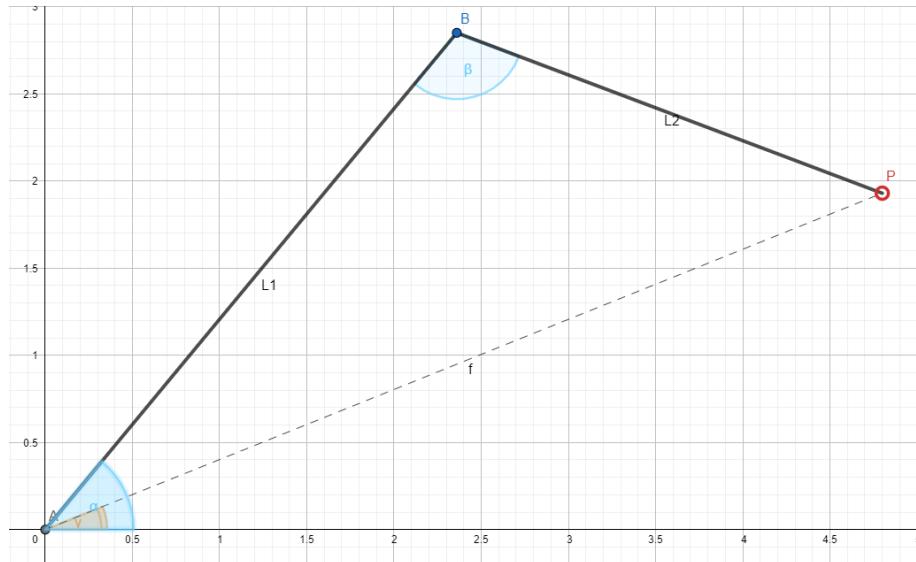


Figura 15: ROSBOT visto in pianta

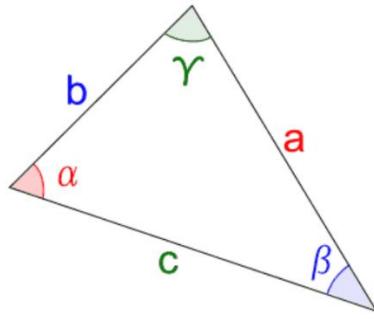
$$X_p = L_1 \cdot \cos(\alpha) + L_2 \cdot \sin\left(\beta - \left(\frac{\pi}{2} - \alpha\right)\right)$$

$$Y_p = L_1 \cdot \sin(\alpha) - L_2 \cdot \cos\left(\beta - \left(\frac{\pi}{2} - \alpha\right)\right)$$

5.2 Cinematica inversa

Nel problema della cinematica inversa sono note le coordinate del TCP e si vogliono calcolare gli angoli che gli assi devono assumere affinché la posizione venga raggiunta. La risoluzione avviene tramite il teorema de coseno detto anche di Carnot.

Enunciato: in un triangolo qualsiasi il quadrato della misura di un lato è dato dalla somma dei quadrati delle misure degli altri due lati, meno il loro doppio prodotto moltiplicato per il coseno dell'angolo tra essi compreso.



$$c^2 = b^2 + a^2 - 2 \cdot b \cdot a \cdot \cos(\gamma)$$

$$\gamma = \cos^{-1} \left(\frac{c^2 - b^2 - a^2}{-2 \cdot b \cdot a} \right)$$

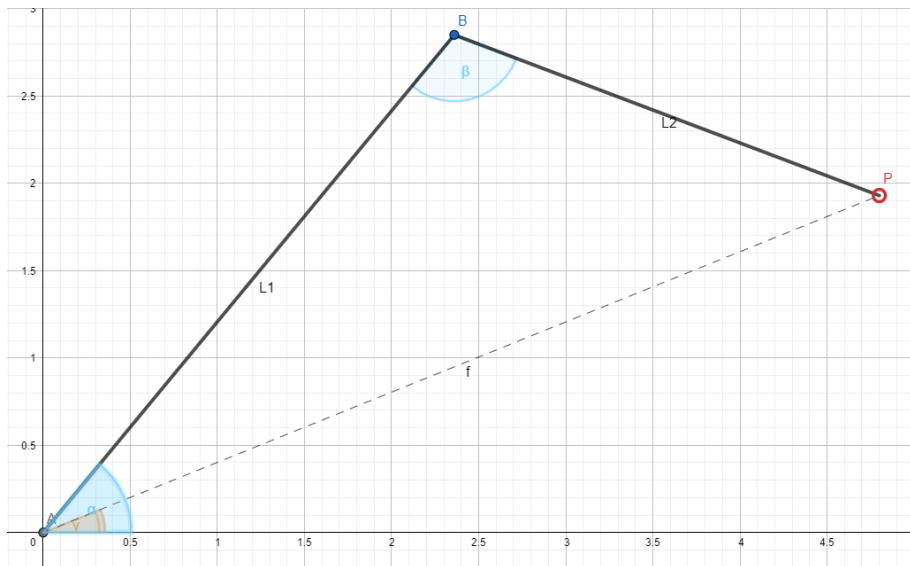


Figura 16: ROSBOT visto in pianta

Considerando il triangolo ABP, essendo noti tutti i lati, possiamo applicare il teorema di Carnot per calcolare l'angolo α - γ e l'angolo β . Il lato AB misura 370mm, il lato BC 260mm e la lunghezza del lato AP si ottiene applicando pitagora:

$$f = \sqrt{X_p^2 + Y_p^2}$$

Per ottenere l'angolo α bisogna calcolare l'angolo γ nel modo seguente:

$$\gamma = \text{atan2}(Y_p, X_p)$$

Quindi i calcoli da implementare nel microcontrollore sono i seguenti:

$$\alpha = \gamma + \cos^{-1} \left(\frac{-L_2^2 + L_1^2 + f^2}{2 \cdot L_1 \cdot f} \right)$$

$$\beta = \cos^{-1} \left(\frac{-L_1^2 + L_2^2 + f^2}{2 \cdot L_2 \cdot f} \right)$$

5.3 Legge di moto

La legge di moto adottata è chiamata tre tratti. Essa è la formula che descrive la velocità assunta dal robot in ogni istante di tempo e come suggerito dal suo nome si può scomporre in tre tratti:

- **Accelerazione:** è la prima parte del movimento dove il robot modifica la sua velocità da una velocità minima fino alla velocità massima.

$$V = a \cdot t$$

- Dove V è la velocità attuale, a è l'accelerazione del robot e t è il tempo attuale.
- **Velocità costante** è il secondo tratto dove la velocità rimane costante.

$$V = V_{max}$$

- **Decelerazione** è il terzo tratto dove la velocità diminuisce fino ad arrivare alla velocità minima.

$$V = V_{max} - a \cdot t$$

Questa movimentazione ha il vantaggio di essere facile da implementare su un microcontrollore e richiede poca potenza di calcolo, però ha un'accelerazione discontinua che può generare oscillazioni durante gli spostamenti.



Figura 17: Legge di moto Tre tratti

6 Meccanica

La progettazione meccanica è stata effettuata in SolidWorks 2020 e i componenti sono stati realizzati in alluminio e in PET-G stampato in 3D. Meccanicamente il progetto è semplice infatti è composto da due aste e 3 giunti: La prima asta è incernierata al telaio tramite un riduttore 1:50 e alla seconda tramite un altro riduttore 1:50, la seconda asta permette lo scorrimento verticale dello stelo dell'asse Z alla sua estremità.

I due motori brushless rotativi sono montati sullo stesso asse in modo da ridurre il momento di inerzia del braccio, il moto del motore che muove la seconda asta è trasmesso tramite una cinghia dentata con rapporto 1:1.

Avremmo potuto ridurre ulteriormente il momento di inerzia portando il riduttore dell'asse 2 nel fulcro dell'asse 1 però i giochi causati dalla trasmissione a cinghia si sarebbero ripercossi direttamente sulla seconda asta. Mantenendo il riduttore a valle della trasmissione a cinghia invece i giochi “passano attraverso il riduttore” e vengono ridotti di 50 volte. Inoltre questi riduttori ci permettono di ottenere giunti molto rigidi e robusti

Nella figura seguente si vede bene la struttura essenziale:

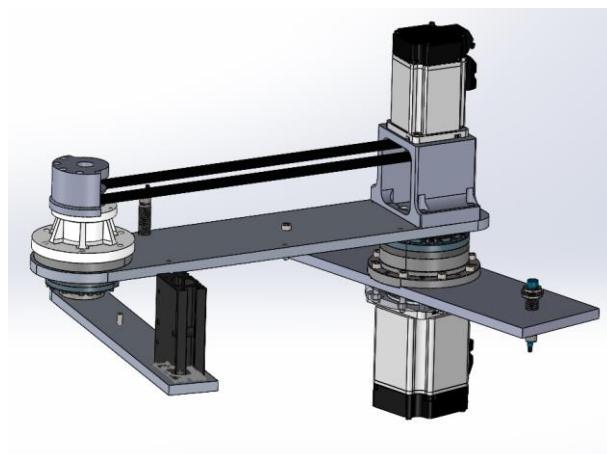


Figura 18: La struttura meccanica

6.1 I riduttori

I riduttori che abbiamo utilizzato sono Harmonic Drive HFUC-20-2UH e HFUC-25-2UH con rapporto di riduzione 1:50. Di seguito sono riportate alcune caratteristiche da [catalogo](#):

Tabella 1: Datasheet Armonic Drive

Table 10.2

	Unit	HFUC-20-2UH						HFUC-25-2UH					
		30	50	80	100	120	160	30	50	80	100	120	160
Ratio	i []	30	50	80	100	120	160	30	50	80	100	120	160
Repeatable peak torque	T _p [Nm]	27	56	74	82	87	92	50	98	137	157	167	176
Average torque	T _A [Nm]	20	34	47	49	49	49	38	55	87	108	108	108
Rated torque	T _R [Nm]	15	25	34	40	40	40	27	39	63	67	67	67
Momentary peak torque	T _M [Nm]	50	98	127	147	147	147	95	186	255	284	304	314
Maximum input speed (oil lubrication)	n _m (max) [rpm]	10000						7500					
Maximum input speed (grease lubrication)	n _m (max) [rpm]	6500						5600					
Average input speed (oil lubrication)	n _{av} (max) [rpm]	6500						5600					
Average input speed (grease lubrication)	n _{av} (max) [rpm]	3500						3500					
Moment of inertia	J _m [x10 ⁻⁴ kgm ²]	0.193						0.413					
Weight	m [kg]	0.98						1.5					

La lubrificazione è effettuata con grasso quindi si possono raggiungere velocità di ingresso minori rispetto alla lubrificazione con olio, ma sono comunque sufficienti per la movimentazione di ROSBOT.

Sono riduttori indicati per applicazioni ad alta dinamica che abbiamo smontato da un robot Comau guasto.



Figura 19: Harmonic Drive

I riduttori armonici hanno una struttura molto semplice che permette di ottenere elevati rapporti di riduzione in poco spazio, inoltre hanno teoricamente gioco nullo.

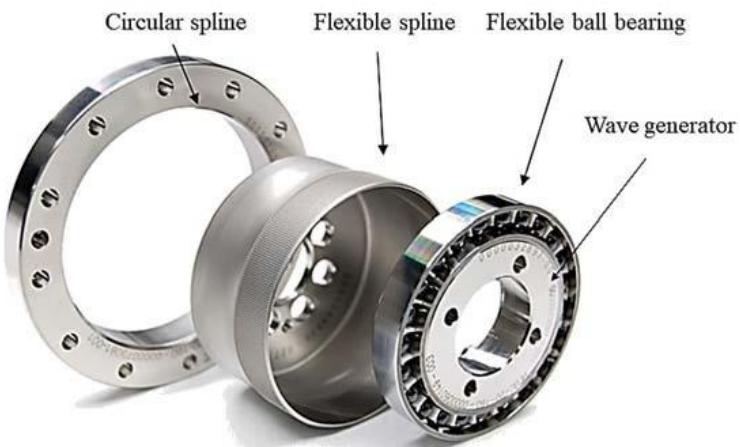


Figura 20: Come è fatto un Harmonic Drive

In primo piano si vede il generatore di onde a cui è calettato il motore in ingresso, esso è simile ad un cuscinetto di forma ellittica che ruotando genera deformazioni sul rotore flessibile dentato a cui è calettato il carico. Infine si ha una struttura rigida dentata esterna all'interno della quale ruota il rotore.

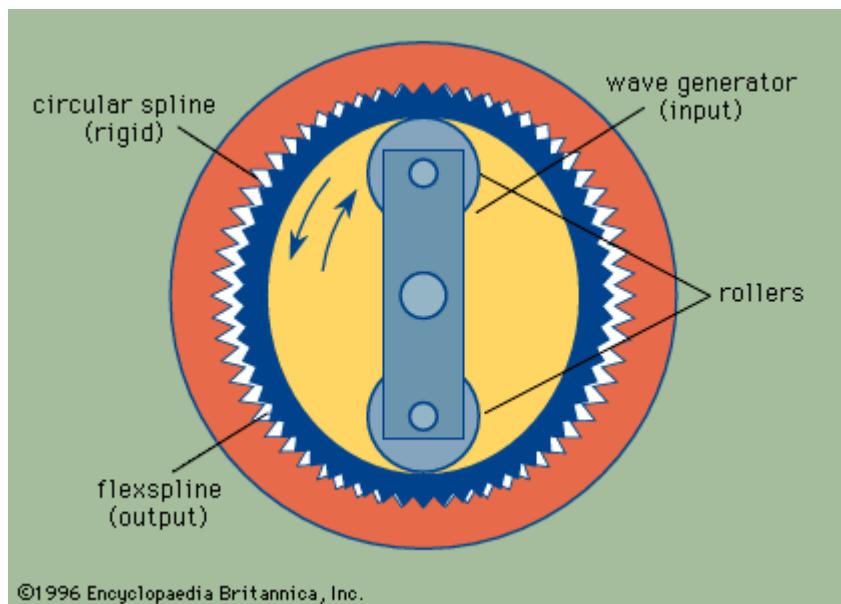


Figura 21: Funzionamento di un Harmonic Drive

6.2 I giunti

I giunti, grazie alla compattezza dei riduttori, hanno una struttura molto semplice:

6.2.1 Asse1

Come si vede in figura l'albero motore entra direttamente nel riduttore, per ottenere un corretto centraggio dei componenti abbiamo realizzato un foro di centraggio $\phi 68$ nel telaio:

Il riduttore ha una flangia $\phi 68$, il motore è concentrato all'anello di centraggio che è a sua volta concentrato al telaio. In questo modo tutti i componenti sono in asse tra loro.

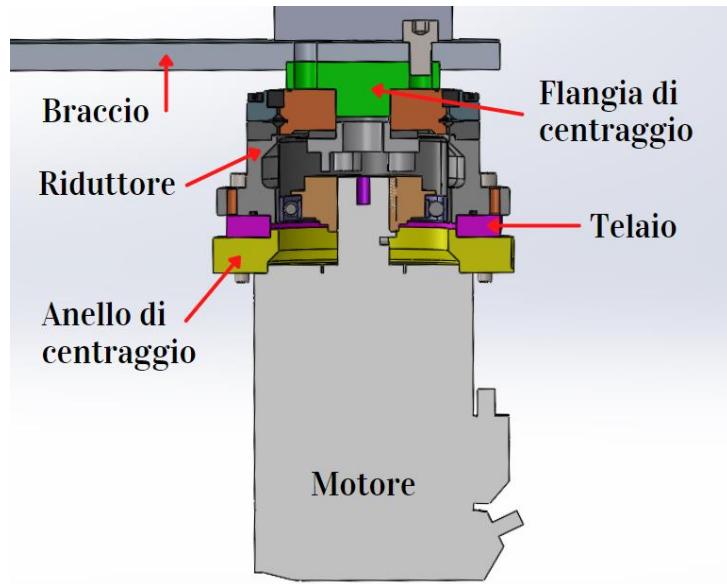


Figura 22: Componenti del primo giunto

Il montaggio del braccio invece avviene tramite 8 viti M8 che si filettano nel riduttore. Braccio e riduttore sono distanziati di 7mm grazie ad una flangia di centraggio in acciaio interposta.

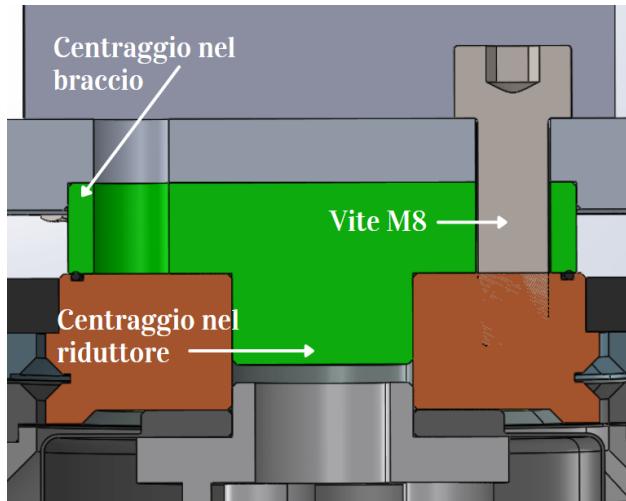


Figura 23: Fissaggio al braccio

6.2.2 Asse 2

Per quanto riguarda il secondo giunto invece la potenza arriva attraverso una trasmissione a cinghia, infatti nella foto seguente si vede una puleggia calettata ad un albero che entra nel riduttore. Il riduttore non regge sforzi radiali dal lato input quindi abbiamo progettato un banco cuscinetti con due 6201 RS distanziati di 10mm per mantenere l'albero coassiale al riduttore.

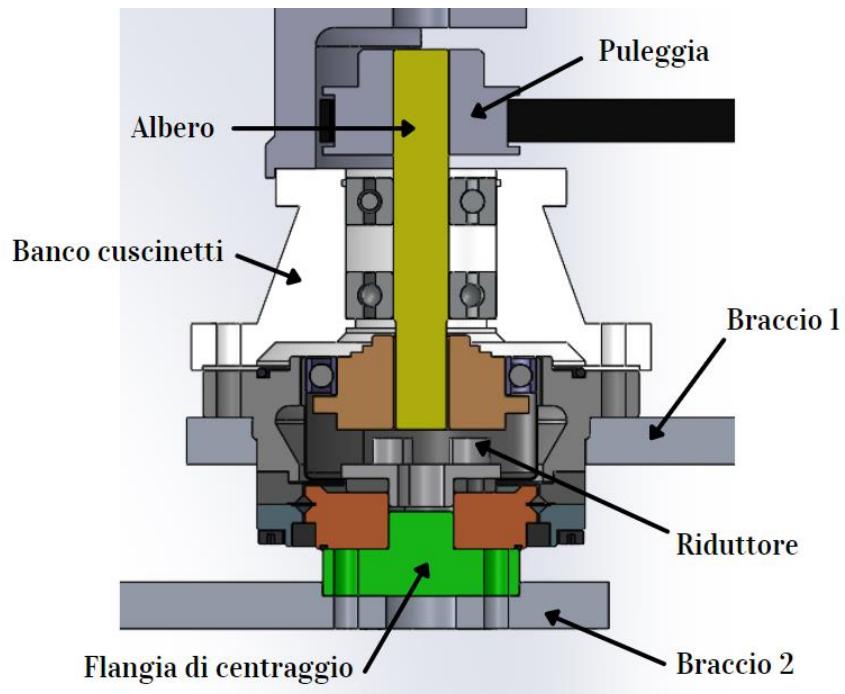


Figura 24: Componenti del secondo giunto

Il sistema di fissaggio al secondo braccio è analogo a quello utilizzato nel primo giunto.

La trasmissione a cinghia permette di posizionare il secondo motore in una posizione centrata rispetto all'asse di rotazione così da ridurre l'inerzia di ROSBOT. Il riduttore è stato comunque lasciato all'estremità del primo braccio per i seguenti motivi:

1. I giochi e le elasticità dovute alla cinghia vengono ridotte di 50 volte.
2. La coppia che deve trasmettere la cinghia non è amplificata di circa 50 volte.

Il tensionamento della cinghia avviene grazie a delle asole che permettono di muovere il motore del secondo asse. In fase di montaggio o manutenzione si svita il motore e lo si avvicina alla puleggia condotta per rimuovere la cinghia. In fase di montaggio si fa calzare la cinghia sulla puleggia motrice, si tira il motore in modo da aumentare la tensione e poi lo si fissa.

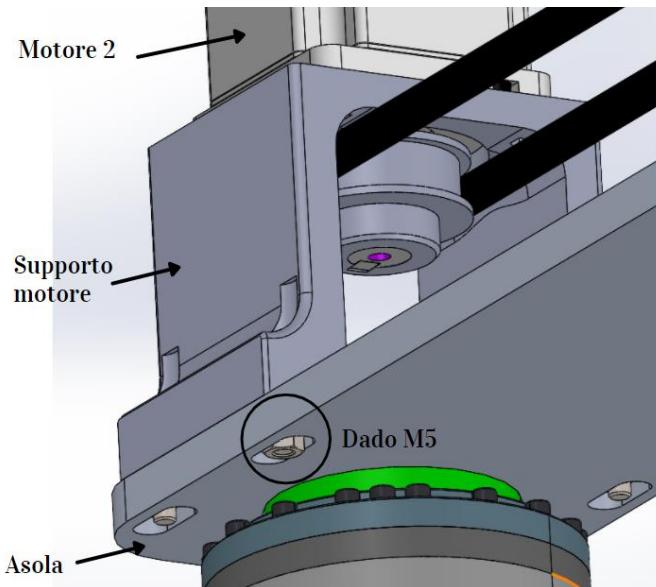


Figura 25: Tensionamento cinghia

6.2.3 Asse Z

Il giunto dell'asse Z è costituito dal motore stesso.

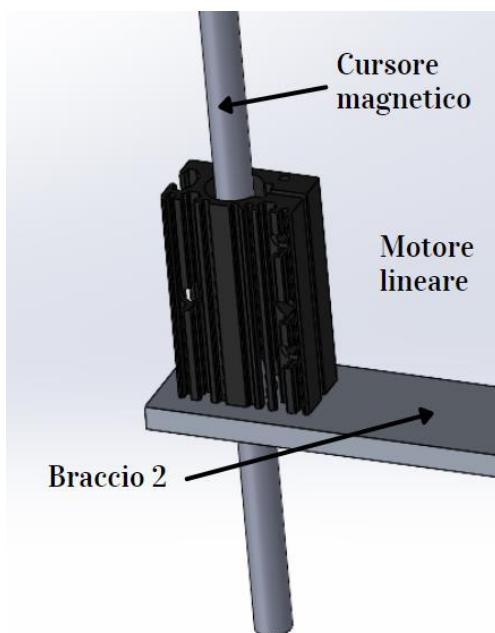


Figura 26: Giunto Z

Il cavo di questo motore deve essere fissato in modo da ridurre al minimo le torsioni e gli spostamenti per minimizzare l'usura ed impedire che si attorcigli o impigli. Abbiamo risolto questo problema tramite un perno folle che permette al cavo di ruotare liberamente rispetto al braccio 2.

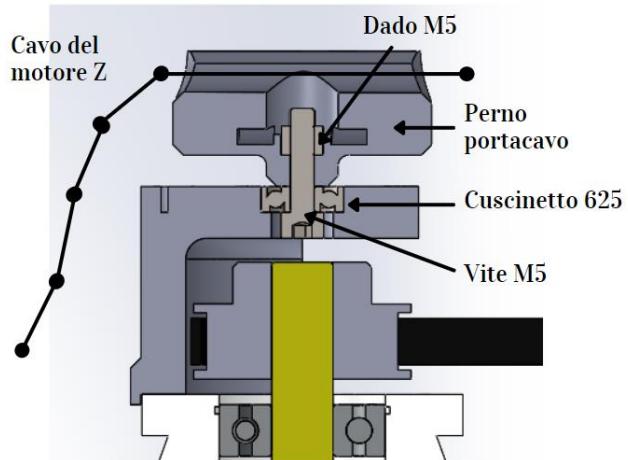


Figura 27: Perno portacavo

7 Elettronica

Il quadro elettrico può essere suddiviso in quattro zone.

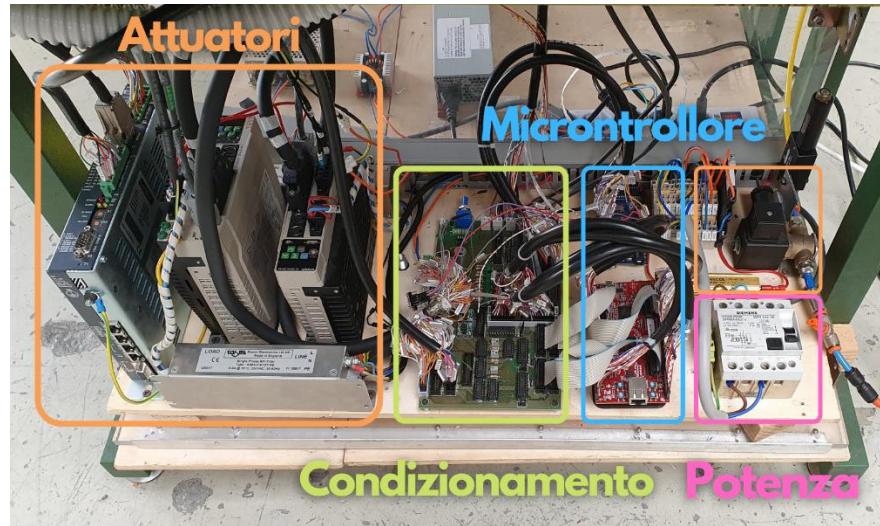


Figura 28: Quadro elettrico

Gli attuatori si occupano di trasformare gli impulsi elettrici in movimentazioni o azioni grazie ad un'elettronica di potenza. I microcontrollori risolvono la cinematica del robot generando i segnali necessari al suo funzionamento. Il condizionamento dei segnali viene eseguito da una scheda elettronica che si occupa di far comunicare correttamente fra di loro gli attuatori e i microcontrollori. La potenza viene distribuita attraverso dei morsetti e sezionata da un interruttore differenziale.

7.1 Attuatori rotativi

Per questo progetto sono stati adottati due servomotori rotativi Omron.

7.1.1 Motore Brushless

I motori Omron impiegati sono di tipo sincrono brushless. Questa tipologia di azionamenti è utilizzata in ambito industriale per le sue elevate prestazioni e la sua elevata precisione che conferiscono al progetto una notevole accuratezza nei movimenti e nel posizionamento. Questa tipologia di motori è composta da uno statore avvolto, un rotore a magneti permanenti e un sensore di posizione.

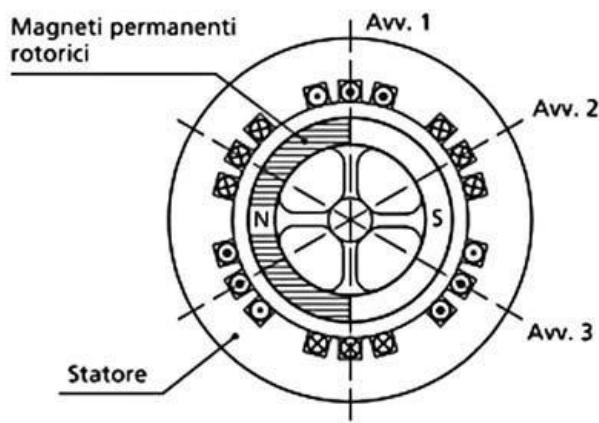


Figura 29: Struttura di un motore brushless

Il loro principio di funzionamento si basa sulla generazione di un campo magnetico rotante all'interno dello statore attraverso tre diversi avvolgimenti che vengono alimentati con tre correnti sfasate tra di loro di 120° elettrici. Queste correnti vengono generate sfruttando la tecnica del PWM, essa consiste nel generare un segnale ad onda quadra con frequenza fissa e duty cycle variabile. Il campo magnetico rotante viene generato in modo da mantenere un angolo di 90° rispetto a quello di rotore durante la movimentazione per ottenere la coppia maggiore possibile, il sensore di posizione quindi assume una funzione fondamentale per conoscere istante per istante la posizione del rotore.

7.1.2 Struttura interna del driver Omron

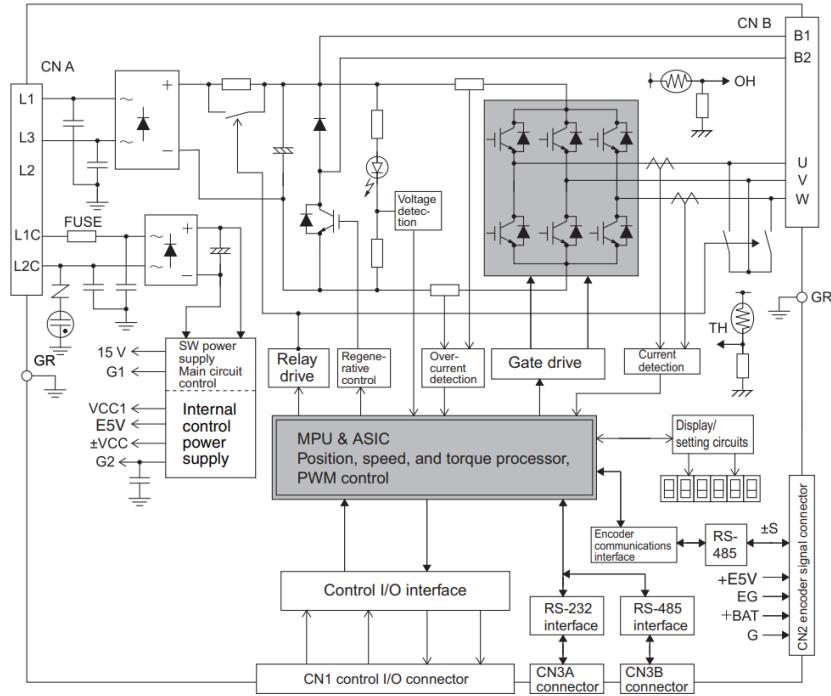


Figura 30: Struttura driver Omron

Come si può osservare dalla figura precedente, che rappresenta la struttura dell'elettronica di controllo del motore, vi sono due parti fondamentali evidenziate in grigio: gli interruttori elettronici di potenza e il microprocessore.

Gli interruttori elettronici di potenza sono dei componenti che permettono di generare delle correnti sinusoidali all'interno degli avvolgimenti il cui nome tecnico è *IGBT* (*Insulated Gate Bipolar Transistor*).

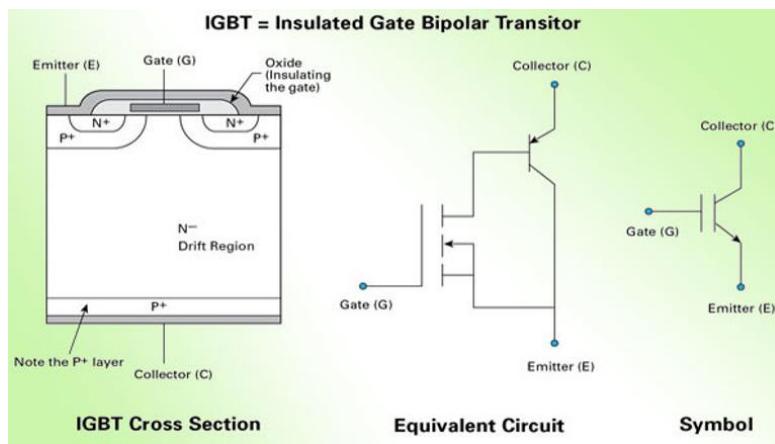


Figura 31: IGBT

Questa tipologia di transistor permette di sfruttare i vantaggi sia dei transistor MOSFET che BJT. Il comando in tensione dei primi permette il controllo con correnti di assorbimento quasi nulle e quindi perdite di potenza minime. Il comando diretto della corrente dei secondi permette una generazione semplice e precisa di forme d'onda diverse.

In questo caso la generazione della forma d'onda viene effettuata tramite dei segnali *PWM* (*Pulse Width Modulation*) comandati dal microprocessore. Quest'ultimo riceve in ingresso i comandi e i dati rilevati dall'encoder e li interpola tra loro attraverso un controllo digitale di tipo *PID* (*Proportional Integrative Derivative*) per controllare la movimentazione del motore.

7.1.3 PID

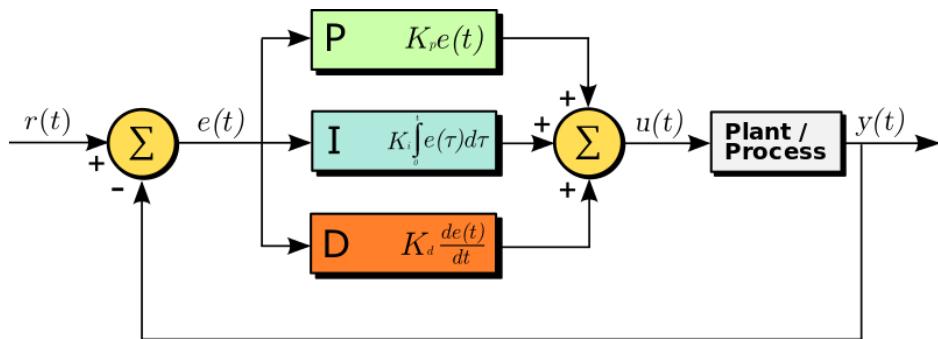


Figura 32: Struttura PID

I controllori PID, come illustrato dalla figura, sono composti da un valore di riferimento $r(t)$ a cui viene sottratto il valore attuale della grandezza controllata $y(t)$ (posizione, velocità ...). Il risultato di questa operazione viene chiamato errore $e(t)$ che viene elaborato dalle tre componenti illustrate di seguito:

- P (azione proporzionale): fornisce un'azione la cui entità dipende direttamente dal valore dell'errore. Questa componente da sola non permette di raggiungere il valore di riferimento desiderato infatti vi sarà sempre un errore.
- I (azione integrativa): fornisce un'azione la cui entità è direttamente proporzionale all'integrale nel tempo della variabile errore. Questa componente tiene conto della storia del sistema e permette di azzerare l'errore rallentando il sistema.
- D (azione derivativa): fornisce un'azione la cui entità dipende direttamente dalla derivata dell'errore. Questa componente cerca di prevedere l'andamento futuro della grandezza controllata. Essa aumenta la velocità del sistema, però aumentando il rischio che esso diventi instabile.

Questo controllore viene caratterizzato dai tre parametri che moltiplicano le tre azioni (kp , Ti , Td). Questi coefficienti vengono settati attraverso vari metodi empirici. Il più facile da utilizzare è quello per due punti che si basa sul grafico della risposta forzata del sistema soggetto a scalino.

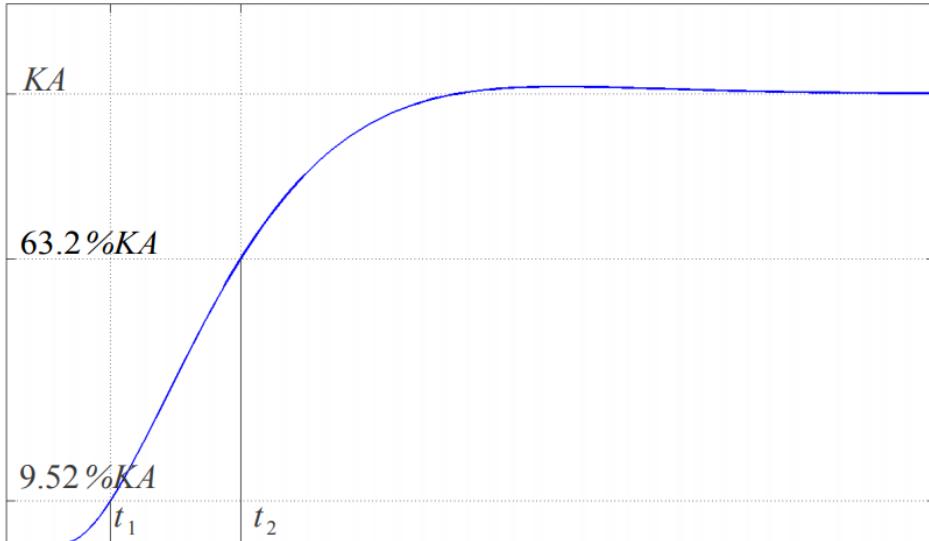


Figura 33: risposta allo scalino

A è il guadagno del sistema, t_1 è il tempo in cui il sistema raggiunge il 9.52% del valore di regime e t_2 è il tempo in cui il sistema raggiunge il 63.2%. K invece indica l'ampiezza dello scalino, solitamente si adotta K unitario. Risolvendo il seguente sistema

$$\begin{cases} t_1 = 0.1\tau + L \\ t_2 = \tau + L \end{cases}$$

Si ottengono i valori τ e L che a loro volta vengono utilizzati nella seguente tabella per ricavare i valori dei parametri (kp , Ti , Td) corretti.

Tabella 2: Settaggio parametri PID

Tipo di Controllore	Kp	Ti	Td
P	$\frac{\tau}{KL}$		
PI	$0.9 \frac{\tau}{KL}$	$3L$	
PID	$1.2 \frac{\tau}{KL}$	$2L$	$0.5L$

7.1.4 Regolazione dei parametri

I parametri dei driver possono essere regolati tramite il programma CX-Drive, che comunica attraverso una comunicazione seriale RS232. La procedura di settaggio è la seguente:

1. Collegare al driver il cavo per la comunicazione seriale al connettore CN3B e al computer attraverso l'adattatore RS232 -> USB.

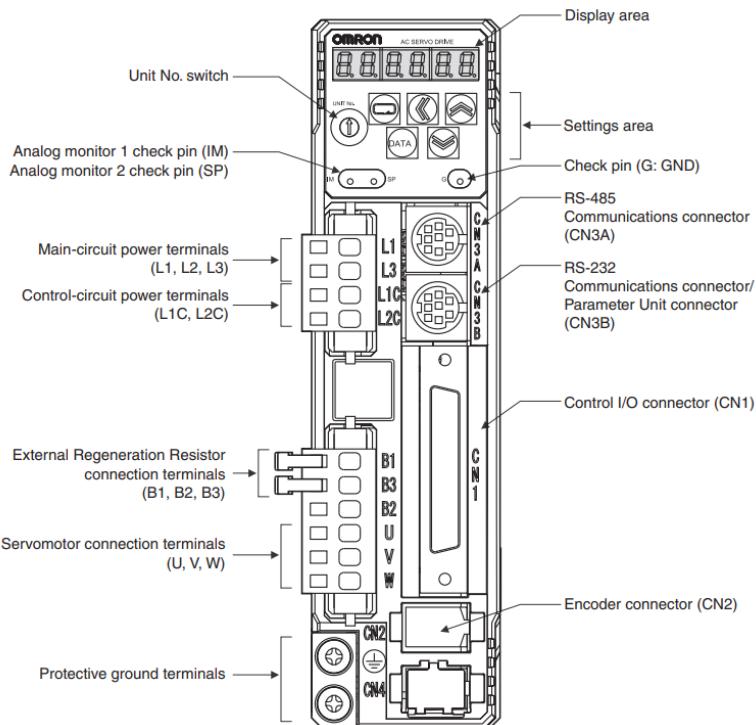


Figura 34: Connettori driver Omron

2. Avviare il programma CX-Drive sul computer.

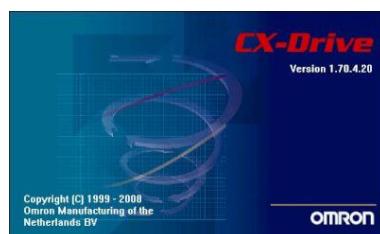


Figura 35: CX-Drive

3. Una volta aperto configurare un nuovo driver della famiglia desiderata e impostare la porta USB corrispondente.
4. Eseguire il collegamento online del driver utilizzando il pulsante evidenziato nella figura seguente.

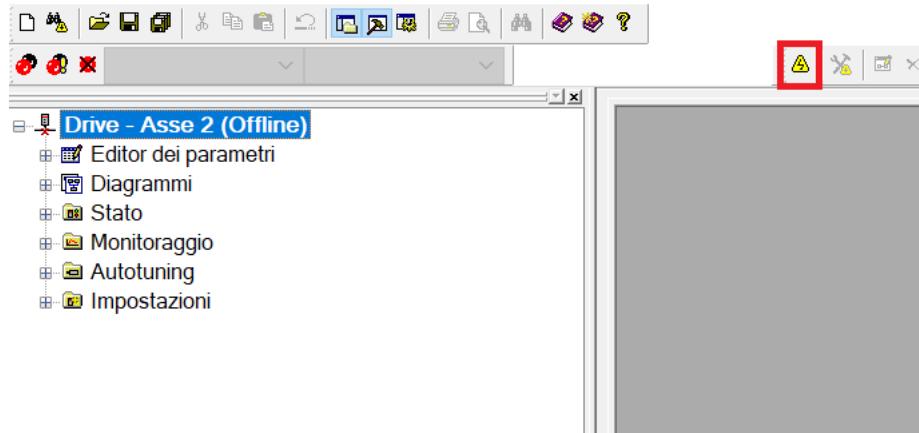


Figura 36: Collegamento driver Omron

5. Aprire il gestore dei parametri dove è possibile settare tutti i parametri del driver.

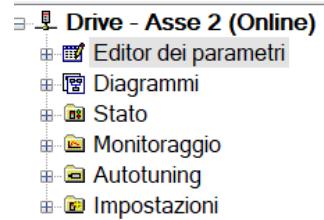


Figura 37: Editor dei parametri

In alternativa si può utilizzare i diagrammi per cambiare i parametri necessari.

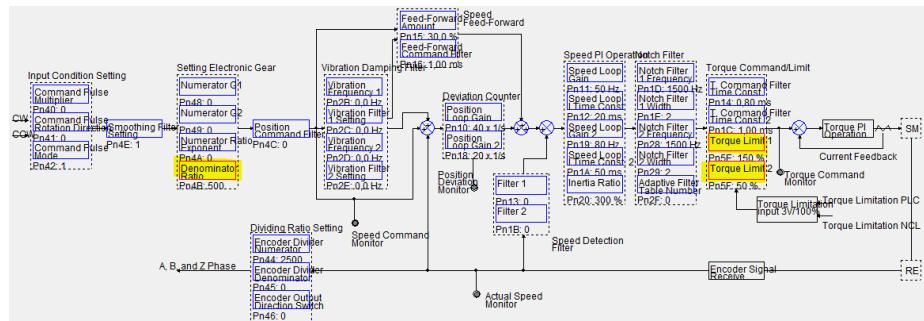


Figura 38: Diagramma controllo in posizione

I parametri da noi cambiati sono quelli evidenziati in giallo. Per *denominator ratio* si intendono il numero di passi in cui viene suddiviso l'angolo giro, *Torque Limit 1* e *2* rappresentano le limitazioni della coppia. Quest'ultimo parametro è stato diminuito perché le coppie limite dei riduttori sono inferiori a quelle dei motori.

7.1.5 Interfaccia del driver

Il driver fornisce tre tipi di gestione del motore.

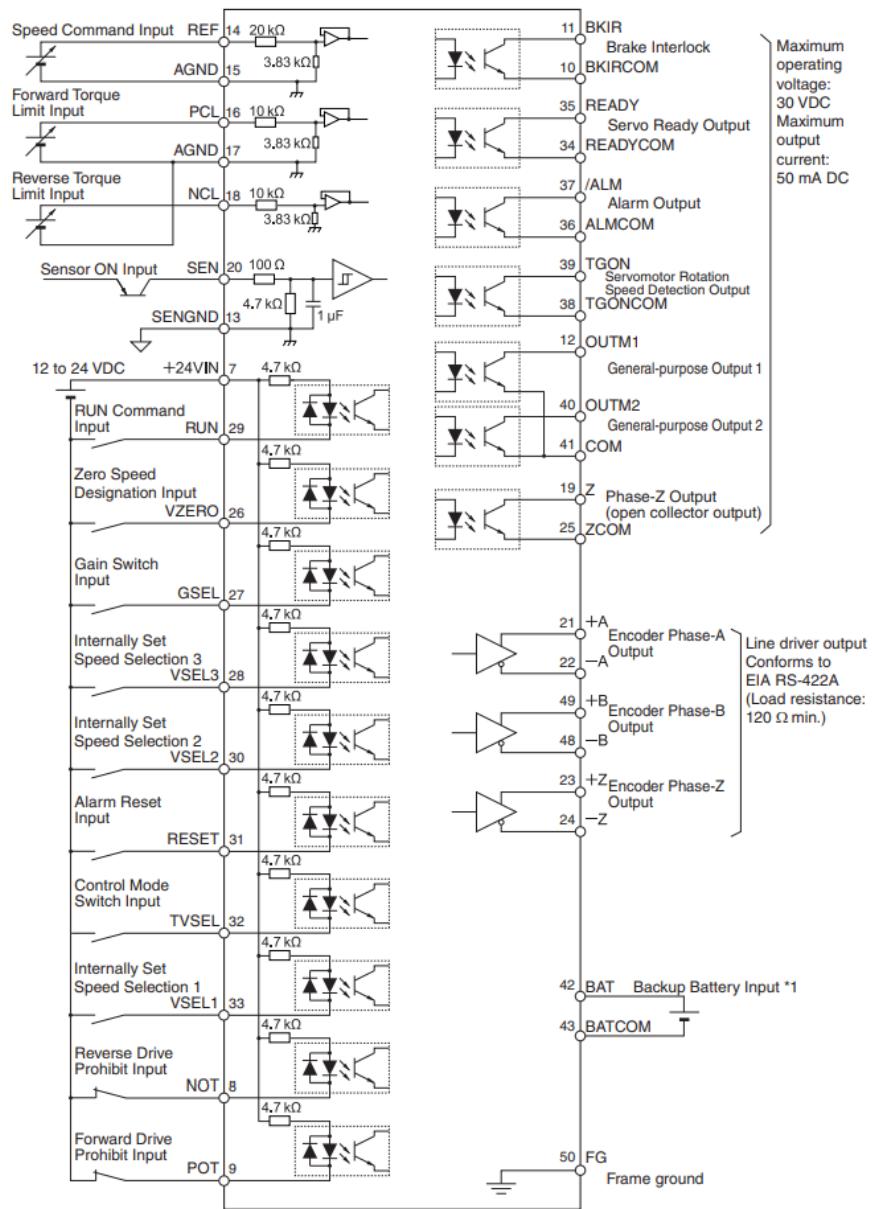


Figura 39: Comando in velocità

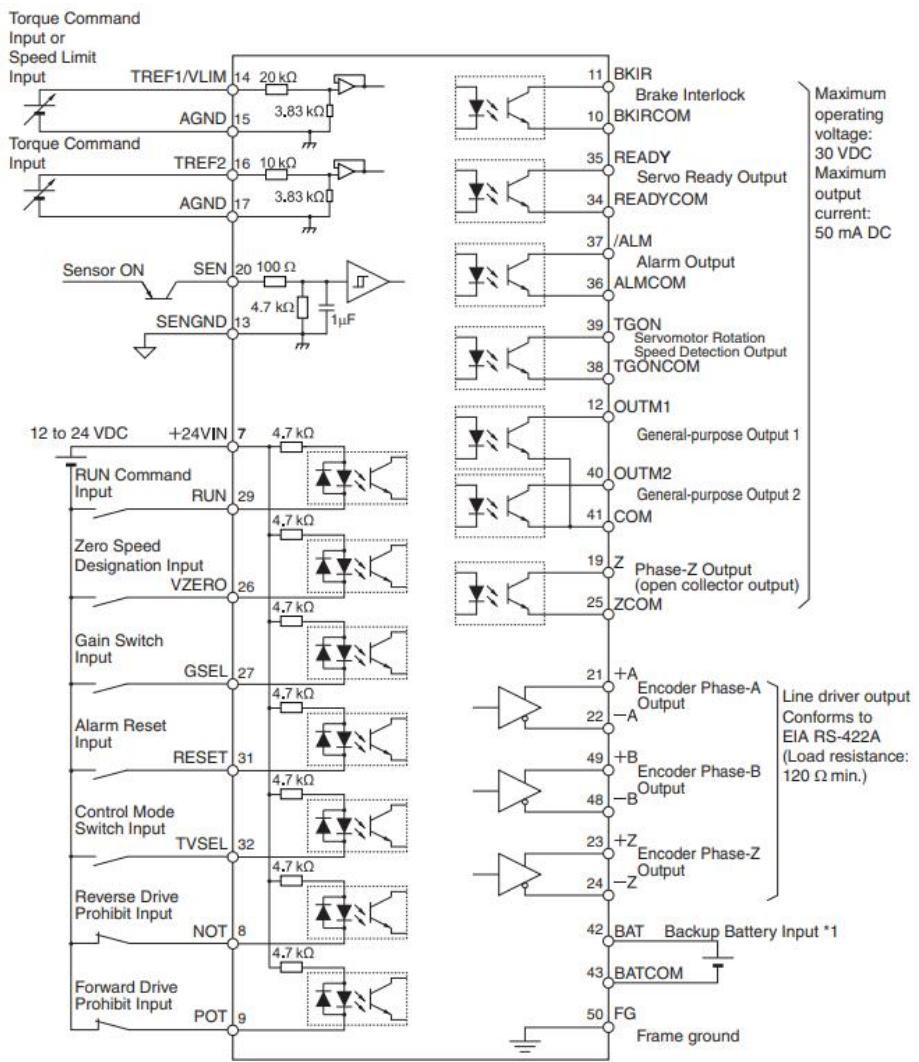


Figura 40: Comando in coppia

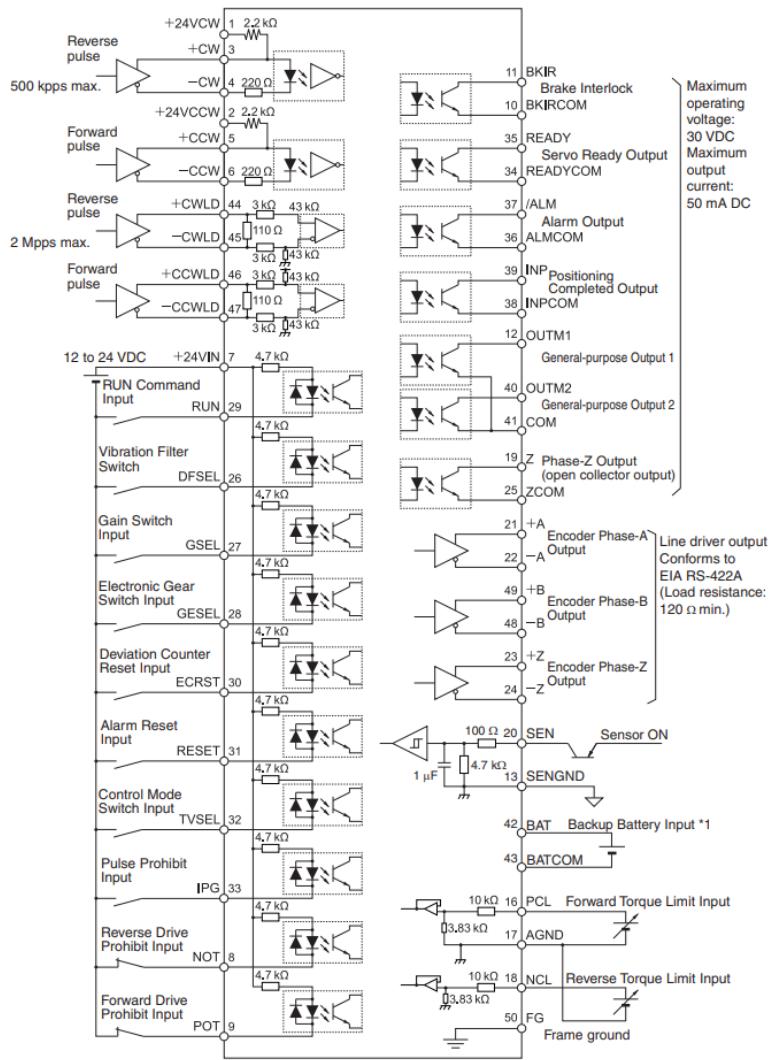


Figura 41: Comando in posizione

Il metodo di controllo da noi utilizzato è quello di posizione con una gestione dei motori Omron ad impulsi.

Tabella 3: Pin utilizzati driver Omron

Pin	Nominazione	Descrizione
3	+CW	Ingresso degli impulsi che faranno ruotare il motore in verso orario
4	-CW	Riferimento di massa per l'ingresso per gli impulsi orari
5	+CCW	Ingresso degli impulsi che fanno muovere il motore in verso antiorario
6	-CCW	Riferimento di massa per l'ingresso per gli impulsi antiorari
7	+24VIN	Riferimento per gli ingressi optoisolati
8	NOT	Ingresso negato che inibisce l'ingresso degli impulsi per la rotazione oraria
9	POT	Ingresso negato che inibisce l'ingresso degli impulsi per la rotazione antioraria
21	+A	Fase A dell'encoder
22	-A	Riferimento della fase A dell'encoder
23	+Z	Fase Z dell'encoder
24	-Z	Riferimento della fase Z dell'encoder
34	READYCOM	Riferimento per il segnale READY
35	READY	Segnale che determina quando il driver è pronto a ricevere i comandi
36	ALMCOM	Riferimento per il segnale /ALM
37	/ALM	Segnale di allarme del driver
48	-B	Riferimento della fase B dell'encoder
49	+B	Fase B dell'encoder

7.2 Attuatore lineare

L'asse Z di ROSBOT è costituito da un motore brushless lineare con primario fisso e secondario mobile LinMot PS01-23x80-R con driver E1100-RS.

Il principio di funzionamento è lo stesso del brushless rotativo spiegato precedentemente: il primario genera un campo magnetico traslante che interagisce con il secondario, un sensore di posizione fornisce il feedback della posizione del secondario al driver che controlla il primario di conseguenza.

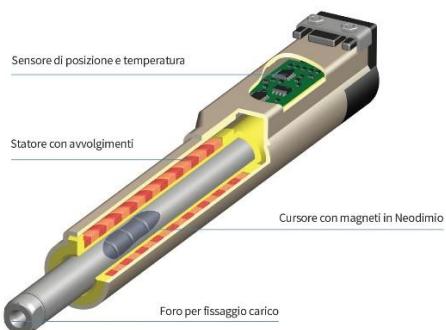


Figura 42: Struttura di un motore brushless lineare

7.2.1 Il driver

Il driver del motore ha la seguente struttura:

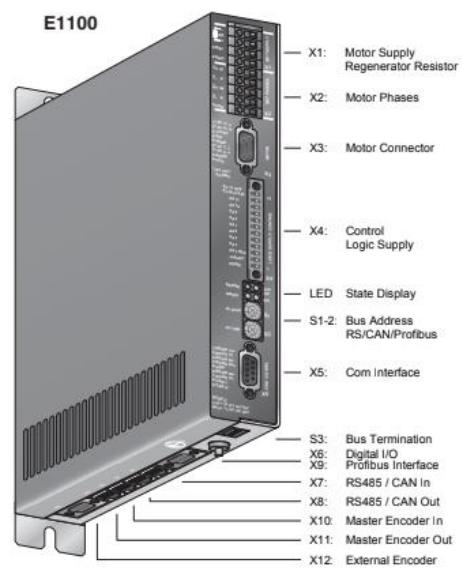


Figura 43: Driver E1100 LinMot

- Connettore X1:** Collegamento dell'alimentazione del motore in DC tra 24V e 80V 8A/15A/25A. Inizialmente utilizzavamo un vecchio trasformatore collegato ad un ponte a diodi e condensatore, ma la potenza era molto limitata e creava molti problemi di movimentazione. La configurazione attuale, che non è ottimale, è costituita da un alimentatore DC 12V 42A collegato ad uno step-up di tensione che ci permette di avere 55V stabili e 1.2A di corrente massima al motore. La resistenza di carico è opzionale e si abilita dalla configurazione software del driver.
- Connettore X2:** A questo connettore si collegano i cavi delle fasi del motore.
- Connettore X3:** Questo connettore contiene i segnali del motore tra cui posizione e temperatura.
- Connettore X4:** Questo connettore presenta la porta di alimentazione 24V della logica del driver e 9 I/O settabili da software.

GND	
24V	
X4.3	OUT isMoving: TRUE in movimento
X4.4	IN
X4.5	IN
X4.6	IN Trigger
X.4.7	IN Homing
X4.8	IN Switch On
X4.9	OUT Err: TRUE in stato di errore
X4.10	IN Reset
X4.11	OUT isHomed: TRUE se homing fatto
Enable	IN

- Connettore X5:** Questo connettore permette la comunicazione seriale RS232, RS485 o CAN tra computer e driver per la sua configurazione.

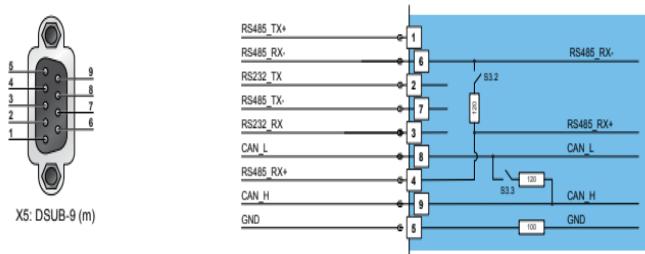


Figura 44: Connettore X5

7.2.2 Configurazione con LinMot Talk

La configurazione del driver avviene tramite il software dedicato LinMot Talk. Durante la realizzazione del progetto abbiamo utilizzato il protocollo di comunicazione seriale RS232 attraverso il connettore X5. Per connettere il cavo al PC bisogna utilizzare un adattatore RS232 – USB. Di seguito viene illustrato il processo di collegamento e settaggio:

- Collegare il cavo di comunicazione grigio al connettore X5 e al PC tramite l'adattatore.

Andare in “gestione dispositivi” di Windows ed individuare la COM utilizzata.

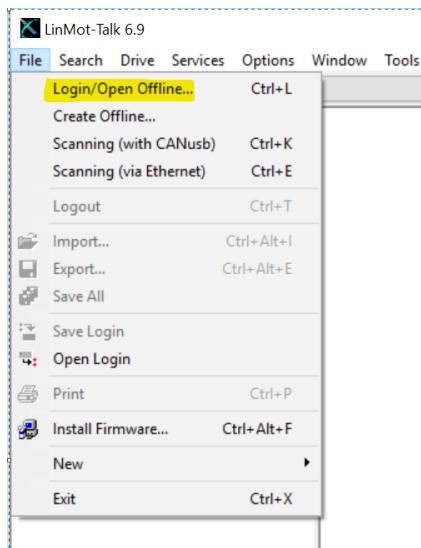


Figura 45: Login

- Nella tendina selezionare la COM corretta e spuntare RS232

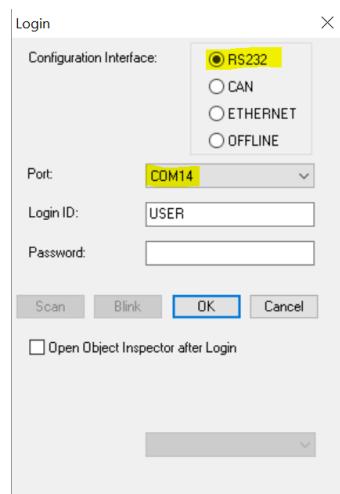


Figura 46: Connessione

- A questo punto avverrà la connessione tra driver e computer. Si aprirà il pannello di controllo del motore da cui si possono settare i parametri di funzionamento, settare gli I/O e scrivere programmi.

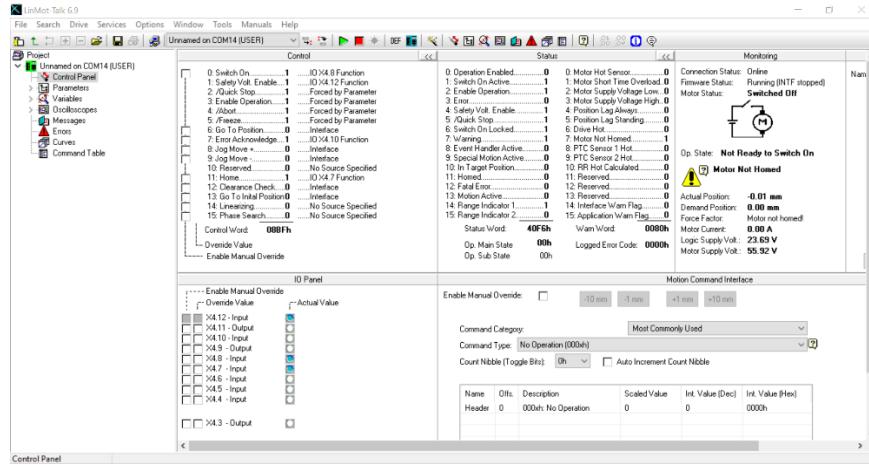


Figura 47: Pannello di controllo LinMot

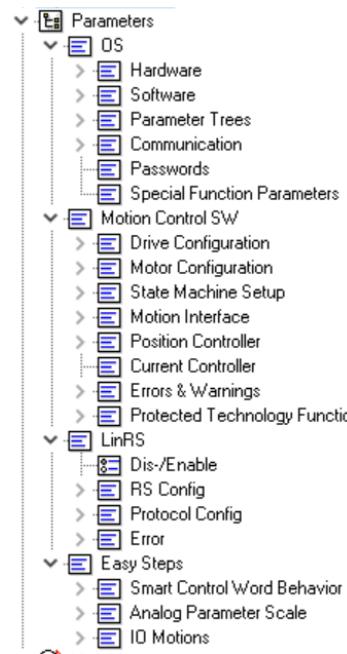


Figura 48: Parametri LinMot

- A causa delle limitazioni sull'alimentazione del motore lineare abbiamo ridotto i valori di corrente assorbiti da esso per evitare danni.

Name	Value	Raw Data	Value...	UPID	Type
Maximal Current	1.2A	04B0h	*** A	119Eh	SInt16
Maximal Motor Supply Voltage	60 V	1770h	*** V	11A7h	UInt16
Phase Resistance	10.1 Ohm	03F2h	*** O...	119Fh	UInt16
Phase Resistance Definition Temp	20 °C	00C8h	*** °C	120Bh	SInt16
Phase Inductance	1.4 mH	000Eh	*** mH	11A0h	UInt16
Force Constant	11.04 N/Apk	0450h	*** N...	11A1h	UInt16
Zero Position (ZP)	158 mm	00181BE0h	*** mm	11A2h	SInt32
Shortened Stroke (SS)	290 mm	002C4020h	*** mm	11A3h	SInt32
Maximal Stroke	350 mm	003567E0h	*** mm	11A4h	SInt32
Edge Force Constant	6.9 N/Apk	02B2h	*** N...	11A5h	UInt16
Extension Cable Resistance	0.29 Ohm	001Dh	*** O...	11A6h	UInt16

Figura 49: Corrente Massima

- La programmazione dei cicli avviene attraverso sequenze di comandi in tabella che vengono richiamate quando l'input X4.6 di trigger sul connettore X4 riceve un segnale.
- Il comando **VAI Go To Pos** indica al motore di raggiungere la posizione indicata con determinati parametri di velocità, accelerazione e decelerazione. Questi settaggi vengono impostati nel momento in cui viene inserita la funzione e i loro valori si possono vedere nelle immagini sottostanti. Il driver dopo aver letto questo comando si sposta subito al successivo.
- Il comando **Wait until Motion Finished** blocca la lettura dei comandi successivi fino a che il movimento precedentemente imposto non finisce. Senza questo comando il motore non si sarebbe mosso.
- Il comando **Wait Time** è un semplice delay a cui bisogna passare il tempo in millisecondi.

Nella porzione di destra delle immagini si legge il numero della riga a cui il driver deve spostarsi dopo aver eseguito un comando. Per esempio esaminando la prima immagine: il driver calcola il profilo di movimento alla riga 15, si sposta alla riga 16 dove aspetta la fine del movimento, legge la riga 17 e poi passa alla 7.

15	GO_UP	VAI Go To Pos (010h)	Pos: 0 mm	Vel: 0.5 m/s	Acc: 4 m/s^2	Dec: 3 m/s^2		16 (WAIT_POS)
16	WAIT_POS	Wait until Motion Finished (211h)						17 (WAIT_20)
17	WAIT_20	Wait Time (210h)	Time: 20 ms					7 (FALSE)

Figura 50: Sequenza richiamata con trigger X4.6 e input X4.4 alto

30	SCENDI	VAI Go To Pos (010xh)	Pos: 80 mm	Vel: 0.5 m/s	Acc: 3 m/s^2	Dec: 4 m/s^2	31 (CURRENT)
31	CURRENT	Wait until Motion Finished (211xh)					32 (WAIT_20)
32	WAIT_20	Wait Time (210xh)	Time: 20 ms				7 (FALSE)

Figura 51: Sequenza richiamata con trigger X4.6 e input X4.4 basso

Nella figura 51 si vede il programma del processo di discesa dello stelo, nella figura 50 si vede il processo di salita.

Un approccio alternativo può essere quello di imporre un movimento di discesa che porta lo stelo in battuta con il piano di lavoro. Usando il comando **Wait until Current Greater than** invece del comando **Wait until Motion Finished** si può bloccare il movimento quando la corrente assorbita dal motore raggiunge una certa soglia. Quando lo stelo andrà in battuta contro un oggetto o il piano di lavoro la corrente supererà la soglia e il movimento si bloccherà. Subito dopo tramite il comando **VAI Go To Pos** si fa risalire lo stelo. Questo approccio permette di afferrare oggetti che hanno misure in direzione verticale molto diverse tra loro. Originariamente applicavamo questo metodo, ma a causa della limitata potenza di alimentazione avevamo molti problemi nel momento in cui la corrente aumentava.

7.3 Attuatore di presa

Il meccanismo di presa è costituito da una ventosa che sfrutta l'effetto venturi per creare una depressione.



Figura 52: Ventosa

Il flusso dell'aria è gestito da un'elettrovalvola comandata a 24 V.

I pezzi che possono essere afferrati da questo sistema devono essere caratterizzati da una superficie di presa per lo più liscia per consentire una presa salda e il loro peso deve essere inferiore di 300 g.

7.4 Microcontrollori

7.4.1 Hercules RM57Lx

Il microcontrollore Hercules RM57Lx viene utilizzato per risolvere la cinematica inversa di ROSBOT, gestire i segnali in ingresso e comunicare via seriale col microprocessore Raspberry. Questa scheda è stata adottata per via delle sue elevate prestazioni e per la sua predisposizione per gestire tutte le funzioni di timing necessarie per leggere e generare impulsi.

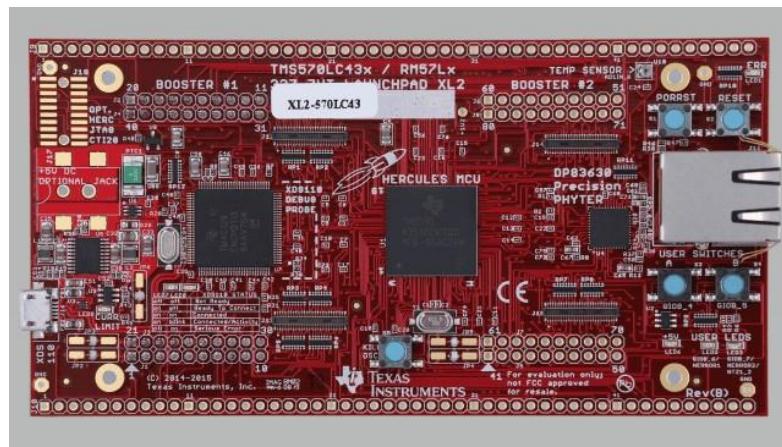


Figura 53: Hercules RM57Lx

La scheda è composta da due microcontrollori, il primo esegue il programma, mentre il secondo si occupa delle funzioni di timing.

7.4.1.1 Struttura

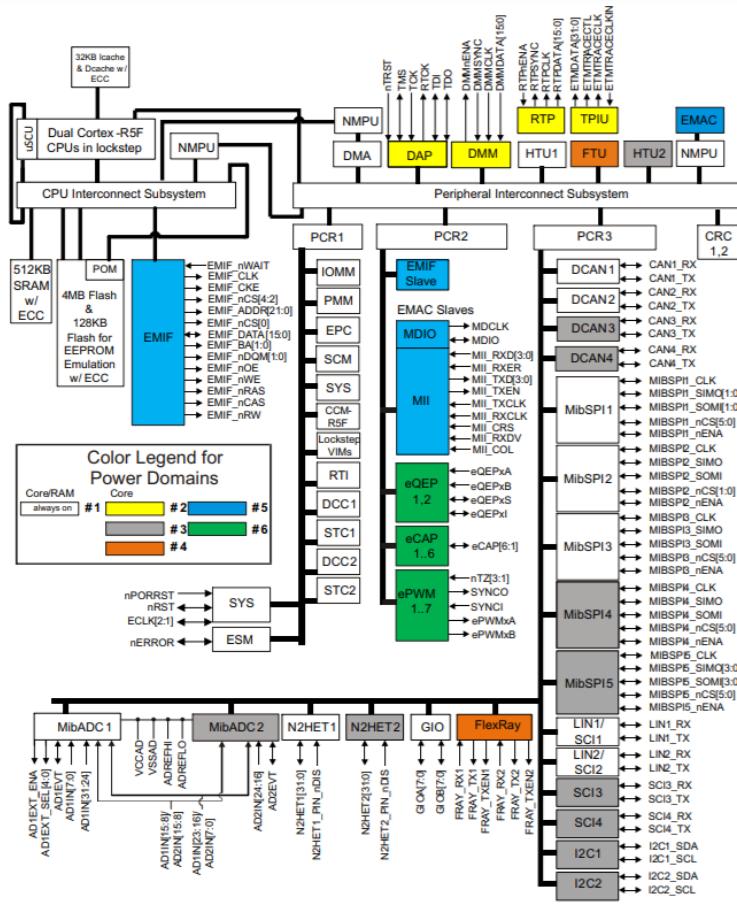


Figura 54: Struttura Hercules RM57Lx

Il microcontrollore è organizzato in blocchi che eseguono funzioni dedicate, quelle da noi utilizzate sono le seguenti:

N2HET (High-End Timer)

Questo blocco è composto da un timer ad alta precisione che viene utilizzato per generare i segnali PWM e permette di analizzare segnali in ingresso.

In questo progetto viene utilizzato principalmente per generare i segnali che comandano i motori ad impulsi.

GIO (General-Purpose Input / Output)

Questo blocco si occupa delle due porte di input/output (PORTA, PORTB). Esso gestisce i settaggi hardware dei pin quali: le resistenze di pull-up e pull-down, open drain e la lettura del fronte di salita o discesa per la funzione di interrupt.

SCI (Serial Common Interface)

Questo blocco gestisce le quattro porte di comunicazione seriale (SCI1, SCI2, SCI3, SCI4) di cui solo una è cablata come USB per permettere una comunicazione veloce ed efficace con il computer.

eQEP (Enhanced Quadrature Encoder Pulse)

Questo blocco legge i segnali provenienti dagli encoder in quadratura di fase. Questo metodo di trasmissione della posizione è tipico degli encoder incrementali e necessita di tre segnali: due per gli impulsi e uno per contare i giri. La direzione di rotazione dei motori è comunicata attraverso lo sfasamento dei segnali.

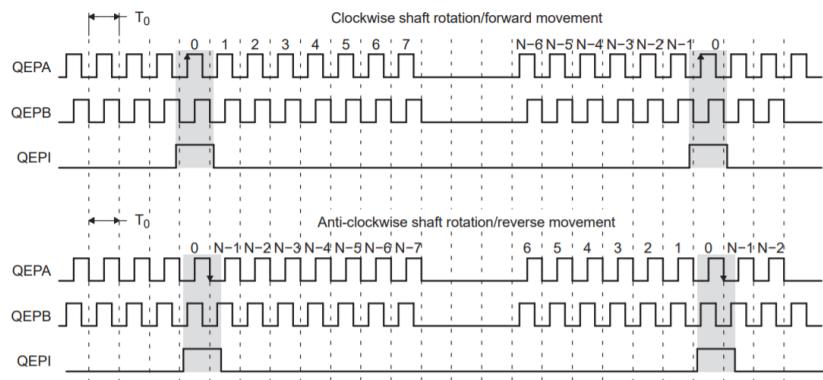


Figura 55: Impulsi in quadratura di fase

Come si può osservare dalla figura precedente per sapere la direzione di rotazione bisogna considerare l'istante di tempo in cui vi è il fronte di salita del primo segnale se il secondo segnale è ad un livello logico alto il motore starà girando in senso anti orario, altrimenti in senso orario.

7.4.1.2 HALCoGen

È il programma che attraverso un'interfaccia grafica genera il codice necessario per tutti i settaggi hardware e le funzioni che permettono di gestire i vari blocchi.

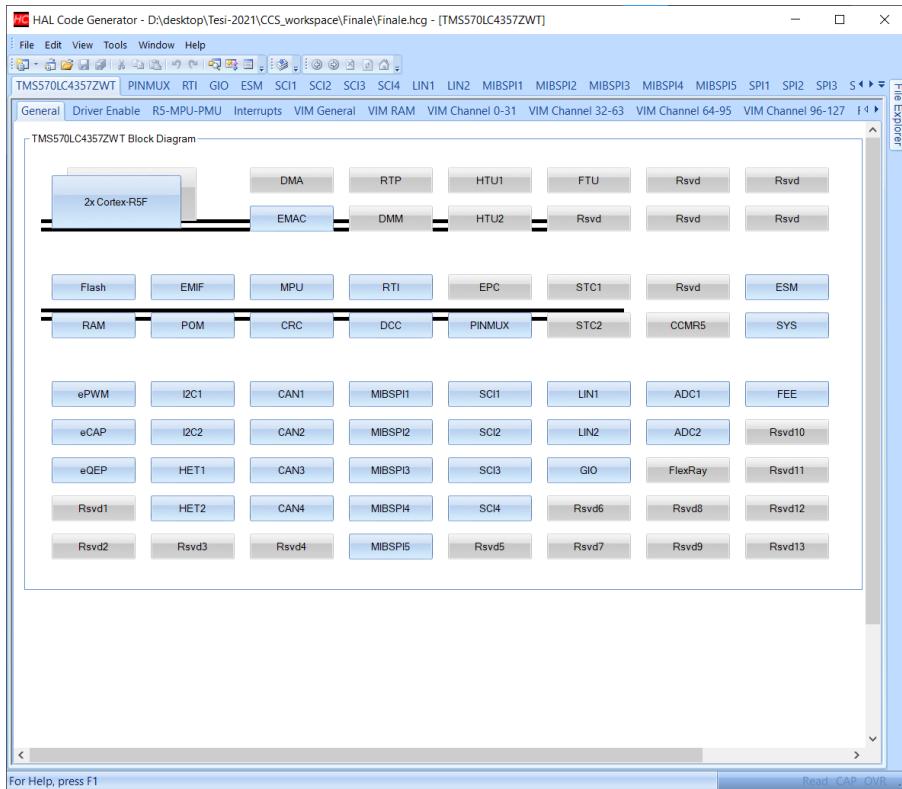


Figura 56: HALCoGen

Come si può osservare nella figura precedente questo programma permette la gestione dei blocchi prima elencati, oltre alla gestione generale della scheda. La prima cosa da fare per poter utilizzare la scheda è abilitare i driver dei blocchi desiderati nella sezione *Driver Enable*.

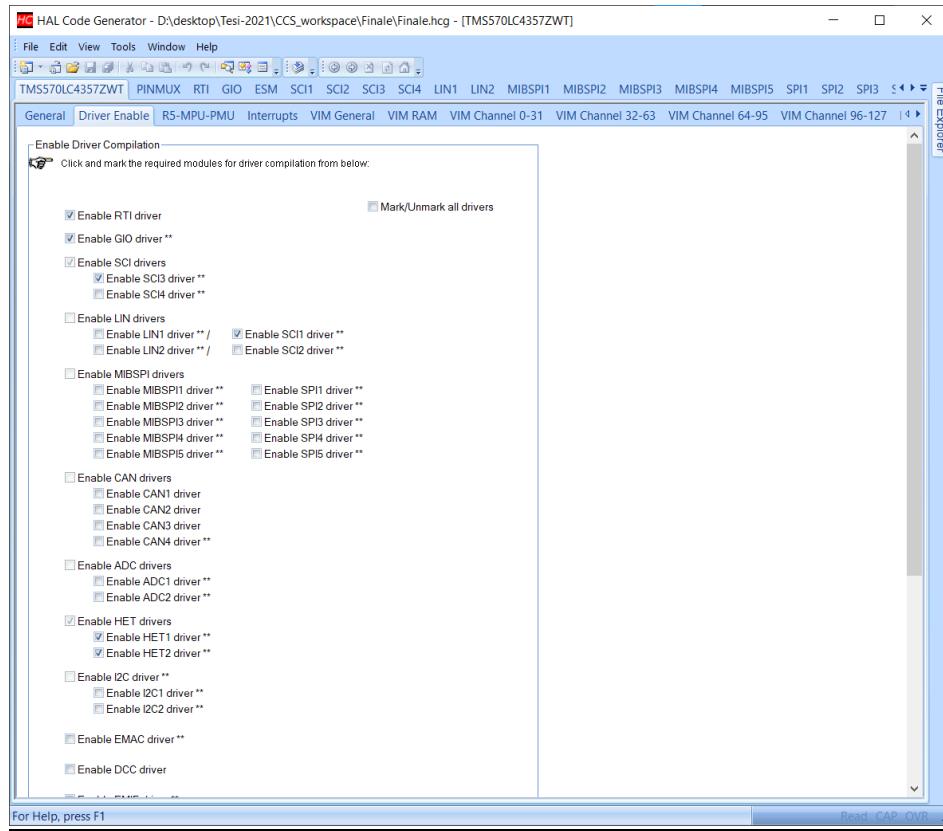


Figura 57: Settaggio Driver

Successivamente bisogna recarsi nelle varie sottosezioni per gestire i vari blocchi, qui di seguito illustreremo i settaggi adottati per la movimentazione di ROSBOT.

GIO

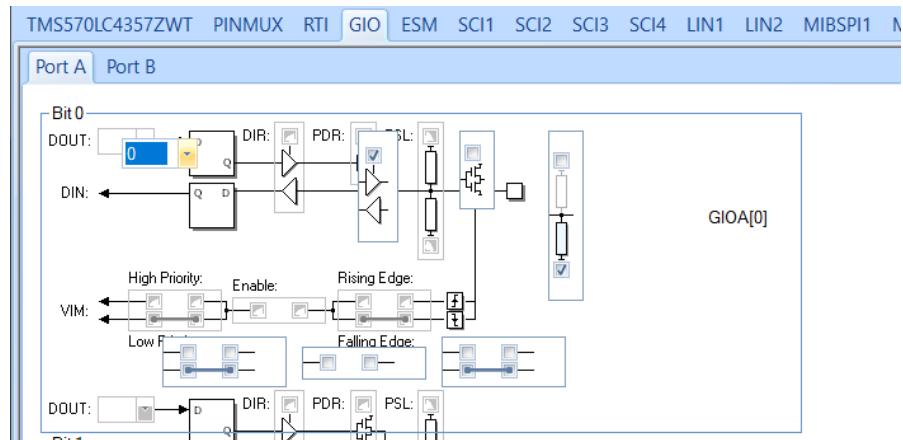


Figura 58: GIO

Come si può osservare dalla figura precedente si possono vedere le due sezioni dedicate alle porte I/O che contengono a loro volta i singoli pin. Ognuno di questi può avere una resistenza di pull-up o pull-down, open drain, input/output e interrupt.

SCI

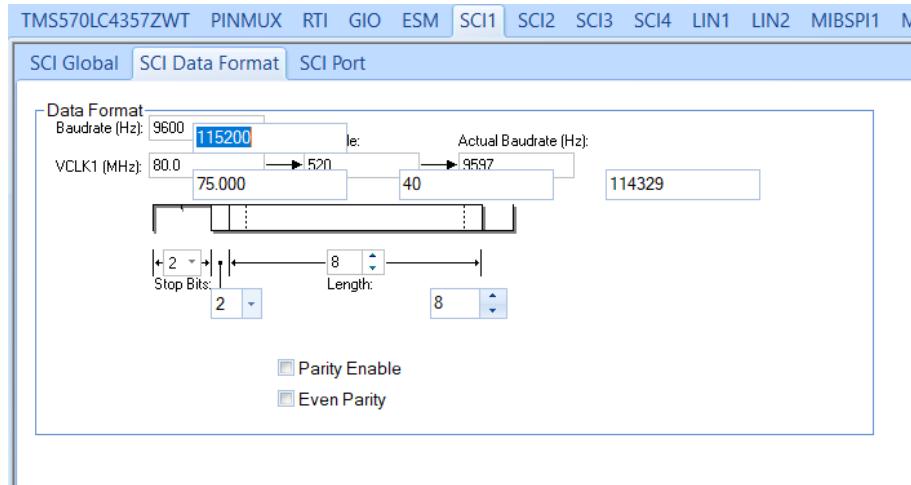


Figura 59: SCI Data Format

Come si può osservare dalla figura precedente nella sezione SCI sono presenti tre schede. Nella prima vengono gestiti gli aspetti generali, nella seconda viene impostato il formato di comunicazione e nella terza vengono settate le funzioni hardware dei pin.

Nel nostro caso come si può osservare dall'immagine abbiamo utilizzato una frequenza di comunicazione di 115200 Hz, due bit di stop e nessuna parità.

HET

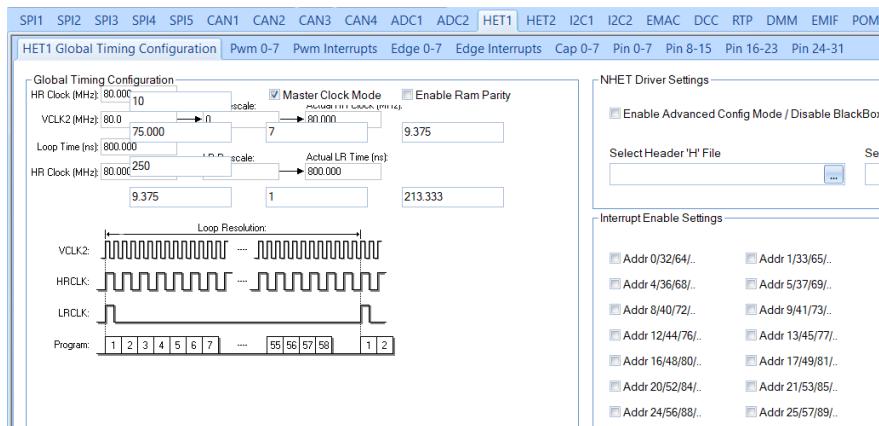


Figura 60: HET

Come si può osservare dalla figura precedente nella sezione HET vi sono varie sottosezioni. In quella denominata *HET1 Global Timing Configuration* si deve impostare la frequenza di confronto a cui lavorerà il modulo dedicato. Nel nostro caso il periodo del timer è di 250 ns, questo parametro definisce la risoluzione del modulo.

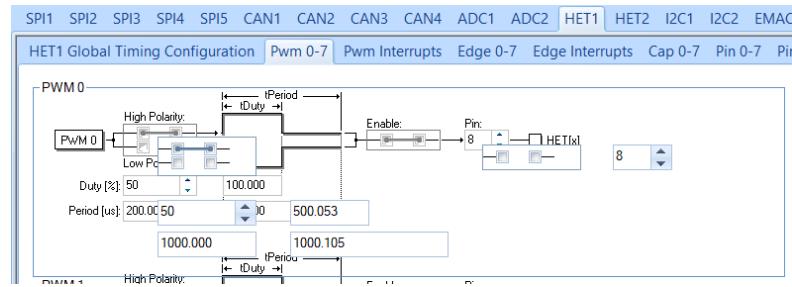


Figura 61: HET PWM

Nella sezione *Pwm 0-7* si assegnano i pin fisici a ogni modulo che genera il PWM e si impostano i valori di default del periodo e del duty cycle.

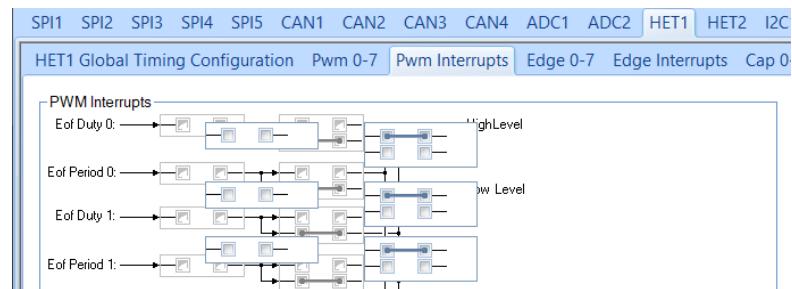


Figura 62: HET PWM Interrupt

Nella sezione *Pwm Interrupts* si impostano gli eventuali interrupt generati dal modulo indicando l'istante di generazione (*E of Duty* o *E of Period*) e la linea utilizzata (*High Level* o *Low Level*).

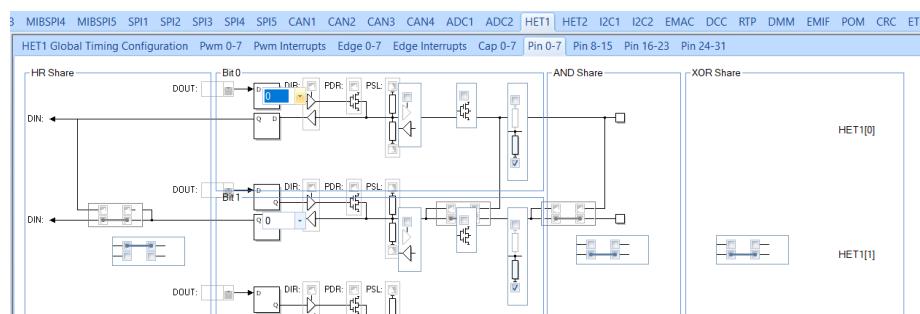


Figura 63: HET Pin

Nella sezione *Pin 0-7* vengono settate le configurazioni hardware dei vari pin in uscita, consentendo anche di andare a leggere uno stesso pin da due differenti moduli (Pin sharing).

eQEP

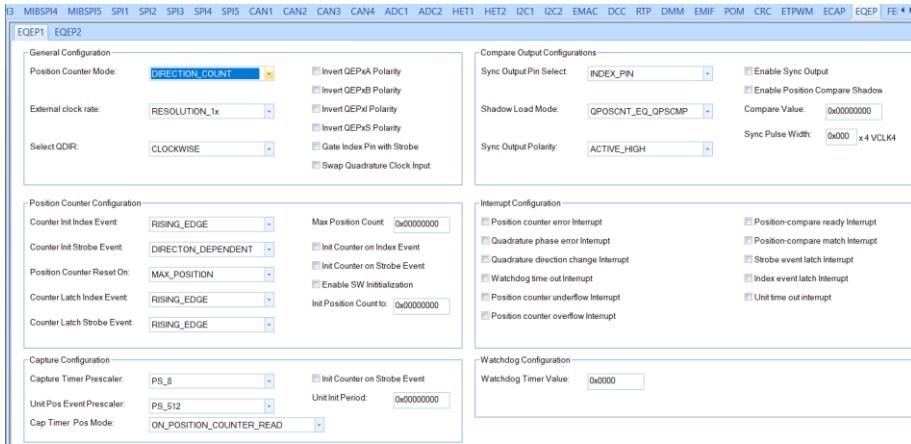


Figura 64: eQEP

Nella sezione riguardante la lettura dei segnali in quadratura di fase vi sono tutti i settaggi per riuscire a leggere la posizione in maniera corretta.

Creazione di un nuovo progetto:

1. Premere in File -> New il tasto Project

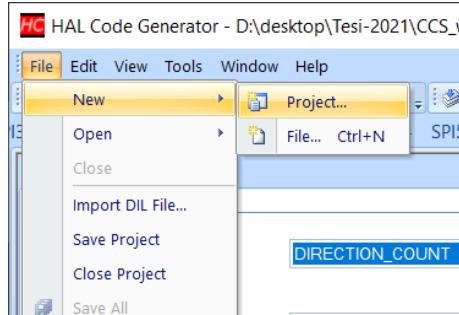


Figura 65: File -> New -> Project

2. Selezionare la scheda utilizzata e scegliere il nome del progetto.

⚠️ Il nome del progetto scelto dovrà essere lo stesso che si sceglie per il progetto in Code Composer Studio.

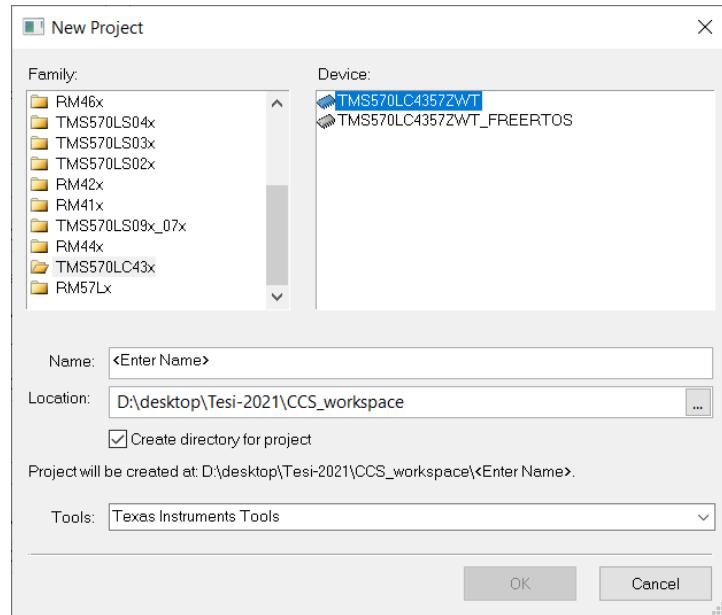


Figura 66: New Project

3. Settare tutti i parametri desiderati come sopra indicato
4. Premere il tasto F5 oppure File -> Generate Code

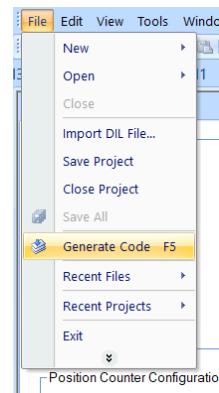


Figura 67: File -> Generate Code

5. Il codice viene generato nella cartella del progetto.

7.4.1.3 Code Composer Studio

È l'ambiente di sviluppo creato dalla Texas Instruments per programmare i microcontrollori. Esso dispone di una funzionalità di debugger che consente di ispezionare il funzionamento del programma grazie all'uso dei *break-point* e alla visualizzazione del contenuto delle variabili.

Il programma è stato suddiviso in diversi file che raggruppano funzioni e variabili riguardanti lo stesso argomento.

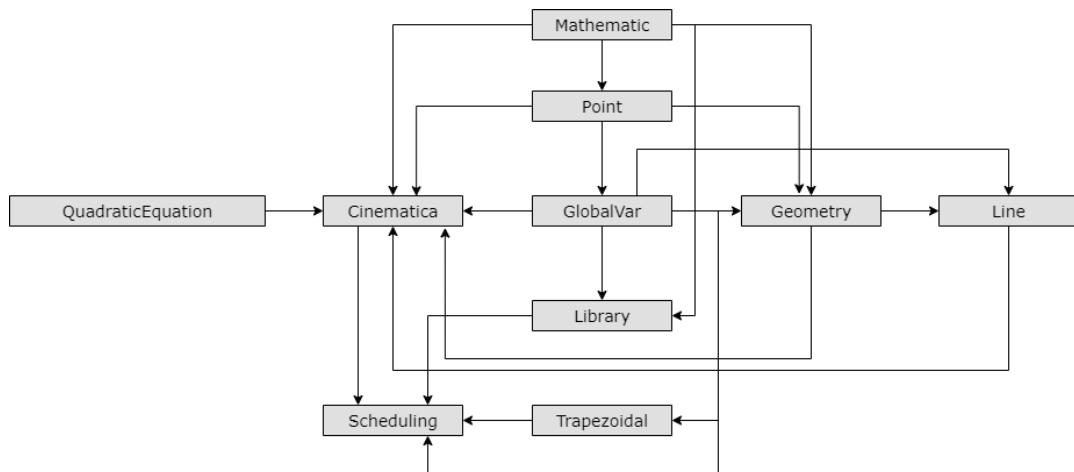


Figura 68: Struttura delle dipendenze

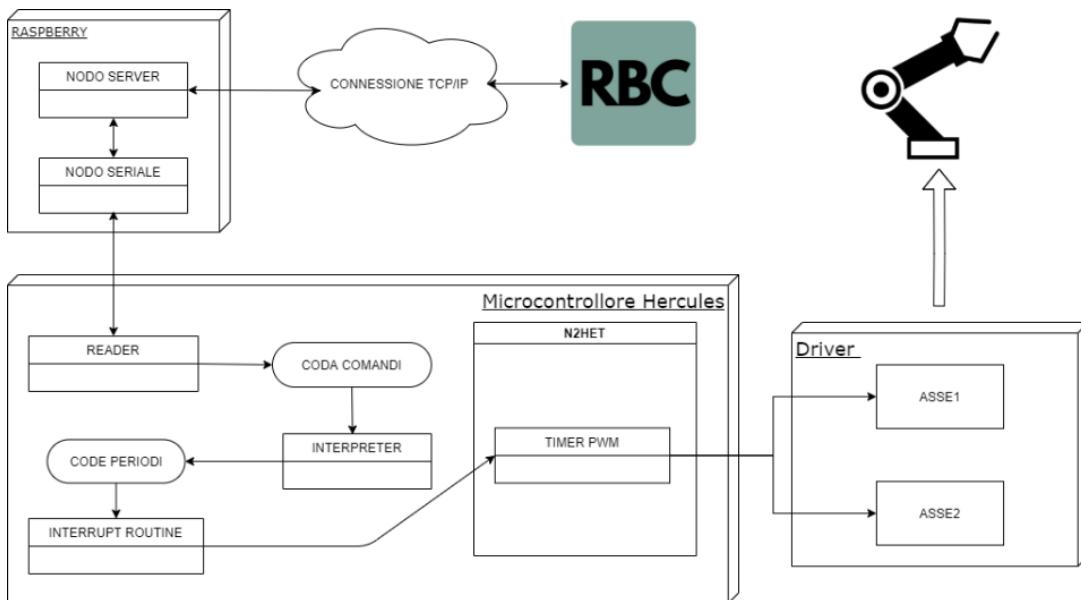


Figura 69: Struttura logica di ROSBOT

La logica del programma si basa sulla figura precedente dove vi è una funzione *reader* che legge i messaggi inviati tramite seriale alla scheda e li codifica in comandi, successivamente la funzione *interpreter* interpreta i comandi e popola le code dei periodi che infine saranno lette dalle routine di interrupt per generare gli impulsi opportuni.

GlobalVar

In *GlobalVar* vi è tutto ciò che è comune a tutto il progetto, quindi per lo più strutture e variabili esterne.

Questo file necessita delle seguenti inclusioni:

- Point.h

```
typedef struct
{
    int mainAngle;
    int secondAngle;
    float dAngle;
} MotorAngle;
```

La struttura *MotorAngle* viene utilizzata per contenere tutte le informazioni riguardanti gli angoli dei motori.

Membri:

- *mainAngle*: angolo del motore in considerazione.
- *secondAngle*: angolo fittizio del motore non in considerazione riferito al principale.
Questo membro viene utilizzato nella movimentazione lineare.
- *dAngle*: minimo angolo risolvibile dal motore.

```
typedef struct
{
    unsigned int period;
    bool isPeriod;
    bool isEnd;
    bool isCW;
} Period;
```

La struttura *Period* serve per passare i dati relativi ai periodi all'interno del programma.

Membri:

- *period*: contiene il periodo in μs .
- *isPeriod*: è un membro che indica se il periodo generato è utilizzabile oppure no.
- *isEnd*: è un membro booleano che se è attivo indica l'ultimo periodo e quindi la fine della movimentazione.

- *isCW*: è una variabile booleana che indica il verso di rotazione.

```
bool isPointReachable(Point _point)
```

La funzione *isPointReachable* restituisce un valore booleano che indica se il punto *_point* è raggiungibile dal robot.

```
void emergencyMsg(void)
```

La funzione *emergencyMsg* quando chiamata invia un messaggio di errore al microprocessore.

```
void finishedMsg(void)
```

La funzione *finishMsg* invia il messaggio di fine movimentazione al microprocessore.

```
void emergenza(void)
```

La funzione *emergenza* pone il robot in uno stato di emergenza bloccando tutti i motori.

```
void reset(void)
```

La funzione *reset* resetta lo stato del robot riattivando tutti i motori.

Mathematic

In *Mathematic* vi sono tutti gli strumenti supplementari alla classe standard del linguaggio C *math.h*.

```
float micronRound(float _number, int _rounder)
```

La funzione *micronRound* restituisce il numero arrotondato di *_rounder* volte del numero - *_number*.

```
float absolute(float _number)
```

La funzione *absolute* restituisce il valore assoluto di un numero con la virgola.

Point

In *Point* vi sono tutte le funzioni e le strutture necessarie a gestire i punti.

```
typedef struct
{
    float x;
    float y;
    float z;
} Point;
```

La struttura *Point* contiene le tre coordinate spaziali di un punto.

Membri:

- *x*: coordinata x del punto
- *y*: coordinata y del punto
- *z*: coordinata z del punto

```
typedef struct
{
    float x;
    float y;
}Point2D;
```

La struttura *Point2D* contiene le due coordinate di un punto piano.

Membri:

- *x*: coordinata x del punto
- *y*: coordinata y del punto

```
float pointDistance(Point2D _point1, Point2D _point2)
```

La funzione *pointDistance* restituisce la distanza fra *_point1* e *_point2*.

```
bool pointCopy(Point _point1, Point* _point2)
```

La funzione *pointCopy* copia il punto in tre dimensioni *_point1* nella locazione puntata da *_point2*.

```
bool point2DCopy(Point2D _point1, Point2D* _point2)
```

La funzione *point2DCopy* copia il punto in due dimensioni *_point1* nella locazione puntata da *_point2*.

```
bool isPointEq(Point _point1, Point _point2)
```

La funzione *isPointEq* restituisce un valore logico vero se i due punti tridimensionali sono uguali.

```
bool isPoint2DEq(Point2D _point1, Point2D _point2)
```

La funzione *isPoint2DEq* restituisce un valore logico vero se i due punti bidimensionali sono uguali.

```
Point2D point2D(Point _point)
```

La funzione *point2D* restituisce un punto bidimensionale che contiene le coordinate x e y di un punto tridimensionale.

Geometry

In *Geometry* vi è tutto ciò che riguarda i calcoli geometrici.

Questo file necessita delle seguenti inclusioni:

- GlobalVar.h
- Point.h
- Mathematic.h

```
typedef struct
{
    float module;
    float phase;
} Vector2D;
```

La struttura *Vector2D* contiene tutte le informazioni per utilizzare i vettori di due dimensioni all'interno del programma.

Membri:

- *module*: è il modulo del vettore
- *phase*: è la fase del vettore

```
float calcModule(Point2D _point)
```

La funzione *calcModule* dato un punto bidimensionale *_point* calcola il modulo del vettore che congiunge quel punto all'origine degli assi.

```
float calcPhase(Point2D _point)
```

La funzione *calcPhase* dato un punto bidimensionale *_point* calcola la fase del vettore che congiunge quel punto all'origine degli assi rispetto all'asse x di riferimento.

QuadraticEquation

In *QuadraticEquation* vi è tutto ciò che serve per risolvere delle equazioni di secondo grado organizzate nel seguente modo:

$$a \cdot x^2 + b \cdot x + c = 0$$

```
float delta(float _a, float _b, float _c)
```

La funzione *delta* restituisce il calcolo del delta della funzione di secondo grado.

```
float solveQE(float _a, float _b, float _c, bool _isPositive)
```

La funzione *solveQE* restituisce una delle due soluzioni dell'equazione in funzione della variabile booleana *_isPositive*.

Cinematica

In *Cinematica* vi è tutto ciò che serve per risolvere la cinematica inversa e diretta.

Questo file necessita delle seguenti inclusioni:

- GlobalVar.h
- Geometry.h
- Mathematic.h
- Line.h
- QuadraticEquation.h
- Point.h

```
void calcAngles(const Point2D _point, int* _alpha, int* _beta)
```

La funzione *calcAngles* calcola gli angoli dei motori fornito un punto (*_point*) del piano. I valori degli angoli vengono posizionati nelle locazioni di memoria puntate da *_alpha* e *_beta* in passi (per esempio se l'angolo calcolato dell'asse 1 è di π il corrispondente in passi è di 25000);

```
void calcMotorAngles(const Point2D _point, bool _isAlpha)
```

La funzione *calcMotorAngles* scrive nelle variabili esterne *alpha*, *beta* e *betaAbs* i valori degli angoli calcolati rispetto al punto *_point* considerando se i calcoli devono essere fatti per l'asse 1 o 2.

```
float calcJointSpeed(Vector2D _velocity, Point2D _point, bool _isAlpha)
```

La funzione *calcJointSpeed* restituisce il valore della velocità angolare in Hertz del motore selezionato attraverso la variabile booleana *_isAlpha*. I parametri di ingresso di cui ha bisogno sono il vettore *_velocity* e il punto *_point* in cui si trova ROSBOT.

```
bool calcJoint1(float _alpha, Point2D* _point)
```

La funzione *calcJoint1* calcola la posizione della fine del primo asse e la mette nella locazione di memoria puntata da *_point*.

```
bool calcAlpha(float _beta, Line _targetLine, Point2D _previousPoint, Point2D _lastPoint, bool  
_isAlpha)
```

La funzione *calcAlpha* calcola il valore dell'angolo del primo motore in funzione del secondo e il risultato lo inserisce nella variabile del motore selezionato dalla variabile booleana *_isAlpha*.

```
bool calcBeta(float _alpha, Line _targetLine, Point2D _previousPoint, Point2D _lastPoint, bool  
_isAlpha)
```

La funzione *calcBeta* calcola il valore dell'angolo del secondo motore in funzione del primo e il risultato lo inserisce nella variabile del motore selezionato dalla variabile booleana *_isAlpha*.

```
bool calcPoint(float _alpha, float _beta, Point2D* _point)
```

La funzione *calcPoint* calcola il punto in cui si trova il robot partendo dagli angoli dei motori *_alpha* e *_beta* ponendo il risultato nella locazione puntata da *_point*.

Library

In *Library* vi è tutto ciò che serve per la gestione dei messaggi e dei comandi.

Questo file necessita delle seguenti inclusioni:

- GlobalVar.h
- Mathematic.h

```
typedef struct
{
    char type;
    Point point;
    float parameter;
}Command;
```

La struttura *Command* contiene tutti i parametri necessari a distinguere tutti i comandi descritti nella sezione Comunicazione.

Membri:

- *type*: rappresenta il codice operativo del comando, per come è stato dichiarato può ammettere fino a un massimo di 246 comandi diversi, poiché i primi 10 comandi sono di servizio.
- *point*: rappresenta i parametri che di solito sono punti o distanze.
- *parameter*: rappresenta un parametro aggiuntivo del messaggio che di solito è la velocità.

```
typedef struct
{
    size_t size;
    int index;
    int* elements;
}Queue;
```

La struttura *Queue* contiene tutti i parametri per generare una pila FIFO (First Input First Output) che abbia un limite di elementi ma che il suo numero non sia statico nel tempo ma dinamico. Il suo funzionamento si basa sull'allocazione dinamico della memoria per un array di puntatori interi.

Questa struttura per essere utilizzata ha bisogno delle seguenti funzioni:

```
bool queueInitializer(Queue* _queue, unsigned int _size, unsigned int _sizeOf);
bool pushQueue(Queue* _queue, int* _ptr);
int* popQueue(Queue* _queue);
```

Queste funzioni verranno spiegate in seguito.

Membri:

- *size*: indica il numero massimo di elementi contenibili nella coda
- *index*: indica il numero di elementi contenuti dalla coda
- *elements*: è il puntatore alla locazione di memoria dove si trova l'array di puntatori interi degli elementi

```
bool queueInitializer(Queue* _queue, unsigned int _size, unsigned int _sizeOf)
```

La funzione *queueInitializer* serve ad inizializzare le code con una lunghezza massima impostata da *_size* e una grandezza di allocazione di *_sizeOf*. Essa ritorna un valore logico vero se il procedimento va a buon fine altrimenti falso.

```
bool pushQueue(Queue* _queue, int* _ptr)
```

La funzione *pushQueue* inserisce il puntatore a una variabile *_ptr* all'interno di una coda *_queue*. Essa ritorna un valore logico vero se il processo è andato a buon fine altrimenti falso.

```
int* popQueue(Queue* _queue)
```

La funzione *popQueue* restituisce il primo elemento della coda *_queue*.

```
bool reader(Queue* _commands)
```

La funzione *reader* si occupa di tutta la comunicazione con il microprocessore e popola la coda *_commands* con i comandi eseguibili da ROSBOT.

```
Command* parser(char* _message)
```

La funzione *parser* svolge il compito di decodificare i messaggi inviati dal microprocessore *_message* per renderli comandi eseguibili.

```
double stringToNumber(char* _string, int* _index)
```

La funzione *stringToNumber* dato un puntatore ad un array di char *_string* e un indice *_index*, che serve per tener conto di quanti caratteri sono stati convertiti, restituisce un valore numerico a doppia precisione.

```

bool readMovment(char* _string, Command* _command, int* _index)
bool readCondition(char* _string, Command* _command, int* _index)
bool readSetting(char* _string, Command* _command, int* _index)

```

Queste funzioni, dati in ingresso un puntatore ad un array di char *_string* e un indice *_index*, il quale indica quanti caratteri sono stati processati, riempie il comando puntato da *_command*. Esse restituiscono un valore logico vero se l'operazione va a buon fine altrimenti falso.

Line

In *Line* vi è tutto ciò che è necessario per gestire delle rette considerate nella seguente forma:

$$y = m \cdot x + q$$

Questo file necessita delle seguenti inclusioni:

- Geometry.h
- GlobalVar.h

```

typedef struct
{
    float slope;
    float offset;
} Line;

```

La struttura *Line* contiene tutte le informazioni necessarie a descrivere una retta.

Membri:

- *slope*: indica la pendenza *m* della retta
- *offset*: indica la traslazione della retta *q*

```

bool setLine2Point(Point2D _point1, Point2D _point2, Line* _line)

```

La funzione *setLine2Point* dati in ingresso due punti *_point1* e *_point2* calcola la retta passante per i due punti ponendola nella locazione di memoria puntata da *_line*. Essa restituisce un valore logico vero se l'operazione va a buon fine altrimenti falso.

```

bool setLineSIPo(Point2D _point, float _slope, Line* _line)

```

La funzione *setLineSIPo* dati in ingresso un punto *_point* e la pendenza *_slope* restituisce una retta nella locazione di memoria puntata da *_line*. Essa restituisce un valore logico vero se l'operazione va a buon fine altrimenti falso.

```
bool setOrthogonalLine(Point2D _point, Line _line, Line* _orthogonalLine)
```

La funzione *setOrthogonalLine* dati in ingresso un punto *_point* e una retta *_line* restituisce la retta ortogonale passante per il punto nella locazione di memoria puntata da *_orthogonalLine*. Essa restituisce un valore logico vero se l'operazione va a buon fine altrimenti falso.

```
bool isOnLine(Point2D _point, Line _line)
```

La funzione *isOnLine* data una retta *_line* controlla che il punto *_point* si trovi sulla retta, in caso affermativo restituisce un valore logico vero altrimenti falso.

```
bool isLineEq(Line _line1, Line _line2)
```

La funzione *isLineEq* controlla che le rette *_line1* e *_line2* siano uguali, in caso affermativo restituisce un valore logico vero altrimenti falso.

```
bool pointIntersLines(Line _line1, Line _line2, Point2D* _point)
```

La funzione *pointIntersLine* date in ingresso due rette *_line1* e *_line2* restituisce nella locazione di memoria puntata dal puntatore *_point* il punto di intersezione delle due rette. Essa restituisce un valore logico vero se l'operazione va a buon fine altrimenti falso.

```
float lineY(float _x, Line _line)
```

La funzione *lineY* dati una retta *_line* e una coordinata *_x* restituisce il valore della coordinata y corrispondente.

Trapezoidal

In *Trapezoidal* vi è tutto ciò che è necessario per gestire una legge di moto a tre tratti.

Questo file necessita delle seguenti inclusioni:

- GlobalVar.h

```
typedef struct
{
    float tAcc;
    float maxSpeed;
    float time;
}Trapezoidal;
```

La struttura *Trapezoidal* contiene tutti i dati necessari a caratterizzare una legge di moto a tre tratti.

Membri:

- *tAcc*: tempo di accelerazione in secondi
- *maxSpeed*: massima velocità che viene raggiunta
- *time*: tempo totale di movimentazione

```
bool setTrapezoidal(float _maxSpeed, float _time, float _tAcc, Trapezoidal* _law)
```

La funzione *setTrapezoidal* dati la massima velocità *_maxSpeed*, il tempo totale di movimentazione *_time* e il tempo di accelerazione *_tAcc* restituisce una legge compilata nella locazione di memoria puntata da *_law*. Essa restituisce un valore logico vero se l'operazione va a buon fine altrimenti falso.

```
float getTrapezoidalSpeed(float _time, Trapezoidal _law)
```

La funzione *getTrapezoidalSpeed* dati un tempo *_time* e una legge tre tratti *_law* restituisce la velocità corrispondente all'istante di tempo.

Scheduling

In *Scheduling* vi è tutto ciò che è necessario per la pianificazione del moto.

Questo file necessita delle seguenti inclusioni:

- Library.h
- GlobalVar.h
- Cinematica.h
- Trapezoidal.h

```
bool interpreter(Queue* _cmd, Queue* _queue1, Queue* _queue2, Queue* _queue3)
```

La funzione *interpreter* dato in ingresso un comando lo interpreta e genera i periodi degli impulsi che servono a movimentare ROSBOT. Essa li mette nelle code *_queue1*, *_queue2* e *_queue3* che in seguito verranno lette dalle routine di interrupt per generare gli impulsi. Essa restituisce un valore logico vero se l'operazione va a buon fine altrimenti falso.

```
void homing(void)
```

La funzione *homing* esegue la routine che posiziona il robot in una posizione di riferimento nota.

```
void asse1Step(int _step, int _velMax, Queue* _periods)
```

La funzione *asse1Step*, dati i passi da compiere *_step* e la velocità massima *_velMax*, popola la coda *_periods* con i periodi degli impulsi del motore dell'asse 1.

```
void asse2Step(int _step, int _velMax, Queue* _periods)
```

La funzione *asse2Step* dati i passi da compiere *_step* e la velocità massima *_velMax* popola la coda *_periods* con i periodi degli impulsi del motore dell'asse 2.

```
void unrelated(Point2D _point, float _vel, Queue* _periods1, Queue* _periods2)
```

La funzione *unrelated* dati il punto di arrivo *_point* e la velocità massima *_vel* popola le code *_periods1* e *_periods2* con i periodi degli impulsi dei motori dell'asse 1 e 2 per generare un movimento simultaneo dei due motori con un tempo di movimentazione uguale.

7.4.1.4 Comunicazione

I comandi di movimento vengono inviati alla Hercules tramite comunicazione seriale da Raspberry, la loro sintassi è elencata nelle tabelle sottostanti. Quando il micro riceve il messaggio verifica la sua sintassi e in caso positivo risponde “Ok!”, in caso negativo risponde “Er!”. Quando il comando viene effettivamente eseguito il micro risponde “Fns!”. Viene anche effettuata una verifica sulla effettiva raggiungibilità dei punti comandati, infatti nel caso in cui questi fossero fuori dall’area di lavoro la scheda risponderebbe “Un!”. Quando il robot si trova nello stato di emergenza risponde “|Emg!”.

Tabella 4: Messaggi di movimento

Movimenti				
Tipo messaggio	Codice operativo	Parametri	Descrizione	
M	4	X<posizione> Y<posizione> V<velocità>	spostamento scorrelato	
M	5	P<angolo> S<angolo> V<velocità>	spostamento dei giunti	
M	6	Z<numero programma>	fa eseguire al motore lineare il comando programmato	
M	7	S<statoGripper>	comando che imposta lo stato del gripper	
M	8		homing	

Tabella 5: Messaggi condizionali

Condizioni				
Tipo messaggio	Codice operativo	Parametri	Descrizione	
C	1	T<millisecondi>	pausa di tempo in millisecondi	

Tabella 6: Messaggi di settaggio

Settaggi				
Tipo messaggio	Codice operativo	Parametri	Descrizione	
S	2		Reset	
S	3		Emergenza	
S	4	A<accelerazione>	accelerazione	

7.4.1.5 Cablaggio

I nomi dei connettori sulla scheda di condizionamento e sulla Hercules corrispondono. I connettori J1/J3, J2/J4, J5/J7 e J6/J8 sono cablati tramite cavo flat a 20 poli. I connettori J9 e J10 invece sono composti da 5 connettori da 10 poli etichettati.

J1/J3		
pin	descrizione	connessione
1		
2		
3	SCI3RX	comunicazione arduino mega
4	SCI3TX	
5	GIOA_7	sensore homing asse 1
6		
7	EQEP1A	ENCODER PHA M1
8	GIOA_6	sensore homing asse 2
9	N2HET1_4	allarme MZ
10	N2HET1_9	allarme M1
11		
12	GND	
13		
...		
20		

J4/J2		
pin	descrizione	connessione
1	N2HET1_2	allarme M2
2	N2HET1_18	catena di emergenza
3	N2HET1_16	IN EDGE1
4		
5	N2HET1_14	IN EDGE2
6	N2HET1_12	IN EDGE3
7	GIOA_5	servo ready M1
8	EQEP2I	ENCODER PHZ M2
9	GIOA_1	servo ready M2
10		
11	GND	
12		
13		
...		
20	GIOB_2	temperatura alarm

J5/J7		
pin	descrizione	connessione
1		
2		
3		
4		
5	EQEP1I	ENCODER PHZ M1
6		
7		
8	EQEP1B	ENCODER PHB M1
9	N2HET1_24	DIREZIONE MZ
10		
11		
12	GND	
13		
...		
20		

J8/J6		
pin	descrizione	connessione
1	N2HET1_10	TRIGGER MZ
2	N2HET1_28	IMPULSI MZ
3	N2HET1_8	CW M1
4	N2HET1_23	CCW M1
5		
6	N2HET1_11	CW M2
7		
8		
9		
10		
11	GND	
12	EQEP2A	ENCODER PHA M2
13		
...		
20		

J9		
pin	descrizione	connessione
1		
2	GND	
3		
4	GND	

5		
6		
7		
8	N2HET1_27	CCW M2
9		
10	N2HET2_19	GPO1 H
11	N2HET2_5	GPO2 H
12		
13		
14	N2HET2_4	GPO3 H
15	GIOA_4	tensione alarm
16	GND	
17	N2HET2_3	GPO4 H
18	N2HET2_2	GPO5 H
19		
20	N2HET2_7	GPO7 H
21		
22	N2HET2_14	IN EDGE4
23	N2HET2_23	IN EDGE5
24	N2HET2_11	IN EDGE6
25	GND	
26	N2HET2_10	IN EDGE7
27	N2HET2_9	IN EDGE8
28	N2HET2_22	IN EDGE9
29		
30		
31		
32		
33		
34	GND	
35		
...		
50		

J10		
pin	descrizione	connessione
1		
2	GND	
3		
4	GND	
5		
6	GIOB_3	catena di emergenza
7	GIOA_0	RUN M1
8	I2C2_SCL	I2C COMMUNICATION

9	I2C2_SDA	
10		
11		
12		
13		
14	GND	
15		
16		
17		
18	GIOA_3	RUN M2
19	N2HET2_8	IN EDGE10
20		
21		
22	EQEP2B	ENCODER PHB M2
23	GND	
24		
25		
26		
27		
28		
29		
30		
31		
32	GND	
33		
34	N2HET2_17	GPO6 H
35		
36		
37	GIOB_1	ENABLE MZ
38		
39	GIOB_0	GPIO1
40		
...		
50		

7.4.2 Arduino Mega

Il microcontrollore Arduino Mega viene utilizzato per espandere gli ingressi e le uscite della scheda Texas Instruments. Gli I/O vengono comandati tramite seriale e vengono gestite tutte le funzioni che non necessitano di una velocità elevata.

7.4.2.1 Programma

Il programma implementa lo stesso parser della scheda Texas Instruments e permette di leggere e settare gruppi di pin. Le possibili sintassi per l'invio dei comandi sono le seguenti:

- **six,x,x!** : Messaggio di lettura degli input specificati.
- **s1,0,1!** : Risposta con lo stato dei pin richiesti.
- **sox,x,x|vuv!** : Messaggio di impostazione output specificando lo stato di ogni pin.
- **sOx,x,x|v!** : Messaggio di impostazione output specificando lo stato del gruppo di pin.

7.4.2.2 Cablaggio

Di seguito è elencato il cablaggio dei pin.

J_ard			
pin	descrizione	connessione	Arduino Mega
1	GND		
2	GND		
3	+5V		
4	SERIALE RX	SCI RX	0
5	SERIALE TX	SCI TX	1
6	INIBIZIONE	CCW inibita M1	2
7	INIBIZIONE	CW inibita M1	3
8	INIBIZIONE	CCW inibita M2	4
9	INIBIZIONE	CW inibita M2	5
10		freno M2	6
11		homing MZ	7
12		LAMPADA 1	14
13		LAMPADA 2	15
14		LAMPADA 3	16
15		LAMPADA 4	17
16		LAMPADA 5	18
17		EL. VALVOLA 24V	19
18		GPO2 24V	20
19		GPO3 24V	21
20		GPO4 24V/5V	8
21		GPO5 24V/5V	9
22		GPO6 24V/5V	10
23		GPO7 24V/5V	11

24	GPO8 24V/5V	12
25	GPO9 24V/5V	13
26	GPO10 24V/5V	52
27	GPI1 0V - 30V	50
28	GPI2 0V - 30V	48
29	GPI3 0V - 30V	46
30	GPI4 0V - 30V	44
31	GPI5 0V - 30V	42
32	GPI1 0V - 30V	40
33	GPI7 0V - 30V	38
34	GPI8 0V - 30V	36
35	GPI9 0V - 30V	34
36	GPI10 0V - 30V	32
37		
...		
50		

7.4.3 Arduino DUE

Abbiamo dovuto utilizzare il microcontrollore Arduino DUE in seguito a delle perdite di passi del precedente microcontrollore dovute ad un errata gestione degli interrupt.

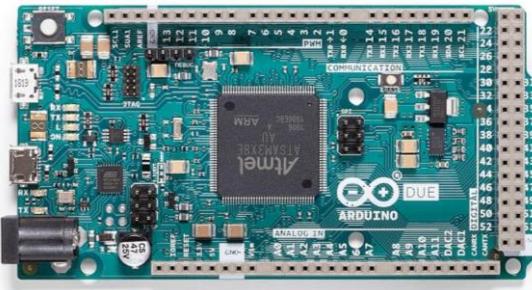


Figura 70: Arduino DUE

Arduino Due è una scheda basata su CPU Atmel SAM3X8E ARM Cortex-M3 (a 32 bit) con clock a 84MHz. È dotato di 54 pin di ingresso/uscita digitale e 4 UART (porte seriali hardware). Tutti i pin sono a interrupt quindi possono essere sfruttati per la gestione asincrona degli eventi come ad esempio le emergenze.

7.4.3.1 Programma

Il linguaggio di programmazione supportato dalla scheda è il C++ che è un linguaggio ad oggetti. Tuttavia il programma è stato sviluppato con una struttura che richiama il linguaggio C, poiché la scheda adottata in partenza supportava solo quello.

Il programma adottato è uguale a quello sviluppato in precedenza a differenza del fatto che per la generazione degli impulsi non sono stati più utilizzate routine di interrupt, ma la libreria AccelStepper.

7.4.3.2 Cablaggio

Di seguito vengono elencati i pin usati per il cablaggio di Arduino Due alla scheda di condizionamento dei segnali.

ARDUINO DUE	PIN	N. PIN CONNETTORE	COLORE CAVO	
0				
1				
2	M1_CW		ROSA	J8/J6
3	M1_CCW		GRIGIO	J8/J6
4	M2_CCW		GIALLO	J9
5	M2_CW	6 PIN	VERDE	J8/J6
6	PROXY_M1		MARONE	J1/J3

7	PROXY_M2		BIANCO	J1/J3
8	M1_RUN		GIALLO-MARRONE	J10
9			BIANCO-GIALLO	J10
10	MZ_EN	6 PIN	MARRONE-VERDE	J10
11	EMG_CHN		BIANCO-VERDE	J4/J2
12	READY_M1		GRIGIO-ROSA	J4/J2
13	M2_RUN		ROSSO-BLU	J9
14				
15				
16				
17				
18	TX	2 PIN	NERO	J1/J3
19	RX		ROSSO	J1/J3
20				
21				
22	ALM_M1	8 PIN	BLU	J1/J3
23				
24	ALM_M2		BIANCO-BLU	J4/J2
25				
26	ALM_M3		ROSAMARRONE	J1/J3
27				
28	TRIGGER_Z		BIANCO-ROSA	J8/J6
29				
30	PIN_Z1		MARRONE-ROSA	J8/J6
31				
32	PIN_Z2 - RST		GRIGIO-MARRONE	J5/J7
33				
34			BIANCO-GRIGIO	
35				
36			ROSSO-BLU	
37				
38	MZ_MOVING	1 PIN	VIOLA	J4/J2
...				
52				
GND	GND	1 PIN	BIANCO-NERO	J9
GND	GND	1 PIN	BIANCO-ROSSO	J9

Tutti i connettori sono stati marcati con etichetta per semplificare il collegamento, il pin numero 1 nel connettore maschio sulla scheda di condizionamento corrisponde al pin marcato con un triangolo sul connettore femmina.

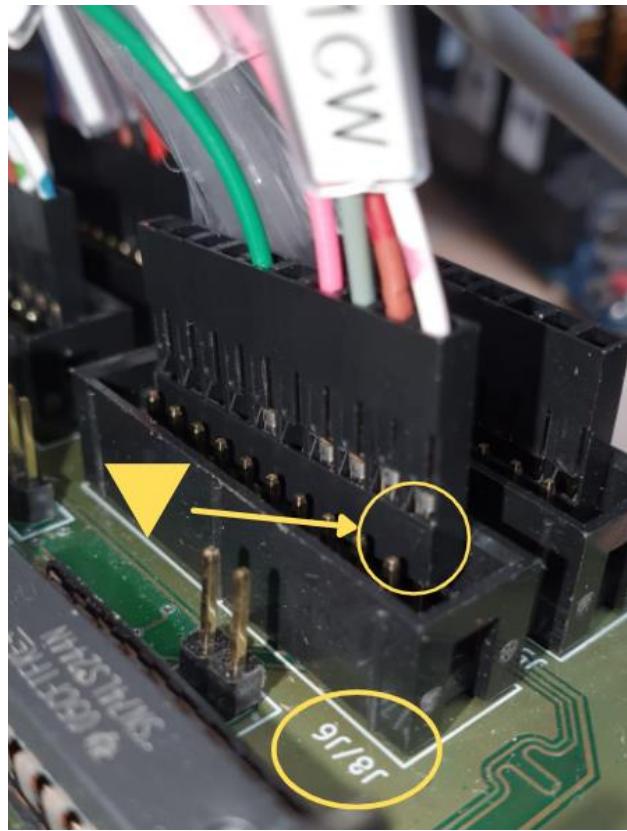


Figura 71: Collegare i connettori

7.5 Condizionamento

Per pilotare i driver con il microcontrollore c'è stato bisogno di sviluppare una scheda di condizionamento segnali che adattasse i livelli di tensione di lavoro dei due sistemi. Nel caso del microcontrollore si hanno delle tensioni di lavoro di 3,3 V e una corrente massima di source di 2 mA, mentre nel caso dei driver è necessaria una tensione di 3 V con una corrente minima richiesta di 9 mA per gli ingressi veloci, questo perché gli ingressi sono gestiti da dei fotoaccoppiatori.

- Position Command Pulse Input (Photocoupler Input)
- Line Driver Input (500 kpps Maximum)
- (+CW:3, -CW:4, +CCW:5, -CCW:6)

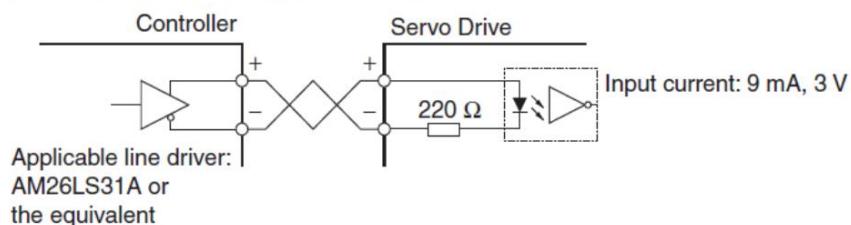


Figura 72: Ingressi 3 V driver Omron

Nel caso degli ingressi di altro genere invece le tensioni di lavoro sono di 24 V.

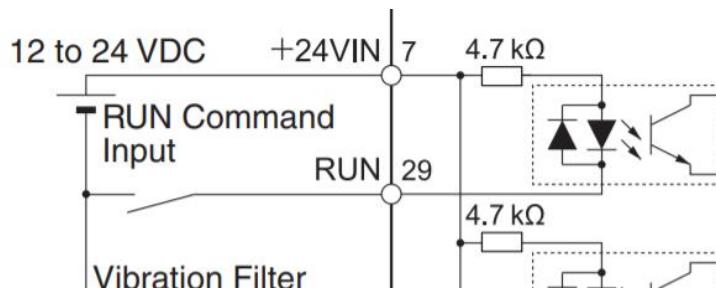


Figura 73: Ingressi 24 V driver Omron

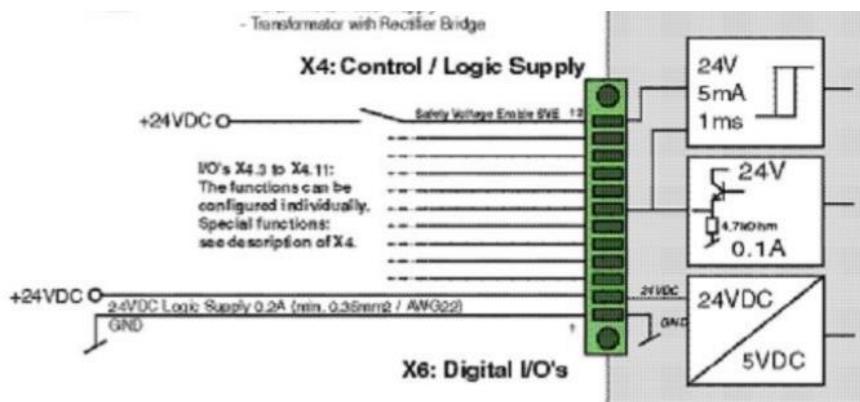


Figura 74: ingressi 24 V driver LinMot

7.5.1 Struttura

La scheda è suddivisa in zone funzionali in base ai segnali da generare.

7.5.1.1 Uscite 24 V con amplificatori operazionali

Per le uscite che devono variare a una frequenza inferiore a 1 kHz abbiamo adottato degli amplificatori operazionali LM324N in configurazione da comparatore. Questa configurazione necessita di una tensione di soglia che è stata generata attraverso un partitore di tensione reso stabile da un amplificatore operazionale in configurazione buffer.

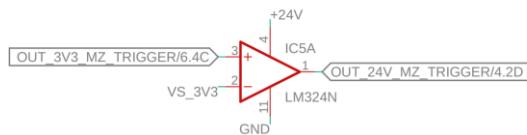


Figura 75: amplificatore operazionale comparatore

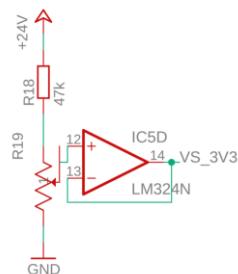


Figura 76: amplificatore operazionale buffer

7.5.1.2 Uscite a transistor

Per le uscite che devono far circolare nel carico una corrente elevata ci siamo serviti di un array di transistor BJT NPN in configurazione Darlington ULN2003A.

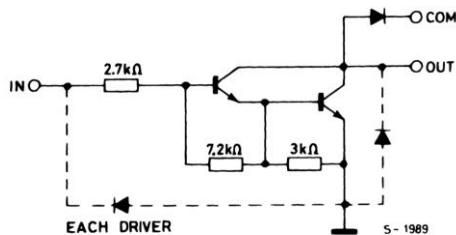


Figura 77: ULN2003A

Questi transistor supportano una corrente massima di 400 mA e possiedono un guadagno di 1000. Abbiamo scelto questo integrato perché possiede già un diodo di protezione da collegare all'alimentazione del driver.

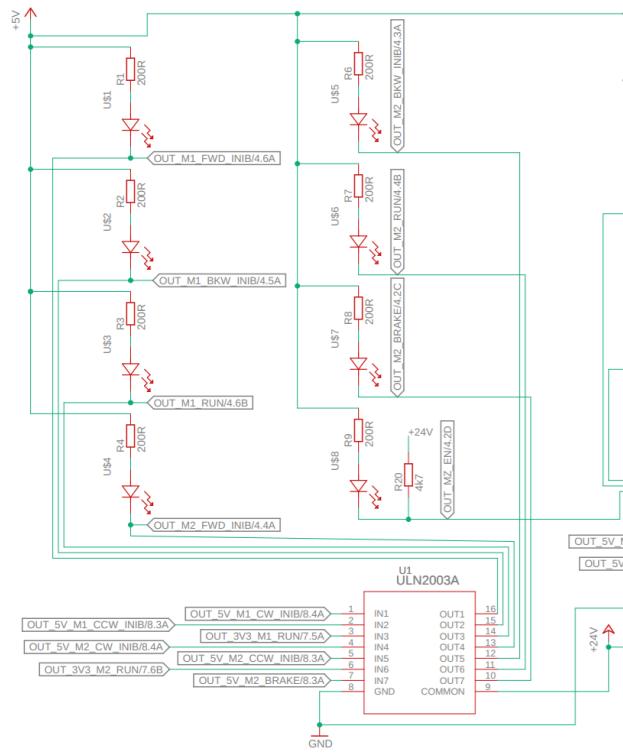


Figura 78: schema di controllo uscite a BJT

Inoltre vi sono collegati dei led con una resistenza per indicare le uscite abilitate.

7.5.1.3 Uscite 5 V ad alta frequenza

Per le uscite che devono avere delle frequenze di lavoro elevate sono stati adoperati dei buffer con ingresso a trigger di Smith SN74LS244N.

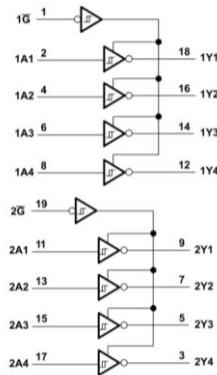


Figura 79: Diagramma logico SN74LS244N

La tensione di soglia è di 2 V, mentre quella di uscita è di circa 3,75 V con un'alimentazione di 5 V.

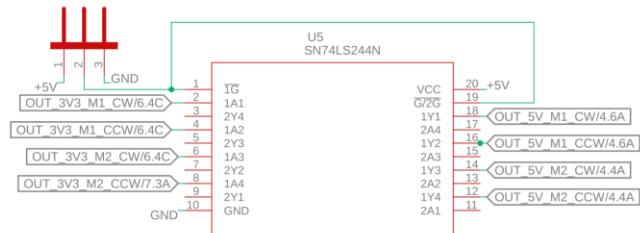


Figura 80: Circuito SN74LS244N

Nel circuito è stata inserita la possibilità di abilitare e disabilitare i buffer attraverso il cortocircuito dei pin di enable alternativamente a 0 V (attivato) o a 5 V (disattivato).

7.5.2 Connettori

I connettori sono stati scelti in modo da agevolare il più possibile il cablaggio ed evitare errori accidentali. Inoltre sono stati predisposti ingressi e uscite supplementari per agevolare decisioni future di espansione del progetto.

7.5.2.1 Hercules RM57Lx

per il microcontrollore abbiamo deciso di replicare allo stesso modo i connettori anche se non utilizziamo tutte le sue uscite/ingressi, quindi abbiamo adottato 6 connettori a cassetta: 4 da 20 pin e 2 da 50 pin.

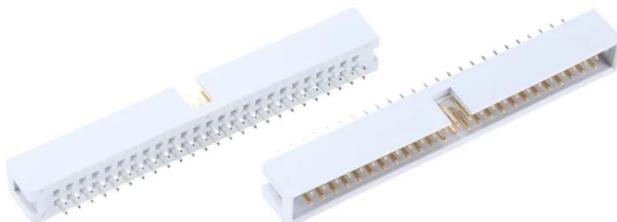


Figura 81: Connettori a cassetta 50 pin



Figura 82: connettore a cassetta 20 pin

I connettori utilizzati sono stati scelti per la presenza di un dentino che rende impossibile un errato collegamento.

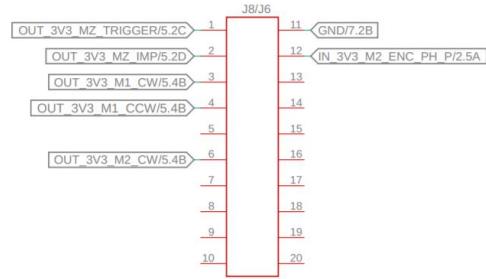


Figura 83: Circuito connettore 20 pin J8/J6

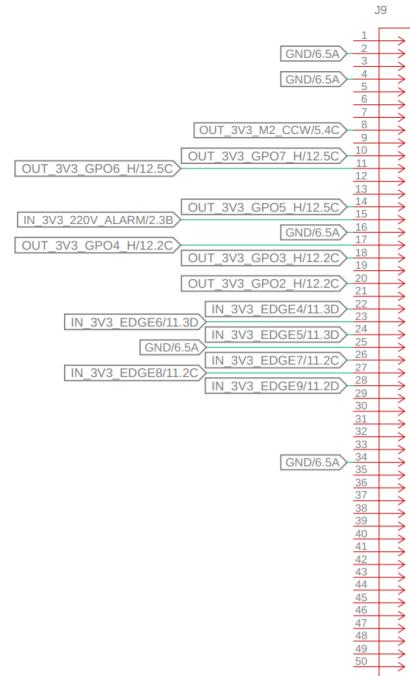


Figura 84: Circuito connettore 50 pin J9

7.5.2.2 Driver Omron

Per i driver Omron abbiamo deciso di replicare allo stesso modo il connettore di comando (CN1), quindi abbiamo adottato un connettore a cassetta da 50 pin. I pin che non sono stati utilizzati sono stati portati in 3 connettori a cassetta da 10 pin l'uno per permettere un futuro sviluppo di nuove funzionalità senza bisogno di cambiare la scheda di controllo.

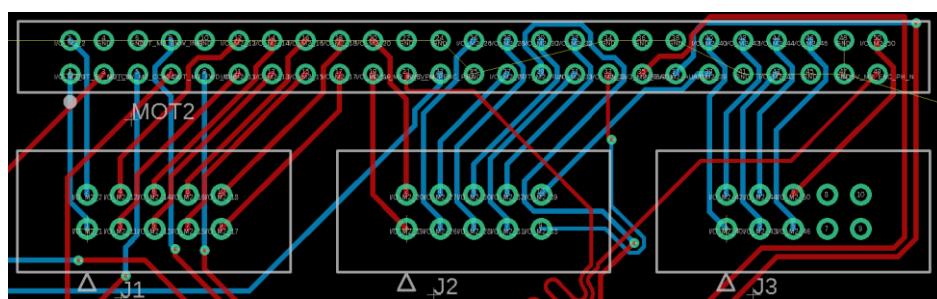


Figura 85: Circuito driver Omron

7.5.2.3 Driver LinMot

Per il driver LinMot abbiamo utilizzato un connettore a vaschetta da 10 pin.

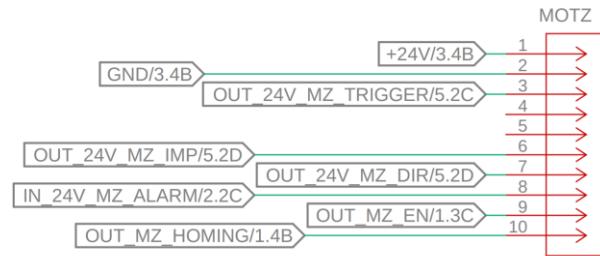


Figura 86: Circuito driver LinMot

7.5.2.4 Arduino Mega

Per Arduino Mega è stato utilizzato un connettore a cassetta da 50 pin.

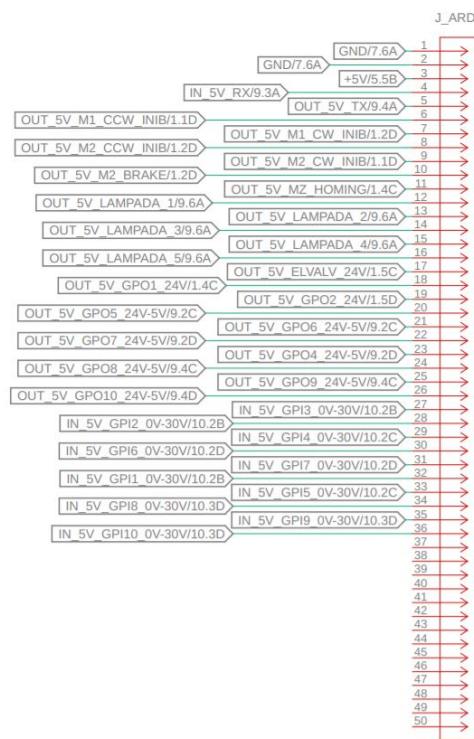


Figura 87: Connettore Arduino Mega

7.5.3 Alimentazioni

La scheda viene alimentata da una tensione di 24 V che viene portata tramite un connettore a vite a due vie, per consentire una sostituzione più agevole in caso di malfunzionamento.



Figura 88: Connettore a vite 2 vie

Nella scheda vi sono due step-down che servono a generare le tensioni di 5 V e 3,3 V necessarie al funzionamento dei circuiti elettronici.



Figura 89: Step-down

Lo step-down per abbassare la tensione sfrutta la tecnica PWM, che genera impulsi ad alta frequenza e di conseguenza disturbi elettromagnetici. Per questo motivo è necessario posizionare le piste di controllo lontane da esso.

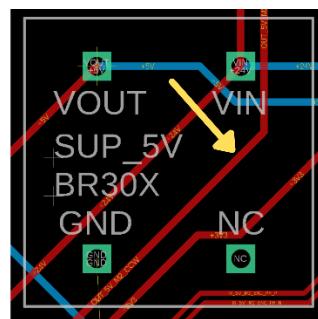


Figura 90: Posizionamento pista errato

Come indicato in figura dalla freccia vi è una pista di segnale esattamente sotto a questo componente che viene altamente disturbata. Per questo motivo siamo stati costretti a cablare in maniera mobile un secondo circuito di condizionamento lontano dallo step-down.

8 Interfaccia con il mondo esterno: ROS

ROS è un meta-sistema operativo in continua crescita, per cui sono state sviluppati librerie software open source e tools al fine di aiutare gli sviluppatori di applicazioni robot. Acronimo di Robot Operating System, a discapito del nome non si identifica in un sistema operativo nel senso stretto del nome. Viene definito infatti come meta-operating system, inglobando sì le caratteristiche tipiche di un vero e proprio sistema operativo (astrazione dell'hardware sottostante, gestione dei processi, package management, gestione dei dispositivi) ma arricchendolo con elementi tipici di un middleware (fornisce l'infrastruttura per la comunicazione tra processi/macchine differenti), e di un framework (tools di utilità per lo sviluppo, debugging e simulazione).

ROS opera essenzialmente su piattaforme Unix-Based, anche se sono innumerevoli le piattaforme sperimentali. ROS attualmente supporta tre differenti linguaggi: C++, Phyton, LISP.

Ci si potrebbe probabilmente chiedere perché usare ROS e quali vantaggi avremmo per il nostro software. D'altronde per quanto la curva di apprendimento possa essere bassa, diventare abili nell'uso di un nuovo framework richiede sempre tempo. In realtà sono innumerevoli le situazioni in cui l'uso di ROS apporta un effettivo grande contributo:

- **Sistemi Distribuiti:** Innumerevoli robot ai giorni nostri contano su software basato sulla cooperazione di molti processi che non risiedono sulla stessa macchina. Alcuni possono integrare in sé più computer, ognuno dei quali controlla un gruppo di sensori o attuatori. Uno stesso task può essere raggiunto con la coordinazione di più robot che devono agire all'unisono per risolvere il problema. *In tutti i suddetti casi coesiste la necessità di poter comunicare tra processi*, siano essi insiti nello stesso robot o suddivisi tra differenti. ROS permette di risolvere questo problema basandosi su una struttura di nodi, messaggi e topic.
- **Standardizzazione e riuso del codice:** Il rapido progresso della robotica ha portato allo sviluppo di algoritmi sempre più complessi ed efficaci per ovviare ai tipici problemi quali ad esempio la navigazione, il mapping di una zona, la scelta del percorso migliore. Di pari passo avere un'implementazione stabile di tali algoritmi senza dover ogni volta re implementarli risulterebbe utile. Ciò è garantito da ROS e dalla grande comunità di

ricerca che lo sostiene alle spalle grazie a packages debuggati che possono essere usati facilmente.

- **Testing:** Il testing su software per applicazioni nell'ambito della robotica richiede più tempo ed è più incline alla presenza di errori. Per di più non sempre è disponibile fisicamente il robot e anche se fosse accessibile spesso il processo di testing risulta macchinoso e lento. ROS permette invece di sviluppare sistemi in cui la parte di controllo a livello hardware sia separata dalla gestione dei meccanismi di più alto livello, e di testare quest'ultimi simulando l'hardware e il software di basso livello.

8.1 Struttura di ROS

ROS è basato su un'architettura a grafo dove il processamento avviene nei nodi, che comunicano tra loro in maniera asincrona attraverso lo scambio di messaggi che vengono pubblicati su topic. I nodi possono pubblicare e/o sottoscriversi ai topic.

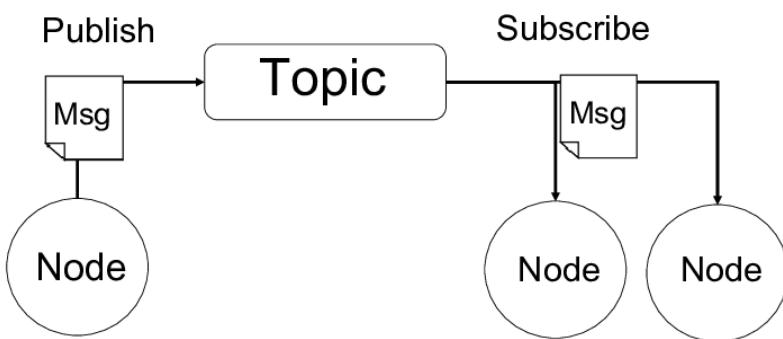


Figura 91: Struttura di ROS

8.2 Messaggi

La comunicazione tra nodi avviene tramite scambio di messaggi. In ROS un messaggio è una struttura dati fortemente tipizzata. Sono supportati tutti i tipi standard primitivi (integer, floating point, boolean etc.) così come array di tipi primitivi e costanti. Ogni messaggio può essere composto da altri messaggi o array di altri messaggi, arbitrariamente nidificati come se fossero una matraccia. La struttura di un messaggio è descritta da un semplice file di testo denominato msg file, di estensione .msg, contenuto nelle sottocartelle dei package. Un msg file è costituito da due parti: campi (fields) e costanti (constants). I campi sono i dati spediti all'interno del messaggio, mentre le costanti sono valori numerici utili a interpretare il significato dei campi.

8.3 Topic

I topic costituiscono il mezzo di comunicazione asincrono, unidirezionale, per lo scambio di messaggi tra nodi, secondo una semantica di tipo publish/subscribe. Ci possono essere più publisher e subscriber concorrenti per un singolo topic, e un singolo nodo può pubblicare e/o sottoscriversi a più topics. In generale, publisher e subscriber non sono consapevoli dell'esistenza degli altri, disaccoppiando così la produzione dell'informazione dal suo consumo. Ogni topic è fortemente tipizzato dal tipo di messaggio che viene pubblicato, e i nodi possono ricevere solo messaggi il cui tipo faccia matching. Ciò determina che all'interno del topic sia possibile scrivere o leggere un solo tipo di messaggio.

8.4 Nodi

Un nodo è un processo che compie una qualsiasi attività computazionale all'interno del sistema ROS. Essendo ROS progettato per essere modulare e fine-grained, tipicamente un sistema basato su di esso comprende numerosi nodi, e in tal contesto sono interpretabili come moduli software, ognuno dei quali incaricato di gestire un aspetto del comportamento del robot, come ad esempio la parte decisionale, il movimento, l'azionamento dei motori etc. Un sistema il cui carico computazionale venga ripartito tra i vari nodi di cui è costituito ha innanzitutto il vantaggio di una maggiore tolleranza agli errori, potendo gestire il crash del singolo nodo. La complessità del codice è ridotta se confrontata coi sistemi monolitici e i dettagli implementativi sono nascosti in quanto i singoli nodi offrono un'interfaccia composta da una API minimale.

8.4.1 ROS e ROSBOT

Un elemento fondamentale del nostro progetto è rendere ROSBOT facilmente accessibile dall'esterno tramite un sistema di comunicazione solido e standard. Per questo motivo abbiamo deciso di implementare l'invio dei comandi e la ricezione dello stato del robot tramite ROS. Proiettando ROSBOT in una realtà aziendale questo sistema risulterebbe molto comodo perché permetterebbe a tutti i dispositivi appartenenti alla stessa rete a cui è connesso il robot di monitorarlo e controllarlo.

Abbiamo installato ROS Noetic su Ubuntu Server 20.04 su Raspberry Pi 4, il nostro sistema si basa su due nodi e due topic:

1. Un nodo è un **server TCP/IP** che apre la comunicazione con il mondo esterno a qualsiasi dispositivo con un programma o app di comunicazione TCP/IP. Siccome non esiste una release stabile di ROS per Windows, quindi tramite questo nodo eliminiamo la necessità di avere ROS installato sulla macchina che controlla il robot. Abbiamo controllato ROSBOT da diversi PC e smartphone senza problemi.
2. Il secondo nodo **comunica i comandi via seriale** al microcontrollore che effettivamente movimenta il robot.

Questi nodi comunicano tra loro tramite due topic:

1. Il primo topic supporta un messaggio personalizzato da noi che contiene tutte le informazioni necessarie alla movimentazione del robot: tipologia di comando, coordinate target, velocità, stato della pinza, movimentazione dell'asse Z, condizioni booleane e ritardi. Il nodo server riceve dall'esterno queste informazioni tramite un linguaggio da noi definito, le interpreta e popola un messaggio per poi pubblicarlo su questo topic. Il nodo seriale invece si sottoscrive a questo topic e riceve le informazioni, le traduce in un messaggio comprensibile al microcontrollore e lo invia.
2. Il secondo topic supporta un messaggio di tipo String che contiene lo stato attuale del robot: emergenza, attesa comando, reset, esecuzione del comando e comando eseguito. Il nodo seriale, siccome è in collegamento diretto con il microcontrollore riceve lo stato attuale del robot e lo pubblica su questo topic. Il nodo server è sottoscritto al topic e non appena un messaggio è disponibile lo legge e lo invia al client a cui è collegato.

Per semplificare il controllo di ROSBOT abbiamo realizzato un programma in VB.net con interfaccia grafica che implementa un client TCP/IP. Il software permette di inviare comodamente comandi, mostrare lo stato attuale del robot e creare cicli di funzionamento.

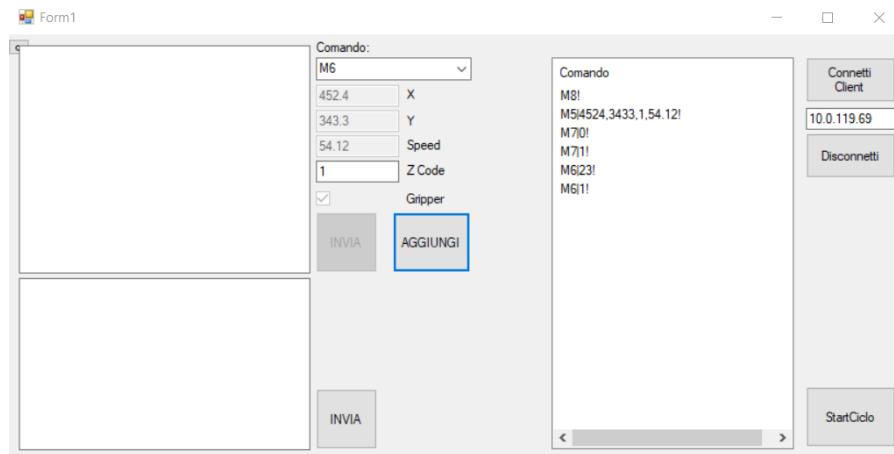


Figura 92: ROSBOT-Client

8.4.2 Installazione

Il sistema è stato testato e sviluppato su Ubuntu Server 20.04.02 LTS che può essere scaricato dalla [pagina ufficiale](#) di Ubuntu. L’alternativa più semplice per installare il sistema operativo su Raspberry Pi è scaricare il programma Raspberry Pi Imager che permette di selezionare il sistema operativo desiderato e integra la procedura di caricamento (flash) su scheda SD.

Una volta installato Ubuntu bisogna procedere con il primo avvio di Raspberry. Dopo qualche minuto vengono chiesti nome utente e password di accesso che di default sono “ubuntu” e “ubuntu”.

Per connettersi alla rete bisogna spegnere Raspberry ed inserire la sua scheda SD in un PC. Apparirà una risorsa “System-boot” in cui si trova il file “network-config”. Dopo aver aperto il file bisogna rimuovere gli “#” per ottenere la seguente struttura:

```
wifis:
wlan0:
  dhcp4: true
  optional: true
  access-points:
    <wifi network name>:
      password: "<wifi password>"
```

Ora bisogna reinserire la scheda SD dentro Raspberry e avviarlo, al termine della procedura di avvio Ubuntu stamperà a schermo che la connessione a internet è attiva. Ora si potrebbe procedere con l’installazione di ROS, ma Ubuntu Server non ha interfaccia grafica e diventa difficile muoversi tra i file se non si è pratici.

Per passare all'interfaccia grafica bisogna installare la versione Desktop di Ubuntu seguendo questi comandi:

```
sudo apt update  
sudo apt upgrade  
sudo systemctl reboot  
sudo apt install ubuntu-desktop  
sudo systemctl reboot
```

Ora si può procedere con l'installazione di ROS Noetic seguendo la guida ufficiale.

8.4.3 ROS Workspace

Da terminale eseguire i seguenti comandi:

1. ***mkdir ros_ws*** (Con questo comando verrà creata la directory `ros_ws` in cui faremo il setup dell'ambiente ROS.)
2. ***cd ros_ws*** (Con questo comando ci spostiamo nella directory `ros_ws`)
3. ***mkdir src***
4. ***cd src***
5. ***catkin_init_workspace*** (Questo comando inizializza l'ambiente di sviluppo)
6. ***cd ..*** (Ci spostiamo indietro da `src` a `ros_ws`).
7. ***catkin_make*** (Questo comando crea tutti i file necessari e le cartelle per l'utilizzo di ROS)

 Questo comando va lanciato quando si è posizionati nella cartella `ros_ws` e non `src`.

8. ***source /opt/ros/noetic/setup.bash***
9. ***source devel/setup.bash***
10. ***cd /ros_ws/src***
11. ***catkin_create_pkg NOME_PACCHETTO rospy roscpp std_msgs geometry_msgs***
(Questo comando permette la creazione di un nuovo pacchetto, dopo il nome del pacchetto si inseriscono le dipendenze che questo deve avere).
12. Dentro la cartella del pacchetto creare una cartella chiamata “scripts” utilizzando il comando ***mkdir***.
13. All'interno della cartella salvare gli script dei nodi.

8.4.4 Il messaggio

I dati contenuti nel comando da inviare al robot sono stati raggruppati in unico messaggio ROS da noi implementato che contiene i seguenti dati:

```
string commandType #Contiene il tipo di comando
geometry_msgs/Point32 nextPoint #Contiene tre variabili di tipo float x,y,z
float32 speed #Contiene la velocità
float32 acceleration #Contiene l'accelerazione
int32 gripper #Stato della pinza
int32 zAxisCommand #Programma dell'asse Z
int32 inputNumber #GPIO da verificare per una condizione if
```

```
int32 waitTime #Delay
```

8.4.5 Il nodo server

All'avvio lo script apre una porta per la comunicazione TCP/IP tramite il modulo *socket* di Python e attende una connessione da un client esterno.

```
if __name__ == '__main__':
    HOST = '10.0.119.69' # Nome dell'host o IP
    PORT = 12000      # Porta usata dal server
    rospy.init_node('server_scara') #nome del nodo
    pub=rospy.Publisher('/scara_commands',command,queue_size=10)
    sub = rospy.Subscriber('/scara_status', String ,statusSerial)
    #Il nodo pubblica sul topic '/scara_commands' e si sottoscrive a '/scara_status'
    subServer((HOST,PORT))
```

Questo tipo di comunicazione ha bisogno dell'indirizzo IP dell'HOST e di una porta. L'indirizzo IP deve corrispondere a quello con cui Raspberry è registrata sulla rete a cui è connessa. Per conoscerlo basta lanciare un comando *ifconfig* da terminale. Questo indirizzo va inserito nella variabile "HOST" presente nello script. Una possibilità di miglioramento dello script è leggere questo parametro da un file di testo presente sul computer in modo che non ci sia bisogno di aprire il file Python. Per quanto riguarda la porta bisogna scegliere un numero positivo maggiore di 1023, nel nostro caso abbiamo scelto la porta 12000.

Quando la comunicazione con un client esterno è stabilita lo script prosegue con l'inizializzazione del corrispondente nodo ROS. Questo nodo pubblica sul topic *"/scara_commands"* che accetta messaggi di tipo *command.msg* e si sottoscrive al topic *"/scara_status"* che accetta messaggi di tipo *String*.

Successivamente lo script legge costantemente il buffer della comunicazione e non appena riceve un messaggio in ingresso verifica che la sua sintassi sia corretta, in caso positivo risponde al client con la stringa "\$OK", in caso negativo risponde "\$ERR".

```
def ricevi_comandi () :
    global stato
    global msg_comando
    global conn
    #Istanzia publisher(topic, tipo topic, buffer msg)
    rate = rospy.Rate(50) #50Hz
    msg_comando = command()
```

```

while not rospy.is_shutdown():
    try:
        richiesta = conn.recv(4096)
        data = str(richiesta, "utf-8")
        letturaComando(data)
    except socket.timeout:
        print("timeout")
        rate.sleep()

```

Il messaggio che viene inviato dal client deve avere la seguente sintassi:

<NOME COMANDO> + “|” + <PARAMETRI SEPARATI DA , > + “!” (comando con parametri)
<NOME COMANDO> + “!” (comando senza parametri)

Ad esempio considerando i seguenti comandi:

1. M5 di movimento in joint: *M5/23,456,1,300!*
2. M8 di homing: *M8!*
3. S4 di settaggio accelerazione: *S4/1000!*

Ricevuto il comando lo script per prima cosa verifica la presenza del “!” e lo rimuove dalla stringa, successivamente verifica la presenza del “|” per capire se il messaggio contenga parametri o meno.

1. Nel caso in cui non ci sia il carattere “|” allora lo script inserisce la stringa che ha ricevuto nel parametro *commandType* del messaggio ROS e il messaggio viene pubblicato sul topic “scara_commands”. Ad esempio nel caso di un “M8!” verrà inserito “M8” in *commandType*.
2. Nel caso in cui “|” sia presente allora il messaggio viene separato in due parti che vengono inserite nella lista di stringhe “type” tramite il metodo *split()* della classe String di Python. Il metodo *split* cerca il carattere “|” e divide la stringa originale in due sottostringhe. Nel primo elemento della lista ci sarà la sottostringa che identifica il *commandType*, nel secondo elemento ci saranno i parametri separati da “,”. A questo punto il secondo elemento della lista viene a sua volta separato usando come separatore il carattere “,” e le sottostringhe risultanti verranno aggiunte alla lista “parametri”. A questo punto i parametri vengono passati ad un metodo che in accordo con il *commandType* popola il messaggio da pubblicare sul topic “scara_commands”

```

def letturaComando (data) :
    global stato
    global msg_comando
    global parametri
    global conn
    risposta = ""

    if data.startswith("M") or data.startswith("C") or data.startswith("S"):
        if data.endswith(endString):
            data = data.replace(endString,"")
            if "|" in data:
                tipo = data.split("|")
                msg_comando.commandType = tipo[0]
                parametri = tipo[1].split(",")
                print(tipo[0])
                print(parametri)
                composer(data)
                pub.publish(msg_comando)
                risposta = okMessage
                msg_comando = command()
            else:
                msg_comando.commandType = data
                pub.publish(msg_comando)
                risposta = okMessage
                msg_comando = command()
        else:
            risposta = errMessage
            print("ERR: Il messaggio deve finire con !")
    else:
        risposta = errMessage
        print("ERR: Il messaggio deve finire con !")

    conn.sendall(risposta.encode())

```

Il nodo si sottoscrive al topic “scara_status” su cui vengono pubblicati messaggi relativi allo stato effettivo del robot. Quando un messaggio sul topic è disponibile il nodo esegue il metodo *statusSerial* dove ripubblica il messaggio al client. In questo modo si ha un feedback sullo stato del robot. Gli stati in cui il robot può trovarsi sono i seguenti:

1. COMANDO: <comando> (Il robot sta eseguendo il comando)
2. COMANDO_ESEGUITO (Il robot ha eseguito il comando inviato)
3. EMERGENZA (Fungo di emergenza attivato)
4. RESET_ESEGUITO (Reset del robot eseguito)
5. ERROR (Il robot ha riscontrato un errore nei comandi ricevuti)

```
def statusSerial (msg) :  
    global conn  
    fixed = str(msg).replace("data:",")"  
    fixed = fixed.replace("","",)  
    fixed = fixed.replace(' ','')  
    fixed = fixed + "%"   
    print(fixed)  
    conn.sendall(fixed.encode())
```

8.4.6 Il nodo seriale

All'avvio questo script tenta la connessione seriale con il microcontrollore di ROSBOT, se la connessione riesce allora lo script procede altrimenti mostra un messaggio di errore e si chiude automaticamente.

```
try:  
    seriale = serial.Serial(port = portName, baudrate = baudRate)  
    seriale.stopbits = 1  
    print("Seriale connessa con il dispositivo: " + portName)  
except:  
    print("[ERR] Connessione seriale non riuscita ...")
```

Successivamente lo script avvia il nodo e lo sottoscrive al topic “scara_commands” e gli permette di pubblicare sul topic “scara_status”.

```
rospy.init_node('comSerial_scara') #nome del nodo  
sub = rospy.Subscriber('/scara_commands', command ,comSerial)  
pub = rospy.Publisher('/scara_status', String ,queue_size = 10)  
print("Nodo Inizializzato correttamente!")
```

Quando è disponibile un comando sul topic “scara_commands” allora il programma si interrompe e salta direttamente al metodo *comSerial* (come se ci fosse un interrupt). In questo metodo il messaggio viene letto e poi tradotto nella sintassi implementata nel microcontrollore. Se il messaggio è di emergenza (S3!) o reset (S2!) questo viene subito inviato al microcontrollore, gli altri messaggi invece vengono aggiunti ad una coda che viene svuotata in modo asincrono.

```
q = queue.Queue(maxsize = -1)  
def comSerial(_command):  
    global isExecutable  
    global emrSTATE  
    global seriale  
    global stato  
  
    type = _command.commandType  
    zCommand = _command.zAxisCommand  
    targetPoint = _command.nextPoint  
    speed = _command.speed  
    acceleration = _command.acceleration  
    targetX = Decimal(_command.nextPoint.x)
```

```

targetY = Decimal(_command.nextPoint.y)
targetZ = Decimal(_command.nextPoint.z)
speed = Decimal(_command.speed)
acceleration = Decimal(_command.acceleration)
targetX = round(targetX,prec)
targetY = round(targetY,prec)
targetZ = round(targetZ,prec)
speed = round(speed,prec)
acceleration = round(acceleration,prec)
gripper = _command.gripper
delay = _command.waitTime
messaggio = ""

if type == "M1" or type == "M2" or type == "M3" or type == "M4":
    parametri = "X"+str(targetX) + "Y"+str(targetY) + "V"+ str(speed)
    messaggio = type + parametri
if type == "M5":
    parametri = "P"+str(targetX) + "S"+str(targetY) + "V"+ str(speed)
    messaggio = type + parametri
if type == "M6":
    messaggio = type + "Z"+ str(zCommand)
if type == "M7":
    messaggio = type + "S"+ str(gripper)
if type == "M8" or type == "S2" or type == "S3":
    messaggio = type
if type == "C1":
    messaggio = type + "T" + str(delay)
if type == "S4":
    messaggio = type + "A" + str(acceleration)
messaggio = messaggio + "!"

if(messaggio == "S3!"):
    emrProcedure()

elif(messaggio == "S2!"):
    clearSerial()
    risposta = sendSerial(messaggio)
    if risposta == okMessage:
        if(isFromSerial(finishCommandMessage)):
```

```

    clearSerial()
    q.queue.clear()
    stato = _ATTESA_COMANDO
    pub.publish(rstString)
    print("RESET effettuato")

else:
    q.put(messaggio)
    print("Aggiungo " + messaggio + " alla coda")

```

Lo script continua a rimanere in un ciclo *while* con frequenza 50Hz che preleva comandi dalla coda e li invia al robot quando esso ha completato il comando precedente. Questo sistema permette di inviare anche decine di comandi al nodo che verranno eseguiti in base ai tempi di esecuzione dettati dal robot.

Questo nodo inoltre, essendo in diretto collegamento con il robot pubblica il suo stato sul topic “scara_status”. I comandi di stato sono i seguenti:

1. &COMANDO: <comando> (Il robot sta eseguendo il comando)
2. &COMANDO_ESEGUITO (Il robot ha eseguito il comando inviato)
3. &EMERGENZA (Fungo di emergenza attivato)
4. &RESET_ESEGUITO (Reset del robot eseguito)
5. &ERROR (Il robot ha riscontrato un errore nei comandi ricevuti)

8.4.7 Il client

Il programma del client è stato sviluppato per PC Windows in Vb.net:

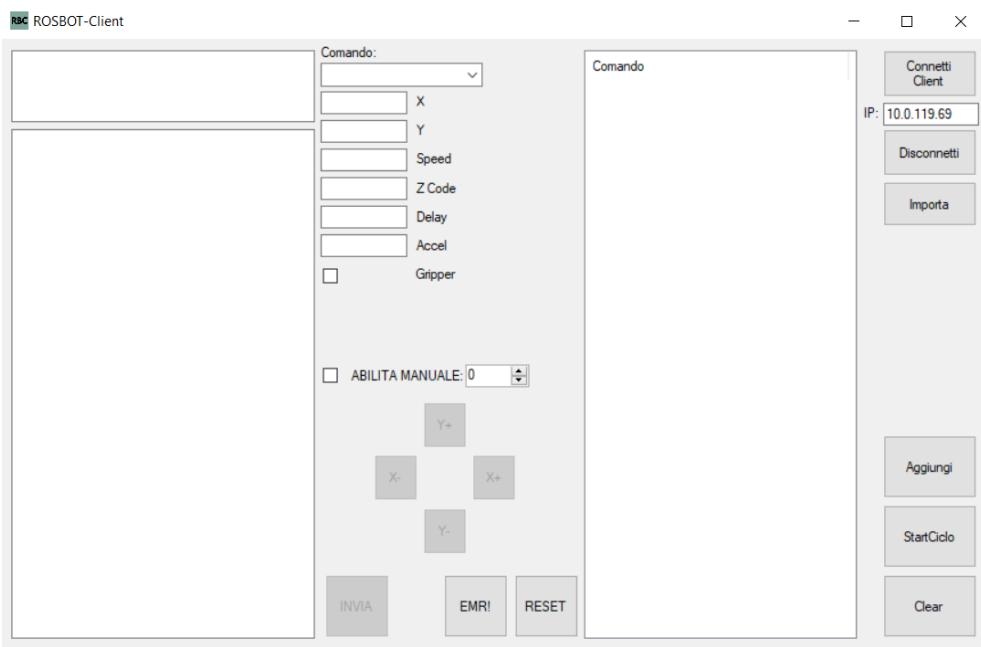


Figura 93: Client

All'avvio del programma bisogna connettere il client al server inserendo l'IP di cui abbiamo parlato nelle pagine precedenti. In caso di collegamento effettuato si riceve un messaggio di conferma sia lato client che server. Tramite la tendina si può scegliere il tipo di comando desiderato e automaticamente si abiliteranno i campi necessari per quel comando. A questo punto il comando può essere inviato direttamente al server oppure lo si può inserire in una lista che viene prodotta ciclicamente.

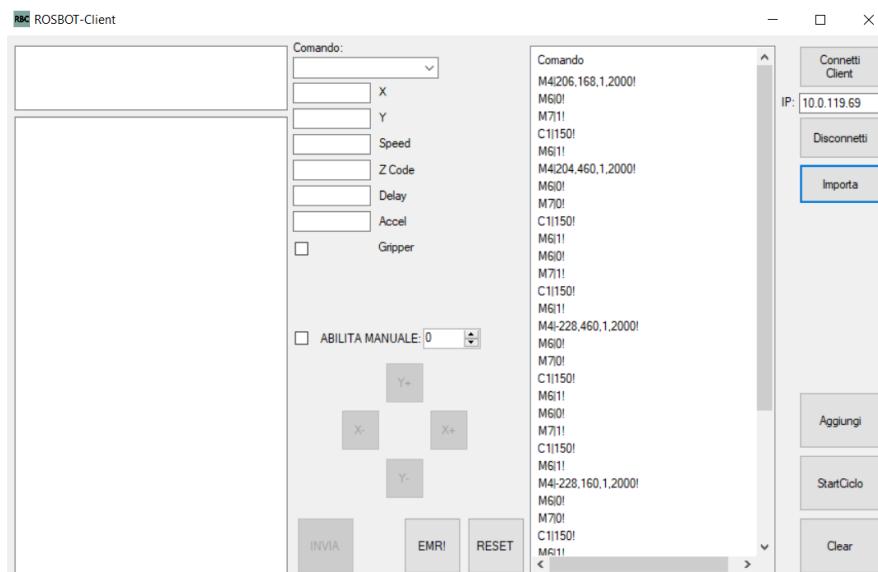


Figura 94: Client

Per favorire la creazione di cicli più o meno complessi è stato implementato uno pseudo linguaggio di programmazione. I comandi terminati da “!” vengono elencati in un file .TXT che viene caricato e letto dal programma. Il codice di seguito implementa un semplice pick-and-place tra due punti.

```
M4,-100,340,2000!
M6,0!
M7,1!
C1,150!
M6,1!
M4,72,510,2000!
M6,0!
M7,0!
C1,150!
M6,1!
M4,300,300,2000!
M4,72,510,2000!
M6,0!
M7,1!
C1,150!
M6,1!
M4,-100,340,2000!
M6,0!
M7,0!
C1,150!
M6,1!
```

Il client sfrutta un timer che ogni 10ms richiama un metodo chiamato *tmrData_Tick* in cui viene verificata la presenza di messaggi nel buffer TCP/IP. Tutti i messaggi di stato sono terminati dal carattere “%” quindi quando viene verificato il buffer la stringa viene suddivisa con il metodo *split* e le sottostringhe risultanti vengono aggiunte ad una coda di stringhe.

```
'Se ci sono dati disponibili
Dim cmd As String = ""
If Client.Available > 0 Then
    'Li legge dallo stream
    Dim Buffer(Client.Available - 1) As Byte
    NetStream.Read(Buffer, 0, Buffer.Length)
    Dim Msg As String = UTF8.GetString(Buffer)
```

```

'If Msg.StartsWith("&") Then 'MESSAGGI DI STATO ROBOT
For Each n As String In Msg.Split("%")
    If Not n = "" Then
        queue_comando.Enqueue(n)
    End If
Next

' End If
End If

```

Successivamente, nel caso in cui ci siano messaggi nella coda, viene estratta una stringa da essa e in base al suo valore viene determinato lo stato del robot e di conseguenza il comportamento che deve seguire il software.

```

If queue_comando.Count > 0 Then
    cmd = queue_comando.Dequeue()
    If cmd.StartsWith("&") Then
        cmd = cmd.Replace("&", "")
        If cmd.Contains("COMANDO:") Then
            state = commandState
        Else
            state = cmd
        End If
    End If
    lblStato.Text = cmd

```

Una volta valorizzata la variabile *state* con lo stato del robot una struttura *select-case* gestisce il funzionamento del programma e lo coordina con lo stato attuale del robot.

Il metodo per inviare comandi si chiama *inviaComando*, esso invia il comando desiderato e attende la risposta “\$OK” o “\$ERR” dal server. Il timer però potrebbe interferire con questi messaggi che invia il server al client “rubandoli”. Per risolvere questo problema la prima istruzione del metodo *inviaComando* ferma il timer e l’ultima lo riavvia.

```

Private Sub inviaComando(comando As Comando)
    Dim risp As String
    Dim Buffer() As Byte
    tmrData.Stop()
    If Not String.IsNullOrEmpty(comando.getCommandType) Then
        Buffer = UTF8.GetBytes(comando.getCommandType + comando.getValue +
endCommand)
    End If

```

```
txtLog.AppendText("Inviato: " & comando.get CommandType + comando.getValue +  
endCommand + Environment.NewLine())  
NetStream.Write(Buffer, 0, Buffer.Length)  
risp = controllo().Trim  
If Not risp = okMessage Then  
    state = errorState  
    MsgBox("ERRORE DI COMUNICAZIONE CON IL SERVER!")  
    Exit Sub  
End If  
txtLog.AppendText("Ricevuto: ")  
txtLog.AppendText(risp)  
txtLog.AppendText(Environment.NewLine)  
End If  
tmrData.Start()
```

9 Possibili miglioramenti

9.1 ROS2

ROS2 è un’evoluzione di ROS in cui non è necessaria la presenza del master detto *roscore* per il funzionamento di tutti i nodi presenti nella rete. Quest’architettura ha il vantaggio di essere più stabile in quanto l’evenienza di un crash del master non esiste. Tuttavia questo framework, essendo più recente, non è così diffuso e le informazioni tuttora presenti non sono sufficienti per le nostre capacità di sviluppo.

9.2 Sistema di visione

Un’integrazione aggiuntiva molto interessante sarebbe un sistema di visione basato sulle librerie open source OpenCV. Si potrebbe realizzare un sistema di visione standalone installato su un Raspberry Pi dedicato che comunica i dati al Raspberry Pi server tramite ROS.

9.3 Periferiche aggiuntive

Potrebbero essere sviluppati nodi ROS da installare sul Raspberry di ROSBOT predisposti per altri sistemi industriali che supportano ROS come ad esempio macchine utensili, sistemi di visione, magazzini...

9.4 Implementazione di nuove movimentazioni

Si possono realizzare la movimentazione lineare e circolare. La movimentazione lineare è già stata scritta e testata su simulatore, manca la fase di debug vera e propria e la verifica dell’implementazione su microcontrollore. Un passo ulteriore sarebbe la possibilità di raccordare i movimenti tramite traiettorie raccordate.

10 Ringraziamenti

Questa tesi rappresenta per noi un grande traguardo. Dal primo anno di università avevamo iniziato quasi per gioco la progettazione di un robot industriale. Verso la metà del secondo anno abbiamo visto questo progetto come un'opportunità di tesi e contemporaneamente ci è stata offerta l'opportunità di riprogettare un robot preesistente quindi abbiamo subito colto l'occasione.

In primo luogo vogliamo ringraziare i nostri cari che hanno reso possibile il nostro percorso di studi. Vogliamo inoltre ringraziare tutte le persone e aziende che hanno contribuito e sostenuto il nostro progetto.

Il professor Adamini Riccardo, nostro relatore; il professor Marti Giuseppe ha fornito un importante sostegno tecnico; il professor Frassine Massimo e gli ingegneri Baù Marco e Ferrari Vittorio hanno fornito un indispensabile aiuto per lo sviluppo elettronico; l'azienda C.T.B. Circuiti Stampati SNC e in particolare la D.ssa Tonini Roberta hanno fornito aiuto e schede elettroniche; UNIBS ha finanziato parte del progetto; Automazioni Industriali Srl e i il suo personale ci hanno fornito supporto tecnico, materiali e un'area di lavoro senza la quale in questo progetto non sarebbe stato sviluppato.