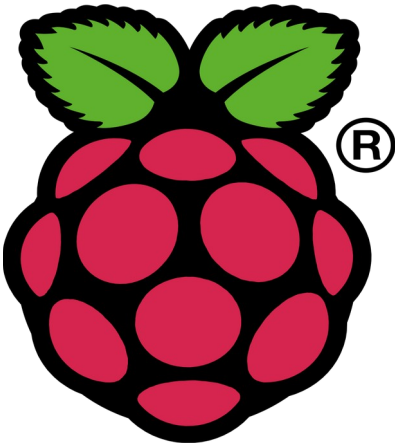
MetalPhoenix



Maze Solver

Hardware and Software documentation

Nome e cognome	Matricola
Stefano Fattore	259869
Marco Pinori	

Indice

1. Introduzione

In questa documentazione viene spiegato come il software interagisce con l’hardware.

Viene descritto, punto per punto, come è stato realizzato il robot ed i suoi componenti, il remote controller, la tecnologia di comunicazione fra le varie parti e il labirinto fisico per la fase di testing.

In questa fase introduttiva viene anche analizzata una problematica relativa alla rotazione per mezzo del sensore MPU6050 (giroscopio).

• Problematica

Il sensore MPU6050 (giroscopio) da noi adoperato presenta un problema, per cui i valori dell’asse Y, ovvero l’asse che rappresenta lo yaw o, in maniera informale, la rotazione vista dall’alto del robot, genera con assoluta imprevedibilità, valori inesatti.

Dunque su un dataset contenente i valori dello yaw espressi in gradi sono presenti impurità, o valori inesatti. Questi valori acquisiscono uno scostamento che, rispetto ai valori corretti, rende il software per la rotazione del robot completamente inadoperabile.

• Soluzioni

1. **Sostituzione del sensore**

   Si è pensato alla sostituzione del sensore per ovviare a questa problematica, ma il problema ha continuato a persistere.

2. **Librerie e schede di sviluppo**

   Sono state adottare differenti librerie (proprietarie e open-source) e schede di sviluppo (Arduino e NodeMCU) per testare il modulo stesso, ma il problema ha continuato a persistere.

3. **Filtro Z-Score**

   Sono state applicate svariate tecniche per il filtraggio dei valori inesatti, tra cui lo Z-Score tramite la libreria numpy di Python.

   Fondamentalmente, uno z-score è il numero di deviazioni standard rispetto alla media di un punto informativo. Qualsiasi valore al di sopra o al di sotto dello z-score viene considerato inesatto.

   Ma ciò non è bastato alla rimozione del valore inesatto dal dataset.

4. **Adozione del remote controller**

   Infine, per ovviare alla problematica sopra citata, è stato realizzato un modulo per la gestione della rotazione del robot, come meglio descritto nel punto 4 di questo documento.

2. Robot

• Attuatori

Il robot (2.10) dispone di tre tipologie di attuatori:

- **Motori**

  Il robot è dotato di quattro motori DC da 3V-6V connessi a degli appositi gear per modulare la forza e la velocità di rotazione. (2.1)

- **Buzzer**

  Buzzer o cicalino passivo controllato in governato Pulse Width Modulation (PWM) capace di emettere diverse frequenze di suoni, incluse le note musicali. (2.2)

- **Led**

  Strip Led Adafruit composta da 8 elementi, collocata sul pianale/governatore dei motori dc e servo. (2.3)

• Sensori

- **Ultrasuoni**

  Sensore ad ultrasuoni, modulo HC-SR04, dotato di trigger o emittente e di echo o ricevitore. Facendo partire l'impulso ad ultrasuoni dal trigger si attende il suo ritorno, viene misurato il tempo per compiere questa tratta in modo da poter calcolare con precisione la distanza dell'oggetto posto davanti al modulo. (2.4)

- **Infrarossi**

  Sensore di tracciamento di linea. Un modulo di produzione proprietaria FreeNove, il quale incorpora tre elementi trasmettitore-ricevitore a infrarossi capaci di rilevare una linea di colore scuro tendente al nero posta sul pavimento. (2.5)

- **Fotoresistori**

  Fotoresistori collocati sulla parte anteriore del prototipo controllati in maniera analogica. I valori restituiti sono trasformati in digitali tramite l'ADC. (2.6)

• Altro

Sul robot è stato posizionato un push-button per permettere al robot di comunicare il suo corretto posizionamento all'entrata del labirinto. (2.7)

- ## Scheda di Sviluppo

  Il robot è pensato per funzionare con una scheda RaspberryPi Model B+, dunque è stato adottato un SoC RPi4 dotato di 4GB di RAM e CPU 4-Core 1.5 GHz. (2.8)

- ## Board

  Componente fondamentale del robot sulla quale sono montati tutti i componenti, quali sensori e attuatori, nonché anche il SoC RPi4.

  Tale board ha i seguenti utilizzi:
  - Fornire un’interfaccia di comunicazione verso tutti i componenti tramite bus seriale I2C e pin GPIO.
  - Alimentazione di tutti componenti, inclusa la scheda RaspberryPi.
  - Governatore per i motori dc e servo.

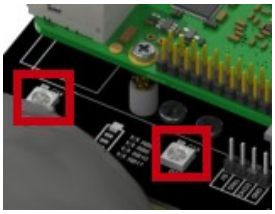
  Inoltre, è dotata di modulo PCA9685 16-Channel 12-Bit driver per la gestione dei motori dc e servo, modulo PCF8591, una chip per acquisizione dati CMOS a 8 bit, dotato di quattro ingressi analogici, un'uscita analogica e un'interfaccia seriale I2C.
  Su tale board sono presenti due switch che permettono rispettivamente l’accensione/spegnimento del RaspberryPi e dei componenti. (2.9)

- ## Immagini

[2.1] Motore

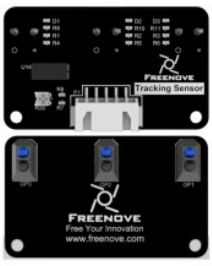
[2.2] Buzzer

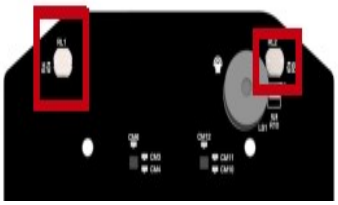
[2.3] Led



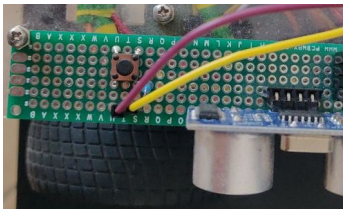
[2.4] Ultrasuoni

[2.5] Infrarossi



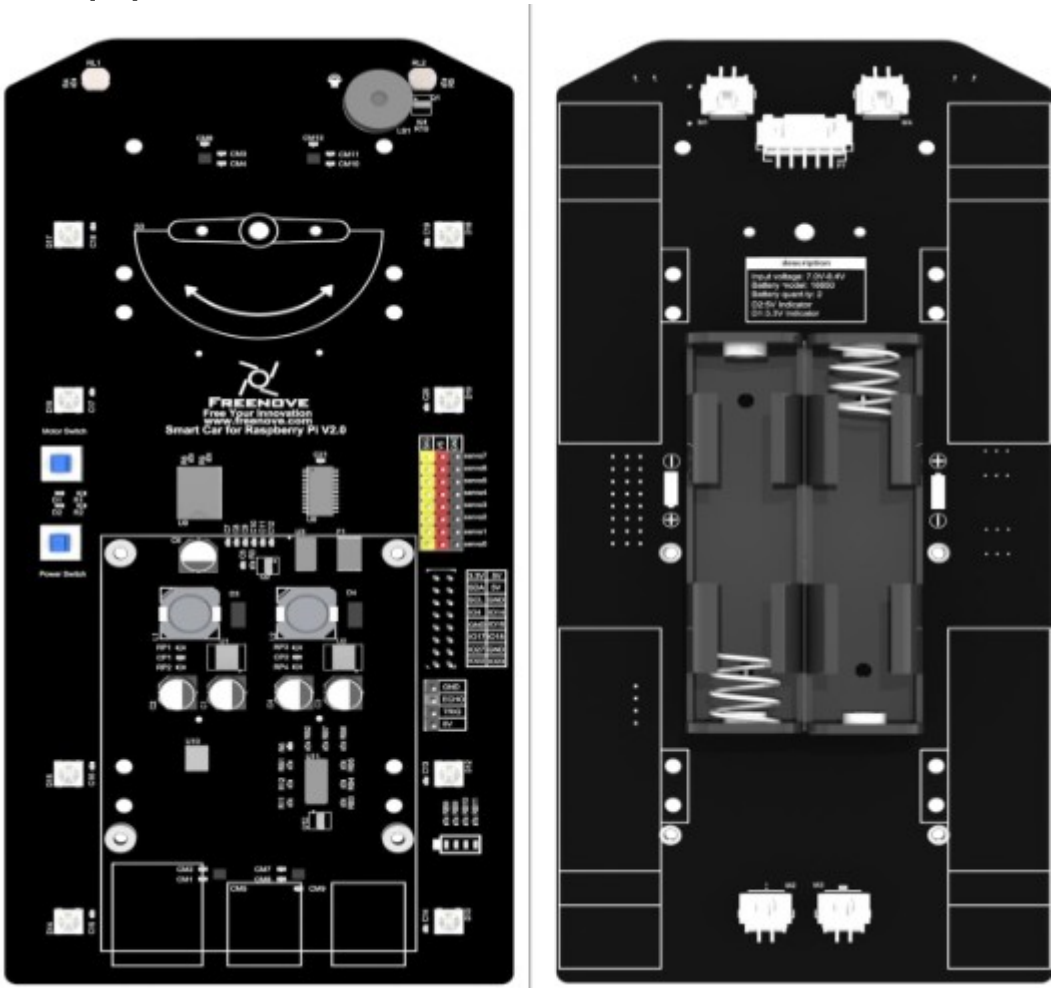


[2.7] push-button

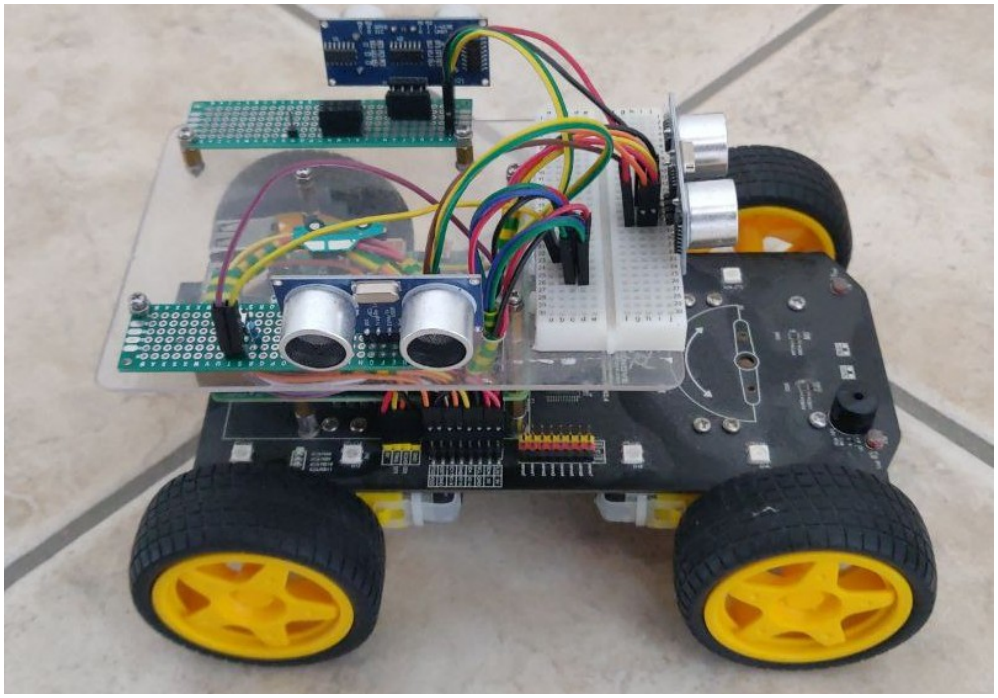


[2.8] Scheda di sviluppo

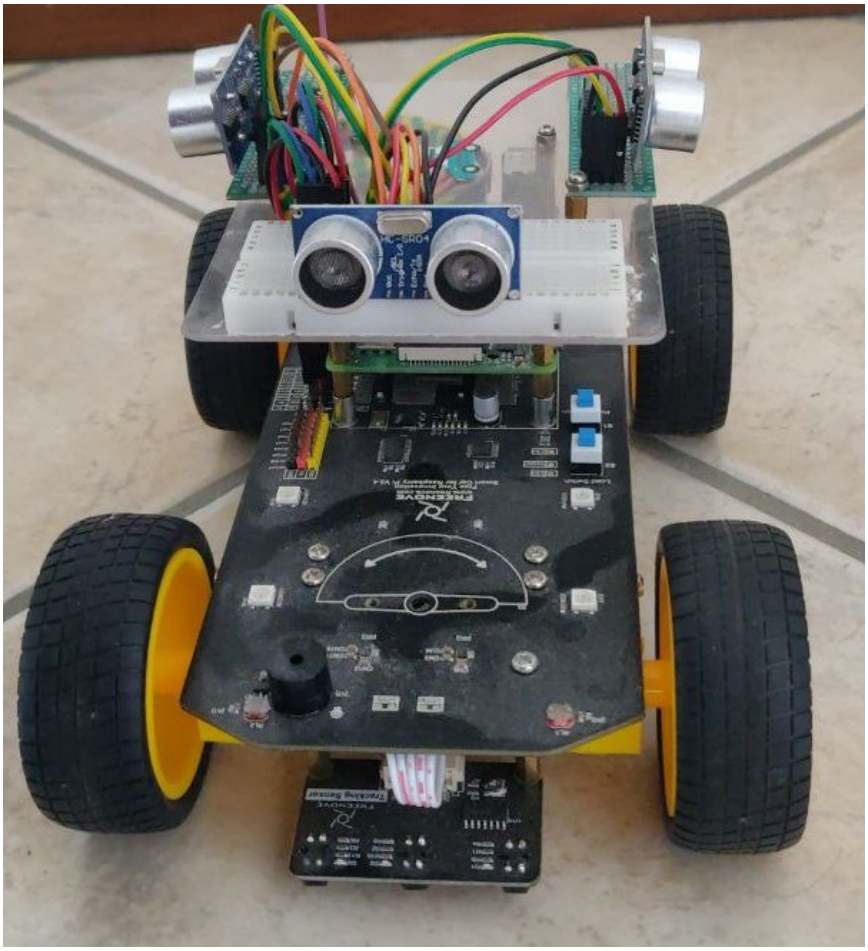
[2.9] Board



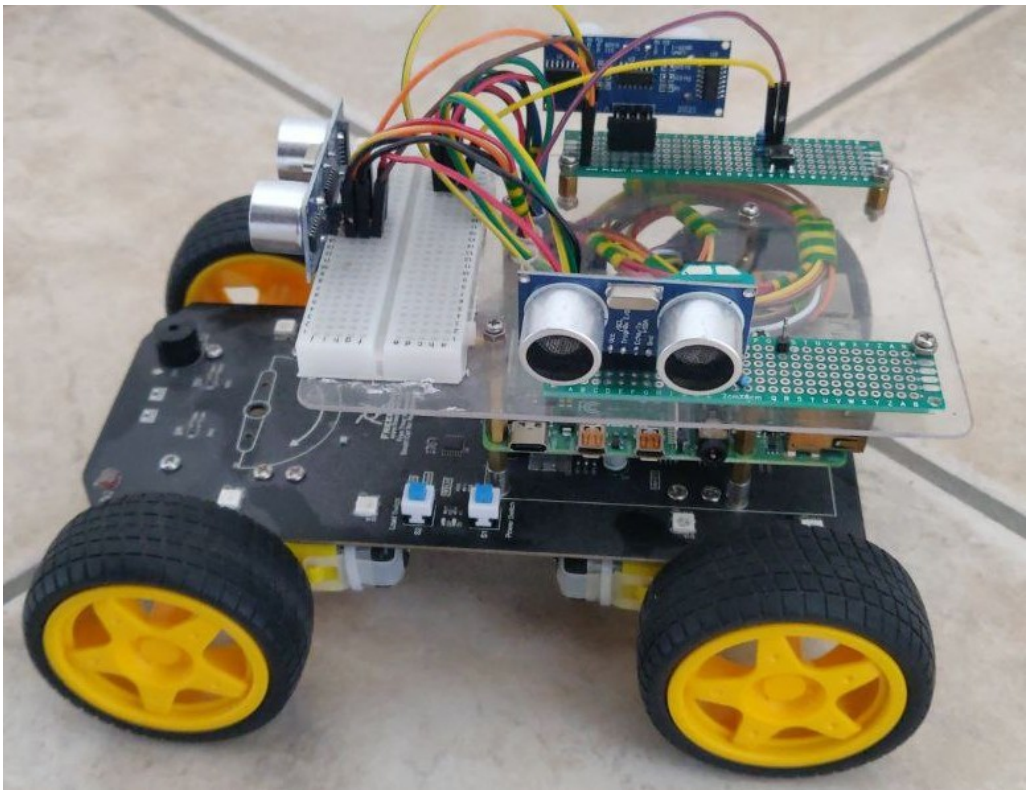
[2.10] Robot



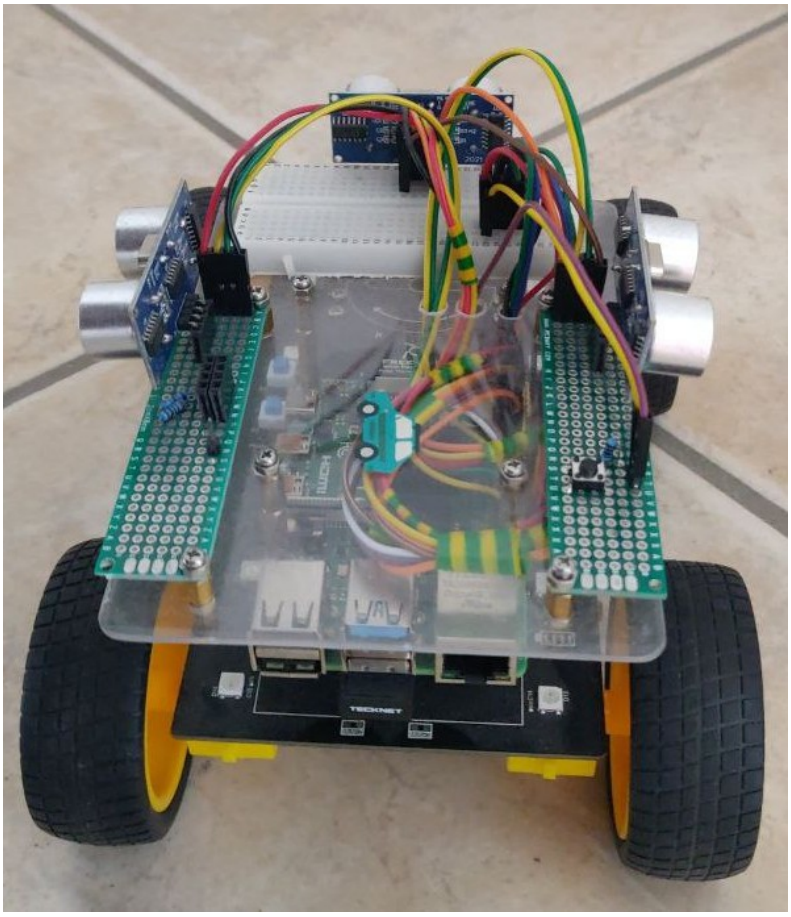
vista laterale destra



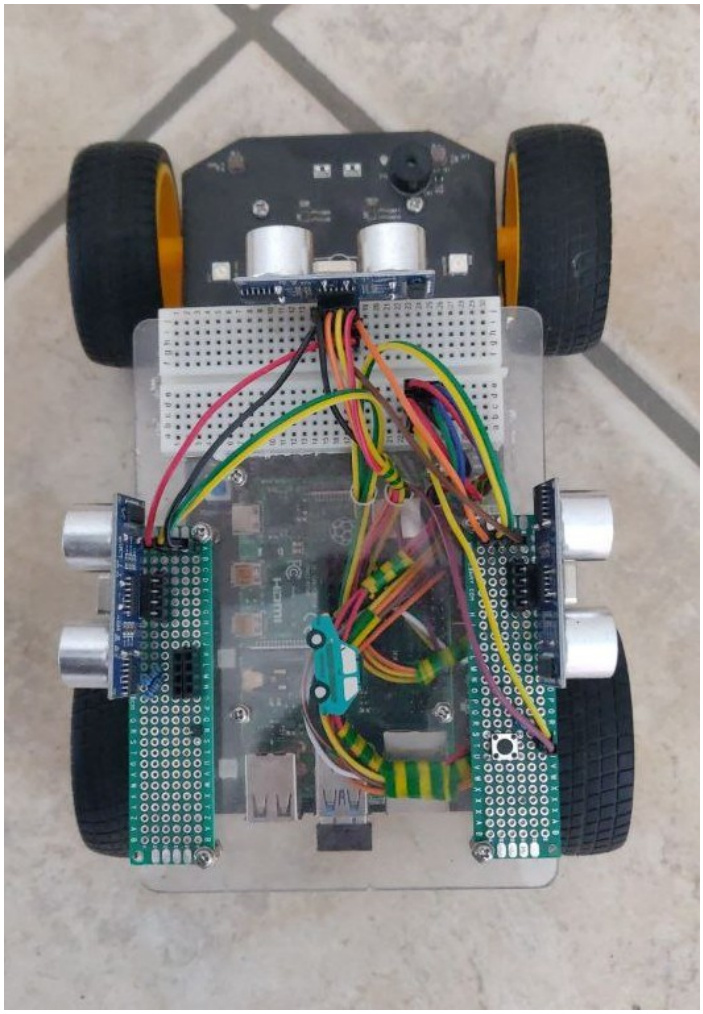
vista frontale



vista laterale sinistra



vista posteriore



vista dall’alto

3. **Robot API**

Per l’interfacciamento fra i comandi ad alto livello ed i componenti del robot (attuatori e sensori) sono state sviluppate una serie di API per ogni tipologia di componente. (3.1)

Per alcuni di essi è bastato gestire i pin fisici della scheda di sviluppo (RaspberryPi 4B+), mentre per altri si è ricorso all’utilizzo di librerie esterne che gestiscono i bus dati I2C. (3.2)

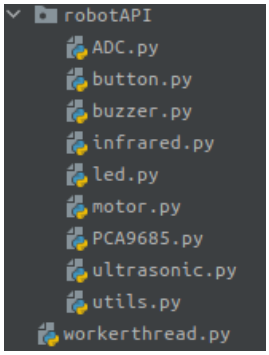
Le API sono state implementate completamente in Python3 includendo funzioni ad alto livello, multi-threading e callbacks.

Per alcuni dei sensori, come ad esempio sensori ad ultrasuoni e infrarossi si è ritenuto opportuno adottare la programmazione multi-threading in modo da limitare il possibile delay fra l’invocazione della funzione ad alto livello per la ricezione dei dati.

Così facendo, si è rimandato tale task al thread opportuno, in modo da garantire la continua presenza di dati, i quali vanno semplicemente recuperati, eliminando di fatto la cattura fisica degli stessi. (3.3)

- **Immagini**

[3.1] API



[3.2] Librerie

```
from rpi_ws281x import *
from lib.robotAPI.utils import ROBOTAPIConstants as RC


class Led:
    def __init__(self):
        # Control the sending order of color data
        self.ORDER = "RGB"
        # Create NeoPixel object with appropriate configuration.
        self.strip = Adafruit_NeoPixel(
            RC.LED_COUNT, RC.LED_PIN, RC.LED_FREQ_HZ, RC.LED_DMA, RC.LED_INVERT, RC.LED_BRIGHTNESS, RC.LED_CHANNEL)
        # Intialize the library (must be called once before other functions).
        self.strip.begin()

    def LED_TYPR(self, order, R_G_B):
        B = R_G_B & 255
        G = R_G_B >> 8 & 255
        R = R_G_B >> 16 & 255
        Led_type = ["GRB", "GBR", "RGB", "RBG", "BRG", "BGR"]
        color = [Color(G, R, B), Color(G, B, R), Color(R, G, B),
                 Color(R, B, G), Color(B, R, G), Color(B, G, R)]
        if order in Led_type:
            return color[Led_type.index(order)]
```

[3.3] Multi-threading

```python
class Infrared:
    def __init__(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(RC.IR_LEFT, GPIO.IN)
        GPIO.setup(RC.IR_MID, GPIO.IN)
        GPIO.setup(RC.IR_RIGHT, GPIO.IN)

        self.__left_status: bool = False
        self.__mid_status: bool = False
        self.__right_status: bool = False

        self.__discover = RobotThread(target=self.__detect, name='ir_discover')

    def virtual_destructor(self) -> str:
        return self.__discover.bury()

    def begin(self):
        if not self.__discover.is_alive():
            self.__discover.start()

    def __detect(self):
        self.__left_status = False
        self.__mid_status = False
        self.__right_status = False

        while True:
            self.__left_status = True if self.__left_status else self.__left
            self.__mid_status = True if self.__mid_status else self.__mid
            self.__right_status = True if self.__right_status else self.__right

            sleep(0.05)

    @property
    def __left(self) -> bool:
        return bool(GPIO.input(RC.IR_LEFT))

    @property
    def __mid(self) -> bool:
        return bool(GPIO.input(RC.IR_MID))

    @property
    def __right(self) -> bool:
        return bool(GPIO.input(RC.IR_RIGHT))

    @property
    def status(self) -> tuple:
        return int(self.__left_status), int(self.__mid_status), int(self.__right_statu
```

4. **Remote Controller**

Per ovviare alle problematiche relative al modulo giroscopio MPU6050, come accennato nella sezione introduttiva, si è pensato alla realizzazione di un modulo DIY per controllare la rotazione del robot. (4.1)

• Componenti

Composto da cinque push-button ed un potenziometro logaritmico da un 1kΩ.

Quattro dei push-button sono utilizzati per il movimento del robot (su, giù, destra, sinistra).
Il quinto è utilizzato per confermare l'avvenuta rotazione, mentre, il potenziometro regola la velocità della rotazione stessa. (4.2)

• Scheda di Sviluppo

Il remote controller è realizzato tramite una scheda di sviluppo **ESP-12 AMICA 1.0** (4.3). Tale scheda è stata programmata nell'ambiente di sviluppo 'ArduinoIDE' in C++ usufruendo delle librerie rilasciate dalla casa produttrice Espressif. (4.4)

Il remote-controller risultante è in grado di comunicare gli stati dei push-button e del potenziometro tramite seriale USB ad una interfaccia wrapper in Python deputata all'invio degli appositi messaggi in formato JSON verso il PhysicalBody.

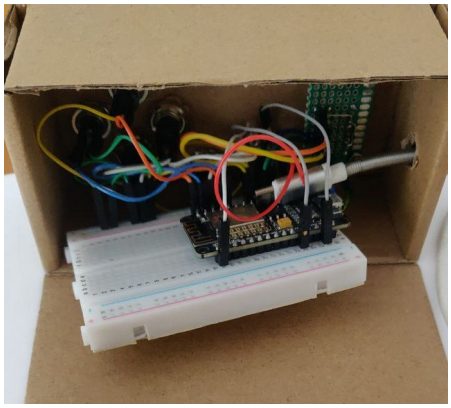
• Access Point

In aggiunta, il remote controller è capace di generare un segnale WiFi in modalità Access Point (AP) con basso carico di lavoro in modo da velocizzare le comunicazioni fra i vari componenti.

A tale segnale vi sono collegati il Controller ed il PhysicalBody.

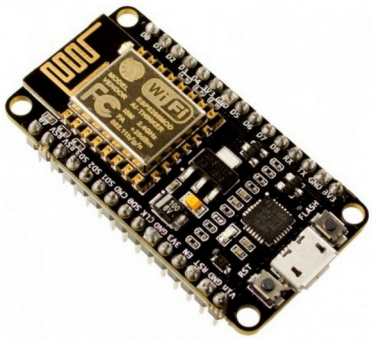
- **Immagini**

[4.1] Remote controller



[4.2] Componenti

[4.3] ESP-12 AMICA



[4.4] Source Code Snippet

```
void setup()
{
  Serial.begin(9600);
  //Serial.println();

  pinMode(Key.UPPER, INPUT_PULLUP);
  pinMode(Key.LOWER, INPUT_PULLUP);
  pinMode(Key.RIGHT, INPUT_PULLUP);
  pinMode(Key.LEFT,  INPUT_PULLUP);
  pinMode(Key.MODE,  INPUT_PULLUP);

  pinMode(LedGroup.LED_SELF_IP,   OUTPUT);
  pinMode(LedGroup.LED_FIRST_IP,  OUTPUT);
  pinMode(LedGroup.LED_SECOND_IP, OUTPUT);
  pinMode(LedGroup.LED_THIRD_IP,  OUTPUT);

  delay(1000);

  WiFi.mode(WIFI_AP);
  WiFi.softAP(ssid, pass);
  //Serial.println("AP started");
  //Serial.println("IP address: " + WiFi.softAPIP().toString());

  delay(1000);

  //Serial.println("Starting MQTT broker");
  myBroker.init();

  init_Keys();
  init_LedGroup();

  toogle_led(WiFi.softAPIP().toString());
}


void loop()
{
  Key.keys[0].pressed = digitalRead(Key.UPPER) ? 0:1;
  Key.keys[1].pressed = digitalRead(Key.LEFT)  ? 0:1;
  Key.keys[2].pressed = digitalRead(Key.LOWER) ? 0:1;
  Key.keys[3].pressed = digitalRead(Key.RIGHT) ? 0:1;
  Key.keys[4].pressed = digitalRead(Key.MODE)  ? 0:1;

  Key.Potentiometer.pot_val = (((analogRead(Key.POT) * 3495) / 1024) / 50) * 50;

  detectCommand();
}
```

```
void detectCommand()
{
  uint8_t* currState;
  uint8_t* oldState;
  uint8_t* cmd;

  for (uint8_t i = 0; i < 5; i++)
  {
    currState = &Key.keys[i].pressed;
    oldState  = &Key.keys[i].old;
    cmd       = &Key.keys[i].cmd;

    if (*currState == 1 && *currState != *oldState)
    {
      switch (*cmd)
      {
        case 'w':
          myBroker.publish("CMD", "FORWARD");
          Serial.println("FORWARD");
          break;
        case 's':
          myBroker.publish("CMD", "BACKWARD");
          Serial.println("BACKWARD");
          break;
        case 'a':
          myBroker.publish("CMD", "LEFT");
          Serial.println("LEFT");
          break;
        case 'd':
          myBroker.publish("CMD", "RIGHT");
          Serial.println("RIGHT");
          break;
        case 'm':
          myBroker.publish("CMD", "DONE");
          Serial.println("DONE");
          break;
      }

      Key.keys[i].old = *currState;
    }
    else if (*currState == 0 && *currState != *oldState)
    {
      myBroker.publish("CMD", "STOP");
      Serial.println("STOP");

      Key.keys[i].old = *currState;
    }
  }

  if (Key.Potentiometer.pot_val != Key.Potentiometer.old_pot_val)
  {
    myBroker.publish("CMD", (String)(Key.Potentiometer.pot_val + 600));
    Key.Potentiometer.old_pot_val = Key.Potentiometer.pot_val;

    Serial.println((String)(Key.Potentiometer.pot_val + 600));
  }

  delay(20);
```

5. Comunicazione

• Message broker

Per permettere la comunicazione fra i vari componenti si è adoperato un message broker come Redis con tecnologia Pub/Sub. Redis si occupa della condivisione di strutture dati di tipo associativo (Key -> Value).

I dati inviati su questo message broker vengono associati a chiavi univoche e pubblicate sugli opportuni Topic. (5.1 e 5.2)

• Limitazioni

Libredis, usato con la metodologia canonica, soffre di limitazioni, in quanto, nella fase di ascolto di nuovi messaggi su un determinato Topic, bisogna effettuare l'operazione all'interno di un ciclo infinito. Dunque, il normale flusso del programma verrebbe completamente bloccato.

• Soluzione

Per ovviare a questa problematica si è ricorso alla programmazione multi-threading, dove il sopracitato ciclo deputato all'ascolto di nuovi messaggi viene eseguito in un thread distaccato dal principale. (5.3)

• Struttura Dati Condivisa

Per quanto riguarda la persistenza dei dati all'interno dell'esecuzione del software, è stata implementata una struttura dati statica suddivisa in opportune sezioni per ogni tipologia di messaggio. Essa viene replicata in ogni istanza eseguita, in maniera condivisa.

Riceve come input i valori inviati al message broker e se essi differiscono dai dati precedentemente immagazzinati, viene impostato un flag booleano di modifica avvenuta (True), permettendo l'invio dei nuovi dati. Una volta effettuata tale operazione, il flag viene impostato su in valore booleano negativo (False).

Questa metodologia viene applicata sia per la fase di invio dei dati, sia per la fase di ricezione, allo scopo di limitare i dati che navigano all'interno del database di Redis qualora siano esattamente identici ai precedenti.

Un esempio potrebbe essere l'invio dei dati dei sensori di prossimità (ultrasuoni) quando il robot è fermo. I dati saranno sempre gli stessi, dunque è inutile re-inviarli (5.4)

- **Immagini**

[5.1] Chiavi univoche e [5.2] Topics

```
class __RedisData:
    class Connection:
        Host = __REDIS_CFG__['HOST']
        Port = __REDIS_CFG__["PORT"]

    class Topic:
        Body = __REDIS_CFG__["BODY_TOPIC"]
        Remote = __REDIS_CFG__["REMOTE_CONTROLLER_TOPIC"]
        Controller = __REDIS_CFG__["CTRL_TOPIC"]

    class Key:
        Btn = __REDIS_CFG__["BTN_KEY"]
        SELF = __REDIS_CFG__["SELF_KEY"]
        RC = __REDIS_CFG__["RC_KEY"]
        MPU = __REDIS_CFG__["MPU_KEY"]
        Led = __REDIS_CFG__["LED_KEY"]
        Motor = __REDIS_CFG__["MOTORS_KEY"]
        Buzzer = __REDIS_CFG__["BUZZER_KEY"]
        Infrared = __REDIS_CFG__["INFRARED_KEY"]
        Ultrasonic = __REDIS_CFG__["ULTRASONIC_KEY"]
```

[5.3] Multi-threading

```
self.__redis_message_handler = None
self.__redis = Redis(host=ControllerData.Connection.Host, port=ControllerData.Connection.Port,
                     decode_responses=True)
self.__redis.flushall()
self.__pubsub = self.__redis.pubsub()
self.__pubsub.psubscribe(**{ControllerData.Topic.Body: self.__on_message})
self.__pubsub.psubscribe(**{ControllerData.Topic.Remote: self.__on_remote})
```

```
def __on_remote(self, msg):
    if RemoteControllerData.is_enabled:
        _key = msg['data']
        _value = self.__redis.get(_key)

        if _key == RemoteControllerData.Key.RC:
            data = json.loads(_value)
            RemoteControllerData.on_values(data['rc_cmd'], data['rc_spd'])

        if RemoteControllerData.is_engaged and RemoteControllerData.is_done:
            self.__new_buzzer(True, True)
            time.sleep(0.3)
            self.__new_buzzer(False, True)

            self.__new_led(False, True, None, True)

            self.__remote.dismiss()
            RemoteControllerData.engaged(False)
            ControllerData.Machine.set_ready(1)

# DONE
# callback receiver
def __on_message(self, msg):
    _key = msg['data']
    _message = self.__redis.get(_key)

    ControllerData.Machine.on_values(_message)
```

[5.4] Struttura dati condivisa

```python
class ControllerData(__RedisData):
    class __Value:...

    class Machine:...

    class Motor(__Value):
        __rum = None
        __lum = None
        __rlm = None
        __llm = None

        @classmethod
        def on_values(cls, rum: int, lum: int, rlm: int, llm: int):
            if cls.__rum == rum and cls.__lum == lum and cls.__rlm == rlm and cls.__llm == llm:
                super().change(False)
            else:
                cls.__rum = rum
                cls.__lum = lum
                cls.__rlm = rlm
                cls.__llm = llm

                super().change(True)

        @classmethod
        @property
        def values(cls):
            data = dict()
            data['rum'] = cls.__rum
            data['lum'] = cls.__lum
            data['rlm'] = cls.__rlm
            data['llm'] = cls.__llm

            super().change(False)

            return json.dumps(data, indent=0)

    class Led(__Value):
        __arrow = 0
        __frlb = None

        @classmethod
        def on_arrow(cls, arrow: bool, frlb: FRLB):
            cls.__arrow = int(arrow)
            cls.__frlb = frlb.value if frlb is not None else None

        @classmethod
        def leds(cls):
            data: dict = dict()
            data['status'] = cls.status()
            data['arrow']  = cls.__arrow
            data['frlb']   = cls.__frlb

            cls.__arrow = 0
            cls.__frlb = None

            return json.dumps(data, indent=0)
```

6. **Labirinto**

- **Realizzazione**

  Per la realizzazione del labirinto fisico (6.1) si è adoperato il cartone come materiale da costruzione, modellandolo al punto da generare un percorso privo di cicli che raggruppa i principali casi in cui il robot può incappare.

  Dunque, sono stati realizzati vicoli ciechi di complessità differente, snodi o giunzioni, entrata ed uscita.

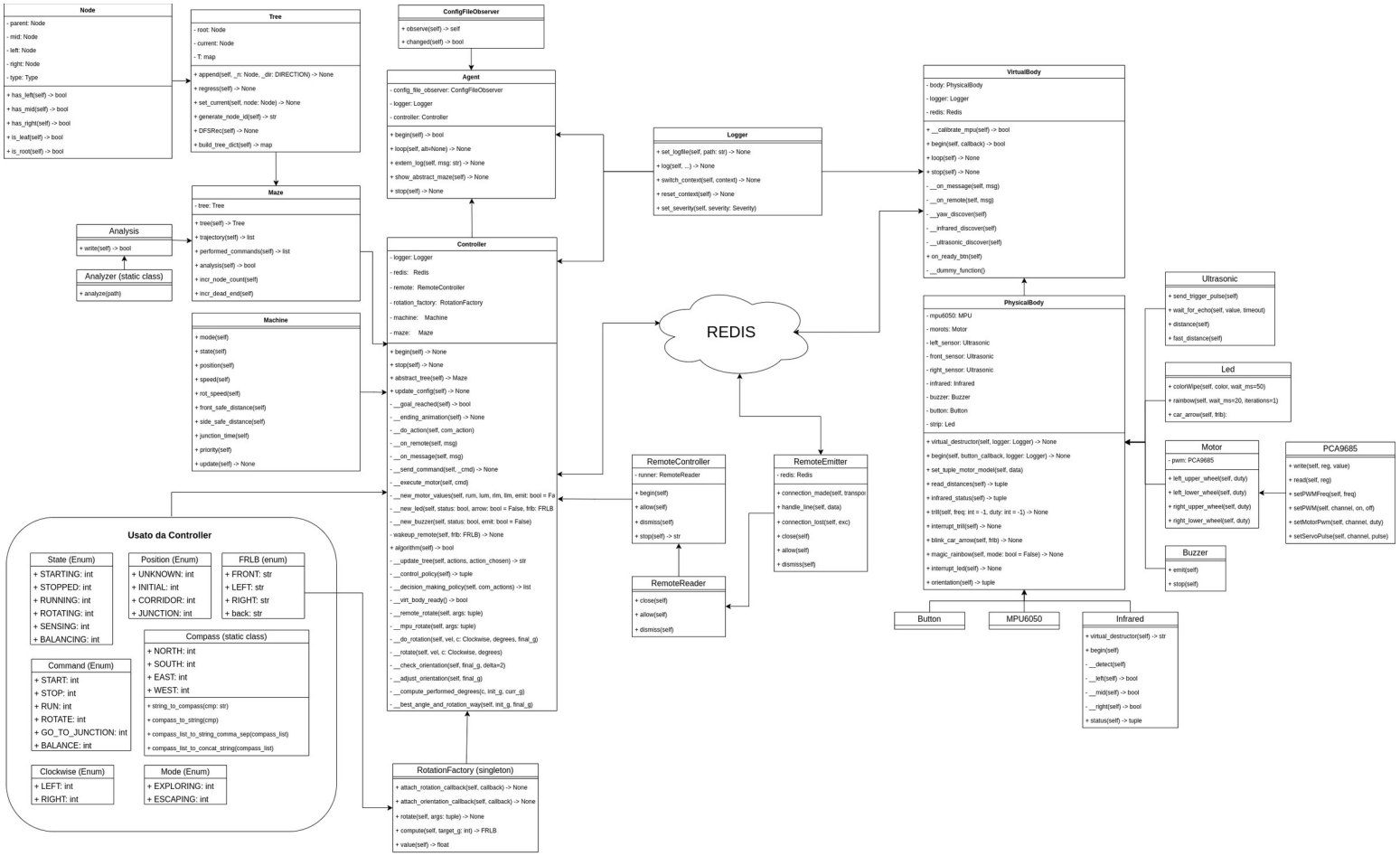
  Si è pensato anche alla potenziale modularità del labirinto stesso:
  alcuni dei vicoli ciechi possono trasformarsi in entrate/uscite, in modo da far compiere al robot, nello stesso labirinto, percorsi diversi.

- **Immagine**

  [6.1] Labirinto

7. **Class Diagrams**

- **Core Class Diagram**



- **Class Diagram Struttura Dati Condivisa**