

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](#_Toc112688787)

[SERVE GIGI PER PARTE DI MACHINE LEARNING 1](#_Toc112688788)

[Repository GitHub del progetto 1](#_Toc112688789)

[Tecnologie utilizzate 1](#_Toc112688790)

[Hardware 2](#_Toc112688791)

[Schema funzionale dell’hardware 2](#_Toc112688792)

[Funzionamento dei due encoder 2](#_Toc112688793)

[Attenuazione in tensione dei due segnali A e B 3](#_Toc112688794)

[Funzionamento dei ponti ad H 4](#_Toc112688795)

[Interfacciamento e debug 4](#_Toc112688796)

[Schema elettrico della scheda Arduino Nano 5](#_Toc112688797)

[Schema elettrico della scheda dei ponti ad H 5](#_Toc112688798)

[Firmware Arduino Nano 6](#_Toc112688799)

[Interfacciamento con gli encoder 6](#_Toc112688800)

[Funzionamento della classe RI32 7](#_Toc112688801)

[Classe RI32 8](#_Toc112688802)

[Interfacciamento con i motori 9](#_Toc112688803)

[Modalità PWM in correzione di fase 10](#_Toc112688804)

[Classe LMD18200 11](#_Toc112688805)

[Schema di controllo in velocità 12](#_Toc112688806)

[Classe SpeedController 13](#_Toc112688807)

[Schema di controllo in posizione 13](#_Toc112688808)

# 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 fare muovere tale carrellino da un punto di partenza ad un punto di arrivo evitando al più un ostacolo che si frappone sul suo percorso.

Più nello specifico, …

## SERVE GIGI PER PARTE DI MACHINE LEARNING

# [Repository GitHub del progetto](https://github.com/LoZioo/RoboticSystems-Project)

## 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](https://www.falstad.com/circuit/circuitjs.html).
  + Schede di sviluppo [Arduino Nano](https://www.amazon.it/Elegoo-Scheda-Microcontrollore-ATmega328P-Arduino/dp/B0716S43Q2/ref=sr_1_1_sspa?__mk_it_IT=%C3%85M%C3%85%C5%BD%C3%95%C3%91&crid=1Z0EFB0MPF833&keywords=elegoo+nano&qid=1661682817&sprefix=elegoo+nano%2Caps%2C137&sr=8-1-spons&psc=1&spLa=ZW5jcnlwdGVkUXVhbGlmaWVyPUExOTFTM0I1RkFCVDVaJmVuY3J5cHRlZElkPUEwNzU5MTUwM1UxMDVGRlZDQTZJTiZlbmNyeXB0ZWRBZElkPUEwODMwNDA4MTNPWEY0MzVTU005RyZ3aWRnZXROYW1lPXNwX2F0ZiZhY3Rpb249Y2xpY2tSZWRpcmVjdCZkb05vdExvZ0NsaWNrPXRydWU=) ed [ESP32-Cam](https://www.amazon.it/Sviluppo-Fotocamera-USB-TTL-Conversione-Seriale/dp/B094FRMXSZ/ref=sr_1_6?keywords=esp32+cam&qid=1661682759&sprefix=esp32%2Caps%2C104&sr=8-6).
  + Modulo Bluetooth [HC-05](https://www.amazon.it/AZDelivery-ricetrasmettitore-wireless-Bluetooth-Comunicazione/dp/B0722MD4FY/ref=sr_1_1_sspa?__mk_it_IT=%C3%85M%C3%85%C5%BD%C3%95%C3%91&crid=VCMQS6S8EDFM&keywords=HC-05&qid=1661683082&sprefix=hc-05%2Caps%2C93&sr=8-1-spons&smid=A1X7QLRQH87QA3&spLa=ZW5jcnlwdGVkUXVhbGlmaWVyPUFUWllOVEZOVUpSWTQmZW5jcnlwdGVkSWQ9QTAyODI1NjMzUkhTQ1RLMVAzMTlYJmVuY3J5cHRlZEFkSWQ9QTA0NTE3NzFVSFo4Q1ZQUDFVM1Umd2lkZ2V0TmFtZT1zcF9hdGYmYWN0aW9uPWNsaWNrUmVkaXJlY3QmZG9Ob3RMb2dDbGljaz10cnVl&th=1) (serial over Bluetooth)
  + 2x ponte ad H singolo [LMD18200](https://www.ti.com/lit/ds/symlink/lmd18200.pdf?ts=1661683009317&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FLMD18200%253Futm_source%253Dgoogle%2526utm_medium%253Dcpc%2526utm_campaign%253Dasc-null-null-GPN_EN-cpc-pf-google-eu%2526utm_content%253DLMD18200%2526ds_k%253DLMD18200%2526DCM%253Dyes%2526gclid%253DCjwKCAjwpKyYBhB7EiwAU2Hn2ZTW0jX3bP8a9QMBnMB4dV7YMbWM8N3cSoBCGZ0LFfB1N3rDOxUEcxoCvgUQAvD_BwE%2526gclsrc%253Daw.ds).
  + 2x encoder rotativi 1000 ticks [RI-32](https://www.hengstler.de/gfx/file/shop/encoder/RI32/Datasheet_RI32_en.pdf).
  + Batteria ai polimeri di litio 11.1V (alimentazione logica).
  + Batteria al piombo 12V (alimentazione motori).
* **Firmware**
  + Linguaggio di programmazione C++.
  + IDE [VS Code](https://code.visualstudio.com/) + [PlatformIO](https://platformio.org/).
  + Framework [Arduino](https://www.arduino.cc/).
  + Sistema operativo real-time [FreeRTOS](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html).
  + [SerialPlot](https://hackaday.io/project/5334-serialplot-realtime-plotting-software).

# Hardware

Il primo passo per lo sviluppo di questo progetto è stato sicuramente 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 con la seguente figura:

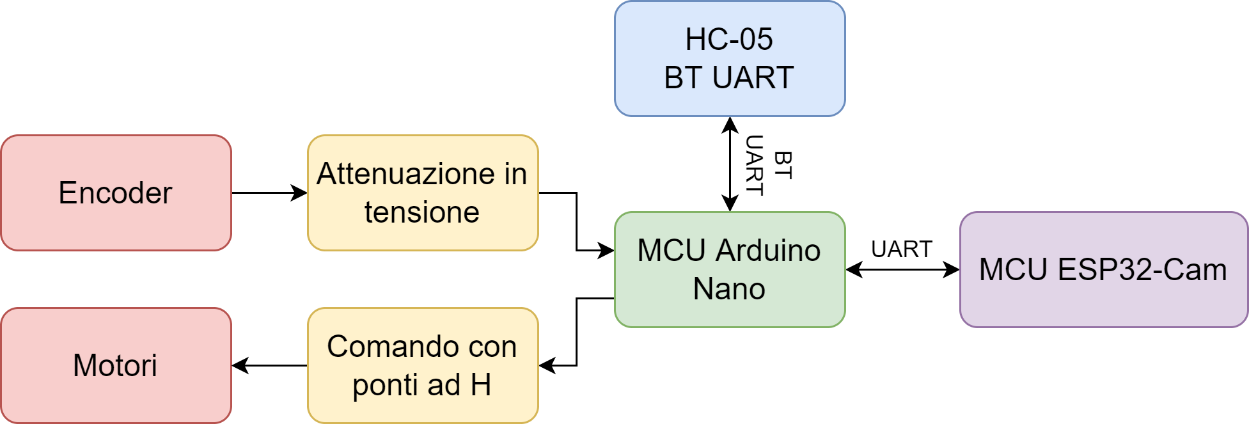


Figura : schema funzionale dell'hardware dell'intero progetto.

## 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.

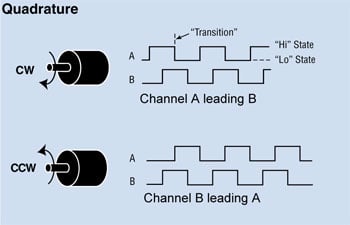


Figura : esempio di due segnali in quadratura in uscita da un encoder rotativo.

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 canale (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:

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

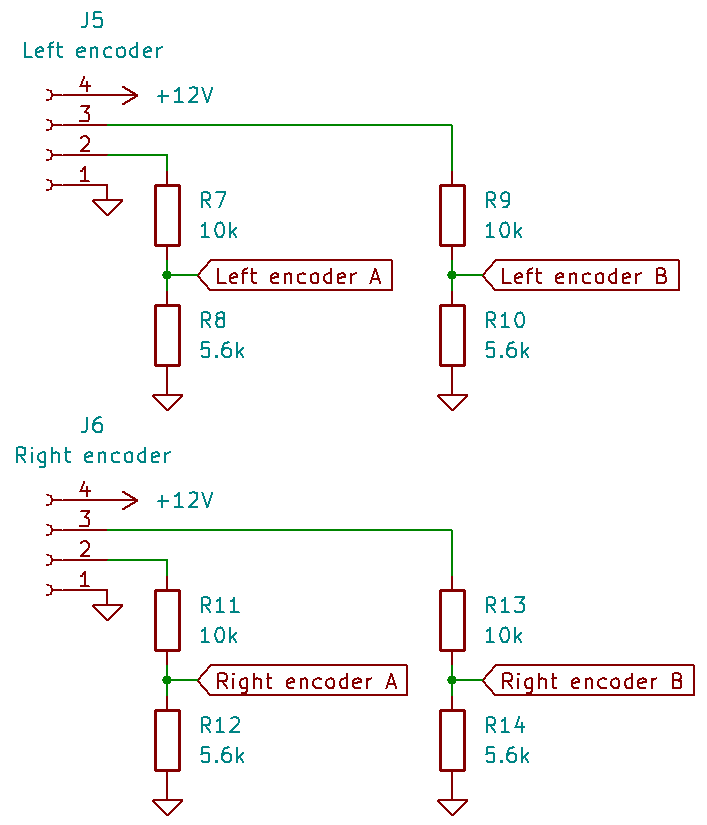


Figura : circuito di adattamento in tensione per due coppie di segnali in quadratura (in totale, quattro segnali da leggere).

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 ad 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.

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

Questa seriale su bluetooth si è rivelata molto utili anche per scopi di debug: a tale scopo è stata anche 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 stato 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

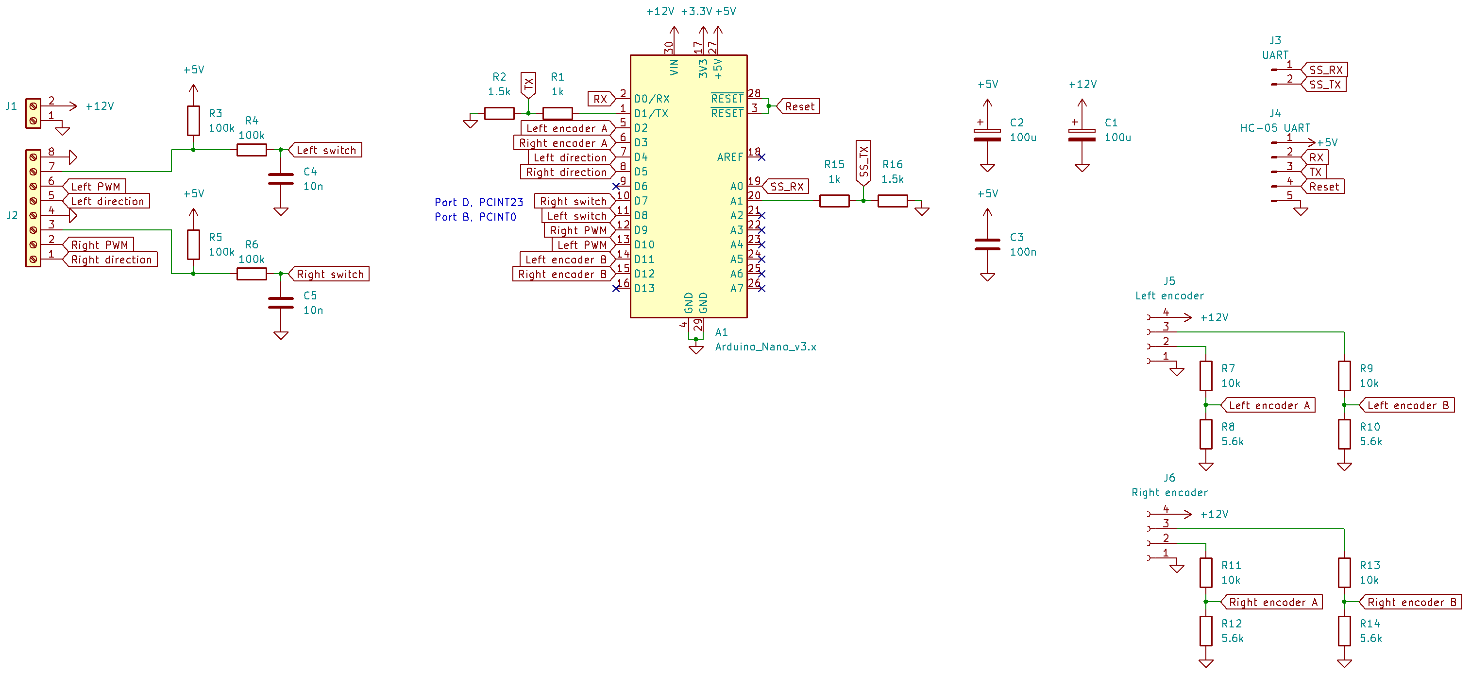


Figura : schema elettrico della scheda Arduino Nano.

## Schema elettrico della scheda dei ponti ad H

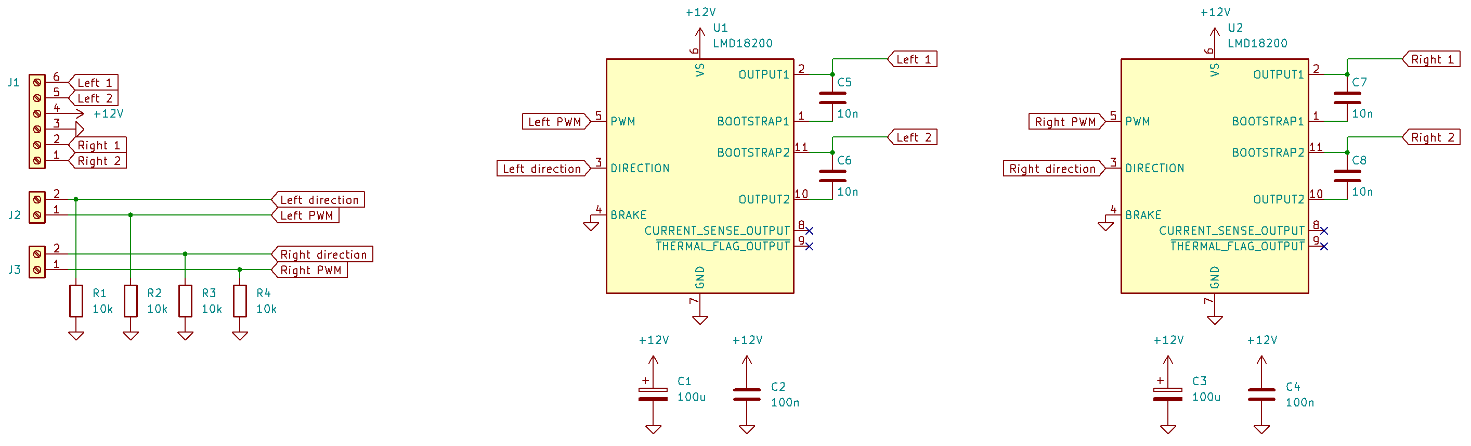


Figura : schema elettrico della scheda dei ponti ad H.

# 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, come già spiegato, 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(2), ENC\_L\_ISR, RISING);

attachInterrupt(digitalPinToInterrupt(3), 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\_l) 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 .

La discretizzazione avviene ad istanti di tempo fissati ed ugualmente spaziati; è stato quindi scelto un periodo di campionamento di , cioè una frequenza di campionamento pari a .

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

1. Calcolo di :

Dove:

* + è l’arco d’angolo percorso per ruota tra l’istante e l’istante .
  + è pari al numero di giri che l’encoder produce (a datasheet).
  + è pari al numero di tick contati per ruota tra l’istante e l’istante (dTicks\_l/r).

1. Calcolo di :

Dove:

* è lo spazio percorso per ruota tra l’istante e l’istante .
* è il raggio in metri delle due ruote di misura.

Per il successivo calcolo dello spazio percorso per ruota tra l’istante e l’istante , basterà accumulare nel tempo ; questo spazio percorso verrà identificato con .

1. Calcolo di :

Dove:

* è la velocità istantanea per ruota.
* , come già discusso, rappresenta il periodo di campionamento dei dati.

1. Calcolo di e :

Dove è la wheelbase delle due ruote di misura (la loro distanza).

1. Calcolo di e :
2. Calcolo della posa :

Dove:

* Nel calcolo di ed , il utilizzato è relativo all’istante e non all’istante , come si potrebbe erroneamente pensare.
* è una funzione di normalizzazione angolare, la quale aggiunge o sottrae all’angolo in questione fino a quando il suddetto non ricade nel range .

## 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 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.

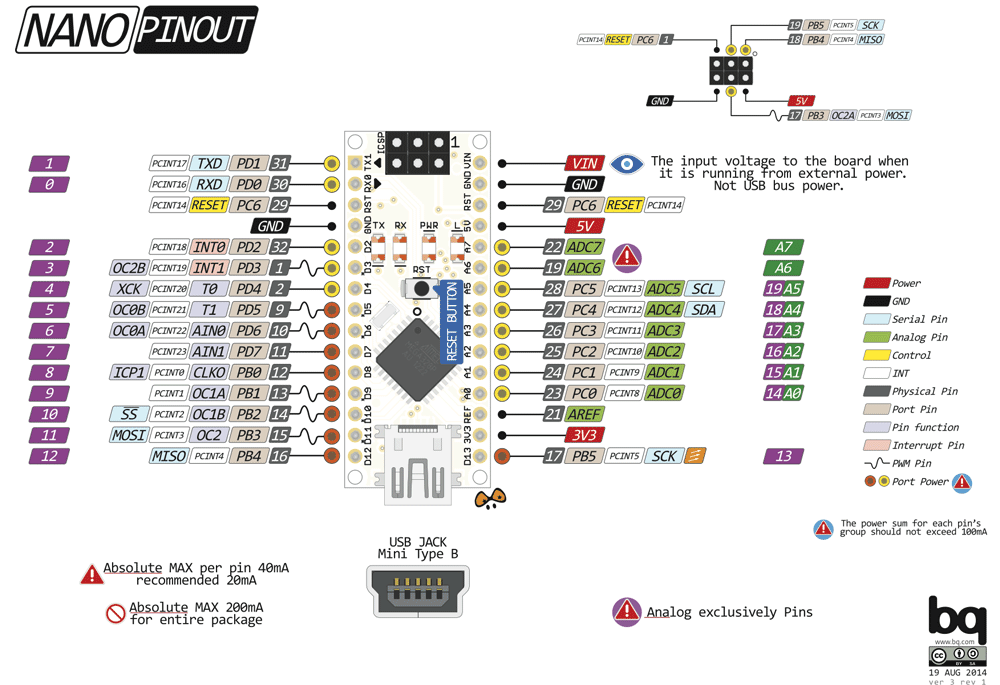


Figura : pinout standard Arduino Nano.

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 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:

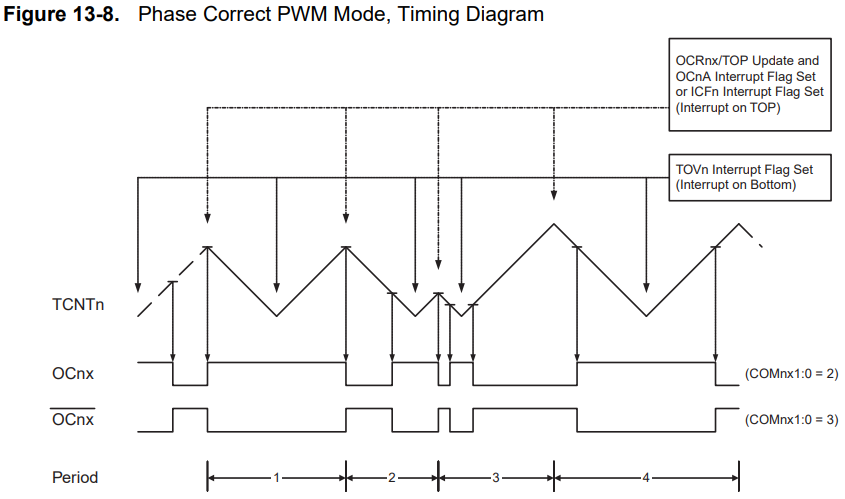


Figura : dal datasheet dell'ATmega328, pag.129: grafici di funzionamento della modalità PWM in correzione di fase.

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 è verificata si avrà infatti un cambio di stato del corrispettivo pin associato ad **OC1A/B** in concomitanza con certi 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:

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 .

## 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 invece 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 direttamente al segno ed al valore assoluto di pwm\_direction\_speed, 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 ed , 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:

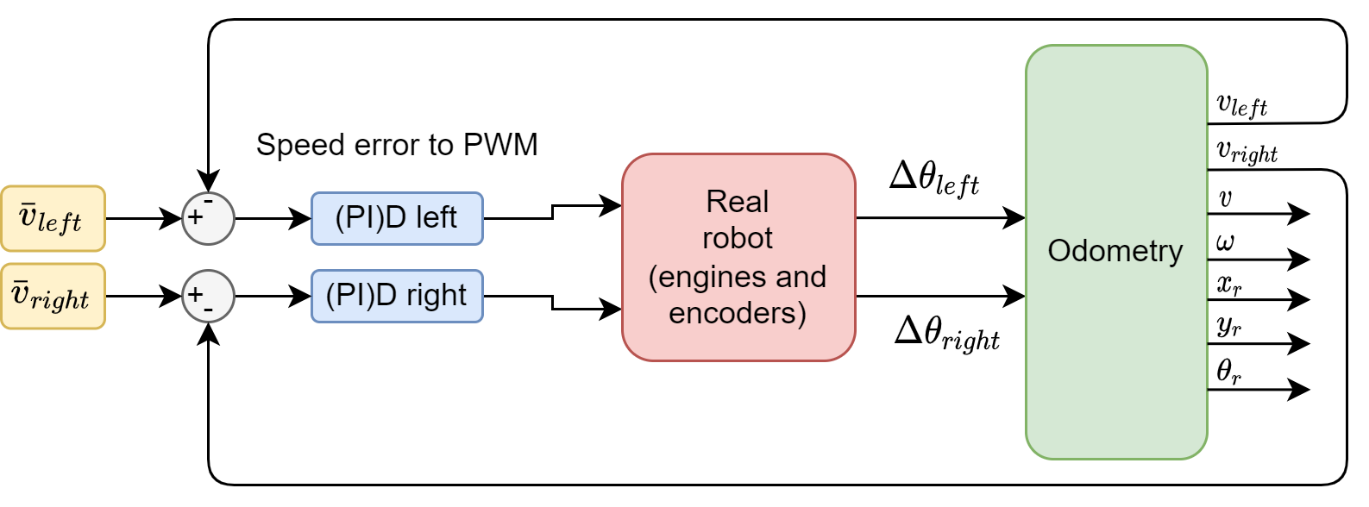


Figura : schema di controllo in velocità del carrellino.

Dallo studio di questa tipologia di carrellino fisico a due ruote motrici, essendo le due velocità ed 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 , in quanto si può decidere di effettuare direttamente un controllo in velocità date e; 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 ed che il carrellino deve raggiungere.

Il metodo, per poter funzionare correttamente, deve anch’esso essere chiamato ogni 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(…).

## 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:

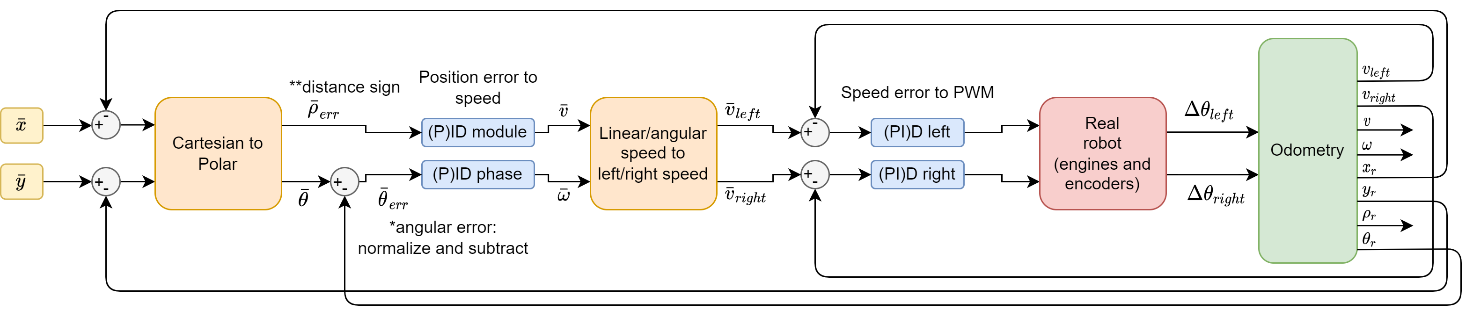


Figura : schema di controllo in posizione del carrellino.

Il funzionamento dello schema può essere riassunto in questi punti chiave:

1. Rilevamento delle attuali coordinate dagli encoder tramite odometria.
2. Calcolo errore in posizione .
3. Conversione cartesiana-polare .
4. A questo punto, l’errore in modulo può essere direttamente utilizzato come errore per controllare un PID mentre, per il calcolo di , bisogna necessariamente tener conto di per poi calcolare effettivamente l’errore di fase .
5. Se :

Questo caso particolare serve semplicemente a tener conto del fatto che il punto target 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 all’errore di orientamento .

1. Ottenuta quindi la coppia , basta utilizzare i due PID chiamati PID module e PID phase per poter realizzare la conversione .
2. Converti e richiama quindi il controllo in velocità descritto precedentemente.

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 PositionController::evaluate(…).

DOCUMENTARE ANCHE IL DEBUG CON SERIALPLOT