# POLITECNICO DI TORINO

**Master Degree**
**in Mechatronic Engineering**

Master Degree Thesis

# Motion Control Architecture for Service Robotics


**Supervisor**
prof. Marcello Chiaberge

**Candidate**
Nunzio Villa

Academic Year 2021-2022

# Summary

In recent years, there has been an increase in the use of robots in everyday human activities, both domestic and professional, which go beyond the industrial sector. The IFR (International Federation of Robotics) defines service robotics as "a robot that operates autonomously or semi-autonomously to perform services useful for the well-being of human beings, excluding the manufacturing sector". The main applications in the home are the cleaning of domestic environments, care of the elderly, and entertainment of children. In the professional field, on the other hand, the applications are innumerable both in natural environments (land, sea, and space) and in artificial environments (offices, hospitals, and urban infrastructures). Examples of these applications are rescuing in hostile areas, monitoring and data collection actions, inspection, and maintenance, in the logistics field for storage and movement of material, and the medical one for surgery and rehabilitation. The use of robots in these areas intensified in 2020 during the COVID-19 pandemic and is expected to further increase in the coming years as evidenced by the 2020 IFR report [1]. This thesis aims to develop both a software and hardware architecture that can be used in most of the aforementioned cases. Therefore, the common needs of these robots will be analyzed first and then they will be implemented in the final design. The final architecture resulting from this analysis will have to respond to a very important requirement such as the versatility that will be the cornerstone of the project.

# Contents

# Chapter 1

# Introduction

## 1.1 Hardware requirements

Before defining what the requirements in the software part may be, it is necessary to talk about the hardware requirements and in particular the main components to be installed to obtain an architecture that can be used as a starting point, ready to use, for the main applications of service robotics. First of all, it is necessary to think about the power supply. The technical characteristics of the battery must be analyzed based on factors such as component consumption and desired autonomy. Another fundamental thing is the electric motors that will have the task of making the robot move. To understand what are the essential characteristics that a robot must have, first of all, we need to analyze the environment in which it will operate. For example in an indoor environment, this information can be translated into a speed limit to avoid creating damages to people or objects with which the robot may collide. To limit the speed it is necessary to know it and for this it may be useful to install encoders on the robot wheels. As regards collisions, it is necessary to avoid them, and for this reason, it is necessary a sensor that can sense the presence of an obstacle. Given the simplicity of implementation and the low cost, sonar is commonly used. The last component that completes the list of things a robot needs to move autonomously is the microcontroller that will have the task of processing sensors data and piloting DC motors. In the following sections, more details about these components will be given.

## 1.2 Software requirements

To understand what are the requirements in terms of software design it is necessary to understand what robots have to do and who the final user is. The tasks that the robot must accomplish can be summarized in moving around the environment,

acquiring data from the sensors, and communicating with other devices using serial protocols.

From this list, we define the user requirements which describe what the user expects to receive and represent a high-level view of the software. These requirements can be split more thoroughly describing the functional requirements that specify how the software works by defining the interaction with the user or the other devices. In our case we can summarize this requirement as follow:

- **Receive vector data from a PC through a USB port.**This vector contains the linear velocities of the robot along the main axis X and Y and the angular speed around the Z-axis. Moreover, it should be able to turn on/off a pair of led lights and follow a predefined path when receiving instructions.

- **Send vector data containing the sensor measurements.**The microcontroller is responsible to acquire the data provided by the sensors mounted on the robot, elaborate, and compact them to send to the PC.

- **Control the DC motors.** The microcontroller must use the input provided by the PC and the sensors' data to control the DC motors.

As regards the low-level requirements is requested to develop a C++ library, that can be deployed on a microcontroller like Arduino, able to perform the above tasks. In figure 1.1, a sketch of the software architecture is shown.

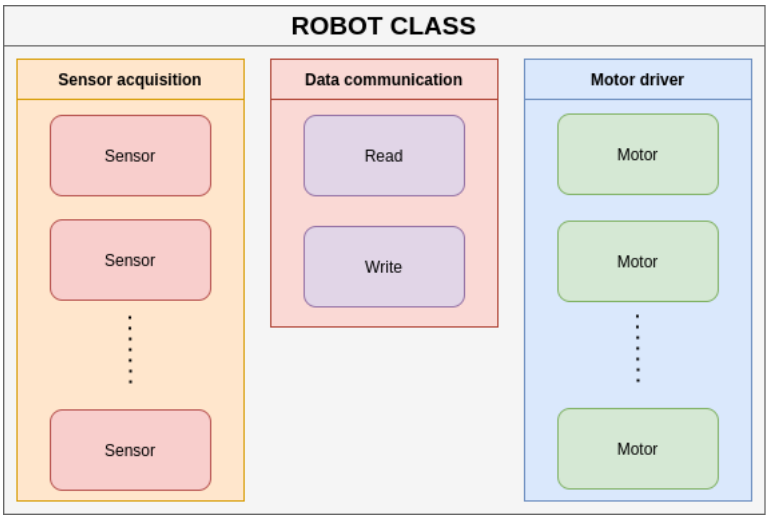



Figure 1.1. Software Architecture

## 1.3 Robot architecture

Among the commercial products, it was necessary to choose one that meets the required characteristics. The choice has fallen into a kit that contains most of the components seen in the previous section (Figure 1.2). The architecture taken as a reference consists of 4 DC motors each having an encoder for measuring the position, and therefore the speed of the motors, and an embedded gearbox that increases the torque transmitted to the wheels. The motors drive four mecanum Omni-directional wheels [3] that allow the robot to move in any direction (Figure 1.3) by applying different speeds at each motor. Moreover, it is equipped with four ultrasonic sensors (front, rear, left, and right) and an Arduino Duemilanove that can be programmed using the Arduino IDE.

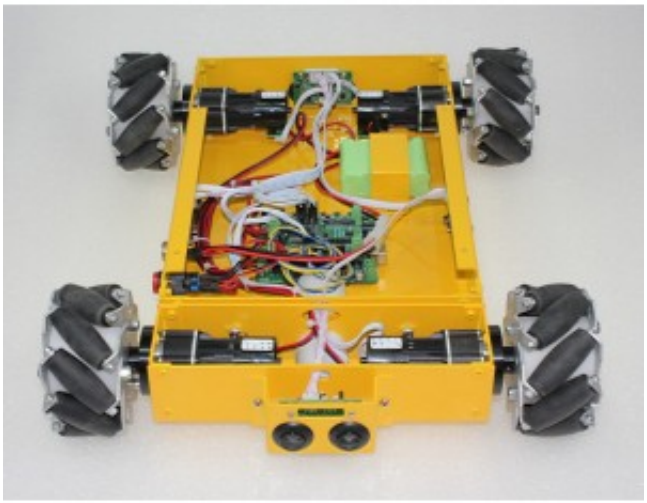


Figure 1.2. Nexus4WD robot

## 1.4 Robot kinematic

To be able to develop the software part responsible for the robot movements, it is necessary to describe the kinematic model of the robot. In the last section, the Mecanum Omni-wheels have been included in the list of components. They are special wheels designed for differential robots and allow them to move in any direction.

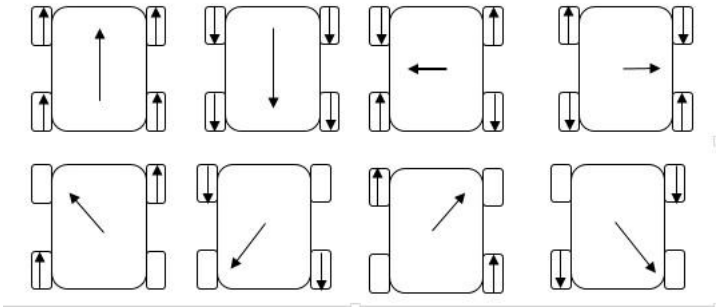


Figure 1.3. Robot motion according to wheels' speeds

According to [5] and taking into account the reference frames of the robot (Figure 1.4) and of the wheel (Figure 1.5), the equation that links the wheel's angular speed and the robot velocities is given by:

$$\boldsymbol{\omega} = T\boldsymbol{v} \tag{1.1}$$

where T is called transformation matrix and is given by:

$$T = \frac{1}{r}\begin{bmatrix} 1 & -1 & -(l_x + l_y) \\ 1 & 1 & (l_x + l_y) \\ 1 & 1 & -(l_x + l_y) \\ 1 & -1 & (l_x + l_y) \end{bmatrix} \tag{1.2}$$

where $r$ is radius if the wheel and $l_x$ and $l_y$ are half of the distance between front wheels and half of the distance between the front wheel and the rear wheels respectively. The inverse relation between the angular speed of the wheels and the robot's velocity can be computed with the equation 1.3 where $T'$ is the inverse matrix of $T$.

$$\boldsymbol{v} = T'\boldsymbol{\omega} \tag{1.3}$$

with

$$T' = \frac{r}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -\frac{1}{l_x+l_y} & \frac{1}{l_x+l_y} & -\frac{1}{l_x+l_y} & \frac{1}{l_x+l_y} \end{bmatrix} \tag{1.4}$$

Expanding 1.1, the following relation is obtained:

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{r}\begin{bmatrix} 1 & -1 & -(l_x + l_y) \\ 1 & 1 & (l_x + l_y) \\ 1 & 1 & -(l_x + l_y) \\ 1 & -1 & (l_x + l_y) \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ \omega_z \end{bmatrix} \tag{1.5}$$

From which four equations can be derived:

$$\begin{cases} \omega_1 & = \frac{1}{r}(v_x - v_y - (l_x + l_y)\omega_z \\ \omega_2 & = \frac{1}{r}(v_x + v_y + (l_x + l_y)\omega_z \\ \omega_3 & = \frac{1}{r}(v_x + v_y - (l_x + l_y)\omega_z \\ \omega_4 & = \frac{1}{r}(v_x - v_y + (l_x + l_y)\omega_z \end{cases} \tag{1.6}$$

These equations will be used to transform the velocities of the robot into motor speeds.

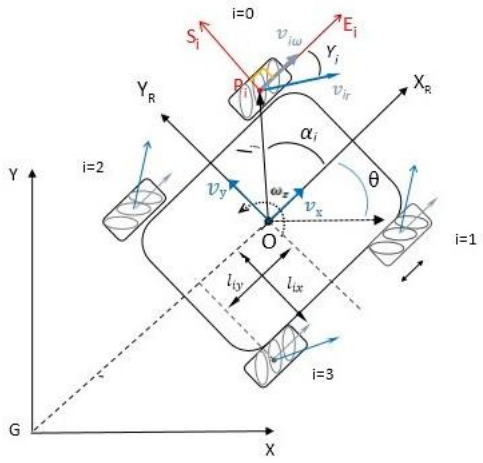



Figure 1.4. Robot coordinate systems

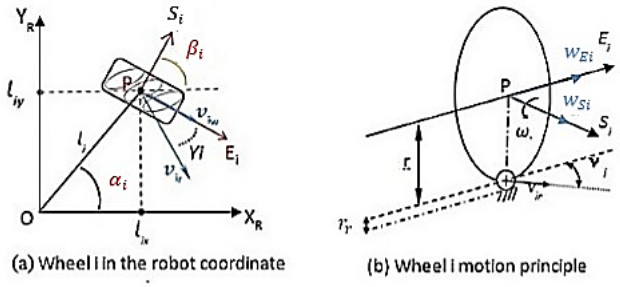



Figure 1.5. Wheels coordinate systems

# Chapter 2

# Hardware Architecture

## 2.1 Battery

To power supply the components it is necessary to install a battery. The nominal voltage is chosen considering the power supply voltage of the DC motors that in this case is 12 V (Fig 2.1). The other important parameters of a battery are the capacity and the maximum current that can be supplied. The capacity is responsible for the battery life and consequently of the robot's autonomy. The maximum deliverable current must be higher than the one requested by the motors. The last two characteristics are the size and the weight that must be minimized to increase the performance of the robot. To satisfy all these requirements, a Lithium Polymer battery has been chosen. This technology has a high density of charge which means a high capacity in small size and low weight. Since not only the 12 V power supply is necessary but also the 5 V to power up the electronic components, the use of a DC/DC converter must be considered to convert the 12 V provided by the battery to a power supply that can be handled by all the devices.

## 2.2 Motors driver

The electric DC motors are driven by a strategy called PWM (Pulse Width Modulation) in which the motors are powered with DC voltage that follows a square wave signal where the maximum value is the nominal value while the lower value is the ground. If the frequency $F_t$ of the square wave is high enough, the power supply that the motor receives can be approximated to the product of the duty cycle *d.c.* and the nominal value $V_N$ whereby duty cycle is the ratio between the wave's ON time and the period of the wave itself.

**Series 2342 ... CR**

| Values at 22°C and nominal voltage | | 2342 S | 006 CR | 012 CR | 018 CR | 024 CR | 036 CR | 048 CR | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Nominal voltage | $U_N$ | 6 | 12 | 18 | 24 | 36 | 48 | V |
| 2 | Terminal resistance | $R$ | 0,4 | 1,9 | 4,1 | 7,1 | 15,9 | 31,2 | Ω |
| 3 | Efficiency, max. | $\eta_{max.}$ | 81 | 80 | 81 | 81 | 81 | 81 | % |
| 4 | No-load speed | $n_0$ | 9 000 | 8 100 | 8 000 | 8 500 | 8 100 | 8 000 | min⁻¹ |
| 5 | No-load current, typ. (with shaft ø 3 mm) | $I_0$ | 0,17 | 0,075 | 0,048 | 0,038 | 0,024 | 0,017 | A |
| 6 | Stall torque | $M_H$ | 87,2 | 80 | 86,5 | 85,4 | 91,4 | 84,4 | mNm |
| 7 | Friction torque | $M_R$ | 0,98 | 1 | 0,99 | 0,99 | 0,99 | 0,95 | mNm |
| 8 | Speed constant | $k_n$ | 1 650 | 713 | 462 | 366 | 231 | 170 | min⁻¹/V |
| 9 | Back-EMF constant | $k_E$ | 0,604 | 1,4 | 2,16 | 2,73 | 4,34 | 5,87 | mV/min⁻¹ |
| 10 | Torque constant | $k_M$ | 5,77 | 13,4 | 20,7 | 26,1 | 41,4 | 56,1 | mNm/A |
| 11 | Current constant | $k_I$ | 0,173 | 0,075 | 0,048 | 0,038 | 0,024 | 0,018 | A/mNm |
| 12 | Slope of n-M curve | $\Delta n/\Delta M$ | 103 | 101 | 92,5 | 99,5 | 88,6 | 94,8 | min⁻¹/mNm |
| 13 | Rotor inductance | $L$ | 13,5 | 65 | 150 | 265 | 590 | 1 050 | µH |
| 14 | Mechanical time constant | $\tau_m$ | 6 | 6 | 6 | 6 | 6 | 6 | ms |
| 15 | Rotor inertia | $J$ | 5,6 | 5,7 | 6,2 | 5,8 | 6,5 | 6 | gcm² |
| 16 | Angular acceleration | $\alpha_{max.}$ | 160 | 140 | 140 | 150 | 140 | 140 | ·10³rad/s² |
| 17 | Thermal resistance | $R_{th1}$ / $R_{th2}$ | 3 / 15 | | | | | | K/W |
| 18 | Thermal time constant | $\tau_{w1}$ / $\tau_{w2}$ | 6,5 / 490 | | | | | | s |
| 19 | Operating temperature range: | | | | | | | | |
| | – motor | | -30 ... +100 | | | | | | °C |
| | – winding, max. permissible | | +125 | | | | | | °C |
| 20 | Shaft bearings | | ball bearings, preloaded | | | | | | |
| 21 | Shaft load max.: | | | | | | | | |
| | – with shaft diameter | | 3 | | | | | | mm |
| | – radial at 3 000 min⁻¹ (3 mm from bearing) | | 20 | | | | | | N |
| | – axial at 3 000 min⁻¹ | | 2 | | | | | | N |
| | – axial at standstill | | 20 | | | | | | N |
| 22 | Shaft play: | | | | | | | | |
| | – radial | ≤ | 0,015 | | | | | | mm |
| | – axial | = | 0 | | | | | | mm |
| 23 | Housing material | | steel, black coated | | | | | | |
| 24 | Mass | | 88 | | | | | | g |
| 25 | Direction of rotation | | clockwise, viewed from the front face | | | | | | |
| 26 | Speed up to | $n_{max.}$ | 11 000 | | | | | | min⁻¹ |
| 27 | Number of pole pairs | | 1 | | | | | | |
| 28 | Magnet material | | NdFeB | | | | | | |
| **Rated values for continuous operation** | | | | | | | | | |
| 29 | Rated torque | $M_N$ | 14 | 17 | 18 | 17 | 19 | 18 | mNm |
| 30 | Rated current (thermal limit) | $I_N$ | 2,9 | 1,5 | 1 | 0,78 | 0,53 | 0,38 | A |
| 31 | Rated speed | $n_N$ | 7 140 | 6 090 | 6 040 | 6 470 | 6 160 | 5 910 | min⁻¹ |

Figure 2.1. Fahaulaber 2342 12CR Datasheet

$$d.c. = \frac{T_{ON}}{T} \tag{2.1}$$

$$V_M = d.c. \cdot V_N \tag{2.2}$$

To obtain a variable voltage proportional to the duty cycle of a square wave a particular electronic circuit called H-bridge is used (Figure 2.3). Thanks to this circuit it is possible to drive the electric motor in both directions, forward and backward (Figure 2.4). This circuit consists of 4 MOSFETs that are piloted in pairs.

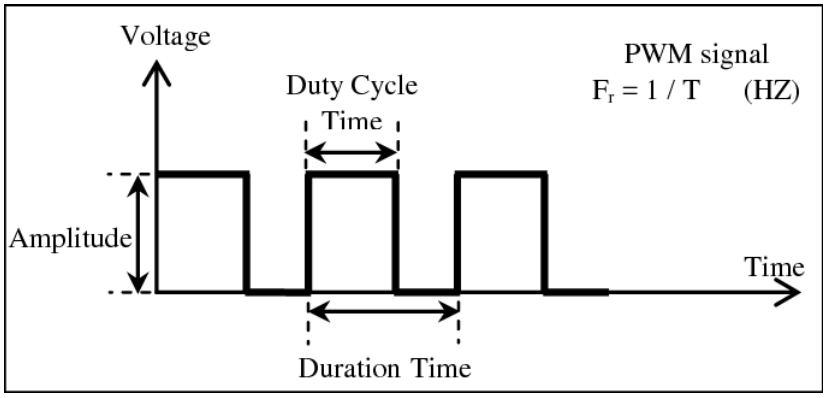



Figure 2.2. PWM signal

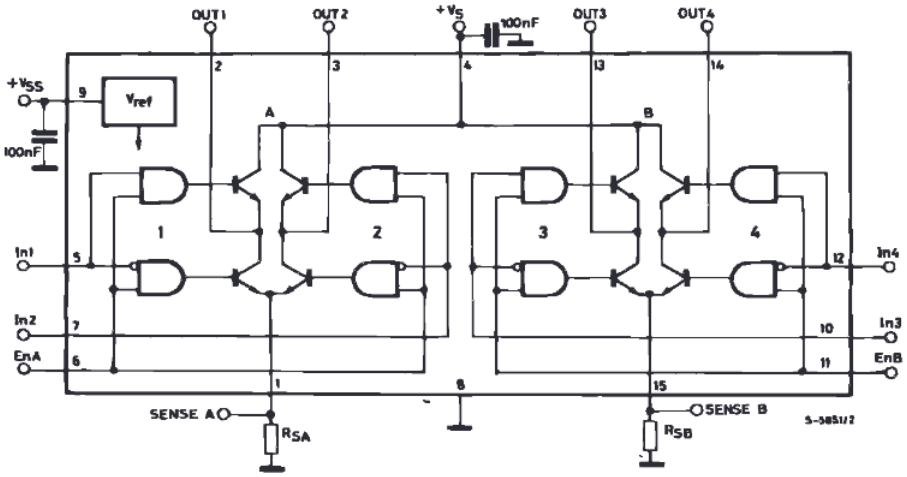



Figure 2.3. L298 H-bridge

## 2.3 Microcontroller

One of the most important components in a robot is the microcontroller. After some research, Teensy 4.1 (Figure 2.5) has been chosen since it offers better performance concerning the most common Arduino despite their price is comparable. This board has the advantage of being able to be programmed through the Arduino IDE and many of Arduino's libraries are compatible with the adopted board. Arduino Duemilanove (or UNO that uses the same processor), despite his good

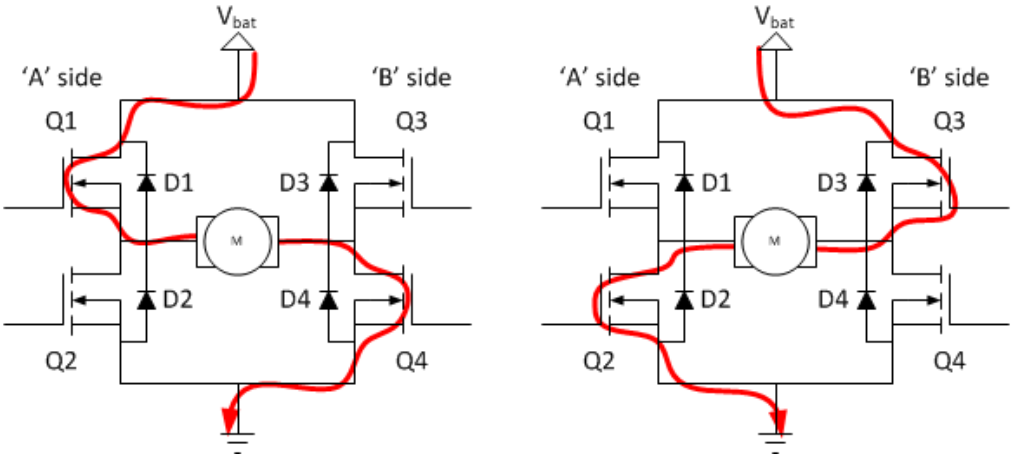



Figure 2.4. Forward and backward signal

performances in terms of computational power, clock frequency, and memory size can not be comparable with the Teensy 4.1 board. Teensy 4.1 has 55 I/O pins and all of them have interrupts capability and 18 pins can be used as analog input since are connected to ADCs. In Table 2.3 are resumed the main characteristics of the Arduino UNO and Teensy 4.1.

Figure 2.5. Teensy 4.1 board

## 2.4 Ultrasonic Sensors

To avoid frontal collision an ultrasonic sensor has been used. It has been mounted on chassis throughout a 3D printed support. The model used for our scope is the HC-SR04, a common sensor used in Arduino home-made projects. This sensor has been chosen since its implementation it is quite simple. The working principle of this sensor is quite simple. A pulse ultrasonic signal is sent from a transmitter and the microcontroller measures the time the wave spends to return to the receiver placed next to the transmitter. By knowing the speeds of sound throughout air the distance from obstacles can be measured.

| | | Arduino UNO | Teensy 4.1 |
|---|---|---|---|
| **General** | **Dimensions [mm]** | 68.58 x 53.34 | 60.96 x 17.78 |
| | **Pricing [€]** | 20-23 | 25-28 |
| **Connectivity** | **I/O Pins** | 14 | 55 |
| | **PWM Pins** | 6 | 35 |
| | **Analog Pins** | 6 | 18 |
| **Computing** | **Processor** | ATMega328P | ARM CortexM7 |
| | **Flash Memory** | 32 kB | 8 MB |
| | **SRAM** | 2 kB | 1 MB |
| | **EEPROM** | 1 kB | 4 kB |
| | **Clock speed** | 16 MHz | 600 MHz |
| | **Voltage Level** | 5V | 3.3V |
| | **USB Connectivity** | A/B USB | Micro-USB |
| **Communication** | **Serial Ports** | 1 | 8 |
| | **SPI Support** | Yes (1x) | Yes (3x) |
| | **CAN Support** | No | 3 (1x CAN FD) |
| | **I2C Support** | Yes (1x) | Yes (3x) |

Table 2.1. Comparison between Arduino UNO and Teensy 4.1

$$d = s_s \cdot \Delta T \tag{2.3}$$

where d is the distance from obstacles, $s_s$ is the speed of sound and $\Delta T$ is the time the wave spends to go and return. The speed of sound is not a constant since it depends on the medium and its local properties that continuously change; however, considering the conditions in which the robot will operate, the value of the sound speed of the air does not vary much in normal operating condition and therefore can be considered constant and equal to 343 m/s.


Figure 2.6. HC-SR04 ultrasonic sensor

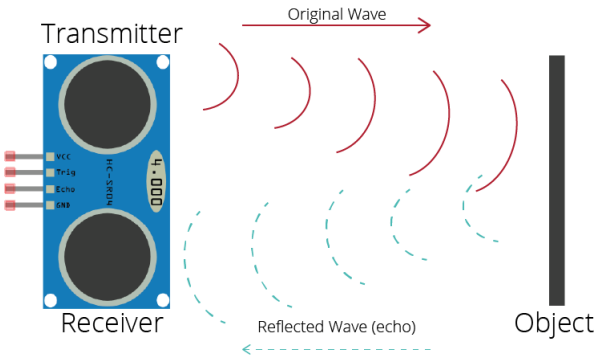



Figure 2.7. Ultrasonic sensor working principle

## 2.5 Encoders

The working principle of the encoders is quite simple. The encoders can sense the motor direction in both senses by detecting holes or marks using 2 digital pins placed next to each other. By considering the Figure 2.8, if the mark is sensed by Pin1 first and by Pin2 after then this means that the encoder, and therefore the encoder to which is connected, rotate clockwise. Contrary if the mark is sensed by the Pin2 and then by Pin1 the encoder is rotating counter-clockwise. To estimate angular displacement as well as the angular speed, each time a marker is sensed first by the Pin1, a counter is incremented in the other case the counter is decremented. By multiplying the counter value with the encoder resolution the displacement can be computed while computing the time between 2 markers the angular speed can obtained. Knowing the angular speed of the wheels is necessary to implement feedback control loop strategy.

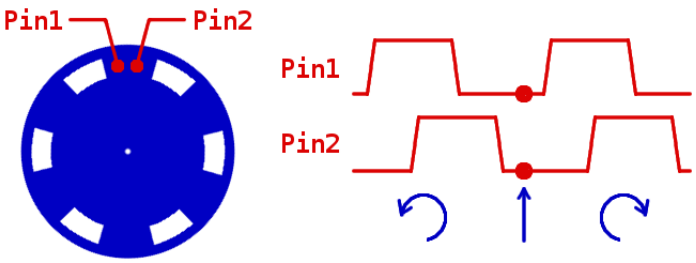



Figure 2.8. Encoders working principle

## 2.6 Current Sensors

An important sensor to add to the system is the current sensor that can be used to realize a closed control loop strategy to pilot the DC motors. The sensor model used in this case is the INA286, a current sensor that exploits the shunt method to measure the current flowing, in both directions, throughout a custom load resistor (the shunt). This sensor returns an analog output that is converted by the Teensy 4.1 embedded ADC. To avoid aliasing a low pass active filter stage (Figure 2.10)has been inserted downstream of the sensor. Aliasing occurs when a system acquires data at an insufficient sampling rate. If a signal contains frequencies higher than Nyquist's, they are mixed with the sampling frequency in the converter's sampler and mapped to frequencies lower than Nyquist's. During sampling, different signals are mixed, which are indistinguishable from each other. Nyquist frequency is the minimum frequency at which an analog signal must be sampled without losing information.

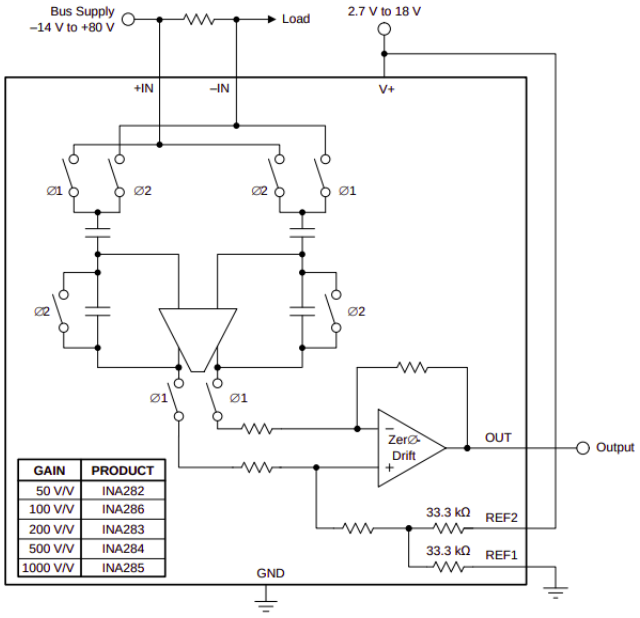



Figure 2.9. INA286 schematic circuit

## 2.7 Voltage Sensor

Another useful sensor to add to a robot's sensor suite is a voltage sensor. It is a necessary component that allows measuring the battery voltage. This information

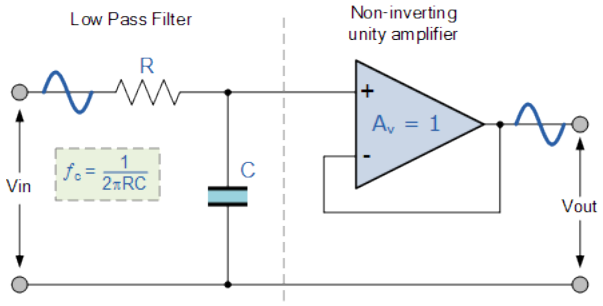



Figure 2.10. Low pass active filter with unitary gain

can be used to evaluate the state of charge of the battery and implement different control strategies based on this data As previously done for the current sensor, here too a low-pass filter has been inserted downstream of the sensor to avoid aliasing phenomena. To realize the voltage sensor a subtractor amplifier stage (Figure 2.11) was used whose transfer function is expressed in the equation 2.4.

$$V_{out} = V_1 \cdot \left(\frac{R_3}{R_2 + R_3}\right) \cdot \left(\frac{R_1 + R_f}{R_1}\right) - V_2 \cdot \left(\frac{R_f}{R_1}\right) \tag{2.4}$$

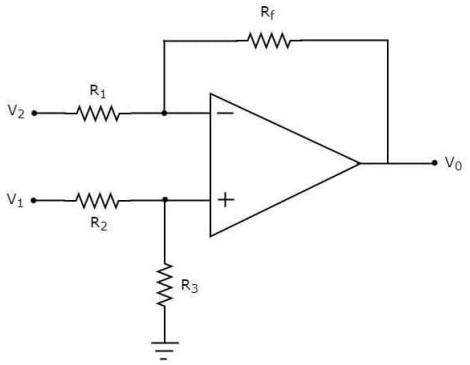



Figure 2.11. Subtractor amplifier stage

# Chapter 3

# Software Architecture

## 3.1 Intoduction

This chapter will discuss the software part of the hardware architecture described so far. All the code has been designed to be as versatile and adaptable as possible to the different functions that the robot has to perform. In particular, this system was conceived as part of a more complex architecture in which the microcontroller represents the interface between the high-level layer represented by a PC that sends commands and a low-level one composed of the microcontroller, the sensor, and all the other auxiliary components. All the software part is structured in such a way that the firmware that will be loaded on the microcontroller is as easy to read as possible and simple to modify and customize. For this reason, most of the code has been implemented in small functions within the robot class (Figure 3.1).

## 3.2 Robot class

This section is the core of the software part of the thesis. All the functions that are called in the main firmware (Section 3.4) reside in this class. In attachment A.1 you can find the source code of the class. The main functions are the ones aimed to accomplish the tasks discussed in 1.2 and are called **read()**,**write()** and **cmd_vel()**. During the development, many other functions have been implemented to simplify the readability of the code

### 3.2.1 read() function

This function is responsible to read the serial port to acquire the data provided by the PC connected to the microcontroller using the USB port. These data are stored in private variables of the class and used for internal processing. The data

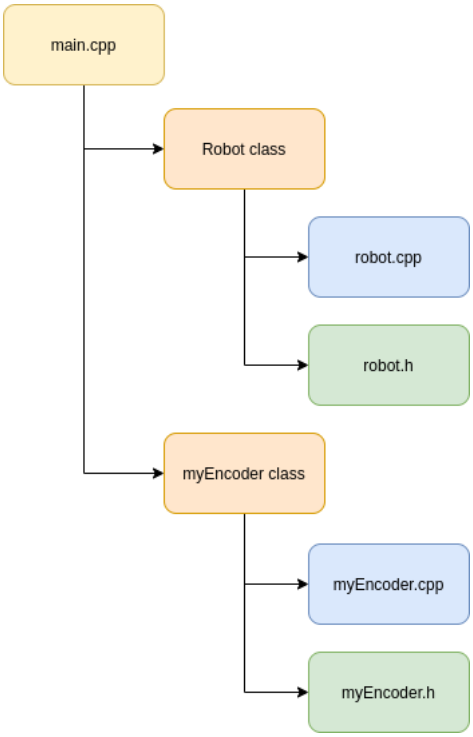



Figure 3.1. Code structure

to be read are the linear speed along $X$ and $Y$ axis and the angular speed around the $Z$ axis, the trigger to turn on a pair of led lights mounted to illuminate the environment, and the trigger a particular action that the robot has to perform.

From a kinematic analysis, by knowing the maximum speed of the electric motors, the maximum linear and angular speeds can be calculated as follow:

$$\begin{aligned} \left|V_{x,max}\right| &= |n_{max} \cdot r| = 0.65 \\ \left|V_{y,max}\right| &= |n_{max} \cdot r| = 0.65 \\ \left|\omega_{z,max}\right| &= \left|\frac{n_{max} \cdot r}{l + w}\right| = 2.16 \end{aligned} \tag{3.1}$$

Where $n_{max}$ is the electric motor's maximum speed, $r$ is the radius of the wheels, while $l$ and $w$ are the wheel-base along the X and Y axis. The obtained numbers should be sent using floats as a data type that are composed of 4 bytes each. To speed up the data transmission a simple operation can be done before sending the data since a resolution of 2 floating points in linear and angular speeds is enough for this scope. You can notice in 3.2 that multiplying by 100 the velocities' maximum value, the obtained results don't exceed the number 255 which is the maximum

integer that can be sent using a single byte. In this way, the three velocities can be sent using only 3 bytes but the information about the sign is lost. To overcome this drawback, an extra byte is used to store all the other information such as the velocities' sign and 2 triggers 1 for the lights and the other one for a special path execution. All of these data are Boolean information that can be stored in a single bit. This simple coding operation reduces the buffer size from 14 bytes to only 4.

$$\begin{aligned} V_{x,max}^{'} &= round(100 \cdot V_{x,max}) = 65 \\ V_{y,max}^{'} &= round(100 \cdot V_{y,max}) = 65 \\ \omega_{z,max}^{'} &= round(100 \cdot \omega_{z,max}) = 216 \end{aligned} \tag{3.2}$$

The PC sends the information stored in a buffer of 4 Bytes composed as shown in table 3.1, where the last byte $I$ contains the sign information of the speeds and the trigger of the light and of the particular path generation (tab. 3.2). The first and the second bytes contain an integer value between 0 and 65 while the third one is between 0 and 216.

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| $V_x^{'}$ | $V_y^{'}$ | $\omega_z^{'}$ | I |
| $0 \div 65$ | $0 \div 65$ | $0 \div 216$ | |

Table 3.1. Buffer structure of read function.

| MSB | | | | | | | LSB |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | Show | Lights | $\omega_z$ sign | $V_y$ sign | $V_x$ sign |

Table 3.2. Coding of extra byte.

### 3.2.2 write() function

The idea behind this function is simple. Since sensors' data are collected and stored in variables to be processed by the microcontroller, they can be sent to the PC to be used in a high-level control. The effort requested for this function is relatively low while many are the benefits. The data sent to the PC are the wheels' speed, Current sensors' data, Voltage sensor' data and Ultrasonic sensor' data.

As already done in the read function, also, in this case, a coding and decoding process can be used to transfer the message between the PC and the microcontroller to reduce the amount of data to be sent. To find out the range of value

the wheels' speed can take, we have to analyze the datasheet of the electric motors. The no-load speed of the electric motors is 8100 rpm. This is the maximum speed the motor can have when it is powered by the nominal Voltage in a no-load condition. For this reason, we can assume that the motor's speed will be lower or equal to this value. The wheel is connected to the motor through a gearbox with a transformation ratio of 1/64 that reduces the maximum wheels' speed to 127 rpm. By considering that the motor can rotate forward and backward we can assume that the wheel speed is between -127 and 127. Since with a single byte, an integer from 0 to 255 can be sent, a coding procedure can be done to reduce the wheel speed data to a single byte. Since the resolution of the units is enough for this scope, by adding 128 to the speed value we can send it through a single byte. When it will be read from the PC a decoding process is necessary.

As regards the current sensors, estimating the maximum value is not so easy. The current flowing in the electric motor is strictly related to the load applied to the motor itself. From the datasheet, the rated power for a continuous operation is 1.5 A, so we can assume that 2.5 A of the maximum value in continuous operation is enough. In this maximum are not considered the current peaks that can be generated in transient conditions since the time they spend at values bigger than 2.5 A is negligible. The current can flow through the motor in both directions so it can take a value between -2.5 A and 2.5 A and the current sensor transform it into a value between 0 and 3.3 V that is read by the ADC of the microcontroller by following the transfer function.

$$V = \frac{3.3}{5} i = 0.66i \tag{3.3}$$

For the scope of this work, two digits floating value of the current is acceptable. To send this data we send a byte for the value e a bit of an extra byte for the sign.

In the following tables are resumed the information about the buffer structure of the write function (Table 3.4) and the coding of the byte for the current's sign (Table reftab:currentsigns).

| MSB | | | | | | | LSB |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $i_4$ sign | $i_3$ sign | $i_2$ sign | $i_1$ sign |

Table 3.3. Coding of currents' sign byte.

As regards the voltage sensor used to check the battery level some consideration must be done. First of all, we have to define the maximum and the minimum voltage level. In our case, we used a LiPo battery pack with 3 cells in series. The maximum voltage that each cell can reach without risk of explosion is 4.2 V while the lower voltage without risk of damage to the battery is 3.2. So, we can calculate

the range of a 3S LiPo battery that is 9.6÷12.6 V resulting in a range of 3 V. To be sure to sense the correct voltage value we can consider admit voltage between 10 and 14 V. in this way we consider completely discharging a battery at 10 V and we can measure battery voltage up to 14 V. By using the subtractor stage explained in Section 2.7, a transfer function is created to transform a voltage between 10 and 14 V to a voltage between 0 and 3.3 V that is the maximum input voltage of Teensy's ADCs. The ADC of the Teensy is configured as a 10-bit resolution ADC sampling the input voltage from 0 to 3.3 V into a digital value from 0 to 1023. Then the obtained value is converted into an integer between 0 and 255 to be sent using a single byte. Through this simple operation, a resolution of 0.0157 is sufficient for the scope of this work.

The last sensor data to send from the microcontroller to the PC through the serial port is the one carried out by the ultrasonic sensor. The maximum and minimum values of the ultrasonic sensor from the datasheet are 400 and 2 cm. For the scope of this work, it is sufficient to measure up to 255 cm and send the result in only one byte as an integer. In this case, the operation to send the data is a saturation operation that is set to 255 all the values $\geq 255$. Since to compute the distance between the obstacle with the ultrasonic sensor it is necessary to wait for the wave that goes to the obstacle and return often a different strategy is adopted. When the distance grows up also the time spent by the wave grows up. For example, if the obstacle is at 50 cm from the robot the time to the wave to go and return is about 3 ms. To reduce the time in which the microcontroller waits for the wave a timeout is added. This timeout is responsible to stop the function and return that the obstacle is out of range. In this case, a timeout of 1.75 ms is set that allow measuring distance up to 30 cm.

| 1 | $\cdots$ | 4 | 5 | $\cdots$ | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|
| $n_1$ | $\cdots$ | $n_4$ | $i_1$ | $\cdots$ | $i_4$ | Signs | $V_{sense}$ | d |
| $0 \div 255$ | .. | $0 \div 255$ | $0 \div 250$ | $\cdots$ | $0 \div 250$ | S | $0 \div 255$ | $0 \div 30$ |

Table 3.4. Buffer structure of write function.

### 3.2.3 cmd_vel() function

This section will describe the main function of the robot class, the cmd_vel function. It is the core of the project since it makes the robot able to move. To perform this task many steps are necessary and for this reason, this function has been divided into 5 simpler functions called **speedSaturator**, **rateLimiter**, **eStop**, **transformationMatrix** and **writeSpeeds**

The first function is used to compute the maximum angular speed $\omega_z$ and linear

velocities $V_x$ and $V_y$. The microcontroller should always check the reference input for 2 main reasons: avoid mechanical failures and evaluate the feasibility of the reference.

The firsts shall be produced by high motor speed caused by overvoltage. The maximum rpm that the motor can handle is 11.000 but in normal use, it should work at the nominal one that is, as said before, 9100. The other check that the microcontroller has to do regards the feasibility of the given inputs. The robot has physical constraints that must be satisfied. In fact, due to the limit on the motor speeds, if the robot turns around its center, it can't go forward. This means that if incoherent inputs are sent by the PC, the microcontroller has to manage them. This function performs this task by saturating the speed that exceeds its maximum feasible value. To compute the maximum feasible value, we have to assign priority to the 3 velocity speeds since are dependent on each other. The feasible values are all the ones contained inside the octahedron (Fig. 3.2) where the height is the maximum angular velocity while the sides are equal to the double of $V_x$ and $V_y$. The one that has the major priority is the angular speed $\omega_z$ and its maximum feasible value coincides with the maximum value that is equal to $4\pi\ rad/s$. So, when an input angular speed is provided microcontroller checks if is lower than the maximum value if yes the input is accepted otherwise it is saturated. Then the maximum value of the $V_y$ is computed based on the actual value of $\omega_z$ by the following equation:

$$V_{y,max} = r * n_{max} - (l + w) \cdot \omega_z \tag{3.4}$$

Then the value of $V_y$ is computed by saturating the input to the found value. The same logic is used for the $V_x$ value by computing the $V_{x,max}$ value with 3.5.

$$V_{x,max} = V_{y,max} - \left|V_y\right| \tag{3.5}$$

The flowchart of this function is shown in figure 3.3.

The second function is used to smooth the velocity profile and avoid abrupt accelerations that could cause a high value of current flowing into the electric motors. To obtain this result a simple algorithm has been developed. The idea is that each iteration the reference value enters the function as input. Then, the smoothed speed is computed by adding to the previous value of the velocity, a term composed by the difference between the reference value and the previous one. The figure 3.4 shows the flowchart of the function.

Another simple but important function is the *eStop* one. It is responsible to set only the velocity along the $X$ axis to 0 if an obstacle is present in front of the robot and the speed is positive. This function has been implemented to avoid frontal collisions.

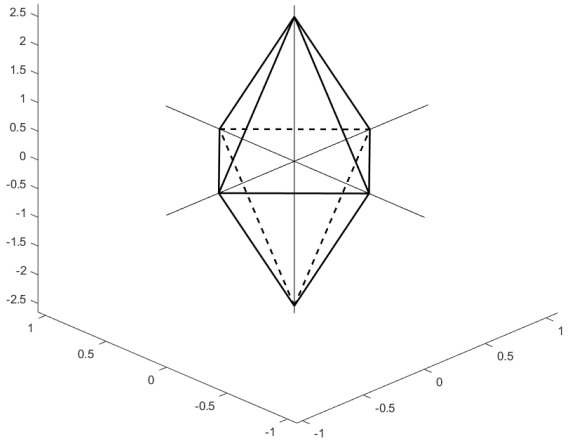



Figure 3.2. Speed limit graph

The fourth function uses the kinematic equations 1.6 to transform the input received from the PC and processed by the previous functions to compute the angular speed of the wheels and store them in a vector. Since the kinematic of the robot is taken into account only in this function, changing the robot will be sufficient to update the kinematic relations in this function to adapt the entire code to the new robot.

The output vector data of the previous function is the input of the last function that uses this data to compute the PWM signal to send to the motors' drivers.

## 3.3 myEncoder class

This class has been used to acquire encoder data, in particular, to compute the angular speed from the encoders. Since the encoder is widely used an open-source library was available. This library was developed by Paul Stoffregen a software developer of PJRC, the company that designed Teensy boards. Through this class, you can assign a value to the counter and read the value of the counter. This class uses the interrupt strategy to count the encoder tick. An interrupt is an request of interrupt the normal execution process of the microcontroller. if the request is accepted, the microcontroller stops the execution, saves its state,

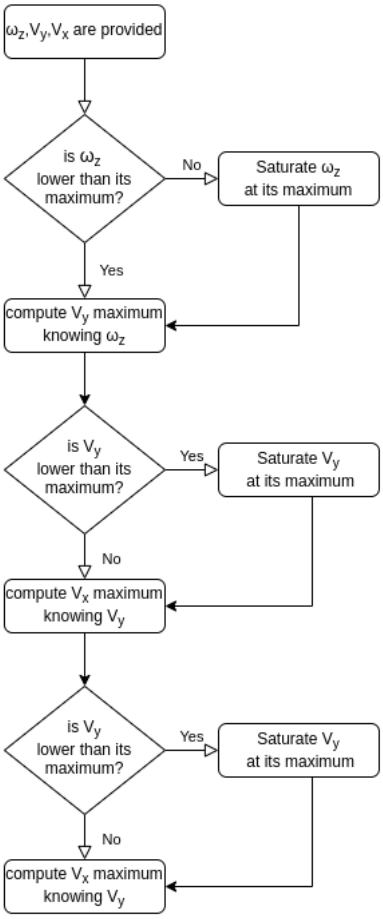



Figure 3.3. Speed saturation algorithm flowchart

and calls a function called interrupt handler that contains the interrupt routine instructions. When the interrupt routine has finished, the normal routine process is restarted from the interrupt point. I improved this class by adding a method (myEncoder::speed()) to compute the encoder speed.

## 3.4 The firmware

Analyzed the single pieces that make up the project, it's time to glue them together in the firmware that will be loaded and then executed on the microcontroller. The firmware, how is shown in section A.3, is composed of a few lines of code since the entire complexity of the algorithms resides inside the robot class. In lines 1-2 the Arduino library and the Robot class are included. At lines 4-5 an instance of the

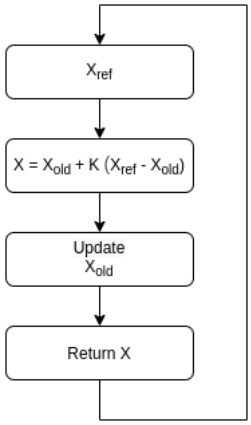



Figure 3.4. Rate limiter algorithm flowchart

class is created and a pointer to a float vector of 3 components is initialized. Then the loop routine starts. First of all, the read() function is called and the velocities are stored in the pointer. Then the data from the sensor data are acquired using the four functions. At the acquisition phase follow the cmd_vel() function explained in Section 3.2.3. This implementation is only an example used for testing all the functionalities of the robot. It can be adapted by the user based on the use case.

# Chapter 4

# Marvin Project

## 4.1 Overview

In the previous sections, a general architecture describing the common hardware that can be found on mobile robots employed in service robotics has been discussed. Moreover, a complete design of a motion architecture was presented from the power train design to the sensor choice. Then, the software part was developed to control the robot and make it move by acquiring the sensor data and piloting the DC motors. In the following sections, we will discuss a particular application of this work: the Marvin Project. In figure 4.1 is shown the low-level hardware architecture employed. It contains all the hardware already with the addition of some modules that in the has been added to future implementation of the robot.

## 4.2 Marvin Hardware Architecture

The Marvin project was born to create a mobile robot to be used in a domestic environment to help and assist elderly people. The robot has to navigate autonomously in a partially unknown environment and perform some tasks like leading the assisted person from one room to another one or calling an emergency number when the function is triggered by the assisted person with a vocal command. By starting from the already discussed architecture, in this project, some components have been added to accomplish some predefined tasks depending on the use case. An embedded computer was added to run the high-level algorithm for navigation purposes. To perform this task the ROS2 framework has been used. ROS2 is the second version of ROS (Robotic Operating System) which is an open-source set of software libraries and tools that helps the development of robot applications. One of the projects that have been used in the Marvin project is NAV2 which is a navigation tool intended to find a safe path from point A to point B. It can also

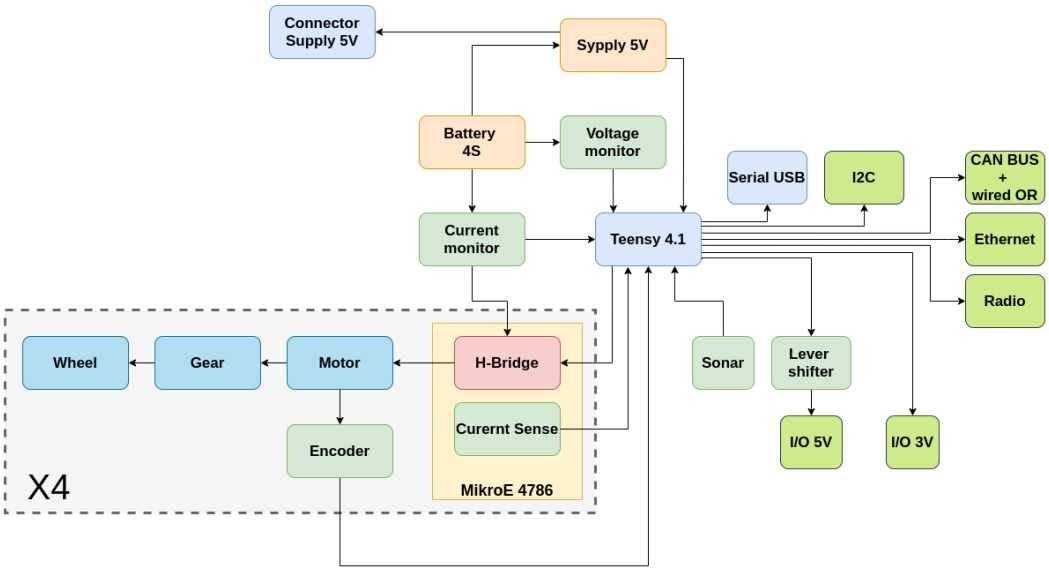



Figure 4.1. Marvin's hardware architecture

be applied in other applications that involve robot navigation since it contains a dynamic path planning algorithm to avoid obstacles. To navigate NAV2 needs a map that can be built using a depth camera or a LiDAR. A stereo camera is composed of 2 sensors placed next to each other at a fixed distance like human eyes giving accurate real-time depth perception. It can achieve this by using the two sensors to triangulate similar pixels from both 2D planes. The LiDAR is a sensor composed of light emitters and receivers that compute the time that the impulse of light generated by the emitter spend to go and return to the receiver can compute the distance from an obstacle. A tracking camera with an embedded IMU has been used to obtain odometry data using an algorithm of visual odometry[2] (VO). This algorithm can determine the position and the orientation of a robot by analyzing 2 consecutive frames and calculating the displacement of 2 points (or pixels) of interest and the time it takes to make that movement. In this case, the camera used for the VO is a stereo camera but exist algorithms exploiting mono cameras. By combining the depth and the tracking stereo cameras it is possible to use NAV2 within a SLAM (Simultaneous Localization and Mapping) algorithm. SLAM[4] is a chicken-and-egg problem that is very common in autonomous vehicles, mobile robot, and drones, in which the robot build and update a map of an unknown environment while trying to localize itself in it. In recent years many SLAM algorithms that fuse different kinds of sensors were born. In particular, this project has been used a SLAM algorithm that uses a 3D LIDAR to obtain a point cloud used to generate the map while as regards the localization part

the job was entrusted to a stereo camera with 2 fish-eye lenses and an integrated IMU obtaining a Visual Inertial Odometry (VIO). The Depth stereo camera has been mounted on vertical support whose orientation can be changed using a servo mechanism controlled by the embedded computer. Other components added to the robot are 2 led lights, in the front and rear of the vehicle. These lights are intended to illuminate the environment to help cameras see in dark conditions. Cameras are used to get odometry information and to create point clouds to avoid obstacles. The last component is a USB speaker with an embedded microphone. It has been mounted on the robot to allow communication between the robot and the assisted person. The embedded computer has a vocal assistant that allows the robot to receive commands by using the voice.

## 4.3 Board Prototype

The high number of components needed for the project made necessary the design of a PCB through which it was possible to connect all the devices more efficiently. In fact, in the first development part, the test of the components was made by using a breadboard and some wire for the connection. Gradually the complexity of the projected increase and the project was migrated on matrix board to build the prototype of the board. When the prototype was tested and everything worked properly, a PCB was designed.

To design the PCB has been used the software KiCad, an open-source software suite for Electronic Design Automation (EDA). The programs handle Schematic Capture, and PCB Layout with Gerber output. Using this software has been created the schematic (Annex B). To power supply the components, R-78B5.0-2.0 DC/DC converter has been added that takes the battery voltage of the battery and carries out a 5 V constant voltage. To decouple the 2 different power supplies 4 capacitors have been added.

To measure the voltage and the current provided by the battery, 2 sensors have been implemented. To design the voltage sensor has been used a subtractor stage as explained in section 2.7 and an active low pass filter to avoid aliasing phenomena. The OPA2134 operational amplifier has been used to implement the 2 stages.

The current monitor has been implemented using the INA286 SoC, a current shunt current sensor that returns an analog value proportional to the current flowing into the shunt resistor. This analog input is filtered using, also in this case, a low pass active filter.

As regards the current sensors related to the DC motors, the current monitor embedded on the motor drivers has been used. It provides an analog input proportional to the current flowing into the driver. Since the current may be positive

or negative, it was implemented in a particular configuration that uses half of the scale of the sensor for the positive value and half of the scale for the negative ones. Also in this case, a filter in between the sensor output and the ADC input of the microcontroller.

The schematic was designed not only considering the necessary components to this project but also keeping in mind the future expansion and upgrade of the platform. Some extra components have been added like a CAN bus connector and its interface module that might be used to have a more robust serial communication between the microcontroller and the other devices. Moreover, it has been added a radio module to connect a radio control that in some cases may be useful. Since the Teensy 4.1 can not provide a 3.3V voltage signal a level shifter was added to be able in future upgrade to use also components that require a 5 V input signal since Teensy 4.1 provides only a 3.3 V output signal.

# Chapter 5

# Conclusions

During the last decade, there has been a notable increase in interest in mobile robots for service applications. This phenomenon has led to the production by many companies of multiple models of mobile robots. In the first part of the thesis, both hardware and software of many robots available in the market have been analyzed and the common modules have been extrapolated to be used as guidelines in this project. After this preliminary analysis, in which the common needs of the mobile robots have been defined, we moved on to the design and development of the software and hardware part.

This work aimed to create a framework that could be used as a basis for the development of mobile robots in the main fields of application of service robotics. This was achieved by successfully developing a working prototype for home care applications for the elderly in the project called Marvin described above in which both the hardware and the software parts have been integrated.

# Appendix A

# Source code

## A.1 Robot Class

### A.1.1 Robot.cpp

```
#include "Robot.h"
#include <string.h>
#include <myEncoder.h>

#include <stdio.h>
 // FRONT LEFT, FRONT REAR, REAR LEFT, REAR RIGHT
int encoderPhaseA[4] = {encoder_FL_A, encoder_FR_A, encoder_RL_A, encoder_RR_A};
 // FRONT LEFT, FRONT REAR, REAR LEFT, REAR RIGHT
int encoderPhaseB[4] = {encoder_FL_B, encoder_FR_B, encoder_RL_B, encoder_RR_B};

myEncoder *encoder[4];          // Create a pointers vector of 4 Encoder objects

Robot::Robot()
{
  /*  This is the class constructor.
      In this function are initialized all the variables and pins
      needed for the class.
  */
  for(int i=0; i<4; i++)
  {
    // Set PWM pins frequency to 20kHz
    analogWriteFrequency(pwmMotor[i],20000);
    // set direction pins mode to OUTPUT
    pinMode(dirMotor[i], OUTPUT);
    // set PWM command pins mode to OUTPUT
    pinMode(pwmMotor[i], OUTPUT);
    pinMode(currentPin[i], INPUT);
    analogWrite(pwmMotor[i],0);                 // set motors speed to 0
    digitalWrite(dirMotor[i], HIGH);
  }
  // set ultrasonic sensors trig pins mode to OUTPUT
  pinMode(trigPin, OUTPUT);
  // set ultrasonic sensors echo pins mode to INPUT
  pinMode(echoPin, INPUT);
  pinMode(lightPin, OUTPUT);
```

```
  digitalWrite(lightPin,LOW);
  // When HIGH inverters are ON, otherwise are OFF
  digitalWrite(not_stby,HIGH);

  t_old  = micros();
  // initialize Serial communication with 115200 BOUDRATE
  Serial.begin(115200);
  // wait until Serial port is opened
  while(!Serial){}
  // print a message when serial port is ready
  Serial.write("Serial port is ready!\n");
  for(int i = 0; i < 4; i++){
  encoder[i] = new myEncoder(encoderPhaseA[i], encoderPhaseB[i]);
  }
}
/*-------------------------------------------------------------------*/
Robot::~Robot(){}
/*-------------------------------------------------------------------*/
void Robot::write()
{
  /* This function is used to write to the serial port */
  // create buffer bigger enough to contains all data
  unsigned char write_buf[100];
  for (int i = 0; i < 4; i ++){
    write_buf[i] = (unsigned char)this->getRPM(i);
    if (write_buf[i] >= 0){
      write_buf[4] |= sign[i];
    }
  }

  Serial.write(write_buf, sizeof(write_buf));   // write data to the serial port
}
/*-------------------------------------------------------------------*/
void Robot::read(float* velocitiesPtr)
{
  /*  This function is used to read data from serial port.
      In particular, the data interested in are the 3 velocities:
        - Along X axis and extra byte to understand if positive or negative
        - Along Y axis and extra byte to understand if positive or negative
        - Around Z axis and extra byte to understand if positive or negative
        - and extra byte to understand if positive or negative
  */

  byte buffer[bufferSize];
  if (Serial.available() > 0){
    Serial.readBytesUntil(terminator,buffer,100);
    // Serial.println("Il messaggio inizia qui:");
    for(int i = 0; i < 3; i++){
      velocitiesPtr[i] = (float)buffer[2*i]/100;
      if (buffer[2*i+1] == 1){
        velocitiesPtr[i] = -velocitiesPtr[i];
      }
    }
    if ((buffer[6] & 0x01) != 0){
      lightON();
    }
    else if ((buffer[6] & 0x01) != 0){
      unsigned long t0 = micros();
      showMe(t0);
    }
    else
```

```
      lightOFF();
  }
  // Serial.println(buffer[3]);
}
/*-------------------------------------------------------------------*/
float Robot::getRPM(int encoderNumber)
{
  /*  This function uses Encoder objects array
      to get wheels speed in a more compact way
  */
  float speed;
  if (encoderNumber < 0 || encoderNumber > 3){
    Serial.print("Insert a correct encoder number");
  }
  speed = encoder[encoderNumber]->speed();
  if(digitalRead(dirMotor[encoderNumber]) == LOW){
    speed = -speed;
  }
  // Serial.println(speed[encoderNumber]);
  return speed;
}
/*-------------------------------------------------------------------*/
void Robot::getRPM(float *wheelSpeed)
{
  for(int i = 0; i < 4; i++){
    wheelSpeed[i] = getRPM(i);
  }
}
/*-------------------------------------------------------------------*/
float Robot::getCurrent(int currentSensor)
{
  float current;
  int analogVal = analogRead(currentPin[currentSensor])+analogOffset;
  if(   analogVal >= ((resolution+1)/2-analogTH) &&
        analogVal <= ((resolution+1)/2+analogTH)){
    current = 0.0;
  }
  else{
current = (float)(analogVal*Voltage_max/resolution-Voffset)/(sensorGain*windings);
  }
  return current;
}
/*-------------------------------------------------------------------*/
float Robot::getDistance(){
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  unsigned long duration = pulseIn(echoPin, HIGH, timeout);
  if (duration == 0){
    return 0;
  }
  else{
    return duration * 0.034 / 2;
  }
}
/*-------------------------------------------------------------------*/
bool Robot::eStop(){
  /*  This function is responsible to stop
      the robot when sonar sense an obstacle
```

```
      inside the set range.
  */
  bool last_state = 0;
  float distance = getDistance();
  bool filterOutput = sonarFilter(distance);
  if (distance == 0 && filterOutput == 1){
    last_state = 0;
  }
  else if (distance != 0 && filterOutput == 1) {
    last_state = 1;
  }
  return last_state;
}
/*-------------------------------------------------------------------*/
void Robot::forward(int pwmVal)
{
  /*  This function is used to move the robot forward
      by imposing a PWM signal to the inverter.
      The PWM value is the argument of the function.
  */
  for(int i=0; i<4; i++)
  {
    digitalWrite(dirMotor[i], HIGH);
    analogWrite(pwmMotor[i], pwmVal);
  }
}
/*-------------------------------------------------------------------*/
void Robot::backward(int pwmVal)
{
  /*  This function is used to move the robot backward
      by imposing a PWM signal to the inverter.
      The PWM value is the argument of the function.
  */
  for(int i=0; i<4; i++)
  {
    digitalWrite(dirMotor[i], LOW);
    analogWrite(pwmMotor[i], pwmVal);
  }
}
/*-------------------------------------------------------------------*/
void Robot::left(int pwmVal)
{
  /*  This function is used to move the robot left
      by imposing a PWM signal to the inverter.
      The PWM value is the argument of the function.
  */
  digitalWrite(dirMotor[0], LOW);
  digitalWrite(dirMotor[1], HIGH);
  digitalWrite(dirMotor[2], LOW);
  digitalWrite(dirMotor[3], HIGH);
  for(int i=0; i<4; i++)
  {
    analogWrite(pwmMotor[i], pwmVal);
  }
}
/*-------------------------------------------------------------------*/
void Robot::right(int pwmVal)
{
  /*  This function is used to move the robot right
      by imposing a PWM signal to the inverter.
      The PWM value is the argument of the function.
```

```
  */
  digitalWrite(dirMotor[0], HIGH);
  digitalWrite(dirMotor[1], LOW);
  digitalWrite(dirMotor[2], HIGH);
  digitalWrite(dirMotor[3], LOW);
  for(int i=0; i<4; i++)
  {
    analogWrite(pwmMotor[i], pwmVal);
  }
}
/*-------------------------------------------------------------------*/
void Robot::turn(char* cw_or_ccw, int pwmVal)
{
  /*  This function is used to turn the robot clockwise
      or counter-clockwise by imposing a PWM signal to
      the inverter and a string to select the direction.
      The string and the PWM value are the arguments of the function.
  */
  if(strcmp(cw_or_ccw,"cw") == 0)
  {
    digitalWrite(dirMotor[0], HIGH);
    digitalWrite(dirMotor[1], LOW);
    digitalWrite(dirMotor[2], LOW);
    digitalWrite(dirMotor[3], HIGH);
  }
  else
  {
    digitalWrite(dirMotor[0], LOW);
    digitalWrite(dirMotor[1], HIGH);
    digitalWrite(dirMotor[2], HIGH);
    digitalWrite(dirMotor[3], LOW);
  }
  for(int i=0; i<4; i++)
  {
    analogWrite(pwmMotor[i], pwmVal);
  }
}
/*-------------------------------------------------------------------*/
void Robot::cmd_vel(float Vx, float Vy, float Vz)
{
  /*  This function is used to move the robot
      by imposing 3 velocities: Vx, Vy and Vz.
      The PWM value is the argument of the function.
  */
  float velocities[3];
  velocities[0] = Vx;
  velocities[1] = Vy;
  velocities[2] = Vz;
  float *velocitiesPtr = velocities;

  speedSaturator(velocitiesPtr); // saturate speeds with an Octahedron shape
  rateLimiter(velocitiesPtr);  // limit speed rate
  if (eStop() && velocitiesPtr[0] > 0){
    velocitiesPtr[0] = 0.0;
  }
  float *rpm = new float[4];
  transformationMatrix(velocitiesPtr,rpm);
  writeSpeeds(rpm);
}
/*-------------------------------------------------------------------*/
void Robot::stop()
```

```
{
  /*  This function is used to stop the robot
      by imposing a PWM signal equal to 0.
  */
  for (int i = 0; i<4; i++){
    analogWrite(pwmMotor[i], 0);
  }
}
/*-------------------------------------------------------------------*/
float Robot::saturation(float x, float x_max)
{
// Controllo se all'interno del range, altrimenti saturo
  if(x >= x_max){
    x = x_max;
  }
  else if (x <= -x_max){
    x = -x_max;
  }
  return x;
}
/*-------------------------------------------------------------------*/
void Robot::speedSaturator(float* velocitiesPtr)
{
  float Vx = velocitiesPtr[0];
  float Vy = velocitiesPtr[1];
  float Vz = velocitiesPtr[2];

  float Vz_max = n_max*r/(l+w);        // Calcolo Vz massima
  Vz = saturation(Vz, Vz_max);
  float Vy_max = r*n_max-(l+w)*Vz;     // Calcolo Vy massima
  Vy = saturation(Vy, Vy_max);
  float Vx_max = Vy_max - abs(Vy);
  Vx = saturation(Vx, Vx_max);

  velocitiesPtr[0] = Vx;
  velocitiesPtr[1] = Vy;
  velocitiesPtr[2] = Vz;
}
/*-------------------------------------------------------------------*/
void Robot::rateLimiter(float *velocitiesPtr){
  float K = 0.5;
  unsigned long dt = micros()-t_old;
  for (int i = 0; i < 3; i++){
    velocitiesPtr[i] = V_old[i]+ K*dt*1e-6;
    V_old[i] = velocitiesPtr[i];
  }
}
/*-------------------------------------------------------------------*/
void Robot::lightON(){
  digitalWrite(lightPin, HIGH);
}
/*-------------------------------------------------------------------*/
void Robot::lightOFF(){
  digitalWrite(lightPin, LOW);
}
/*-------------------------------------------------------------------*/
void Robot::showMe(unsigned long t0){
  float Vx0, Vy0, Th, Vz, k, kd;
  float R = 1.0;
  bool endShow = true;
  while(!endShow){
```

```
    float t = (micros() - t0) * 1e-6;
    if (t >= 0 and t < 15){
      k = M_PI * (t/30 - 1/(2*M_PI) * sin(2*M_PI * t/30));
      kd = M_PI/30 * (1-cos(2*M_PI * t/30));
      Vx0 = -R * sin(k) * kd;
      Vy0 = R * cos(2*k) * kd;
      Vz = 0;
      Th = M_PI;
    }
    if (t >= 15 and t < 37.5){
      kd = M_PI/15;
      k = kd*t - M_PI/2;
      Vx0 = -R * sin(k) * kd;
      Vy0 = R * cos(2*k) * kd;
      Vz = 0;
      Th = M_PI;
    }
    if (t >= 37.5 and t < 60){
      kd = M_PI/15;
      k = kd*t - M_PI/2;
      Vx0 = -R * sin(k) * kd;
      Vy0 = R * cos(2*k) * kd;
      Vz = kd;
      Th = k - M_PI;
    }
    if (t >= 60 and t < 75){
      k = M_PI * ((t-15)/30 - 1/(2*M_PI) * sin(2*M_PI*(t-15)/30)) + 2*M_PI ;
      kd = M_PI/30 * (1-cos(M_PI * (t-15)/15));
      Vx0 = -R * sin(k) * kd;
      Vy0 = R * cos(2*k) * kd;
      Vz = kd;
      Th = k - M_PI;
    }
    if (t >= 75 and t < 105){
      k = 2*M_PI * ((t-75)/30 - 1/(2*M_PI) * sin(2*M_PI*(t-75)/30)) + 6*M_PI ;
      kd = 2*M_PI/30 * (1-cos(2*M_PI * (t-75)/30));
      Vx0 = 0;
      Vy0 = -2*R * cos(k) * kd;
      Vz = ((2*R * cos(k))/(4*R*R * sin(k)*sin(k) +1)) * kd;
      Th = M_PI + atan2(-2 * R * sin(k), R);
    }
    if (t > 105) {
      endShow = false;
      Vx0 = 0;
      Vy0 = 0;
      Vz = 0;
    }
    float Vel1 = Vx0 * cos(Th) + Vy0 * sin(Th);
    float Vel2 = Vy0 * cos(Th) - Vx0 * sin(Th);
    float Vel3 = Vz;
    cmd_vel(Vel1,Vel2,Vel3);
  }
}
/*-------------------------------------------------------------------*/
bool Robot::sonarFilter(float sonarDistance){
  /*  This function is responsible to "filter"
      sonar output. An output (inside or outside)
      is considered valid if repeated for n times.
      The maximum value for n is 8.
  */
  filter = filter << 1;
```

```
  int n  = 3;
  for (int i = 0; i < n; i++){
    filter &= ~(1UL << (7-i));
  }
  if (sonarDistance != 0){
    filter |= 1UL << 0;
  }
  if (filter == 7 || filter == 0){
    return 1;
  }
  else
    return 0;
}
/*-------------------------------------------------------------------*/
void Robot::transformationMatrix(float *velocities, float *speedsPtr){
  float Vx = velocities[0];
  float Vy = velocities[1];
  float Vz = velocities[2];
  speedsPtr[0] = (Vx + Vy + (l+w)*Vz)*30/M_PI/r;
  speedsPtr[1] = (Vx - Vy - (l+w)*Vz)*30/M_PI/r;
  speedsPtr[2] = (Vx - Vy + (l+w)*Vz)*30/M_PI/r;
  speedsPtr[3] = (Vx + Vy - (l+w)*Vz)*30/M_PI/r;
}
/*-------------------------------------------------------------------*/
void Robot::writeSpeeds(float *speedsPtr){
  float speeds[4];
  for (int i = 0; i < 4; i++){
    speeds[i] = speedsPtr[i];
  }
  for (int i=0;i<4;i++)
  {
    if(speeds[i] < 0)
    {
      digitalWrite(dirMotor[i],LOW);
      speeds[i] = -speeds[i];
    }
    else
    {
      digitalWrite(dirMotor[i],HIGH);
    }
    analogWrite(pwmMotor[i],map(speeds[i],0,n_max*30/M_PI,0,255));
  }
}
```

### A.1.2 Robot.h

```
#ifndef Robot_hpp
#define Robot_hpp

#include "Arduino.h"
#include "definePins.h"
#include "robotParameters.h"

class Robot
{
    private:
        /* Pins definition */
        const int dirMotor[4] = {motorDirFR, motorDirFL, motorDirRR, motorDirRL};
        const int pwmMotor[4] = {motorPwmFR, motorPwmFL, motorPwmRR, motorPwmRL};
```

```
    const int not_stby = notStby;
    const int currentPin[4] = {current_1, current_2, current_3, current_4};
    const int echoPin = echo;
    const int trigPin = trig;
    // Serial
    const char terminator = '\x02'; //terminator byte for Serial Communication
    // Sonar
    // Minimum distance in cm from obstacle then call stop() function
    const int distanceMin = 15;
    const int timeout = (long)2*distanceMin/0.034; //us
    unsigned char filter;
    // move robot
    float V_old[3];
    unsigned long t_old = 0;
    void writeSpeeds(float* speedsPtr);
    bool sonarFilter(float sonarDistance);
    float saturation(float x, float x_max);
    void transformationMatrix(float *velocities, float *speedsPtr);
    void speedSaturator(float* velocitiesPtr);
    void rateLimiter(float *velocitiesPtr);
    // Current sensor
    unsigned const char Nbit = 10;
    const int resolution = pow(2,Nbit)-1;
    const float Voltage_max = 5.0;
    const int analogTH = 2;
    const int analog0 = 471;
    const int analogOffset = resolution/2-analog0;
    const float Voffset = Voltage_max/2;
    const float sensorGain = 0.034429;      // 0.037  before calibration
    const unsigned char windings = 10;


    const int sign[4] = {0x01, 0x02, 0x04, 0x08};

public:
    Robot();                                    // Library Constructor
    ~Robot();                                   // Library Deconstructor
    // Serial write and read
    void read(float * velocitiesPtr);           // Read from serial port
    void write();                               // Write on serial port
    // Move Robot
    void stop();                                // Stop Robot
    bool eStop();
    void forward(int pwmVal);                   // Move the robot forward
    void backward(int pwmVal);                  // Move the robot backward
    void left(int pwmVal);                      // Move the robot left
    void right(int pwmVal);                     // Move the robot right
    void turn(char* cw_or_ccw, int pwmVal);     // Turn the robot
    void cmd_vel(float Vx,float Vy,float Vz); // Move the robot any direction
    //Encoders
    void getRPM(float *wheelSpeed);             // Get 4 wheels speeds
    float getRPM(int encoderNumber);            // Get 1 wheel speeds
    // Ultrasonic sensors
    float getDistance();                        // Get sonar distance
    // current sensor
    float getCurrent(int motorNumber);          // Get 1 motor current
    // Lights
    void lightON();                             // turn on lights
    void lightOFF();                            // turn off lights
    // show time
    void showMe(unsigned long t0);              // Show
```

```
};
#endif
```

### definePins.h

```
const int motorDirRR = 22;
const int motorDirRL = 20;
const int motorDirFL = 17;
const int motorDirFR = 15;

const int motorPwmRR = 23;
const int motorPwmRL = 219;
const int motorPwmFL = 18;
const int motorPwmFR = 14;

const int notStby = 21;

const int encoder_RR_A = 0;
const int encoder_RL_A = 3;
const int encoder_FL_A = 5;
const int encoder_FR_A = 6;

const int encoder_RR_B = 1;
const int encoder_RL_B = 2;
const int encoder_FL_B = 4;
const int encoder_FR_B = 7;

const int echo = 9;
const int trig = 10;

const int current_1 = 25;
const int current_2 = 24;
const int current_3 = 26;
const int current_4 = 27;

const int lightPin = 28;

const int bufferSize = 100;
```

### robotParameters.h

```
#include <math.h>
const float r = 0.0515;          // wheel radius [m]
const float l = 0.15162;         // wheelbase [m]
const float w = 0.14933;         // wheelbase [m]
const float n_max = 4*M_PI;      // omaga max [rad/s]
const float V_max = r*n_max;
```

## A.2 myEncoder Class

### A.2.1 myEncoder.h

```
#ifndef myEncoder_h
#define myEncoder_h

#include "Arduino.h"

class myEncoder
{
public:
    myEncoder(uint8_t PinA, uint8_t PinB);
    void updateEncoder();
    unsigned long read();
    unsigned long readAndReset();
    void write(unsigned long value);
    float speed();

private:
    static myEncoder* sEncoder;
    static void updateEncoderISR();
    volatile unsigned long pulses = 0;
    unsigned long time0;
};

myEncoder* myEncoder::sEncoder = 0;

myEncoder::myEncoder(uint8_t PinA, uint8_t PinB)
{
    pinMode(PinA, INPUT);
    pinMode(PinB, INPUT);
    time0 = micros();
    sEncoder = this;
    attachInterrupt(PinA, myEncoder::updateEncoderISR, RISING);
}

void myEncoder::updateEncoderISR()
{
    sEncoder->updateEncoder();
}

void myEncoder::updateEncoder()
{
    noInterrupts();
    pulses++;
    interrupts();
}

unsigned long myEncoder::read()
{
    noInterrupts();
    unsigned long pulses_copy = pulses;
    interrupts();
    return pulses_copy;
}

unsigned long myEncoder::readAndReset()
{
    noInterrupts();
    unsigned long pulses_copy = pulses;
    pulses = 0;
    interrupts();
    return pulses_copy;
}
```

```
void myEncoder::write(unsigned long value)
{
    noInterrupts();
    pulses = value;
    interrupts();
}

float myEncoder::speed()
{
    unsigned long dt = micros()-time0;
    // Serial.print(dt);
    // Serial.print("\t");
    unsigned long pulses = sEncoder->readAndReset();
    // Serial.print(pulses);
    // Serial.print("\t");
    float speed = pulses*60000000.0/(4*768.0*dt);
    time0 = micros();
    return speed;
}
#endif
```

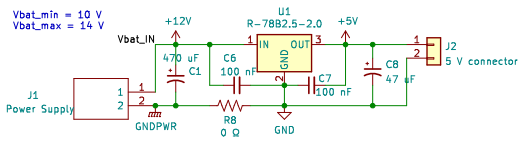
## A.3 Firmware

```
#include <Arduino.h>
#include "Robot.h"

Robot rc;
float* velocities = new float[3];

void setup() {}

void loop() {
  rc.read(velocities);
  rc.write();
  rc.cmd_vel(velocities[0], velocities[1], velocities[2]);
}
```

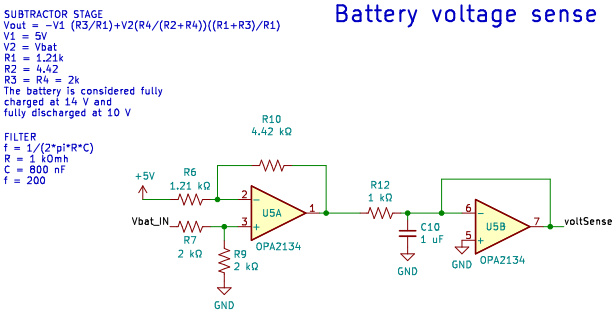
# Appendix B

# PCB schematic

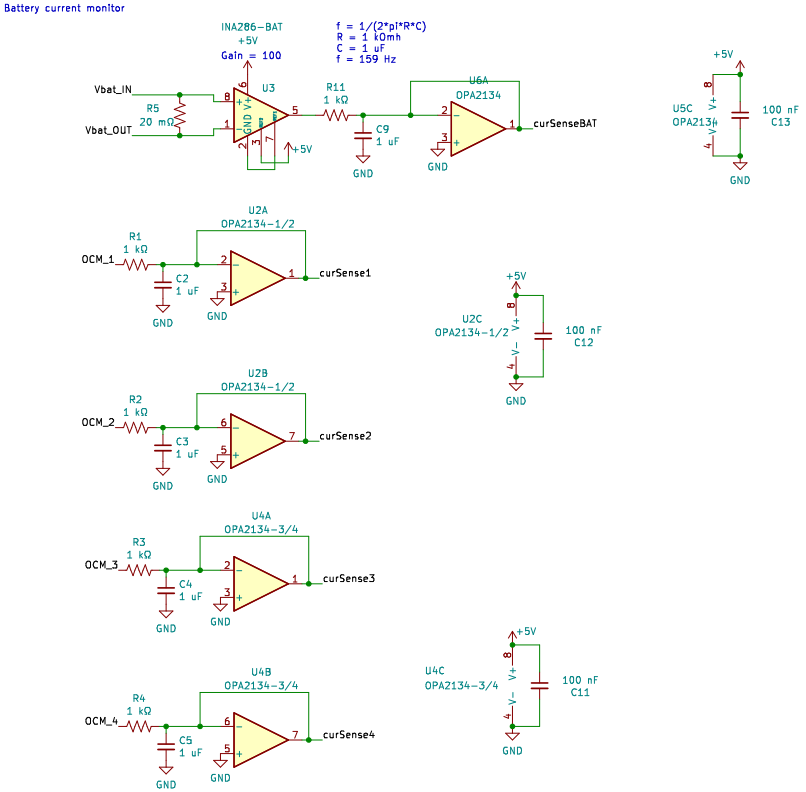
## Power supply



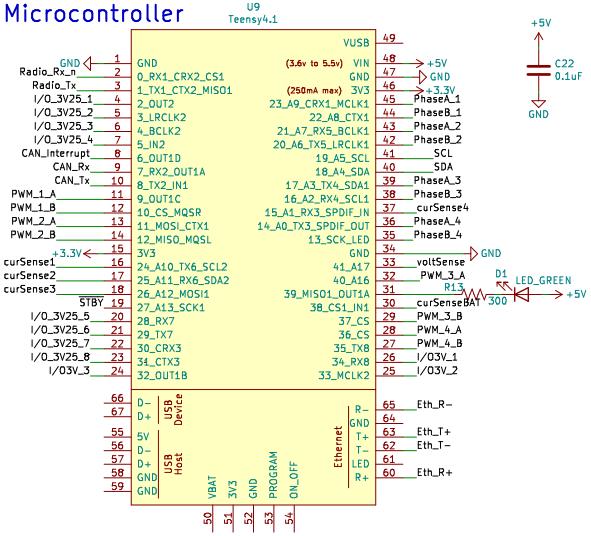

## Battery voltage sense



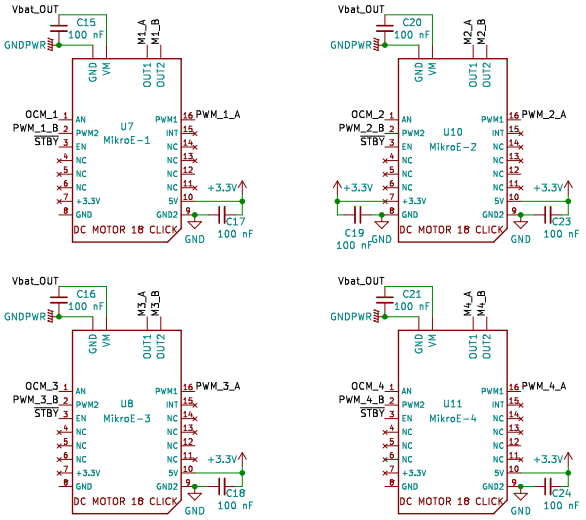

## Current monitor



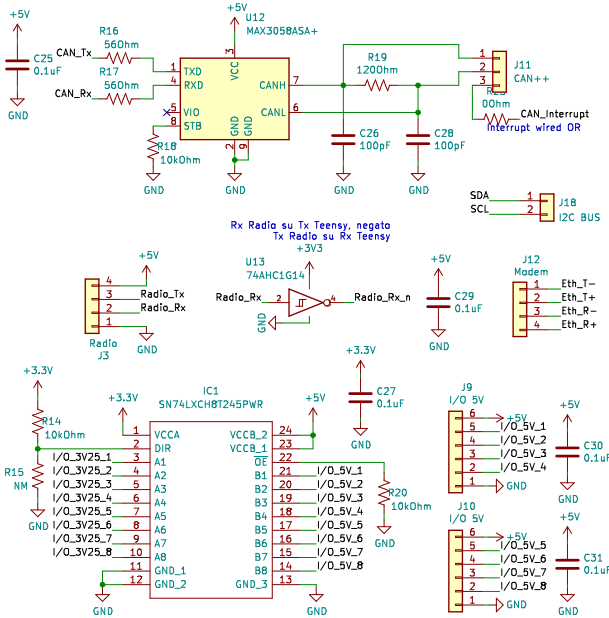

## Microcontroller



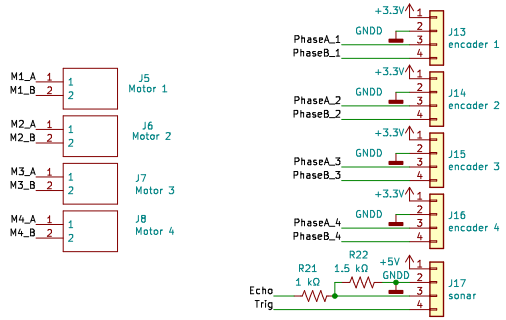

## Motor drver




## SENSORS CAN BUS




## Connectors



# List of Figures

# List of Tables

# Bibliography

[1] I. W. R. 2020. Ifr world robotics 2020, https://ifr.org/downloads/press2018/Presentation_WR_2020.pdf.

[2] Y. Alkendi, L. Seneviratne, and Y. Zweiri. State of the art in vision-based localization techniques for autonomous navigation systems. *IEEE Access*, 9:76847–76874, 2021.

[3] I. B. Erland. Rad fuer ein laufstabiles, selbstfahrendes fahrzeug. (DE2354404A1), May 1974.

[4] A. R. Khairuddin, M. S. Talib, and H. Haron. Review on simultaneous localization and mapping (slam). In *2015 IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 85–90, 2015.

[5] H. Taheri, B. Qiao, and N. Ghaeminezhad. Article: Kinematic model of a four mecanum wheeled mobile robot. *International Journal of Computer Applications*, 113(3):6–9, March 2015.