Politecnico di Torino

Tirocinio Curriculare

PIC4SeR

Report

Andrea Grillo
S282802

A.A. 2022/2023

Introduzione

Il presente documento viene compilato a conclusione del Tirocinio Curriculare del corso di Laurea triennale in Ingegneria Informatica.
Di seguito una breve overview degli argomenti trattati:

- Obiettivi da raggiungere
- Descrizione generale del sistema da realizzare
- Analisi del sistema già proposto in passato
- Individuazione limiti del sistema proposto
- Ideazione del nuovo sistema
- Descrizione dettagliata del processo di sviluppo del nuovo sistema
- Roadmap per la messa in funzione del sistema

Obiettivi

L'attività di tirocinio svolta ha come oggetto l'aggiunta di sensori alla piattaforma robotica Paquitop.

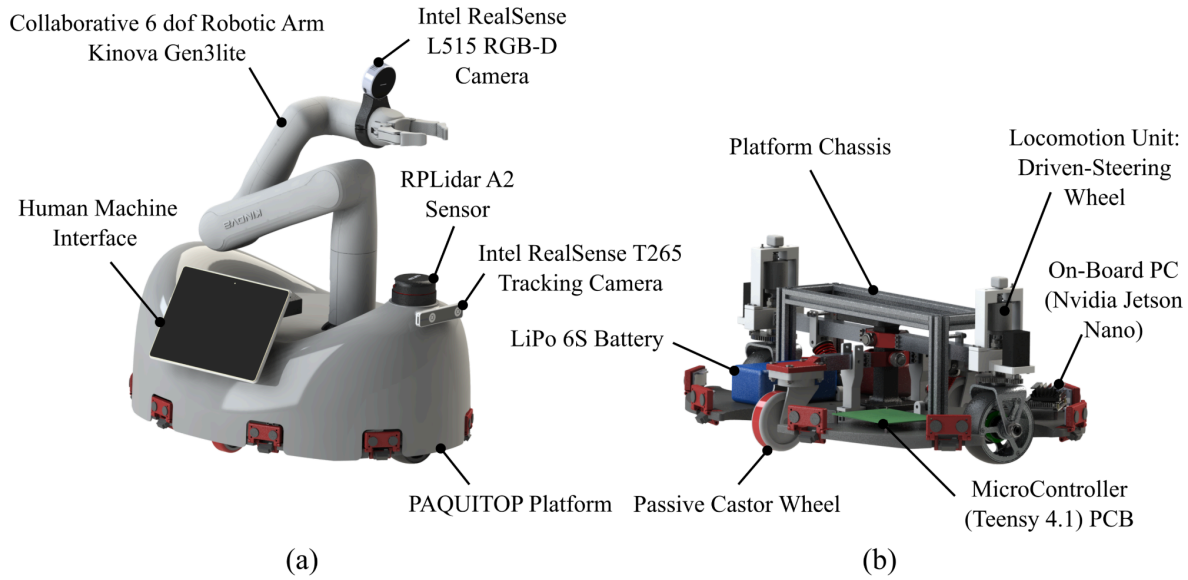

fig 1: piattaforma PAQUITOP

Gli obiettivi di tale aggiunta sono:

- Ridondanza rispetto alla sensoristica già presente. A bordo della piattaforma robotica è già presente un sensore LIDAR, collegato al controllore di alto livello. Si vuole aggiungere un livello di sicurezza in più nella rilevazione di ostacoli esterni, collegato direttamente al controllore di basso livello.
- Aggiunta funzionalità di rilevazione ostacoli non rilevabili da LIDAR. Il LIDAR effettua una mappatura in 2D dell'ambiente circostante, e non è in grado di rilevare oggetti che non si trovano sul piano di scansione. In particolare si desidera poter rilevare la presenza di gradini "in discesa".

Sistema da realizzare

L'idea è di posizionare 8 gruppi sensori, posizionati in modo da coprire l'intero campo di mobilità della piattaforma. Ogni gruppo ha le proprie funzionalità di rilevamento ostacoli e comunica con il controllore centrale di basso livello i dati rilevati.

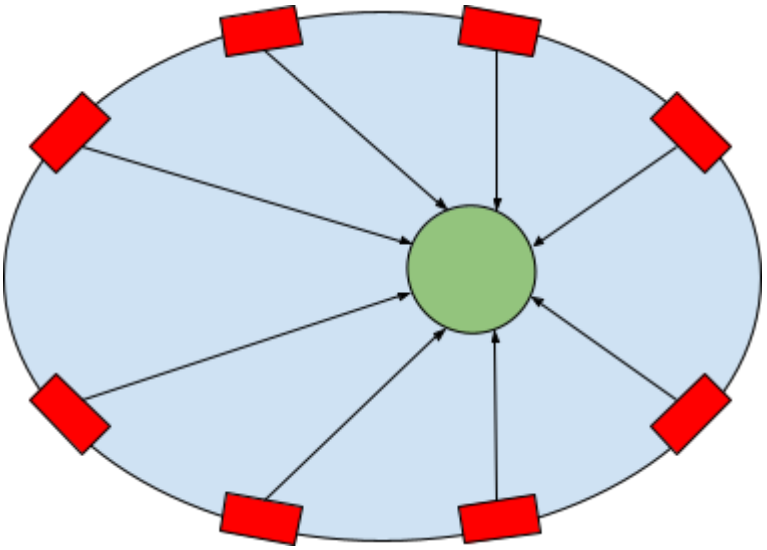

fig 2: schema generale del sistema

Legenda:

- ***Grigio**: Piattaforma robotica*
- ***Verde**: Microcontrollore*
- ***Rosso**: Gruppi sensori*

Sensori utilizzati

Le funzioni che ogni gruppo sensori deve realizzare sono le seguenti:

- Rilevamento distanza da ostacolo sul piano di movimento
- Rilevamento gradino

Per svolgere tali funzioni su ogni gruppo sensori sono presenti 2 dispositivi, un sonar e un laser. Il primo viene utilizzato per la rilevazione della distanza sul piano di movimento, invece il campo di visione del laser è diretto verso il suolo, per la rilevazione dei gradini.

Scheda di acquisizione dati

Entrambi i sensori hanno un output analogico, in cui la tensione di uscita è legata secondo una legge nota al dato misurato. L'informazione deve essere quindi convertita in digitale tramite un ADC per poter essere elaborata ed utilizzata dal controllore.

Non è possibile far svolgere tale conversione direttamente sulla scheda dove risiede il controllore centrale, in quanto le leggi che legano le grandezze da misurare alle

tensioni in uscita, fanno corrispondere piccole variazioni di tensione a sostanziali cambi di distanza misurata.
Tali variazioni di tensione sarebbero non sarebbero apprezzabili se la conversione fosse effettuata da un ADC a bordo del controllore centrale, a causa della lunghezza del cavo, che implica effetti resistivi e capacitivi non trascurabili, e dalle interferenze con i numerosi segnali di potenza che sono presenti a bordo della piattaforma.
È quindi necessario sviluppare una scheda di acquisizione dati che sia posizionata insieme ai gruppi sensori, che faccia le rilevazioni delle grandezze misurate e che comunichi in modo affidabile i risultati delle misurazioni svolte al controllore centrale.
Obiettivo di questo tirocinio è la progettazione, realizzazione e messa in funzione di tale scheda di acquisizione.

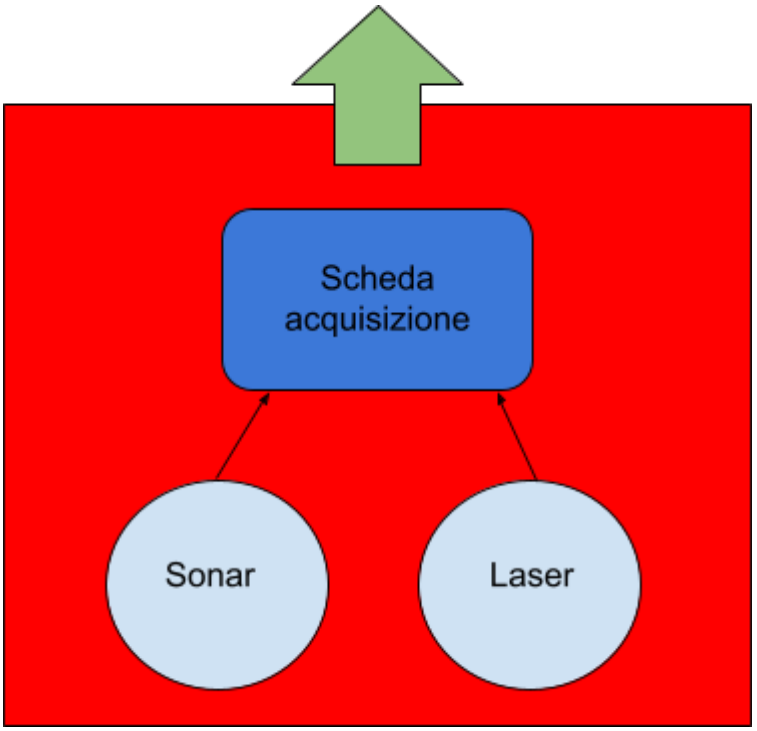

fig 3: dettaglio gruppo sensori

Sistema proposto

Il sistema proposto inizialmente consiste in un una scheda di acquisizione dati composta unicamente da componenti analogici, quali amplificatori operazionali e comparatori di soglia.
L'obiettivo del circuito è comparare i segnali in uscita dai due sensori a dei segnali di soglia, e generare dei segnali digitali che indicano il superamento o meno di tali soglie. Tali segnali digitali vengono inviati alla scheda di controllo centrale, che incorpora delle porte logiche OR per l'aggregazione dei segnali e la generazione degli interrupt per il microcontrollore, nonché un multiplexer per l'interrogazione sulla provenienza di tali input.
Il microcontrollore genera un segnale PWM, atto ad impostare i valori di soglia. A bordo di ogni scheda di acquisizione tale segnale viene filtrato con dei filtri RC passabasso, al fine di avere delle tensioni il più possibile stabili e costanti da poter confrontare con i segnali di uscita dei sensori a bordo.

Ogni scheda di acquisizione da in uscita tre segnali:

- ***G*** - *green*: per indicare che i valori sono in un range ottimale
- ***Y*** - *yellow*: per indicare una fascia di rischio ostacolo intermedia
- ***R*** - *red*: per indicare l'elevata vicinanza di un ostacolo

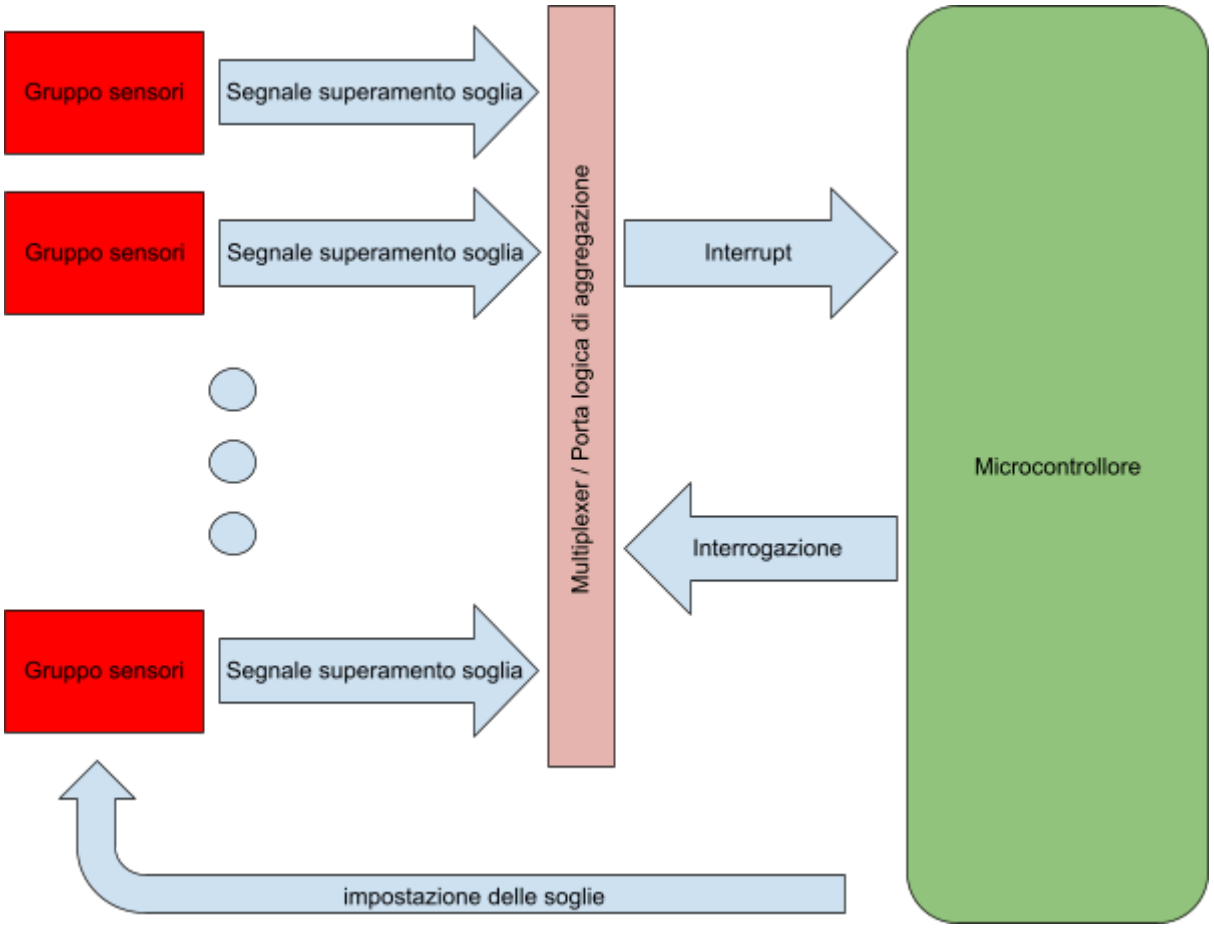

fig 4: schema di funzionamento sistema proposto

Flow chart microcontrollore

Il microcontrollore centrale, dato il sistema precedentemente raffigurato, deve quindi seguire il seguente flow chart per la ricezione di un segnale di superamento soglia da un gruppo sensore:

1. Ricezione interrupt da porta logica OR di aggregazione.
2. Interrogazione mediante bus I2C del multiplexer, in modo da conoscere il gruppo sensore di provenienza di tale interrupt. Il microcontrollore dovrà effettuare una richiesta I2C per ogni singolo gruppo sensore ad ogni ricezione di interrupt.
3. Elaborazione e utilizzo dati: il microcontrollore provvede ad utilizzare i dati raccolti per l'eventuale modifica della traiettoria precedentemente impostata.

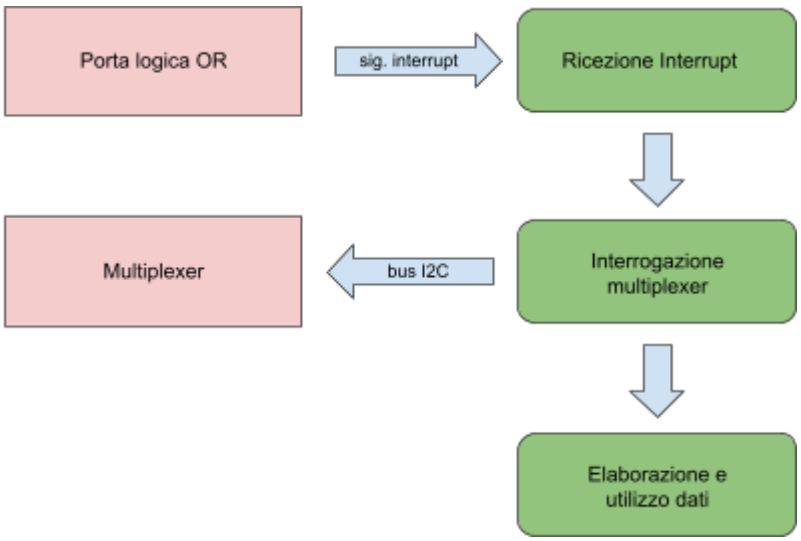

fig 5: flow chart microcontrollore

Limiti del sistema proposto

Nella analisi del sistema proposto sono emersi diversi limiti, che hanno portato all'ideazione di un sistema alternativo che superi tali limiti e che sia più flessibile nella realizzazione, nell'utilizzo e nella futura aggiunta di nuove funzionalità.

I limiti riscontrati sono i seguenti:

- Eccessivo tempo necessario per la ricezione di un allarme - alla ricezione dell'interrupt bisogna poi effettuare 8 interrogazioni al multiplexer via I2C - protocollo piuttosto lento - per capire quale sensore lo ha mandato.
- Scarsa precisione delle misure, a causa dei filtri RC che non sono in grado di dare in uscita un segnale perfettamente continuo, e sono fortemente influenzati dalle incertezze sui componenti utilizzati - *10%* su resistori e condensatori. In prossimità delle soglie si verifica sempre instabilità del segnale del comparatore a causa delle oscillazioni del valore di soglia.
- Grande numero di connessioni elettriche, da ogni gruppo sensori deve partire un gruppo di 8 fili che va alla scheda centrale. La scheda centrale prevede un numero di connettori pari al numero di gruppi sensori.
- Scarsa adattabilità a sensori diversi. Adattando i filtri RC si può utilizzare il sistema con altri sensori, però ciò implica un nuovo dimensionamento dei componenti e operazioni di saldatura per l'adattamento.

Ideazione nuovo sistema

Il nuovo sistema si pone come obiettivo in prima battuta di superare i limiti del precedente sistema.

L'idea è di avere una scheda di acquisizione composta da un microcontrollore *STM32*, che legge tramite *ADC* integrato i valori di tensione in uscita dai sensori, ed è connessa ad un bus *CAN* che interconnette tutti i gruppi sensore e il modulo centrale.

I vantaggi sarebbero quindi:

- Immediata conoscenza del sensore che ha generato l'allarme una volta ricevuto. Ogni pacchetto CAN conterrà un identificatore di chi lo ha inviato.
- Completa configurabilità in tempo reale: il controllore centrale può in qualsiasi momento impostare le soglie di allarme mandando un messaggio CAN ai sensori.
- Completa adattabilità a diversi sensori. Utilizzando l'ADC del microcontrollore per la lettura dei segnali dei sensori, è possibile riconfigurare il tutto unicamente via software, senza dover cambiare la configurazione elettronica.
- Facilità del cablaggio e maggiore semplicità del modulo centrale. Il numero di connessioni per sensore scende a 4 (*5V, GND, CAN_HI, CAN_LOW*) e i sensori possono essere collegati in una configurazione a *Daisy Chain*, in cui ogni gruppo sensore viene connesso in cascata a quello successivo, e il collegamento al modulo centrale avviene con un unico connettore.
- Possibilità di aggiungere e rimuovere gruppi sensori senza effettuare modifiche di alcun tipo al modulo centrale, ne ai gruppi sensore già presenti.

Tale sistema rende flessibile l'aggiunta di nuove funzionalità, agendo solamente via software senza andare a modificare l'hardware progettato.
Di seguito degli esempi di tali nuove funzionalità:

- Possibilità di ricezione dati di distanza: i gruppi sensore possono inviare via CAN, oltre agli allarmi quando vengono superate le soglie, le misurazioni di distanza effettuate, che possono essere utilizzate per altri scopi.
- Possibilità di delegazione di task di elaborazione ai microcontrollori presenti sui gruppi sensori, scaricando il controllore centrale.
- Possibilità di effettuare operazioni distribuite, in cui la comunicazione avviene tra i singoli gruppi sensori e non verso il controllore centrale.

Di seguito una rappresentazione dello schema hardware del nuovo sistema, si noti il collegamento a *Daisy Chain*, e la possibilità di posizionare il microcontrollore agli estremi ma anche in una posizione centrale della catena.

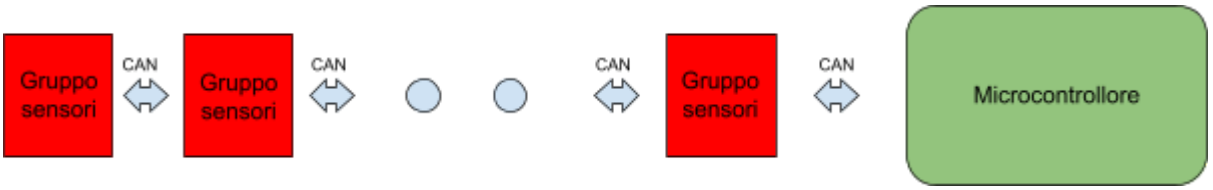

fig 6: schema nuovo sistema

Aspetto critico nuovo sistema

Il principale aspetto critico nello sviluppo del nuovo sistema si è rivelato la scarsa competenza nella progettazione elettronica di sistemi a microcontrollore. Ciò da una parte ha reso più lunga la fase iniziale di scelta del microcontrollore, definizione dello schematico e del layout della PCB della scheda di acquisizione.

D’altra parte, ha permesso a chi scrive di acquisire nuove competenze utili nell’ambito dell’elettronica e nella progettazione di PCB con microcontrollori ST.

Un aiuto sostanziale si è avuto a partire dal contatto con STMicroelectronics, azienda produttrice dei microcontrollori utilizzati, che ha dato supporto nella scelta del modello specifico di chip da utilizzare.

Scheda acquisizione dati

Come anticipato, sulla scheda acquisizione dati è presente un microcontrollore *STM32*.
Grazie al supporto dell'azienda produttrice, si è selezionato il modello *STM32G0B1KBT6*, che combina le ridotte dimensioni del package, insieme alla presenza di un ADC a *12bit*, e alla seriale *CAN* integrata. Questo ha permesso di rendere semplice lo sviluppo del circuito della scheda, in quanto ha ridotto il numero di componenti esterni da aggiungere.

Sulla scheda sono presenti i seguenti componenti:

- ☐ Gestione alimentazione: regolatore tensione e condensatori di decoupling
- ☐ Transceiver CAN
- ☐ LED segnalazione alimentazione
- ☐ LED programmabile
- ☐ Connessioni verso i sensori e verso il bus CAN

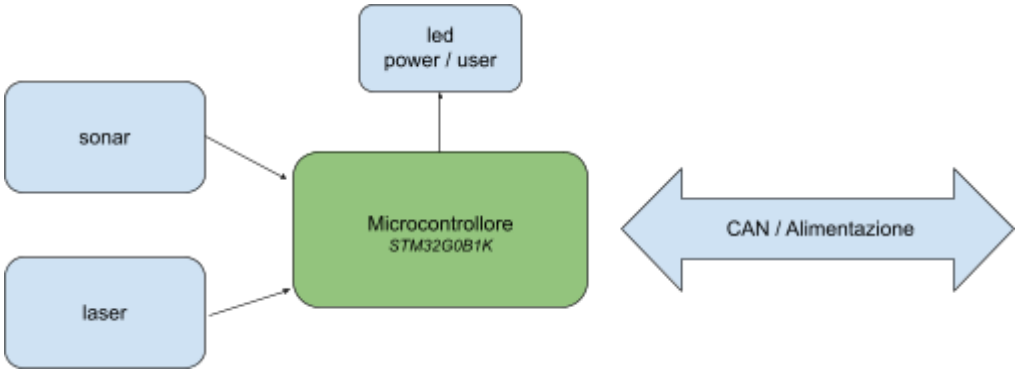

fig 7: dettaglio nuova scheda acquisizione dati

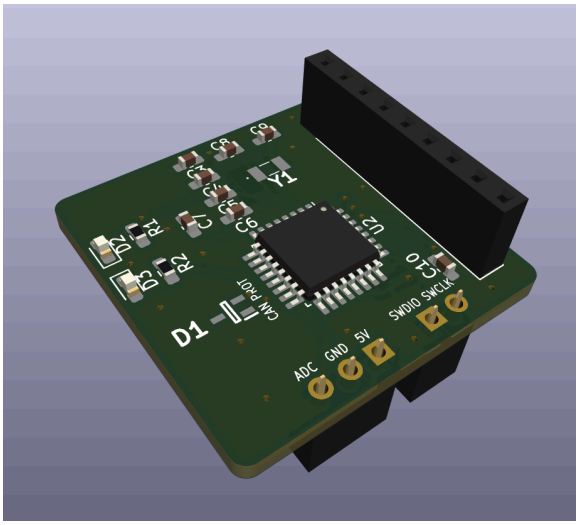

fig 8: render del PCB della scheda

Processo di sviluppo

Lo sviluppo si è articolato in più fasi. Esse sono:

1. Progettazione funzionale
2. Scelta dei componenti
3. Definizione dello schematico
4. Layout e routing delle PCB
5. Progettazione firmware di controllo
6. Stesura codice firmware
7. Testing e debugging

Software utilizzati

I software che sono stati utilizzati per lo sviluppo sono i seguenti:

- **KiCAD** - *Definizione schematico e progettazione PCB.* Software open-source sviluppato e supportato presso il CERN, era già in utilizzo presso il gruppo di ricerca.
  https://www.kicad.org/download/

- **STM32CubeIDE** - *Programmazione controllori STM32.* Software fornito da STMicroelectronics per la configurazione e programmazione dei microcontrollori. Utilizzato nella configurazione del microcontrollore.
  https://www.st.com/en/development-tools/stm32cubeide.html#st-get-software

- **Arduino IDE** - *Programmazione microcontrollore centrale Teensy* - software già utilizzato per la stesura del codice di controllo della piattaforma robotica, utilizzato per sviluppare la libreria per il controllo dei sensori.
  https://www.arduino.cc/en/software

- **Stm32duino** - *Framework programmazione controllori STM32.* Insieme di librerie per la programmazione in ambiente Arduino dei microcontrollori STM32.
  https://github.com/stm32duino/Arduino_Core_STM32

Gerarchia delle cartelle per i progetti e la documentazione

Allo scopo di rendere più facile la fruizione del materiale prodotto si fornisce di seguito la descrizione della struttura gerarchica dello stesso.
Il materiale si trova in una cartella condivisa “Tirocinio Andrea Grillo”, mantenuta sulla piattaforma Dropbox.

- 00 FIGURE
  immagini e figure esplicative utilizzati nella stesura della documentazione
- 01 REPORT
  documentazione e report finale
- 02 MATERIALE DI PARTENZA
  progetti e documentazione progetto già esistente
- 03 TEST E SIMULAZIONI
  report di test e simulazioni svolte sul sistema
- 04 ACQUISTI
  documenti relativi ad acquisti effettuati durante lo sviluppo
- 05 PROGETTI
  cartella contenente i progetti delle PCB realizzate
- 06 SCHEDE ST ESEMPIO
  documentazione schede Nucleo e Evaluation di STMicroelectronics utilizzate come esempio durante lo sviluppo
- 07 DATASHEET
  documentazione e datasheet dei componenti utilizzati nella progettazione delle schede elettroniche
- 08 FIRMWARE
  firmware sviluppato per le schede sensori e libreria per la ricezione dei dati
- 09 CAD
  progetti cad della piattaforma robotica

Progettazione elettronica

Scelta componenti

La prima fase operativa della progettazione del sistema è stata quella della scelta dei componenti da utilizzare.

Microcontrollore

Il componente la cui scelta è stata fondamentale perché centrale nella definizione del funzionamento dell’intero circuito è il microcontrollore.
Di seguito le caratteristiche necessarie ricercate:

- Presenza di seriale *CAN* integrata
- Presenza di convertitore *ADC* integrato con almeno due canali
- Package di ridotte dimensioni, numero ridotto di *PIN*
- Possibilità di saldatura manuale del componente, esclusi quindi package *BGA* e similari

Il primo dispositivo preso in considerazione è stato il modello STM32F103, a causa della sua grande diffusione e della enorme disponibilità di materiale di supporto in rete.
A seguito di richiesta di valutazione della scelta posta all'azienda produttrice STMicroelectronics, è emerso che la serie F dei microcontrollori STM32 è in fase di obsolescenza, ed è quindi raccomandato l'utilizzo di microcontrollori della più recente serie G. Per questo motivo la scelta finale è ricaduta sul modello *STM32G0B1KBT6*, che soddisfa tutti i requisiti richiesti.

Altri componenti

Il resto dei componenti necessari alla realizzazione del circuito si possono dividere nei seguenti gruppi, con i relativi componenti:

- Gestione alimentazione, regolazione e filtraggio
  - regolatore di tensione 3v3 AMS1117
- Circuiteria di contorno del microcontrollore, cristallo oscillatore, condensatori di bypass, connessione di debug e programmazione
  - cristallo oscillatore NX3215SA
- LED di segnalazione alimentazione e programmabile
  - LED di colore rosso e verde
- Comunicazione seriale CAN, transceiver e protezione contro scariche elettrostatiche
  - transceiver CAN MCP2562-H-SN
  - protezione ESDCAN03
- Connettori
  - connettori 2,54mm per laser, sonar, programmazione
  - connettori CAN 20776004YY

Schematico

Di seguito verranno descritte nel dettaglio tutte le componenti dello schematico realizzato, analizzando le scelte effettuate e il dimensionamento dei componenti passivi.

Microcontrollore

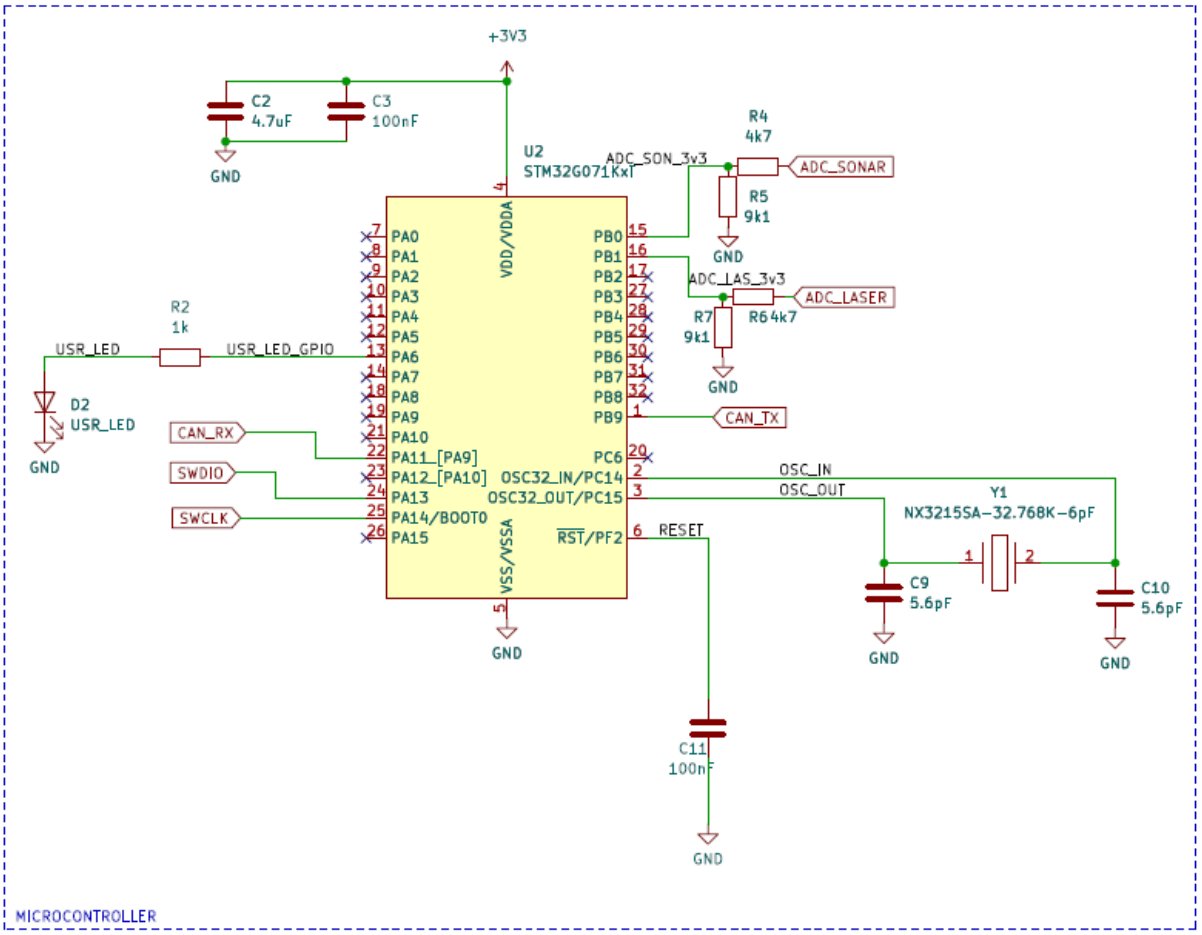

fig 9: schematico sezione microcontrollore

Il microcontrollore ha le seguenti connessioni:

- Cristallo oscillatore, scelto a partire da scheda di esempio NUCLEO di STMicroelectronics, a frequenza 32,768KHz, con relativi condensatori di carico di 5,6pF
- Condensatore 100nF collegato al pin RESET, per evitare oscillazioni che riavvierebbero il microcontrollore
- CAN_TX, CAN_RX segnali CAN verso il transceiver
- SWCLK, SWDIO segnali di programmazione e debug del microcontrollore, verso connettore apposito
- USR_LED_GPIO LED di segnalazione programmabile, collegato a GPIO PA6, con relativa resistenza di limitazione della corrente
- Condensatori di bypass per filtraggio alimentazione, due in parallelo da rispettivamente 4,7uF e 100nF
- ADC_SONAR e ADC_LASER con relativi partitori di tensione, che convertono il range di tensione in ingresso da 0-5V a 0-3,3V

Alimentazione

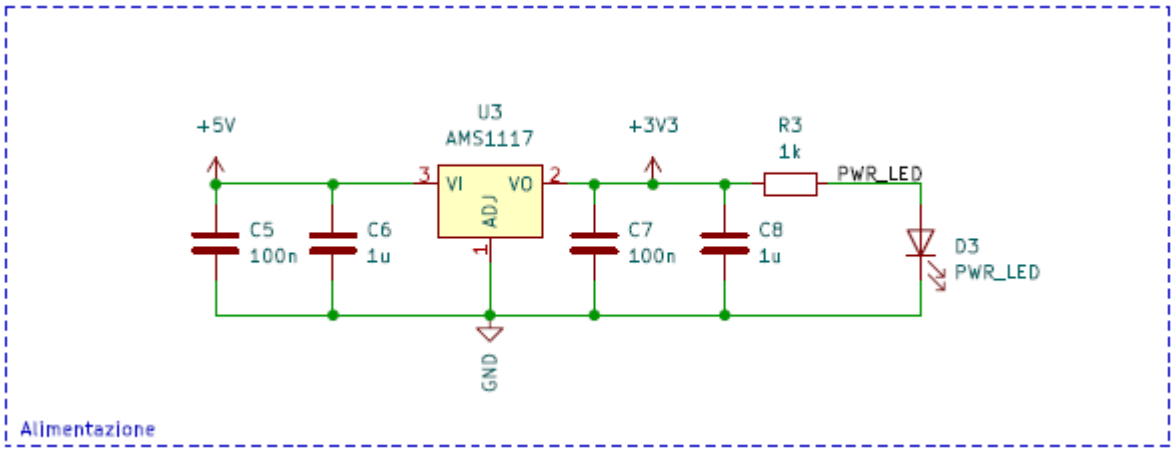

fig 10: schematico sezione alimentazione

Dal connettore che connette le schede sensori al microcontrollore centrale arriva l'alimentazione a 5V. I sensori sono alimentati a 5V, il transceiver CAN ha bisogno di 5V per poter funzionare, mentre il microcontrollore è alimentato a 3,3V, per cui è necessario inserire un regolatore di tensione. È stato scelto il regolatore lineare LDO AMS1117 in quanto molto diffuso e facilmente reperibile. A monte del regolatore sono presenti due condensatori di bypass in parallelo, uno bulk da 1uF e uno di capacità inferiore, di 100nF. Anche a valle del regolatore sono presenti due condensatori bypass delle medesime capacità. Per avere un'indicazione di quando la scheda è alimentata, è inoltre stato collegato un diodo LED direttamente alla linea a 3,3V, in serie con una resistenza da 1kOhm, che limita la corrente che scorre attraverso il diodo.

CAN

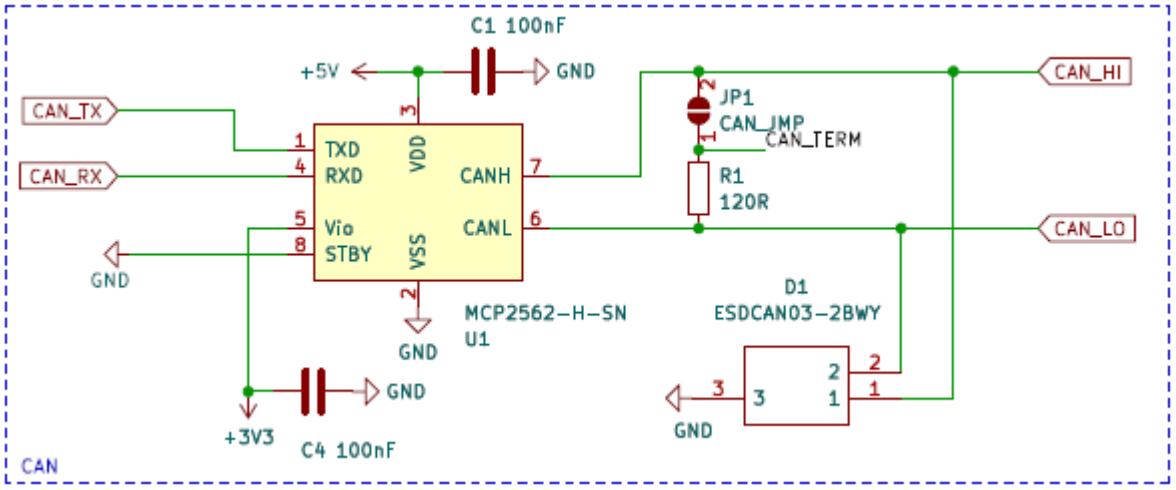

fig 11: schematico sezione CAN

Per quanto riguarda la parte riguardante la seriale CAN, i due componenti principali sono il transceiver, MCP2562-H-SN, e la protezione ESDCAN03-2BWY. Il transceiver funziona anche da level-shifter, tra i 3,3V del microcontrollore e i 5V del bus CAN. Per

questo motivo ha bisogno di entrambe le alimentazioni, con relativi condensatori di bypass da 100nF.
È prevista inoltre la possibilità di configurare la terminazione del bus attraverso l'uso di un jumper a saldare sulla scheda, CAN_JMP. La terminazione di un bus CAN è costituita da una resistenza da 120Ohm tra i due conduttori CAN_HI e CAN_LO. Il chip di protezione ESD, è collegato ai due connettori del bus, e ha un collegamento a massa per poter scaricare eventuali picchi elettrostatici.

Connettori

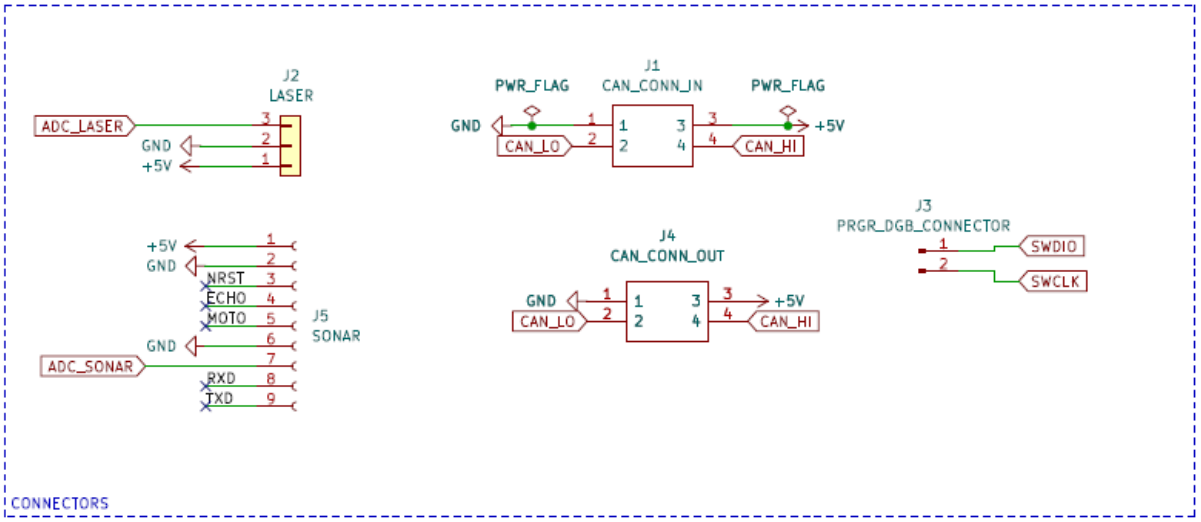

fig 12: schematico sezione connettori

La scelta dei connettori ha seguito quella del precedente progetto per quanto riguarda le connessioni verso il laser e il sonar.
È stato poi aggiunto un connettore per il debug e la programmazione, che è stato mantenuto il più piccolo possibile in quanto non sarà in uso durante il funzionamento normale di tali schede. In tale connettore non sono presenti connessione di massa e alimentazione, in quanto per la programmazione sarà possibile collegare tali segnali attraverso gli altri connettori.
I due connettori CAN, ingresso ed uscita, sono invece stati scelti dal catalogo dell'azienda Molex. Essi forniscono un buon aggancio, consentendo comunque facilità di connessione e disconnessione dei cavi.

Layout e Routing PCB

Successivamente alla scelta dei componenti e alla definizione dello schematico, si è proceduto alla progettazione fisica della scheda. Utilizzando la scheda del progetto precedente come punto di partenza, le dimensioni e le posizioni dei principali connettori sono state scelte in modo da mantenerle uguali. La scheda presenta quindi una forma quadrata di dimensioni 27x27mm. Su un lato è stato posizionato il connettore per il sonar, che sarà anche di sostegno. Su un lato adiacente sono presenti sia il connettore di programmazione che quello relativo al sensore laser.
Un buon criterio per il posizionamento è quello di mantenere tutta la componentistica da un solo lato, in modo da poter consentire la saldatura della scheda facendo uso di una piastra riscaldata o un forno per saldatura in modo semplice. Tale soluzione non

è stata in questo caso praticabile in quanto la scheda ha dimensioni molto ridotte, e non è possibile posizionare tutti i componenti su un solo lato.
Il primo elemento ad essere posizionato è stato il microcontrollore, da cui si è proceduto con l'aggiunta di tutti gli altri circuiti SMD principali, tra cui il transceiver, il regolatore di tensione e i connettori CAN.
Infine sono stati posizionati i componenti passivi necessari al funzionamento del circuito. Si è prestata particolare attenzione ai condensatori di decoupling, posizionati il più vicino possibile ai piedini di alimentazione di ogni componente. Altro componente il cui posizionamento è critico è il cristallo oscillatore, che deve trovarsi vicino ai due relativi piedini del microcontrollore, con il più vicino possibile i due condensatori di load.
A seguire è stato effettuato il routing di tutte le piste di collegamento. Inizialmente si sono impostati dei piani di massa per entrambi i livelli della PCB, e poi si è cominciato instradando i segnali di alimentazione, per cui si è scelta una dimensione della traccia più grande.
Si è cercato di ridurre al minimo il numero di vie passanti, utilizzandole solo dove necessario per impossibilità di routing su un solo piano.
I due piani di massa sono stati collegati insieme da più vie, in modo da rendere il collegamento della massa più stabile possibile e cercare di evitare loop di grandi dimensioni.
Di seguito il layout dei due livelli progettati:

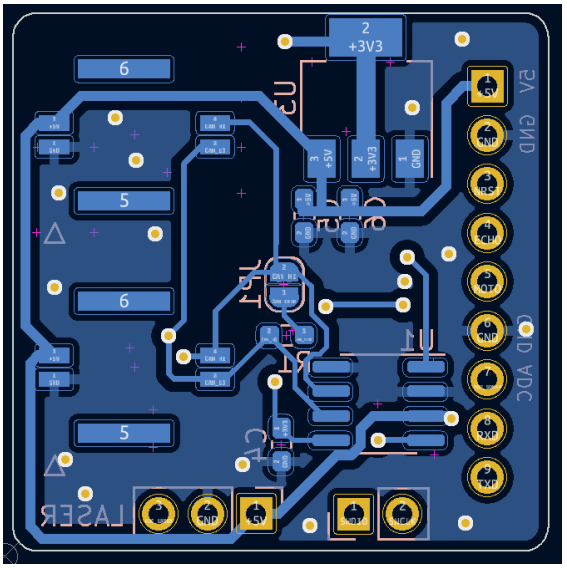

fig 13: layout layer superiore

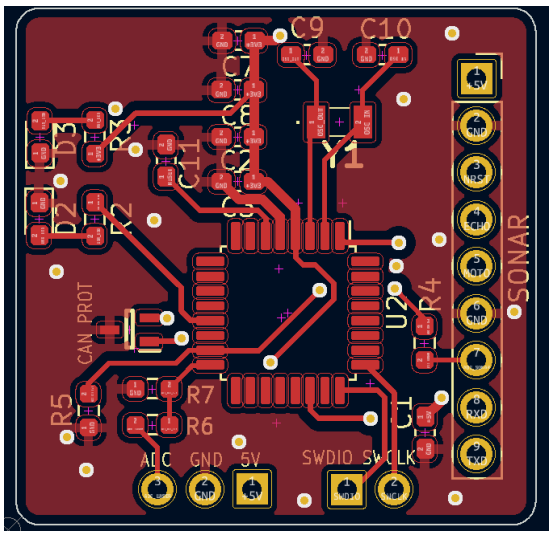

fig 14: layout layer inferiore

Progettazione scheda transceiver

Per la comunicazione con il controllore centrale della piattaforma robotica, è stato necessario progettare una scheda che contenesse il transceiver per la comunicazione CAN.
Il microcontrollore utilizzato supporta la seriale CAN, ma ha bisogno di un chip esterno per la traduzione dei segnali TX e RX nei segnali differenziali CAN_HI e CAN_LO.
Lo schematico di tale scheda aggiunta è analogo a quello della parte CAN della scheda sensori.

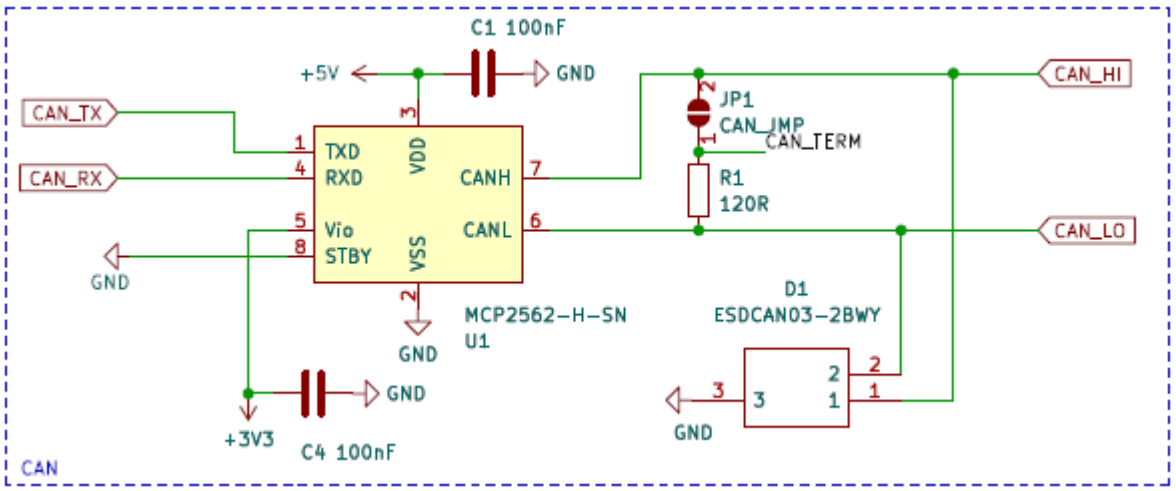

fig 15: schematico scheda transceiver

Per quanto riguarda il layout, si è fatto in modo che tale scheda potesse adattarsi alla PCB già presente per il controllo della piattaforma, agganciandosi a dei connettori appositamente lasciati liberi per eventuali espansioni e fissandosi “a castello” attraverso l’uso di un foro di montaggio della PCB esistente, con l’aggiunta di un distanziale.
Sulla scheda è presente, oltre al transceiver,
Di seguito il layout della scheda progettata:

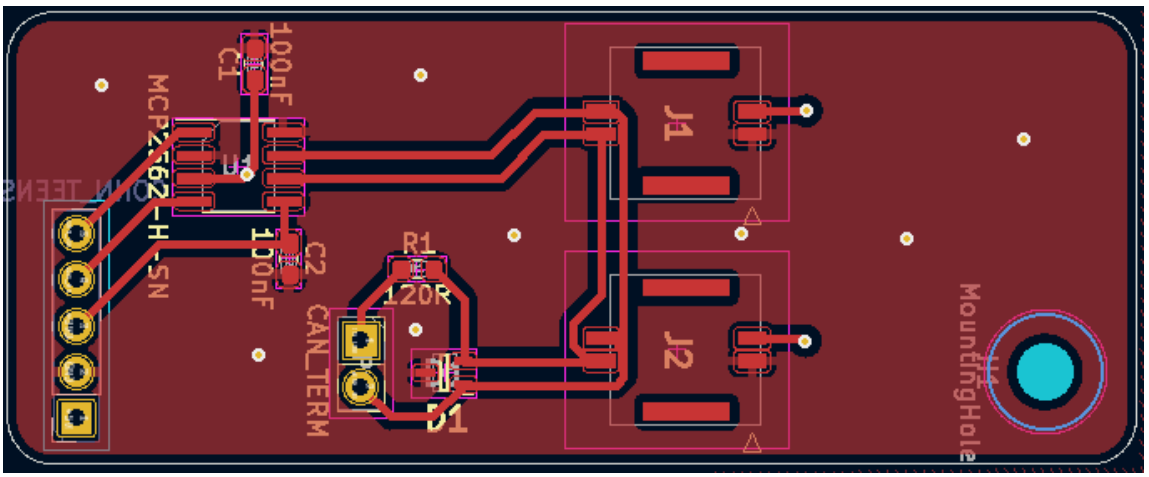

fig 16: layout scheda transceiver

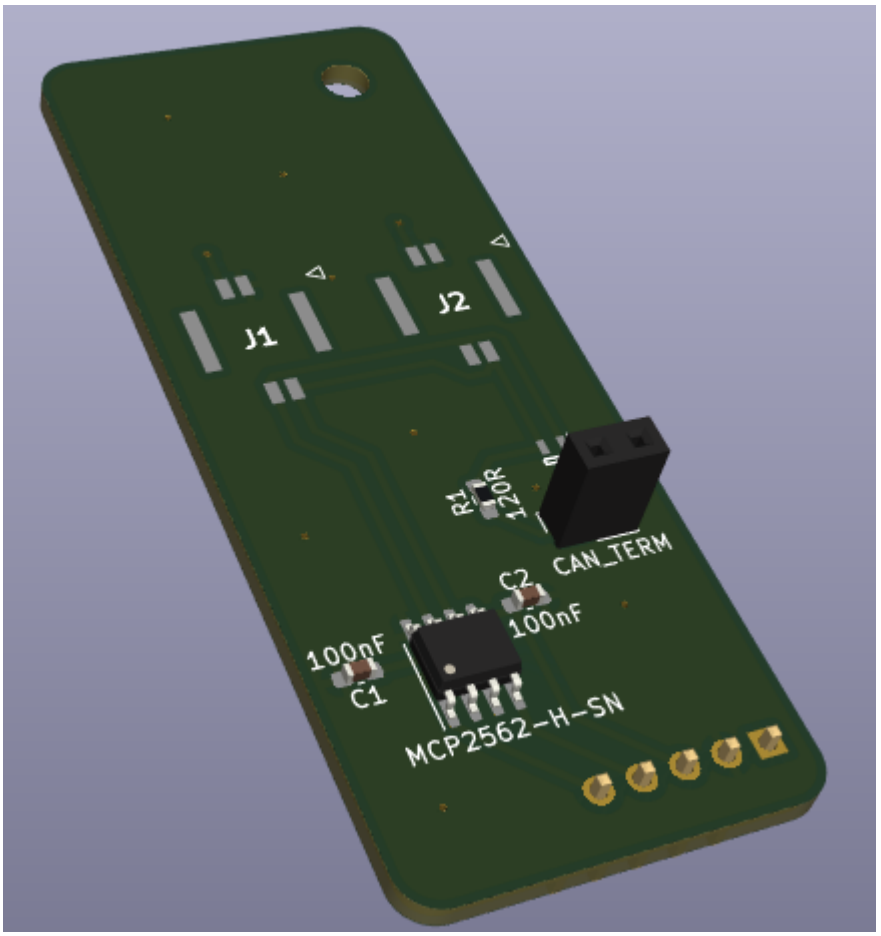

fig 17: render scheda transceiver

Progettazione firmware di controllo

In una prima versione del firmware per il microcontrollore a bordo della scheda di acquisizione verranno implementate le seguenti funzioni:

- impostazione dinamica in tempo reale delle soglie di allarme

- impostazione dinamica del proprio identificatore in base alla posizione di montaggio
- salvataggio impostazioni su Flash a bordo per mantenimento anche da spento
- invio messaggio di allarme su superamento soglie
- invio della lettura dei dati dei sensori su richiesta del controllore centrale

Per quanto riguarda il microcontrollore centrale, si procederà all'implementazione di una libreria apposita per la gestione dei sensori che avrà le seguenti funzionalità:

- impostazione delle soglie di tutti i sensori ad un valore
- ricezione allarme
- query del valore di distanza ad un sensore specifico

Date le suddette specifiche, si è proceduto con il definire l'architettura del firmware di controllo per il microcontrollore STM32, che deve lavorare in cooperazione con il firmware presente sul microcontrollore centrale della scheda robotica, per cui viene inoltre realizzata una libreria apposita.

Scelta del framework di sviluppo

Si è scelto di utilizzare il framework di sviluppo STM32duino per il codice delle schede sensori, che rende possibile la scrittura e il caricamento del codice attraverso l'IDE Arduino.
Dato che l'IDE Arduino è già stato utilizzato per il codice esistente della piattaforma robotica, si ritiene che il suo utilizzo renda più coerenti gli eventuali ulteriori sviluppi, e che faciliti eventuali modifiche al codice.
Ciononostante, si è reso necessario il ricorso alle funzioni native STM32 per quanto riguarda la gestione del CANBUS, in quanto non vi è una libreria STM32duino per la gestione di tale libreria.
Per quanto riguarda la libreria per la ricezione dei messaggi, il codice già presente è stato scritto facendo uso dell'IDE Arduino per la programmazione della Teensy, quindi si è scelto di realizzare una libreria compatibile con tale codice. La libreria è stata realizzata seguendo il paradigma della programmazione a oggetti, nel linguaggio C++.

Flowchart firmware

Le operazioni che le schede sensori devono svolgere possono essere schematizzate nei 3 blocchi rappresentati in figura, e devono essere eseguite ciclicamente.

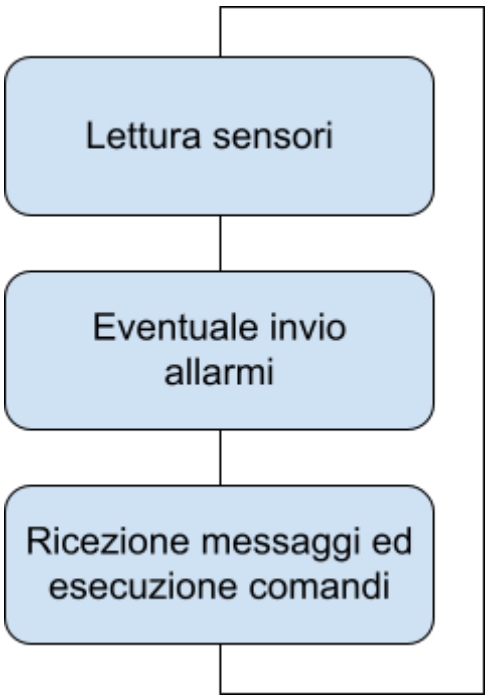

fig 18: flowchart firmware

Il microcontrollore deve effettuare la lettura dall'ADC per conoscere i valori misurati dai sensori, in seguito procede alla verifica delle soglie. Se uno dei valori letti supera le soglie impostate viene inviato l'allarme al microcontrollore centrale.
In seguito viene controllato l'arrivo di messaggi sulla seriale CAN. I messaggi che possono arrivare possono essere delle seguenti tipologie:

- Set ID Can - messaggio di impostazione dell'ID can della scheda sensori
- Set threshold - messaggio di impostazione delle soglie
- Dist request - messaggio di richiesta dei valori misurati dai sensori

Protocollo di comunicazione

Per la comunicazione sul bus CAN delle schede sensori con il microcontrollore centrale è stato definito un semplice protocollo di comunicazione, basato su un totale di 7 tipologie di messaggi, la cui struttura e composizione viene illustrata nella tabella a seguire:

Nome	**ID**	**DLC**	**Data**
SET_ID_CAN	0x12	2	0: CMD | 1: CAN_ID
SET_THRESHOLD	0x13	7	0: CMD | 1,2: Y_TH | 3,4: R_TH, 5,6: L_TH
DIST_REQUEST	0x14	1	0: CMD
ALARM_YELLOW	0x15	4	0: CMD | 1: CAN_ID | 2,3: DIST_SONAR
ALARM_RED	0x16	4	0: CMD | 1: CAN_ID | 2,3: DIST_SONAR

ALARM_LASER	0x17	2	0: CMD | 1: CAN_ID
DIST_ANS	0x18	6	0: CMD | 1: CAN_ID | 2,3: LASER | 4,5: SONAR

I comandi SET_ID_CAN, SET_THRESHOLD e DIST_REQUEST sono utilizzati dal controllore centrale per interfacciarsi con le schede sensori, mentre i restanti sono utilizzati dalle schede sensori per mandare gli allarmi e rispondere alle richieste del controllore centrale.

Libreria Teensy

La libreria sviluppata consta di due file:

- Sensors.h
- Sensors.cpp

Nel primo sono presenti le definizioni delle costanti necessarie al funzionamento del sistema, nonché la definizione della classe Sensors, che espone le funzioni pubbliche necessarie.
Il codice scritto dipende dipende dalla libreria FlexCan_T4, facente parte della toolchain Teensy per l'IDE Arduino. Essa permette la gestione della seriale CAN.
La classe Sensors è così composta:

- Metodi pubblici:
  - Costruttore - inizializza le soglie
  - begin - inizializza la seriale CAN, invia le soglie ai sensori
  - setThreshold - imposta le soglie e le invia ai sensori
  - update - controllo arrivo allarmi e richiamo funzione di callback - da richiamare ad ogni ciclo del loop
  - requestDistance - invio di una richiesta di distanze ad un sensore
  - getThreshold - ritorna le soglie impostate
- Metodo privato:
  - sendThreshold - metodo interno per l'invio sulla seriale CAN delle soglie ai sensori
- Attributi privati:
  - threshold - soglie attuali
  - callback - puntatore alla funzione di callback
  - canBus - oggetto di classe FlexCAN_T4 per la gestione della seriale CAN

Sono stati inoltre definiti i seguenti tipi di dato, con i relativi membri:

- threshold_t
  - yellowThreshold
  - redThreshold
  - laserThreshold
- dist_t
  - distLaser
  - distSonar
  - error
- callback_t

Le costanti definite sono:

- CAN_BAUDRATE - baud rate della seriale CAN
- MY_ID - ID per la seriale CAN
- REQ_TIMEOUT - timeout per la risposta alla richiesta di distanza ad un sensore
- enum sensori - enum che contiene tutti gli ID CAN delle schede sensori

Integrazione nella logica di controllo

Una volta reso funzionante il sistema, è necessario integrarlo nel sistema di controllo esistente. I dati che vengono ricevuti dai sensori devono dunque essere elaborati ed utilizzati nell'elaborazione della traiettoria e quindi nell'impostazione dei valori di riferimento dei motori.
Un'analisi su una possibile implementazione dell'elaborazione di tali dati viene fornita di seguito.
Uno schema del sistema in esame è il seguente:

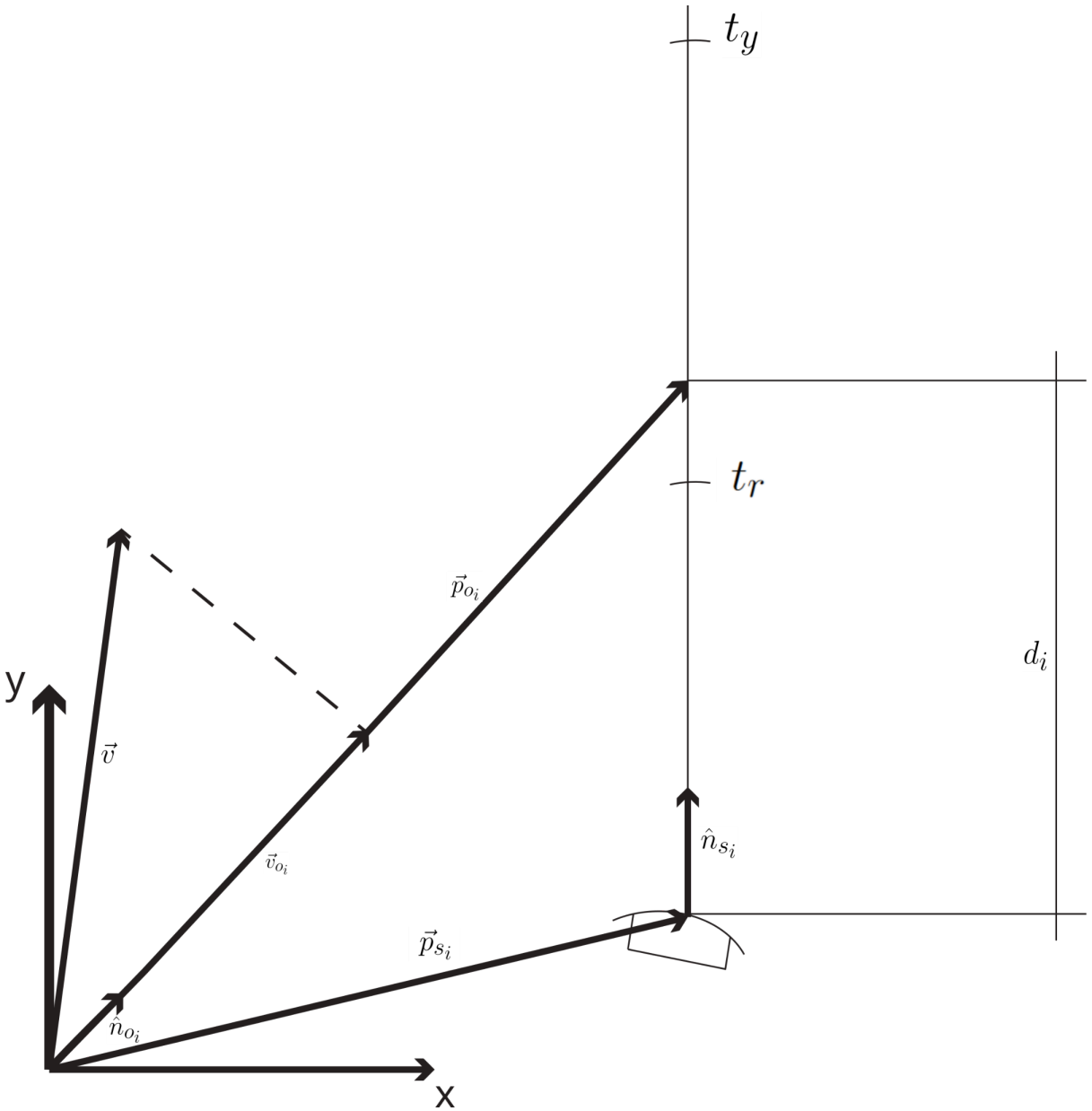

fig 19: schema del sistema

Dove il sistema di riferimento xy è solidale alla piattaforma robotica, con l'origine nel suo centro. $\vec{p}_{si}$ indica la posizione del sensore *i*-esimo rispetto al centro. Tale sensore effettua una misurazione di distanza lungo la retta normale ad esso, identificata dal

versore $\hat{n}_{si}$. Lungo tale retta vengono identificate le due soglie t_r e t_y (red/yellow threshold).
Supponiamo dunque che un ostacolo venga rilevato dal sensore sonar alla distanza d_i. Il vettore poi posizione dell'ostacolo rispetto al centro della piattaforma è dato dalla somma vettoriale di $\vec{p}_{si}$ e di $d_i \cdot \hat{n}_{si}$.
Il vettore $\vec{v}$ indica la velocità istantanea della piattaforma robotica, $\vec{v}_{oi}$ è la proiezione di tale vettore lungo la direzione di $\vec{p}_{oi}$.
Risulta quindi utile identificare i valori di posizione e i rispettivi versori normali degli 8 sensori installati a priori, in modo da avere tali dati pronti per i calcoli successivi.
I dati sono stati estratti dalla tavola della vista in pianta della piattaforma robotica, di seguito:

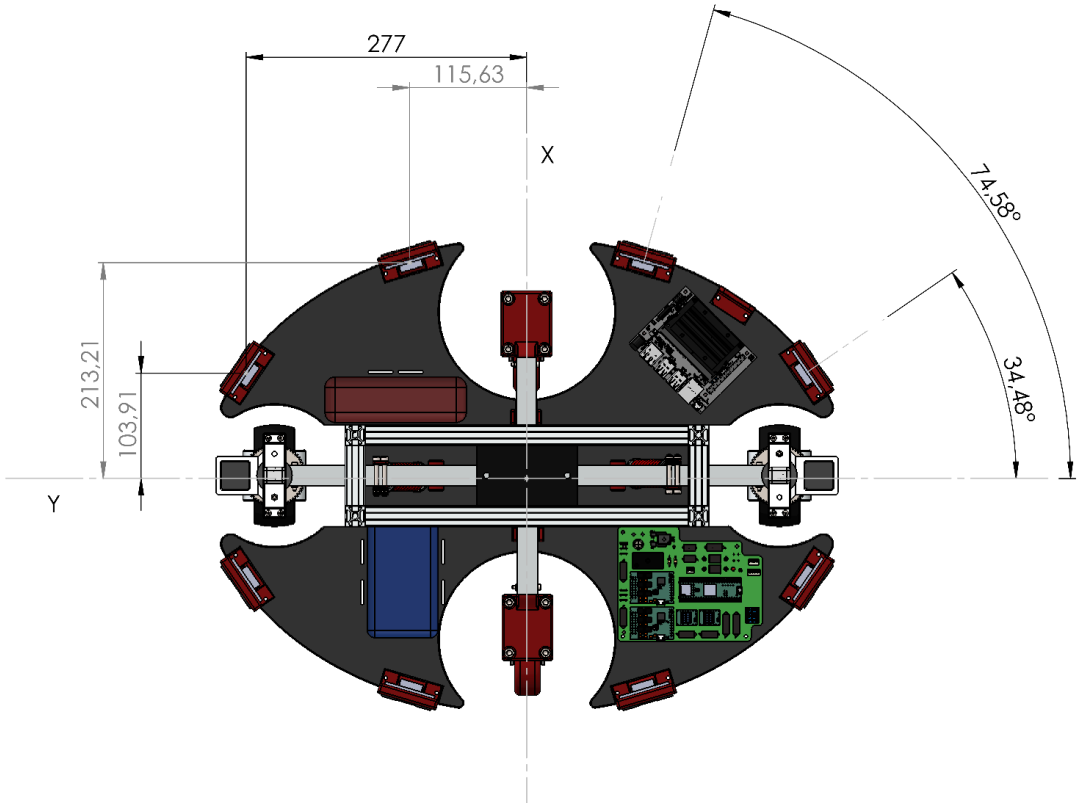

fig 20: tavola pianta Paquitop

I dati necessari risultano quindi essere:

Id sensore	Posizione sensore (x/y) [mm]	Versore normale (x,y) [mm]
1	(-103.91 / -277)	(-0.5661 / -0.8243)
2	(-115.63 / -213.21)	(-0.964 / -0.2659)
3	(-115.63 / 213.21)	(-0.964 / 0.2659)
4	(-103.91 / 277)	(-0.5661 / 0.8243)
5	(103.91 / 277)	(0.5661 / 0.8243)
6	(115.63 / 213.21)	(0.964 / 0.2659)

7	(115.63 / -213.21)	(0.964 / -0.2659)
8	(103.91 / -277)	(0.5661 / -0.8243)

Procedura all'arrivo di un allarme

La procedura che viene scatenata all'arrivo di un allarme si articola in diversi passi. Nel caso di un allarme relativo al sensore sonar, i passi sono i seguenti:

1. Calcolo del vettore posizione dell'ostacolo $\vec{p_{oi}} = \vec{p_{si}} + d_i \cdot \vec{n_{si}}$
2. elaborazione del segno del prodotto scalare tra la velocità della piattaforma robotica e la posizione dell'oggetto
3. in caso di segno positivo, la velocità viene riscalata di un fattore variabile in base al tipo di allarme, se giallo o rosso, rispettivamente fattori n e m
4. se il segno è negativo, nessuna azione viene eseguita

I passi nel caso di ricezione di un allarme relativo al sensore laser sono invece:

1. Il vettore posizione dell'ostacolo viene computato a priori, in quando si considera la distanza rilevata dal laser come una distanza fissa
2. Se il prodotto scalare tra la velocità della piattaforma e il vettore posizione è positivo, la velocità viene impostata a *0* e il movimento viene arrestato
3. Se il segno è negativo, la velocità viene riscalata di un fattore *k*

Valori plausibili per i fattori n, m e k possono essere:

- $n = 2$
- $m = 10$
- $k = 5$

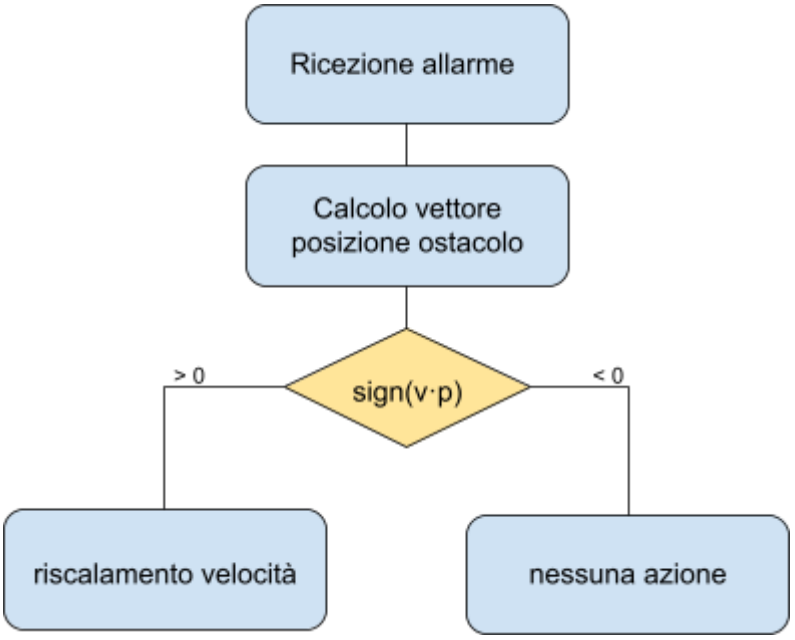

fig 17: flowchart ricezione allarme

Simulazione allarmi

Al fine di fare una simulazione di come il sistema risponde alla rilevazione di un ostacolo, è stato sviluppato un foglio di calcolo basato su Excel, che permette di calcolare, al variare dei parametri, come il sistema agisce sulle velocità impostate. Gli input sono l'ID del sensore e la distanza rilevata da esso. A partire da tali dati il sistema fa i calcoli descritti al precedente paragrafo e riporta come verrebbero riscalate le velocità in uscita dall'algoritmo.
Tale file è disponibile nella cartella Dropbox, sottocartella '03 TEST E SIMULAZIONI'.
Di seguito viene riportata una simulazione eseguita a partire da allarmi e velocità iniziali generate casualmente.
Tale simulazione è stata svolta utilizzando i seguenti parametri:

- $n = 2$
- $m = 10$
- $sogliaRosso = 200$

dati input					*dati calcolati*				
ID sensore	*Velocità (x/y) [mm/s]*		*Distanza [mm]*	*Tipo allarme*	*Posizione ostacolo (x/y) [mm]*		*Velocità · Posizione [mm]*	*Velocità ricalcolate (x/y) [mm/s]*	
4	5	-42	136	rosso	-180,89	389,10	-17246,89	0,5	-4,2
8	-84	21	145	rosso	185,99	-396,52	-23950,53	-8,4	2,1
4	27	-79	19	rosso	-114,66	292,66	-26216,25	2,7	-7,9
5	-3	55	61	rosso	138,44	327,28	17585,20	-0,3	5,5
4	-86	1	14	rosso	-111,83	288,54	9906,38	-8,6	0,1
2	45	-28	287	giallo	-392,29	-289,52	-9546,75	22,5	-2,8
3	-30	4	262	giallo	-368,19	282,87	12177,44	-15	0,4
3	10	-83	260	giallo	-366,27	282,34	-27097,25	5	-8,3
5	-28	60	30	rosso	120,893	301,72	14718,73	-2,8	6

Firmware

Schede sensori

```
/**
 * Firmware for sensor boards
 * Paquitop
 *
 * To be run on: STM32G0B1KBT6 on custom PCB
 *
 * Author: Andrea Grillo S282802
 *
 */
```

```
#include <EEPROM.h>

#define PIN_LASER       PB1   // pin of the mcu to which the laser
is connected
#define PIN_SONAR       PB0   // pin of the mcu to which the sonar
is connected
#define PIN_LED         PA6   // pin of the mcu to which the LED is
connected
#define MAIN_ID         0x01  // can ID of the main board
(receiver)
#define ALARM_TIMEOUT 2000  // minimum time between the alarms to
be sent

// CAN commands
enum {
  SET_ID_CAN = 0x12,
  SET_THRESHOLD,
  DIST_REQUEST,
  ALARM_YELLOW,
  ALARM_RED,
  ALARM_LASER,
  DIST_ANS
};

// global variables to handle CANBUS
FDCAN_HandleTypeDef hfdcan1;
FDCAN_RxHeaderTypeDef RxHeader;
uint8_t RxData[8];
FDCAN_TxHeaderTypeDef TxHeader;
uint8_t TxData[8];

// CAN ID of this board
uint8_t myCanId;
// thresholds
uint16_t yellowThreshold, redThreshold, laserThreshold;
// flag to activate sending of distance
uint8_t distRequested = 0;
// time
long alarmTime = -1;

// prototipi delle funzioni
static void MX_FDCAN1_Init();
static void readFromFlash();
static void writeToFlash();

void setup() {
```

```
  /* read config from FLASH memory*/
  readFromFlash();

  /* init CAN serial */
  MX_FDCAN1_Init();

  /* setting OUTPUT led */
  pinMode(PIN_LED, OUTPUT);
  digitalWrite(PIN_LED,HIGH);
  delay(500);
  digitalWrite(PIN_LED,LOW);
}

void loop() {
  static int laser, sonar, alarmSend;

  /* timeout update to send alarms */
  if(alarmTime != -1 && (millis() - alarmTime) >= ALARM_TIMEOUT)
alarmTime = -1;

  /* reading data from sensors */
  laser = analogRead(PIN_LASER);
  sonar = analogRead(PIN_SONAR);

  /**
   * Conditions to be met to send an alarm:
   *  - time elapsed from last alarm sent >= TIMEOUT_ALARM
   *  - RED: sonar < redThreshold
   *  - YELLOW: redThreshold < sonar < yellowThreshold
   *  - LASER: laser > laserThreshold
   */
  if(alarmTime == -1) {
      alarmSend = 1;
      TxData[1] = myCanId;
      TxHeader.DataLength = FDCAN_DLC_BYTES_2;

      if(laser > laserThreshold) /* Laser alarm */
      TxData[0] = ALARM_LASER;
      else if(sonar < redThreshold)  /* RED alarm */
      TxData[0] = ALARM_RED;
      else if(sonar < yellowThreshold) /* YELLOW alarm */
      TxData[0] = ALARM_YELLOW;
      else alarmSend = 0;

      if(alarmSend) {
      if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &TxHeader,
```

```
TxData) != HAL_OK)
      Error_Handler();

      alarmTime = millis();
      digitalWrite(PIN_LED,HIGH);
      delay(200);
      digitalWrite(PIN_LED,LOW);
      }
  }

  /**
   * If we received a distance request
   * send requested data
   */
  if(distRequested) {
      TxHeader.DataLength = FDCAN_DLC_BYTES_6;

      TxData[0] = DIST_ANS;
      TxData[1] = myCanId;
      TxData[2] = laser >> 8;
      TxData[3] = laser;
      TxData[4] = sonar >> 8;
      TxData[5] = sonar;

      if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &TxHeader,
TxData) != HAL_OK)
      Error_Handler();

      distRequested = 0;
      digitalWrite(PIN_LED,HIGH);
      delay(200);
      digitalWrite(PIN_LED,LOW);
  }
}

/**
 * CAN init
 */
static void MX_FDCAN1_Init(void) {
  /**
   * CAN settings for timings and clock division
   * generated with STM32CUBEIDE. To be tuned with working boards.
   */
  hfdcan1.Instance = FDCAN1;
  hfdcan1.Init.ClockDivider = FDCAN_CLOCK_DIV1;
  hfdcan1.Init.FrameFormat = FDCAN_FRAME_FD_BRS;
```

```
  hfdcan1.Init.Mode = FDCAN_MODE_NORMAL;
  hfdcan1.Init.AutoRetransmission = ENABLE;
  hfdcan1.Init.TransmitPause = ENABLE;
  hfdcan1.Init.ProtocolException = DISABLE;
  hfdcan1.Init.NominalPrescaler = 1;
  hfdcan1.Init.NominalSyncJumpWidth = 16;
  hfdcan1.Init.NominalTimeSeg1 = 63;
  hfdcan1.Init.NominalTimeSeg2 = 16;
  hfdcan1.Init.DataPrescaler = 1;
  hfdcan1.Init.DataSyncJumpWidth = 4;
  hfdcan1.Init.DataTimeSeg1 = 5;
  hfdcan1.Init.DataTimeSeg2 = 4;
  hfdcan1.Init.StdFiltersNbr = 1;
  hfdcan1.Init.ExtFiltersNbr = 0;
  hfdcan1.Init.TxFifoQueueMode = FDCAN_TX_FIFO_OPERATION;
  if (HAL_FDCAN_Init(&hfdcan1) != HAL_OK)
      Error_Handler();

  /* Start the FDCAN module */
  if (HAL_FDCAN_Start(&hfdcan1) != HAL_OK)
      Error_Handler();

  if (HAL_FDCAN_ActivateNotification(&hfdcan1,
FDCAN_IT_RX_FIFO0_NEW_MESSAGE, 0) != HAL_OK)
      Error_Handler();

  /* Prepare Tx Header */
  TxHeader.IdType = FDCAN_STANDARD_ID;
  TxHeader.TxFrameType = FDCAN_DATA_FRAME;
  TxHeader.ErrorStateIndicator = FDCAN_ESI_ACTIVE;
  TxHeader.BitRateSwitch = FDCAN_BRS_OFF;
  TxHeader.FDFormat = FDCAN_CLASSIC_CAN;
  TxHeader.TxEventFifoControl = FDCAN_NO_TX_EVENTS;
  TxHeader.MessageMarker = 0;
  TxHeader.Identifier = MAIN_ID;
}


/**
 * Callback function to receive CAN messages
 * WARNING: do NOT change function name
*/
void HAL_FDCAN_RxFifo0Callback(FDCAN_HandleTypeDef *hfdcan,
uint32_t RxFifo0ITs)
{
  if((RxFifo0ITs & FDCAN_IT_RX_FIFO0_NEW_MESSAGE) == RESET)
```

```
      return;

  /* Retrieve Rx messages from RX FIFO0 */
  if (HAL_FDCAN_GetRxMessage(hfdcan, FDCAN_RX_FIFO0, &RxHeader,
RxData) != HAL_OK)
      Error_Handler();

  /* We can handle only standard messages, not FD */
  if(RxHeader.IdType != FDCAN_STANDARD_ID)
      return;

  /* Data Parsing */
  if(RxData[0] == SET_ID_CAN && RxHeader.DataLength ==
FDCAN_DLC_BYTES_2) { /* Set ID CAN */
      myCanId = RxData[1];
      writeToFlash();
  } else if(RxData[0] == SET_THRESHOLD && RxHeader.DataLength ==
FDCAN_DLC_BYTES_7) { /* SET_THRESHOLD */
      yellowThreshold = RxData[1] << 8 | RxData[2];
      redThreshold = RxData[3] << 8 | RxData[4];
      laserThreshold = RxData[5] << 8 | RxData[6];
      writeToFlash();
  } else if(RxHeader.StdId == myCanId && RxData[0] == DIST_REQUEST
&& RxHeader.DataLength == FDCAN_DLC_BYTES_1){ /* DIST_REQUEST */
      distRequested = 1;
  }
}

/* read config from FLASH */
void readFromFlash() {
  eeprom_buffer_fill();
  myCanId = eeprom_buffered_read_byte(1);
  yellowThreshold = eeprom_buffered_read_byte(2) << 8 |
eeprom_buffered_read_byte(3);
  redThreshold = eeprom_buffered_read_byte(4) << 8 |
eeprom_buffered_read_byte(5);
  laserThreshold = eeprom_buffered_read_byte(6) << 8 |
eeprom_buffered_read_byte(7);
}

/* write config on FLASH */
void writeToFlash() {
  eeprom_buffered_write_byte(1, myCanId);

  eeprom_buffered_write_byte(2, yellowThreshold >> 8);
  eeprom_buffered_write_byte(3, yellowThreshold);
```

```
  eeprom_buffered_write_byte(4, redThreshold >> 8);
  eeprom_buffered_write_byte(5, redThreshold);

  eeprom_buffered_write_byte(6, laserThreshold >> 8);
  eeprom_buffered_write_byte(7, laserThreshold);

  eeprom_buffer_flush();
}
```

Libreria Teensy

Sensors.h

```
/**
 * Sensors Class to manage CANBUS lowlevel sensors.
 */

#ifndef SENSORS_CLASS_H
#define SENSORS_CLASS_H

#include "FlexCAN_T4.h"

#define CAN_BAUDRATE 250000
#define MY_ID 0x01
#define REQ_TIMEOUT 1000

enum {
  sensor1   = 0x11,
  sensor2   = 0x12,
  sensor3   = 0x13,
  sensor4   = 0x14,
  sensor5   = 0x15,
  sensor6   = 0x16,
  sensor7   = 0x17,
  sensor8   = 0x18,
};

enum {
  SET_ID_CAN = 0x12,
  SET_THRESHOLD,
  DIST_REQUEST,
  ALARM_YELLOW,
  ALARM_RED,
  ALARM_LASER,
```

```
  DIST_ANS
};

typedef struct {
  uint16_t yellowThreshold, redThreshold, laserThreshold;
} threshold_t;

typedef struct {
  uint16_t distLaser, distSonar;
  bool error = false;
} dist_t;


typedef void (*callback_t)(int sensor, int threshold);

class Sensors {
  public:
      Sensors(threshold_t threshold, callback_t c);
      void begin();
      void setThreshold(threshold_t threshold);
      void update();
      dist_t requestDistance(int sensorId);
      threshold_t getThreshold();

  private:
      threshold_t threshold;
      callback_t callback;
      FlexCAN_T4<CAN1, RX_SIZE_256, TX_SIZE_16> canBus;
      void sendThreshold();

};

#endif
```

Sensors.cpp

```
#include "Sensors.h"
#include <stdio.h>

/**
 * Class Sensors constructor.
 */
```

```
Sensors::Sensors(threshold_t threshold, callback_t c)
      : threshold(threshold), callback(c) {}


/**
 * Initialization function.
 * CANBUS setup and sending of thresholds to every sensor.
 */
void Sensors::begin() {
  canBus.begin();
  canBus.setBaudRate(CAN_BAUDRATE);

  sendThreshold();
}

/**
 * Update function, to be called for every loop cycle.
 * Checks if any alarm has arrived and calls callback function.
 */
void Sensors::update() {
  CAN_message_t msg;

  if(canBus.read(msg) && msg.id == MY_ID &&
      (msg.buf[0] == ALARM_RED || msg.buf[0] == ALARM_YELLOW ||
msg.buf[0] == ALARM_LASER)) {
      callback(msg.buf[0],msg.buf[1]);
  }
}

/**
 * Function to set threshold for all sensors.
 * Updates the private attribute and sends the threshold
 * to all the sensors.
 */
void Sensors::setThreshold(threshold_t threshold) {
      this->threshold = threshold;

      sendThreshold();
}

/**
 * Private function to send the threshold to a specific sensor.
 */
void Sensors::sendThreshold() {
  CAN_message_t msg;
```

```
  msg.buf[0] = SET_THRESHOLD;
  msg.buf[1] = threshold.yellowThreshold >> 8;
  msg.buf[2] = threshold.yellowThreshold;
  msg.buf[3] = threshold.redThreshold >> 8;
  msg.buf[4] = threshold.redThreshold;
  msg.buf[5] = threshold.laserThreshold >> 8;
  msg.buf[6] = threshold.laserThreshold;
  msg.len = 7;

  canBus.write(msg);
}

/**
 * Function to request measured distance to a specific sensor.
 * Sends a request message and then waits up to REQ_TIMEOUT
milliseconds
 * for the answer, else returns error constant DIST_ERR.
 *
 * WARNING:
 * the function is blocking, if alarm messages arrive between the
request
 * and the response, they would be discarded.
 */
dist_t Sensors::requestDistance(int sensorId) {
  CAN_message_t msg;
  dist_t distance;

  msg.id = sensorId;
  msg.buf[0] = DIST_REQUEST;
  msg.len = 1;

  canBus.write(msg);

  long startTime = millis();
  bool flag = true;

  while(flag) {
      if(canBus.read(msg) && msg.id == MY_ID && msg.buf[0] ==
DIST_ANS && msg.buf[1] == sensorId) {
      flag = false;
      }

      if(millis() - startTime > REQ_TIMEOUT) {
      distance.error = true;
      return distance;
      }
```

```
  }

  distance.distLaser = msg.buf[2] << 8 | msg.buf[3];
  distance.distSonar = msg.buf[4] << 8 | msg.buf[5];

  return distance;
}

/**
 * Getter function to return actual threshold.
 */
threshold_t Sensors::getThreshold() {
      return threshold;
}
```

Codice di prova

```
#include "Sensors.h"

void alarmCallback(int, int);

// istanza della classe Sensors, imposto soglie di default e
funzione di callBack per la ricezione degli allarmi
Sensors sensors({.yellowThreshold = 7, .redThreshold = 9,
.laserThreshold = 45}, alarmCallback);

void setup() {
  // inizializzo la classe sensori
  sensors.begin();

  // controllo le soglie gia impostate
  threshold_t checkSoglie = sensors.getThreshold();
  Serial.print("Soglie impostate: giallo: ");
  Serial.print(checkSoglie.yellowThreshold);
  Serial.print(", rosso: ");
  Serial.print(checkSoglie.redThreshold);
  Serial.print(", laser: ");
  Serial.println(checkSoglie.laserThreshold);

  // variazione delle soglie
  sensors.setThreshold({.yellowThreshold = 1, .redThreshold = 2,
.laserThreshold = 27});

  // richiesta distanze ad un sensore
  dist_t distanza = sensors.requestDistance(sensor1);
```

```
  if(distanza.error) {
      Serial.println("Non è stato possibile rilevare la distanza
dal sensore 1.");
  } else {
      Serial.print("Distanze attualmente rilevate dal sensore 1:
laser: ");
      Serial.print(distanza.distLaser);
      Serial.print(", sonar: ");
      Serial.println(distanza.distSonar);
  }
}

void loop() {
  // da chiamare ad ogni ciclo di loop, verifica se arrivati nuovi
messaggi di allarme
  sensors.update();
}

// funzione di callback, richiamata quando si riceve un allarme
void alarmCallback(int sensorId, int soglia){
  if(soglia == ALARM_YELLOW) {
      Serial.print("Ricevuto allarme giallo da sensore: ");
      Serial.println(sensorId);
  }
  else if(soglia == ALARM_RED) {
      Serial.print("Ricevuto allarme rosso da sensore: ");
      Serial.println(sensorId);
  } else if(soglia == ALARM_LASER) {
      Serial.print("Ricevuto allarme laser da sensore: ");
      Serial.println(sensorId);
  }


  if(sensorId == sensor3) {
      Serial.println("L'allarme è stato ricevuto dal sensore 3");
  }
}
```

Messa in funzione del sistema

Per la messa in funzione del sistema è necessaria una serie di attività volte al debugging e al tuning dello stesso.
Di seguito un elenco di tali attività, da effettuare una volta ricevute le schede sensori prodotte:

1. Verifica generale schede, saldatura header, eventuale jumper terminazione CAN
2. Test caricamento firmware schede (ST-LINK)
3. Acquisto ST-LINK https://www.mouser.it/ProductDetail/STMicroelectronics/ST-LINK-V2?qs=H4BOwPtf9MC1sDQ8j3cy4w%3D%3D&mgh=1&vip=1
4. Verifica configurazione clock
5. Verifica funzionamento LED di segnalazione
6. Verifica funzionamento seriale CAN (configurazione time quanta)
7. Verifica lettura corretta ADC
8. Montaggio sistema - supporti 3D, collegamento cavi
9. Test generale dell'invio degli allarmi
10. Scelta soglie (giallo/rosso sonar, laser)
11. Integrazione nel sistema di controllo (scelta parametri di riscalamento)
12. Tuning dei parametri