UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea triennale in Ingegneria Informatica e dell'Automazione

---




# Studio e progettazione di un sistema propulsivo per robot biomimetici basato su CAN bus e servomotori BLDC

Study and design of a propulsive system for biomimetic robots based on CAN bus and BLDC servomotors

Relatore:

Prof. David Scaradozzi

Candidata:

Bianca Secci

Anno Accademico 2025/2026

# Sommario

Il presente lavoro ha come obiettivo principale la migrazione del sistema di controllo della coda di un robot biomimetico subacqueo da una comunicazione $I^2C$ al bus CAN. Per raggiungere questo scopo, il progetto prevede l'adattamento della logica del firmware preesistente alla nuova architettura hardware, includendo la configurazione e la programmazione di microcontrollori ESP32. Si mira inoltre all'integrazione di servomotori dotati di interfaccia CAN nativa, al fine di riprodurre il movimento sinusoidale e coordinare in modo efficace l'azionamento dei tre giunti propulsivi. Un ulteriore obiettivo fondamentale del lavoro consiste nel predisporre un sistema per l'acquisizione e la visualizzazione dei dati provenienti dai servomotori e dalle unità di misura inerziale (IMU). Questo passaggio è essenziale per verificare sperimentalmente in laboratorio la corrispondenza tra il movimento teorico richiesto e il reale comportamento fisico osservato nel sistema.

# Indice

**1 Introduzione** **6**

1.1 Contesto e motivazioni . . . . . 6

1.2 Obiettivi del lavoro . . . . . 6

1.3 Struttura della tesi . . . . . 7

**2 Architettura iniziale e componenti hardware** **8**

2.1 Versione di partenza dei codici master e slave . . . . . 8

2.1.1 Master . . . . . 8

2.1.2 Slave . . . . . 9

2.2 Materiali e strumenti utilizzati . . . . . 11

2.2.1 Servomotori Hitec MDB961WP-CAN . . . . . 11

2.2.2 M5Unified . . . . . 17

2.2.3 Atom S3R M5 . . . . . 19

2.2.4 AtomPoE . . . . . 21

2.2.5 Adattatori MiniCAN . . . . . 22

2.2.6 INA226-1A . . . . . 23

2.2.7 Relay Unit2 . . . . . 24

**3 Fondamenti teorici** **26**

3.1 CAN-BUS . . . . . 26

3.1.1 Controller Area Network . . . . . 26

3.1.2 Trasmissione dati . . . . . 26

3.1.3 I frame . . . 27
3.1.4 Data frame . . . 27
3.1.5 Data field . . . 28
3.1.6 Standard ISO applicabili . . . 28
3.1.7 Libreria `twai.h` . . . 29
3.2 Ethernet . . . 30
3.2.1 Concetti minimi di rete: indirizzo IP . . . 31
3.2.2 MAC address . . . 31
3.2.3 Porta (port) . . . 31
3.2.4 TCP vs UDP . . . 31
3.2.5 HTTP e REST API . . . 32
3.2.6 DHCP . . . 32
3.2.7 Socket . . . 32
3.3 I2c . . . 33
3.3.1 Struttura del bus e segnali . . . 33
3.3.2 Indirizzamento e trasmissione dei dati . . . 33

**4 Implementazione del sistema** **35**

4.1 Implementazione della trasmissione CAN . . . 35
4.1.1 Messaggio di richiesta . . . 37
4.1.2 Messaggio di risposta . . . 37
4.2 Divisione master/slave multi-core . . . 38
4.2.1 Master . . . 39

4.2.2 Slave . . . . . 43

4.3 Generazione del movimento sinusoidale . . . . . 45

4.3.1 Analisi delle variabili: fase, frequenza, ampiezza . . . . . 45

4.3.2 Variabili di centro e offset (posizionamento spaziale) . . . . . 46

4.3.3 Variabili di stato . . . . . 47

4.3.4 Struttura del payload (7 byte) . . . . . 48

4.3.5 Checksum . . . . . 48

4.4 Implementazione della comunicazione Ethernet . . . . . 49

4.4.1 Architettura dei protocolli e gestione di rete . . . . . 50

4.4.2 Struttura multitasking e gestione concorrente delle sessioni . . . . . 51

4.4.3 Classe TeePrint . . . . . 52

4.4.4 Il protocollo di discovery e lo script PowerShell . . . . . 53

4.5 Funzionalità della versione `IMU_CAN` . . . . . 55

4.5.1 Setup registri . . . . . 55

4.5.2 SYNC (`0x7FF`) . . . . . 57

4.5.3 Stream automatico . . . . . 58

4.5.4 Motor Lock/Unlock . . . . . 59

4.5.5 Lettura IMU . . . . . 61

4.6 Prodotto finale . . . . . 62

4.6.1 Funzionalità della versione `worlds` . . . . . 64

**5 Test e validazione** **69**

5.1 Configurazione sperimentale e scenari di test . . . . . 69

5.2 Procedure di verifica e criteri di validazione . . . . . 70

5.3 Risultati dei test . . . . . 77

**6 Conclusioni** **77**

6.1 Sviluppi futuri . . . . . 78

# 1 Introduzione

## 1.1 Contesto e motivazioni

Il presente lavoro illustra lo studio e le sperimentazioni condotte presso il laboratorio LabMACS dell'Università, nell'ambito delle attività di ricerca dedicate alla robotica biomimetica e alla progettazione di veicoli subacquei autonomi (AUV, *Autonomous Underwater Vehicles*). L'attività costituisce il sottoprogetto `CAN_FIN` del progetto `LL_CAN_FIN_PowerBlock`, dedicato alla piattaforma subacquea TRITONE sviluppata dal team del laboratorio.

La dinamica complessiva del veicolo è descritta mediante il modello di Newton–Eulero, che tiene conto delle forze idrostatiche, idrodinamiche e dei contributi legati al moto del corpo rigido. Il componente specifico oggetto del progetto è invece la coda, responsabile della propulsione del sistema e costituita da tre segmenti articolati. Il suo comportamento è descritto attraverso un modello basato sulla teoria dei corpi allungati di Lighthill.

La base di partenza del progetto `LL_Fin_PowerBlock` era un firmware di basso livello per l'hardware POSEIDRONE. Il programma riceveva comandi da una stazione di controllo remota e gestiva sia la regolazione della profondità, tramite un motore brushless dedicato alla movimentazione verticale, sia i movimenti propulsivi di base della coda: avanzamento rettilineo, svolta a destra e svolta a sinistra. Questi ultimi erano ottenuti mediante tre motori, responsabili dei moti rotazionali oscillatori dei tre giunti. In questa configurazione, un singolo microcontrollore gestiva i motori della coda, il motore di movimentazione verticale, la lettura dei sensori e gli altri task del sistema.

Il progetto iniziale prevedeva il disaccoppiamento della gestione dei tre giunti mediante l'aggiunta di tre microcontrollori e tre unità di misura inerziale (IMU), collegando i nuovi nodi al sistema di controllo preesistente tramite comunicazione $I^2C$. Questa architettura distribuita costituisce il punto di partenza del lavoro descritto nella presente tesi.

La motivazione del lavoro risiede nella necessità di rendere più affidabile e reattivo lo scambio di informazioni e comandi tra il modulo di controllo principale, ossia la "testa" del robot, e i nodi della coda, in vista dell'utilizzo del veicolo in ambiente acquatico.

## 1.2 Obiettivi del lavoro

L'obiettivo principale è stato migrare il sistema di controllo della coda dalla comunicazione $I^2C$ al bus CAN, adattando la logica del firmware preesistente alla nuova architettura hardware. Il lavoro ha previsto la configurazione e la programmazione dei microcontrollori ESP32 e l'integrazione di

servomotori con interfaccia CAN nativa, per riprodurre il movimento sinusoidale e coordinare l'azionamento dei tre giunti.

Un ulteriore obiettivo è stato predisporre l'acquisizione e la visualizzazione dei dati dei servomotori e delle IMU, così da verificare in laboratorio la corrispondenza tra il movimento richiesto e il comportamento osservato.

## 1.3 Struttura della tesi

Il presente elaborato ripercorre cronologicamente le fasi di sviluppo del progetto ed è strutturato come segue. A seguito di questa introduzione, il lavoro propone una ricerca teorica preliminare volta ad analizzare le specifiche e il funzionamento della trasmissione su bus CAN, unita allo studio delle caratteristiche operative dei servomotori selezionati e delle principali funzionalità studiate ed implementate. Nei capitoli successivi viene dettagliata la fase implementativa, descrivendo l'ambiente di sviluppo e la stesura del firmware per i microcontrollori ESP32, illustrando come i codici preesistenti siano stati migrati e adattati al nuovo protocollo. Infine, vengono discussi i risultati delle sperimentazioni condotte in laboratorio per validare il nuovo sistema di trasmissione del robot subacqueo.

# 2 Architettura iniziale e componenti hardware

## 2.1 Versione di partenza dei codici master e slave

La configurazione di partenza prevede un controllo distribuito della coda: il master seleziona il movimento e trasmette i parametri ai tre nodi slave, mentre ciascuno slave genera localmente il segnale PWM destinato al proprio servomotore. Sul bus $I^2C$ vengono quindi scambiati comandi e risposte, non i singoli impulsi di pilotaggio dei motori. Di seguito sono descritte separatamente la versione del master su M5StickC Plus e le funzioni del firmware slave I2C-tritone.

### 2.1.1 Master

**Inizializzazione e riconoscimento dei nodi** Il master inizializza la scheda, la seriale di debug e la UART a 115200 baud, quindi configura il bus $I^2C$ a 100 kHz sui pin GPIO32 (SDA) e GPIO33 (SCL). I tre segmenti sono associati agli indirizzi `0x20`, `0x21` e `0x22`. La funzione `scanEnabledSlaves()` verifica la risposta di ciascun indirizzo e aggiorna i flag di abilitazione dei nodi. Questi flag permettono alle procedure di movimento di riconoscere quali dispositivi risultano disponibili. Al termine dell'avvio viene richiamato il movimento rettilineo.

**Definizione dei movimenti** Per ogni segmento sono definite un'ampiezza e una fase, mentre la frequenza è comune ai tre nodi. Il master non calcola continuamente la posizione dei servo: prepara i parametri della legge sinusoidale che sarà eseguita dagli slave. Le funzioni `moveStraight()`, `moveRight()` e `moveLeft()` selezionano rispettivamente il movimento rettilineo e le due modalità di curvatura. Nella prima gli offset sono nulli; nelle altre viene applicato al primo segmento un bias di $+10°$ oppure $-10°$, mantenendo le ampiezze e le fasi previste.

Sono inoltre presenti tre pose statiche, ottenute annullando l'ampiezza dell'oscillazione: primo segmento a $-45°$, coda interamente allineata oppure primo segmento a $+45°$. Gli altri due segmenti rimangono a zero. La funzione `cambiaMovimento(freq,bias)` consente invece di variare frequenza e bias mediante un comando esterno, limitandoli rispettivamente agli intervalli 0–4 Hz e $-10° \ / \ +10°$.

**Trasmissione e aggiornamento dei parametri** La funzione `sendSetCommand()` costruisce una riga nel formato `SET freq,amp,phase,offset`, terminata da un carattere di nuova riga, e la invia all'indirizzo del nodo interessato. Dopo la scrittura verifica la risposta testuale `ACK`, effettuando fino a tre tentativi di lettura. La funzione restituisce un esito negativo se la trasmissione fallisce o se non viene rilevata la conferma attesa.

L'aggiornamento del movimento è suddiviso in due fasi: prima vengono trasmessi i parametri ai singoli nodi; successivamente `sendUpgradeToEnabled()` invia `UPGRADE` all'indirizzo general call `0x00`. Dopo un'attesa di 50 ms, il master interroga separatamente gli slave abilitati per verificarne le conferme. In caso di risposte mancanti segnala il problema e ripete la scansione dei dispositivi. La separazione fra preparazione e applicazione dei parametri permette di coordinare l'aggiornamento dei segmenti senza modificarli già durante l'invio dei singoli `SET`.

**Interfaccia utente e ciclo principale** Il pulsante A permette di scorrere fra i tre movimenti sinusoidali, mentre il pulsante B seleziona le pose statiche. Un timer hardware genera un interrupt ogni 20 ms; la routine di interrupt si limita a impostare un flag, lasciando al `loop()` l'aggiornamento dello stato dei pulsanti e la gestione delle modalità. La UART viene controllata nel ciclo principale e riceve righe contenenti una coppia `frequenza,bias`, da passare alla funzione di aggiornamento del movimento. Il tempo mantenuto dal master non sostituisce il riferimento temporale locale con cui gli slave generano la sinusoide.

### 2.1.2 Slave

**Avvio e organizzazione dei task** Nel firmware slave, `app_main()` richiama `setup()`, che inizializza il debug, il LED di attività, il bus dei sensori, l'IMU BMI270, il PWM del servo e l'interfaccia $I^2C$ rivolta al master. Vengono quindi creati tre task FreeRTOS: `i2c_slave_task` per la ricezione, `control_task` per il controllo e l'acquisizione, e `alive_task` per il lampeggio del LED. La logica operativa è affidata a questi task, mentre il punto di ingresso rimane in attesa.

Il nodo svolge due ruoli distinti: è slave sul collegamento con il controllore centrale ed è master sul bus locale dei sensori. La configurazione della ricezione comprende un indirizzo individuale, l'abilitazione della general call, un buffer RX di 128 byte, un buffer TX di 4096 byte e una coda software di 16 messaggi da 128 byte.

**Ricezione ed elaborazione dei comandi** Il task di ricezione legge i byte in arrivo e ricostruisce le righe terminate da `\n`, anche quando un comando è suddiviso fra più letture. Rimuove l'eventuale carattere `\r` e inserisce ogni riga completa nella coda. Se questa è piena, elimina il messaggio più vecchio per accogliere quello nuovo. La funzione `processCommandOnce()`, richiamata dal task di controllo, estrae ed elabora i comandi disponibili.

Con `SET`, i quattro parametri vengono memorizzati nelle variabili temporanee, senza sostituire quelli attivi. Con `UPGRADE`, il nodo prepara la conferma, trasferisce i parametri temporanei nelle variabili operative e richiede l'azzeramento del riferimento temporale locale. Il comando comune serve a coordinare i nodi, ma viene elaborato nei rispettivi cicli software: non costituisce da solo

una garanzia di applicazione nello stesso istante. La risposta testuale `ACK` va inoltre distinta dal bit di riconoscimento del protocollo $I^2C$.

**Generazione della sinusoide e del PWM** La funzione `muoviSinusoidale()` calcola la posizione richiesta secondo la relazione

$$\theta_{\text{rif}}(t) = \theta_{\text{offset}} + A \sin(2\pi f t + \varphi),$$

dove $A$ è l'ampiezza in gradi, $f$ la frequenza in hertz, $\varphi$ la fase in radianti e $\theta_{\text{offset}}$ l'angolo medio. Il valore richiesto viene confrontato con quello precedente, limitando la variazione a 2 gradi per ciclo di controllo. La conversione successiva limita l'angolo all'intervallo $\pm 45^\circ$ e lo associa a un impulso compreso fra 1250 e 1750 $\mu$s.

Il driver LEDC produce il PWM a 50 Hz con risoluzione di 12 bit. La funzione `angleToMicroseconds()` converte l'angolo nella durata dell'impulso e `inviaPulse()` determina il duty cycle da applicare. Questi limiti agiscono sul comando di uscita; non equivalgono a una validazione preventiva dei quattro valori ricevuti con `SET`. Se il riferimento richiede variazioni superiori a quelle consentite, il comando effettivo risulta limitato e non riproduce esattamente la sinusoide richiesta.

**Acquisizione dei sensori e memorizzazione** Il task di controllo acquisisce tensione e corrente tramite i due INA226 e i dati del giroscopio tramite la BMI270. La lettura della corrente utilizza due canali con shunt differenti: il firmware seleziona il canale per basse correnti o quello per correnti maggiori mediante soglie di commutazione distinte, riducendo le commutazioni ripetute vicino al confine fra i due intervalli. Applica inoltre un offset di calibrazione; in caso di letture non valide le funzioni di misura possono restituire `NAN`.

Corrente, tensione, tempo e componente Z del giroscopio vengono conservati in buffer circolari da 150 campioni. Il comando `READVA` ne prepara una copia, detta snapshot, che separa i dati da trasmettere da quelli successivamente acquisiti. I comandi `READVAI`, `READVAV`, `READVAT` e `READIMU` restituiscono rispettivamente correnti, tensioni, tempi e giroscopio dello snapshot, mediante una risposta testuale delimitata da `BEGIN` ed `END`. In assenza di una copia pronta, la risposta è `NO_SNAPSHOT`. Queste sono funzionalità dello slave I2C-tritone e non implicano che il master di partenza le richiami tutte.

**Temporizzazione del controllo** Il task `control_task` è pianificato ogni 40 ms con `vTaskDelayUntil`. A ogni iterazione aggiorna il tempo, gestisce i comandi ricevuti, calcola il movimento e acquisisce i sensori. Per il calcolo temporale utilizza l'intervallo realmente trascorso, limitandolo a 0,1 s in caso di ritardi. Il task di attività fa invece lampeggiare il LED a 1 Hz.

Occorre distinguere tre temporizzazioni: il master controlla periodicamente l'interfaccia ogni 20 ms, lo slave aggiorna il riferimento di movimento ogni 40 ms e il PWM hardware ripete gli impulsi ogni 20 ms. La frequenza del bus $I^2C$ riguarda invece il trasferimento dei bit e non coincide con nessuno di questi cicli.

## 2.2 Materiali e strumenti utilizzati

### 2.2.1 Servomotori Hitec MDB961WP-CAN

Il modello Hitec MDB961WP-CAN-DroneCAN-28V è un attuatore "ultra-premium" da 20 mm, studiato specificamente per applicazioni ad alta coppia. Progettato con un motore brushless (senza spazzole) molto efficiente e un encoder magnetico, garantisce prestazioni e affidabilità senza compromessi. Supporta in modo nativo i protocolli di comunicazione su bus industriali e aeronautici, come CAN e DroneCAN (precedentemente noto come UAVCAN v0.0).




Figura 1: Servomotore Hitec MDB961WP-CAN.

La seguente scheda tecnica riassume i parametri operativi ed elettrici, inclusi i valori limite.

| Parametro | Specifica / Valore limite |
|---|---|
| Protocolli supportati | CAN (2.0A, 2.0B) / DroneCAN / UAVCAN v0.0 |
| Tensione operativa (range) | Da 8,00 V (min) a 32,00 V (max) |
| Coppia di stallo (torque) | 60,00 kg/cm (equivalenti a 833,24 oz·in) |
| Velocità a vuoto (max) | 0,14 s / 60° |
| Corrente in esercizio | A partire da 500 mA |
| Corrente di stallo (picco) | Fino a 6700 mA (max) |
| Rotazione | Programmabile tramite software digitale |
| Dimensioni | 40,0 × 20,0 × 39,0 mm (standard classe 20 mm) |
| Peso | 90,00 g |

Oltre alle caratteristiche meccaniche, un elemento rilevante dei servomotori Hitec MDB961WP-CAN è l'architettura elettronica basata su registri interni. Attraverso il bus CAN è possibile accedere ai parametri di funzionamento e alle informazioni di stato, distinguendo i registri di sola lettura da quelli modificabili. Il servomotore può quindi essere sia comandato sia interrogato dal sistema di controllo, consentendo di associare alla generazione del movimento attività di configurazione, monitoraggio e diagnostica.

**Organizzazione e rappresentazione dei dati** I registri sono organizzati in unità da 2 byte, corrispondenti a 16 bit. Nei messaggi applicativi i valori seguono la codifica little-endian: il byte meno significativo viene trasmesso per primo, seguito da quello più significativo. Il software deve rispettare questo ordine sia nella composizione dei comandi sia nella ricostruzione dei valori ricevuti.

I dati numerici non coincidono necessariamente con le grandezze fisiche espresse nelle unità di uso comune, ma richiedono i fattori di conversione previsti per ciascun registro. Per esempio, nella rappresentazione della posizione, 4096 passi corrispondono a 90°, mentre nel registro `REG_VOLTAGE` il valore 100 corrisponde a 1,00 V. La corretta interpretazione dei dati richiede quindi di considerare, oltre all'indirizzo, anche l'intervallo ammesso e la scala della grandezza rappresentata.

**Configurazione, persistenza e campi di bit** La modifica di un parametro e il suo salvataggio permanente sono operazioni distinte. Per conservare i parametri di configurazione che prevedono la memorizzazione non volatile viene utilizzato il registro `REG_CONFIG_SAVE`, all'indirizzo `0x70`. Per le impostazioni che richiedono un riavvio, il salvataggio deve essere seguito da un reset o da un ciclo di spegnimento e riaccensione affinché la nuova configurazione diventi operativa.

Alcuni registri contengono campi di bit, nei quali ogni bit rappresenta una specifica opzione o condizione. Nei registri di configurazione scrivibili, l'aggiornamento deve seguire una procedura di lettura, modifica e riscrittura: si legge il valore corrente, si interviene soltanto sui bit interessati mediante operatori bitwise e si preservano quelli riservati. Nei registri di stato di sola lettura, come `REG_EMERGENCY_STOP` (`0x48`), le maschere servono invece a riconoscere le condizioni di errore segnalate, senza modificarne il contenuto.

**Funzionalità accessibili tramite i registri** L'accesso ai registri mette a disposizione tre gruppi principali di funzionalità:

- Configurazione del movimento: impostazione dei limiti di corsa, dei parametri di position lock (OLP), delle modalità di rotazione e delle reazioni previste in presenza di anomalie.
- Configurazione della comunicazione: modifica dell'identificativo del nodo, del baud rate e del formato CAN, secondo le opzioni supportate dal dispositivo e dal firmware. La disponibilità di CAN e DroneCAN permette inoltre l'integrazione in differenti architetture di rete.
- Telemetria e diagnostica: lettura della posizione, della tensione di alimentazione, degli errori e delle temperature disponibili per lo specifico modello. Queste informazioni possono essere utilizzate dal controllore per monitorare il funzionamento e sviluppare logiche di controllo basate sullo stato misurato.

L'architettura a registri consente pertanto di adattare il comportamento dell'attuatore alle esigenze del sistema e di verificarne lo stato durante il funzionamento. Le tabelle seguenti riportano la mappa dei registri del protocollo CAN Hitec, suddivisa per categoria, indicando anche le funzionalità riservate a specifiche serie o versioni del firmware.

**Status (sola lettura)**

| Addr | Nome | R/W | Range | Unità/Note |
|---|---|---|---|---|
| 0x0C | REG_POSITION | R | 0-16383 | Posizione attuale, 4096=90° |
| 0x0E | REG_VELOCITY | R | 0-65535 | Velocità, pos/100msec |
| 0x10 | REG_TORQUE | R | 0-4095 | Duty PWM motore, 4095=100% |
| 0x12 | REG_VOLTAGE | R | 0-65535 | Tensione ingresso, 100=1.00V |
| 0x14 | REG_MCU_TEMPER | R | -57/196 | Temperatura MCU, °C |
| 0x18 | REG_TURN_COUNT | R/W | -32760/32760 | Conteggio giri, 1=360° |
| 0x1A | REG_32BITS_POSITION_L | R | 0-65535 | Posizione assoluta, word basso |
| 0x1C | REG_32BIT_POSITION_H | R | 0-65535 | Posizione assoluta, word alto |
| 0x48 | REG_EMERGENCY_STOP | R | 0-65535 | Bitmask errori (vedi sotto) |
| 0x74 | REG_PRODUCT_NO | R | 0-65535 | Numero prodotto |
| 0xC8 | REG_TIME_L | R | 0-65535 | Tempo funzionamento, sec, word basso |
| 0xCA | REG_TIME_H | R | 0-65535 | Tempo funzionamento, word alto (×65536 sec) |
| 0xFC | REG_VERSION | R | 0-65535 | Versione firmware |
| 0xFE | REG_VERSION_BIT_INVERSE | R | 0-65535 | Versione firmware (bit invertiti) |

REG_EMERGENCY_STOP bitmask: bit 8 = errore posizione min, bit 9 = errore posizione max, bit 10 = errore temp MCU sotto soglia, bit 11 = errore temp MCU sopra soglia, bit 13 = errore tensione sotto soglia, bit 14 = errore tensione sopra soglia.

### Status (solo serie SG)

| Addr | Nome | R/W | Range | Unità/Note |
|---|---|---|---|---|
| 0x16 | REG_CURRENT | R | 0-65535 | Corrente, mA |
| 0xD0 | REG_MOTOR_TEMP | R | -32767/32767 | Temp motore (richiede formula) |
| 0xD2 | REG_TEMP | R | -32767/32767 | Temp interna servo (richiede formula) |
| 0xD4 | REG_HUM | R | 0-32767 | Umidità interna (richiede formula) |

### Action (comandi movimento)

| Addr | Nome | R/W | Range | Unità/Note |
|---|---|---|---|---|
| 0x1E | REG_POSITION_NEW | R/W | 0-16383 | Nuova posizione target, 4096=90° |
| 0x24 | REG_TURN_NEW | R/W | -32760/32760 | Nuovo numero di giri (solo Turn Mode), 1=360° |

### Comunicazione

| Addr | Nome | R/W | Range | Unità/Note |
|---|---|---|---|---|
| 0x32 | REG_ID | R/W | 0-255 (0-127 se UAVCAN) | ID univoco servo, 0=broadcast |
| 0x38 | REG_CAN_BAUDRATE | R/W | 0-8 | 0=1000k, 1=800k, 2=750k, 3=500k, 4=400k, 5=250k, 6=200k, 7=150k, 8=125k |
| 0x3C | REG_CAN_BUS_ID_H | R/W | 0-8191 | Parte alta CAN Bus ID |
| 0x3E | REG_CAN_BUS_ID_L | R/W | 0-2047 (CAN 2.0A) | Parte bassa CAN Bus ID |
| 0x40 | REG_SAMPLE_POINT | R/W | 0-1 | 0=50%, 1=87.5% |
| 0x6A | REG_CAN_MODE | R/W | 0-1 | 0=CAN 2.0A, 1=CAN 2.0B |

**Mode**

| Addr | Nome | R/W | Range | Unità/Note |
|---|---|---|---|---|
| 0x44 | REG_RUN_MODE | R/W | 0-1 | 0=Multi-Turn, 1=Servo Mode |
| 0x9A | REG_POS_LOCK_TIME | R/W | 0-5000 | Tempo attivazione OLP, sec |
| 0x9C | REG_POS_LOCK_ TORQUE_RATIO | R/W | 0-100 | % coppia durante OLP |
| 0xB0 | REG_POSITION_MAX_ LIMIT | R/W | 0-16383 | Limite massimo posizione, 4096=90° |
| 0xC2 | REG_POSITION_MID | R/W | 0-16383 | Posizione centrale, default 8192 |
| 0xB2 | REG_POSITION_MIN_ LIMIT | R/W | 0-16383 | Limite minimo posizione, 4096=90° |

**Configuration**

| Addr | Nome | R/W | Range | Unità/Note |
|---|---|---|---|---|
| 0x6E | REG_DEFAULT | W | 0-65535 | 3855=ripristino fabbrica, 65535=carica salvato |
| 0x70 | REG_CONFIG_SAVE | R/W | 0-65535 | 65535=salva tutti i registri |

Per l'impiego subacqueo, i servomotori sono stati alloggiati in contenitori stagni Blue Trail Engineering della serie SER-200X, progettati per ospitare un servo e trasmetterne il movimento all'esterno mediante un albero di uscita in titanio, dotato di tenuta, mantenendo protetti l'attuatore e i collegamenti elettrici. Sebbene i servomotori siano già resistenti all'acqua, questa protezione aggiuntiva è stata adottata per l'utilizzo in profondità, dove la pressione idrostatica impone requisiti ulteriori rispetto alla sola resistenza all'acqua. Nel corso del progetto, i servomotori inseriti nei contenitori sono stati sottoposti in laboratorio a prove a una pressione equivalente a una profondità di 100 m.



Figura 2: Contenitore stagno Blue Trail Engineering SER-200X per servomotori.

### 2.2.2 M5Unified

La libreria M5Unified è un'interfaccia software unificata sviluppata da M5Stack per i dispositivi basati su ESP32. Il suo scopo principale è fornire un livello di astrazione comune e coerente, consentendo di compilare ed eseguire lo stesso codice su numerosi dispositivi della famiglia M5Stack senza dover riscrivere le procedure di accesso all'hardware specifiche di ciascuna scheda.

La libreria opera insieme a M5GFX e centralizza l'accesso alle periferiche integrate attraverso il namespace `m5`. In base alle caratteristiche del dispositivo utilizzato, permette di gestire display, pannelli touch, pulsanti fisici, altoparlanti, microfoni, alimentazione e sensori interni, tra cui accelerometri e giroscopi. Questa organizzazione semplifica la portabilità del firmware e riduce la dipendenza dalle funzioni di basso livello della singola scheda.

**Impiego nel progetto** Nell'architettura sviluppata, M5Unified è utilizzata nei firmware dei nodi slave basati su M5Stack AtomS3R per gestire l'hardware locale e, in particolare, la IMU integrata. Le operazioni principali sono le seguenti:

- **Configurazione manuale della porta $I^2C$**: la porta interna viene inizializzata esplicitamente tramite `M5.In_I2C.setPort()`, associandola ai pin dedicati `IMU_SDA = 45` e `IMU_SCL = 0`. Questa configurazione permette di comunicare correttamente con il sensore inerziale integrato.

- **Inizializzazione del sensore inerziale**: la IMU BMI270 viene avviata tramite `M5.Imu.begin()`, specificando la porta $I^2C$ interna e il modello di scheda `m5::board_t::board_M5AtomS3R`.
- **Lettura dei dati giroscopici**: nel ciclo principale eseguito sul core 1, il firmware richiama periodicamente `M5.Imu.update()` e `M5.Imu.getImuData()`. I valori di rotazione sui tre assi vengono memorizzati nelle variabili `gyro_x`, `gyro_y` e `gyro_z`, per essere successivamente trasmessi al nodo master tramite il bus CAN.

La lettura della IMU viene mantenuta sul core 1, separandola dai task CAN e di controllo vincolati al core 0. Tale scelta limita gli accessi concorrenti alle risorse gestite da M5Unified e rende più prevedibile l'esecuzione dei task temporizzati.

### 2.2.3 Atom S3R M5

L'M5Stack AtomS3R è un controllore programmabile per applicazioni IoT ed embedded caratterizzato da dimensioni compatte e da un'elevata integrazione delle periferiche. Il dispositivo utilizza il SoC ESP32-S3-PICO-1-N8R8, dotato di un processore dual-core Xtensa LX7 a 32 bit con frequenza fino a 240 MHz e connettività Wi-Fi a 2,4 GHz. Rispetto ai precedenti modelli della famiglia ATOM, integra un'antenna 3D potenziata, progettata per migliorare le prestazioni e la stabilità del collegamento radio.




Figura 3: M5Stack AtomS3R: vista inferiore e vista del display.

**Caratteristiche hardware** Il modulo misura 24,0 × 24,0 × 12,9 mm e dispone di 8 MB di memoria Flash e 8 MB di PSRAM Octal. La memoria Flash ospita il firmware e i dati persistenti, mentre la PSRAM amplia la memoria disponibile per l'esecuzione delle applicazioni.

L'interfaccia utente comprende un display IPS a colori da 0,85 pollici, con risoluzione di $128 \times 128$ pixel, e un pulsante programmabile collocato sotto il display. La sensoristica comprende un'IMU BMI270 a sei assi, che integra accelerometro e giroscopio, e un sensore geomagnetico BMM150 a tre assi. Quest'ultimo è collegato all'interfaccia $I^2C$ ausiliaria, o Sensor Hub, della BMI270, attraverso la quale vengono acquisiti anche i dati magnetici. Il dispositivo dispone inoltre di un emettitore a infrarossi con angolo di emissione dichiarato di 180° e portata fino a 12,46 m in assenza di ostacoli.

**Alimentazione e interfacce di espansione** Un circuito integrato nella scheda converte l'alimentazione da 5 V a 3,3 V. Il connettore USB Type-C permette sia l'alimentazione sia il caricamento del firmware e supporta le funzionalità USB OTG e USB Serial/JTAG. L'espansione

verso dispositivi esterni avviene tramite una porta HY2.0-4P e sei pin GPIO disponibili sul fondo del modulo, affiancati dai collegamenti di alimentazione.

**Assorbimento in modalità sleep** La corrente assorbita a riposo varia in funzione del punto di ingresso dell'alimentazione. La tabella riporta i valori dichiarati per un'alimentazione a 5 V; questi dati si riferiscono alla modalità sleep e non al funzionamento attivo del controllore.

| Ingresso di alimentazione (5 V) | Corrente in modalità sleep |
|---|---|
| Pin di alimentazione GPIO-5V | 11,63 $\mu$A |
| Porta Grove | 10,75 $\mu$A |
| USB, incluse le perdite del resistore PD | 92,50 $\mu$A |

**La famiglia ESP32 (Espressif)** ESP32 è una famiglia di System-on-Chip (SoC) prodotti da Espressif, molto diffusi nell'IoT per il rapporto prezzo/prestazioni e per l'integrazione di connettività wireless direttamente nel chip. Le varianti principali includono:

- ESP32 originale: doppio core Xtensa LX6, fino a 240 MHz, Wi-Fi b/g/n e Bluetooth Classic + BLE
- ESP32-S3: doppio core Xtensa LX7 a 240 MHz, supporto per accelerazione AI/ML (istruzioni vettoriali), più GPIO, Wi-Fi + BLE 5.0
- ESP32-C3: singolo core RISC-V a 160 MHz, più economico, Wi-Fi + Bluetooth 5 LE, con funzioni di sicurezza avanzate (secure boot RSA-3072, crittografia flash AES-128-XTS)
- ESP32-PICO-D4: versione System-in-Package che integra anche flash e oscillatore nel chip stesso, riducendo i componenti esterni necessari

Caratteristiche comuni a tutta la famiglia: memoria SRAM interna (tipicamente 400–520 KB), supporto a periferiche come SPI, $I^2C$, $I^2S$, UART, ADC/DAC, PWM e sensori capacitivi, modalità a basso consumo (deep sleep) e ampia compatibilità con Arduino IDE, ESP-IDF e PlatformIO.

**Sviluppo tramite PlatformIO** L'AtomS3R supporta diversi ambienti di sviluppo, fra cui UiFlow2, Arduino IDE, PlatformIO ed ESP-IDF. PlatformIO consente di organizzare il progetto, gestire le dipendenze e configurare compilazione e caricamento tramite il file `platformio.ini`.

**Sviluppo tramite ESP-IDF** ESP-IDF (Espressif IoT Development Framework) è il framework nativo di Espressif, utilizzabile per sviluppare applicazioni in C e C++. Consente di gestire

direttamente i task FreeRTOS, le periferiche e le modalità di risparmio energetico tramite le API del framework. Rispetto a un'applicazione basata su Arduino, permette di lavorare senza il relativo livello di astrazione, intervenendo sulla configurazione dei driver e sulle risorse hardware, come le interfacce $I^2C$ e quelle impiegate per il display. Questa possibilità è utile quando il progetto richiede un controllo dettagliato della temporizzazione e della comunicazione fra dispositivi.

### 2.2.4 AtomPoE

L'Atomic PoE Base di M5Stack è una base di espansione per la famiglia ATOM che integra la connettività Ethernet e l'alimentazione Power over Ethernet (PoE). Collegandola a uno switch o hub PoE compatibile, è possibile trasferire dati e fornire alimentazione attraverso lo stesso cavo di rete, riducendo il numero di collegamenti necessari.




Figura 4: Base di espansione M5Stack Atomic PoE Base.

La comunicazione Ethernet è gestita dal controllore W5500, che integra lo stack TCP/IP, otto socket hardware indipendenti e un'interfaccia di rete a 10/100 Mbit/s. Il collegamento con il microcontrollore avviene tramite SPI. Tra i protocolli supportati figurano TCP, UDP e IPv4, utilizzabili per lo scambio di comandi e dati attraverso una rete cablata.

La base misura 64,0 × 24,0 × 17,5 mm, pesa 17 g e presenta un intervallo di temperatura operativa dichiarato da 0 a 40°C. Queste caratteristiche ne consentono l'impiego come interfaccia compatta per applicazioni embedded che richiedono accesso Ethernet e alimentazione sul medesimo collegamento.

### 2.2.5 Adattatori MiniCAN

L'adattatore M5Stack Unit Mini CAN è un modulo compatto per il collegamento di un microcontrollore al bus CAN. Utilizza il transceiver TJA1051T/3, che realizza l'interfaccia fisica tra i segnali di trasmissione e ricezione del controllore e la linea CAN. Supporta velocità di comunicazione fino a 1 Mbit/s.




Figura 5: Adattatore M5Stack Unit Mini CAN.

Il collegamento al microcontrollore avviene tramite il connettore HY2.0-4P, che espone i segnali CAN_TX e CAN_RX, oltre all'alimentazione a 5 V e alla massa. Il modulo può essere alimentato anche attraverso il morsetto HT3.96, con una tensione compresa tra 9 e 24 V, grazie al convertitore DC–DC integrato.

Il dispositivo dispone di protezioni contro le scariche elettrostatiche e di accorgimenti per la compatibilità elettromagnetica, ma non offre isolamento galvanico. Le dimensioni sono 32,0 × 24,0 × 10,2 mm, il peso è di 8 g e l'intervallo di temperatura operativa dichiarato va da 0 a 40°C.

Durante la fase sperimentale e lo sviluppo del firmware è emerso un problema di terminazione del bus, divenuto evidente soltanto dopo il collegamento simultaneo dei tre servomotori e dei relativi nodi di controllo. In questa configurazione la comunicazione risultava instabile e numerosi

messaggi CAN venivano persi.

Una linea CAN deve essere terminata, nella configurazione convenzionale, mediante due resistenze da 120 Ω collocate alle estremità fisiche del bus; il loro collegamento in parallelo produce una resistenza equivalente di circa 60 Ω. Gli adattatori Mini CAN impiegati disponevano però ciascuno di una propria resistenza di terminazione interna. Nel sistema completo erano presenti quattro adattatori, uno per il master e tre per gli slave: mantenendo attive tutte le resistenze, il valore equivalente risultava pari a

$$R_{\text{eq}} = \frac{120\ \Omega}{4} = 30\ \Omega.$$

Questo carico eccessivo sulla linea causava la perdita continua di frame e rendeva il bus difficilmente utilizzabile. Durante le prove con tre terminazioni attive, corrispondenti a una resistenza equivalente di

$$R_{\text{eq}} = \frac{120\ \Omega}{3} = 40\ \Omega,$$

la comunicazione si è invece mantenuta stabile, pur trattandosi di un valore inferiore ai 60 Ω previsti dalla configurazione CAN nominale.

Si è quindi scelto di disattivare la resistenza interna dell'adattatore collegato al master, lasciando attive quelle presenti sui tre nodi slave. Sebbene lo standard CAN (ISO 11898-2) preveda rigorosamente due sole terminazioni da 120 Ω ai capi della linea (60 Ω), il carico per cui si è optato, a 40 Ω, non ha danneggiato l'hardware grazie alle protezioni termiche e di sovracorrente del transceiver NXP TJA1051T/3 integrato. La stabilità del segnale è stata inoltre garantita dalla brevità dei cavi sul banco. Tuttavia, trattandosi di un compromesso puramente sperimentale, nel cablaggio definitivo a bordo del veicolo verrà valutato se ripristinare la topologia standard a 60 Ω, rimuovendo fisicamente le resistenze dai nodi intermedi.

### 2.2.6 INA226-1A

L'INA226 è un circuito integrato ad alta precisione per il monitoraggio della tensione di bus, della corrente e della potenza, dotato di interfaccia $I^2C$. Viene ampiamente impiegato nei sistemi embedded per misurare il consumo energetico e analizzare il comportamento elettrico dei carichi collegati.




Figura 6: Modulo M5Stack Unit INA226-1A.

Nel progetto il modulo INA226-1A è collocato a monte del nodo master, lungo la linea di alimentazione generale del sistema. Le misure acquisite descrivono quindi lo stato della batteria che alimenta l'intera architettura, fornendo valori complessivi di tensione, corrente assorbita e potenza utilizzata.

Dalla documentazione il modulo ha un limite di misurazione di 1 A, mentre i servomotori Hitec arrivano a correnti di picco di 6,7 A ciascuno. Essendo posizionato sulla linea generale, l'attuale sensore è chiaramente sottodimensionato per reggere l'assorbimento totale dei tre attuatori (circa 20 A in stallo). In questa fase del progetto il modulo è servito esclusivamente per testare il codice e la logica di acquisizione. Come sviluppo futuro, andrà sostituito con un sensore capace di supportare correnti adeguate.

Poiché il sensore osserva l'alimentazione comune e comunica direttamente con il master tramite $I^2C$, la procedura di acquisizione dei suoi dati è indipendente dalla specifica tecnologia impiegata per la comunicazione con i segmenti della coda. La stessa struttura di lettura può pertanto essere utilizzata sia nella configurazione con coda $I^2C$ sia in quella basata su bus CAN.

### 2.2.7 Relay Unit2

M5Stack Unit 2Relay è un modulo di commutazione a due canali destinato al controllo di carichi elettrici mediante segnali logici provenienti da un microcontrollore. Integra due relè meccanici normalmente aperti, controllabili indipendentemente tramite livelli GPIO, e permette di separare elettricamente il circuito di comando dai dispositivi alimentati attraverso i contatti dei relè.



Figura 7: Modulo M5Stack Unit 2Relay.

Ogni canale supporta la commutazione di carichi fino a 250 V in corrente alternata oppure 28 V in corrente continua, con una corrente nominale massima di 8 A. Il collegamento al microcontrollore avviene tramite un'interfaccia Grove HY2.0-4P, che fornisce l'alimentazione e i due segnali logici necessari al comando dei relè. Due indicatori LED indipendenti consentono di verificarne rapidamente lo stato di attivazione.

# 3 Fondamenti teorici

## 3.1 CAN-BUS

### 3.1.1 Controller Area Network

Il Controller Area Network, noto anche come CAN-bus, è uno standard seriale per bus di campo (principalmente in ambiente automotive), di tipo multicast, introdotto negli anni ottanta dalla Robert Bosch GmbH, per collegare diverse unità di controllo elettronico (ECU). Il CAN è stato espressamente progettato per funzionare senza problemi anche in ambienti fortemente disturbati dalla presenza di onde elettromagnetiche e può utilizzare come mezzo trasmissivo una linea a differenza di potenziale bilanciata. L'immunità ai disturbi EMC può essere ulteriormente aumentata utilizzando cavi a doppino intrecciato.

Sebbene inizialmente applicata in ambito automotive, come bus per autoveicoli, attualmente è usata in molte applicazioni industriali di tipo embedded, dove è richiesto un alto livello di immunità ai disturbi. Il bit rate può raggiungere 1 Mbit/s per reti lunghe meno di 40 m. Velocità inferiori consentono di raggiungere distanze maggiori (ad es. 125 kbit/s per 500 m). Il protocollo di comunicazione del CAN è standardizzato come ISO 11898-1 (2015). Questo standard descrive principalmente lo strato di scambio dati (data link layer), composto dallo strato sottostante (sublayer) "logico" (Logical Link Control, LLC), dallo strato sottostante del Media Access Control, (MAC) e da alcuni aspetti dello strato "fisico" (physical layer) descritto dal modello ISO/OSI (ISO/OSI Reference Model). I protocolli di tutti gli altri layer sono lasciati alla libera scelta del progettista della rete.

### 3.1.2 Trasmissione dati

Il CAN trasmette dati secondo un modello basato su bit "dominanti" e "recessivi", in cui i bit dominanti sono gli 0 logici e i bit recessivi sono gli 1 logici. Se un nodo trasmette un bit dominante e un altro un bit recessivo, allora il bit "dominante" "vince" fra i due (realizzando una combinazione AND logico). Con questa tecnica, quando viene trasmesso un bit recessivo, e contemporaneamente un altro dispositivo trasmette un bit dominante, si ha una collisione, e solo il bit dominante è visibile in rete (tutte le altre collisioni sono invisibili). In pratica avviene che un bit dominante è "asserito" dalla generazione di una tensione fra i conduttori, mentre un bit recessivo è semplicemente ignorato. Si è così sicuri che ogni volta che si impone una differenza di potenziale, tutta la rete la rileva, e quindi "sa" che si tratta di un bit dominante.

Solitamente, quando usata in un bus differenziale, si applica lo schema CSMA/BA (Carrier

Sense Multiple Access/Bitwise Arbitration): se due o più dispositivi incominciano a trasmettere contemporaneamente, si applica un meccanismo di arbitraggio basato sulla priorità per decidere a quale dispositivo permettere di proseguire la trasmissione. Durante la trasmissione ogni nodo controlla lo stato del bus e confronta il bit ricevuto con il bit trasmesso. Se un bit dominante è ricevuto mentre un bit recessivo è trasmesso il nodo interrompe la trasmissione (ossia perde l'arbitrato). L'arbitrato è eseguito durante la trasmissione del pacchetto dei dati di identificazione del nodo. I nodi che incominciano contemporaneamente a trasmettere inviano un ID dominante a 0 binario, che incomincia con il bit alto. Non appena il loro ID è rappresentato da un numero più grande (quindi a priorità minore) i nodi stessi inviano un bit 1 (recessivo) e aspettano la risposta di uno 0 (dominante), quindi interrompono la trasmissione. Al termine dell'invio degli ID, tutti i nodi sono tornati allo stato di OFF, e il messaggio con la priorità corrente massima può liberamente transitare.

### 3.1.3 I frame

Tutti i frame (detti anche "messaggi") incominciano con un bit di "start-of-frame" (SOF). I frame del CAN possono essere di quattro tipi:

- Data frame: frame contenente i dati che il nodo trasmette.
- Remote frame: frame che richiede la trasmissione di un determinato identificatore.
- Error frame: frame trasmesso da un qualsiasi nodo che ha rilevato un errore.
- Overload frame: frame che introduce un ritardo fra data frame e/o remote frame.

### 3.1.4 Data frame

Sono i frame che eseguono l'effettiva trasmissione dei dati. I messaggi possono avere due formati:

- Base frame format: con 11 bit di identificazione.
- Extended frame format: con 29 bit di identificazione.

Lo standard CAN deve obbligatoriamente riconoscere il formato base frame e può opzionalmente riconoscere il formato extended frame format (che, tuttavia, deve essere tollerato). Il CAN base permette $2^{11} = 2\ 048$ tipi di messaggi diversi, ma da specifiche Bosch se ne possono usare solo 2 031. Nella versione extended si possono avere fino a $2^{29} = 536\ 870\ 912$ tipi di messaggi.

Il formato del Base frame ha la seguente struttura:

| Nome del campo | Lunghezza (numero di bit) | Funzione |
|---|---|---|
| Start-of-frame | 1 | Indica l'avvio della sequenza di trasmissione |
| Identificatore | 11 | Identificatore (unico) dei dati |
| Richiesta remota di trasmissione (RTR) | 1 | Deve essere un bit dominante |
| Bit aggiuntivo di identificazione (IDE) | 1 | Deve essere un bit dominante |
| Bit riservato (r0) | 1 | Riservato |
| Codice di lunghezza dati (DLC) | 4 | Numero di byte per codificare ciascun dato (0-8 byte) |
| Campo dati | 0-8 byte | Dati da trasmettere (la lunghezza è specificata dal campo DLC) |
| CRC | 15 | Controllo di parità a ridondanza |
| delimitatore CRC | 1 | Deve essere un bit recessivo |
| Slot ACK | 1 | Il trasmettitore invia un bit recessivo e ogni ricevitore può confermare la ricezione con un bit dominante |
| Delimitatore ACK | 1 | Deve essere un bit recessivo |
| End-of-frame (EOF) | 7 | Devono essere bit recessivi |

### 3.1.5 Data field

Nell'implementazione, il Data field corrisponde all'unica parte che il programmatore personalizza per specificare l'operazione da eseguire e gli eventuali valori da trasmettere. Nel formato CAN classico, questo campo contiene da 0 a 8 byte e costituisce il contenuto informativo, o payload, del frame. Non è invece necessario costruire manualmente, bit per bit, l'intero frame: una volta forniti al driver l'identificatore CAN, il formato del messaggio, la lunghezza e i dati, il controller CAN gestisce automaticamente l'arbitraggio, il calcolo del CRC, il bit stuffing, i delimitatori e il meccanismo di riconoscimento ACK. Occorre quindi distinguere i parametri del frame configurati dal software dai campi e dai meccanismi generati automaticamente dall'hardware. Queste funzioni appartengono principalmente al livello di collegamento dati; la conversione dei segnali logici nei segnali elettrici del bus è invece affidata al transceiver, che opera a livello fisico.

### 3.1.6 Standard ISO applicabili

- ISO 11898-2:2003: CAN ad alta velocità

- ISO 11898-3:2006: CAN fault-tolerant (a bassa velocità)
- ISO 11992-1:2003: CAN fault-tolerant per autoveicoli
- ISO 11898-5:2007: CAN alta velocità/selective wake-up
- SAE J2411: CAN single-wire CAN (SWC)

Lo standard ISO 11898-2 utilizza per i segnali una linea bilanciata a due fili. È il layer fisico più usato in applicazioni per autotrazione e controlli industriali.

Lo standard ISO 11898-4 definisce il tipo di comunicazione del CAN detto time-triggered (TTCAN), basato su un protocollo di layer fornito di un orologio di sistema per schedulare l'invio dei messaggi.

### 3.1.7 Libreria `twai.h`

TWAI, acronimo di *Two-Wire Automotive Interface*, è il driver nativo incluso nell'ESP-IDF di Espressif per la gestione hardware del protocollo CAN classico sui microcontrollori ESP32. L'inclusione dell'header `driver/twai.h` permette di configurare e utilizzare il controller CAN integrato nel microcontrollore, mentre la conversione dei segnali logici TX e RX nei livelli elettrici differenziali `CAN_H` e `CAN_L` è affidata a un transceiver esterno. Nel sistema sviluppato tale funzione è svolta dal transceiver TJA1051T/3 integrato negli adattatori M5Stack Unit Mini CAN.

La libreria viene utilizzata sia nel firmware master sia nei firmware slave, consentendo ai nodi di trasmettere comandi, ricevere configurazioni e scambiare dati di telemetria attraverso un'interfaccia a basso livello. Le principali strutture e funzioni impiegate nel progetto sono descritte di seguito.

- **Configurazione generale e installazione del driver**: la struttura `twai_general_config_t` definisce i pin di trasmissione e ricezione, la modalità operativa e i parametri delle code interne. La macro `TWAI_GENERAL_CONFIG_DEFAULT()` fornisce una configurazione iniziale, successivamente installata tramite `twai_driver_install()`. In modalità `TWAI_MODE_NORMAL` il nodo partecipa attivamente alla comunicazione e alla generazione degli ACK; la modalità `TWAI_MODE_LISTEN_ONLY` viene invece utilizzata durante le prove di ascolto passivo.
- **Configurazione della velocità**: la struttura `twai_timing_config_t` stabilisce i parametri temporali del bus. Nel progetto viene impiegata la macro `TWAI_TIMING_CONFIG_1MBITS()`, che configura il controller per una velocità di 1 Mbit/s.

- **Filtri di accettazione**: la struttura `twai_filter_config_t` determina quali identificatori vengono inseriti nella coda di ricezione. La configurazione `TWAI_FILTER_CONFIG_ACCEPT_ALL()` accetta tutti i frame, lasciando al firmware il compito di distinguere i messaggi broadcast, come il comando di sincronizzazione con identificatore `0x7FF`, dai pacchetti destinati a uno specifico nodo.

- **Avvio del controller**: dopo l'installazione, la funzione `twai_start()` porta il controller nello stato operativo e abilita la ricezione e la trasmissione dei frame sul bus.

- **Trasmissione e ricezione**: le funzioni `twai_transmit()` e `twai_receive()` operano sulla struttura `twai_message_t`. Essa contiene l'identificatore CAN, gli indicatori del formato del frame, il DLC e un campo dati di lunghezza massima pari a 8 byte nel CAN classico. Il master utilizza questa struttura per distribuire comandi e parametri di movimento, mentre gli slave la impiegano per ricevere le configurazioni e trasmettere telemetria e dati della IMU.

## 3.2 Ethernet

Ethernet è un insieme di tecnologie e protocolli standardizzati (principalmente IEEE 802.3) utilizzati per creare reti locali (LAN) cablate, permettendo a computer, server e periferiche di comunicare e scambiare dati. Ideata negli anni '70 presso lo Xerox PARC, è diventata lo standard industriale più diffuso per le connessioni fisiche grazie alla sua affidabilità, velocità e sicurezza. Il funzionamento si basa sulla trasmissione di dati in frame attraverso cavi fisici (come doppini intrecciati o fibra ottica), utilizzando indirizzi univoci detti MAC per identificare mittenti e destinatari.

In questo progetto si è reso necessario l'utilizzo di Ethernet perché il collegamento USB diretto tra il PC e il microcontrollore AtomS3R limita la distanza tra i dispositivi e vincola l'accesso al sistema alla presenza di un cavo dedicato. L'introduzione di una connessione di rete consente invece di inviare comandi e ricevere dati da postazioni collegate alla stessa rete locale, senza richiedere un collegamento USB diretto con ciascuna di esse. Il microcontrollore diventa così un dispositivo indirizzabile in rete, rendendo più flessibili il controllo e il monitoraggio del sistema; un eventuale accesso da reti esterne richiede una configurazione specifica. A tale scopo è previsto l'impiego della Atomic PoE Base W5500, che fornisce all'AtomS3R l'interfaccia Ethernet mediante il chip W5500 e consente di alimentarlo attraverso lo stesso cavo di rete grazie alla tecnologia Power over Ethernet (PoE), riducendo il cablaggio e la necessità di un'alimentazione separata.

### 3.2.1 Concetti minimi di rete: indirizzo IP

Un indirizzo IP è un identificatore logico assegnato a un'interfaccia di rete per consentire la comunicazione mediante il protocollo Internet (IP). Un dispositivo dotato di più interfacce può quindi possedere più indirizzi. IP opera al livello di rete e utilizza gli indirizzi di origine e destinazione contenuti nei pacchetti per permetterne l'instradamento, anche tra reti differenti. L'indirizzamento IP è dunque distinto dall'indirizzamento MAC impiegato per la comunicazione Ethernet sul collegamento locale.

L'indirizzo è composto da 32 bit, suddivisi in quattro ottetti, ognuno un numero compreso tra 0 e 255: `192.168.1.50`. Un aspetto fondamentale è la distinzione tra la parte dell'indirizzo che identifica la rete e quella che identifica l'interfaccia al suo interno. I primi 24 bit costituiscono il prefisso di rete e gli 8 bit rimanenti identificano l'host. La maschera corrispondente è `255.255.255.0`. L'assegnazione può essere statica, mediante configurazione manuale, oppure dinamica, attraverso un server DHCP. Nel secondo caso, il server fornisce al dispositivo l'indirizzo e altri parametri di rete; nell'allocazione dinamica, l'assegnazione ha una durata definita, detta lease, che può essere rinnovata.

### 3.2.2 MAC address

È un numero identificativo univoco della scheda di rete stessa (non dipende dalla rete a cui si è collegati). Nel codice appare come un array di byte tipo `{0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED}`. L'unica specifica è che nella rete non ci siano dispositivi con stesso MAC.

### 3.2.3 Porta (port)

Un dispositivo può far girare più servizi di rete contemporaneamente (es. un sito web e un gioco online). Per distinguerli si usa la porta: un numero da 0 a 65535 associato al servizio. Come ad esempio:

- porta 80 = servizi web (HTTP);
- porta 443 = servizi web cifrati (HTTPS);

### 3.2.4 TCP vs UDP

Sono due protocolli di trasporto, cioè due modi diversi di far viaggiare i dati sopra Ethernet/IP:

| | TCP | UDP |
|---|---|---|
| Affidabilità | garantisce che i dati arrivino, nell'ordine giusto (se serve rispedisce i pacchetti persi) | "spara e dimentica": nessuna garanzia di arrivo o ordine |
| Velocità/overhead | più lento, più controlli | più veloce, più leggero |
| Uso tipico nel progetto | il server REST (HTTP viaggia su TCP) | il bridge seriale-su-rete (dati "in tempo reale", tollerano qualche perdita) |

Nel progetto viene usato un canale TCP, sulla porta 5000, per lo scambio di comandi tra il PC e il microcontrollore. Il canale UDP invece, sulla porta 5001, serve solo all'avvio del programma per trovare l'IP dell'Atom in rete.

### 3.2.5 HTTP e REST API

HTTP è il protocollo che usa il browser per parlare con un sito web: il client (es. il PC) manda una richiesta (request) a un server, il server risponde. Le richieste hanno un "verbo":

- `GET` → "dammi delle informazioni" (non cambia nulla sul server, solo lettura);
- `POST` → "esegui un'azione / manda dei dati da elaborare" (può cambiare lo stato del server).

Una REST API è semplicemente un modo standard di organizzare queste richieste per far dialogare due programmi, scambiandosi dati in formato JSON (un formato testuale semplice per rappresentare dati strutturati, es. `{"frequenza": 0.5, "ampiezza": 40}`).

### 3.2.6 DHCP

DHCP (Dynamic Host Configuration Protocol): un server nella rete (solitamente il router) assegna automaticamente l'IP, la subnet, il gateway e i server DNS al dispositivo quando questo si collega.

### 3.2.7 Socket

Un socket è un'interfaccia attraverso la quale un'applicazione invia e riceve dati sulla rete. Nel caso di TCP, il server rimane in ascolto su una porta e il client richiede una connessione al suo indirizzo IP. Dopo l'instaurazione della connessione, i due estremi possono scambiarsi un flusso bidirezionale di byte, del quale TCP gestisce il trasporto affidabile e ordinato.

In questo progetto, l'espressione "socket grezzo" indica l'utilizzo di una normale connessione TCP con un protocollo applicativo testuale definito nel firmware, senza adottare HTTP o REST. Non va quindi confusa con i raw socket di basso livello, che consentono di accedere direttamente ai pacchetti dei protocolli sottostanti. Il termine "grezzo" si riferisce qui al formato dei dati scambiati, non all'assenza delle funzioni di TCP. Il PC può collegarsi mediante un terminale di rete o uno script e utilizzare gli stessi comandi testuali già previsti per la seriale USB.

I delimitatori `<<<END>>>` implementati sono necessari perché TCP trasporta un flusso di byte e non conserva i confini dei messaggi applicativi: una riga può essere ricevuta in più letture, oppure una lettura può contenere più righe. È quindi il firmware a riconoscere i comandi completi e il client a individuare la conclusione delle risposte.

## 3.3 I2c

Il protocollo $I^2C$ (Inter-Integrated Circuit) è un sistema di comunicazione seriale sincrono che permette a più circuiti di condividere lo stesso bus. Nel progetto costituisce il collegamento della configurazione originaria tra il microcontrollore master e i nodi di controllo dei tre segmenti della coda.

### 3.3.1 Struttura del bus e segnali

La comunicazione utilizza due linee: SDA (Serial Data), che trasporta i dati, e SCL (Serial Clock), che fornisce il riferimento temporale per la trasmissione dei bit. Il master avvia gli scambi e genera il clock; i dispositivi slave, identificati da un indirizzo, rispondono quando vengono selezionati. Anche durante una lettura, nella quale lo slave trasmette i dati, la comunicazione è avviata dal master.

Le linee impiegano uscite di tipo open-drain e resistenze di pull-up: ciascun dispositivo può portare il segnale a livello basso oppure rilasciarlo, mentre il livello alto è ottenuto tramite il pull-up. I dispositivi condividono un riferimento di massa. Capacità del bus e resistenze influenzano i tempi di salita dei segnali e limitano le condizioni di funzionamento del collegamento.

### 3.3.2 Indirizzamento e trasmissione dei dati

Una transazione inizia con la condizione START, generata portando SDA da alto a basso mentre SCL è alto. Nel formato a 7 bit, il master invia l'indirizzo del destinatario seguito dal bit di direzione: 0 per la scrittura e 1 per la lettura. I dati sono poi trasferiti in byte di otto bit, partendo dal bit più significativo.

Dopo ogni byte, il nono impulso di clock è riservato al riconoscimento: il ricevitore porta SDA a livello basso per inviare ACK oppure la lascia alta per NACK. Al termine di una lettura, il master segnala con NACK che non richiede altri byte.

# 4 Implementazione del sistema

## 4.1 Implementazione della trasmissione CAN

La prima parte del lavoro si è occupata dello studio e della decodifica dei messaggi di comunicazione e dei registri dei servomotori Hitec MDB961WP-CAN. A tale scopo è stato utilizzato il software messo a disposizione dalla casa produttrice nella sezione Software & Firmware del sito ufficiale: `https://hitecrcd.com/software-firmware/`. La figura 8 riporta la finestra di test utilizzata durante questa fase del lavoro.

Per predisporre il test, il servomotore è stato collegato al bus CAN mediante l'apposito adattatore e alimentato a 12 V. Alla stessa linea è stato connesso un M5Stack Atom S3R, collegato tramite USB al PC per visualizzare i dati ricevuti sul monitor seriale di Visual Studio Code.

Parallelamente, è stato aperto il software Hitec in una finestra separata, selezionando il protocollo CAN 2.0A e la modalità Servo Mode. Tramite i pulsanti `Center`, `Left` e `Right` sono stati comandati gli spostamenti del servomotore verso le posizioni centrale, sinistra e destra. In corrispondenza di ciascun comando, sul monitor seriale sono stati osservati i messaggi in transito sul bus, così da mettere in relazione il movimento richiesto con i frame contenenti la posizione da raggiungere.

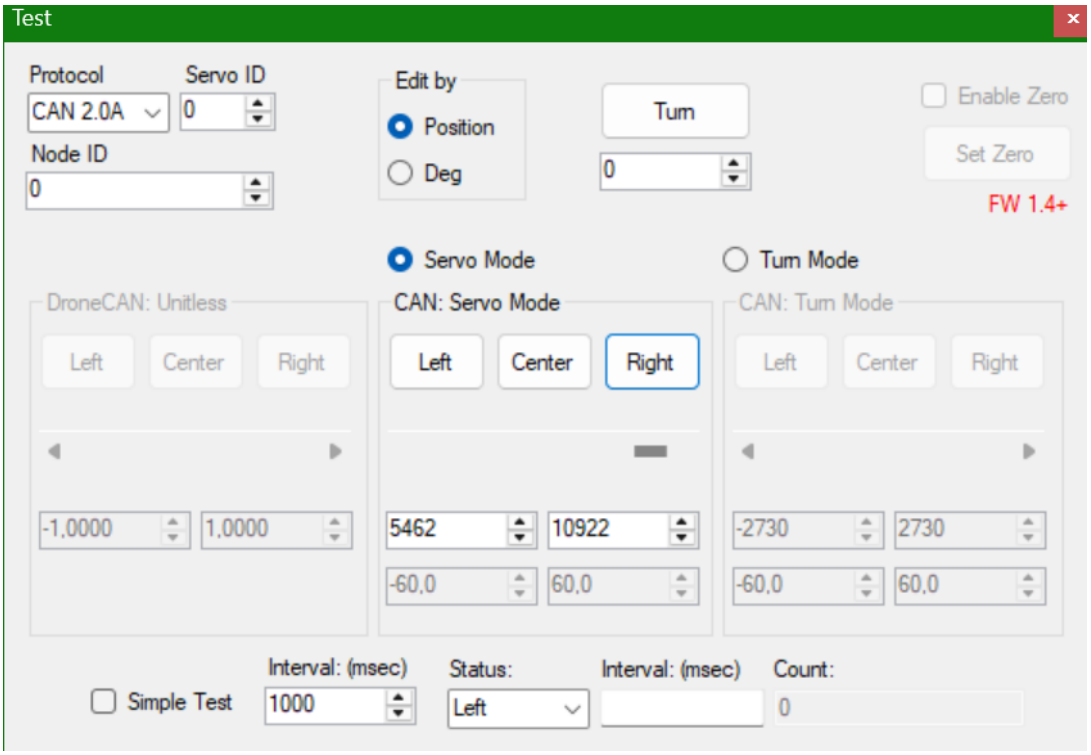



Figura 8: Finestra di test del software Hitec utilizzato per le prove sui servomotori.

Il codice eseguito sull'Atom svolge la funzione di sniffer passivo: attraverso le funzioni di ricezione del driver CAN, rese disponibili dall'header `driver/twai.h`, acquisisce i frame in transito verso il servomotore e le eventuali risposte, riportandoli sul monitor seriale. In questa configurazione l'Atom è utilizzato per osservare la comunicazione, senza generare i comandi di movimento, che vengono inviati dal software Hitec.

Di seguito sono riportati due estratti del codice dello sniffer.

Configurazione iniziale:

```
void app_main(void)
{
    ESP_LOGI(TAG, "=== CAN SNIFFER (listen-only) - avvio ===");

    twai_general_config_t g_config = TWAI_GENERAL_CONFIG_DEFAULT(CAN_TX_PIN, CAN_RX_PIN,
    ↪  TWAI_MODE_LISTEN_ONLY);
    twai_timing_config_t t_config = TWAI_TIMING_CONFIG_1MBITS();
    twai_filter_config_t f_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();
```

Estratto del ciclo di ricezione e stampa dei frame:

```
while (1) {
        twai_message_t message;

        // Attende un frame fino a 2 secondi
        esp_err_t result = twai_receive(&message, pdMS_TO_TICKS(2000));

        if (result == ESP_OK) {
            char data_str[64] = {0};
            int offset = 0;
            for (int i = 0; i < message.data_length_code; i++) {
                offset += snprintf(data_str + offset, sizeof(data_str) - offset,"%02X ",
                ↪  message.data[i]);
            }

            printf("ID:0x%03lX %s  DLC:%d  Dati: %s\n", (unsigned long)message.identifier,
            ↪  message.extd ? "(ext)" : "(std)", message.data_length_code, data_str);
```

Le prove di comunicazione hanno permesso di delineare il formato dei messaggi di richiesta e di risposta utilizzati nello scambio di dati. Le sequenze riportate di seguito rappresentano esclusivamente il contenuto del Data field, non l'intero frame CAN. L'header applicativo, l'ID del servo, l'indirizzo del registro e il checksum Hitec sono quindi tutti contenuti nel campo dati: in particolare, l'ID del servo non coincide con l'identificatore del frame CAN e il checksum applicativo è distinto dal CRC gestito dal controller.

### 4.1.1 Messaggio di richiesta

Il Data field del messaggio di richiesta era costituito dalla seguente sequenza di cinque byte:

[0x96, 0x01, 0x0C, 0x00, 0x7B]

| Byte | Valore | Nome | Significato |
|---|---|---|---|
| 1° | 0x96 | Header di Richiesta | Identifica l'inizio di un messaggio di richiesta del protocollo applicativo Hitec. |
| 2° | 0x01 | ID del Servo | Identifica il servo destinatario della richiesta in questo caso il 1°. |
| 3° | 0x0C | Indirizzo Registro | Specifica il registro da leggere: 0x0C (12 in decimale) corrisponde a REG_POSITION. |
| 4° | 0x00 | Lunghezza Dati | Il valore 0x00 indica una richiesta di lettura, senza dati da scrivere nel registro. Non rappresenta il DLC del frame CAN. |
| 5° | 0x7B | Checksum | Checksum applicativo della richiesta, calcolato come $0x88 - 0x01 - 0x0C - 0x00 = 0x7B$. È distinto dal CRC del frame CAN. |

### 4.1.2 Messaggio di risposta

Alla richiesta di lettura, il servo ha risposto con un frame CAN il cui Data field conteneva la seguente sequenza di sette byte:

[0x69, 0x01, 0x0C, 0x02, 0xF4, 0x1F, 0x22]

| Byte | Valore | Nome | Significato |
|---|---|---|---|
| 1° | 0x69 | Header di Risposta | Identifica l'inizio di un messaggio di risposta nel formato Hitec utilizzato. |
| 2° | 0x01 | ID del Servo | Identifica il servo che ha generato la risposta, in questo caso il 1°. |
| 3° | 0x0C | Indirizzo Registro | Identifica il registro a cui si riferiscono i dati restituiti: REG_POSITION. |
| 4° | 0x02 | Lunghezza Dati | Indica i due byte del valore restituito dal registro. Non rappresenta il DLC, che per questo frame vale 7. |
| 5° | 0xF4 | Dato LOW | Byte meno significativo del valore di posizione restituito. |
| 6° | 0x1F | Dato HIGH | Byte più significativo del valore di posizione restituito. |

| Byte | Valore | Nome | Significato |
|---|---|---|---|
| 7° | 0x22 | Checksum Risposta | Checksum applicativo contenuto nella risposta, distinto dal CRC del frame CAN. |

Il passo successivo ha riguardato la modifica del codice, passando dall'ascolto passivo alla trasmissione dei pacchetti di byte acquisiti in precedenza. Inviando nuovamente i comandi corrispondenti alle posizioni centrale, sinistra e destra, è stato riscontrato il movimento desiderato del servomotore.

Di seguito sono riportati gli estratti relativi alla preparazione del frame e alle sequenze di byte utilizzate.

```
void invia_frame_hitec(uint8_t *dati)
{
    twai_message_t messaggio_tx;
    messaggio_tx.identifier = 0x000;    // ID REALE scoperto dallo sniffer
    messaggio_tx.extd = 0;              // Standard 11-bit
    messaggio_tx.rtr = 0;
    messaggio_tx.data_length_code = 7;  // DLC REALE scoperto dallo sniffer


    // Copia i 7 byte nel pacchetto
    for (int i = 0; i < 7; i++) {
        messaggio_tx.data[i] = dati[i];
    }
```

```
uint8_t cmd_centro[7]   = {0x96, 0x00, 0x1E, 0x02, 0x00, 0x20, 0x40};
uint8_t cmd_sinistra[7] = {0x96, 0x00, 0x1E, 0x02, 0x56, 0x15, 0x8B};
uint8_t cmd_destra[7]   = {0x96, 0x00, 0x1E, 0x02, 0xAA, 0x2A, 0xF4};
```

## 4.2 Divisione master/slave multi-core

L'architettura sviluppata mantiene una struttura concettualmente simile a quella del precedente sistema basato su comunicazione $I^2C$: anche nella nuova implementazione l'intero sistema può essere suddiviso in un nodo master e in più nodi slave. La differenza principale riguarda il mezzo di comunicazione e la maggiore autonomia assegnata ai controllori locali. La distribuzione dei compiti tra dispositivi, affiancata all'esecuzione concorrente di task sui processori multi-core ESP32-S3, permette di separare l'interazione con l'operatore, il calcolo delle traiettorie e la raccolta della telemetria.

La rete è costituita complessivamente da quattro nodi:

- **Nodo master (AtomS3R/Ext)**: gestisce l'interfaccia seriale non bloccante mediante una macchina a stati finiti, acquisisce dall'operatore i parametri fisici dell'onda, quali ampiezza, frequenza, offset e sfasamento, e li distribuisce ai nodi slave. Il master riceve inoltre i dati di ritorno e li memorizza in buffer circolari FIFO allocati in RAM, rendendoli disponibili per il successivo scaricamento e l'analisi.
- **Nodi slave 1, 2 e 3 (AtomS3R)**: ciascun nodo è associato localmente a un servomotore Hitec. Ogni slave calcola il riferimento di posizione con un periodo di controllo di 20 ms, applica le rampe necessarie per rendere graduali le variazioni di movimento e controlla lo stato energetico del motore. Parallelamente, acquisisce i dati giroscopici dalla propria IMU e li trasmette al master a intervalli regolari di 37 ms.

In questa configurazione il master svolge quindi le funzioni di coordinamento, configurazione e raccolta dei dati, mentre gli slave eseguono localmente il controllo temporale dei rispettivi attuatori. Tale suddivisione riduce il carico computazionale del nodo centrale e consente ai tre segmenti della coda di operare in modo coordinato senza rinunciare al controllo locale in tempo reale.

### 4.2.1 Master

Nel nodo master, `setup()` e `loop()` vengono eseguiti, nella configurazione predefinita del framework Arduino per ESP32-S3, sul core 1. Durante la fase di inizializzazione vengono inoltre creati i task dedicati alla gestione dell'interfaccia seriale, alla ricezione dello stream e, quando l'hardware Ethernet è disponibile, alla comunicazione di rete.

```
stampa_menu(Serial);

BaseType_t risultato_task_seriale = xTaskCreate(
    task_seriale, "seriale", 4096, NULL,
    tskIDLE_PRIORITY + 2, NULL);

BaseType_t risultato_task_stream = xTaskCreate(
    task_stream, "stream", 4096, NULL,
    tskIDLE_PRIORITY + 2, NULL);

if (risultato_task_seriale != pdPASS ||
    risultato_task_stream != pdPASS) {
    Serial.println("[ERRORE] Creazione task seriale/stream fallita.");
    while (true)
        delay(1000);
}
```

```
if (ethernet_avviata) {
    // Il task Ethernet viene creato soltanto se
    // l'inizializzazione del W5500 e' riuscita.
    BaseType_t risultato_task_ethernet = xTaskCreate(task_ethernet, "ethernet",
    ↪  6144,NULL,tskIDLE_PRIORITY + 2, NULL);

    if (risultato_task_ethernet != pdPASS) {
        Serial.println("[ETH ERRORE] Creazione task Ethernet fallita.");
    }
}
else {
    Serial.println("[ETH] Ethernet non avviata. Il controllo USB rimane disponibile.");
}
```

I due task principali, `task_seriale` e `task_stream`, sono creati con la stessa priorità, pari a `tskIDLE_PRIORITY + 2`. Essi possono quindi avanzare in modo concorrente, lasciando allo scheduler FreeRTOS la ripartizione del tempo di esecuzione. Anche `task_ethernet`, quando viene avviato, utilizza la stessa priorità e non impedisce il funzionamento tramite USB in caso di mancata inizializzazione del W5500.

Poiché il codice utilizza `xTaskCreate()` e non `xTaskCreatePinnedToCore()`, i task aggiuntivi non sono vincolati esplicitamente al core 1: la loro assegnazione ai core disponibili è affidata allo scheduler.

**Task seriale e gestione event-driven dei comandi** Il task `task_seriale` coincide funzionalmente con l'esecuzione periodica di `gestisci_seriale()`. Il ciclo introduce una pausa di 10 ms tra due interrogazioni successive della porta seriale:

```
void task_seriale(void* arg) {
    // Task che coincide con la funzione gestisci_seriale().
    (void)arg;
    while (true) {
        gestisci_seriale();
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
```

L'intervallo di 10 ms non costituisce un vincolo temporale rigido, poiché il task deve reagire all'input dell'operatore e non generare comandi a frequenza costante. A ogni iterazione viene verificata la presenza di dati mediante `Serial.available()`; se sono disponibili nuovi caratteri, la funzione aggiorna la macchina a stati finiti del menu. Soltanto quando un comando risulta completo viene eseguita l'azione associata ed eventualmente trasmesso un frame CAN. La

comunicazione prodotta da questo task è quindi puramente event-driven e non prevede invii ciclici.

```
static void stampa_menu(void) {
    Serial.printf("\n--- MENU PARAMETRI MASTER (CODA DISTRIBUITA) ---\n");
    Serial.printf("0 = HOME: Comando rientro coordinato locale sugli Slave\n");
    Serial.printf("1 = SIN: Invia parametri correnti a tutti gli Slave\n");
    Serial.printf("2 = SINISTRA: Sposta offset coda di -5°\n");
    Serial.printf("3 = DESTRA: Sposta offset coda di +5°\n");
    Serial.printf("4 = SYNC: Avvia i task sinusoide in simultanea (Broadcast 0x7FF)\n");
    Serial.printf("5 = DUMP IMU: Snapshot simultaneo dei buffer IMU degli Slave (CSV)\n");
    Serial.printf("6 = DUMP BUFFER: Snapshot simultaneo dei buffer stream servo\n");
    Serial.printf("7 = MOTOR FREE: Sgancia coppia (Task slave in riposo a consumo zero)\n");
    Serial.printf("8 = MOTOR LOCK: Richiesta posizione On-Demand + rientro guidato\n");
    Serial.printf("9 -> poi 1: Entra in MODIFICA parametri -> Scegli Servo\n");
    Serial.printf("9 -> poi 0: Esegue l'UPGRADE inviando i nuovi parametri via CAN\n");
    Serial.printf("-----------------------------------------------------------\n");
}
```

| Comando | Modifica dello stato | Azione eseguita |
|---|---|---|
| 0 | `home_in_corso = true`; stato motore normale | Richiama `invia_a_tutti_gli_slave()` per coordinare il rientro in posizione Home. |
| 1 | `onda_attiva = true`; `home_in_corso = false` | Trasmette a tutti gli slave i parametri correnti della sinusoide. |
| 2 / 3 | Aggiorna `offset_manuale_gradi` di $-5$ o $+5$ gradi | Invia la nuova configurazione ai tre slave. |
| 4 | Nessuna modifica locale permanente | Trasmette un frame di sincronizzazione con ID `0x7FF` e DLC pari a 0. |
| 6 | Congela temporaneamente il buffer dello stream | Stampa il contenuto acquisito e successivamente riabilita l'aggiornamento del buffer. |
| 7 | Stato motore libero; onda disabilitata | Invia agli slave il comando di rilascio della coppia. |
| 8 | Stato motore in rientro; Home abilitato | Invia agli slave la richiesta di blocco e rientro guidato. |
| 9, poi 0 | Copia i parametri dal buffer di modifica a quelli attivi | Esegue l'upgrade della configurazione e la trasmette a tutti gli slave. |

**Task stream** Il task `task_stream` utilizza una ricezione CAN bloccante con un timeout di 100 ms. La funzione `twai_receive()` sospende quindi l'esecuzione fino all'arrivo di un frame oppure fino alla scadenza del timeout; in assenza di messaggi, il task riprende il ciclo e avvia una

nuova attesa. Il funzionamento è pertanto guidato dagli eventi presenti sul bus e non da una trasmissione periodica generata dal master.

Quando viene ricevuto un frame, il task distingue due tipologie di dati. I messaggi dello stream del servomotore sono riconosciuti dal primo byte pari a `0x56`: il secondo byte deve contenere un identificativo compreso tra 1 e 3. Se non è in corso un'operazione di dump, il frame viene salvato nel buffer circolare associato al servo mediante la funzione `salva_in_buffer_circolare()`, che gestisce la scrittura protetta. Durante il dump, invece, i nuovi campioni vengono scartati per mantenere invariato il contenuto dei buffer durante la stampa.

Il task gestisce inoltre i pacchetti IMU provenienti dagli slave, identificati dal byte iniziale `0x55` e dagli identificatori CAN compresi tra `0x301` e `0x303`. Anche questi dati vengono memorizzati soltanto quando il dump non è attivo.

```
void task_stream(void* arg) {
    (void)arg;
    // Nuovo messaggio disponibile nel buffer TWAI, senza periodicita' fissa.
    twai_message_t messaggio_rx;

    while (true) {
        if (twai_receive(&messaggio_rx, pdMS_TO_TICKS(100)) == ESP_OK) {
            // Stream servo standard: primo byte 0x56.
            if (messaggio_rx.data_length_code >= 2 &&
                messaggio_rx.data[0] == 0x56) {
                uint8_t id_servo = messaggio_rx.data[1];
                if (id_servo >= 1 && id_servo <= 3 && !dump_in_corso) {
                    salva_in_buffer_circolare(id_servo - 1, &messaggio_rx);
                }
            }
            // Pacchetti IMU: primo byte 0x55, CAN ID 0x301--0x303.
            else if (messaggio_rx.identifier >= 0x301 &&
                     messaggio_rx.identifier <= 0x303 &&
                     messaggio_rx.data_length_code >= 8 &&
                     messaggio_rx.data[0] == 0x55) {
                uint8_t id_slave = messaggio_rx.identifier - 0x300;
                if (!dump_in_corso)
                    salva_in_buffer_imu(id_slave - 1, &messaggio_rx);
            }
        }
    }
}
```

### 4.2.2 Slave

Rispetto al nodo master, negli slave è necessario controllare esplicitamente l'assegnazione dei task ai due core, così da separare le operazioni CAN, caratterizzate da vincoli temporali più stringenti, dalla gestione della IMU interna tramite la libreria M5Unified. Nella configurazione adottata, i principali task di ricezione, controllo del servomotore e trasmissione CAN sono vincolati al core 0 mediante `xTaskCreatePinnedToCore()`. Le funzioni `setup()` e `loop()`, eseguite dal framework Arduino sul core 1, rimangono invece dedicate all'inizializzazione e all'aggiornamento periodico dei dati della IMU.

Questa separazione evita che il carico prodotto dalla comunicazione CAN interferisca con l'accesso alla IMU. I valori del giroscopio vengono aggiornati nel `loop()` ogni 37 ms e successivamente letti dal task `task_trasmissione_can_imu`, incaricato di inviarli al master.

```
void setup() {
    Serial.begin(115200);
    delay(500);
    Serial.println();
    Serial.println("\n--- SYSTEM BOOT (ARCHITETTURA EQUILIBRATA) ---");

    // Configurazione dell'I2C interno utilizzato dalla IMU sul core 1.
    M5.In_I2C.setPort(I2C_NUM_0, IMU_SDA, IMU_SCL);
    bool imuOk = M5.Imu.begin(&M5.In_I2C, m5::board_t::board_M5AtomS3R);

    if (!imuOk) {
        Serial.println("ERRORE: IMU BMI270 non trovata.");
        while (true) {
            delay(1000);
        }
    }
    Serial.println("IMU inizializzata correttamente.");

    mutex_stato = xSemaphoreCreateMutex();

    // I task CAN e di controllo sono vincolati al core 0.
    xTaskCreatePinnedToCore(task_ricezione_can, "rx_can", 4096, NULL, tskIDLE_PRIORITY + 1,
    ↪  NULL, 0);
    xTaskCreatePinnedToCore(task_sinusoide, "sinusoide", 4096, NULL, tskIDLE_PRIORITY + 2,
    ↪  NULL, 0);
    xTaskCreatePinnedToCore(task_trasmissione_can_imu, "tx_imu", 4096, NULL, tskIDLE_PRIORITY +
    ↪  2, NULL, 0);
}

void loop() {
    delay(37);
```

```
    if (M5.Imu.update()) {
        const auto data = M5.Imu.getImuData();

        // I valori vengono letti dal task di trasmissione CAN della IMU.
        gyro_x = data.gyro.x;
        gyro_y = data.gyro.y;
        gyro_z = data.gyro.z;

        Serial.printf("X:%.2f Y:%.2f Z:%.2f\n",
                      gyro_x, gyro_y, gyro_z);
    }
    else {
        Serial.println("IMU non pronta...");
    }
}
```

**Task di ricezione CAN** Il task `task_ricezione_can`, assegnato al core 0 con priorità `tskIDLE_PRIORITY + 1`, esegue una chiamata bloccante a `twai_receive()` con timeout di 50 ms. Quando non sono disponibili messaggi, il task rimane sospeso e non occupa inutilmente il processore. La sua priorità è inferiore a quella del task sinusoidale; se entrambi sono pronti per l'esecuzione, lo scheduler assegna quindi la CPU a `task_sinusoide`.

Il task gestisce due categorie principali di frame:

- i frame con identificatore `0x7FF`, utilizzati come comando di sincronizzazione broadcast, impostano la variabile `start_sync` e consentono l'avvio coordinato del task sinusoidale;
- i frame con identificatore `0x100 + SLAVE_ID` contengono la configurazione destinata allo specifico nodo e aggiornano i parametri condivisi all'interno di una sezione protetta dal mutex `mutex_stato`.

**Task sinusoidale** Il task `task_sinusoide` è eseguito sul core 0 con priorità `tskIDLE_PRIORITY + 2`. Il ciclo di controllo utilizza `vTaskDelayUntil()` con periodo di 20 ms: a differenza di un ritardo calcolato a partire dalla fine di ogni iterazione, questo metodo mantiene un riferimento temporale assoluto e impedisce l'accumulo progressivo del ritardo dovuto al tempo di calcolo interno.

Ai fini delle prove sperimentali e del monitoraggio del movimento, poiché il firmware slave comprende anche la lettura dell'unità di misura inerziale (IMU), il dispositivo AtomS3R è stato smontato e saldato all'interno del contenitore del servomotore. In questa posizione, il sensore integrato si muove solidalmente con il segmento azionato dal servo e ne rileva direttamente l'oscillazione.

## 4.3 Generazione del movimento sinusoidale

Il `task_sinusoide()` ospita tutta la parte relativa alla traiettoria del movimento sinusoidale, passaggio fondamentale nello sviluppo del progetto per riprodurre l'oscillazione dei segmenti della coda. La scelta dei servomotori Hitec MDB961WP-CAN e della comunicazione CAN è stata motivata in particolare dalla disponibilità di registri interni accessibili e dalla possibilità di intervenire direttamente sui parametri di funzionamento attraverso i messaggi di comando.

Queste caratteristiche, tuttavia, non consentono di affidare al servomotore la generazione autonoma di una sinusoide: nella configurazione utilizzata, le funzionalità disponibili permettono spostamenti fra posizioni assegnate, con tratti a velocità uniforme oppure con rampe di velocità, corrispondenti a fasi ad accelerazione costante. Per ottenere un movimento sinusoidale è quindi necessario aggiornare nel tempo la posizione richiesta, anziché trasmettere un solo comando che definisca l'intera oscillazione.

La soluzione adottata prevede un microcontrollore dedicato a ciascun servomotore. Questi dispositivi sono slave rispetto al microcontrollore master collegato al PC, dal quale ricevono le indicazioni di movimento, ma svolgono a loro volta il ruolo di controllori dei rispettivi servomotori. La distinzione master–slave descrive quindi la gerarchia funzionale del sistema. Ogni nodo calcola in tempo reale una successione di posizioni desiderate lungo la sinusoide e le invia al servo mediante singoli comandi CAN. Il movimento sinusoidale viene così costruito attraverso l'aggiornamento progressivo del riferimento di posizione, mentre al servomotore è affidata l'esecuzione degli spostamenti richiesti.

Il calcolo della posizione istantanea avviene tramite la seguente equazione:

$$\text{posizione_target} = \text{centro_attuale} + \text{ampiezza_posizione} \times \sin(\text{angolo_radianti}).$$

Per far sì che l'onda avanzi nel tempo, l'angolo (la fase) viene incrementato a ogni ciclo:

$$\text{angolo_radianti} = \text{angolo_radianti} + 2\pi \times \text{frequenza_hz} \times \Delta t,$$

dove $\Delta t$ è il tempo trascorso, calcolato come `LOOP_PERIODO_MS / 1000.0f`.

### 4.3.1 Analisi delle variabili: fase, frequenza, ampiezza

- `frequenza_hz`: determina la velocità dell'oscillazione in cicli al secondo.
- `angolo_radianti`: rappresenta la fase corrente della sinusoide. Parte da 0 e cresce fino a $2\pi$, per poi sottrarre $2\pi$ e ripetere il ciclo, garantendo un'onda continua e senza salti di fase.
- `ampiezza_impostata`: è l'ampiezza bersaglio desiderata dall'utente, convertita da gradi a passi macchina tramite la costante di calibrazione `GRADI_A_PASSI`, pari a 16384/360.

- `ampiezza_posizione`: è l'ampiezza reale applicata nell'istante corrente. Se differisce da `ampiezza_impostata`, il programma la adegua gradualmente aggiungendo o sottraendo `step_ampiezza`. Questo meccanismo evita movimenti violenti o assorbimenti anomali di corrente quando si cambia l'ampiezza di colpo da menu.

- `step_ampiezza`: è il massimo incremento di ampiezza consentito per ogni ciclo macchina (20 ms), calcolato in base al limite di gradi al secondo `VELOCITA_AMPIEZZA_GRADI_SEC`.

```
// 1. RAMPA AMPIEZZA
float target_amp_reale =
    (l_onda_attiva && !l_home_in_corso && !l_motore_libero)
        ? l_ampiezza_target
        : 0.0f;
float diff_amp = target_amp_reale - ampiezza_attuale_passi;
if (fabsf(diff_amp) > 0.01f) {
    if (diff_amp > 0.0f)
        ampiezza_attuale_passi += step_ampiezza;
    else
        ampiezza_attuale_passi -= step_ampiezza;
    if (fabsf(target_amp_reale - ampiezza_attuale_passi) <
        step_ampiezza)
        ampiezza_attuale_passi = target_amp_reale;
}
```

### 4.3.2 Variabili di centro e offset (posizionamento spaziale)

- `POSIZIONE_DEFAULT` (8192): è il centro geometrico assoluto del servocomando, corrispondente a metà corsa sulla risoluzione di 16384 passi del controller Hitec.

- `offset_menu_gradi`, `offset_manuale_gradi`, `offset_centro_gradi`: permettono di traslare il punto di lavoro della sinusoide rispetto al centro geometrico. L'offset totale definisce il nuovo zero virtuale sul quale oscilla l'onda.

- `centro_target`: è la posizione statica di riposo desiderata in quel momento, calcolata come 8192 più l'offset totale espresso in passi.

- `centro_attuale`: come per l'ampiezza, è il centro operativo in tempo reale. Insegue `centro_target` gradualmente tramite `step_centro`, per garantire transizioni fluide quando si sposta l'asse di rotazione mediante i comandi, ad esempio quelli di spostamento a destra e a sinistra.

```
// 2. RAMPA CENTRO (Offset)
float diff_centro = l_centro_target - centro_attuale_passi;
if (fabsf(diff_centro) > 0.01f) {
```

```
    if (diff_centro > 0.0f)
        centro_attuale_passi += step_centro;
    else
        centro_attuale_passi -= step_centro;
    if (fabsf(l_centro_target - centro_attuale_passi) < step_centro)
        centro_attuale_passi = l_centro_target;
}
```

### 4.3.3 Variabili di stato

Le variabili descritte di seguito appartengono al nodo master e vengono trasmesse agli slave insieme ai parametri di configurazione. Ciascuno slave aggiorna di conseguenza le corrispondenti variabili di stato locali, indicate nel codice con i nomi `l_onda_attiva` e `l_home_in_corso`, utilizzate dal task sinusoidale per determinare il comportamento del servomotore.

- `onda_attiva`: variabile booleana che abilita o disabilita il calcolo del seno e l'avanzamento della fase.
- `home_in_corso`: variabile booleana che forza l'azzeramento graduale di `ampiezza_posizione`. Quando l'ampiezza arriva a zero, e quindi lo smorzamento dell'onda è concluso, il sistema attende che anche `centro_attuale` coincida con il centro assoluto 8192 per completare il rientro e fermare ogni movimento.

```
// Costruisco il messaggio.
twai_message_t msg = {};
msg.identifier = 0x100 + id;
msg.extd = 0;
msg.rtr = 0;
msg.data_length_code = 8;

msg.data[0] =
    (onda_attiva ? 0x01 : 0x00) |
    (home_in_corso ? 0x02 : 0x00) |
    (stato_motore == MOTORE_LIBERO ? 0x04 : 0x00) |
    (stato_motore == MOTORE_RIENTRO ? 0x08 : 0x00);
msg.data[1] = (uint8_t)(f * 100.0f);
...
```

Nella versione CAN, rispetto alla precedente I2C, la codifica della frequenza avviene su un singolo byte moltiplicando il valore per 100. Questo introduce un limite massimo rappresentabile di 2,55 Hz. Tale limitazione è stata considerata accettabile, in quanto frequenze superiori risulterebbero meccanicamente critiche a causa dell'elevata resistenza idrodinamica durante il nuoto, inoltre in fase di sperimentazione si è comunque deciso di usare un clamping di sicurezza a 1 Hz.

**La funzione di invio della posizione** La funzione `invia_posizione_autonoma` traduce il valore numerico calcolato nel `loop()` in un pacchetto di rete conforme al protocollo CAN 2.0A specifico per i servocomandi Hitec (Normal Packet Format).

**Configurazione del frame CAN** Viene istanziato un pacchetto `twai_message_t` con identificativo CAN base `0x000`, in formato standard a 11 bit, e una lunghezza dati (DLC) di 7 byte.

### 4.3.4 Struttura del payload (7 byte)

- `data[0] = 0x96`: header fisso di inizio frame richiesto dall'elettronica Hitec.
- `data[1] = 0x00`: ID del nodo. Il valore `0x00` funge da broadcast, comandando simultaneamente qualsiasi servo connesso al bus.
- `data[2] = 0x1E`: indirizzo del registro. Punta al registro di memoria interno `REG_POSITION_NEW`, l'unico che accetta i comandi di posizionamento target in tempo reale.
- `data[3] = 0x02`: lunghezza dei dati. Indica al firmware del servo che il valore di posizione in arrivo occupa esattamente 2 byte.
- `data[4]` e `data[5]`: payload della posizione. La variabile a 16 bit `valore_posizione` viene divisa nei suoi due byte costituenti secondo il formato Little Endian. `data[4]` contiene i bit meno significativi, estratti con `& 0xFF`, mentre `data[5]` contiene quelli più significativi, isolati tramite lo scorrimento a destra `» 8`.

### 4.3.5 Checksum

Il byte `data[6]` contiene il checksum di verifica richiesto dal protocollo. Si calcola sottraendo da `0x88` la somma dei byte da `data[1]` a `data[5]`, escludendo l'header, e mantenendo il risultato su 8 bit: `checksum = (0x88 - somma) & 0xFF`. In modo equivalente, si può partire da `0x88` e sottrarre, uno alla volta, ciascun byte, applicando la maschera a 8 bit al risultato finale.

**Trasmissione fisica** La funzione API nativa `twai_transmit` invia il frame validato sul bus CAN, con un tempo di blocco (timeout) impostato a 5 millisecondi mediante `pdMS_TO_TICKS(5)`, per non bloccare il microcontrollore in caso di cortocircuito sui cavi CAN High e CAN Low.

```
// Analizzo ogni byte del messaggio ricevuto via CAN; la coda RX
// permette di non perdere i messaggi.
```

```
twai_message_t rxMsg;
while (true) {
    if (twai_receive(&rxMsg, pdMS_TO_TICKS(10)) == ESP_OK) {
        // Comando di sincronizzazione.
        if (rxMsg.identifier == SYNC_COMMAND_ID) {
            xSemaphoreTake(mutex_stato, portMAX_DELAY);
            start_sync = true;
            xSemaphoreGive(mutex_stato);
        }
        // Messaggio destinato allo specifico slave.
        else if (rxMsg.identifier == (uint32_t)(0x100 + SLAVE_ID) &&
            rxMsg.data_length_code == 8) {
            xSemaphoreTake(mutex_stato, portMAX_DELAY);
            onda_attiva = (rxMsg.data[0] & 0x01) != 0;
            home_in_corso = (rxMsg.data[0] & 0x02) != 0;
            motore_libero = (rxMsg.data[0] & 0x04) != 0;
            motore_rientro = (rxMsg.data[0] & 0x08) != 0;

            // Imposto i parametri della sinusoide dai dati ricevuti.
            frequenza_hz = rxMsg.data[1] / 100.0f;
            ampiezza_target_passi = (float)(rxMsg.data[2] | (rxMsg.data[3] << 8));
            sfasamento_rad = (float)(rxMsg.data[4] | (rxMsg.data[5] << 8)) / 1000.0f;
            centro_target_passi = (float)(rxMsg.data[6] | (rxMsg.data[7] << 8));
            xSemaphoreGive(mutex_stato);
        }
    }
    vTaskDelay(pdMS_TO_TICKS(5));
}
```

## 4.4 Implementazione della comunicazione Ethernet

Il codice sviluppato fino a questa fase costituisce una versione completa e funzionante del sistema, ma non rappresenta ancora la soluzione finale del progetto. Successivamente, infatti, è stata implementata la comunicazione Ethernet per consentire il controllo remoto del microcontrollore master; inoltre, questa versione è stata in seguito integrata nel codice finale, progettato per permettere il monitoraggio e il controllo sia della coda basata su comunicazione CAN sia di quella basata su $I^2C$.

Per distinguerla dalla successiva versione unificata, nel seguito questa implementazione verrà indicata con il nome `IMU_CAN`. In questa sezione viene analizzata l'introduzione della comunicazione Ethernet e il suo impiego per inviare comandi al codice eseguito dal microcontrollore master.

L'integrazione della connettività cablata nel sistema si basa sull'utilizzo del modulo M5Stack Atomic PoE Base, nella variante hardware A091, equipaggiato con il controllore Ethernet Wiznet W5500. La caratteristica principale di questo chip è l'implementazione hardware dello stack

TCP/IP (TCP/IP Hardwired Stack), che supporta protocolli quali TCP, UDP, IPv4, ICMP e ARP. Questa architettura solleva il microcontrollore ESP32-S3 dalla gestione a basso livello dei pacchetti di rete, preservando risorse di calcolo utili per le operazioni eseguite in tempo reale, come il calcolo delle traiettorie dei servomotori e il campionamento del sensore IMU.

La tecnologia Power over Ethernet consente di trasmettere alimentazione e dati sul medesimo cavo di rete. Nella configurazione utilizzata durante le prove, tuttavia, il dispositivo è stato collegato direttamente alla porta Ethernet di un PC, generalmente priva di funzionalità PoE. L'alimentazione è stata quindi fornita separatamente attraverso la porta USB Type-C dell'AtomS3R, mentre il collegamento RJ45 è stato impiegato esclusivamente per il trasferimento dei dati.

### 4.4.1 Architettura dei protocolli e gestione di rete

Il sistema utilizza due canali di comunicazione separati, ottimizzati per funzioni differenti:

- **TCP (Transmission Control Protocol), porta 5000**: protocollo orientato alla connessione, scelto per l'affidabilità e il controllo di flusso. Viene utilizzato per ricevere i comandi ASCII dell'utente e trasferire i blocchi di telemetria, come i dati dell'IMU e le correnti dei servomotori.
- **UDP (User Datagram Protocol), porta 5001**: protocollo privo di connessione e caratterizzato da un overhead ridotto, utilizzato per il meccanismo di discovery e per la configurazione automatica dell'indirizzo IP mediante messaggi broadcast.

La strategia di assegnazione degli indirizzi IP è stata progettata per garantire un funzionamento di tipo plug-and-play in diversi scenari di collegamento. All'avvio del sistema o alla connessione del cavo, il firmware tenta di acquisire un indirizzo da un server DHCP, ad esempio quando il dispositivo è collegato a un router o a una rete locale gestita. Se la richiesta non riceve risposta entro il tempo previsto, come avviene tipicamente nel collegamento diretto tra PC e AtomS3R, il sistema applica una configurazione APIPA, o link-local, utilizzando l'indirizzo statico `169.254.83.82` e la maschera di sottorete `255.255.0.0`.

In assenza di un server DHCP, anche Windows assegna automaticamente alla propria interfaccia Ethernet un indirizzo appartenente alla classe `169.254.x.x`. I due dispositivi risultano quindi collocati nella stessa rete logica e possono comunicare senza richiedere all'operatore l'impostazione manuale di indirizzi statici.

### 4.4.2 Struttura multitasking e gestione concorrente delle sessioni

La logica di rete è incapsulata in un task FreeRTOS dedicato, denominato `task_ethernet`. La separazione dal resto del firmware garantisce che un'eventuale mancata inizializzazione o anomalia del modulo W5500 non comprometta il funzionamento del nodo master, che rimane controllabile tramite l'interfaccia seriale USB. Il task esegue ciclicamente le seguenti operazioni:

- `inizializza_ethernet()`: viene chiamata nel setup(). Inizializza i pin SPI (senza usare pin di interrupt o reset) e applica immediatamente l'indirizzo di fallback. Se rileva che il cavo di rete è già collegato, prova a richiedere un indirizzo DHCP, altrimenti lascia il sistema in attesa che il cavo venga inserito.
- `configura_rete_con_link_attivo()` e `applica_ip_fallback()`: se il cavo viene collegato, la prima funzione tenta di ottenere un IP dal DHCP. Se la richiesta fallisce (timeout), il sistema ripiega automaticamente sull'IP statico chiamando la seconda.
- `gestisci_stato_link()`: verifica ogni 250 ms la presenza fisica del collegamento Ethernet. Se il cavo viene scollegato, chiude e azzera i socket e ripristina la configurazione link-local, preparando il sistema a una nuova negoziazione al collegamento successivo.
- `gestisci_discovery_udp()`: ascolta i pacchetti in arrivo sulla porta 5001. Se riceve il comando `DISCOVER_ATOM_MASTER`, risponde con il proprio IP e porta TCP. Se riceve `CONFIG_ATOM_MASTER`, applica l'indirizzo IP statico richiesto dal PC e si riavvia.
- `gestisci_rinnovo_dhcp()`: mantiene valido nel tempo il lease DHCP. Nella configurazione adottata, la libreria M5-Ethernet mette a disposizione due socket; durante il rinnovo viene pertanto chiuso temporaneamente il socket UDP associato alla porta 5001 e riaperto immediatamente al termine dell'operazione.

**Parsing dei comandi TCP**: il ciclo di rete acquisisce i byte ricevuti dal client e li inserisce in un buffer fino al riconoscimento del carattere di terminazione `\n`, ignorando gli eventuali caratteri di ritorno a capo `\r`.

Per evitare collisioni e condizioni di competizione tra i comandi ricevuti contemporaneamente tramite USB ed Ethernet, il firmware utilizza due istanze separate della struttura `SessioneComandi`, denominate `sessione_usb` e `sessione_ethernet`. Ciascuna istanza possiede una propria macchina a stati e un proprio buffer di ricezione. Di conseguenza, un'operazione composta da più passaggi avviata tramite USB, ad esempio la selezione del comando `9` seguita dal parametro `1`, non viene alterata dall'arrivo simultaneo di una richiesta di telemetria tramite TCP. L'esecuzione del comando interpretato, effettuata mediante la funzione `elabora_riga_comando`, è inoltre protetta dal semaforo di mutua esclusione `mutex_comandi`.

### 4.4.3 Classe TeePrint

Quando viene elaborato un comando, il microcontrollore deve inviare al mittente l'esito dell'operazione o i dati richiesti, che nel caso della telemetria possono comprendere migliaia di righe in formato CSV. Per consentire contemporaneamente il monitoraggio locale e il debugging, lo stesso flusso deve essere riportato anche sul monitor seriale USB. La soluzione adottata è la classe `TeePrint`, basata sul polimorfismo del framework Arduino/C++ e derivata dalla classe `Print`.

```
class TeePrint : public Print {
public:
    TeePrint(Print& prima, Print& seconda) : _prima(prima), _seconda(seconda) {}
    size_t write(uint8_t valore) override {
        size_t a = _prima.write(valore);
        size_t b = _seconda.write(valore);
        return (a < b) ? a : b;
    }
    size_t write(const uint8_t* buffer, size_t dimensione) override {
        size_t a = _prima.write(buffer, dimensione);
        size_t b = _seconda.write(buffer, dimensione);
        return (a < b) ? a : b;
    }
private:
    Print& _prima;
    Print& _seconda;
};
```

Le classi `HardwareSerial`, utilizzata per la comunicazione seriale, ed `EthernetClient`, associata al socket TCP, espongono entrambe l'interfaccia definita da `Print`. La classe `TeePrint` può quindi conservarne i riferimenti e agire come wrapper comune. Quando il parser riconosce un comando ricevuto tramite Ethernet, viene creato localmente l'oggetto `TeePrint uscita_doppia(Serial, client)`. Ogni chiamata di stampa effettuata dalla logica applicativa viene così convertita in chiamate ai metodi `write()`, che duplicano i byte in due direzioni:

1. **Direzione locale**: i dati vengono inviati al riferimento `_prima`, associato a `Serial`, e raggiungono il monitor seriale del PC attraverso l'interfaccia USB dell'ESP32-S3.
2. **Direzione di rete**: gli stessi dati vengono inviati al riferimento `_seconda`, associato al client TCP. Il flusso attraversa il bus SPI, viene gestito dal W5500 e viene trasmesso al client remoto attraverso il collegamento Ethernet.

Il metodo restituisce il minore tra i numeri di byte effettivamente scritti sui due canali, mediante l'espressione `(a < b) ? a : b`. In questo modo, i livelli superiori possono rilevare una scrittura

parziale qualora uno dei due canali, in particolare quello TCP, non riesca ad accettare l'intero flusso.

### 4.4.4 Il protocollo di discovery e lo script PowerShell

Dal lato PC, la localizzazione del nodo master e la successiva interazione vengono automatizzate mediante uno script PowerShell. Al boot del codice master lo script viene lanciato su un terminale powershell e in pochi secondi stabilisce la connessione, fornendo poi una interfaccia per comandi che imita al massimo quella che si ha su seriale. L'utente vede scritto: `ATOM:` e inserisce i comandi; per terminare la sessione scrive: `exit`.

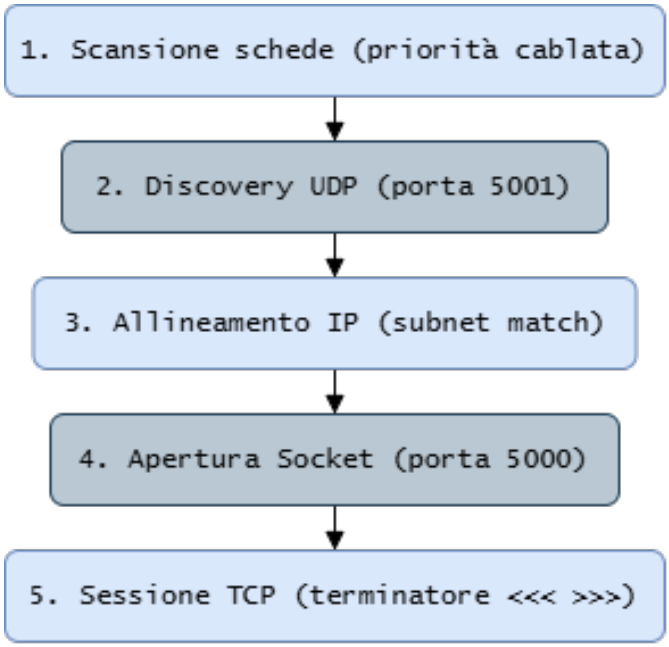



Figura 9: Flusso operativo dello script PowerShell per il discovery e l'apertura della sessione TCP.

Lo script esegue le seguenti operazioni:

- **Identificazione dell'interfaccia e priorità**: la funzione `Get-ActiveIPv4Interfaces` individua le schede di rete Windows attive e assegna la priorità più alta alle connessioni Ethernet cablate, con tipo di mezzo `802.3`, rispetto alle interfacce Wi-Fi `802.11`. Questa selezione riduce i ritardi dovuti ai timeout sulle reti non pertinenti.
- **Calcolo delle sottoreti**: le funzioni `Convert-IPv4ToUInt64` e `Get-SubnetData` convertono gli indirizzi IPv4 in valori interi a 64 bit. Lo script può quindi utilizzare operazioni logiche bit a bit e gli operatori di scorrimento `-shl` e `-shr` per determinare gli indirizzi di rete e di broadcast.

- **Discovery UDP e riconfigurazione dinamica (Auto-IP)**: questa fase costruisce il payload del datagramma UDP in funzione dell'operazione richiesta. Nel normale processo di ricerca viene trasmessa la stringa `DISCOVER_ATOM_MASTER`. Se è attivo il flag `$Configure`, lo script richiama invece la funzione `Get-DesiredAtomIP`, che calcola un indirizzo compatibile a partire dall'indirizzo IPv4 e dalla lunghezza del prefisso dell'interfaccia selezionata. Il nuovo indirizzo e il prefisso della sottorete vengono quindi inseriti nel messaggio `CONFIG_ATOM_MASTER;<ip>;<prefix>`. Il datagramma viene inviato sia all'indirizzo di broadcast della sottorete specifica sia all'indirizzo di broadcast globale `255.255.255.255`.

```
# Da Send-DiscoveryOnInterface
if ($Configure) {
    $desiredIP = Get-DesiredAtomIP `
        -PcIPAddress $Interface.IPAddress `
        -PrefixLength $Interface.PrefixLength
    if ([string]::IsNullOrWhiteSpace($desiredIP)) {
        return $null
    }
    $messageText = "CONFIG_ATOM_MASTER;$desiredIP;$($Interface.PrefixLength)"
}
else {
    $messageText = "DISCOVER_ATOM_MASTER"
}
$message = [System.Text.Encoding]::ASCII.GetBytes($messageText)
```

- **Allineamento automatico degli indirizzi**: se il PC e l'AtomS3R appartengono a reti non direttamente raggiungibili, lo script calcola un indirizzo valido nella sottorete del PC e invia il messaggio `CONFIG_ATOM_MASTER;<ip>;<subnet>`. Il firmware applica la nuova configurazione durante l'esecuzione, reinizializza l'interfaccia di rete e risponde comunicando il nuovo indirizzo.

- **Shell TCP interattiva**: dopo aver individuato l'indirizzo definitivo, lo script crea un oggetto .NET `TcpClient` e apre la connessione sulla porta 5000. La risposta viene letta una riga alla volta mediante `StreamReader.ReadLine()`. Se il metodo restituisce `null`, la connessione è stata chiusa dall'Atom e viene generata un'eccezione. Per ogni riga ricevuta, lo script verifica la presenza del marcatore `<<<END>>>`, lo rimuove dal testo prima della visualizzazione e stampa soltanto il contenuto non vuoto. Il ciclo termina esclusivamente quando viene rilevato tale marcatore, assicurando che la risposta sia stata acquisita completamente.

```
# Da Console-Atom
while ($true) {
    $line = $reader.ReadLine()
    if ($null -eq $line) {
        throw "Connessione chiusa dall'Atom."
    }
```

```
        $end = $line.Contains("<<<END>>>")
        $text = $line.Replace("<<<END>>>", "").TrimEnd()
        if ($text.Length -gt 0) {
            Write-Host $text
        }
        if ($end) {
            break
        }
    }
```

L'impiego del marcatore esplicito di fine trasmissione impedisce che la shell restituisca il controllo all'operatore prima di aver ricevuto l'intero contenuto prodotto dal comando. Questo comportamento è particolarmente importante durante il trasferimento di blocchi estesi di telemetria, poiché evita che l'output venga frammentato o sovrapposto ai comandi successivi.

## 4.5 Funzionalità della versione `IMU_CAN`

In questa sezione verranno illustrate le varie funzionalità implementate nella versione `IMU_CAN`, volte a imitare il più possibile il precedente modello propulsivo con comunicazione I2C e motori a PWM.

### 4.5.1 Setup registri

Al fine di sfruttare al meglio le funzionalità dei servomotori Hitec MDB961WP-CAN, è stata studiata una configurazione base per i principali registri interni dei dispositivi, che comprendono ad esempio modalità di lavoro, velocità della trasmissione di informazioni, parametri limite per velocità del motore, attivazione di modalità di stream automatico di dati interni. Questo schema svolge anche il ruolo di manuale per l'utente che vuole utilizzare i dispositivi con il codice sviluppato in questo progetto.

| Indirizzo | Valore | Funzione e configurazione |
|---|---|---|
| 0x32 | 1, 2, 3 | Identificativo univoco assegnato rispettivamente ai tre servomotori. |
| 0x38 | 0 | Velocità del bus CAN impostata a 1000 Kbit/s. |
| 0x6A | 0 | Selezione del formato CAN 2.0A. |
| 0x44 | 1 | Attivazione della modalità Servo Mode. |
| 0xB0 | 10922 | Limite massimo della posizione ammessa. |
| 0xB2 | 5462 | Limite minimo della posizione ammessa. |
| 0x2E | 50 | Periodo di trasmissione automatica dei dati pari a 50 ms. |

| Indirizzo | Valore | Funzione e configurazione |
|---|---|---|
| 0x30 | 1 / 0 | Abilitazione dello stream automatico con 1; disabilitazione con 0. |
| 0xE2 | 0x1612 | Inserimento nello stream dei registri di corrente e tensione. |
| 0xE4 | 0xD4D0 | Inserimento nello stream dei registri di umidità e temperatura. |
| 0xE6 | 0x0E0C | Inserimento nello stream dei registri di velocità e posizione. |
| 0x4E | 4 | Banda morta utilizzata per ridurre il jitter di posizione. |
| 0x54 | 1000 | Limite massimo della velocità del servomotore. |
| 0x70 | 65535 | Salvataggio permanente delle modifiche effettuate. |

Per impostare i valori nei rispettivi registri è stato sviluppato un codice a parte, sempre in ambiente PlatformIo, denominato `main_unificato_ascolto_passivo()`; combina infatti funzionalità per ascolto passivo (sniffer), quando inserito in una linea con più dispositivi collegati da un unico bus CAN che si scambiano messaggi, con funzioni di debug e lettura/scrittura di registri raw (seguendo la sintassi esadecimale).

La funzione seguente gestisce l'ascolto passivo del bus senza bloccare il ciclo principale. Se il controller CAN non è installato o la modalità di ascolto è disabilitata, la funzione termina immediatamente; in caso contrario, acquisisce e stampa un numero limitato di frame a ogni iterazione.

```
static void servicePassiveCanListening() {
  if (!canInstalled || !passiveListenEnabled) {
    return;
  }

  twai_message_t rx = {};
  uint8_t framesRead = 0;

  while (framesRead < PASSIVE_LISTEN_MAX_FRAMES_PER_LOOP &&
         twai_receive(&rx, 0) == ESP_OK) {
    printPassiveFrame(rx);
    framesRead++;
  }
}
```

Per impostare i registri invece è sufficiente inviare l'indirizzo, in formato `0x..`, premere invio e poi inviare il valore che si desidera impostare. Ad esempio: > `0x70` > `65535`.

La figura seguente mostra un esempio del log prodotto durante la ricezione dello stream automatico generato dal servomotore con identificativo 1.

```
RX_PASSIVE CAN_ID=0x0 STD DATA DLC=8 BYTES=56 01 D0 FB 06 D4 B2 55
RX_PASSIVE CAN_ID=0x0 STD DATA DLC=8 BYTES=56 01 CA 00 00 C8 53 01
RX_PASSIVE CAN_ID=0x0 STD DATA DLC=8 BYTES=56 01 12 B3 04 0C 0C 20
RX_PASSIVE CAN_ID=0x0 STD DATA DLC=8 BYTES=56 01 D0 FA 06 D4 B2 55
RX_PASSIVE CAN_ID=0x0 STD DATA DLC=8 BYTES=56 01 CA 00 00 C8 54 01
RX_PASSIVE CAN_ID=0x0 STD DATA DLC=8 BYTES=56 01 12 B3 04 0C 0C 20
RX_PASSIVE CAN_ID=0x0 STD DATA DLC=8 BYTES=56 01 D0 FA 06 D4 B2 55
RX_PASSIVE CAN_ID=0x0 STD DATA DLC=8 BYTES=56 01 CA 00 00 C8 54 01
```

Figura 10: Esempio di log dello stream automatico del servomotore 1.

### 4.5.2 SYNC (0x7FF)

Il comando selezionato mediante la voce `4 = SYNC` del menu del master fornisce un segnale comune di avvio ai tre nodi slave. A tale scopo il master trasmette un frame CAN standard con identificatore `0x7FF` e DLC pari a zero: il messaggio non contiene quindi un payload, poiché l'informazione è rappresentata dalla sola ricezione dell'identificatore. Come ogni frame CAN, esso è fisicamente visibile da tutti i nodi connessi al bus; i firmware slave riconoscono convenzionalmente `0x7FF` come comando di sincronizzazione.

All'accensione, la variabile `start_sync` è inizializzata a `false` e il task `task_sinusoide` di ciascuno slave rimane in attesa. In questa fase la traiettoria non viene calcolata, la fase non avanza e il servomotore non avvia l'oscillazione. Quando il task di ricezione CAN intercetta il frame `0x7FF`, acquisisce il mutex dello stato, imposta `start_sync = true` e sblocca il task sinusoidale.

Poiché lo stesso frame viene ricevuto dai tre slave durante la medesima trasmissione sul bus, l'avvio delle traiettorie avviene con uno scarto temporale ridotto, dovuto principalmente alla latenza di schedulazione locale dei singoli microcontrollori. Dopo lo sblocco, ogni slave inizializza il proprio riferimento temporale FreeRTOS e prosegue autonomamente con il ciclo di controllo della sinusoide.

```
void invia_comando_sync(void) {
    twai_message_t msg = {};
    msg.identifier = SYNC_COMMAND_ID;  // 0x7FF
    msg.extd = 0;
    msg.rtr = 0;
    msg.data_length_code = 0;

    if (twai_transmit(&msg, pdMS_TO_TICKS(50)) == ESP_OK) {
        Serial.printf(
            "[SYNC] Segnale di avvio (0x7FF) inviato in broadcast!\n");
    }
```

```
    else {
        Serial.printf("[ERRORE] Invio segnale SYNC fallito.\n");
    }
}
```

Il comando ha comportamento *one-shot*: viene utilizzato una sola volta per avviare i task dopo l'accensione. Le successive modifiche di frequenza, ampiezza, offset o sfasamento aggiornano i parametri protetti dal mutex, ma non riportano `start_sync` a `false` e non arrestano il riferimento temporale già avviato.

### 4.5.3 Stream automatico

Lo stream automatico permette ai servomotori Hitec di trasmettere periodicamente sul bus i valori dei registri interni senza richiedere una lettura esplicita da parte del master per ogni campione. La funzionalità viene configurata attraverso i registri già riportati nella tabella precedente: `0x2E` imposta il periodo di trasmissione a 50 ms, mentre `0x30` abilita o disabilita lo stream. I registri `0xE2`, `0xE4` e `0xE6` definiscono rispettivamente le coppie di grandezze relative a corrente e tensione, umidità e temperatura, velocità e posizione.

Ogni messaggio dello stream contiene otto byte ed è riconosciuto dal firmware del master mediante l'header applicativo `0x56` presente nel primo byte. Il formato utilizzato è il seguente:

- `data[0] = 0x56`: header dello stream automatico Hitec;
- `data[1]`: identificativo del servomotore trasmittente, compreso tra 1 e 3;
- `data[2]`: indirizzo del primo registro trasmesso;
- `data[3]` e `data[4]`: valore del primo registro, codificato nei byte LOW e HIGH;
- `data[5]`: indirizzo del secondo registro trasmesso;
- `data[6]` e `data[7]`: valore del secondo registro, codificato nei byte LOW e HIGH.

I frame percorrono lo stesso bus CAN a 1 Mbit/s utilizzato per i comandi di configurazione e di movimento: lo stream non costituisce quindi un canale separato. Sul master, il task `task_stream` rimane bloccato sulla funzione `twai_receive()` con un timeout di 100 ms. Quando riceve un frame con header `0x56`, verifica l'identificativo del servo e salva il messaggio nel buffer circolare corrispondente, senza effettuare immediatamente la conversione nelle relative unità fisiche.

Per ciascun servomotore è presente un'istanza separata di `buffer_stream`, protetta da un mutex dedicato. Ogni campione conserva il timestamp di ricezione e gli otto byte grezzi del frame.

Quando viene raggiunta la capacità definita da `BUFFER_STREAM_LEN`, l'indice di scrittura riparte dall'inizio e i campioni più vecchi vengono progressivamente sovrascritti, mantenendo disponibili i dati più recenti senza interrompere la ricezione.

```
void salva_in_buffer_circolare(uint8_t idx_servo,
                               twai_message_t *frame) {
    xSemaphoreTake(buffer_stream[idx_servo].mutex, portMAX_DELAY);

    uint16_t pos = buffer_stream[idx_servo].indice_scrittura;
    buffer_stream[idx_servo].campioni[pos].timestamp_ms = millis();
    memcpy(buffer_stream[idx_servo].campioni[pos].data,
           frame->data, 8);

    buffer_stream[idx_servo].indice_scrittura =
        (pos + 1) % BUFFER_STREAM_LEN;
    if (buffer_stream[idx_servo].indice_scrittura == 0) {
        buffer_stream[idx_servo].buffer_pieno = true;
    }

    xSemaphoreGive(buffer_stream[idx_servo].mutex);
}
```

Il comando `6 = DUMP BUFFER` avvia lo scaricamento sequenziale dei dati dei tre servomotori. Prima della stampa, il master imposta `dump_in_corso = true`: il task di ricezione continua a leggere i frame dal driver TWAI, ma non li inserisce nei buffer. In questo modo il contenuto rimane congelato durante il dump e i campioni non vengono sovrascritti mentre sono trasferiti sulla seriale o sulla connessione TCP. Al termine dell'operazione il flag viene riportato a `false` e la memorizzazione dello stream riprende normalmente.

### 4.5.4 Motor Lock/Unlock

I comandi `7 = MOTOR FREE` e `8 = MOTOR LOCK` permettono rispettivamente di disattivare e riattivare la coppia dei servomotori. Lo stato richiesto viene codificato dal master nel primo byte del frame di configurazione: il bit 2 indica la richiesta di motore libero, mentre il bit 3 segnala la procedura di rientro e riaccoppiamento. Gli slave ricevono tali flag e aggiornano le variabili condivise `motore_libero` e `motore_rientro` sotto la protezione del mutex `mutex_stato`.

Con il comando `MOTOR FREE`, il master disabilita l'onda e richiede lo sgancio del motore. Prima di rimuovere la coppia, lo slave riduce la velocità massima attraverso il registro `0x54`, invia al servo la posizione centrale 8192 e attende il completamento del rientro. Successivamente scrive il valore `0x0200` nel registro `REG_POWER_CONFIG`, con indirizzo `0x46`, portando il servo nello stato

libero. La fase e l'ampiezza correnti vengono azzerate, evitando che alla successiva riattivazione il movimento riparta da un punto arbitrario della sinusoide.

Il comando `MOTOR LOCK` esegue la sequenza inversa. Il master imposta lo stato di rientro, mantiene disabilitata l'onda e abilita la procedura di home. Lo slave riduce nuovamente la velocità, invia il riferimento centrale e scrive `0x0000` nel registro `0x46`, riattivando la coppia. Dopo il tempo assegnato al rientro, pari a `TEMPO_RIENTRO_MS` (3000 ms), la velocità massima viene riportata al valore normale e lo stato locale del motore torna a `NORMALE`.

Nella revisione finale del firmware, queste attese sono gestite mediante una macchina a stati non bloccante basata su `millis()`. In questo modo il task sinusoidale continua a essere schedulato durante i tre secondi necessari alle transizioni, senza utilizzare un lungo `vTaskDelay()` e senza provocare successivi cicli di recupero ravvicinati.

Codice nel master:

```
case 7:
        stato_motore = MOTORE_LIBERO;
        home_in_corso = false;
        onda_attiva = false;
        invia_a_tutti_gli_slave();
        uscita.printf("[INFO] MOTOR FREE inviato: coppia disattivata.\n");
        break;

    case 8:
        stato_motore = MOTORE_RIENTRO;
        invia_a_tutti_gli_slave();
        uscita.printf(
            "[INFO] MOTOR LOCK inviato: richiesta snapshot posizione attiva.\n");
        break;
```

La funzione `inoltra_comando_power()` traduce lo stato logico nella scrittura del registro di potenza del servo:

```
void inoltra_comando_power(bool motor_free_on) {
    invia_registro_servo(
        0x46, motor_free_on ? 0x0200 : 0x0000);
}
```

Una volta completato il `MOTOR LOCK`, il comando `1 = SIN` può riattivare l'onda. Poiché la fase e l'ampiezza sono state azzerate durante lo sgancio, la traiettoria riparte in modo controllato dal centro, applicando nuovamente le rampe previste dal task sinusoidale.

### 4.5.5 Lettura IMU

Ogni nodo slave utilizza la IMU BMI270 integrata nell'AtomS3R per acquisire le velocità angolari sui tre assi. La lettura del sensore viene eseguita sul core 1 tramite M5Unified, mentre la trasmissione dei campioni sul bus CAN è affidata al task `task_trasmissione_can_imu`, vincolato al core 0. Questa separazione evita che le operazioni $I^2C$ della IMU condividano lo stesso flusso di esecuzione con le interruzioni e le operazioni del driver TWAI.

Il ciclo principale richiama periodicamente `M5.Imu.update()` e copia i valori restituiti da `M5.Imu.getImuData()` nelle variabili globali `gyro_x`, `gyro_y` e `gyro_z`. Il task CAN legge tali valori, li moltiplica per 100 e li converte in interi con segno a 16 bit, conservando due cifre decimali durante la trasmissione.

I frame inviati dagli slave utilizzano gli identificatori compresi tra `0x301` e `0x303`, ottenuti mediante l'espressione `0x300 + SLAVE_ID`. Il payload di otto byte è organizzato come segue:

- `data[0] = 0x55`: header che identifica un pacchetto di telemetria IMU;
- `data[1]`: identificativo dello slave trasmittente;
- `data[2]` e `data[3]`: componente X del giroscopio, rispettivamente byte LOW e HIGH;
- `data[4]` e `data[5]`: componente Y del giroscopio, rispettivamente byte LOW e HIGH;
- `data[6]` e `data[7]`: componente Z del giroscopio, rispettivamente byte LOW e HIGH.

```
void task_trasmissione_can_imu(void *arg) {
    const TickType_t periodo = pdMS_TO_TICKS(37);

    while (true) {
        vTaskDelay(periodo);

        if (start_sync) {
            int16_t gx_i = (int16_t)(gyro_x * 100.0f);
            int16_t gy_i = (int16_t)(gyro_y * 100.0f);
            int16_t gz_i = (int16_t)(gyro_z * 100.0f);

            twai_message_t msg = {};
            msg.identifier = 0x300 + SLAVE_ID;
            msg.extd = 0;
            msg.rtr = 0;
            msg.data_length_code = 8;

            msg.data[0] = 0x55;
            msg.data[1] = SLAVE_ID;
```

```
            msg.data[2] = gx_i & 0xFF;
            msg.data[3] = (gx_i >> 8) & 0xFF;
            msg.data[4] = gy_i & 0xFF;
            msg.data[5] = (gy_i >> 8) & 0xFF;
            msg.data[6] = gz_i & 0xFF;
            msg.data[7] = (gz_i >> 8) & 0xFF;

            twai_transmit(&msg, pdMS_TO_TICKS(2));
        }
    }
}
```

**Scelta del periodo di 37 ms** Il periodo di 37 ms corrisponde a una frequenza di campionamento e trasmissione di circa 27 Hz. Tale valore è stato scelto intenzionalmente perché non è multiplo né del ciclo di controllo della traiettoria, pari a 20 ms, né del periodo dello stream automatico dei servomotori, pari a 50 ms. Essendo 37 coprimo rispetto a entrambi, gli istanti di attivazione dei diversi flussi scorrono progressivamente gli uni rispetto agli altri, anziché coincidere sempre negli stessi punti temporali.

Questa scelta non elimina la contesa, che viene comunque risolta dall'arbitraggio CAN, ma riduce la formazione di picchi periodici nei quali calcolo della traiettoria, stream dei servo e trasmissione IMU richiederebbero contemporaneamente l'accesso al bus o al processore. Il traffico viene così distribuito più uniformemente nel tempo, limitando il jitter e la probabilità di ritardi ricorrenti. Le coincidenze con il ciclo da 20 ms si ripetono ogni 740 ms, mentre quelle con lo stream da 50 ms si ripetono ogni 1850 ms, invece di verificarsi a ogni iterazione.

Sul master, `task_stream` riconosce i pacchetti IMU controllando l'intervallo degli identificatori, il DLC e l'header `0x55`. I tre valori vengono ricostruiti come interi a 16 bit con segno, divisi per 100 e memorizzati nel buffer circolare `buffer_imu` associato allo slave. Ogni buffer è protetto da un mutex dedicato e conserva gli ultimi campioni ricevuti.

Il comando `5 = DUMP IMU` congela temporaneamente l'aggiornamento dei buffer tramite il flag `dump_in_corso` e stampa i dati dei tre nodi in formato CSV, con le colonne `X,Y,Z`. Al termine del trasferimento il flag viene disattivato e la raccolta della telemetria riprende normalmente.

## 4.6 Prodotto finale

Durante le prove sulle due versioni della coda del pesce robotico è emersa la necessità di utilizzare un unico codice, denominato `worlds`, caricabile sul microcontrollore master AtomS3 Ext. L'obiettivo era poter scegliere all'avvio quale coda testare: quella precedente, con comunicazione

$I^2C$ e motori pilotati in PWM, oppure quella nuova, con servomotori dotati di registri interni e comunicazione CAN, senza dover caricare ogni volta un firmware diverso.

Dopo un primo tentativo basato su un unico file contenente tutte le funzionalità, il programma è stato riorganizzato in una struttura modulare a tre file. Il risultato è un solo firmware che contiene due ambienti operativi distinti, dei quali viene attivato soltanto quello scelto dall'operatore. Questa separazione permette di mantenere le due logiche di controllo indipendenti, condividendo la fase iniziale di avvio e l'hardware del master.

- `main.cpp` – **Avvio e selezione**: gestisce l'inizializzazione della comunicazione seriale e del modulo Ethernet W5500 della base PoE, eseguita una sola volta. Attende poi un carattere via USB oppure tramite una connessione TCP sulla porta 5000: `1` seleziona l'ambiente $I^2C$, mentre `2` seleziona quello CAN. Una volta ricevuta la scelta, chiude il server temporaneo di selezione e passa il controllo all'ambiente richiesto, mantenendo aperta la connessione del client già collegato.
- `i2c_wrld.cpp` – **Master $I^2C$**: contiene le funzioni di controllo della coda precedente e assume la gestione del microcontrollore quando viene selezionata questa modalità. La classe `TeeStreamI2C` gestisce l'ingresso e l'uscita dei comandi attraverso la seriale USB e il client Ethernet. Le attività principali sono distribuite tra task FreeRTOS, mentre l'accesso al bus $I^2C$ è protetto da mutex per evitare che più task eseguano contemporaneamente operazioni di comunicazione.
- `can_wrld.cpp` – **Master CAN**: contiene la logica della nuova coda e attiva il driver TWAI soltanto quando viene scelta la modalità CAN. Utilizza un task Ethernet dedicato e la classe `TeePrintCan` per replicare l'output sulla seriale e sulla connessione di rete. Gestisce inoltre la ricezione asincrona della telemetria dei servomotori e dei dati IMU, memorizzandoli in buffer circolari per il successivo scaricamento.

La suddivisione non è stata introdotta soltanto per rendere il codice più leggibile: ha permesso anche di affrontare i problemi di integrazione e le instabilità osservate nel primo programma combinato. Uno dei problemi principali riguardava il passaggio dal menu iniziale all'ambiente operativo, durante il quale la connessione Ethernet veniva interrotta. Nella nuova struttura, `main.cpp` trasferisce il client già connesso al task dell'ambiente selezionato senza forzarne la disconnessione. Gli script PowerShell possono quindi proseguire sulla stessa sessione TCP, senza dover ristabilire il collegamento dopo la scelta della coda.

In entrambi gli ambienti è stato inoltre introdotto un buffer di linea, `eth_buffer`, che accumula i caratteri ricevuti fino al terminatore di invio `\n`. Solo quando la riga è completa viene passata al parser dei comandi. In questo modo, anche se un comando arriva in più porzioni attraverso

la rete, non viene interpretato prematuramente: si evitano così le perdite di dati dovute alla gestione di stringhe incomplete.

### 4.6.1 Funzionalità della versione `worlds`

In questa versione finale sono state introdotte modifiche e migliorie per semplificare le prove e rendere il sistema più sicuro durante l'utilizzo. L'obiettivo è facilitare il testing delle due versioni della coda, rendendo più agevole per l'utente la gestione dei comandi e delle operazioni di prova.

Per prima cosa è stato riorganizzato il menu dei comandi, cercando di rendere le due modalità il più possibile simili. Le funzioni comuni, come l'attivazione dei relè e l'accesso al sottomenu `CUSTOM`, utilizzano gli stessi numeri; restano invece alcune differenze legate al funzionamento specifico delle due code. All'avvio, `1` seleziona la coda $I^2C$ e `2` quella CAN. Le tabelle seguenti riassumono i comandi disponibili nei rispettivi ambienti.

Tabella 15: Confronto tra i menu principali delle modalità $I^2C$ e CAN.

| Tasto | Modalità $I^2C$ | Modalità CAN |
|---|---|---|
| `0` | `HOME`: azzera i parametri, mantenendo gli offset dei tre segmenti a 5, 6 e 6. | `HOME`: azzera tutti i parametri. |
| `1` | Avvia la sinusoide predefinita con frequenza di 0,3 Hz e valori predefiniti di ampiezza, sfasamento e offset. | Avvia la sinusoide. |
| `2` | Attiva il relè dell'alimentazione a 12 V. | Attiva il relè dell'alimentazione a 12 V. |
| `3` | Disattiva il relè dell'alimentazione a 12 V. | Disattiva il relè dell'alimentazione a 12 V. |
| `4` | `HOME HARD`: imposta i parametri predefiniti con frequenza nulla. | `SYNC`: invia il segnale di sincronizzazione all'avvio, dopo l'attivazione del relè. |
| `5` | `DESTRA HARD`: imposta i parametri predefiniti con frequenza nulla e bias di direzione. | `DUMP IMU`: scarica i dati inerziali memorizzati. |
| `6` | `SINISTRA HARD`: imposta i parametri predefiniti con frequenza nulla e bias di direzione. | `DUMP STREAM`: scarica la telemetria dei servomotori memorizzata. |
| `7` | `ARM`: arma la coda. | Arma il relè della coda, comandato dal pin 3. |
| `8` | `DISARM`: disarma la coda. | Disarma il relè della coda, comandato dal pin 3. |
| `9` | Accede al sottomenu `CUSTOM`. | Accede al sottomenu `CUSTOM`. |

Il sottomenu `CUSTOM` raccoglie le operazioni di configurazione e diagnostica. I tasti riportati nella

tabella seguente si riferiscono quindi al menu aperto con `9`, non al menu principale; il trattino indica una voce non riportata per quella modalità.

Tabella 16: Confronto tra i sottomenu `CUSTOM` delle due modalità.

| Tasto | Modalità $I^2C$ | Modalità CAN |
|---|---|---|
| `0` | – | `UPGRADE`: aggiorna i parametri. |
| `1` | Seleziona lo slave tramite l'indirizzo `0x20`, `0x21` o `0x22`, oppure `0` per il broadcast. Consente di inviare `SET f,a,p,o`, `UPGRADE` o `READVA`. Dopo lo snapshot, nella sessione successiva sono disponibili `READVAV`, `READVAI`, `READVAT` e `READIMU`. | Seleziona il servo 1, 2 o 3 e imposta i parametri `f a p o`. Accetta anche la sola frequenza oppure frequenza e ampiezza. |
| `2` | `DESTRA`: applica un bias alla sinusoide predefinita. | `DESTRA`: applica un bias, sia a coda ferma sia durante il movimento. |
| `3` | `SINISTRA`: applica un bias alla sinusoide predefinita. | `SINISTRA`: applica un bias, sia a coda ferma sia durante il movimento. |
| `5` | Legge i dati del sensore INA226. | Legge i dati del sensore INA226. |
| `6` | `LOS`. | – |
| `7` | – | `MOTOR FREE`: disattiva la coppia dei servomotori. |
| `8` | – | `MOTOR LOCK`: riattiva la coppia dei servomotori. |
| `9` | `CONFIG`. | – |
| `q` | Esce dal sottomenu. | – |

Nei comandi di impostazione dell'onda, `f`, `a`, `p` e `o` indicano rispettivamente frequenza, ampiezza, sfasamento e offset. I parametri predefiniti sono stati ereditati dal progetto precedente e derivano dalle prove e dagli studi svolti dal team meccanico per definire il comportamento della sinusoide necessario a generare il movimento della coda. In particolare, la scelta delle ampiezze e degli sfasamenti tiene conto della lunghezza dei singoli segmenti e della geometria della coda, in modo da ottenere un movimento coordinato tra le articolazioni. I valori riportati nel prospetto dei comandi sono riassunti nella tabella seguente: ampiezza e sfasamento sono comuni alle due modalità, mentre gli offset differiscono. Proprio perché questi parametri sono stati definiti per la coda $I^2C$, potrebbero non essere del tutto adatti alla nuova struttura della coda CAN. Trattandosi ancora dei primi prototipi, i valori ereditati rappresentano quindi un punto di partenza da verificare ed eventualmente adattare attraverso ulteriori prove.

Tabella 17: Parametri predefiniti dell'onda per i tre segmenti della coda.

| Segmento | Ampiezza | Sfasamento | Offset CAN | Offset $I^2C$ |
|---|---|---|---|---|
| 1 | 10,31 | 1,71 | 0 | 5 |
| 2 | 20,26 | −0,41 | 0 | 6 |
| 3 | 26,92 | −1,84 | 0 | 6 |

**Relè** Per gestire separatamente l'alimentazione generale del blocco di controllo e quella della coda è stato inserito un modulo a doppio relè, collegato a due pin dell'AtomS3 Ext master. I due canali vengono attivati e disattivati tramite i comandi da tastiera, con la stessa logica di gestione in entrambe le modalità. Il pin 11 controlla il relè dell'alimentazione a 12 V, mentre il pin 3 controlla il relè della coda.

I pin sono definiti e configurati come uscite durante l'inizializzazione:

```
const int relayTail = 3; // Pin 3 dell'Atom S3 EXT per il Relay
const int relay12V_pin = 11;  //Relay 12V


delay(500);

pinMode(relayTail, OUTPUT);
pinMode(relay12V_pin, OUTPUT);
```

I comandi `2` e `3` gestiscono l'alimentazione a 12 V; i comandi `7` e `8` armano e disarmano il relè della coda. Gli estratti seguenti riportano le chiamate utilizzate nell'ambiente CAN:

```
case 2:
    motorArmed_12v_can(relay12V_pin);
    uscita.printf("[INFO] Relè 12V ACCESO (Pin 11). Attesa boot degli Slave...\n");
    break;
case 3: // Spostato qui il relè 12V OFF
    motorDisarmed_12v_can(relay12V_pin);
    uscita.printf("[INFO] Relè 12V SPENTO (Pin 11).\n");
    break;


case 7:
    motorArmed_can(relayTail);
    uscita.printf("[INFO] Relay armato (Pin 3).\n");
    break;
case 8:
```

```
    motorDisarmed_can(relayTail);
    uscita.printf("[INFO] Relay disarmato (Pin 3).\n");
    break;
```

**INA226-1A** Il modulo INA226-1A è stato inserito per monitorare la tensione delle batterie generali che alimentano l'intero blocco, la corrente assorbita e la potenza utilizzata. La gestione delle misure è la stessa nelle due modalità e non dipende dal protocollo di comunicazione della coda. In ciascun ambiente è presente un task dedicato, `task_ina_i2c` oppure `task_ina_can`, che separa l'acquisizione dai task di gestione dei comandi e della telemetria, limitando le interferenze tra queste attività.

Il sensore utilizza un secondo bus $I^2C$, `Wire1`, indipendente da quello dedicato alla comunicazione. I pin SDA e SCL sono rispettivamente il 9 e il 10, mentre `0x41` è l'indirizzo $I^2C$ del dispositivo, non l'indirizzo di un suo registro interno. Gli estratti seguenti riportano la configurazione utilizzata nell'ambiente CAN:

```
// INA
#define INA_SDA 9
#define INA_SCL 10
const uint8_t INA_ADDR = 0x41;
INA226 ina_can(INA_ADDR, &Wire1);
```

```
// Inizializza il secondo bus I2C indipendente per la INA
Wire1.begin(INA_SDA, INA_SCL, 100000);
```

Le misure vengono acquisite ogni 50 ms, cioè a 20 Hz, e salvate insieme al timestamp in un buffer circolare da 500 campioni. Una volta esaurito lo spazio disponibile, i nuovi campioni sostituiscono quelli più vecchi, mantenendo una finestra di circa 25 s di dati alla frequenza prevista. Il buffer dispone di un mutex per coordinare l'accesso ai dati tra le operazioni di acquisizione e quelle di lettura.

```
// Buffer circolare INA226: 500 campioni, acquisizione continua a 20 Hz.
#define BUFFER_INA_LEN 500
#define INA_SAMPLE_INTERVAL_MS 50

typedef struct {
    uint32_t timestamp_ms;
    float tensione_v;
    float corrente_a;
    float potenza_w;
} CampioneINA;
```

```
typedef struct {
    CampioneINA campioni[BUFFER_INA_LEN];
    uint16_t indice_scrittura;
    bool buffer_pieno;
    SemaphoreHandle_t mutex;
} BufferCircolareINA;

static BufferCircolareINA buffer_ina_can = {};
static bool ina_can_pronta = false;
```

Il task esegue la lettura soltanto quando il sensore è pronto e passa le tre grandezze alla funzione di salvataggio. La temporizzazione è gestita tramite `vTaskDelayUntil()`, utilizzando il riferimento dell'ultimo risveglio per mantenere la periodicità prevista:

```
// Task indipendente: legge INA226 ogni 50 ms e mantiene sempre gli ultimi 500 campioni.
void task_ina_can(void* arg) {
    (void)arg;

    TickType_t ultimo_risveglio = xTaskGetTickCount();

    while (true) {
        if (ina_can_pronta) {
            float tensione = ina_can.getBusVoltage();
            float corrente = ina_can.getCurrent();
            float potenza = ina_can.getPower();

            salva_in_buffer_ina_can(tensione, corrente, potenza);
        }

        vTaskDelayUntil(&ultimo_risveglio, pdMS_TO_TICKS(INA_SAMPLE_INTERVAL_MS));
    }
}
```

Quando l'operatore richiede la lettura INA dal sottomenu `CUSTOM`, una funzione dedicata stampa il contenuto del buffer. L'acquisizione periodica e la visualizzazione su richiesta restano quindi separate: il sistema raccoglie i dati durante le prove senza dover stampare ogni campione appena viene letto.

# 5 Test e validazione

## 5.1 Configurazione sperimentale e scenari di test

Per effettuare le prove di movimento in laboratorio, le due versioni della coda sono state montate su un piedistallo appositamente predisposto.

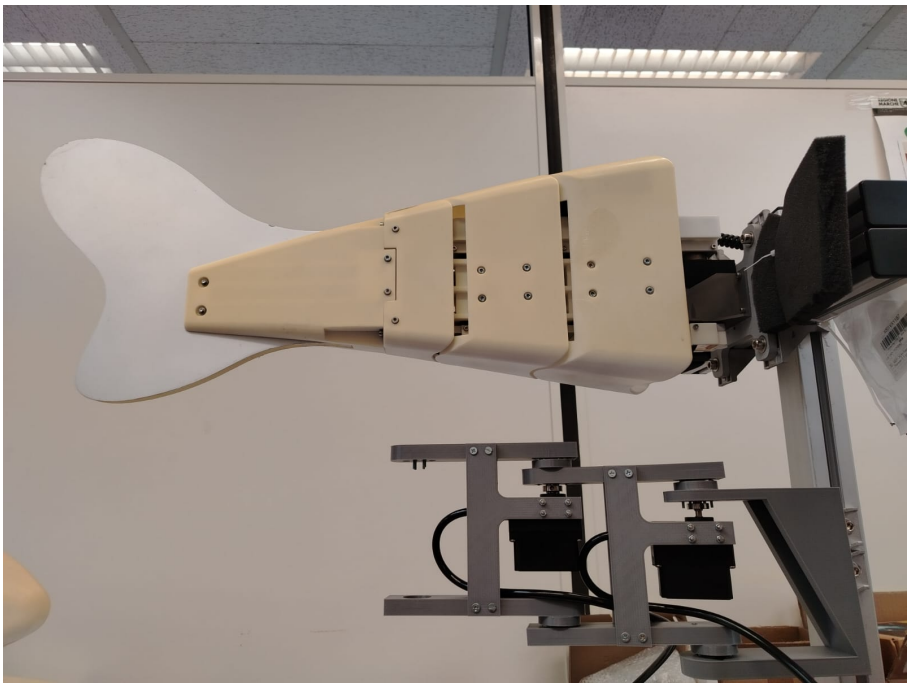


Figura 11: Setup sperimentale per le prove di movimento delle code su piedistallo.

Le prove sono state svolte senza il resto della struttura del pesce robotico, mantenendo ciascuna coda collegata al blocco di controllo principale tramite il rispettivo bus di comunicazione: $I^2C$ per la versione precedente e CAN per quella nuova. Questa configurazione ha permesso di osservare il movimento dei segmenti e verificare il funzionamento del sistema di controllo separatamente dal resto del robot.

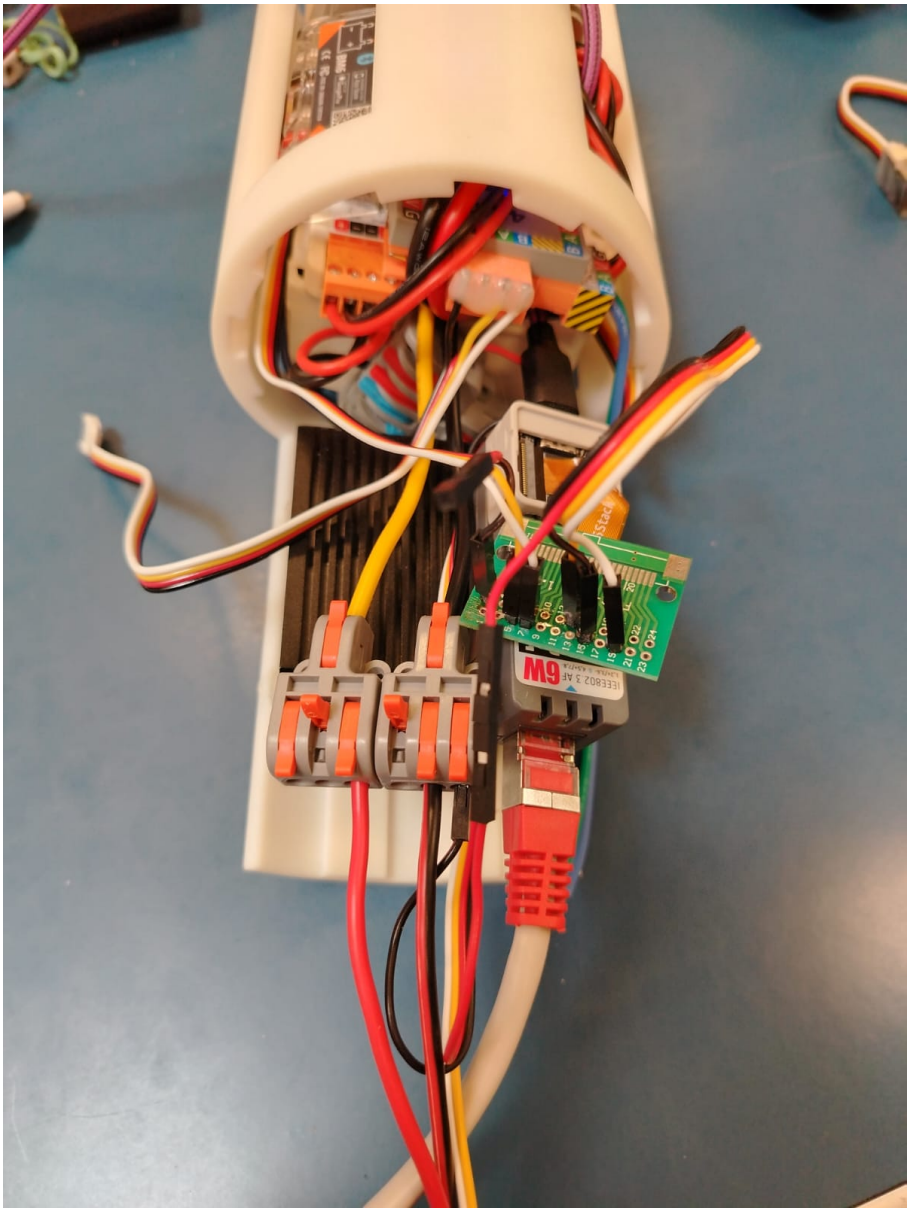


Figura 12: Ulteriore vista della configurazione sperimentale per le prove delle code.

## 5.2 Procedure di verifica e criteri di validazione

La verifica è consistita nell'osservazione del movimento e nell'analisi dei dati raccolti, per valutare se il comportamento della coda corrispondesse a quello desiderato. A questo scopo sono stati sviluppati script di visualizzazione in MATLAB, che permettono di rappresentare e confrontare i dati dei tre servomotori, facilitando l'individuazione di eventuali differenze tra i segmenti della coda.

Il seguente script analizza il log dello stream dei servomotori CAN e genera grafici di confronto delle grandezze acquisite. L'elaborazione si svolge dopo lo scaricamento dei buffer ed è organizzata

nelle fasi seguenti.

**Inizializzazione estesa delle strutture dati** All'interno del ciclo che rileva il dump dei singoli servomotori, lo script inizializza i vettori destinati a raccogliere le grandezze acquisite e i relativi tempi. Oltre a posizione, velocità, tensione e corrente, sono previsti campi specifici per la temperatura del motore e l'umidità interna:

```
% Inizializzazione vettori (inclusi Temperatura e Umidità)
servos.(id_field).p_data = []; servos.(id_field).p_time = [];
servos.(id_field).v_data = []; servos.(id_field).v_time = [];
servos.(id_field).vlt_data = []; servos.(id_field).vlt_time = [];
servos.(id_field).c_data = []; servos.(id_field).c_time = [];
servos.(id_field).temp_data = []; servos.(id_field).temp_time = [];
servos.(id_field).hum_data = []; servos.(id_field).hum_time = [];
```

**Gestione del tempo e dei registri avanzati** L'asse temporale viene ricostruito contando le occorrenze del registro di posizione `0x0C`, con un passo impostato nello script a 50 ms (`dt = 0.050`). Si tratta di un tempo stimato dal conteggio dei campioni poiché lo stream descritto in precedenza prevede un periodo di 50 ms.

Tuttavia, l'impiego di un incremento temporale fisso presenta una limitazione intrinseca in presenza di discontinuità, come avviene durante il "congelamento" dei buffer per le operazioni di dump (riscontrabile in Figura 14). Attualmente, la funzione di esportazione seriale stampa esclusivamente i byte del payload grezzo omettendo il timestamp temporale di ricezione. Come sviluppo futuro, la procedura di esportazione verrà aggiornata per includere nel log il timestamp assoluto generato dal microcontrollore. Questo consentirà agli script di analisi di posizionare ogni campione nella sua esatta collocazione temporale, rappresentando correttamente le pause di acquisizione senza alterare la forma e la continuità visiva della traiettoria.

Le funzioni di supporto decodificano i valori a 16 bit e li assegnano ai campi corrispondenti. Il registro `0xD0` è associato alla temperatura del motore, mentre il valore del registro `0xD4`, relativo all'umidità interna, viene convertito in percentuale di umidità relativa (

```
case hex2dec('D0') % Temperatura Motore / Interna
    s_struct.temp_data(end+1) = valore;
    s_struct.temp_time(end+1) = tempo_attuale;
case hex2dec('D4') % Umidità interna con formula ufficiale Hitec (%RH)
    if valore < 0
        valore = valore + 65536; % Correzione per valori signed/unsigned
    end
    umidita_rh = (125.0 * double(valore) / 65536.0) - 6.0;
```

```
s_struct.hum_data(end+1) = umidita_rh;
s_struct.hum_time(end+1) = tempo_attuale;
```

**Grafici di confronto a sei pannelli** Lo script genera una finestra grafica verticale con sei sotto-grafici, organizzati tramite `subplot(6, 1, x)`. In ciascun pannello vengono confrontate le serie dei servomotori presenti nel log, identificati come Servo 1, 2 e 3, secondo le seguenti grandezze:

- **Posizione**: movimento angolare in gradi rispetto al centro, con limiti di visualizzazione pari a $\pm 65°$.
- **Velocità**: modulo della velocità, espresso in count/s.
- **Tensione**: andamento della tensione di alimentazione misurata dai servomotori. I dati grezzi sono convertiti in volt applicando il fattore di scala indicato dal produttore (100 unità grezze corrispondono a 1 V).
- **Corrente**: andamento della corrente assorbita da ciascun attuatore. I valori trasmessi dal registro in milliampere sono convertiti in ampere per uniformare la visualizzazione.
- **Temperatura**: andamento della temperatura interna del motore. Poiché la documentazione tecnica Hitec non specifica la formula di conversione per questo registro, i valori sono mantenuti in unità grezze (raw). In questa fase prototipale la grandezza è osservata qualitativamente, per individuare eventuali tendenze al surriscaldamento durante cicli di lavoro prolungati.
- **Umidità**: andamento dell'umidità interna registrata dai sensori. I valori grezzi sono convertiti in umidità relativa percentuale (%RH) applicando la formula indicata nella documentazione Hitec.

La visualizzazione permette di confrontare il movimento e gli sfasamenti tra i segmenti, affiancando alle informazioni sull'alimentazione e sull'assorbimento quelle relative alle condizioni interne degli attuatori.

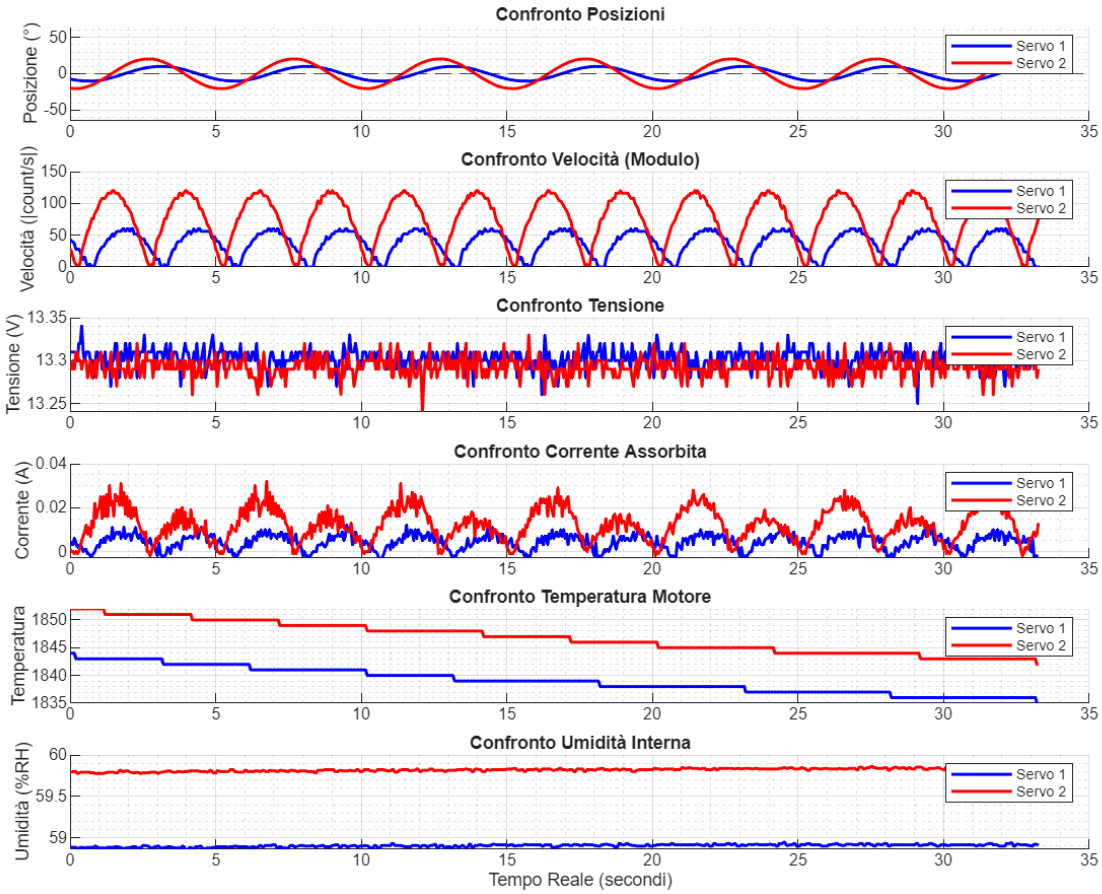



Figura 13: Confronto dei dati di due servomotori CAN: prima acquisizione.

Nella prova riportata in figura 13, i due servomotori sono stati impostati con i parametri predefiniti della sinusoide testata. Nel primo pannello, dedicato alle posizioni, si può osservare che, dopo il transitorio iniziale, lo sfasamento tra le due onde rimane sostanzialmente costante. Questo andamento permette di valutare il mantenimento della relazione di fase prevista tra i due segmenti durante il movimento.

La figura riporta inoltre la temperatura interna del motore, ancora espressa in unità grezze e quindi utile in questa fase per una valutazione qualitativa di eventuali tendenze al surriscaldamento. L'umidità è invece espressa in percentuale di umidità relativa (%RH). Le scale di riferimento e i limiti operativi dovranno essere definiti rispetto alle condizioni attese quando il robot funzionerà a regime.

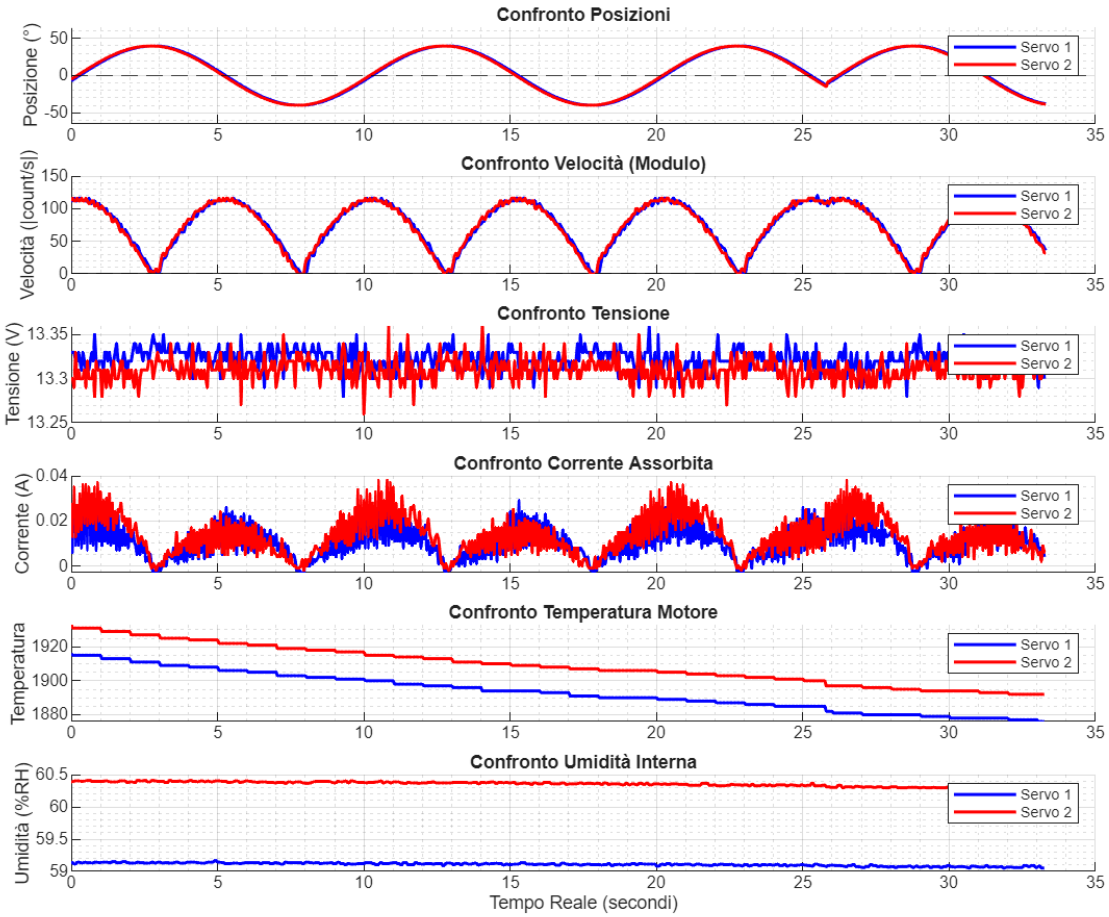



Figura 14: Confronto dei dati di due servomotori CAN: seconda acquisizione.

Nella prova riportata in figura 14, entrambi i servomotori sono stati avviati con parametri identici e a bassa frequenza. Intorno al secondo 10 si osserva una discontinuità nelle curve di posizione, che mette in evidenza un limite della gestione del buffer durante lo scaricamento dei dati. Prima di questa acquisizione era già stato richiesto un dump: per tutta la durata della stampa, il contenuto dei buffer era rimasto congelato e i nuovi campioni ricevuti non erano stati memorizzati. Il salvataggio è ripreso soltanto al termine dell'operazione.

Nel buffer rimangono quindi campioni precedenti e successivi al dump, separati da un intervallo privo di dati. Quando lo script li rappresenta come una sequenza continua, ricostruendo il tempo dal conteggio dei campioni, questa pausa produce un "taglio" nell'onda visualizzata. La discontinuità del grafico non indica quindi, di per sé, un'interruzione del movimento dei servomotori: deriva dalla sospensione della memorizzazione durante il dump, non dal funzionamento del buffer circolare in sé.

Per evitare che il dump successivo contenga ancora questa lacuna, è sufficiente attendere, dopo il termine della stampa, il tempo necessario a rinnovare completamente il contenuto del buffer con nuovi campioni. Tra due richieste consecutive va quindi lasciato un intervallo adeguato alla

capacità del buffer e alla frequenza effettiva di acquisizione.

Durante lo sviluppo, la posizione target e quella misurata sono state costantemente confrontate per calibrare il sistema. Nei grafici di validazione finali, tuttavia, il riferimento non è stato sovrapposto a causa del limite di 8 byte del payload CAN: si è scelto di dare priorità assoluta al monitoraggio della sicurezza hardware (tensioni, correnti, temperature e umidità). Essendo lo stream dei servomotori pienamente configurabile, per le future prove in acqua sarà sufficiente modificare i registri per trasmettere il target e calcolare quantitativamente l'errore RMS di inseguimento.

**Dump IMU** La visualizzazione dei dati IMU è più semplice rispetto a quella dello stream dei servomotori, perché i valori sono già ordinati e decodificati. Lo script può quindi utilizzare direttamente i tempi e le componenti giroscopiche dei singoli slave, senza ricostruire le grandezze a partire dai byte dei registri. I dati vengono rappresentati in tre sotto-grafici, uno per ciascun asse, per confrontare le misure dei diversi nodi.

Il seguente estratto mostra il grafico della velocità angolare sull'asse X, associato al rollio. Ogni slave è identificato da un colore e da una voce in legenda; i limiti verticali sono fissati a ±60 gradi/s per mantenere una scala di confronto costante:

```
% Sotto-grafico 1: Giroscopio Asse X (Roll)
subplot(3, 1, 1);
hold on;
for k = 1:length(elenco_slave)
    s = slaves.(elenco_slave{k});
    num_slave = regexp(elenco_slave{k}, '\d+', 'match');
    plot(s.time, s.x_data, 'Color', colors{mod(k-1, length(colors))+1}, ...
         'LineWidth', 1.5, 'DisplayName', sprintf('Slave %s', num_slave{1}));
end
title('Giroscopio - Asse X (Roll)', 'FontWeight', 'bold');
ylabel('Giro X (dps)');
ylim([-60, 60]); % <--- SCALA FISSA BLOCCATA A ±60 dps
grid on; grid minor;
legend('show', 'Location', 'best');
```

I grafici IMU riportati nelle figure 15 e 16 corrispondono rispettivamente alle due prove illustrate nelle figure 13 e 14. L'andamento delle velocità angolari è coerente con il movimento descritto dalle posizioni dei servomotori nei grafici precedenti.

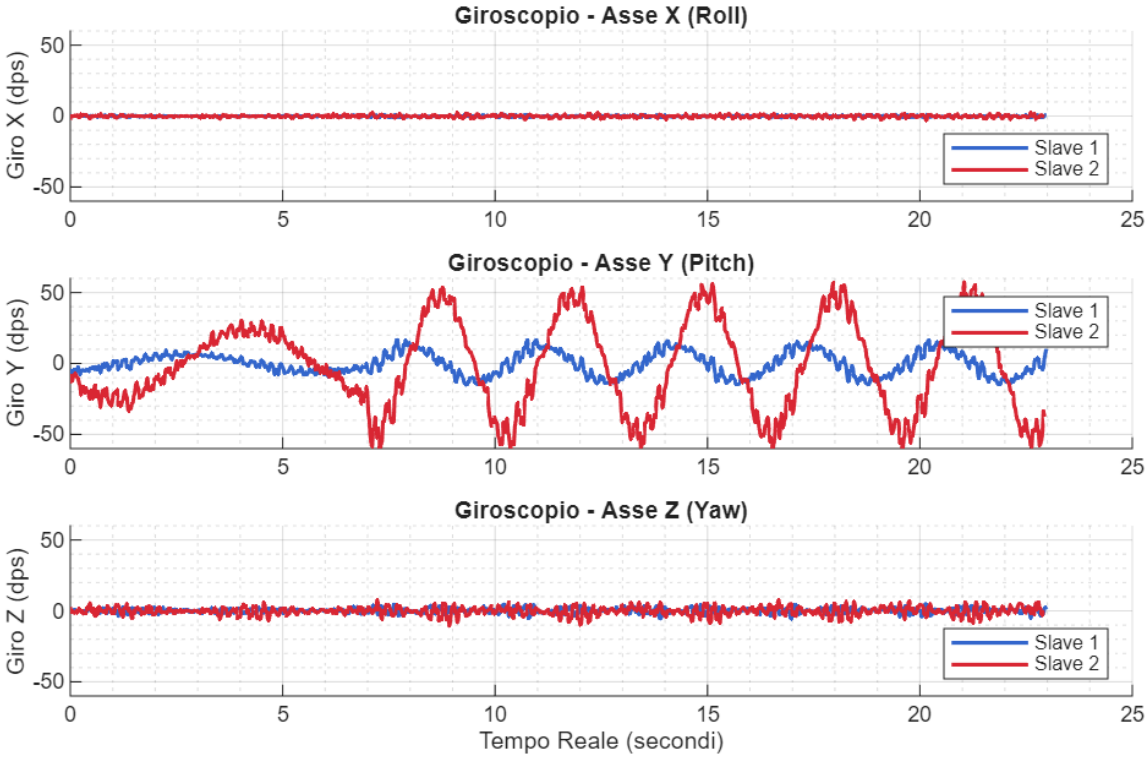



Figura 15: Dati IMU dei due nodi nella prova con i parametri predefiniti della sinusoide.

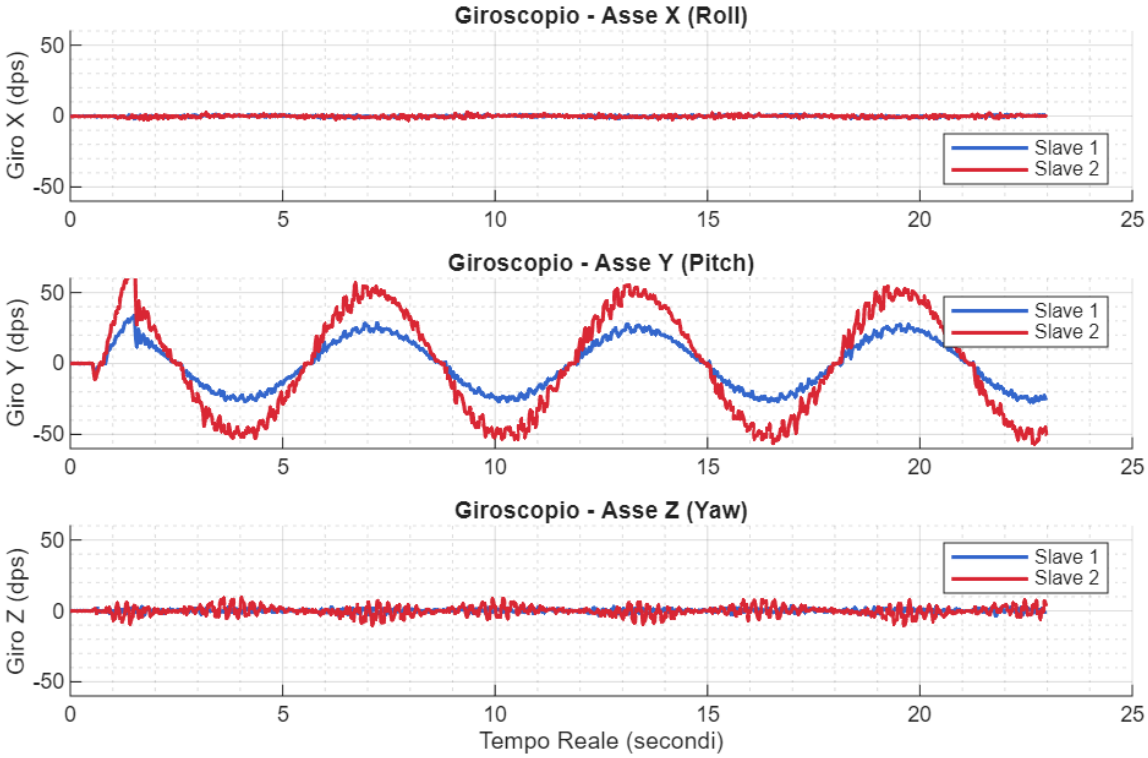



Figura 16: Dati IMU dei due nodi nella prova con parametri identici e a bassa frequenza.

### 5.3 Risultati dei test

Come si evince dai grafici di validazione presentati nella sezione precedente, la campagna sperimentale riportata è stata condotta utilizzando due soli servomotori, identificati come Servo 1 e Servo 2, sebbene l'architettura propulsiva completa del robot preveda tre giunti articolati. Durante il periodo dedicato alle prove di comunicazione e controllo, il terzo attuatore non era collegato al banco di prova, poiché veniva utilizzato contemporaneamente dal team meccanico come modello di riferimento dimensionale per la progettazione e la realizzazione dei supporti di fissaggio al telaio.

L'assenza del terzo nodo non ha tuttavia impedito la validazione dell'infrastruttura di controllo. L'impiego di due servomotori ha permesso di testare le principali funzionalità dell'architettura sviluppata, verificando:

- la stabilità della trasmissione CAN e il corretto indirizzamento dei pacchetti nella logica master–slave;
- l'efficacia del comando broadcast di sincronizzazione (`SYNC`) per l'avvio simultaneo delle traiettorie;
- il mantenimento dello sfasamento angolare tra segmenti diversi, principio fondamentale per la generazione dell'onda propulsiva biomimetica;
- la capacità del bus e dei buffer di gestire l'acquisizione parallela della telemetria di più nodi (stream dei registri e dati IMU) senza saturazione o conflitti.

I test condotti su due nodi costituiscono quindi una verifica rappresentativa del codice e del sistema di comunicazione. L'architettura è predisposta per l'integrazione del terzo segmento senza modifiche alla struttura software.

## 6 Conclusioni

Durante i test è emersa la necessità di collegare il modulo INA226-1A alla linea di alimentazione della batteria centrale. Nella configurazione di laboratorio, infatti, l'alimentazione del blocco era fornita attraverso l'Atom master, utilizzando il collegamento Ethernet con la base PoE oppure la connessione USB al PC impiegata per la comunicazione seriale. Questa alimentazione non era sufficiente per tutti i dispositivi collegati. Il collegamento alla batteria centrale rappresenta quindi una modifica da implementare nelle successive fasi del progetto.

Anche sul modulo 2Relay si è riscontrato un problema simile. La versione utilizzata supporta una corrente massima di 8 A per canale e poiché un singolo relè comanda l'alimentazione dell'intera coda, risulta sottodimensionato per gestire le correnti di picco: la somma dei tre attuatori in stallo raggiunge infatti circa 20 A. In questa fase prototipale il componente è sufficiente per i test a basso carico, ma come sviluppo futuro sarà fondamentale riprogettare l'intero percorso di potenza. Sarà necessario adottare sistemi di commutazione (relè di potenza o interruttori a stato solido) dimensionati correttamente per questi carichi in corrente continua e integrare adeguate protezioni contro i transitori e i sovraccarichi.

Nonostante questo limite, è stato possibile verificare il funzionamento del sensore e del relativo codice di acquisizione e stampa, mantenendo però il relè disattivato. I dati raccolti in queste condizioni non sono rappresentativi del funzionamento del robot a regime: le prove hanno consentito soltanto una validazione funzionale del sistema di misura, senza permettere una valutazione degli assorbimenti durante il movimento della coda.

In conclusione, nonostante la necessità di futuri adeguamenti, l'obiettivo primario del progetto di tesi è stato ampiamente esplorato e approfondito. L'architettura di controllo propulsivo del pesce robotico è stata infatti riprogettata, affrontando la transizione dallo standard $I^2C$ al protocollo CAN bus. L'infrastruttura firmware ha permesso di verificare sul banco di prova la gestione affidabile sia della trasmissione delle traiettorie per la locomozione biomimetica, sia dell'acquisizione parallela della telemetria diagnostica. Le basi poste da questo lavoro definiscono un'architettura di comunicazione decentralizzata e scalabile, fornendo un solido punto di partenza per l'integrazione a bordo del veicolo e le successive validazioni idrodinamiche.

## 6.1 Sviluppi futuri

In vista dell'integrazione del sistema a bordo del veicolo per lo svolgimento dei test idrodinamici, l'architettura richiederà alcuni adeguamenti hardware e software emersi durante le verifiche:

- **Completamento del sistema e monitoraggio dell'errore**: sarà necessario integrare il terzo attuatore, ripristinare la corretta terminazione del bus CAN agli estremi della linea (due resistenze da 120 Ω, equivalenti a 60 Ω in parallelo) e modificare i registri dello stream dati affinché trasmettano anche la posizione target. In questo modo sarà possibile calcolare quantitativamente l'errore RMS di inseguimento durante il funzionamento sotto carico.
- **Dimensionamento dell'hardware di potenza**: il campo di misura corrente dell'INA226-1A, pari a 1 A, e il modulo 2Relay, limitato a 8 A per canale, non sono adeguati ai picchi di assorbimento dei tre servomotori. Si prevede di sostituirli rispettivamente con sensori a effetto Hall e interruttori a stato solido dimensionati con margine per correnti superiori a 20 A, oltre a integrare protezioni adeguate.

- **Sincronizzazione e accesso condiviso ai dati IMU**: attualmente l'acquisizione inerziale è gestita sequenzialmente nel ciclo principale. Per ottenere un campionamento deterministico e indipendente dal traffico CAN, il firmware sarà ottimizzato sfruttando FreeRTOS, il sistema operativo in tempo reale integrato nell'ESP32. Sarà creato un task dedicato alla lettura dell'IMU ad alta frequenza; i campioni saranno depositati in un'area di memoria condivisa e protetta da mutex, garantendo ai task di controllo e di registrazione un accesso sicuro e aggiornato.

# Bibliografia


1. Hitec Commercial Solutions, *Documentazione dei servomotori Hitec MDB961WP-CAN-28V.*
   https://www.hiteccs.com/actuators/product-details/MDB961WP-CAN-28V/

2. M5Stack, *Documentazione di AtomS3R.*
   https://docs.m5stack.com/en/core/AtomS3R

3. M5Stack, *Documentazione di Atomic PoE Base.*
   https://docs.m5stack.com/en/atom/Atomic%20PoE%20Base

4. M5Stack, *Documentazione di Unit Mini CAN.*
   https://docs.m5stack.com/en/unit/Unit-Mini%20CAN

5. M5Stack, *Documentazione della libreria M5Unified.*
   https://docs.m5stack.com/en/arduino/m5unified/helloworld

6. M5Stack, *Documentazione di Unit INA226-1A.*
   https://docs.m5stack.com/en/unit/Unit_INA226-1A

7. M5Stack, *Documentazione di Unit 2Relay.*
   https://docs.m5stack.com/en/unit/Unit_2Relay

8. Espressif Systems, *Documentazione del driver Two-Wire Automotive Interface (TWAI).*
   https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/twai.html

9. Blue Trail Engineering, *Documentazione del contenitore subacqueo SER-200X (Underwater Servo Kit).*
   https://www.bluetrailengineering.com/product-page/underwater-servo-ser-200x

# Ringraziamenti

Desidero esprimere la mia gratitudine al mio relatore, il Prof. David Scaradozzi, per avermi concesso l'opportunità di sviluppare questo elaborato e per la competenza, la supervisione e la costante guida metodologica fornite durante l'intero periodo di stesura del lavoro. I suoi consigli sono stati determinanti per indirizzare al meglio le scelte progettuali.

Un ringraziamento doveroso va a tutto il gruppo di ricerca del laboratorio LabMACS, che ha messo a disposizione un ambiente di lavoro stimolante e professionale. In particolare, desidero ringraziare Luca Marchegiani e Flavia Gioiello, con i quali ho collaborato a stretto contatto nelle attività quotidiane. Il loro indispensabile supporto tecnico, sia nella fase di sviluppo hardware e firmware che durante le sessioni di test e validazione in laboratorio, unito al confronto continuo e costruttivo, si è rivelato fondamentale per il superamento delle criticità pratiche e per la corretta realizzazione di questo progetto.