UNIVERSITÀ degli STUDI di CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

PROGETTO PROGRAMMAZIONE DI SISTEMI ROBOTICI AUTONOMI E LABORATORIO

Davide Scalisi

Luigi Seminara

Phisical Cart 2D

PROGETTAZIONE DI UN SISTEMA ROBOTICO HARDWARE E SOFTWARE PER IL CONTROLLO DI UN CART 2D A DUE RUOTE ED A DUE ENCODER ROTATIVI.

Documentazione e progetto sono stati realizzati da Davide Scalisi (1000038316) e Luigi Seminara (1000037583) per poter conseguire l'esame di Programmazione di Sistemi Robotici autonomi e laboratorio del corso di laurea magistrale in informatica dell'anno accademico 2021/2022 con il docente Corrado Santoro.

ANNO ACCADEMICO 2021/2022

Sommario

Introduzione 1

Repository GitHub del progetto 1

Tecnologie utilizzate 1

Hardware 2

Schema funzionale dell'hardware 2

Funzionamento dei due encoder 2

Attenuazione in tensione dei due segnali A e B 3

Funzionamento dei ponti ad H 4

Interfacciamento e debug 4

Schema elettrico della scheda Arduino Nano 5

Schema elettrico della scheda dei ponti ad H 5

Firmware Arduino Nano 6

Interfacciamento con gli encoder 6

Funzionamento della classe RI32 7

Classe RI32 8

Interfacciamento con i motori 9

Modalità PWM in correzione di fase 10

Classe LMD18200 11

Schema di controllo in velocità 12

Classe SpeedController 13

Schema di controllo in posizione 13

Debug tramite grafici 14

Loop di controllo principale 16

Protocollo di comunicazione binario tra i due MCU 17

Conclusioni sul firmware Arduino Nano 19

Firmware ESP32 20

Gestione interna del Fixed Graph 20

Trasmissione delle coordinate del percorso 20

Interfaccia utente 21

Salvataggio su filesystem LittleFS 22

Realizzazione finale del robot fisico 23

Conclusioni 25

Introduzione

Lo scopo di questo progetto è quello di realizzare un sistema robotico hardware e software in grado di interfacciarsi e controllare un carrellino a due ruote (cart 2D) al fine di eseguire un algoritmo di controllo e di collision avoidance che sia in grado di far muovere tale carrellino da un punto di partenza ad un punto di arrivo evitando una serie di ostacoli virtuali prefissati.

Repository GitHub del progetto

Tecnologie utilizzate

Il progetto si sviluppa in due principali ambiti e per ognuno sono state utilizzate le seguenti tecnologie ed i seguenti software di sviluppo:

- **Hardware**
  - Simulatore di circuiti Falstad.
  - La suite EDA KiCad.
  - Schede di sviluppo Arduino Nano ed ESP32-Cam.
  - Modulo Bluetooth HC-05 (serial over Bluetooth)
  - 2x ponte ad H singolo LMD18200.
  - 2x encoder rotativi 1000 ticks RI-32.
  - Batteria ai polimeri di litio 11.1V (alimentazione logica).
  - Batteria al piombo 12V (alimentazione motori).

- **Firmware**
  - Linguaggio di programmazione C++.
  - IDE VS Code + PlatformIO.
  - Framework Arduino.
  - Sistema operativo real-time FreeRTOS.
  - Software SerialPlot.

Hardware

Il primo passo per lo sviluppo di questo progetto è stato quello di progettare l'hardware necessario al pilotaggio dei due motori ed alla lettura dei due segnali in quadratura (sfasati di 90° tra di loro) in uscita dai due encoder.

Schema funzionale dell'hardware

Più nello specifico, si può riassumere il funzionamento logico della parte hardware del sistema robotico tramite la seguente figura:

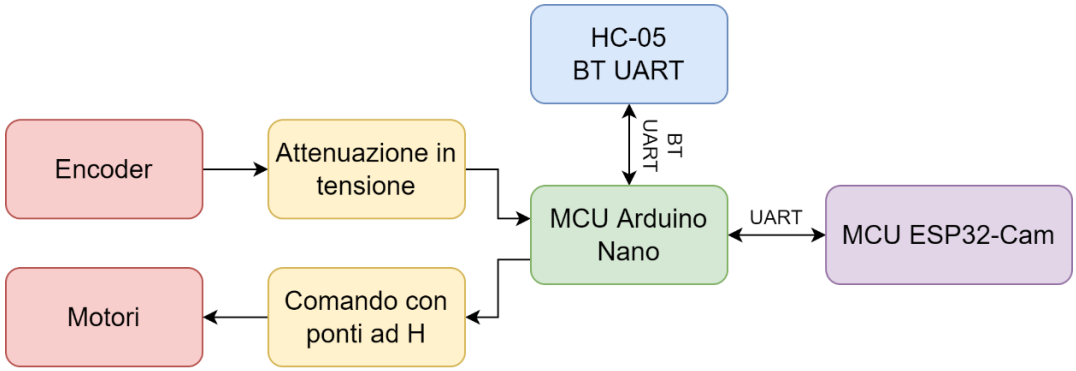

Figura 1: schema funzionale dell'hardware dell'intero progetto.

Funzionamento dei due encoder

I due encoder rotativi RI-32 montati alla destra ed alla sinistra del carrellino del robot costituiscono la parte hardware fondamentale per la realizzazione di un qualsiasi sistema robotico autonomo retroazionato; il compito di questi due componenti è quello di leggere la distanza percorsa nel corso del movimento del robot.

Per ognuno di questi due componenti, vengono dati in uscita due segnali ad onda quadra in quadratura tra di loro, i quali possono facilmente essere acquisiti dalla maggioranza dei microcontrollori in circolazione.

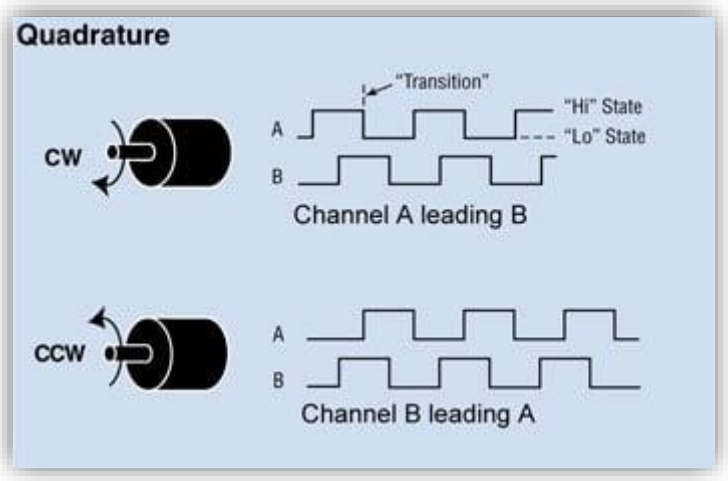

Figura 2: esempio di due segnali in quadratura in uscita da un encoder rotativo.

L’acquisizione e l’elaborazione di questa tipologia di segnali di natura digitale è molto semplice, in quanto basta fare uso della classica tecnica di interrupt hardware su GPIO su uno solo dei due segnali (ad esempio, solamente sul segnale “A”, in figura 2) per poi andare a leggere all’interno della ISR stessa lo stato dell’altro segnale (del canale “B”, in figura 2).

Tramite questo meccanismo è quindi possibile relazionare i due segnali appena acquisiti al verso del moto rotazionale che la ruota di misura in questione sta subendo (orario o antiorario); per quanto riguarda la misura dello spazio percorso, successivamente si vedrà come tradurre la serie di impulsi del segnale stesso in una variazione di posizione.

Attenuazione in tensione dei due segnali A e B

Come suggerito dal datasheet degli encoder RI-32 utilizzati in questo progetto, è fortemente consigliata un’alimentazione intorno ai 12V; questo aspetto necessita quindi di particolare attenzione in quanto i due segnali A e B, se collegati direttamente all’MCU, andrebbero a danneggiare i GPIO dell’MCU stesso, il quale è un ATmega328 operante ad una tensione nominale di 5V.

Per permettere quindi un acquisizione digitale sicura, occorre eseguire uno shift dei due segnali dal range di partenza a quello di arrivo:

$$[0V, \sim 12V] \rightarrow [0V, 5V]$$

Essendo i segnali di tipo unidirezionale e push-pull, il circuito di adattamento si riduce a due semplici partitori resistivi per encoder:

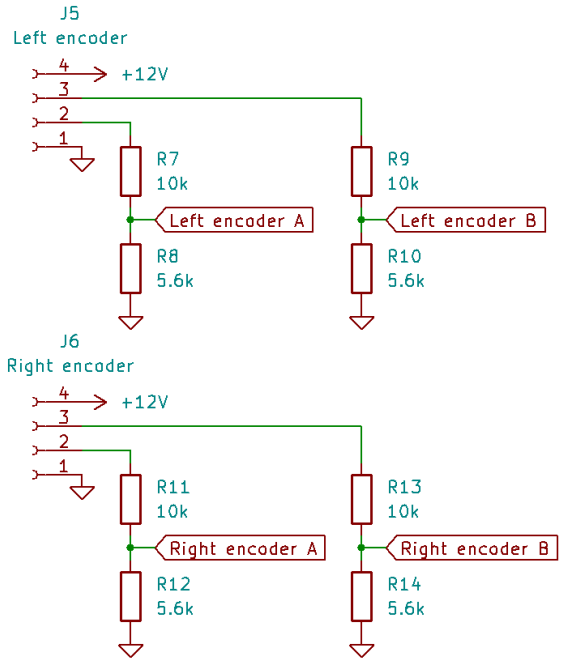

Figura 3: circuito di adattamento in tensione per due coppie di segnali in quadratura (in totale, quattro segnali da leggere).

Successivamente si osserverà come i segnali “Left encoder A” e “Right encoder A” sono connessi agli unici due GPIO dell’ATmega328 che supportano la funzionalità di interrupt hardware.

Funzionamento dei ponti ad H

Per quanto riguarda la parte di controllo motori, l'adattamento in tensione ed in potenza dei segnali PWM e di direzione generati dall'MCU è effettuato tramite due integrati LMD18200; ognuno di questi integrati contiene al suo interno un ponte ad H, cioè un circuito elettrico atto a convertire un segnale PWM ed uno di direzione provenienti da un MCU in un apposito segnale PWM, adattato in tensione e potenza, pronto per essere immesso direttamente nel motore.

L'LMD18200, in particolare, ha un utilizzo molto semplice ed ha un'implementazione interna sia dei dead-time tra i segnali di controllo degli interruttori di potenza, sia del bootstrap degli interruttori alti; questo integrato, per poter funzionare correttamente in maniera minimale, richiede quindi al suo esterno due condensatori di bootstrap per le pompe di carica e delle resistenze di pull-down sugli ingressi PWM e direzione.

Ognuno dei due integrati ha il compito di manovrare il suo rispettivo motore, in modo tale da permettere al carrellino un movimento controllato sia in velocità che in direzione; l'informazione viene trasmessa ai due ponti ad H tramite due segnali PWM e due segnali direzione, per un totale di quattro segnali di controllo:

- **Direzione**:
  - **LOW**: avanti.
  - **HIGH**: indietro.

- **PWM**: in base al duty cycle del segnale stesso, viene determinata la velocità di rotazione del motore, in percentuale dallo 0% al 100%.

Le resistenze di pull-down discusse prima sono necessarie in quanto al riavvio dell'MCU (conseguenza diretta di una singola riprogrammazione), tutti i GPIO si trovano in alta impedenza, cosa che in presenza di interferenze elettriche può portare i due motori a muoversi senza che alcun comando di movimento sia stato effettivamente dato.

Interfacciamento e debug

Per lo sviluppo del sistema software di controllo, si è deciso di progettare un sistema basato su due architetture hardware completamente differenti, alle quali sono stati assegnati i seguenti task:

- **Arduino Nano**: dall'interfacciamento a basso livello al controller in posizione.
- **ESP32-Cam**: dal controller in posizione fino alla collision avoidance e front-end.

Per lo sviluppo del firmware risiedente su Arduino Nano, si è scelto di utilizzare una seriale su bluetooth per la riprogrammazione del target, emulando di fatto un vero e proprio dispositivo USB connesso al computer.

Questa seriale su bluetooth si è rivelata molto utile anche per scopi di debug: è stata infatti realizzata una libreria in grado di eseguire il plot di dati su seriale, in modo tale da verificare tutte le varie curve di risposta e di comando.

Al contrario, per l'ESP32, è stata utilizzata la libreria AsyncElegantOTA, la quale permette di aggiornare di volta in volta il firmware stesso dell'ESP32 tramite OTA WiFi.

La comunicazione tra i due microcontrollori avviene anche tramite UART, con la sola differenza che dal lato ESP32 è stata direttamente utilizzata una seriale hardware, mentre dal lato Arduino Nano è stato necessario utilizzare la libreria NeoSWSerial, la quale si occupa di designare due qualsiasi GPIO per fargli assumere il funzionamento di seriale via software, ovviamente con le dovute limitazioni del caso.

Schema elettrico della scheda Arduino Nano

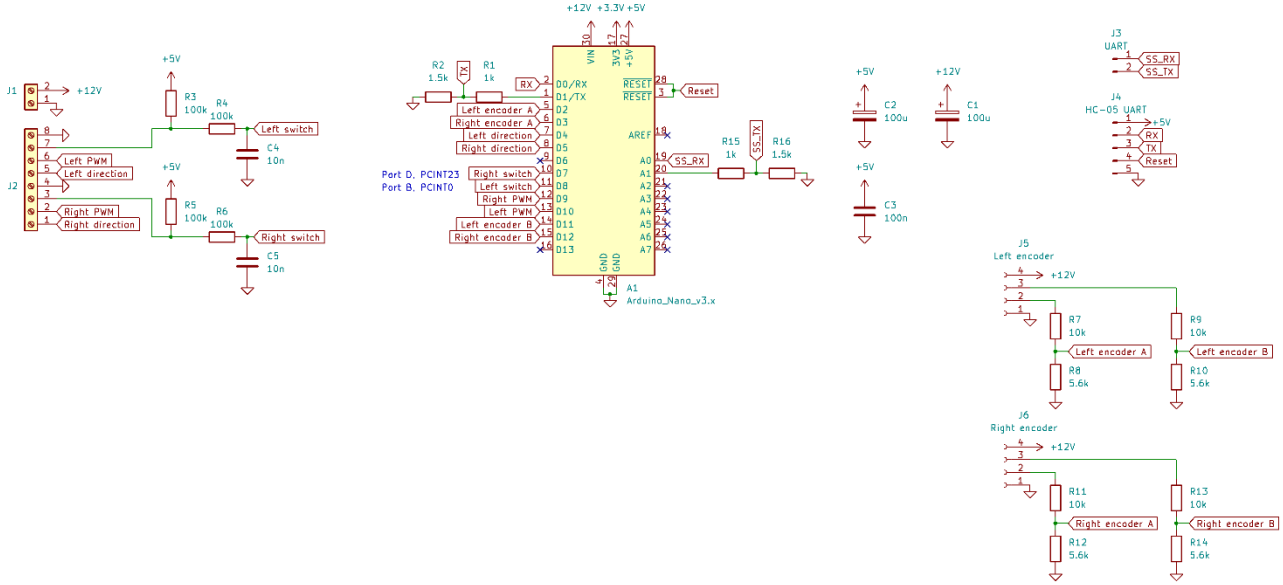

Figura 4: schema elettrico della scheda Arduino Nano.

Schema elettrico della scheda dei ponti ad H

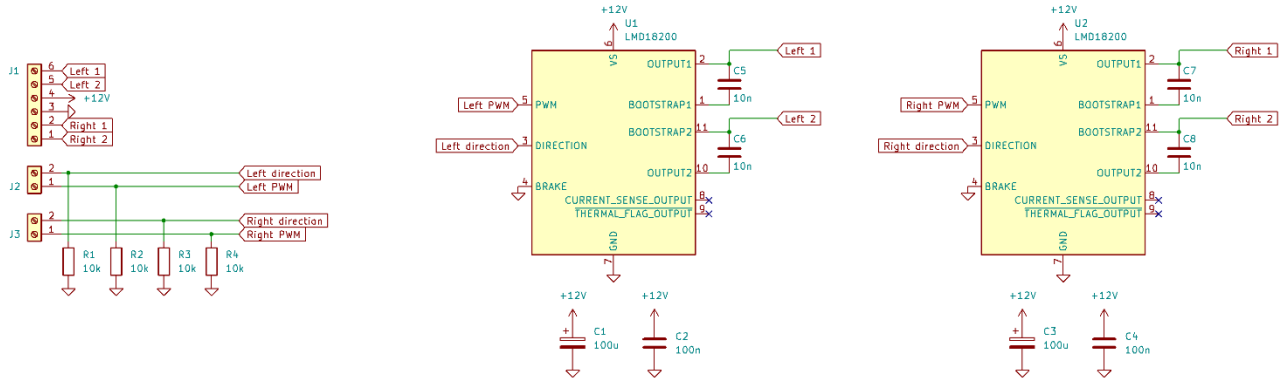

Figura 5: schema elettrico della scheda dei ponti ad H.

Firmware Arduino Nano

In questo capitolo verrà trattata l'effettiva progettazione ed il funzionamento dell'intero sistema di controllo a basso livello che è stato programmato all'interno del firmware di Arduino Nano.

Come accennato precedentemente, si è iniziato in primo luogo a studiare il problema partendo dall'interfacciamento a basso livello con gli encoder e con i motori per poi arrivare a ricavare la struttura logica vera e propria dell'intero sistema di controllo; questo è stato fatto soprattutto andando a studiare e ad applicare le leggi cinematiche che regolano il moto 2D di questa tipologia di carrellino.

Interfacciamento con gli encoder

Come spiegato nel capitolo precedente, tramite i segnali in quadratura in uscita dai due encoder, per ognuna delle due ruote di misura è possibile capire facilmente lo spazio percorso e la direzione verso la quale il carrellino si sta muovendo.

Per ricavare queste due informazioni, l'operazione più semplice è quella di assegnare ai due segnali "A" provenienti dai due encoder, gli unici due GPIO dell'ATmega328 che supportano nativamente la funzionalità di interrupt, cioè i GPIO `2` e `3`:

```
attachInterrupt(digitalPinToInterrupt(pin_l_a), ENC_L_ISR, RISING);
attachInterrupt(digitalPinToInterrupt(pin_r_a), ENC_R_ISR, RISING);

void ENC_L_ISR(){
  digitalRead(RI32::pin_l_b) ? RI32::dTicks_l-- : RI32::dTicks_l++;
}

void ENC_R_ISR(){
  digitalRead(RI32::pin_r_b) ? RI32::dTicks_r++ : RI32::dTicks_r--;
}
```

Come si può osservare facilmente dallo snippet di codice riportato sopra, le due chiamate ad `attachInterrupt` servono a legare all'evento di `RISING` sui pin `2` e `3`, rispettivamente le ISR `ENC_L_ISR` ed `ENC_R_ISR`.

Una volta effettuata questa chiamata, al verificarsi del suddetto evento sui rispettivi GPIO, verranno a loro volta invocate nel software le due procedure specificate sotto forma di ISR e, proprio per questa ragione, il codice al loro interno deve essere il quanto più veloce possibile.

Come si può notare, il codice all'interno delle due procedure ha il solo ed unico compito di incrementare o decrementare una variabile statica (`dTicks_l` oppure `dTicks_r`) all'interno della classe `RI32`, la quale conterrà effettivamente tutto il codice atto ad elaborare questi due conteggi.

Funzionamento della classe RI32

La classe RI32 è la classe vera e propria dove sono codificate le formule di odometria, le quali servono per la vera e propria conversione dei tick degli encoder nell'attuale posa del robot (x, y, θ).

La discretizzazione avviene ad istanti di tempo fissati ed ugualmente spaziati; è stato quindi scelto un periodo di campionamento di $h = 8e - 3$, cioè una frequenza di campionamento pari a $125Hz$.

Le formule utilizzate nei vari passaggi di conversione sono state le seguenti:

1) Calcolo di $\Delta\theta$:

$$\Delta\theta_{left} = \frac{2\pi \cdot \Delta tick_{left}}{ticks}, \qquad \Delta\theta_{right} = \frac{2\pi \cdot \Delta tick_{right}}{ticks}$$

Dove:

- $\Delta\theta$ è l'arco d'angolo percorso per ruota tra l'istante h e l'istante $h + 1$.
- $ticks = 1000$ è pari al numero di giri che l'encoder produce (a datasheet).
- $\Delta tick$ è pari al numero di tick contati per ruota tra l'istante h e l'istante $h + 1$ (`dTicks_l`/`r`).

2) Calcolo di Δs:

$$\Delta s_{left} = \Delta\theta_{left} \cdot r_{enc}, \qquad \Delta s_{right} = \Delta\theta_{right} \cdot r_{enc}$$

Dove:

- Δs è lo spazio percorso per ruota tra l'istante h e l'istante $h + 1$.
- $r_{enc} = 0.021m$ è il raggio in metri delle due ruote di misura.

Per il successivo calcolo dello spazio percorso per ruota tra l'istante h e l'istante $h + 1$, basterà accumulare nel tempo Δs; questo spazio percorso verrà identificato con $s_{left/right}$.

3) Calcolo di Δv:

$$\Delta v_{left} = \frac{\Delta s_{left}}{h}, \qquad \Delta v_{left} = \frac{\Delta s_{left}}{h}$$

Dove:

- Δv è la velocità istantanea per ruota.
- $h = 8e - 3$, come già discusso, rappresenta il periodo di campionamento dei dati.

4) Calcolo di Δs e $\Delta\theta$:

$$\Delta s = \frac{\Delta s_{left} + \Delta s_{right}}{2}, \qquad \Delta\theta = \frac{\Delta s_{right} - \Delta s_{left}}{B_{enc}}$$

Dove $B_{enc} = 0.305m$ è la wheelbase delle due ruote di misura (la loro distanza).

5) Calcolo di v e ω:

$$v = \frac{v_{left} + v_{right}}{2}, \qquad \omega = \frac{v_{right} - v_{left}}{B_{enc}}$$

6) Calcolo della posa (x, y, θ):

$$\begin{cases} x = x + \Delta s \cdot \cos\left(\theta + \frac{\Delta\theta}{2}\right) \\ y = y + \Delta s \cdot \sin\left(\theta + \frac{\Delta\theta}{2}\right) \\ \theta = normalize(\theta + \Delta\theta) \end{cases}$$

Dove:

- Nel calcolo di x ed y, il θ utilizzato è relativo all'istante h e non all'istante $h + 1$, come si potrebbe erroneamente pensare.

- $normalize(\alpha)$ è una funzione di normalizzazione angolare, la quale aggiunge o sottrae all'angolo α in questione π fino a quando il suddetto α non ricade nel range $[-\pi, \pi]$.

Classe RI32

Le formule trattate sono riportate all'interno del metodo `void RI32::evaluate()` il quale, per il corretto funzionamento delle formule stesse, deve essere richiamato ogni h frazioni secondo:

```
void RI32::evaluate(){
  //Left and right cinematic variables.
  //From the last evaluation, dTicks contains the counted ticks.
  float dTheta_l = (2*PI*dTicks_l) / enc_ticks;
  float dTheta_r = (2*PI*dTicks_r) / enc_ticks;

  //From the next interrupt, reset dTicks and start counting again.
  dTicks_l = dTicks_r = 0;

  float ds_l = dTheta_l * enc_radius;
  float ds_r = dTheta_r * enc_radius;

  //Left and right speed.
  v_l = ds_l / dt;
  v_r = ds_r / dt;

  float ds = (ds_l + ds_r) / 2;
  float dTheta = (ds_r - ds_l) / enc_wheelbase;

  //Generic cinematic variables.
  v = (v_l + v_r) / 2;
  omega = (v_r - v_l) / enc_wheelbase;

  x += ds * cos(theta + dTheta/2);
  y += ds * sin(theta + dTheta/2);
  theta = normalize_angle(theta + dTheta);
}
```

Interfacciamento con i motori

L'interfacciamento con i motori è stato realizzato tramite l'utilizzo di due GPIO per ponte ad H, per un totale di quattro GPIO.

I due GPIO delegati al segnale di direzione sono stati scelti senza particolari restrizioni, mentre per quanto riguarda i due GPIO utilizzati per generare i due segnali PWM, la scelta è obbligata nell'uso delle due porte **OC1A** ed **OC1B**, cioè dei pin 9 e 10.

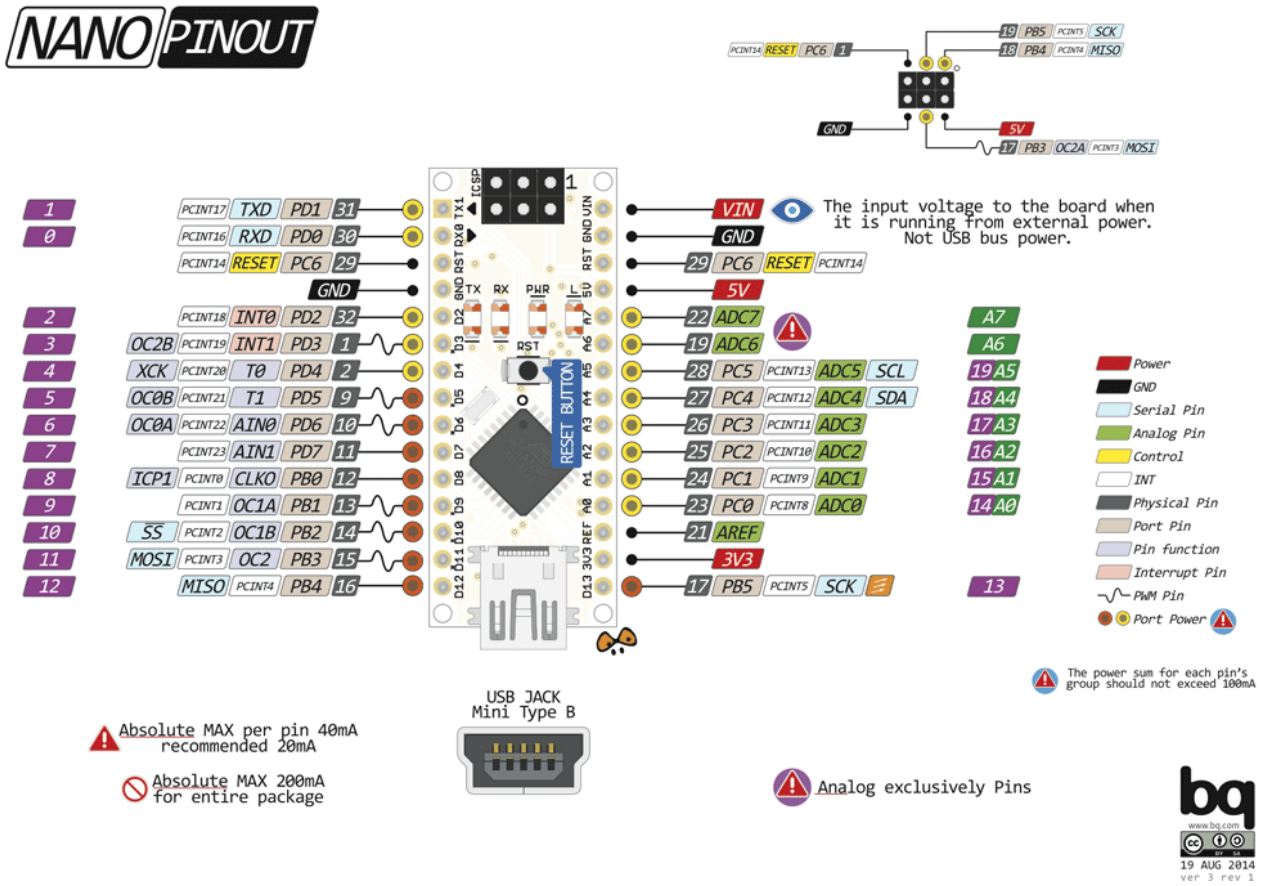

Figura 6: pinout standard di Arduino Nano.

Il motivo sotto l'utilizzo di questi due specifici pin è stata la conseguenza della scelta dell'utilizzo del Timer1 per la generazione dei due voluti segnali PWM.

Il Timer1 è l'unico timer hardware a 16 bit dell'ATmega328 ed è stato quindi scelto per gestire le tempistiche di generazione dei due segnali PWM tramite la modalità di funzionamento chiamata PWM in correzione di fase.

Modalità PWM in correzione di fase

Questa modalità di funzionamento in output compare del Timer1 a 16 bit prevede una modalità di controllo a doppia pendenza:

Figure 13-8. Phase Correct PWM Mode, Timing Diagram

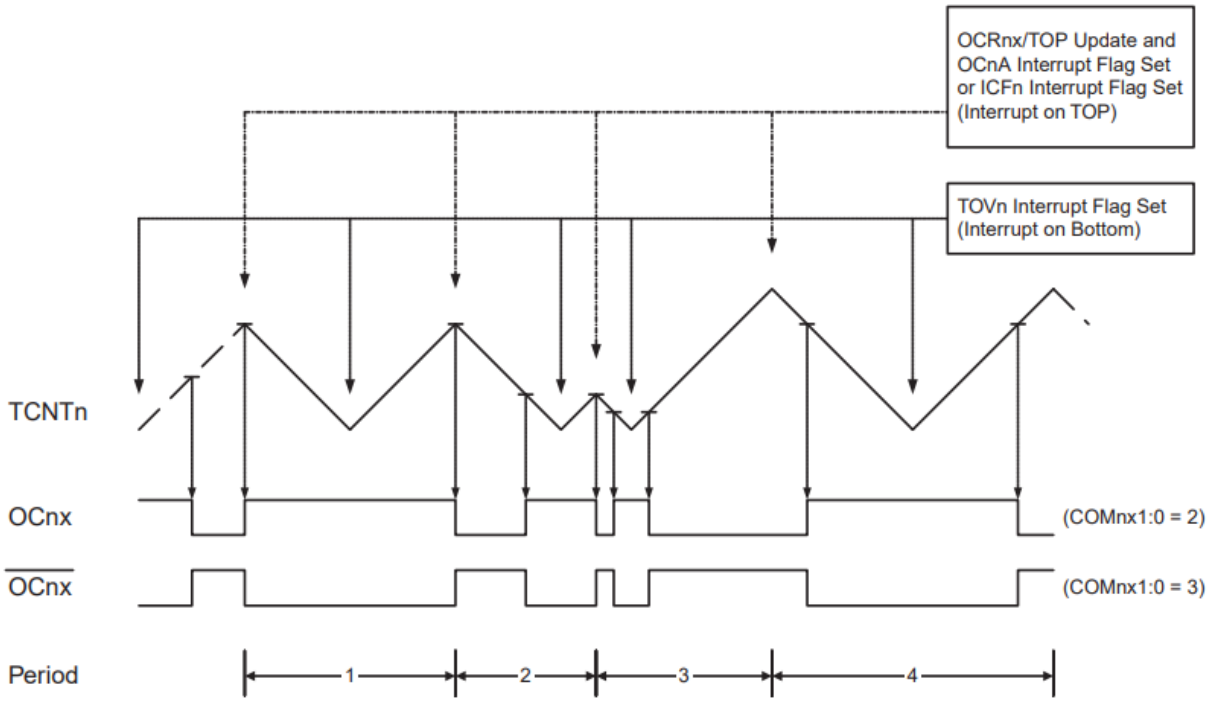

Figura 7: dal datasheet dell'ATmega328, pag.129: grafici di funzionamento della modalità PWM in correzione di fase.

La modalità di controllo a doppia pendenza è data dal contatore **TCNT1** il quale:

- Viene decrementato di un'unità per tick una volta raggiunto il **TOP**.
- Viene incrementato di un'unità per tick una volta raggiunto il **BOTTOM**.

Come si può vedere dal grafico in figura 7, il comportamento dei due pin **OC1A/B** è correlato direttamente al compare match tra i valori contenuti nel registro **TCNT1** ed i due rispettivi registri per porta **OCR1A/B**: una volta che l'uguaglianza si verifica, si avrà un cambio di stato del corrispettivo pin associato **OC1A/B** in relazione con altri bit di configurazione in altri specifici registri.

Tramite questa modalità di controllo, disegnata appositamente per il controllo motori, è possibile ottenere un segnale PWM con un duty cycle anche altamente variabile esente da qualsiasi tipo di artefatto semplicemente agendo sui registri di comparazione **OC1A/B**.

Nell'applicazione corrente, la modalità di funzionamento è impostata come segue:

- $(OCR1A|B = TCNT1) \cap upcounting \rightarrow OC1A|B = HIGH$
- $(OCR1A|B = TCNT1) \cap downcounting \rightarrow OC1A|B = LOW$

La frequenza del segnale PWM è invece stabilita in base all'applicazione: in questo caso, dovendo pilotare due motori in corrente continua, la frequenza scelta è stata di $50kHz$.

Classe LMD18200

Tutte le impostazioni di configurazione del Timer1 sono contenute all'interno del metodo `void LMD18200::begin()`, mentre la configurazione dei bit di gestione del modo di funzionamento dei pin **OC1A/B** è contenuta direttamente all'interno dei due metodi:

- `void LMD18200::__start(bool left, bool right)`
- `void LMD18200::__stop(bool left, bool right)`

```
void LMD18200::begin(){
  ICR1 = 160;
  TCCR1A = (1 << WGM11);
  TCCR1B = (1 << WGM13) | (1 << CS10);
  OCR1A = OCR1B = 0;
}

void LMD18200::__start(bool left, bool right){
  //PB2
  if(left)
    TCCR1A |= (1 << COM1B1);
  //PB1
  if(right)
    TCCR1A |= (1 << COM1A1);
}

void LMD18200::__stop(bool left, bool right){
  //PB2
  if(left){
    TCCR1A &= (uint8_t) ~(1 << COM1B1);
    DDRB |= (1 << DDB2);
  }
  //PB1
  if(right){
    TCCR1A &= (uint8_t) ~(1 << COM1A1);
    DDRB |= (1 << DDB1);
  }
}
```

Per quanto riguarda l'interfacciamento con l'esterno della classe LMD18200, la quale può essere definita come la classe di gestione vera e propria dei due motori, questa dispone direttamente dei due metodi:

- `void LMD18200::left(int16_t pwm_direction_speed)`
- `void LMD18200::right(int16_t pwm_direction_speed)`

I quali servono ad interpretare rispettivamente il segno ed il magnitudo dello spostamento riferendosi rispettivamente al segno ed al valore assoluto di `pwm_direction_speed` ed impostando di conseguenza il motore destro o sinistro ad una percentuale di potenza dallo 0% al 100% nella direzione specificata.

Schema di controllo in velocità

Come primo schema di controllo del robot, la scelta è ricaduta in un controllo in velocità: in questo schema vengono dati in input due velocità target v ed ω, rispettivamente velocità lineare ed angolare, e lo scopo del carrellino è quello di raggiungere e mantenere nel tempo tali velocità.

Lo schema finale risultante dallo studio del problema è stato il seguente:

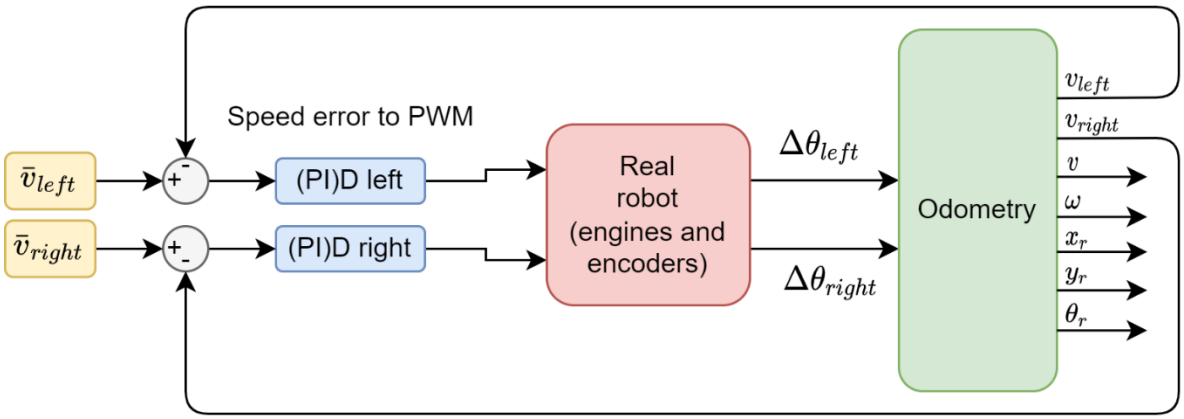

Figura 8: schema di controllo in velocità del carrellino.

Dallo studio di questa tipologia di carrellino fisico a due ruote motrici, essendo le due velocità v ed ω indipendenti, è possibile utilizzare due PID tarati in maniera analoga per il controllo sia della ruota destra che della ruota sinistra.

NB: in questa immagine non è rappresentato a monte un blocco di conversione $(v, \omega) \rightarrow (v_{left}, v_{right})$, in quanto si può decidere di effettuare direttamente un controllo in velocità date v_{left} ev_{right}; in tutti i casi, la conversione è di facile comprensione ed implementazione andando ad utilizzare direttamente alcune delle formule di odometria sopra trattate.

Classe SpeedController

L'intero schema di controllo è direttamente implementato all'interno della classe `SpeedController`, la quale tramite il metodo `void SpeedController::evaluate(…)` permette di specificare due velocità target v ed ω che il carrellino deve raggiungere.

Il metodo, per poter funzionare correttamente, deve essere chiamato ogni h frazioni di secondo in modo tale che le tempistiche dei PID su cui si basa l'algoritmo siano rispettate.

Per quanto riguarda il loop di controllo principale, il funzionamento corretto si ha nel momento in cui avviene prima la chiamata all'aggiornamento dell'odometria tramite `RI32::evaluate()` per poi andare a richiamare il controllo in velocità vero e proprio tramite il metodo `SpeedController::evaluate(…)`.

Tale metodo prende in input due velocità target (v, ω) oppure direttamente (v_{left}, v_{right}) in funzione di un altro terzo parametro booleano; il robot cercherà quindi di raggiungere e mantenere nel tempo questi due valori dati di velocità.

Schema di controllo in posizione

Una volta implementato correttamente il controllo in velocità, il passo successivo è stato quello di implementare il controllo in posizione vero e proprio; lo schema studiato ed utilizzato è stato quindi il seguente:

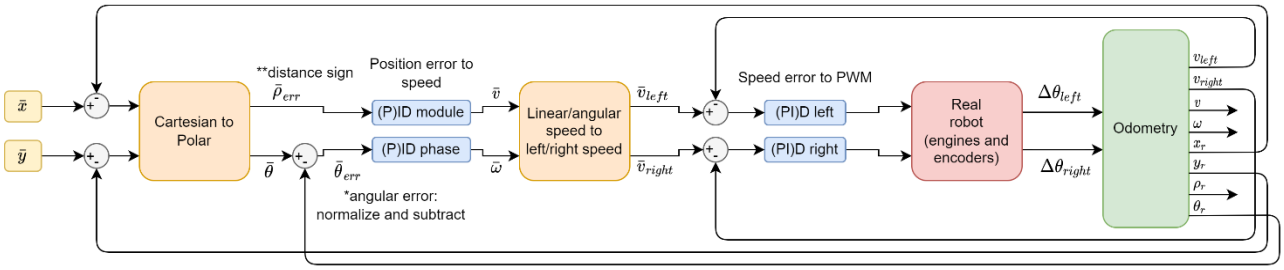

Figura 9: schema di controllo in posizione del carrellino.

Il funzionamento dello schema può essere riassunto nei seguenti punti chiave:

1) Rilevamento delle attuali coordinate (x, y) dagli encoder tramite odometria.
2) Calcolo errore in posizione $(x_r - \bar{x},\ y_r - \bar{y})$.
3) Conversione cartesiana-polare $(x_r - \bar{x},\ y_r - \bar{y}) \rightarrow (\bar{\rho}_{err},\ \bar{\theta})$.

4) A questo punto, l'errore in modulo $\bar{\rho}_{err}$ può essere direttamente utilizzato come errore per controllare un PID mentre per il calcolo di $\bar{\theta}_{err}$, bisogna necessariamente tener conto di θ_r per poi calcolare effettivamente l'errore di fase $\bar{\theta}_{err} = normalize(\bar{\theta} - \theta_r)$.

5) Se $\bar{\theta}_{err} < -\frac{\pi}{2} \ \cup \ \bar{\theta}_{err} > \frac{\pi}{2}$:
   - $\bar{\rho}_{err} = -\bar{\rho}_{err}$
   - $\bar{\theta}_{err} = normalize(\bar{\theta}_{err} + \pi)$

Questo caso particolare serve semplicemente a tener conto del fatto che il punto target $(\bar{x}, \bar{y})$ potrebbe anche trovarsi alle spalle del robot: in questo caso si agisce considerando il modulo dell'errore in posizione come una grandezza con segno mentre si ruota di 180° l'orientamento del robot stesso, andando a sommare π all'errore di orientamento $\bar{\theta}_{err}$.

6) Ottenuta quindi la coppia $(\bar{\rho}_{err},\ \bar{\theta}_{err})$, basta utilizzare i due PID chiamati PID module e PID phase per poter realizzare la conversione $(\bar{\rho}_{err},\ \bar{\theta}_{err}) \rightarrow (\bar{v}, \bar{\omega})$.

7) A questo punto, viene richiamato il controllo in velocità descritto precedentemente anteponendo la seguente conversione: $(\bar{v}, \bar{\omega}) \rightarrow \left(\bar{v}_{left},\ \bar{v}_{right}\right)$.

Per quanto riguarda il loop di controllo principale, analogamente al controllo in velocità, il funzionamento corretto si ha ancora una volta nel momento in cui avviene prima la chiamata all'aggiornamento dell'odometria tramite `RI32::evaluate()` per poi andare a richiamare il controllo in posizione vero e proprio tramite il metodo `void PositionController::evaluate(…);` questo metodo ha come parametri l'ascissa e l'ordinata target che il robot deve raggiungere.

Andando quindi a modificare tali coordinate all'interno del loop di controllo principale, si avrà che il robot seguirà con risposta più o meno decisa, dipendentemente dai valori di taratura dei PID, il cambiamento stesso delle coordinate.

Debug tramite grafici

Per poter effettuare il vero e proprio debug del codice è stato utilizzato il software SerialPlot, il quale permette la ricezione ed il plotting di dati binari provenienti da una porta seriale:

Figura 10: software SerialPlot.

Il software definisce varie impostazioni per la definizione del pacchetto dati binario di cui fare il plot:

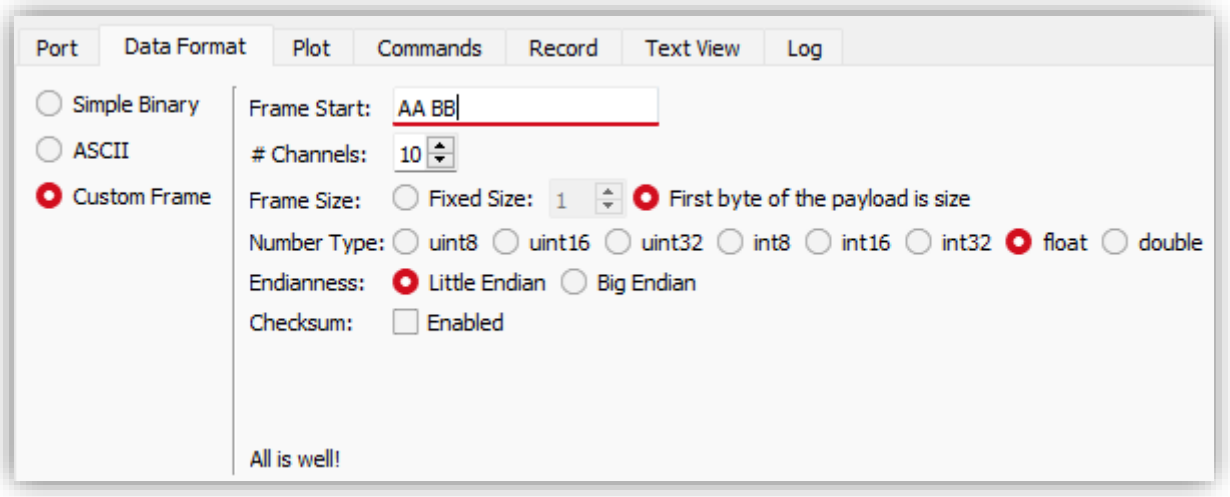

Figura 11: definizione del pacchetto binario.

Sulla base di questa configurazione, è stata realizzata la libreria SerialPlotter, all'interno della quale viene definito il seguente pacchetto dati binario:

```
struct SerialPlotter_frame_t{
  const uint16_t frame_start = 0xBBAA;

  uint8_t payload_length;
  T payload[PAYLOAD_MAXCHANNELS];
} data;
```

Inviando quindi i dati binati tramite l'uso di questa struttura dati di base, è possibile quindi interfacciarsi correttamente con il software SerialPlot il quale, previa corretta configurazione, provvederà ad eseguire graficamente il plot dei dati necessari al debug ed allo sviluppo del firmware.

Loop di controllo principale

```
void loop(){
  //Interrupt occurred.
  if(tick){
    tick = false;

    //Read new data.
    enc.evaluate();

    if(motor.enabled())
      //Main engine controller.
      positionController->evaluate(xy_buf[xy_i].x, xy_buf[xy_i].y);

    //Tollerance check.
    if(
      enc.getRho() > target_rho - settings.tol_rho &&
      enc.getRho() < target_rho + settings.tol_rho
    ){
      if(xy_i < xy_len - 1){
        xy_i++;
        target_rho = hypot(xy_buf[xy_i].x, xy_buf[xy_i].y);
      }

      else{
        //Stop engine (but continue evaluating the encoder position).
        motor.stop();
        positionController->reset();

        //Clear target.
        xy_i = 0;

        xy_buf[xy_i].x = 0;
        xy_buf[xy_i].y = 0;
        target_rho = 0;
      }
    }

    if(!SERIAL_DEBUG && c++ == int(1 / (N_SAMPLES * DELTA_T))){
      c = 0;

      plotter.start();
      plotter.add(…);
      plotter.add(…);
      …
      plotter.plot();
    }
  }

  //Time to check for some serial packets.
  if((tmp = ss.read()) == PACKET_PAYLOAD){
    ss.readBytes((uint8_t*) &packet, sizeof(packet));
    handle_packet();
    ss.write((uint8_t*) &packet, sizeof(packet));
  }
}
```

Lo snippet di codice soprastante riporta il loop di controllo principale e può essere suddiviso in tre principali sezioni:

1) Evaluate della posizione corrente tramite encoder e correzione tramite controllo in posizione.

   Tramite le due istruzioni `enc.evaluate()` e `positionController->evaluate(…)` si riesce rispettivamente a ricavare le informazioni sulla posa dai due encoder ed a rimandarla in feedback ai due motori.

2) Gestione delle tolleranze, del raggiungimento del punto target e del plot dei dati.
   Tramite l'istruzione if di tollerance check, è possibile riferirsi a delle tolleranze precedentemente impostate per poter riconoscere il momento adatto in cui il robot si deve fermare.

3) Gestione dei pacchetti di comando provenienti dall'ESP32.

Protocollo di comunicazione binario tra i due MCU

Lo standard di comunicazione scelto utilizza in questo caso un bus UART, il quale ha il compito di trasportare dati binari da un microcontrollore all'altro; un aspetto elettrico importante sono ancora una volta i livelli logici in tensione che i segnali elettrici del bus assumono: si ha infatti che l'ATmega328 è un microcontrollore operante a 5V, mentre l'ESP32 opera a 3,3V.

È necessario quindi adattare i livelli logici tramite un solo partitore resistivo dal pin Tx della seriale software dell'ATmega328 al pin Rx dell'ESP32 in quanto anche se l'altro bus fisico lavora ad una logica a 3,3V invece che a 5V, questo non crea problemi di compatibilità elettrica.

Per la definizione dei dati binari da mandare, sono state progettate ed utilizzate la seguenti strutture dati contenute nel file header packet.h:

```
template<class T = float> struct __attribute((__packed__)) packet_t{
  packet_data_t com;

  uint8_t argc;
  T argv[PACKET_ARGV_MAXLEN];
};

struct __attribute((__packed__)) xy_t{
  float x = 0;
  float y = 0;
};
```

Tramite la prima struttura dati è stato quindi possibile il passaggio di dati da un microcontrollore ad un altro semplicemente andando a definire i seguenti parametri:

```
packet_data_t com
```

Tipologia di commando da portare a termine; viene comunicato direttamente dal software risiedente sull'ESP32.

```
uint8_t argc
```

Lunghezza del vettore di argomenti che tale comando può richiedere o meno per essere portato a termine

```
T argv[PACKET_ARGV_MAXLEN]
```

Vettore degli argomenti: ha una lunghezza prefissata, in modo tale da permettere una futura espandibilità.

Per quanto riguarda la seconda struttura dati, questa serve al passaggio di array di punti nella porta seriale; questi punti, come verrà spiegato successivamente, rappresentano i punti di un percorso calcolato e comunicato dall'ESP32 tramite uno specifico comando dedicato.

Conclusioni sul firmware Arduino Nano

Il firmware progettato e testato funziona correttamente e supporta quindi tutti i comandi descritti nella libreria packet.h:

packet.h: binary packet definition

Commands

Command	Parameters	Return values	Description
`COMMAND_RESET`	N/A	N/A	Set the zero of the ortogonal system in the current point of the 2D space.
`COMMAND_RESET_ROUTINE`	N/A	N/A	Execute the automatic reset routine by using the two front switches.
`COMMAND_POSE`	N/A	<x, y, theta>	Get the pose of the robot.
`COMMAND_GOTO`	<x, y>	N/A	Go to the <x, y> point.
`COMMAND_WAIT_XY_ARRAY`	<len>	N/A	Await for an array of <x, y> points (floats) of length <len>.
`COMMAND_START|STOP`	N/A	N/A	Start | Stop engine.
`COMMAND_KPID_GET|SET`	N/A | <p_module_kp, p_phase_kp, s_kp, s_ki>	<p_module_kp, p_phase_kp, s_kp, s_ki>	Get | Set the various PID constants.
`COMMAND_TOL_GET|SET`	N/A | <tol_rho>	<tol_rho>	Get | Set the distance tollerance (m).
`COMMAND_MAX_SPEED_GET|SET`	N/A | <max_linear_speed, max_angular_speed>	<max_linear_speed, max_angular_speed>	Get | Set robot maximum speeds (m/s, rad/s).
`COMMAND_LOAD|SAVE`	N/A	N/A	Load | Save current settings from | to EEPROM.

NB: Every command returns `CONTROL_OK` if it's accomplished correctly.

NBB: Every angle is measured in radians by default.

Control codes

Control code	Description
`CONTROL_OK`	Default reply message.
`CONTROL_ERROR`	Generic error.
`CONTROL_INVALID_MSG`	Invalid command.

Figura 12: lista e descrizione dei vari comandi binari disponibili in packet.h.

Tramite l'utilizzo di questi comandi è quindi possibile:

- Fare compiere al carrellino del robot i più basilari movimenti.
- Richiedere la stampa dei dati e delle configurazioni principali del robot.
- Configurare il robot sotto tutti i principali aspetti del sistema di controllo.
- Salvare e caricare la configurazione su EEPROM, in modo da mantenere permanenti i cambiamenti effettuati nelle configurazioni precedenti.

Firmware ESP32

In questo capitolo verrà descritta l'implementazione del firmware risiedente sulla scheda ESP32.

Questo firmware ha lo scopo principale di realizzare una comunicazione via http con il front-end web e di tradurre i comandi inviati dall'utente in dati binari compatibili con lo standard descritto precedentemente.

Inoltre, altro compito fondamentale assegnato a questo microcontrollore decisamente prestante è quello di utilizzare l'algoritmo di Fixed Graph implementato nella libreria FixedGraph.h come algoritmo di collision avoidance.

Gestione interna del Fixed Graph

Tale algoritmo discretizza il piano 2D dove il robot si dovrà spostare, utilizzando uno step prefissato di 200cm con un lato di 10 nodi, portando così la dimensione totale della griglia rappresentante il piano 2D ad una dimensione di $1.8m^2$: quest'area rappresenterà quindi il piano di lavoro del robot stesso.

Una volta fissato lo step da utilizzare e la dimensione della griglia, sono stati implementati due metodi `setVertex` e `clearVertex`, i quali hanno rispettivamente il compito di aggiungere e rimuovere virtualmente un ostacolo all'interno del piano di lavoro del robot.

Essendo il grafo risultante un grafo di nodi rappresentanti i punti 2D del piano, il robot si potrà tranquillamente spostare su questo piano tenendo conto degli ostacoli, i quali sono stati implementati semplicemente azzerando il peso di tutti gli archi uscenti dal nodo specificato nella funzione `clearVertex`.

Una volta specificati i singoli ostacoli, il problema si riduce quindi al problema di ricerca di un cammino minimo da singola sorgente in un grafo unitariamente pesato; per la risoluzione di questa tipologia di problemi, detti di SSSP (Single Source Shortest Path) esiste ed è noto l'algoritmo di Dijkstra, il quale è stato implementato nel metodo `dijkstra`.

Il metodo in questione prende come parametri l'ascissa e l'ordinata della sorgente, mentre tramite l'utilizzo del metodo `getShortestPath`, richiamato dopo l'esecuzione dell'algoritmo di Dijkstra, si riesce ad ottenere il vettore contenente il cammino minimo che ha come sorgente quella specificata nella chiamata di `dijkstra`, mentre come destinazione quella specificata dai parametri di `getShortestPath`, i quali sono proprio l'ascissa e l'ordinata della destinazione desiderata.

Trasmissione delle coordinate del percorso

La traduzione e la codifica in coordinate (x, y) avviene servendosi di un vettore avente come singole componenti delle coppie di punti (struttura dati `xy_t` in packet.h); questo vettore di punti sarà quindi comunicato successivamente al firmware di Arduino Nano per il corretto raggiungimento del target tramite il percorso specificato.

Interfaccia utente

L'interfaccia utente è stata sviluppata tramite l'utilizzo del WiFi integrato dell'ESP32 e di un server asincrono http.

Tramite questa funzionalità ed il multithreading offerto dall'architettura ESP32, è stato possibile realizzare un piccolo portale di configurazione web per la specifica dei vari comandi e delle varie impostazioni del robot:

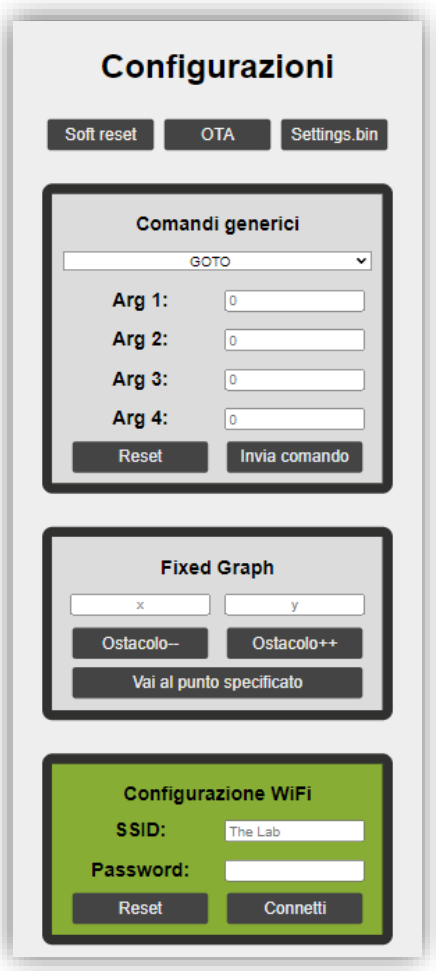

Figura 13: portale web per la configurazione del robot.

L'utilizzo dell'interfaccia utente comprende anche la configurazione del WiFi: se infatti la scheda non dovesse riuscire a connettersi all'access point salvato precedentemente, allora questa andrebbe in modalità di configurazione stand-alone aprendo un Hotspot di nome `RoboticSystems-Project` dove, una volta effettuata la connessione tramite un dispositivo munito di WiFi e di un Web Browser, è possibile ritrovare la pagina in figura 13 all'indirizzo 192.168.4.1.

Tramite la suddetta pagina, sarà quindi possibile specificare un nuovo access point al quale effettuare la connessione WiFi; tramite invece la sezione "Comandi generici" è possibile specificare tutti i comandi con i relativi parametri specificati in figura 12.

La sezione “Fixed Graph” permette invece di aggiungere o rimuovere ostacoli dalla griglia del grafo, in modo tale da permettere l’aggiunta e la rimozione dinamica di ostacoli dal grafo stesso.

Un'altra funzione di questa sezione è quella di fare partire l’algoritmo di shortest path verso la destinazione specificata, in modo tale da calcolare e comunicare alla scheda Arduino Nano tutti i punti per cui il robot dovrà passare per poter raggiungere correttamente il punto target specificato.

Come altre funzionalità, in cima alla pagina sono presenti tre pulsanti per poter eseguire rispettivamente il soft reset della scheda, l’aggiornamento OTA sia del firmware che del filesystem tramite WiFi ed il download del file “settings.bin”, il quale contiene i dati binari delle stringhe di connessione della rete WiFi salvata.

Salvataggio su filesystem LittleFS

Il salvataggio delle impostazioni della scheda e dell’effettivo portale di configurazione avviene direttamente su memoria flash utilizzando il filesystem LittleFS; tramite l’utilizzo di questo filesystem è stato possibile scrivere e fare debug di tutto il codice HTML, CSS e Javascript separatamente.

La scrittura vera e propria del filesystem avviene analogamente alla riscrittura della flash quando si aggiorna il firmware della scheda, con l’unica differenza che gli indirizzi di scrittura di questi dati sono diversi da quelli dell’effettivo bytecode del firmware.

Come descritto prima, il filesystem è aggiornabile tramite OTA allo stesso modo del firmware tramite l’interfaccia web offerta dalla libreria AsyncElegantOTA:

Figura 14: pagina di aggiornamento OTA tramite la libreria AsyncElegantOTA.

Realizzazione finale del robot fisico

Di seguito, alcune fotografie del risultato finale del robot fisico realizzato in laboratorio:

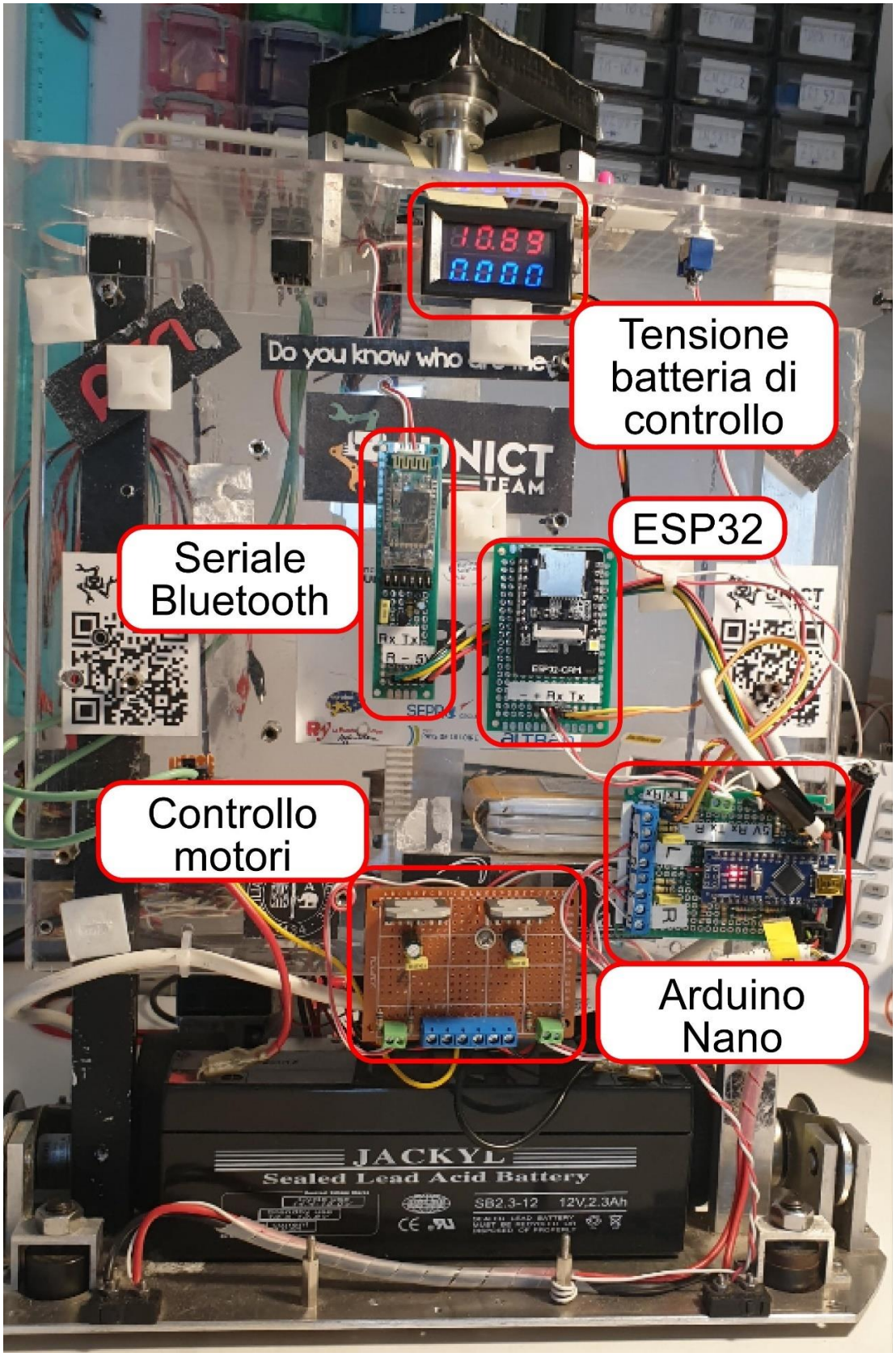

Figura 15: parte frontale del robot.

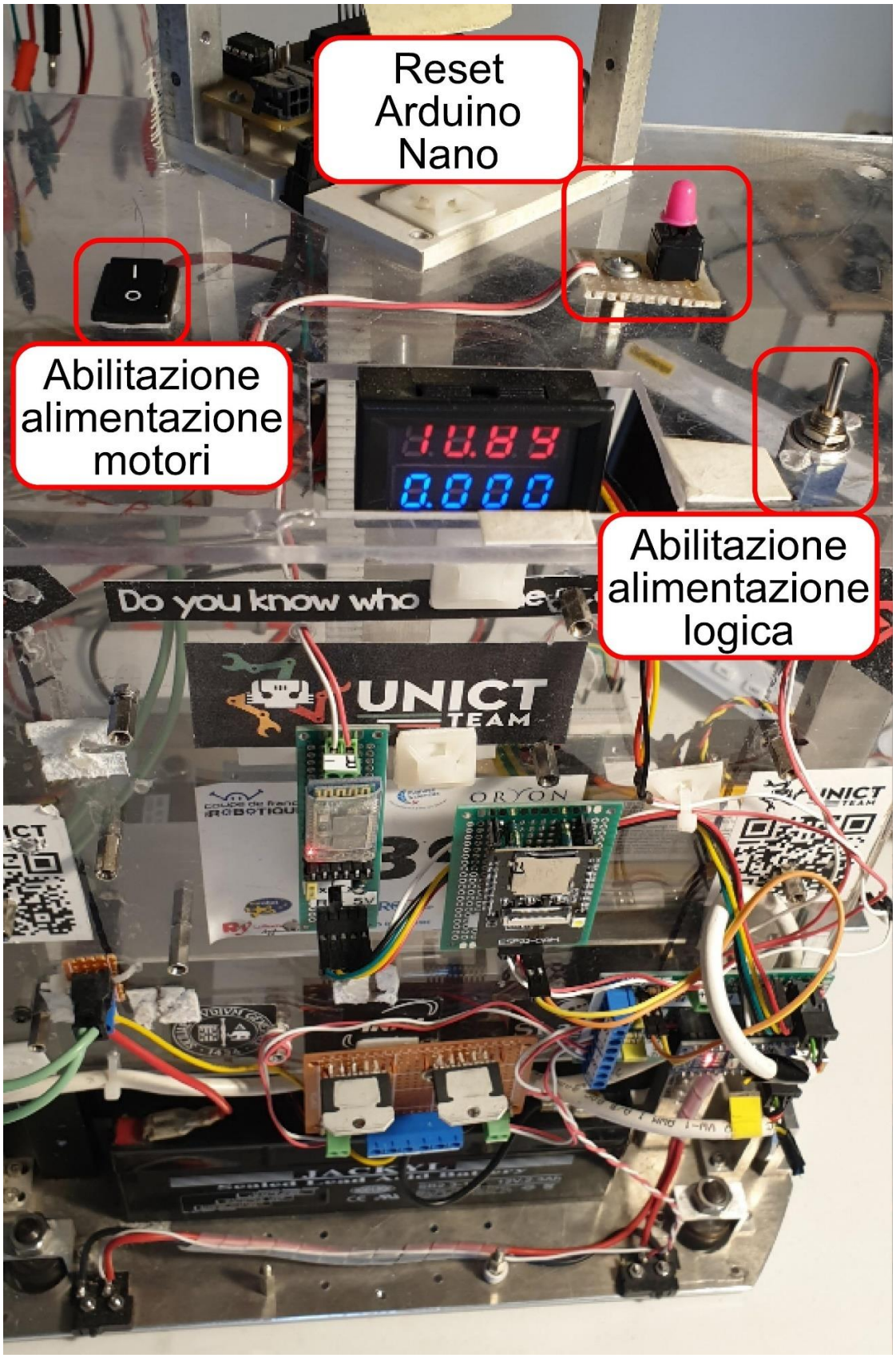

Figura 16: parte superiore del robot.

Conclusioni

Il progetto è stato di notevole difficoltà da realizzare, specialmente per la vastità di argomenti coperti dal progetto stesso; l'elettronica non è stata particolarmente complicata da realizzare, mentre la corretta scrittura del software di controllo ed il conseguente debug sono state operazioni decisamente più ardue.

L'obiettivo iniziale del progetto consisteva nella rilevazione automatica di ostacoli sul cammino del robot stesso tramite la telecamera integrata dell'ESP32-Cam, ma successivamente, vista la notevole difficoltà nell'implementazione del suddetto algoritmo, si è deciso che una realizzazione tramite Fixed Graph dell'algoritmo di controllo delle collisioni sarebbe stata di più facile e rapida implementazione.

Nonostante questo, il progetto è da ritenersi un successo in quanto l'hardware del progetto è funzionante e di facile espandibilità: l'aggiunta di nuove periferiche è senz'altro un'operazione semplice grazie alla modularità con cui il sistema hardware stesso è stato progettato; oltretutto, la presenza degli schematici hardware per il software KiCad permette anche la successiva progettazione di circuiti stampati per permettere anche un assemblaggio facilitato e sicuramente funzionante.

Per quanto riguarda il sistema software, questo si è dimostrato funzionante ed esente da grossi bug e, grazie alla sua natura altamente modulare, risulta anche essere di facile manutenibilità, mentre l'interfaccia web responsive, disegnata per essere utilizzata facilmente da dispositivi mobile, permette la configurazione delle impostazioni del robot, dando la possibilità all'utente di poter eseguire il controllo di quest'ultimo sulla base delle funzionalità che nel codice sono state predisposte.