UAlg
UNIVERSIDADE DO ALGARVE

Software framework implementation for a robot that uses different development boards

Pedro Miguel Francisco Gomes

Dissertação

Mestrado Integrado em Engenharia Electrónica e Telecomunicações

Trabalho efectuado sob a orientação de:

Professor Doutor Helder Aniceto Sousa Daniel

2018

Software framework implementation for a robot that uses different development boards

Declaração de autoria de trabalho

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências incluída.

____

© Copyright - *Pedro Miguel Francisco Gomes, 2018*

A Universidade do Algarve tem o direito, perpétuo e sem limites geográficos, de arquivar e publicitar este trabalho através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, de o divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Acknowledgments

I would like to express my sincere gratitude to my adviser Prof. Helder Aniceto Sousa Daniel for giving me the opportunity to develop this project, for the useful comments and total engagement through the building process of this project.

I would also like to thank my parents for their efforts in providing me the best possible education and for supporting me through my academic experience.

Finally, I would like to thank the University of Algarve which provided the physical resources for this thesis to happen.

Abstract

A software framework is a concrete or conceptual platform where common code with generic functionality can be selectively specialized or overridden by developers or users. Frameworks take the form of libraries where a well-defined application program interface is reusable anywhere within the software under development. A user can extend the framework but not modify the code. The purpose of software framework is to simplify the development environment, allowing developers to dedicate their efforts to the project requirements, rather than dealing with the framework's mundane, repetitive functions and libraries.

A differential wheeled robot is a mobile robot whose movement is based on two separately driven wheels placed on either side of the robot body. It can thus change its direction by varying the relative rate of rotation of its wheels and hence does not require an additional steering motion. If both the wheels are driven in the same direction and speed, the robot will go in a straight line. If both wheels are turned with equal speed in opposite directions, the robot will rotate about the central point of the axis. Otherwise, depending on the speed of rotation and its direction, the center of rotation may fall anywhere on the line defined by the two contact points of the wheels.

The objective of this thesis, is to create a software framework for a wheeled robot, where we can change its development board, without having to make many changes in the code that is used to control the robot. The composition of the robot is: a development board, that allows the control of any electronic devices, that are connected to it; some sensors to detect obstacles; two motors to move the robot; a motor driver to power and control the motors individually; a chassis to assemble the robot; and a battery to power all the electronic devices.

The most important device of the robot is the development board, which allows the control of every single electronic device connected to it through a program. In this project we use three development boards, which are: *Arduino UNO rev3*, *NodeMCU ESP8266 v1.0* and *Raspberry Pi 3 Model B+*. The programming language used to control the devices and the boards is the C++ programming language, because it can be used with all of them. Since all the boards have a different external/internal design, there are some issues that we need to fix with the help of external hardware. Other important devices are the sensors to detect objects, which are: the sensor *HC-SR04*, which uses ultrasonic waves; and, the sensor *Sharp GP2Y0A41SK0F*,

which uses infrared light. The framework also covers two types of servo motors: one that can continuously rotate; and the other one that only rotates about half a circle. The servo motors can be used, for instance, to rotate a range sensor. Then we need two DC motors to move the vehicle. To power up these DC motors, which are controlled with a PWM signal, we need to connect them to a device called motor driver which is connected to a battery. Finally, to assemble the robot we just need to connect all the devices to the development board and attach them to the chassis.

This software framework was created with the purpose of programmatically connect every device (any sensor, motor or other device) to the development board and allow a user to do minimal code changes when he has the need to change the development board. All the devices that the framework supports have a datasheet explaining their behavior and operation, so that it was possible to develop a library to operate and control the device. By joining all these libraries together we have the framework presented here.

The experimental methodology, used in two case studies, will show the features and the limitations of the framework. The first case study shows that the changes the user needs to do when changing boards are minimal but because all the development boards are different, there are some things we can't program without having to make the user of the framework, sacrifice his GPIO connection choices. The second case study, shows that because of how the development boards work internally, there are some things that aren't possible to program to work like they were designed as the other development boards.

Keywords: software framework, wheeled robot, development board, sensors network.

Resumo

Um *framework de software* é uma plataforma conceitual em que um código comum com funcionalidade genérica pode ser seletivamente especializado ou substituído por desenvolvedores ou utilizadores. Os *frameworks* assumem a forma de bibliotecas em que uma interface de um programa bem definido é reutilizável em qualquer lugar dentro do software em desenvolvimento. Um utilizador pode estender a *framework*, mas não pode modificar o código. O objetivo do *framework* é simplificar o ambiente de desenvolvimento, permitindo que os desenvolvedores dediquem os seus esforços aos requisitos do projeto, em vez de lidar com as bibliotecas e funções, comuns e repetitivas, do *framework*.

Um robô com rodas diferenciais é um robô móvel cujo o movimento é baseado em duas rodas acionadas separadamente que estão colocadas em ambos os lados do corpo do robô. Pode assim alterar a sua direção, variando a taxa relativa à rotação das rodas, e portanto, não requer um movimento de direção adicional. Se ambas as rodas forem movidas na mesma direção e velocidade, o robô irá mover-se em linha reta. Se ambas as rodas girarem com velocidade igual e em direções opostas, o robô girará em torno do ponto central do eixo. Caso contrário, dependendo da velocidade de rotação e da sua direção, o centro de rotação pode cair em qualquer lugar na linha definida pelos dois pontos de contato das rodas.

O objetivo desta tese é criar um *framework de software* para um robô de rodas, onde podemos mudar a sua placa de desenvolvimento sem ter que fazer muitas alterações no código que é usado no controlo do robô. A composição do robô é a seguinte: uma placa de desenvolvimento, que permite o controlo de qualquer dispositivo eletrónico que esteja conectado à placa; alguns sensors para detetar obstáculos; dois motores para mover o robô; um *driver* para os motores, para alimentar e controlar os motores individualmente; um chassi, para montar o robô; e uma bateria, para alimentar todos os dispositivos eletrónicos.

O dispositivo mais importante do robô é a placa de desenvolvimento, que permite o controlo de cada dispositivo eletrónico, que está ligado à placa, através de um programa. Neste projeto utilizamos três placas de desenvolvimento, que são: *Arduino UNO rev3*, *NodeMCU ESP8266 v1.0* e *Raspberry Pi 3 Model B +*. A linguagem de programação usada para controlar os dispositivos e as placas é a linguagem de programação C ++, porque pode ser usada por todas as placas. Como todas as placas têm um design externo / interno diferente, existem alguns proble-

mas que precisam de ser salvaguardados com a ajuda de *hardware* externo. Outros dispositivos importantes são os sensores que servem para detetar objetos, e estes são: o sensor *HC-SR04*, que utiliza ondas ultrassónicas; e, o sensor *Sharp GP2Y0A41SK0F*, que usa luz infravermelha. A *framework* também suporta dois tipos de servo motores: um que pode girar continuamente; e o outro só pode rodar cerca de meio círculo. Por exemplo, os servo motores podem ser utilizados para fazer rodar um sensor que deteta objetos. Também precisamos de dois motores DC para mover o robot. Para energizar estes motores, que são controlados por um sinal PWM, precisamos de liga-los a um dispositivo chamado de *driver* para motor que está ligado a uma bateria. Finalmente, para montar o robô precisamos apenas de ligar todos os dispositivos à placa de desenvolvimento e montar todos os dispositivos no chassi.

Este *framework de software* foi criada com o objetivo de ligar todos os dispositivos (qualquer sensor, motor ou outro dispositivo eletrónico), em termos de programação, à placa de desenvolvimento e permitir que um utilizador faça alterações mínimas no código que controla robô quando tiver a necessidade de alterar a placa de desenvolvimento. Todos os dispositivos que a *framework* suporta têm uma ficha de especificações que explica o seu comportamento e como funcionam, e por este motivo foi possível desenvolver uma biblioteca para controlar estes dispositivos. Ao unir todas as bibliotecas temos a *framework*.

A metodologia experimental usada em dois casos de estudo, mostram quais as limitações da *framework*. O primeiro caso de estudo mostra que as alterações que o utilizador precisa de fazer ao trocar as placas são mínimas, mas como todas as placas de desenvolvimento são diferentes existem sempre algumas coisas que não podemos programar sem ter que fazer com que o utilizador da *framework*, sacrifique as suas opções de ligação aos pinos GPIO. No segundo caso de estudo é mostrado que devido à forma de como as placas de desenvolvimento trabalham internamente, existem algumas coisas que não podem ser programadas para funcionar como se fossem as outras placas de desenvolvimento.

Keywords: *framework de software*, robô de rodas, placa de desenvolvimento, rede de sensores.

Index

Acknowledgments . . . . **II**

Abstract . . . . **III**

Resumo . . . . **V**

Index . . . . **VII**

List of Figures . . . . **IX**

List of Tables . . . . **XI**

List of Abbreviations . . . . **XII**

1 Introduction **1**

1.1 Thesis organization 2

2 Development boards **3**

2.1 *Arduino UNO Rev3* 3

2.1.1 Why use *Arduino*? Features and drawbacks. 5

2.1.2 Programming 6

2.2 *NodeMCU ESP8266 v1.0* 8

2.2.1 Why use *NodeMCU ESP8266*? Features and drawbacks. 11

2.2.2 Programming 12

2.3 *Raspberry Pi 3 B+* 14

2.3.1 Why use *Raspberry Pi 3 B+*? Features and drawbacks. 16

2.3.2 Programming 17

2.4 Comparing the three dev boards 19

2.5 Fixing GPIO related problems 20

2.5.1 Using an Analog to Digital Converter (ADC) 20

2.5.2 Using a Multiplexer 23

2.5.3 Using a Logic Level Converter 24

3 Hardware - Sensors and Actuators **26**

3.1 Sensors 27

3.1.1 Ultrasonic sensor *HC-SR04* 27
3.1.2 Analog Infrared Sensor *Sharp GP2Y0A02YK0F* 30
3.2 Actuators 35
3.2.1 Servo Motors Fundamentals 36
3.2.2 Controlling the servos 37
3.3 DC Motors Fundamentals 38
3.4 Controlling the motors 40

4 Software Framework 47
4.1 Understanding the framework 48
4.2 Programming the framework 52
4.2.1 Programming the multiplexer 54
4.2.2 Programming the ADC via SPI Serial Interface 55
4.2.3 Programming the *Ultrasonic Sensor HC-SR04* 59
4.2.4 Programming the *Analog Infrared Sensor Sharp GP2Y0A02YK0F* . . . 60
4.2.5 Programming the Motor Drivers 62
4.2.6 Using the Servo Motors 65

5 Case studies 66
5.1 Case study 1: Automatic path-finding while avoiding obstacles 74
5.2 Case study 2: Robot control over a WiFi connection 76

6 Conclusions and Future Work 80
6.1 Conclusions 80
6.2 Future Work 81

References 82

Appendix 84

List of Figures

2.1 Design of the *Arduino UNO Rev3* board. (arduino.cc, 2018b) 3
2.2 Selecting the *Arduino/Genuino UNO* board in the *Arduino Software IDE*. 7
2.3 Selecting the port that corresponds to the *Arduino UNO* board. 7
2.4 Basic sketch file for *Arduino Software IDE.* 8
2.5 Design of the *NodeMCU v1.0* development kit board. 9
2.6 *NodeMCU v1.0* pin configuration. 9
2.7 Adding *NodeMCU* libraries in *Arduino Software*. 13
2.8 Installing *NodeMCU* libraries. 13
2.9 Design of the *Raspberry Pi 3 B+*. Top view.(raspberrypi.org, 2018b) 14
2.10 Using NOOBS to install the Raspbian distribution.(raspberrypi.org, 2018a) . . . 17
2.11 ADC MCP3008 pin configuration. 23
2.12 Multiplexer CD4051B pin configuration (TexasInstruments, 2017). 24
2.13 Logic Level Converter 25
3.1 Chassis design *Devastator Tank Mobile Robot Platform* (DFRobot, 2018). . . . 27
3.2 Ultrasonic sensor *HC-SR04* design and pins (CytronTechnologies, 2013). . . . 28
3.3 Ultrasonic sensor signals diagram. 29
3.4 Infrared sensor *Sharp GP2Y0A02YK0F*. 31
3.5 Sensor *Sharp GP2Y0A02YK0F* plot of an analog voltage output as a function of the inverse, of the distance to a reflective object. 32
3.6 Sensor *Sharp GP2Y0A02YK0F* plot, with linearized equation in red. 34
3.7 Design of both actuators. 35
3.8 Inside the servo motor. 36
3.9 PWM controls servo position (Jameco, 2018). 38
3.10 Inside the DC motor (ISLProductsInternational, 2018a). 39
3.11 Micro DC Geared Motor with Back Shaft. 40
3.12 PWM signals with different duty cycle waveforms. T_{ON} represents the time the signal is in high state (ON) and T_{OFF} represents the time the signal is in low state (OFF). Period represents a complete cycle (ElectronicWings, 2017). . . . 41
3.13 H-Bridge circuit diagram with a motor at the center. 42
3.14 Top view of *2A Motor Shield For Arduino*. 43
3.15 Top view of *Itead L298 Dual H-Bridge Motor Driver*. 43

3.16 Pin description of *2A Motor Shield For Arduino*. 45
3.17 Pin view of *Itead L298 Dual H-Bridge Motor Driver*. 46
5.1 Robot connection diagram using *Arduino UNO* board. 68
5.2 Robot connection diagram using *NodeMCU ESP8266* board, with the addition of the logic level converter. 69
5.3 Robot connection diagram using *Raspberry Pi 3 Model B+* board, with the addition of the logic level converter. 70
5.4 Fully assembled robot in the chassis with all devices connected to any development board. 71
5.5 Front view of the assembled robot. 72
5.6 Bottom view of the robot where the battery is. 73
5.7 Main screen of the android application that controls the robot. 77
5.8 Android application screens after selecting a mode. 78

List of Tables

2.1 Comparison of the most important specifications between the three dev boards. 19

List of Abbreviations

AC	Alternate Current
CPU	Central processing unit
DC	Direct Current
dev	Development
GPIO	General Purpose Input/Output
I/O	Input/Output
ICSP	In-Circuit Serial Progamming
IDE	Integrated development environment
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IR	Infrared
lib	Library
MCU	Microcontroller Unit
mux	Multiplexer
OS	Operating System
PCB	Printed circuit board
PWM	Pulse Width Modulation
RAM	Random Access Memory
Rev	Revision
RPi	Raspberry Pi
SD	Secure Digital

SoC	System on a chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
USB	Universal Serial Bus
V	Volts

1 Introduction

A software framework is a platform used in application development, which uses a specific programming language with the goal of unifying multiple programming codes created by developers or users, with in mind the easy use of such applications. This framework was developed in C++ language, because it's a common language used between all development boards. In case of this thesis, the software framework was created to facilitate the use of some electronic components which are being used together with three different development boards, which are: *Arduino UNO Rev3*, *NodeMCU ESP8266* and *Raspberry Pi 3 B+*. Therefore, if we connect some of these components to one of the development boards and attach them to a chassis, then it can be used as a wheeled robot.

Wheeled robots are robots that use motorized wheels to navigate around the ground. These robots are very popular among the consumer market, because of its low cost and simplicity. But, because of this simplicity there are many disadvantages related to their mobility, such as having trouble to navigate well over complex obstacles like, areas with low friction, rocky terrain or sharp declines. In this thesis, to be able to program these robots I'm going to use development boards as its core.

Development boards are printed circuit boards containing circuits and hardware designed to ease experimentation with certain microcontrollers. These boards were created to anyone with an interest in creating environments or interactive objects. They are mostly used for education purposes or for prototyping projects. Typically, dev boards are composed of:

- Power circuit, used to power the board and all attached components.
- Programming interface, to program the microcontroller using a computer.
- General Purpose Input/Output, which are digital signal programmable pins used to communicate with external electronic components.

Usually these dev boards can't do a whole lot in their own (there are some that are still considered dev boards but they act more like mini-computers, meaning they run an operating system like a home computer), so we need to hook them up with something.These somethings I'm talking about are electronic components such as:

- Connectivity - using wireless connections like, Wi-Fi, Bluetooth or even 3G or 4G cellular standard.
- Sensors - examples: detect airflow, force, humidity, light, pressure, proximity, temperature, radioactivity, etc.If you can sense it, you can measure it.
- Shields - are modular circuit boards that add extra functionalities to your board. Usually they are attachable to your board.

1.1 Thesis organization

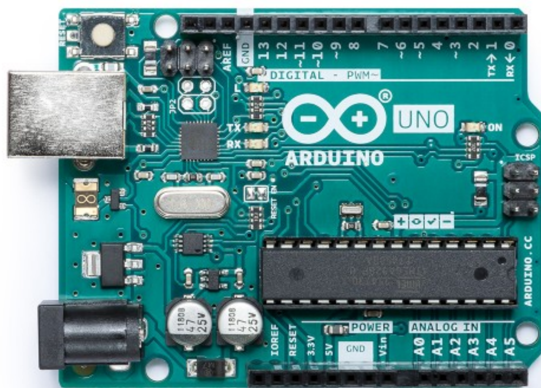
The structure of this thesis is organized into 5 chapters. The first chapter briefs the reader on what will be the focus of the thesis. Chapter 2 describes which dev boards are going to be used in this project, how they can be programmed, its problems and how to fix them, if possible. Chapter 3 explains the fundamentals of how some of the devices work. In chapter 4 the focus is to explain how the software framework works as a whole and show parts of the code on how the devices presented in chapter 3 work in practice. Chapter 5 presents two case studies using the same wheeled robot, which include: one of the development boards; two DC motors; one DC motor drive; three ultrasonic sensors and a chassis (metal plates and wheels, in this case are continuous tracks). Finally, in chapter 6 conclusions and future work are presented and discussed.

2 Development boards

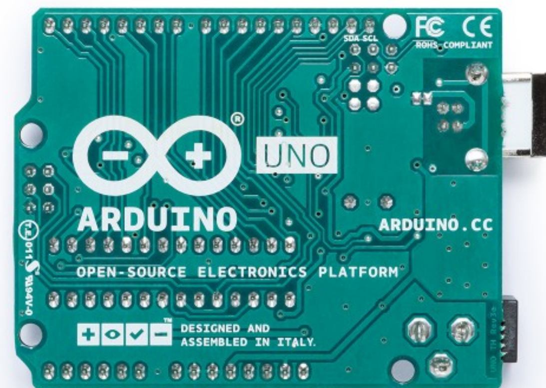
Development boards are printed circuit boards that have circuits and hardware designed to work with some microcontrollers. A microcontroller is a small computer on a single integrated circuit, it's like a basic computer but much simpler and consumes less power. The integrated circuit consists on the same basic computer components such as: CPU, RAM, flash storage and hardware peripherals, such as GPIO pins to connect sensors or other devices. The majority of microcontrollers today are embedded inside of some systems, often a consumer product such as automobiles, telephones, etc.. Usually they are dedicated to a single task and run a specific program. They are mostly used when we want to add intelligence to a system. In this project, we are going to use three different dev boards to control a robot. The dev boards are: *Arduino UNO Rev3*, *NodeMCU ESP8266* and *Raspberry Pi 3 Model B+*.

2.1 *Arduino UNO Rev3*

Arduino UNO is a development board based on ATmega328P microcontroller. It contains everything needed to support the microcontroller. The board can be powered using the USB connection or with an external power (the power source will be selected automatically). The revision 3 differs from all preceding boards because it uses the ATmega16U2 programmed as a USB-to-serial converter, while in other boards they used FTDI USB-to-serial converter. (arduino.cc, 2018b)



(a) Front panel



(b) Back panel. We can see all the circuit lines.

Figure 2.1: Design of the *Arduino UNO Rev3* board. (arduino.cc, 2018b)

Characteristics:

- Digital I/O pins: 14 , of which 6 provide PWM output.
- Analog pins: 6 with 1024 step resolution (10-bit ADC).
- PWM Output: 6 pins.
- 16 MHz quartz crystal.
- USB connection to upload the code and power the device.
- Power jack, to be used with an external power.
- ISCP header, is a protocol to program ATmega328P microcontroller, if needed.
- Reset button, to restart the program from the beginning.

Important specifications:

- Recommend power supply of 7V to 12V, with it's limits being 6V to 20V.
- Circuit operating voltage of 3.3V or 5V, it uses a voltage regulator to limit the power supplied.
- DC current per I/O pin of 20mA to 40mA max.
- Clock speed of 16 MHz, supplied by the quartz crystal.
- Flash Memory of 32kB (ATmega328P) of which 0.5kB are used by the bootloader
- SRAM of 2kB (ATmega328P), it's like the RAM of a computer. It stores values while the program is running.
- EEPROM of 1kB (ATmega328P), it's an older technology that implements re-writable non-volatile memory. It's usually used to store settings and other parameters between resets.

There are many other microcontrollers and microcontroller platforms available for physical computing

2.1.1 Why use *Arduino*? Features and drawbacks.

Thanks to its simple and accessible user experience, *Arduino* has been used in thousands of different projects and applications. The *Arduino Software* is easy-to-use for beginners, yet flexible enough for advanced users (arduino.cc, 2018d). There are many other microcontrollers and microcontroller platforms available, but many of those make microcontroller programming look to messy. *Arduino* wraps it all up in an easy-to-use package , both hardware and software, and simplifies the process of working with microcontrollers by offering some advantages:

- **Open source software and hardware** - *Arduino Software* is open source which creates a way for experienced programmers to create their own extensions. The programming language can be expanded using C++ language but the developer can also use AVR C programming language (AVR is a family of microcontrollers developed by Atmel that uses C programming language to program those type of microcontrollers). The hardware board is also open source meaning experienced designers can make their own versions or improve the *Arduino* board.
- **It has a huge community** - since *Arduino* has been here for too long, it has been used in thousands of different projects and applications. Because of this, we can easily find a lot of libraries that can help in a lot of projects, simplifying the process of coding.
- **Inexpensive boards and peripherals** - because *Arduino* boards are open source, there are a lot of boards similar to *Arduino UNO* that are cheaper than the ones sold by *Arduino* company. Most of the peripherals are very cheap, because they are manufactured in very large scales and the components used to make them are very cheap.
- **Cross-platform** - the *Arduino Software* runs on Linux, Macintosh OSX and Windows, while most of others systems are limited to run on Windows.
- **Easy to use** - The *Arduino Software* (IDE) is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well (arduino.cc, 2018d).
- **It has enough GPIO pins** - which makes it easier for some projects.

Even though *Arduino UNO* board has a lot of advantages over its competitors, there are still some drawbacks:

- **Size** - even though the board seems robust, it can be a drawback if someone needs to build a project and wants to make it as small as possible.
- **Programming language** - we are limited to use only C/C++ as the programming language, which is a bit old and is not as reliable when compared with some of the “new” languages. There are other development boards where you are free to choose the language you want to code with..
- **No embedded WiFi adapter**- if a project needs to communicate via wireless we need to buy external peripherals to make it possible, and is slightly difficult to configure it if we don’t find an already created library. Recently, an *Arduino UNO* board with an integrated *ESP8266* was made available (Arduino, 2018a). Essentially it can be handled the same way as an *Arduino UNO* board with an external *ESP8266* WiFi module connected via serial interface.
- **No Ethernet adapter** - the same problem with WiFi but with a cable. If we need to connect a project to a network or even the internet, we need to buy external peripherals or a shield.

2.1.2 Programming

Arduino is an open-source electronics platform based on easy-to-use hardware and software. *Arduino* boards are able to read inputs and turn it into an output. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. The ATmega328P on the *Arduino UNO* comes preprogrammed with a bootloader that allows the user to upload new code to the *Arduino*, without the use of an external hardware programmer(arduino.cc, 2018d). To do this we need to use the *Arduino* programming language, which is known as sketches and is written in C++, and the *Arduino Software* (IDE).

The *Arduino Software* can be downloaded from the *Arduino* download page. After we download it and install it, we can program the *Arduino* using the sketches. To upload code to the *Arduino* we need to select the following options “Tools > Board > Arduino/Genuino Uno” and by selecting the serial port in “Tools > Port > Port related to the connected board”. We can see a better explanation in the next figures.

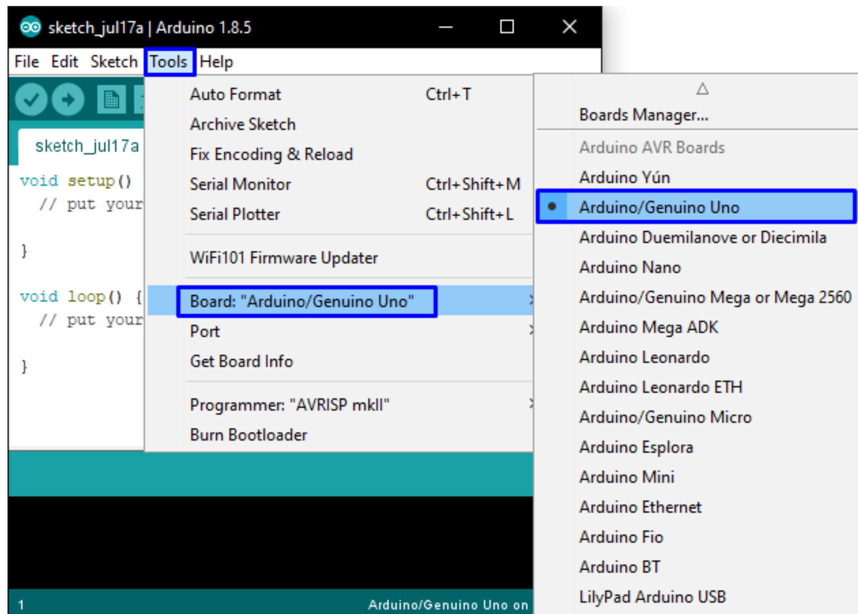


Figure 2.2: Selecting the *Arduino/Genuino UNO* board in the *Arduino Software IDE*.

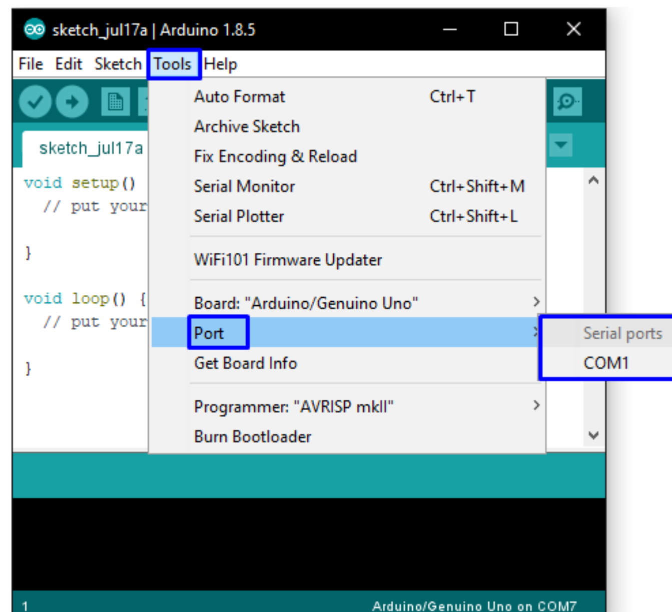


Figure 2.3: Selecting the port that corresponds to the *Arduino UNO* board.

Every sketch, which is a C++ program using the basic *Arduino* framework, needs two void type functions, *setup()* and *loop()*. The *setup()* method will be ran once after the *Arduino* is powered

up and the *loop()* method will ran continuously after the *setup()* method is finished. The *setup()* is where you want to do any initialization steps and in the *loop()* where you want to run the code over and over again.

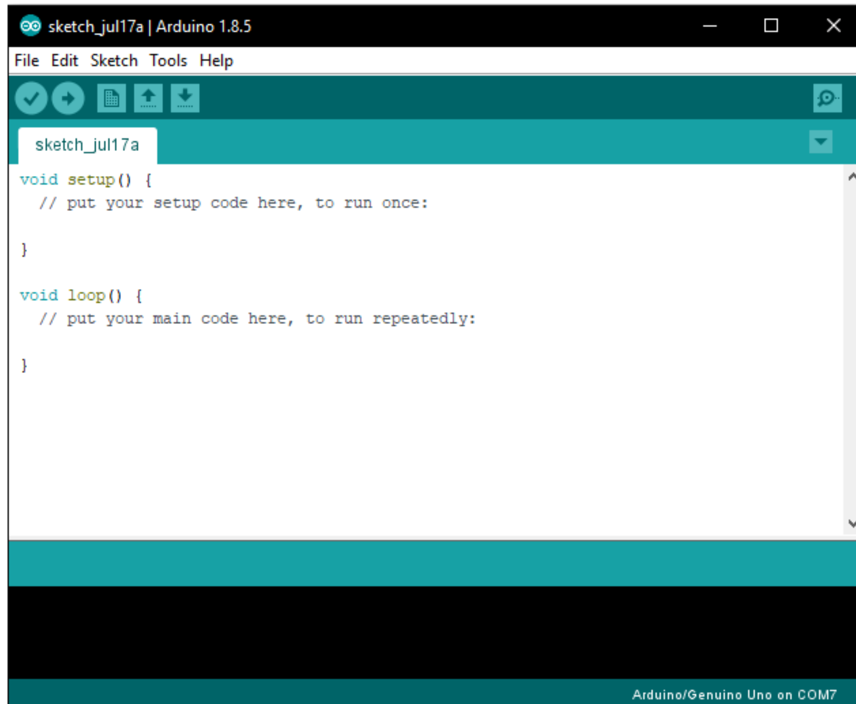


Figure 2.4: Basic sketch file for *Arduino Software IDE.*

The basic *Arduino* framework have already a *main()* function which calls these two functions. This will be further addressed in chapter 4, but the simplified code is:

```
1 main () {
2         setup();
3         for (;;) loop()
4 }
```

2.2 *NodeMCU ESP8266 v1.0*

One of the more more recent development boards is the *NodeMCU ESP8266 v1.0* which is an open source IoT platform. The *NodeMCU* is an open source software and hardware develop-

ment environment that is built around a very cheap system-on-a-chip (A SoC is an integrated circuit that integrates the components of a computer or other electronic systems. A SoC integrates a microcontroller with other advanced peripherals) called ESP8266. *NodeMCU* is an eLua (offers the full implementation of the Lua programming language to the embedded world) based firmware for the *ESP8266* SoC. This development kit is based on the *ESP8266 ESP-12* module, which uses the ESP8266 has its core, which contains a microcontroller and a WiFi adapter. (Espressif, 2018)

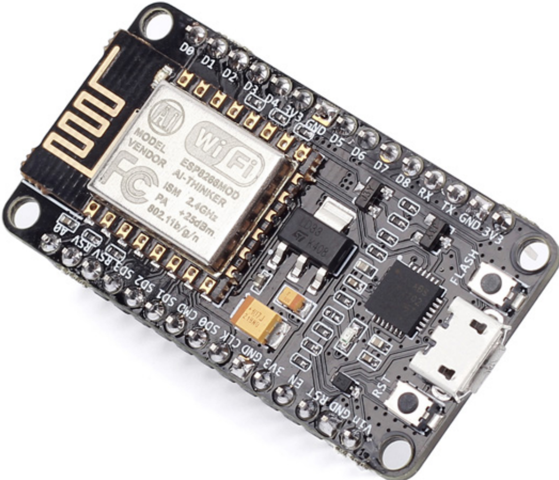


Figure 2.5: Design of the *NodeMCU v1.0* development kit board.

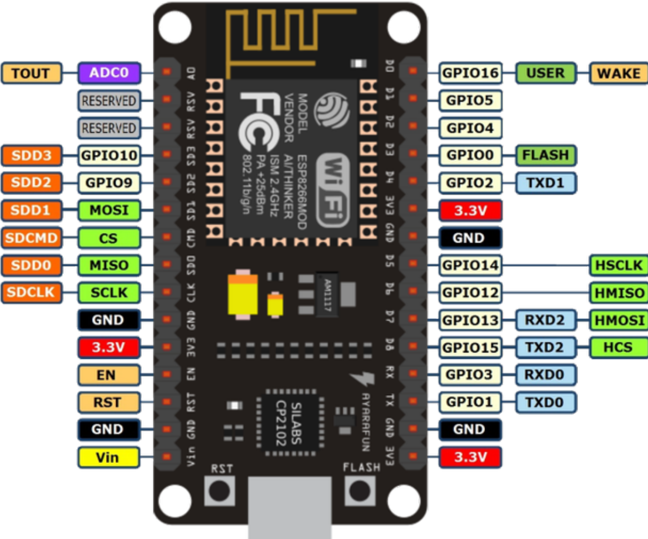


Figure 2.6: *NodeMCU v1.0* pin configuration.

Characteristics:

- Digital I/O pins: 11 (multiplexed with other functions), but in some boards GPIO 9 doesn't work.
- Analog pins: 1 with 1024 step resolution (10-bit ADC).
- PWM Output: 10 pins.
- Micro USB connection to upload the code and power the device.
- Reset button, to restart the program from the beginning.
- Flash button, to get it into upload mode. But this board have a USB to serial adapter onboard which does it automatically.
- IEEE 802.11 b/g/n protocol (WiFi).
- Integrated TCP/IP protocol stack, basically makes the WiFi chip be able to be used.

Important specifications:

- Recommend power supply of 3.3V to 5V, with it's limits being 3.3V to 10V.
- Circuit operating voltage of 3.3V, it uses a voltage regulator to limit the power supplied.
- DC current per I/O pin of 1000mA max.
- Clock speed of 80 MHz to 160 MHz.
- Flash Memory of 4MB
- SRAM of 64kB.

2.2.1 Why use *NodeMCU ESP8266*? Features and drawbacks.

As a chip, the ESP8266 is very hard to access and use. You have to solder wires, with the appropriate analog voltage, to its pins for the simplest tasks such as powering it on or sending a keystroke to the "computer" on the chip. And, you have to program it in low-level machine instructions that can be interpreted by the chip hardware (Yuan, 2017). This problem disappears when the ESP8266 is used as an embedded chip in systems such as the *NodeMCU*, making it a lot easier to program/use.

Features:

- **Open source firmware and hardware** - the ESP8266 firmware is built and distributed by the chip manufacturer's, which makes it possible for the developers to tailor their own firmware. Provides simple programming based on eLua, which is very simple and accessible for it's users to experience because of it's established developer community. All the schemes of the device are available online for anyone that wants to use it.
- **Established community** - isn't as big as the *Arduino* community because it's much more newer, but it's sill much better when compared to others.
- **Multi programming language** - even though the ESP8266 was made to be programmed in eLua, developers of this platform (*NodeMCU ESP8266*) made it possible to also be programmed in MicroPython (an efficient implementation of the Python 3 Programming Language that includes a small subset of the Python standard library and is optimized to run on microcontrollers) and in the *Arduino* environment.
- **Easy to use** - all programming platforms make this device easy to learn and use.
- **Cross-platform** - you can run any platform on these three operating systems: Linux, Macintosh OSX and Windows.
- **Embedded WiFi adapter** - this dev board was designed to be used by the IoT platform and because of this it needs to be able to connect to the Internet.
- **Small size** - since this dev board is very small when compared to others, it makes it possible to be industrialized in a lot of products.

- **Very cheap** - you can buy one for less than 10€, which is way cheaper than most of dev boards out there.

Although the *NodeMCU ESP8266* it's powerful, it's limitations can slow down your project. This limitations are all related to the GPIO pins:

- **Limited GPIO pins** - while this dev board has 17 GPIO pins, you can only access 11 of them and 1 of them (pin SD2 which is GPIO 9) is still unreliable because it only works on some boards. Also pin D0 (GPIO 16) can't use PWM, and pin SDD3 (GPIO 10) can only be used as output.
- **Multiplexed pin functions** - the 11 pins that we can access are multiplexed or share other functions. For instance:
  - If you want to use a SPI connection, on pins D5 to D8, to communicate with a peripheral, you can't use those pins as GPIO.
  - When you want to upload code to the board pins D0, D3, D4, D7, D8, RX and TX can't be used, because they either need to be accessed by the flash memory and/or they need to be used by the USB interface to upload the code.
  - When booting the board, pin D8 can't be connected because ESP8266 needs to access this pin to properly boot. This problem can be resolved by using a low ohm pull down resistor on that pin (the board supposedly has a pull down resistor for this but apparently is not good enough).
- **Limited ADC pins** - basically we only have one pin that receives analog input.

2.2.2 Programming

Has we've seen in the features of the sub-section 2.2.1 the *NodeMCU ESP8266* can be programmed in multiple languages, but to achieve the purpose of this thesis this board will be programmed using the *Arduino Software* environment. To be able to use the *NodeMCU* with the *Arduino Software*, we need follow the following steps.

Go to “ File > Preferences”, or use the shortcut to access it “CTRL + ,” and copy the link “http://arduino.esp8266.com/stable/package_esp8266com_index.json” in the field “Additional Boards Manager URL”.

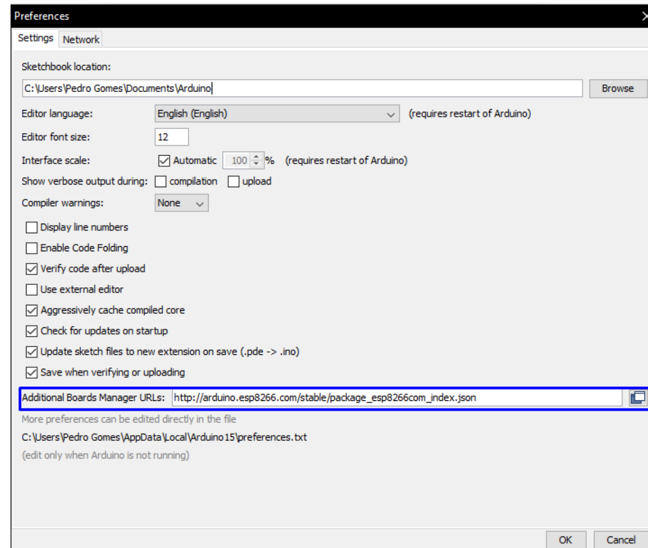
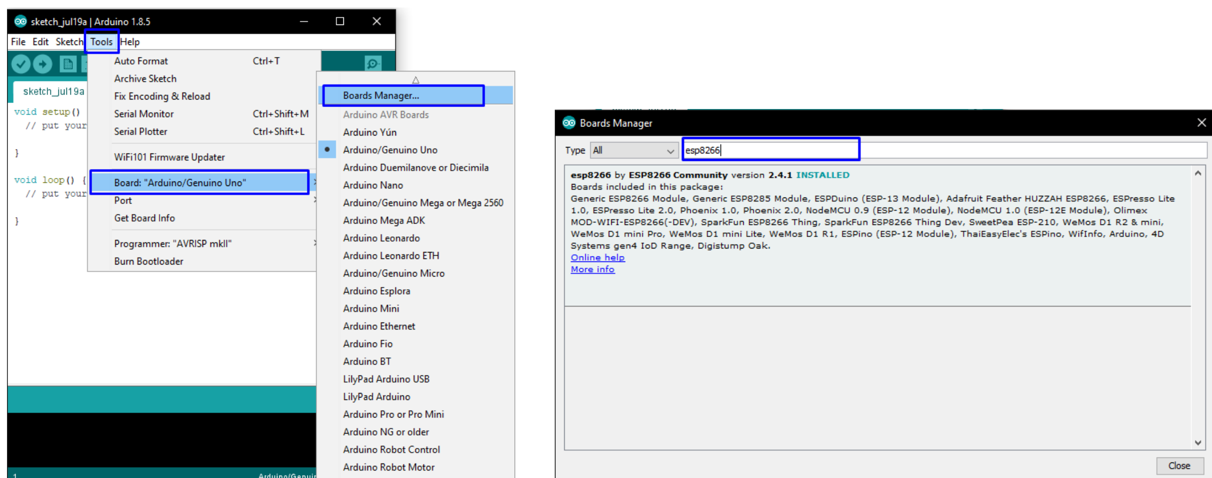


Figure 2.7: Adding *NodeMCU* libraries in *Arduino Software*.

After adding those libraries, we need to install them. For that we need to go to “ Tools > Boards Manager...” and search for “esp8266 by ESP8266 Community”.



(a) Selecting “Boards Manager...” option.

(b) Searching for *NodeMCU ESP8266* libraries.

Figure 2.8: Installing *NodeMCU* libraries.

Once all the steps above are completed, we can program the board with sketches as if we where using an *Arduino*.

2.3 *Raspberry Pi 3 B+*

A *Raspberry Pi* is a credit card-sized computer originally designed for education, inspired by the 1981 BBC Micro. Creator Eben Upton's goal was to create a low-cost device that would improve programming skills and hardware understanding at the pre-university level. But thanks to its small size and accessible price, it was quickly adopted by tinkerers, makers, and electronics enthusiasts for projects that require more than a basic microcontroller. The *Raspberry Pi* is slower than a modern laptop or desktop but is still a complete Linux computer and can provide all the expected abilities that implies, at a low-power consumption level. It was designed for the Linux operating system, and many Linux distributions now have a version optimized for the *Raspberry Pi* (opensource.com, 2018). There are some versions and variations of the *RPi* board, but *RPi 3 B+* has the latest updates, such as better CPU, more RAM, more USB ports, more GPIO pins, etc.. (raspberrypi.org, 2018b)

Figure 2.9: Design of the *Raspberry Pi 3 B+*. Top view.(raspberrypi.org, 2018b)

Characteristics:

- Digital I/O pins: 40.
- Analog pins: 0.
- PWM Output: all 40 pins can be used as PWM but they need to be simulated through some programming language.
- 4 USB 2.0 ports.
- Micro USB connection to power the device.
- Micro SD port for loading your operating system and storing data.
- IEEE 802.11 b/g/n/ac protocol (WiFi).
- Gigabit Ethernet over USB 2.0 adapter.
- Bluetooth 4.2, Bluetooth Low Energy.
- 1 full size HDMI port.

Important specifications:

- Power supply of fixed +-5V, ranging between 4.75V to 5.25V.
- Power supply needs a current of at least 1mA and can use a max of 2500mA.
- Circuit operating voltage of 3.3V. If using 5V peripherals we need a logic level converter.
- DC current per I/O pin of 8mA max, with a 50mA max for all pins.
- Broadcom BCM2837B0, Cortex-A53 64-bit SoC which includes:
  - CPU: 1.2GHz to 1.4GHz Quad-Core ARM Cortex-A53
  - GPU: Broadcom Videocore-IV
  - SDRAM: 1GB LPDDR2
- Flash Memory is the size of the micro SD card, with recommended card size of 8GB.

2.3.1 Why use *Raspberry Pi 3 B+*? Features and drawbacks.

The *RPi 3B+* works like a small computer, so it has more processing power than most micro-controllers but on the other hand it has more power consumption (we either need to constantly power supply it or use very large batteries, like lithium batteries), when compared with the likes of *Arduino UNO* or *NodeMCU ESP8266*. Even though it wastes more power than those dev boards that use microcontrollers, it's still not that much for the electricity bill at the end of the month, which makes it a perfect solution for an "always on" device (meaning you can turn it on, set it up and leave it running somewhere without worrying too much about it).

Features:

- **Open source software and hardware** - since it runs on Linux, which is an open source operating system, it has a lot of open source software that the users can use. It's hardware is open source expect for the SoC which is copyrighted by Broadcom (chip's manufacture), meaning you can do your own board but either use Broadcom SoC or another one.
- **Huge community** - like the *Arduino* the *RPi* also have a very large community, which makes it easier for users to find help or other tools if needed.
- **Multi programming language** - since *RPi* is Linux based you can basically program in any language you want to, but to make things easy it's better to choose a language where you can find libraries which already have GPIO support.
- **Embedded Bluetooth, Ethernet and WiFi adapter.**
- **Large range of GPIO pins.**

When compared to the other two boards the advantages that *RPi* brings to the table are way better, but on the other hand it has a lot of disadvantages, such as:

- **Size** - the *RPi* is still a bit big if you want to try to conceal its presence in a project. So it really depends on it's usage.
- **Expensive** - price range of the *RPi* is 40€-60€, depending where we get it, and we still need to buy an USB power supply, keyboard, mouse, etc..

- **Not so easy to use** - since the *RPi* uses an OS, it is recommend that the users learn how to use it's OS before starting to program on it.
- **No ADC** - if we need to use any sensors that need an analog input we need to buy an external ADC to do it.
- **Processing power** - when it comes to the GPIO programming the *RPi* processing in most cases will be an overkill, because will use more processing power than what it needs.
- **Power consumption** - *RPi* is really power hungry because of it's processing power, so we most likely need to have it always connected to a power adapter.

2.3.2 Programming

Has we saw in the section 2.3.1 to start working with *RPi* is not as easy as using the other two dev board boards. Since *RPi* works like a computer, first we need to install the OS in the micro SD card. To do so, follow the following guide, where you can check the link in the references (raspberrypi.org, 2018a). To help with the installation I'm going to use the OS installation manager "New Out Of Box Software", also known as "NOOBS", and I'm choosing to install the Raspbian distribution which is a port of the Debian, optimized for the *RPi*.

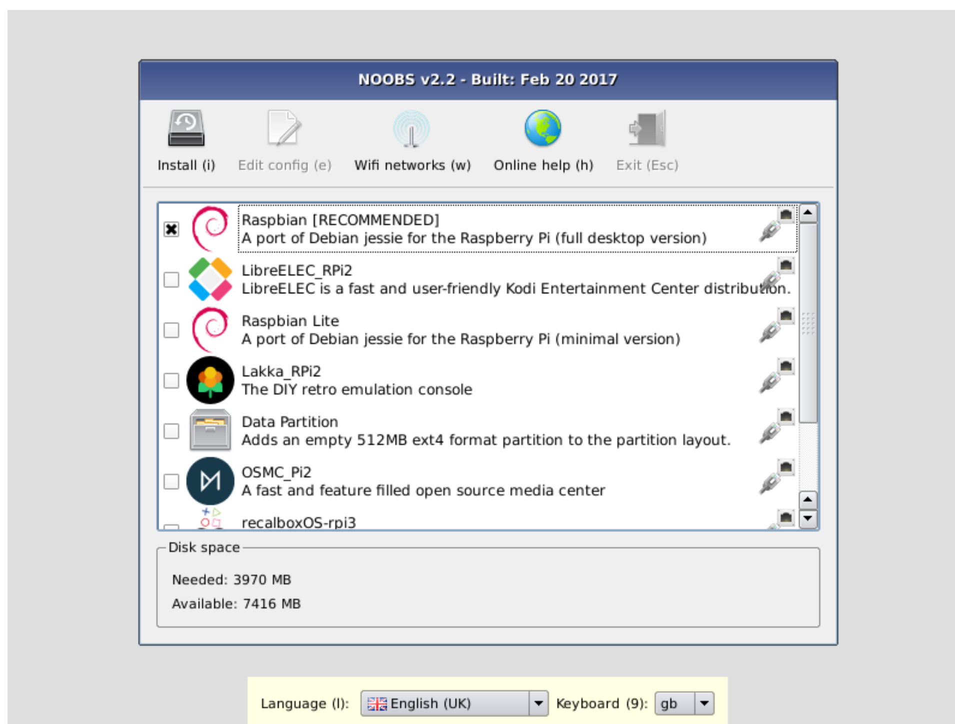


Figure 2.10: Using NOOBS to install the Raspbian distribution.(raspberrypi.org, 2018a)

Since *Arduino* and *NodeMCU* are going to be programmed using the *Arduino Software* and it's programming language is based on C++, in *RPi* I'm also going to program with the C++ programming language. To do so, we need to have the compiler (a program that converts instructions into a machine-code or lower-level form so that they can be read and executed by the computer) related to the C++ language, which is called "gcc". Raspbian distribution already comes with it pre-installed. If you were wondering when we install the *Arduino Software* it already comes with "gcc" pre-installed.

To run the code in the *RPi*, we need to create our own programs using any text editor and compile it using "g++ <filename.cpp> -o <output-file>" in the command line ("gcc" compiles in C language while "g++" compiles C++ language). After that we need to run the "<output-file>", by doing "./<output-file>" in the command line.

If you want to see more information on "g++" program you can run "man g++" in the command line, or search "g++ manual" in your browser. Unlike the *Arduino Software* which you can upload the code to your boards by pressing one button, in Linux distributions you need to do more steps to accomplish the same thing, but when you get used to it it's not a big deal.

2.4 Comparing the three dev boards

	Arduino UNO rev3	*NodeMCU ESP8266 v1.0*	*Raspberry Pi 3 B+*
MCU / SoC	ATmega328P 8-bit	ESP8266 32-bit SoC	Broadcom BCM2837B0 Cortex-A53 64-bit SoC
Clock Speed	16/20MHz	80/160MHz	1.2GHz/1.4GHz
RAM	2kB SRAM	64kB SRAM	1GB LPDDR2 SDRAM
EEPROM	Yes - 1kB	No	No
Board Power Supply	5V	5V	5V
Circuit Operating Voltage	3.3V or 5V	3.3V	3.3V
Flash Memory	32kB	4MB	Micro SD storage size (8GB recommend)
Digital I/O Pins	14	11	40
PWM Pins	6	10	40 (through coding)
Analog Input Pins	6	1 (ADC 10 bit)	0
Connectivity	None	IEEE 802.11 b/g/n Wi-Fi	Gigabit Ethernet IEEE 802.11 b/g/n/ac Wi-Fi
Programming Languages	C/C++	C++ / Python / Lua	Lots of them
Cost[1]	9€ - 21€[2]	7,50€	36€ - 40€

Table 2.1: Comparison of the most important specifications between the three dev boards.

As we can see in the table, all dev boards have very different specifications and because of that each kit should correspond to the needs of each project. Each board still have it's own problems, for instance in the *Arduino UNO* when compared to the other two is far better in terms of GPIO pins, but we are still lacking some kind of connectivity but this problem can be resolved by buying an *Arduino* shield, like an Ethernet shield or WiFi shield.

The dev board *NodeMCU* 1.0 is a very good option since it's very cheap and already comes with some GPIO ports and embedded WiFi, but if we need to use some sensors that needs to use the analog port, we only have one. This problem can be fixed by buying a multiplexer chip, which is very cheap but we also need to program it (see next chapter for more information).

[1] All prices where checked on Amazon ES.
[2] Lower prices are boards using the Arduino design but not it's brand while higher prices are Arduino brand.

Even though the *RPi* has a lot of good features it's still very expensive when compared to the other two, because we need to buy the board and some extra hardware to use it, unless we already have that extra hardware (micro SD card, keyboard, mouse, monitor, etc.). It also has the problem that all of it's GPIO are all digital, meaning if we want to use any device/sensor that needs to use analog ports we just can't use them, unless we buy an ADC chip (converts analog signal to digital) and program it.

2.5 Fixing GPIO related problems

In this section we are going to address all the GPIO problems talked in section 2.4. I'm going to explain how both chips, ADC and multiplexer, work and in a later chapter we will see how can we program them.

2.5.1 Using an Analog to Digital Converter (ADC)

In the world of the electronics we have two type of signals: analog and digital. Summarizing, an analog signal is a continuous signal that contains time-varying quantities and a digital signal refers to an electrical signal that is converted into a pattern of bits. A digital signal is easily represented by a computer because each sample can be defined with a series of bits that are either in the state logic low (0) or logic high (1). Computers or microcontrollers are only capable of detecting digital signals. It understands 0V as binary 0 and 5V as binary 1. Meaning, if a microcontroller is powered with 5V, then it understands 0V as binary 0 and 5V as binary 1, but since the world isn't so simple what happens when a signal is 3.58V? Is is considered a 0 or a 1? Since we need to measure signals that vary (analog signals), a 5V analog sensor may output values between 0.01V and 4.99V.

To decipher these type of signals, we need to use an "Analog to Digital Converter", also known as ADC. An ADC is a very useful tool that converts analog voltage into a digital number. Not every board has the ability to do analog to digital conversions, which is the case of the *RPi*, and because of that we need to buy one. Obviously, it can be used with any digital pins, so if we want to use it with *Arduino UNO* or the *NodeMCU* we can use it, but since those boards already have an ADC we don't need to buy one. The ADC we are using in this thesis is the MCP3008,

but you could you any ADC. We just decided to use this one, because of its characteristics:

- **10-bit resolution** - meaning when the analog signal is converted the digital number will be represented between 0 and $2^{10} - 1 = 1023$.
- **8 input channels** - We can use 8 analog sensors.
- **SPI serial interface** - Is an interface bus commonly used to send data between microcontrollers and small peripherals. It uses separate clock and data lines, along with a select line to choose the device you wish to talk to.
- **Operating voltage** - between 2.7V and 5.5V.

Normally I would explain how technically the communication with the ADC is suppose to work, but since we are using the SPI communication and we have a library to communicate with it we don't need to know all the details. But, what we need to know is:

- To connect the ADC device to the *RPi*, or any other board, to communicate via SPI we need to connect the pins (Grusin, 2018):
  - MCP3008 VDD to *Raspberry Pi* 3.3V
  - MCP3008 VREF to *Raspberry Pi* 3.3V
  - MCP3008 AGND to *Raspberry Pi* GND
  - MCP3008 DGND to *Raspberry Pi* GND
  - MCP3008 CLK to *Raspberry Pi* SCLK
  - MCP3008 DOUT to *Raspberry Pi* MISO
  - MCP3008 DIN to *Raspberry Pi* MOSI
  - MCP3008 CS/SHDN to *Raspberry Pi* CE0
- To read the analog signal, we need to configure an MCU Transmitted Data, which is a data package with a 3 byte size:
  1. First package:

0	0	0	0	0	0	0	1

     (a) last bit, which is 1, is the start bit. Basically means that the ADC is ready to read an analog value from a certain channel.

2. Second package:

SGL/DIFF	D2	D1	D0	X	X	X	X

   (a) In this package we choose the input mode and the channel to read.

   (b) SGL/DIFF - single-ended mode or differential mode. We are always going to use single-ended mode. SGL mode is binary 1 and DIFF mode is binary 0.

   (c) D2/D1/D0 - is the channel we want to read in binary. The ADC has 0 to 7 channels, so if we want to read the 5th channel we need to send 101.

   (d) X's - are don't care, meaning it doesn't matter if you send a 1 or 0. But in programming to simplify the math we send 0's.

3. Third package:

X	X	X	X	X	X	X	X

   (a) X's - are don't care, but we send 0's to simplify the math in the program.

- After the ADC converts the analog signal to digital, we need to read the MCU Received Data, ,which is a data package with a 3 byte size:

1. First package:

?	?	?	?	?	?	?	?

   (a) ? - means we don't know what value is there.

2. Second package:

?	?	?	?	?	Null	B9	B8

   (a) ? - same as above.

   (b) Null - is always binary 0. It means that the ADC knows it's ready to read incoming data.

   (c) BX - is either 1 or 0

3. Third package:

B7	B6	B5	B4	B3	B2	B1	B0

   (a) BX - is either 1 or 0

4. BX is represented by a 10-bit value and when it's converted to decimal it represents a value between 0 and 1023.

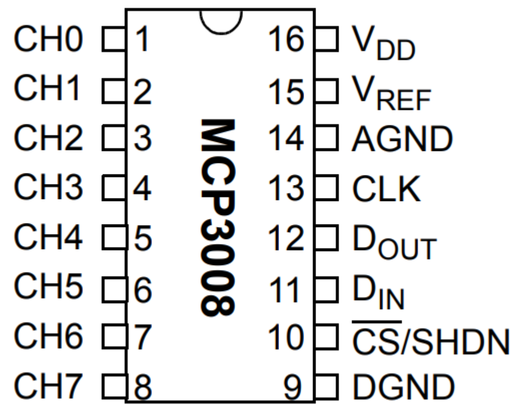


Figure 2.11: ADC MCP3008 pin configuration.

2.5.2 Using a Multiplexer

Multiplexing, or muxing, is the generic term used to describe the operation of combining multiple signals or streams of information over a single transmission channel at the same time. The opposite, is the term called demultiplexing, or demuxing, which recovers different signals on the same single transmission channel and distributes them over different channels. The multiplexer is the device created to do the multiplexing and demultiplexing operations. For this project we are using the mux CD4051B, because:

- **It's very cheap**.
- **Can read a wide range of analog and digital signals** - in this specific case we just need to make sure it covers the operating voltage of most sensors or other devices, which usually is 5V.
  - Digital signals: 3V to 20V
  - Analog signals: 0V to 20V
- **Low resistance** - it's related to signal distortion, meaning the lower the resistance, the less distortion. This just prevents the loss of information when reading the channels. Low resistance is represented by R_{ON} in the datasheet.
- **Channel leakage** - it's almost the same principal with the resistances but in this case is has to do with other components inside the chip. In this case we just need to read the datasheet to know at what voltage it happens.

- **Switch speeds** - these speeds are related to the response time to switch the control inputs. Is represented by t_{ON} in the datasheet.

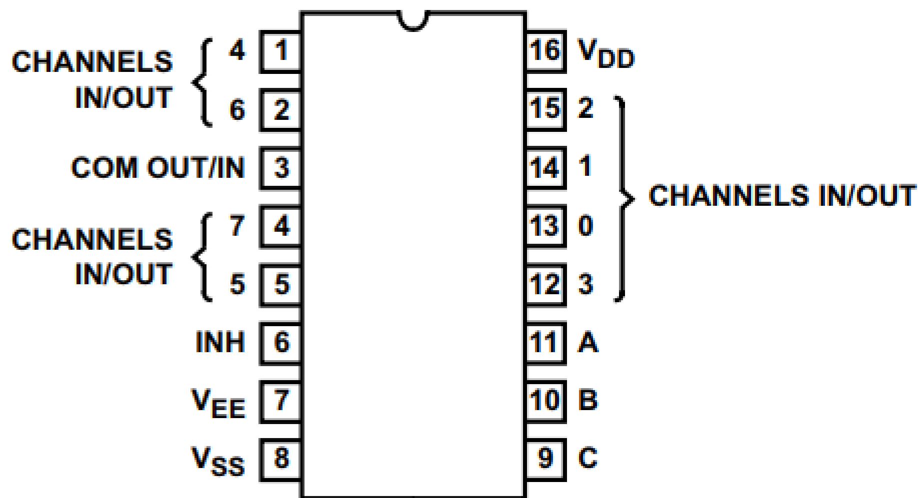


Figure 2.12: Multiplexer CD4051B pin configuration (TexasInstruments, 2017).

The CD4051B device is a single 8-channel multiplexer having three binary control inputs, A, B, and C, and an inhibit input. The three binary signals select 1 of 8 channels to be turned on, and connect one of the 8 inputs to the output. When these devices are used as demultiplexers, the "CHANNEL IN/OUT" terminals are the outputs and the "COMMON OUT/IN" terminals are the inputs (TexasInstruments, 2017).

2.5.3 Using a Logic Level Converter

As we have seen in section 2.4, there are two development boards that need 3.3V for circuit operations. The problem is that most of the sensors or other devices used with these boards need to send a signal, to these boards, of at least 5V and since those boards can only operate with 3.3V, if that happens the board can burn.

To solve this problem we are using a logic level converter. The main purpose of this device is to convert a high voltage to a lower voltage. This device needs to be powered from the two voltages sources (high voltage and low voltage) that your system is using. High voltage (5V) to the "HV" pin, low voltage (3.3V) to "LV", and ground from the system to the "GND" pin. Then we connect our 5V devices to any HVX pin (HV1 to HV4) and connect from LVX pin (LV1 to LV4), respectively, to the development board.

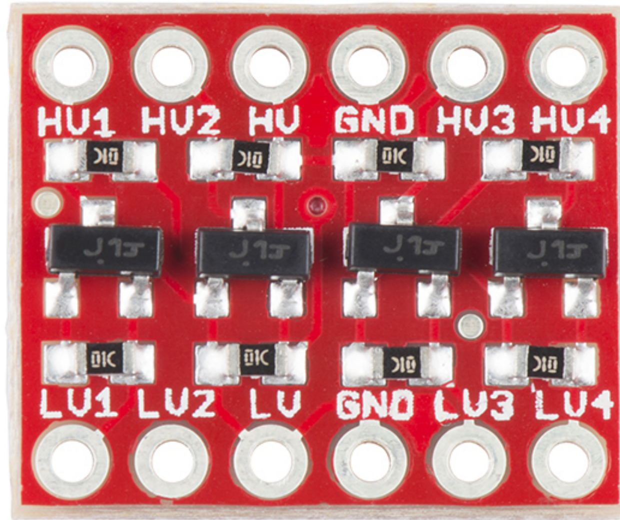


Figure 2.13: Logic Level Converter

3 Hardware - Sensors and Actuators

The main focus of this thesis is to program a software framework (we will see how to do it in the next chapter) for a wheel robot that can use multiple dev boards as its core. Wheeled robots are robots that use motorized wheels to navigate around the ground. To help with the robot navigation we are going to use sensors that can detect objects so the robot can dodge them. Since it's a wheeled robot, it also needs to use some kind of motors to propel itself. Finally, we need to use a chassis to accommodate all the hardware we are going to use, to mount the robot.

For this thesis, we can safely assume that the robot will consist of:

- **One of the three dev boards**: *Arduino UNO*, *NodeMCU* or *Raspberry Pi*.
- **Sensors**, in this case we are going to use proximity and sonar sensors.
- **Motors**, to propel the robot. In this project we are going to use the ones that already come with the chassis platform.
- **Motor driver**, to be able to control the motors.
- **Wheels**, you can use any type of wheels but in this project we are going to use the ones that already come with the chassis platform.
- **Batteries**, to power all the hardware used. It doesn't matter what type of battery it is, we just need to make sure it has enough juice to power all the hardware.
- **Chassis**, the platform that accommodates all the hardware used by the robot.

Figure 3.1: Chassis design *Devastator Tank Mobile Robot Platform* (DFRobot, 2018).

3.1 Sensors

A sensor is a device whose purpose is to detect changes or events from the physical environment and convert that information into a human-readable language, which in electronics translates to a voltage. It's always important to find how a sensor works from a scientific point of view. How can the sensor translate the physical parameter into an electronic parameter. This is important, because this way we can find its limitations and understand how the device works. To understand how these sensors work we just need to look at the datasheet, because in there we have all the information needed.

3.1.1 Ultrasonic sensor *HC-SR04*

This sensor is called ultrasonic sensor because it uses sonar, just like a bat or a whale. Sonar is an object detection system that emits sound pulses and measures their returning after being reflected (echo).

Figure 3.2: Ultrasonic sensor *HC-SR04* design and pins (CytronTechnologies, 2013).

To connect the sensor to any dev board we need to connect the pins as follows:

- **V**cc - to a power source of 5V. It can be distributed by the board or from external sources.
- **GND** - connect to the common ground circuit path.
- **Trigger** - to any digital GPIO pin.
- **Echo** - to any digital GPIO pin.

Sensor features (CytronTechnologies, 2013):

- **Operating voltage**: 5V
- **Working current**: 15mA
- **Ranging distance**: 2cm to 400cm
- **Resolution (Ranging Accuracy)**: 0.3cm (how far it deviates from a “real” measure)
- **Measuring angle**: 30º

- **Effectual angle**: <15° (max angle measurement at which objects are accurately measured)
- **Ultrasonic range**: 40kHz
- **Input trigger pulse**: $10\mu S$

How does the sensor works? The sensor emits an ultrasound wave at 40kHz which travels through air, detects an object on it's way and it will reflect back to the sensor. In order to generate this ultrasound wave to start the measurement, the trigger pin needs to receive a pulse on high state for at least $10\mu S$. That will initiate the sensor, and the sensor transmitter will send out an 8 cycle ultrasonic burst at 40kHz. Then the sensor receiver will wait for the reflected ultrasonic burst. When the receiver detects the wave, it will set the echo pin on a high state and will output the time in microseconds that the sound wave traveled.

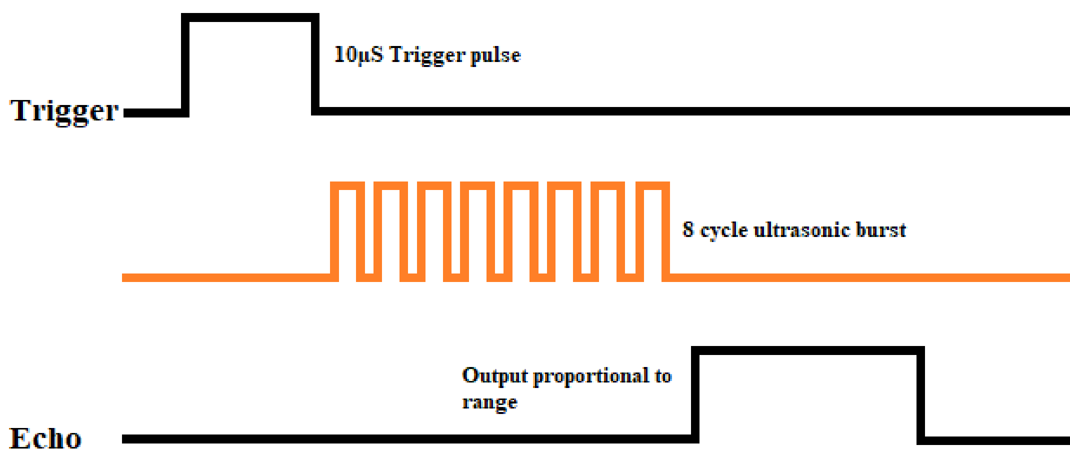


Figure 3.3: Ultrasonic sensor signals diagram.

To determine the distance of the object, we need to use the a formula that correlates distance, time and speed, which translates to:

$$Distance = Speed * Time \tag{3.1}$$

or,

$$d = v * t \tag{3.2}$$

Now we have to consider two things. The first one is that we are using ultrasound waves to detect the objects, so the speed variable is actually the speed of sound, which is $v = 340m/s$, but we will consider $v = 0,034cm/s$, because it facilitates the math if we want to detect an object using *cm*. The second thing is that the sound wave travels twice, the first time when is transmitted and the second time when it's reflected back to the sensor and because of this we need to divide the time by two, which means the formula to calculate the distance of an object is:

$$Distance = Speed\,of\,sound * \frac{Time}{2} \tag{3.3}$$

3.1.2 Analog Infrared Sensor *Sharp GP2Y0A02YK0F*

These types of sensors are called proximity sensors. A proximity sensor is a sensor able to detect nearby objects without any physical contact. They often emit a beam of electromagnetic radiation and look for changes when the signal returns. The *Sharp GP2Y0A41SK0F* is a distance measuring sensor unit, composed of an integrated combination of PSD (position sensitive detector) , IR-LED (infrared emitting diode) and signal processing circuit. The variety of the reflectivity of the object, the environmental temperature and the operating duration are not influenced easily to the distance detection because of adopting the triangulation method. This device outputs the voltage corresponding to the detection distance. It's most used applications are: cleaning robot, personal robot (our case) and sanitary (for instance, sensors to automatically push the toilet water) (SHARP, 2018).

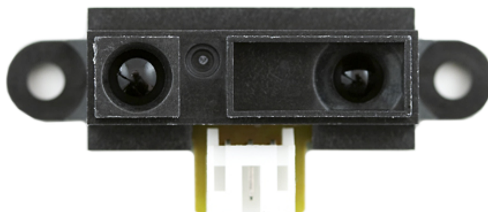


Figure 3.4: Infrared sensor *Sharp GP2Y0A02YK0F*.

The sensor uses a 3-pin JST (white connection in the image above). To connect the sensor to any dev board we need to connect the pins as follows :

- **V**cc - to a power source of 5V. It can be distributed by the board or from external sources.
- **GND** - connect to the common ground circuit path.
- **Vo** - to any analog GPIO pin.

Sensor features (SHARP, 2018):

- **Operating voltage**: 4.5V to 5V
- **Working current**: 12mA
- **Ranging distance**: 4cm to 30cm
- **Resolution (Response time)**: 16.5ms (how long it takes to update the measuring signal)

How does the sensor works? As we can see in figure 3.4, the sensor "eyeballs" are composed by an infrared light emitting diode and a photo diode. The IR diode emits an IR light to an

object and when that light reaches the objects, it's reflected back. The photo diode detects the reflected light and gives a response in terms of a change in resistance. This change is measured as voltage. The amount of reflection and reception varies with the distance, which causes a change in the input voltage. This variation in input voltage is used for proximity detection. The amount of reflected light depends upon the color of the surface from which was reflected. The reflection is different for different colored surfaces. The brighter the color, the more reflection is returned.

To calculate the distance of an object we need linearize the output by looking at the next figure.

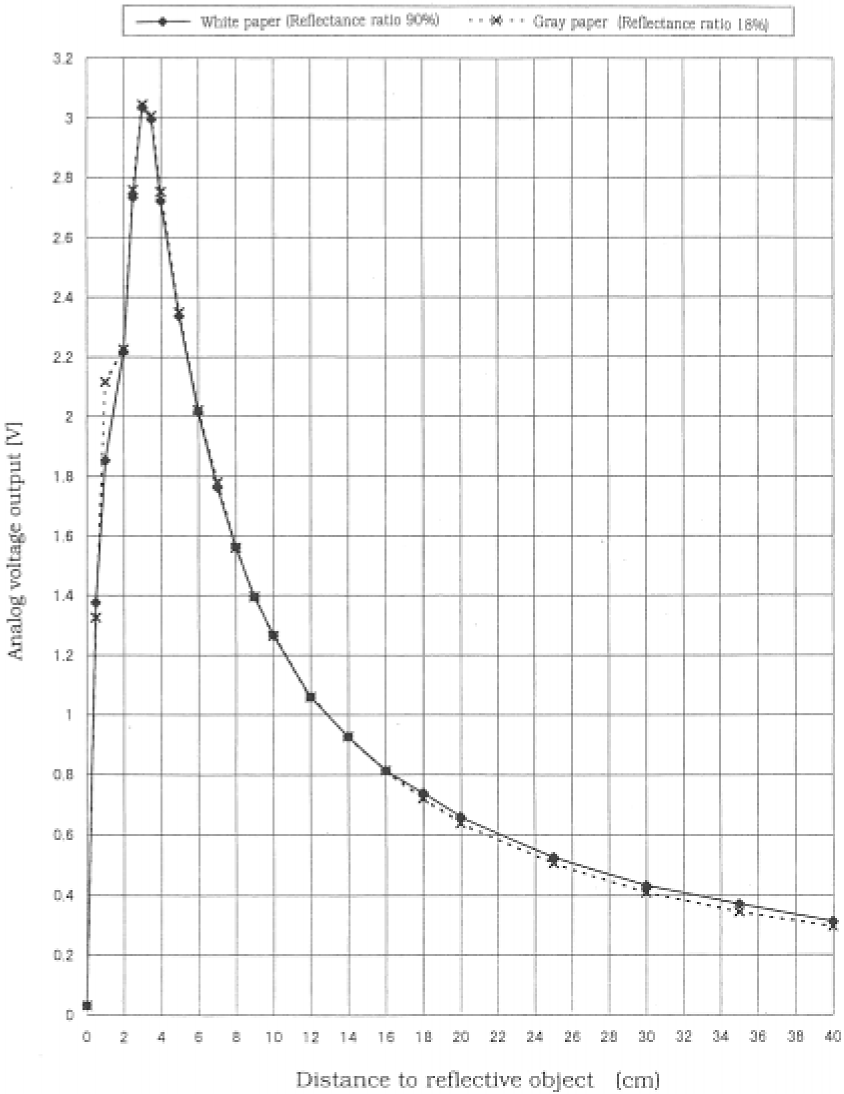


Figure 3.5: Sensor *Sharp GP2Y0A02YK0F* plot of an analog voltage output as a function of the inverse, of the distance to a reflective object.

The relationship between the sensor's output voltage and the inverse of the measured distance

is approximately linear over the sensor's usable range. We can use this plot to convert the sensor output voltage to an approximate distance by constructing a math system of equations that relates the inverse of the output voltage (Y) to distance (X). To do so, we need to use an equation that represents the curve in the plot:

$$y = a * x^b \tag{3.4}$$

But since we want to know the distance, we are going isolate the X value:

$$x = y^a * b \tag{3.5}$$

Now we can use that function to create the system of equations. This system will have 2 equations, considering 2 points in the sensors response graph of figure 3.5, approximately, minimum and maximum values of the voltage (Y axis) and distance (X axis), respectively, when the curve drops. For this sensor and by trial and error, we find out that the best values for fitting the response, correspond to the following coordinates (X,Y): (4, 2.7) and (35, 0.3)

$$\begin{cases} 4 = 2.7^a * b \\ 35 = 0.3^a * b \end{cases} \tag{3.6}$$

Now, if we resolve the system:

$$\begin{cases} b = \frac{4}{2.7^a} \\ - \end{cases} \Rightarrow \begin{cases} - \\ 35 = 0.3^a * \frac{4}{2.7^a} \end{cases} \Rightarrow \begin{cases} - \\ 35 = 4 * \frac{0.3^a}{2.7^a} \end{cases} \Rightarrow$$

$$\Rightarrow \begin{cases} - \\ \frac{35}{4} = \left(\frac{0.3}{2.7}\right)^a \end{cases} \Rightarrow \begin{cases} - \\ a = \frac{log(\frac{35}{4})}{log(\frac{0.3}{2.7})} = -0.98718 \end{cases} \Rightarrow \begin{cases} b = \frac{4}{2.7^a} \\ - \end{cases} \Rightarrow$$

$$\Rightarrow \begin{cases} b = \frac{4}{2.7^{-0.98718}} = 10.663 \\ - \end{cases} \Rightarrow \begin{cases} b = 10.663 \\ a = -0.98718 \end{cases}$$

Finally, we need to change the values of a and b in equation 3.5:

$$x = y^{-0.98718} * 10.663 \tag{3.7}$$

Now, we have a model to find the distance value from the output voltage that the sensor gives. To try to match the plot of this model with the graph of figure 3.5 we need to use equation 3.4, with a and b values found in the system of equations.

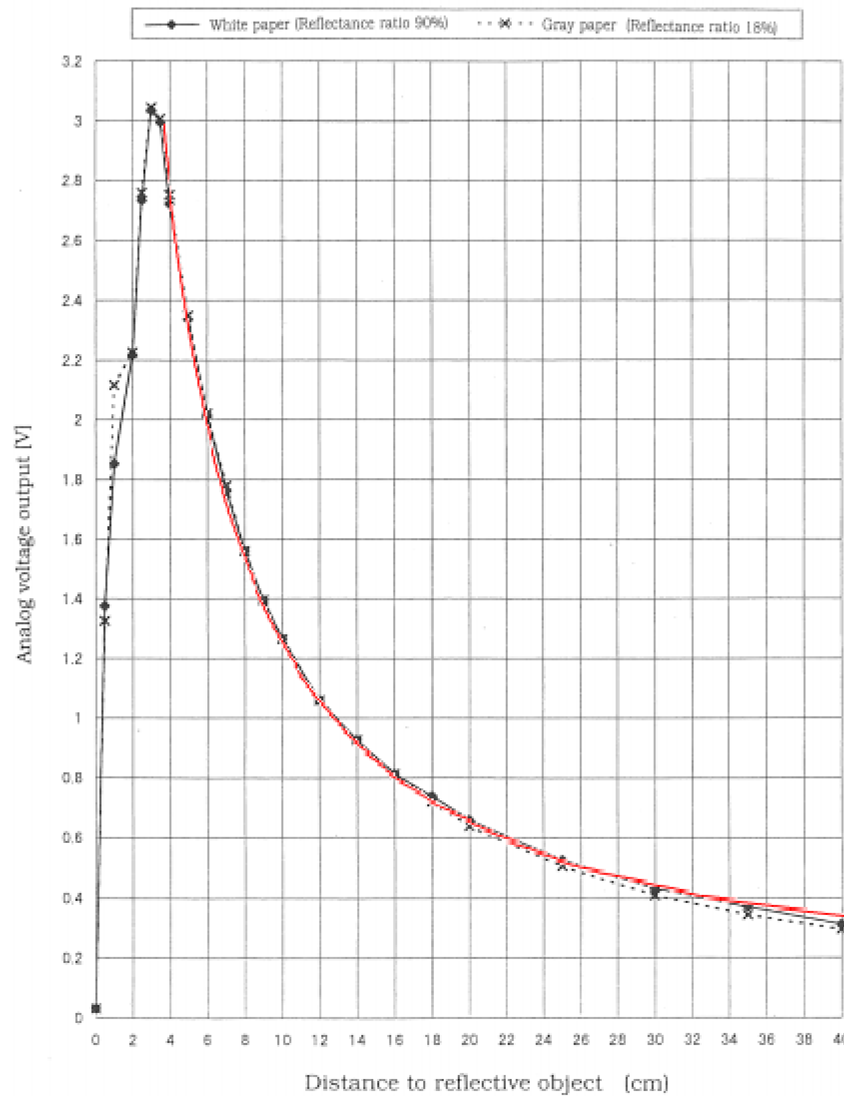


Figure 3.6: Sensor *Sharp GP2Y0A02YK0F* plot, with linearized equation in red.

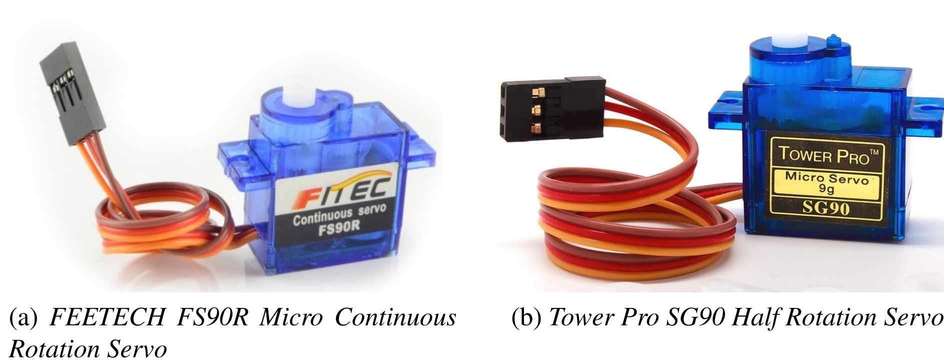
As we can see, finding the best 2 points by trial and error, we can obtain a model that almost matches the sensor response.

3.2 Actuators

An actuator is a component that is responsible for moving and controlling some mechanism, and also converts a control signal into a mechanical action. This control signal can be based on electric, hydraulic, mechanical or thermal values. Actuators acting together with appropriate sensors ties a control system to its environment. Electric actuators often have a motor driven by electric signals. The control signal often comes from a microcontroller running control software.

For this project we are going to consider two kind of servo motors that can be used as actuators. These servos are:

- **FEETECH FS90R Micro Continuous Rotation Servo**
- **Tower Pro SG90 Half Rotation Servo**



(a) *FEETECH FS90R Micro Continuous Rotation Servo* (b) *Tower Pro SG90 Half Rotation Servo*

Figure 3.7: Design of both actuators.

The first servo have continuous rotation, meaning it can rotate 360º, while the *Tower Pro SG90 Servo* can only rotate 180º (that's one of the reasons why is called half rotation servo). This servos can be used to move actuators or even can be used as a platform to drive sensors. Example:

using an ultrasonic sensor on top of a half rotation motor to cover more field of view, instead of using multiple sensors.

3.2.1 Servo Motors Fundamentals

A servo motor is a rotational or translational motor to which power is supplied by a servo amplifier and serves to apply torque or force to a mechanical system, such as an actuator or brake. Servo motors allow for precise control in terms of angular position, acceleration and velocity. This type of motor is associated with a closed-loop control system. A closed loop control system considers the current output and alters it to the desired condition. The control action in these systems is based on the output of motor. It uses a positive feedback system to control motion and final position of the shaft.

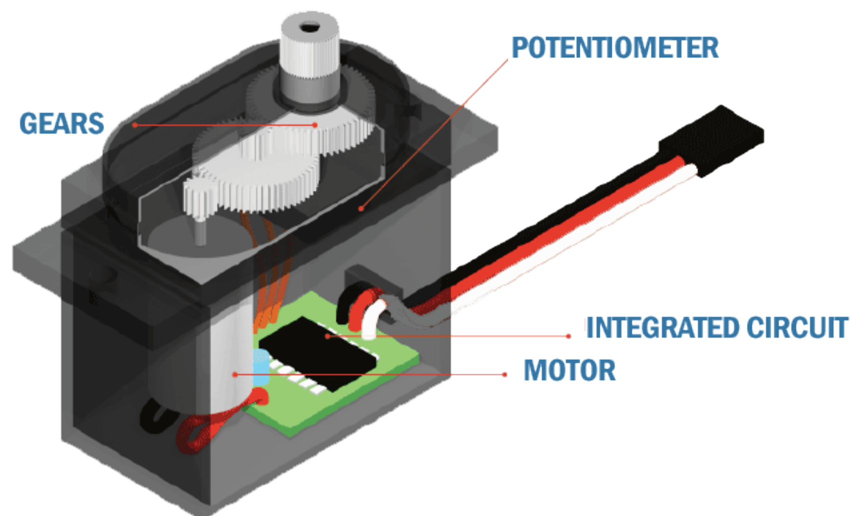


Figure 3.8: Inside the servo motor.

There are two types of current flow in these motors – AC and DC, but in this project we are only going to use the DC version. DC servo motors are not designed for high current surges and are better suited for smaller applications. In a DC motor, speed is directly proportional to the supply voltage. Servo motors come in many sizes and we are using two different types. These types are (ISLProductsInternational, 2018b):

- **Continuous Rotation** - is a servo that does not have a limit on its range of motion. Instead of having the input signal determine which position the servo should rotate to, the continuous rotation servo relates the input to the speed of the output and direction. This enables them to rotate in both clockwise and anti-clockwise.
- **Positional Rotation** - this is the most common type of servo motor. The output shaft only rotates in about half of a circle, or 180º. It has physical stops placed in the gear mechanism to prevent turning beyond these limits to protect the rotational sensor.

Important characteristics when choosing a servo motor:

- **Precision**
- **Speed**
- **Encoder**
- **Versatility**

3.2.2 Controlling the servos

Modern servo motors, such as the ones used in this project, are controlled by sending a PWM signal. This signal represents a series of repeating pulses of variable width where the width of the pulses determines how far the motor will turn. There is a minimum pulse, a maximum pulse, and a repetition rate. The motor's neutral position is defined as the position where the servo has the same amount of potential rotation in both clockwise or counter-clockwise directions. The PWM sent to the motor determines the position of the shaft, and based on the duration of the pulse the rotor will turn to the desired position. Most servo motors expect to see a pulse every 20 milliseconds (ms) and the length of the pulse will determine how far the motor will turn.

When these servos are commanded to move, they will move to the position and hold that position. If an external force pushes against the servo while the servo is holding a position, the servo will resist from moving out of that position. The maximum amount of force the servo can exert is called the torque rating of the servo. Servos will not hold their position forever though; the position pulse must be repeated to instruct the servo to stay in position. (Jameco, 2018)

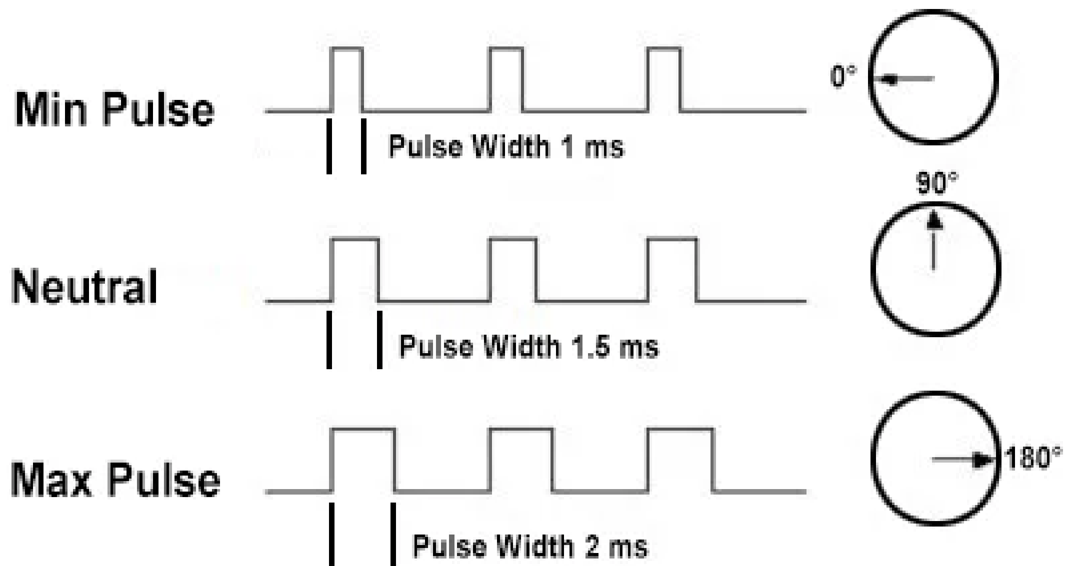


Figure 3.9: PWM controls servo position (Jameco, 2018).

3.3 DC Motors Fundamentals

A DC motor is a rotating electrical device that converts direct current into mechanical energy. An inductor inside the DC motor produces a magnetic field that creates rotary motion as DC voltage is applied to its terminal. Inside the motor is an iron shaft wrapped in a coil of wire. This shaft contains two fixed, North and South, magnets on both sides which causes both a repulsive and attractive force, in turn, producing torque (ISLProductsInternational, 2018a).

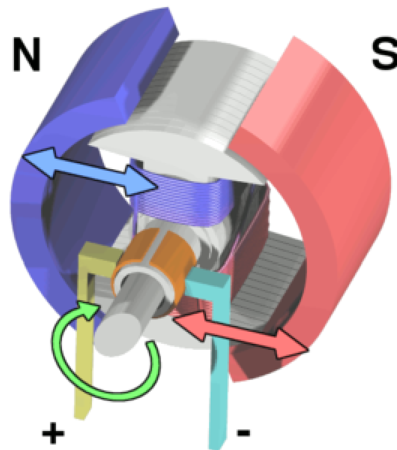


Figure 3.10: Inside the DC motor (ISLProductsInternational, 2018a).

What is a DC Gear Motor? A gear motor is an all-in-one combination of a motor and gearbox. The addition of a gear head to a motor reduces the speed while increasing the torque output. The most important parameters in regards to gear motors are speed (rpm), torque (lb-in) and efficiency (%). In order to select the most suitable gear motor for your application you must first compute the load, speed and torque requirements for your application (ISLProductsInternational, 2018a).

Important characteristics when choosing a DC motor:

- **Voltage**
- **Current**
- **Power**
- **Torque**
- **RPM**
- **Duty Cycle**
- **Rotation (Clockwise or Anti-clockwise)**

To move the robot we are going to use two DC motors which can be seen in the figure below.

Figure 3.11: Micro DC Geared Motor with Back Shaft.

3.4 Controlling the motors

To easily control the motors in this type of projects, we need to use a piece of hardware called "Motor Driver." A motor driver is a circuit used to run a motor, such that they are commonly used as interfaces between MCUs and motors. These circuits are current amplifiers, meaning the input to the circuit of the motor driver is a low current signal. The function of the circuit is to convert the low current signal to a high current signal and is then given to the motor. In motor interfacing with MCUs, the primary requirement for the operation of the MCU is low voltage and small amount of current. But, since the motors require a high voltage and current for its operation, we can say the output of the MCU or CPU is not enough to drive a motor. In such cases direct interfacing of MCUs to the motor is not possible. So we need to use a "Motor Driver Circuit". These circuits, also make it possible to control the speed and direction of the motors.

We can control the speed of a motor by simply controlling the input voltage to the motor. The

most common method of doing that is by using a Pulse Width Modulation (PWM) signal.

Pulse Width Modulation is a technique used in digital signals, that allows generating an analog signal using a digital source. A PWM signal consists of two main components that define its behavior: a duty cycle and a frequency. The duty cycle describes the amount of time the signal is in a high state (ON), as a percentage of the total time it takes to complete one cycle. The frequency determines how fast the PWM completes a cycle (for instance, 1 Hz would be we equal to 1 cycle per second), and therefore how fast it switches between high and low states (ON and OFF). By cycling a digital signal ON and OFF at a fast enough rate, with a certain duty cycle, the output will appear to behave like an analog signal when providing power to devices.

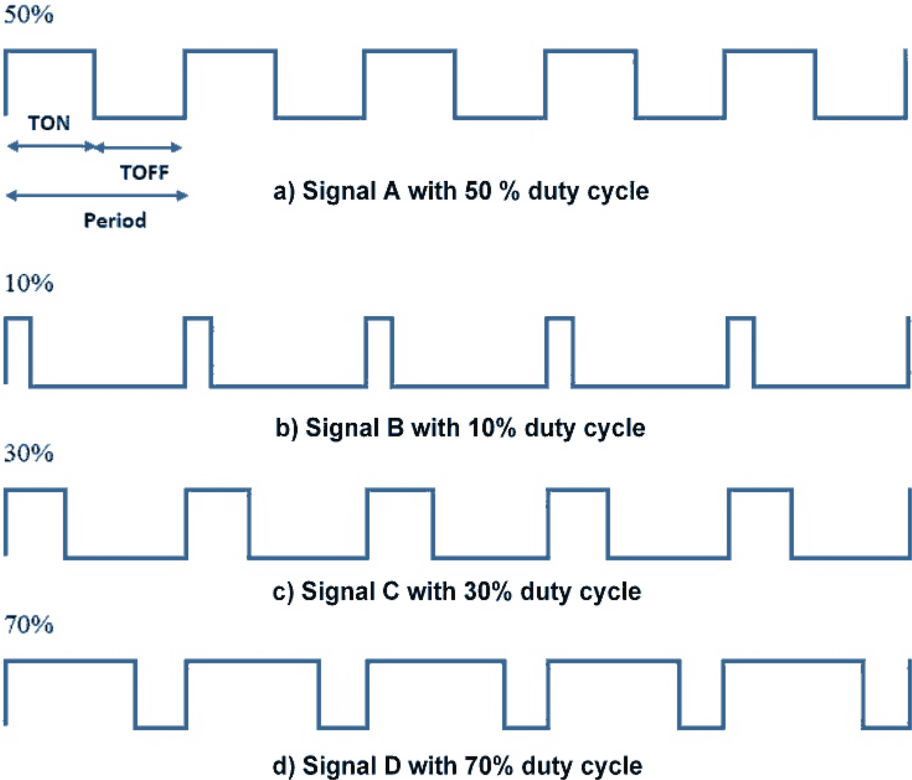


Figure 3.12: PWM signals with different duty cycle waveforms. T_{ON} represents the time the signal is in high state (ON) and T_{OFF} represents the time the signal is in low state (OFF). Period represents a complete cycle (ElectronicWings, 2017).

The fraction for which the signal is ON over a period is known as duty cycle.

$$Duty\,Cycle\,(\%) = \frac{T_{ON}}{Total\,Period} * 100 \tag{3.8}$$

The power applied to the motor can be controlled by varying the width of these pulses and thereby varying the average DC voltage applied to the motors terminals. By changing or modulating the timing of these pulses the speed of the motor can be controlled.

On the other hand, to control the rotation direction of the motor, we need to inverse the direction of the current flow through the motor and the most common way of doing that is by using an H-Bridge. The name H-Bridge is used because of the diagrammatic representation of the circuit. This circuit contains four switching elements (usually transistors) with the motor at the center forming an H like configuration. By activating two particular switches at the same time we can change the direction of the current flow, thus change the rotation direction of the motor.

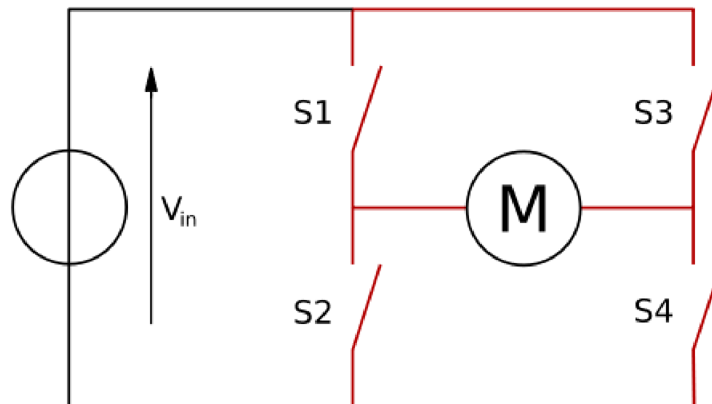


Figure 3.13: H-Bridge circuit diagram with a motor at the center.

In this project we are going to use two different motor drivers:

- **2A Motor Shield For Arduino**
- **Itead L298 Dual H-Bridge Motor Driver**

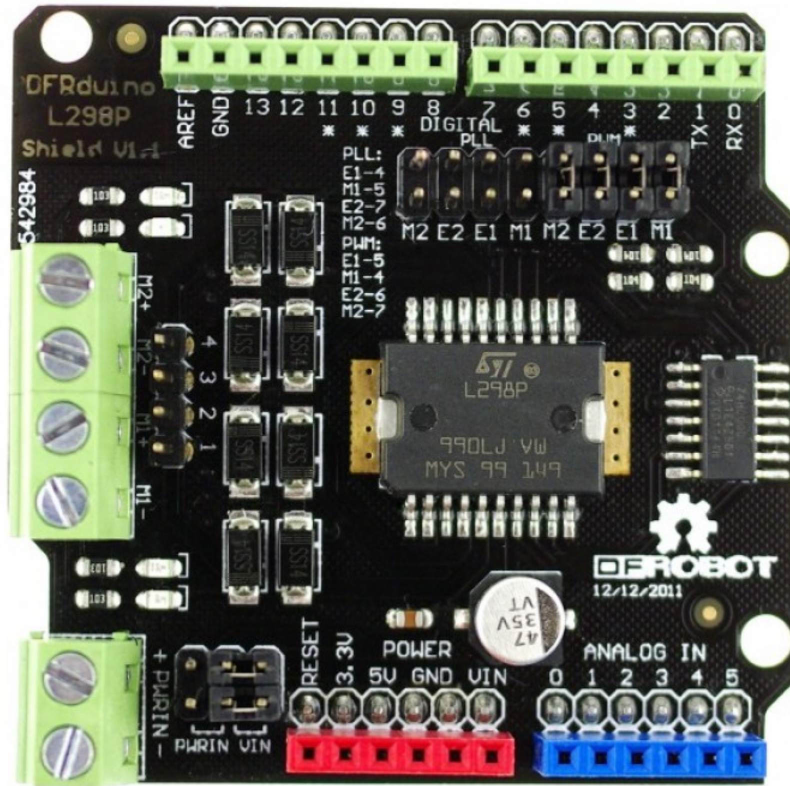


Figure 3.14: Top view of *2A Motor Shield For Arduino.*

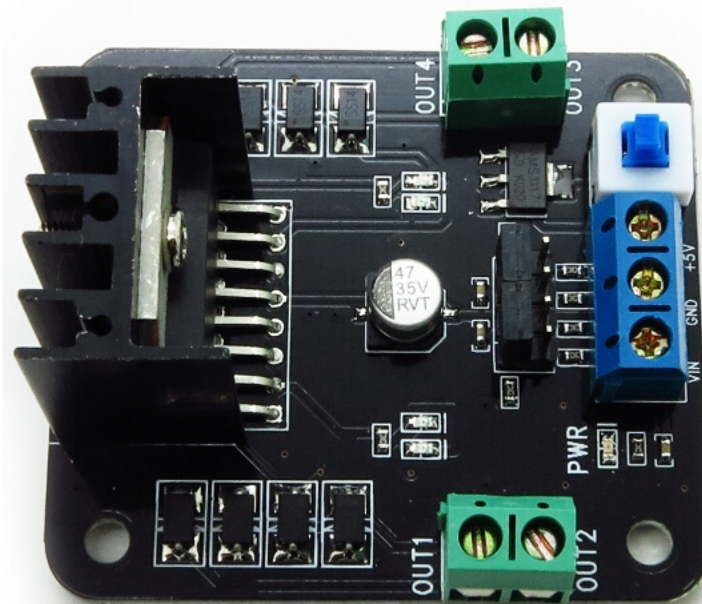


Figure 3.15: Top view of *Itead L298 Dual H-Bridge Motor Driver*.

Basically, these both boards allow us to to easily and independently control two motors of up to 2A each in both directions. The *2A Motor Shield* can only be used with an *Arduino* because its designed to mount an *Arduino*, while the *Itead L298 Dual H-Bridge* can be used with any development board. Besides this problem, the main difference between them is that programmatically the *2A Motor Shield* uses the same pins, for each motor, to control the rotation direction of the motor, while *Itead L298 Dual H-Bridge* needs to use two pins (one is the inverse of the other), for each motor, to control the rotation direction.

Connecting the motors to *2A Motor Shield For Arduino*:

1. **PWRIN Terminal** - external power connection. Vcc and GND.
2. **Motor Terminal** - connect motors by screw to terminals or pin headers above.
   (a) Terminals M1+ and M1- for motor 1, or pin headers 2 or 1, respectively.
   (b) Terminals M2+ and M2- for motor 2, or pin headers 4 or 3, respectively.
3. **Control Pins** - used to control direction and speed of motor. It's basically any digital port, with the exception that to control the speed we need to connect to a pin that supports PWM, or we can use some predefined pins with selection jumpers, which support PWM or PLL (Phased Locked Loop).
   (a) **Control Mode Selection Jumpers**
      i. The PWM mode uses E1 and E2 on *Arduino* pins 5 and 6 generate PWM signal.
      ii. The PLL mode uses M1 and M2 on *Arduino* pins 5 and 6 to generate the phase control signal.
   (b) **PWM Control Mode** - the most common and the one we are going to use.
      i. Digital pin 4 - Motor 2 direction.
      ii. Digital pin 5 - Motor 2 speed.
      iii. Digital pin 6 - Motor 1 direction.
      iv. Digital pin 7 - Motor 1 speed.

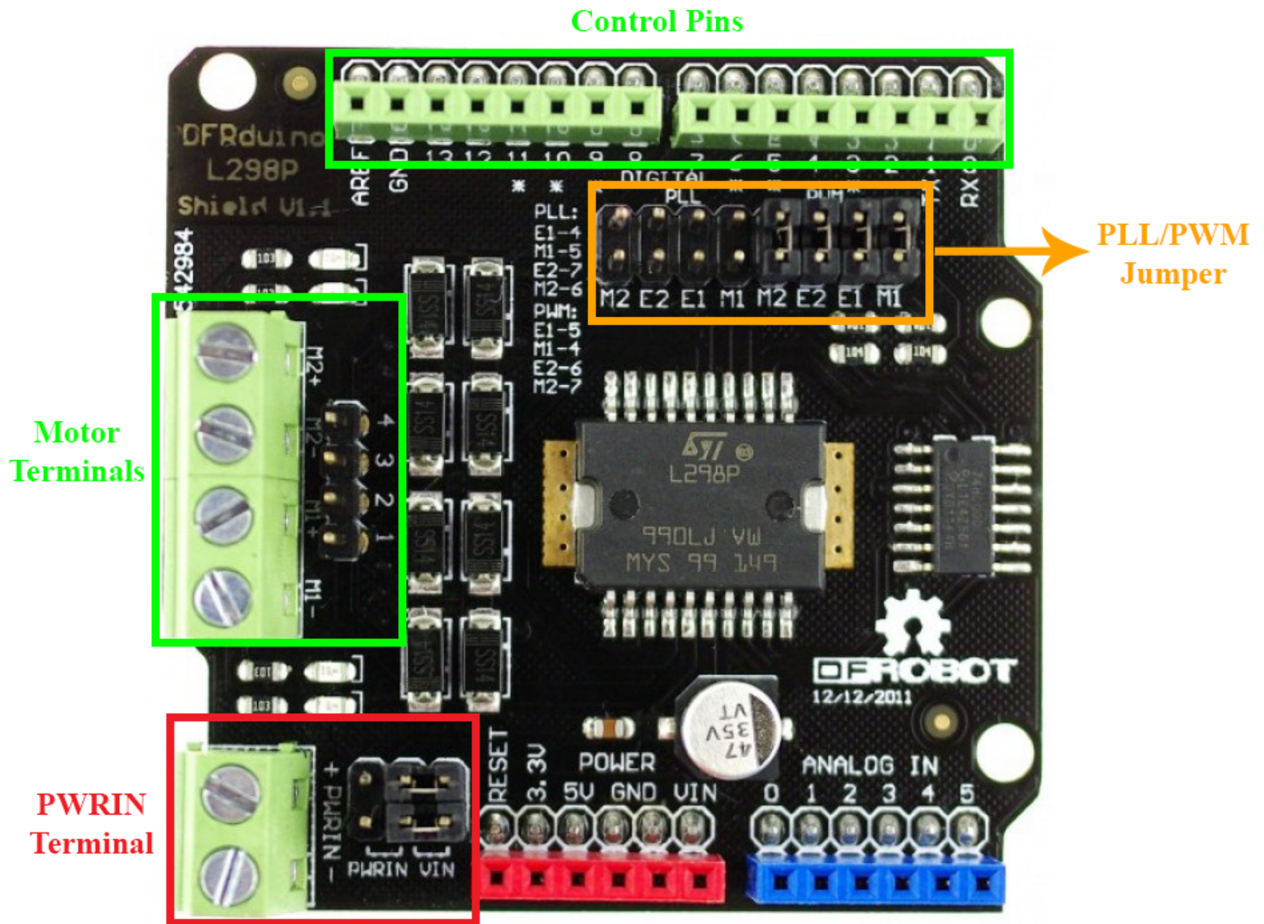


Figure 3.16: Pin description of *2A Motor Shield For Arduino.*

Connecting the motors to *Itead L298 Dual H-Bridge*:

1. **PWRIN Terminal** - external power connection.
   (a) **VIN and GND** - connect motor power supply.
   (b) **+5V** - ideal to power development boards.
2. **Motor 1 Terminal** - connect motor 1 to **OUT1** and **OUT2**.
3. **Motor 2 Terminal** - connect motor 2 to **OUT4** and **OUT3**.
4. **Control Pins** - used to control the direction and speed of motor.
   (a) Pins **ENA** and **ENB**, control the speed and connect to any PWM pin.

(b) Pins **I1** to **I4**, control the direction and connect to any digital pin.

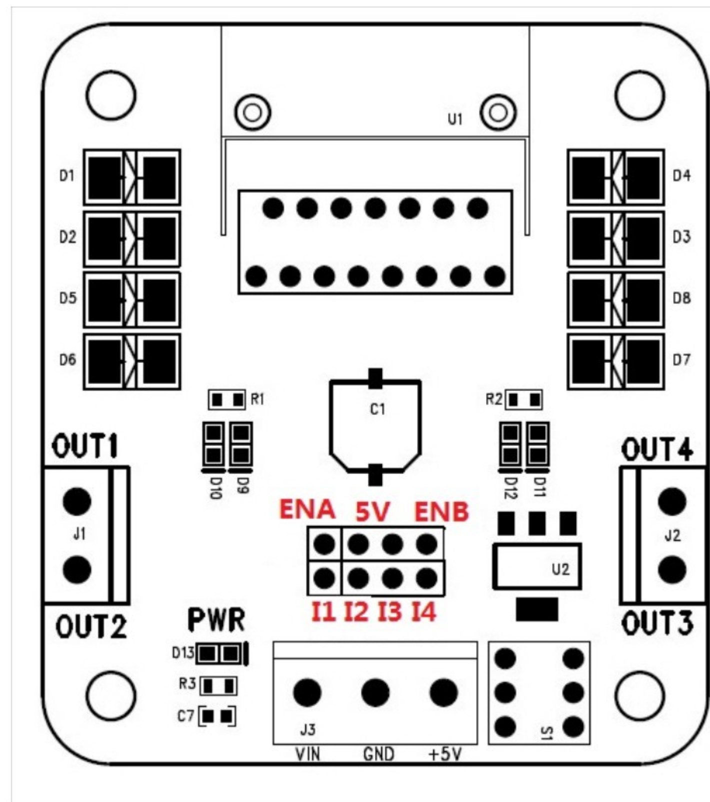


Figure 3.17: Pin view of *Itead L298 Dual H-Bridge Motor Driver*.

4 Software Framework

A software framework is a platform used in application development, which implements reusable code, frequently used in a set of applications, that developers can use, making the development of such applications faster. The purpose of the software framework presented here is to create a platform that can control a set of sensors and actuators, from 3 different processor boards, with no differences, or minimal differences in the code of the application.

The programming language used is C++, because *Arduino UNO* can only be programmed with *Arduino Software,* which is based on C++; *NodeMCU* can also use *Arduino Software;* and *Raspberry Pi* can be programmed in any language. This framework is composed by a set of header files, with *.h* extension. In this case, the header files are called libraries because they implement a set of functions which interface the components used in the robot or missing auxiliary functions in any dev board. The structure of the framework is as follows:

- **framework.h**
  - Libraries used by *Arduino*:
    - **Arduino.h**
    - **cd4051.h**
    - **Servo.h**
    - **WiFi.h**
  - Libraries used by *NodeMCU ESP8266*:
    - **Arduino.h**
    - **cd4051.h**
    - **Servo.h**
    - **ESP8266WiFi.h**
  - Libraries used by *Raspberry Pi*:
    - **pigpio.h**
    - **mcp3008.h**
    - **RPiSockets.h**
      - **RPiClientSocket.h**

        - RPiServerSocket.h
    - **RPiServo.h**
    - **wrapper.h**
- Libraries used by all boards:
    - **infrared-sensor.h**
    - **monitor.h**
    - **motor.h**
    - **sonar-sensor.h**
        - **wiring_pulse.h**

4.1 Understanding the framework

The main focus of the framework is the development of applications that control the robot without having to change much or anything at all in its code, when changing the CPU board. The *Arduino UNO* and the *NodeMCU* share the same programming software, and because of this all the libraries used are pretty much the same with the exception of the WiFi library. In the *Raspberry Pi*, since we don't have a programming software with a preset of libraries, we either need to find an already developed preset of libraries or we need to create them from the beginning. For this project we are going to use a framework that was developed in C/C++ that allow us to communicate with *RPi* GPIO pins, in the same way that the *Arduino Software* allows the same communication in the other two boards. Basically, the *Arduino Software* has its own framework.

Now, since we know that there are some libraries that can't be used in all boards we had the need to separate them automatically in the code, so the user don't have the need to manually change which libraries is going to use. To use this approach we are going to use *macros* and the special operator *defined*. In C/C++ language a *macro* is a fragment of code which has been given a name, and the special operator defined is used in the control structure "#if" and "#elif" to test whether a certain name is defined as a macro. In the *Arduino Software* we already have a preset of macros to distinguish the *Arduino UNO* and the *NodeMCU* (the macros for the *NodeMCU* will only be set when we install the board in the *Arduino Software*, see section 2.2.2), but in

the *Raspberry Pi* we need to use the option “-D <macro name>” when compiling the program. Names of the macros for each board:

- **Arduino:** __AVR__
- ***NodeMCU ESP8266*:** ARDUINO_ESP8266_NODEMCU
- ***Raspberry Pi*:** __RPi__

Example of the macros being used in the framework:

```
#ifdef __AVR__
#include <Arduino.h>
#include "cd4051.h"
#include <Servo.h>
#include <WiFi.h>
#endif

/* RPi Libraries */
#ifdef __RPi__ // Raspberry Pi
#include <pigpio.h>
#include "mcp3008.h"
#include "RPiSockets.h"
#include "RPiServo.h"
#include "wrapper.h"
#endif
```

This way the compiler will only use the libraries when it “detects” which dev board is using. Now that we understand how we can separate the code for each board, we can generally talk about all the libraries to better understand how the framework works.

The **framework.h** is the main file of the framework, which contains all the libraries and other information (functions and constants) that is being used by the robot.

To help match the code between the boards and to be able to use all the GPIO functions, we are

going to use two special libraries. One already comes with the *Arduino Software* and the other one is needed to program the *RPi*.

- **Arduino.h** - is the main library used by the *Arduino Software*. Usually we don't need to call this library because when we upload code to the dev boards, the compiler used by the software usually assumes we are calling it, but we are calling it anyway because we want to prevent any bugs coming from the *Arduino Software*. This library contains all the basic functions to control the boards used by the *Arduino Software*. Obviously, every other single board, like the *NodeMCU*, which is adapted to use the *Arduino Software* will also use this library. To know more about all the functions we can use, consult the following reference (arduino.cc, 2018c).
- **pigpio.h** - is a library for the *Raspberry Pi*, written in the C programming language, which allows the control of the GPIO pins. Unlike the *Arduino* library, we need to manually install it and use some options when compiling the code. To know more about how to install and which functions to use, see the following reference (pigpio, 2018).
  - **wrapper.h** - since we want the code used in all boards should have insignificant changes, we created this file whose main purpose is to abstract away the names of the functions used by **pigpio.h**. The name of the functions and the way they are declared are a bit different from the ones used by the *Arduino Software*, but they do essentially the same. Basically, we just adapt the functions of **pigpio.h** to have the same names and declarations. We also have defined the same constants that are used in the file **Arduino.h**.

The other libraries used by the framework handle the use of all electronic components, such as sensors, motors and the chips, mux and ADC. Some of these devices when used with the *Arduino Software* already have some preset libraries, but in the *RPi* we have to program the from the beginning, minus the functions that are used by the lib **pigpio.h**. To understand how those libs work in *Arduino Software* we can check in their website (arduino.cc, 2018c), or we can use the examples in the software.

Libraries used by *Arduino Software*:

- **Servo.h** - has the name implies, this lib allows the control of any type of servo motor.

- **WiFi.h** - this lib only allows *Arduino* boards to connect to a wireless network, because it was designed to use with an *Arduino* WiFi Shield.
- **ESP8266WiFi.h** - this lib allows any ESP8266 module to connect to a wireless network. If you want to use an ESP8266 WiFi Shield for *Arduino UNO*, you also need to use this lib.

Libraries developed to be used with the *RPi*:

- **RPiSockets.h** - this is just an header file that calls two libraries that are intended to work like both WiFi libs used by *Arduino Software*. Actually, those WiFi libs also have separate files like these ones.
  - **RPiServerSocket.h** - this lib was created to *bind()* to a port and *listen()* for a connection from a client. So a server just waits for a conversation and doesn't start one.
  - **RPiClientSocket.h** - is created to *connect()* to a *listen()* server. The client initiates the connection.
- **RPiServo.h** - created for the *RPi* to work like the **Servo.h** lib. It allows the control of any type of servo motor.

Chips, Motors and Sensors libraries:

- **cd4051.h** - lib to read the channels of the multiplexer *CD4051*.
- **mcp3008.h** - allows to read analog values from the 10-bit ADC *MCP3008*.
- **motor.h** - allows the control of the motor drivers with 4 and 6 pins.
- **infrared-sensor.h** - allows the detection of objects with *Infrared Sensor Sharp GP2Y0A41SK0F*.
- **sonar-sensor.h** - allows the detection of objects with *Ultrasonic Sensor HC-SR04*.
  - **wiring_pulse.h** - this file has a function that counts the total time of sending a pulse and the return to its source.

Other libraries:

- **monitor.h** - this library was created with the intent to join the way we would write between the *Arduino/NodeMCU* and a computer, and between the *RPi* and a computer. Because the *Arduino* and *NodeMCU* are microcontrollers, if we want to communicate with them we need to use a serial communication, which is done by the RX and TX pins of both dev boards. On the other hand, to communicate between the *RPi* and the computer, we just need to normally program in C/C++ language. With this lib, to communicate between them and the computer, the functions used will be the same.

4.2 Programming the framework

Every program created in *Arduino Software* is called *sketch* and uses two special functions, *setup()* and *loop()*. These sketches are written in the text editor and are saved with the file extension *.ino*. Since the focus of the framework is to minimize the changes in the code of the different boards, we had to do a similar approach to the system used in *Arduino Software* for the *RPi*. Since we need to manually compile the programs in *RPi*, and we need to compile them with *.ino* extension, we need to use an option when calling the compiler: “g++ -x c++”. Basically, *g++* is a program that calls GCC (C programming language compiler) and treats some preset extensions files as C++ sources files. If we want to compile a file with another extension, like *.ino*, we need to use “-x c++” which automatically links the file as a C++ source file. This is also useful when precompiling C header files with *.h* extension to use in C++ compilations, which will be the case in our framework.

To simulate the two special functions, *setup()* and *loop()*, in C++, we coded those functions in the main file of the framework, with the following code:

```
1 #ifdef __RPi__
2 void setup(void);
3 void loop(void);
4
5 main()
6 {
```

```
7     if (gpioInitialise() < 0)
8         return 1;
9
10    setup();
11    for(;;) loop();
12    return 0;
13 }
14 #endif
```

The *gpioInitialise()* function initializes the *pigpio.h* library. It must be called every time before using the library functions. After adding those functions we can create our *.ino* files and use them like the *Arduino sketch* language.

Before we start programming the electronic devices for the robot, we need to understand that every device has at least a pin that does the communication between these devices and the dev boards. After we connect a device to the dev boards the communication can be done in two ways, using two different signals. The signal can either be analog or digital, and the communication can be from the device to the dev board, and from the dev board to the device. To be able to do this programmatically we will always use these five functions:

- **pinMode(pin, mode)** - sets the GPIO pin to the mode we want. Modes are usually: *INPUT* (from the device to the dev board) or *OUTPUT* (from the dev board to the device)
- **digitalRead(pin)** - read the value from a specified digital pin, which can be *HIGH* or *LOW*.
- **digitalWrite(pin, level)** - sets the GPIO level as *HIGH* or *LOW*.
- **analogRead(pin)** - this function is *Arduino Software* only and reads the value from a specified analog pin. Since *Arduino* boards and *NodeMCU* board has a 10-bit ADC, the values we get from this function ranges from 0 to 1023.
  - the *RPi* function can only be used when using the *mcp3008.h* lib and the value from a specified channel can be read by the method *analogRead(channel)*. The ADC

MCP3008 also has a 10-bit resolution, so the analog values read from it ranges from 0 to 1023.

- **analogWrite(pin, duty cycle)** - generates a PWM wave with a specified duty cycle ranging between 0 (off) and 255 (fully on).
  - For some reason this function when used by the *NodeMCU* board, the duty cycle ranges between 0 (off) and 1023 (fully on).

4.2.1 Programming the multiplexer

As we have seen in the multiplexer specifications, section 2.5.2, to be able to select which channel we want to read, we need just need to send binary values to pins A, B and C. Programmatically we have:

```
1  int readChannel(int channel)
2  {
3      int i = 0;
4      int b[MUX_BITS] = {0};
5      int mask = 0x1;
6      int readCh = 0;
7
8      if (channel > 7 || channel < 0)
9          return -1;
10
11     for (i = 0; i < MUX_BITS; i++)
12     {
13         b[i] = (mask & channel) ? HIGH : LOW;
14         mask = mask << 1;
15     }
16
17     digitalWrite(A, b[0]);
18     digitalWrite(B, b[1]);
19     digitalWrite(C, b[2]);
```

```
20
21      readCh = analogRead(analogPin);
22
23      return readCh;
24  }
```

First, we verify if the value of the channel is ranging between 0 and 7 (we need to do this because that's how the pins are configured in the hardware).

Next, on lines 11 to 15, we just do a bitwise operation to convert the channel number to binary by comparing the channel number with the value on the *mask* variable, which always starts at 1 and shifts left after every finished bitwise operation (binary shifting is basically adding, not the operation add, a binary 0 to the number and move the number/numbers to the left or right, e.g., if we have 1 and shift to left we get 10, if we shift again we get 100). After doing this operation we save the result in an array and use *digitalWrite()* to set the value on the pins A, B and C, of the mux, to the respective value converted (E.g., if we want to read the channel 5, this number converted to binary is 101, we need to send a signal to the pins A, B and C, which represents 101. Basically, we need to set pin A as *HIGH*, pin B as *LOW* and pin C as *HIGH*).

Finally, after selecting the desired channel, we just need to read the analog value of the common pin in the mux, with *analogRead()* function, which will be a number between 0 and 1023.

4.2.2 Programming the ADC via SPI Serial Interface

The ADC *MCP3008* was designed to use *SPI Serial Interface*, this means that if we try to programmatically control it without using SPI communication, it will lack performance by giving not so much accurate results. Before we start implementing the code to communicate with the ADC via SPI communication, we need to check the theory, in section 2.5.1, to understand what we need to do. Notice that we could also program this ADC to work with the *Arduino Software*, but since the other two boards already have analog ports, we only need to program it to work with the *RPi*.

To program the device to communicate via SPI, we need to use a special set of functions for it:

- **spiOpen(channel, baud rate, flags)** - this function returns a handler for the SPI device that we can use with other SPI functions later.
  - **channel** - the *RPi* has two SPI peripherals, the main and the auxiliary. The main can be used by all versions of the *RPi*, while the auxiliary only exist on “B+” versions. The main SPI has two chip selects (0-1, which relate to pins SPI0 CE0 and SPI0 CE0) and the auxiliary SPI has three chip selects (0-2, which relates to SPI1 CEX, where X is a number between 0 and 2).
  - **baud rate** - velocity of data transferred at baud bits per second. In this function we can use values between 32K and 135M, but we always need to check the ADC datasheet to know the ranging values we can use.
  - **flags** - are used to modify the default behavior of the device. Check *pigpio* library documentation to know the flags in more detail (pigpio, 2018).
- **spiXfer(handler, bufferWrite, bufferRead, countBytes)** - this functions transfers count bytes of data from *bufferWrite* to the SPI device and reads the data from the SPI device which is stored in *bufferRead*. The SPI device is associated to the handle created by *spiOpen()*.
- **spiClose(handler)** - closes the connection of the SPI device identified by the handler in *spiOpen()*.

Now that we know how the SPI connection works programmatically, we can put the theory into practice:

```
1  int analogRead(int channel)
2  {
3      int dataADC = 0;
4      int handlerSpi = 0;
5      char buffer[3];
6
7      if (channel > 7 || channel < 0)
8          return -1;
9
10     handlerSpi = spiOpen(0, 100000, 0);
```

```
11
12      buffer[0] = 1;
13      buffer[1] = (8 + channel) << 4;
14      buffer[2] = 0;
15
16      spiXfer(handlerSpi, buffer, buffer, 3);
17
18      dataADC = ((buffer[1] & 3) << 8) | buffer[2];
19
20      spiClose(handlerSpi);
21      return dataADC;
22  }
```

First, we need to check if the channel of the ADC ranges between 0 and 7, otherwise we can't read any value from it. Next, on line 10, we create the SPI device handler *handlerSpi* with *spiOpen()*. On lines 12 to 14, we need to create a data package which is represented in section 2.5.1 as "MCU Transmitted Data". The "MCU Transmitted Data" is package with 3 bytes of information, meaning each data package has 8 bits:

- The **first data package**, *buffer[0]*, is the starting bit. The last number of the package it's always 1 and the other ones are don't cares (they can either be 0 or 1). To facilitate our math we just assume they are 0 and construct the first package accordingly. Now, since all the other numbers are 0 and the most right bit is 1, we have 00000001 which corresponds to the first data package.

- The **second package**, *buffer[1]*, is the configuration mode plus the channel we want to read. To select the mode plus the channel we need to do the bitwise operation $(8+channel) << 4$. The first part of the operation it's a normal sum, because if we think of it as a bitwise operation, 8 in binary is 1000 and the number 1 represents the first bit of the second package, which is the mode we want to use. Next, we add the channel we want to read, for example, the number 5, which is in binary 101. If we add $8+5$ in binary, we have $1000+101 = 1101$. These bits correspond to the first 4 bits of the package, and the are don't cares. Again, to facilitate the math we say the don't cares are 0. Now, if

we binary shift this number 4 bits to the left, we get 11010000, which corresponds to the second package.

- The **last package**, *buffer[3]*, is represented by don't cares, so we assume they are all 0.

After constructing the package "MCU Transmitted Data" which is stored in the variable *buffer*, we just need to transfer the data to the SPI device with the function *spiXfer()*. After the communication is done, we just need to read the package "MCU Received Data", which is also stored in the *buffer* variable. Since our ADC has a 10-bit resolution, this means we are going to receive 10-bits of information, which are stored in the *buffer* variable. To read the data of the package we extract that information to the variable *dataADC* and we know the number there will range between 0 and 1023. To extract that information we do a bitwise operation between the "MCU Received Data" packages.

- The **first package**, *buffer[0]*, has seen in section 2.5.1, has unknown information, so we discard it.
- The **second package**, *buffer[1]*, only has information in the last 2 bits. To save that information we do $buffer[1]\&3$, in binary is the "AND" operation, and this way we can compare the last 2 bits. Don't forget that the value we want to read is the last package 8 bits plus these last 2 bits. So we shift the result we obtained from the "AND" operation 8 bits to the left $((buffer[1]\&3) << 8)$ and we get XX00000000, where X is the result from the "AND" operation.
- The **last package**, *buffer[2]*, has 8 bits of information, meaning the whole package contains relevant information. Now, we just need to do an OR operation with the shifted number. After comparing all bits, we get a 10-bit value of XXXXXXXXXX, where the first two X's are the result of the "AND" operation, and the other X's are the result of the "OR" operation.

After reading the analog value from the "MCU Received Data" package, we just need to close the SPI connection and return the value to the method.

4.2.3 Programming the *Ultrasonic Sensor HC-SR04*

In section 3.1.1, the theory about how the sensor works, we saw that the sensor works with two signals: the trigger and the echo. In order to generate a signal to detect an object, we need to set the trigger signal on a high state for $10\mu s$ and that will generate an 8 cycle burst signal, which will travel until it detects an object or gets out of range. The signal reflected by the object will travel to the echo pin. If we measure the time length of the signal received on the echo pin by using the equation 3.3, we can detect the distance of an object. Programmatically we have:

```
1  float distance()
2  {
3      float duration = 0.0, distance = 0.0;
4      digitalWrite(TrigPin, LOW);
5      delayMicroseconds(2);
6
7      digitalWrite(TrigPin, HIGH);
8      delayMicroseconds(10);
9      digitalWrite(TrigPin, LOW);
10
11     duration = pulseIn(EchoPin, HIGH);
12         distance = (duration / 2.0) * 0.0343 / 100.0;
13     if (distance > 0.02 && distance < 4.0)
14     {
15         return distance;
16     }
17     else if (distance < 0.02)
18     {
19         return distance = 0.02;
20     }
21     else
22     {
23         return distance = 4.0;
24     }
```

```
25 }
```

Starting on line 4, we set the trigger pin on a low state, for a duration of 2 microseconds, because we need to prevent that the trigger pin doesn't start on a high state and starts generating the 8 cycle burst signal before when we want to.

After this, on line 7, we start generating the 8 cycle burst signal by setting the trigger pin on high state for 10 microseconds, and after we are done generating the signal we set the trigger pin on low state.

On line 11, we use the function *pulseIn(pin, level)*, which returns the time length of the pulse in microseconds. The second parameter (*level*) of *pulseIn()* states that, for example and in our case, if the value is *HIGH*, it waits for the echo pin to go from *LOW* to *HIGH* and starts timing until the pin goes to *LOW* again (just like the graph of the figure 3.3), and then returns the time length of the pulse which is stored in the variable *duration*. After knowing the duration of the pulse, we just apply the equation 3.3 to get the distance of the object (in this method we divide the equation by 100 to get the value in meters, for centimeters we use another method).

Before we return the value of the *distance*, we just verify if the value is ranging between 2cm and 4m (min and max range of the sensor) just to make sure we don't get other random values.

4.2.4 Programming the *Analog Infrared Sensor Sharp GP2Y0A02YK0F*

As we have seen in section 3.1.2, to get the distance of an object using any infrared sensor we just need to find an equation that reflects the graph in the datasheet (figure 3.5). As already addressed, a model of the response of the sensor, is given by the equation $distance = voltage^{-0.98718} * 10.663$. By looking at this equation we know that we need to read the analog value of the analog sensor to get the distance of the object. If we remember from the theory, the analog value is number ranging between 0 and 1023 (ADC's have a 10-bit resolution) and we need to convert that number to a voltage. To do so, we need to use an equation that relates the ADC value to a voltage:

$$\frac{ADC\,Resolution}{System\,Voltage} = \frac{ADC\,Value}{Analog\,Reading\,Voltage} \tag{4.1}$$

The *ADC Resolution* is the max value the ADC can get, meaning if we have an ADC with 10-bit resolution we do $2^x - 1$, where x is the number of the bit resolution, and get the max number it can read. In our case we always get $2^{10} - 1 = 1023$. The *System Voltage* is the voltage of the operating circuit, which in the *Arduino UNO* board is 5V and on the other ones is 3.3V. The *ADC Value* is the number we get when using the function/method *analogRead()*. Finally, the *Analog Reading Voltage* is the voltage we want to know in the sensor equation. Now, if we rearrange the formula to know what is the voltage of the sensor, we have:

$$Analog\,Reading\,Voltage = \frac{ADC\,Resolution}{System\,Voltage} * ADC\,Value \tag{4.2}$$

```
float distance()
{
    float sensorValue = 0.0, voltage = 0.0, distance = 0.0;

#ifdef __AVR__
    sensorValue = analogRead(analogPin);
#elif defined(__RPi__)
    sensorValue = ADC.analogRead(channel);
#elif defined(ARDUINO_ESP8266_NODEMCU)
    sensorValue = mux.readChannel(channel);
#endif
    voltage = (SystemVoltage / ADCbit) * sensorValue;
    distance = pow(voltage, -0.98718) * 10.663 / 100.0;
    if (distance > 0.04 && distance < 0.35)
    {
        return distance;
    }
    else if (distance < 0.04)
    {
        return distance = 0.04;
    }
    else
```

```
23      {
24          return distance = 0.35;
25      }
26  }
```

Looking at the code, on lines 5 to 10, we can see that we have different ways of reading the analog value of the sensor. This happens because of the GPIO problems each board has.

In the case of the *Arduino* board, we can simply use the *analogRead()* function of the *Arduino Software*. For the *RPi*, since the ADC is external, we have to use the lib *mcp3008.h* and the method *analogRead()* of that class. Finally, for the *NodeMCU* board, if we are only using one sensor we could use the *analogRead()* function of the *Arduino Software*, but since we need to use multiple sensors in the robot, we need to use the lib *cd4051.h*, to use the multiplexer, and use the method *readChannel()* of that class, to read the channels where the sensors are connected.

After reading the analog value, we just need to apply it in equation 4.2 and use the voltage result in the infrared sensor equation (equation 3.7). Before returning the distance result, we just make sure the values range between the min and max sensor distance.

4.2.5 Programming the Motor Drivers

To control each motor through the motor drivers, we just need two things:

- A PWM signal to control the speed of the motors.
- A normal digital signal to rotate the motors, in the direction we want.

The other thing we need to know is if we are using the *Arduino Motor Shield*, which can only be used by the *Arduino UNO* board, we have 4 pins to control each motor, while the *L298 Dual H-Bridge*, which can be used by all boards, needs to use 6 pins to control each motor.

Programmatically we created the lib *motor.h*, which is able to control these two type of motor drivers. This lib has a main class called *Motor*, and two sub-classes called *Motor4* and *Motor6*, which either control the motor driver with 4 pins or the motor driver with 6 pins, respectively.

The sub-class *Motor4* can only be used by the *Arduino UNO* board. The main functions of this sub-class, which control each motor individually, can be seen programmatically has:

```
1  void rightMotor(int direction, int speed)
2  {
3      digitalWrite(RMDpin, direction);
4      analogWrite(RMSpin, speed);
5  }
6
7  void leftMotor(int direction, int speed)
8  {
9      digitalWrite(LMDpin, direction);
10     analogWrite(LMSpin, speed);
11 }
```

Each motor methods *rightMotor()* and *leftMotor()*, have two parameters, one for the direction and the other one for the speed. The direction of the motor is controlled by a digital signal, has we can see in lines 3 and 9, and because of it can only have two states, *LOW* or *HIGH*. The high state rotates the motor forward, and the low state rotates the motor backward. Now, by using the methods *rightMotor()* and *leftMotor()*, we can propel the robot in whatever direction we want, either be it forward or backward, or left or right.

To control the speed of the motor, is pretty simple, we just choose a value between 0 (always off) and 255 (always on), and that will generate a PWM signal. If we want better control of the speed of the motor, we can choose the value of the speed has $255 * duty\,cycle(\%)$, and by using this method we can use the equation 3.8 to control the duty cycle of the PWM signal applied.

The sub-class *Motor6*, works in a similar way as the sub-class *Motor4*. The big difference is that the motor with 6 pins needs to use 2 digital signals to control the direction of the motor, instead of one. Programmatically we have:

```
1  void rightMotor(int direction, int speed)
2  {
3      digitalWrite(RMD1pin, direction);
4      digitalWrite(RMD2pin, !direction);
```

```
5        analogWrite(RMSpin, speed);
6    }
7
8    void leftMotor(int direction, int speed)
9    {
10       digitalWrite(LMD1pin, direction);
11       digitalWrite(LMD2pin, !direction);
12       analogWrite(LMSpin, speed);
13   }
```

As we can see, in lines 3-4 and 10-11, to control the direction in both motors, one direction pin needs to be on a high state and the other one needs to do the inverse, which is to be in a low state. For example, if we want to move forward the first direction pin needs to be on *HIGH*, while the second direction pin needs to be on *LOW*. The rest works the same has the other motor driver.

The control of the speed works the same way as the other driver, we just need to generate a PWM signal. The difference is that this driver can be controlled by all dev boards, and the values that we choose to generate the PWM signal aren't the same between all boards. To generate a PWM signal with the boards *Arduino UNO* and *RPi*, we need to use values that range between 0 (always off) and 255 (always on), while with the *NodeMCU* board we need to use values between 0 (always off) and 1023 (always on). This problem has to do with how *analogWrite()* function is programmed in the *Arduino Software*, because to generate a PWM signal, the "analog value" that we want to write in the device must be a value between 0 and the ADC resolution. Both *Arduino UNO* and *NodeMCU*, have an ADC with a 10-bit resolution, but the way that function works with the *Arduino* is different from the *NodeMCU*, it's like they are mapping the values of an ADC with 10-bit resolution to work like an ADC with 8-bit resolution. On the *RPi*, the PWM signal is generated by software and not by hardware. And the function that was created to generate a PWM signal on the *RPi* only allows values between 0 and 255.

4.2.6 Using the Servo Motors

As we have seen in chapter 3.2.2, we have two types of servo motors and we need to send a PWM signal for control. To be able to control them programmatically we need to use the *Servo.h* library (already comes pre-installed with *Arduino Software*) and *RPiServo.h* (was created based on the *Arduino* library and *pigpio* framework). In both libraries we use 3 important functions to control them:

- ***attach()*** - connects the servo to a GPIO pin.
- ***write()*** - writes a value to the servo, controlling the shaft accordingly. On a standard servo, this will set the angle of the shaft (in degrees), moving the shaft to that orientation. On a continuous rotation servo, this will set the speed of the servo (with 0 being full-speed in one direction, 180 being full speed in the other, and a value near 90 being no movement). (Arduino, 2018c)
- ***writeMicroseconds()*** - writes a value in microseconds (μS) to the servo, controlling the shaft accordingly. On a standard servo, this will set the angle of the shaft. On standard servos a parameter value of 1000 is fully counter-clockwise, 2000 is fully clockwise, and 1500 is in the middle. Continuous-rotation servos will respond to the *writeMicroseconds()* function in an analogous manner to the *write()* function. (Arduino, 2018d)

5 Case studies

In this chapter we are going to see how the framework works when used to program a wheel robot that can change it's core controlling board, without suffering many changes in the programming code. Every library used by the framework and also the code source files of both case studies are presented in *Appendix A*.

For this thesis, we are going to show two different case studies. These two different case studies are:

- On the **first case**, we are going to use all three development boards to program a robot that does an automatic path-finding. This means the robot will try to find a path to freely move, on its own, while avoiding obstacles on its way.
- On the **second case**, we are only using the *NodeMCU ESP8266* and the *Raspberry Pi 3* boards, because they are the only ones that have a WiFi connection. In this case study we are going to use an Android application which is able to control both robots via a WiFi connection. The robot will have two modes: an automatic path-finding, which is the same as the first case study; and a manual mode, where the user is able to manually control the direction he wants the robot to move.

The design of the robot will be the same in both case studies, meaning that the only thing that changes are the dev boards, which control the robot. The robot, in both case studies, will consist of:

- **Chassis:** Devastator Tank Mobile Robot Platform
- **Motors:** 2x Micro DC Gear Motor with Back Shaft, one to move each wheel.
- **Motor Driver:** Itead L298 Dual H-Bridge
- **Wheels:** 2x Continuous tracks, basically they are like mini tank wheels.
- **Sensors:** 3x Ultrasonic Sensor HC-SR04, to detect the objects
- **Development boards:** *Arduino UNO Rev3*, *NodeMCU ESP8266* and *Raspberry Pi 3 B+*.

- **Battery**: 5x AA rechargeable batteries
- **Power switch,** which turns the robot on or off.
- **Logic Level Converter,** for the boards *NodeMCU ESP8266* and *Raspberry Pi 3 B+*.

As we can see in the list above, the only things that change are the development boards and the logic level converter which isn't used in the *Arduino UNO* board. With this in mind we can say that the connections between the dev boards and the devices (motor driver and sensors) are pretty much the same. The only difference being where we connect those devices in each board GPIO pins. Since all the devices use digital signals, we can easily connect them in all boards without having the need to use the ADC or the multiplexer. In the next figures, we can see how to connect all devices to each GPIO pins of each development boards to assemble the robot.

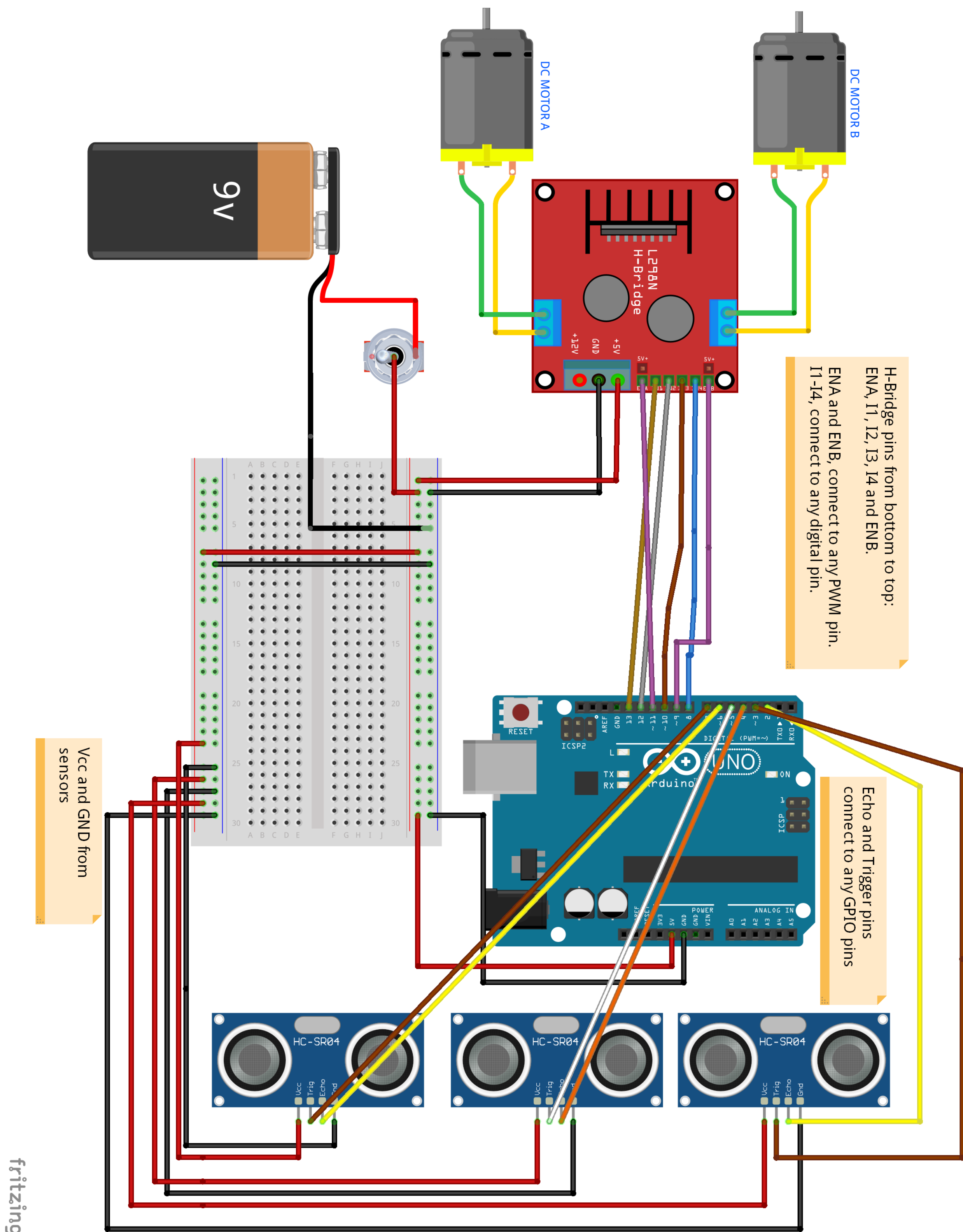


Figure 5.1: Robot connection diagram using *Arduino UNO* board.

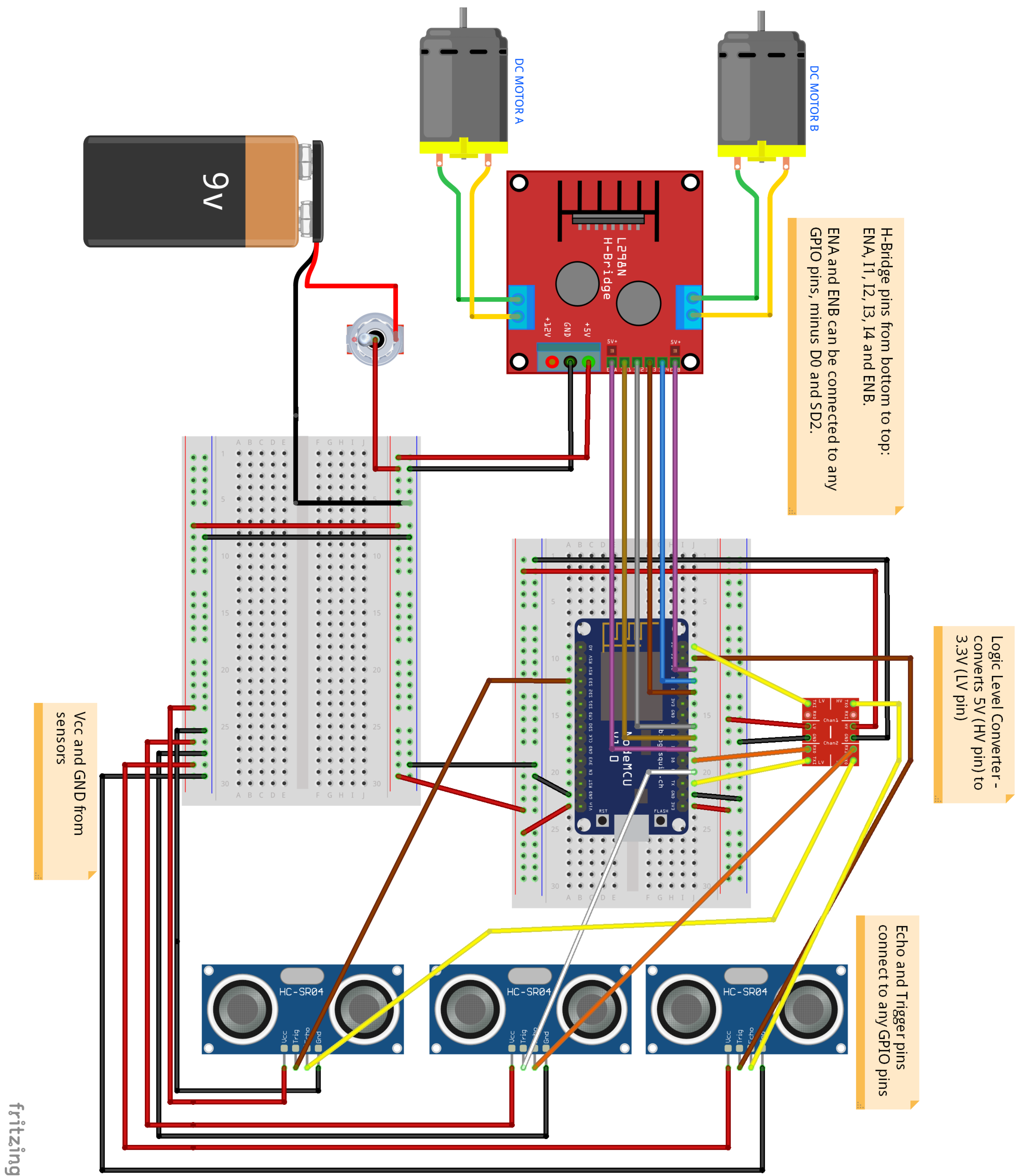


Figure 5.2: Robot connection diagram using *NodeMCU ESP8266* board, with the addition of the logic level converter.

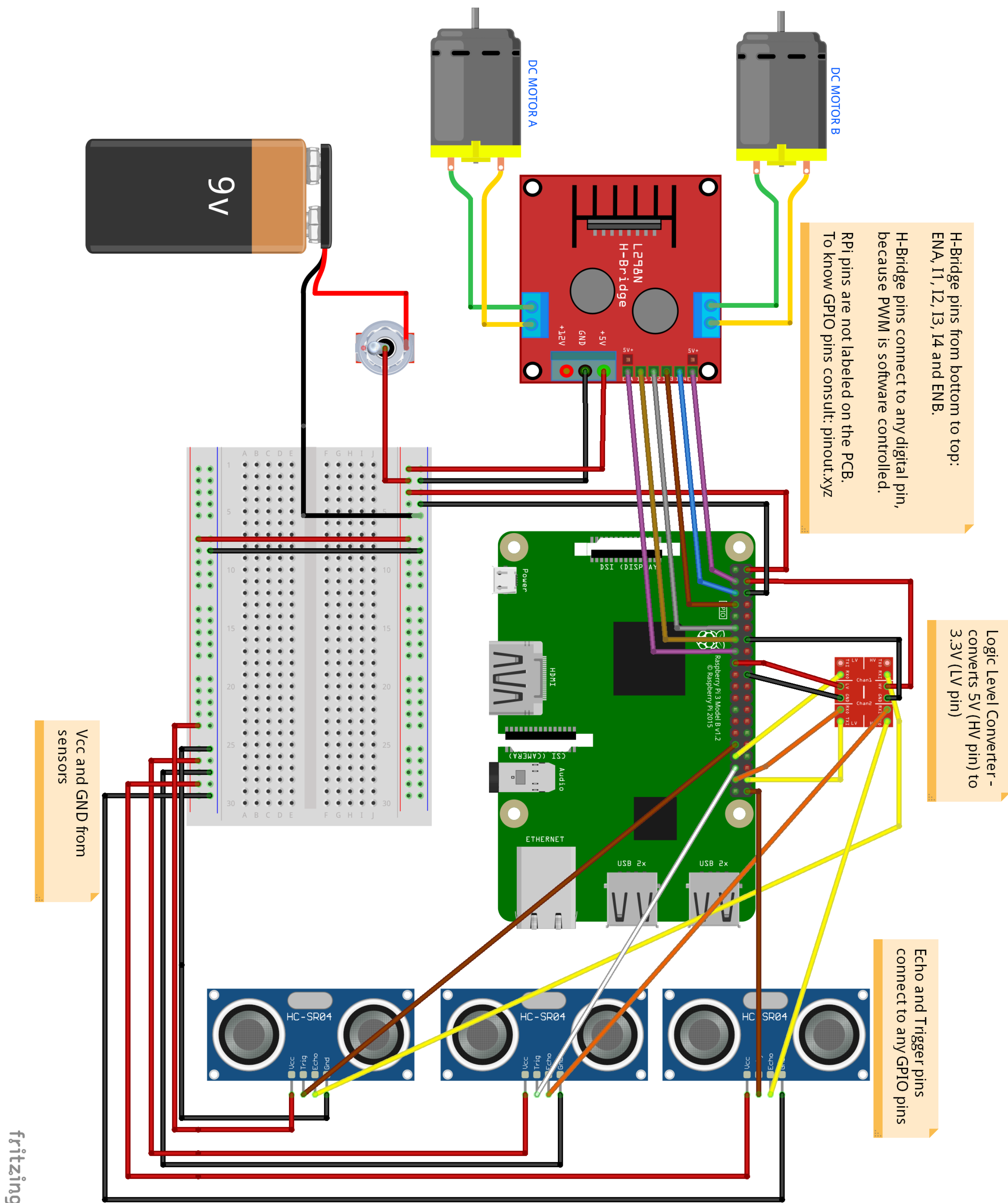


Figure 5.3: Robot connection diagram using *Raspberry Pi 3 Model B+* board, with the addition of the logic level converter.

Finally, we can see how the robot looks when fully assembled.

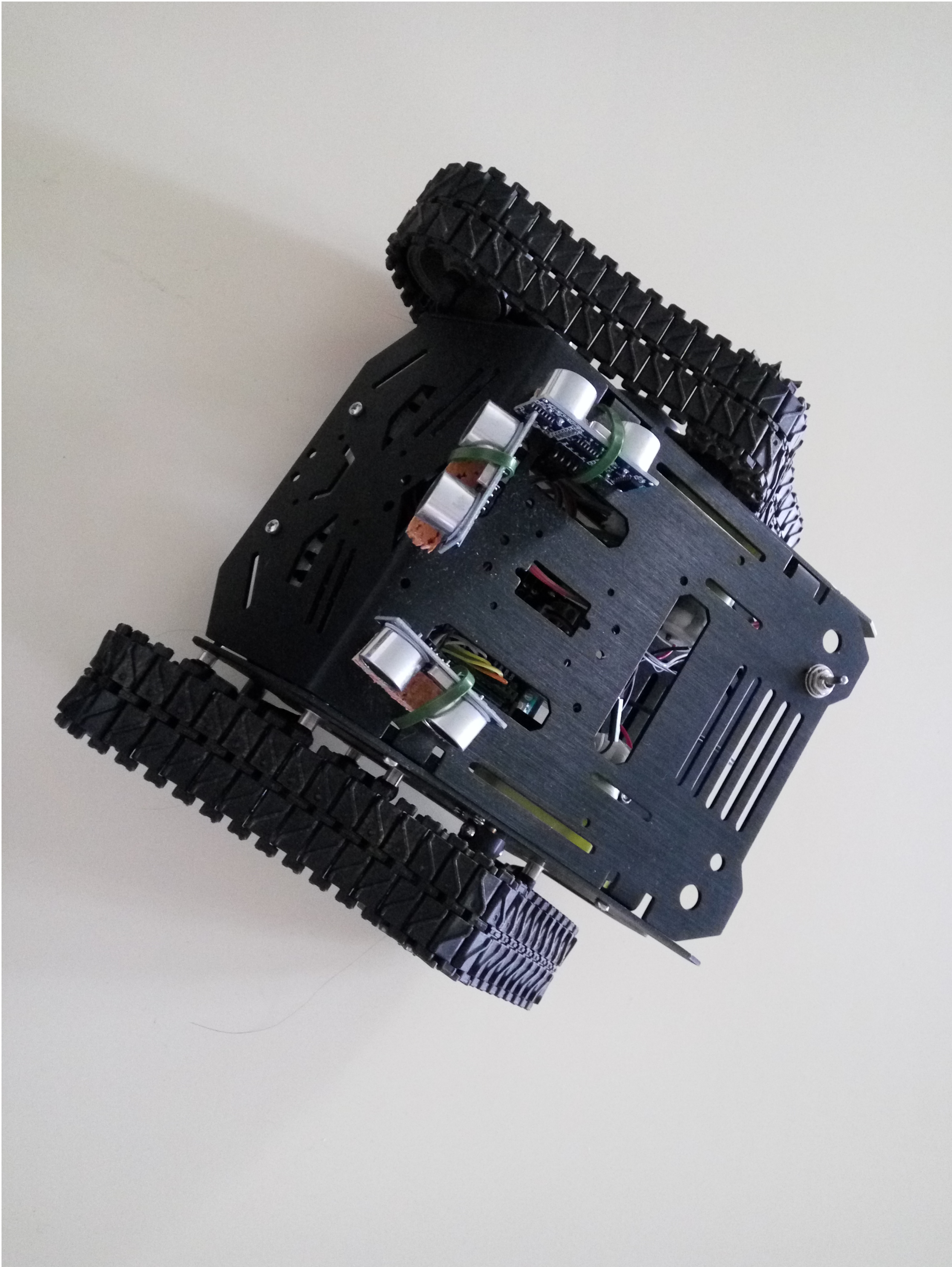


Figure 5.4: Fully assembled robot in the chassis with all devices connected to any development board.

Figure 5.5: Front view of the assembled robot.

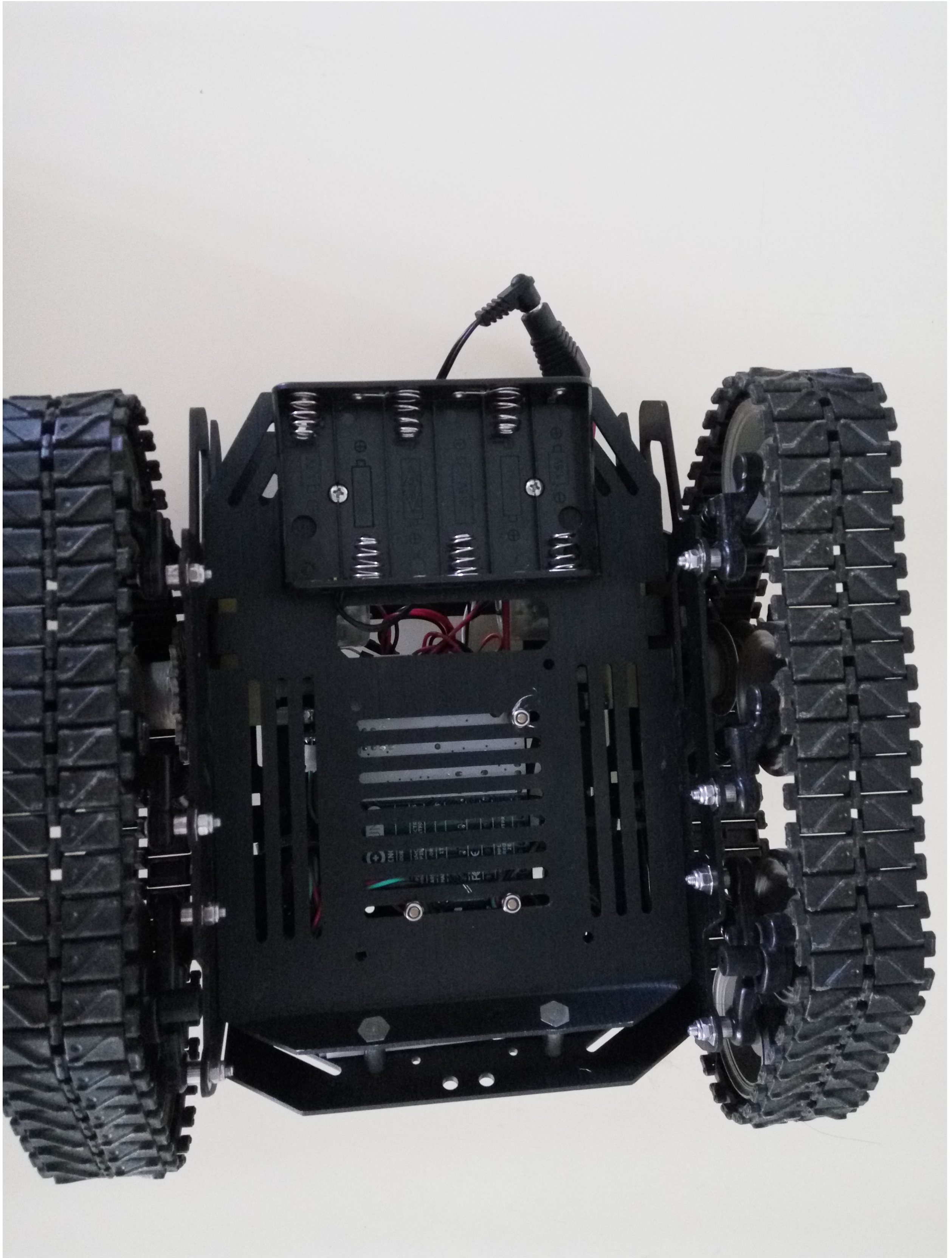


Figure 5.6: Bottom view of the robot where the battery is.

5.1 Case study 1: Automatic path-finding while avoiding obstacles

The objective of this case study is to program the robot for it to move autonomously while avoiding obstacles in its way. To view the full code of case study 1, see appendix A file *robot-NoWifi.ino*.

To do this, first we need to use an algorithm for the robot to move and avoid objects. By using the framework it's pretty simple, we need to create an object of the class *Motor6()* to control the 6 pins (2 pins for direction and 1 pin for the speed) of the motor driver. To move the robot using this class, we can use the methods *moveForward()*, *moveBackward()*, *turnLeft()*, *turnRight()* and *stop()*. Now, to detect objects using the ultrasonic sensors, programmatically we need to create create an object of the class *Sonar()* and use the method *distance()*, which returns the distance of an object. Putting these two classes together we can create the algorithm, for the robot move autonomously and avoid any detected obstacles, as we can see below.

```
1  void loop()
2  {
3      // put your main code here, to run repeatedly:
4      int left = ping(leftSonar);
5      int front = ping(frontSonar);
6      int right = ping(rightSonar);
7
8      //Emergency stop
9      if (right < stopDist && left < stopDist)
10     {
11         motor.stop();
12         exit(1);
13     }
14
15     motor.moveForward(speed);
16
```

```
17        if (right < minDist)
18        {
19            motor.turnLeft(speed);
20            delay(detourTime);
21        }
22
23        if (front < minDist)
24        {
25            motor.moveBackward(speed);
26            delay(detourTime * 3);
27            motor.turnLeft(speed);
28            delay(detourTime * 2);
29        }
30
31        if (left < minDist)
32        {
33            motor.turnRight(speed);
34            delay(detourTime);
35        }
36    }
```

By following the algorithm above, what the robot does is ping the three sensors, which are positioned left, front and right, and when each sensor detects an object it will check if the minimum distance between the sensor and the object is less than 20cm (the value of **minDist** variable) and if it is, it will act accordingly. If the right sensor detects a near object the robot will move left, if the left sensor detects a near object it will move right and if the front sensor detects a near object it will move backward and turn left. In case both left and right sensors are 5cm close to an object, the robot will do an emergency stop and stop the motors from moving. After this emergency stop, for the robot to be able to move again the user needs to manually reposition the robot.

5.2 Case study 2: Robot control over a WiFi connection

In this case study, we are going to use only two development boards, the *NodeMCU* and the *Raspberry Pi*. Since the objective of this case study is to use the WiFi module of the development boards, we couldn't use the *Arduino UNO* board because we didn't have any WiFi shield or module. To see the code of this case study and the code of the android application, see appendix A (the file which contains the code for the robot is *robotWifi.ino*).

The objective of this case study is to connect the robots to an android application, via a WiFi connection, that allows changing between two control modes.

- The **first mode**, is an automatic path-finding, which is the same used in case study 1.
- The **second mode**, the user can manually control the direction which the robot moves. It's the same principle has having a remote control to choose the direction, but instead the direction control uses a smartphone and an android application, specially developed for these purpose.

We can think of this case study like an upgrade of the other one. Sometimes in the automatic mode the robot can do an emergency stop and afterwards will have to be manually re-positioned by the user. In this case, the user instead of having to go to the robot's place, he can just change the mode of the robot to manual and move it to whatever place he wants to.

To use the android application to control the robot is very simple. In the main screen, figure 5.7, we need to input the robot *IP* and *Port* to connect to the robot, via a WiFi connection. If the connection is successful, the button *Automatic* and *Manual* will be enabled, meaning the robot is ready and the user can choose a mode. In automatic mode, the robot will move on its own while avoiding obstacles in it's way, and will only stop on an emergency stop and/or when we press the *QUIT* button, to go back to the main screen and choose the other mode. When using the manual mode, figure 5.8, the user will have the option to move the robot forward (*UP* button), backward (*DOWN* button), left (*LEFT* button) or right (*RIGHT* button). As soon as we press a button, the robot wont stop until another button is pressed, or the *STOP* button is pressed, which will make the robot stop on the spot. For example, if we press the *UP* button the robot will move forward until we press another button, if the robot is moving forward and we

press the *LEFT* button, the robot will stop and rotates anti-clock wise, in a 360º rotation, until we press another button, and so on. If we want to quit the manual mode, just like the automatic mode, we press the *QUIT* button to go to the main screen to choose another mode.

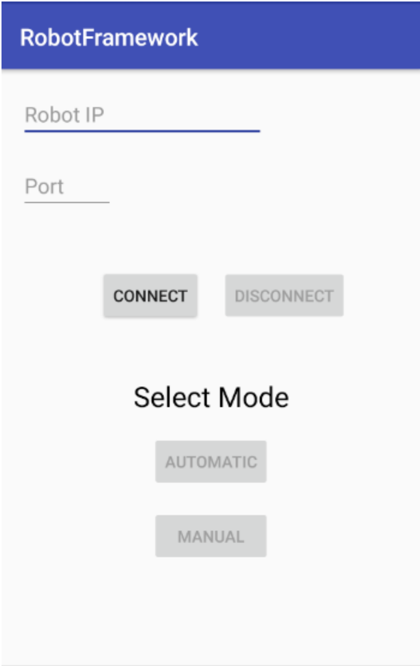
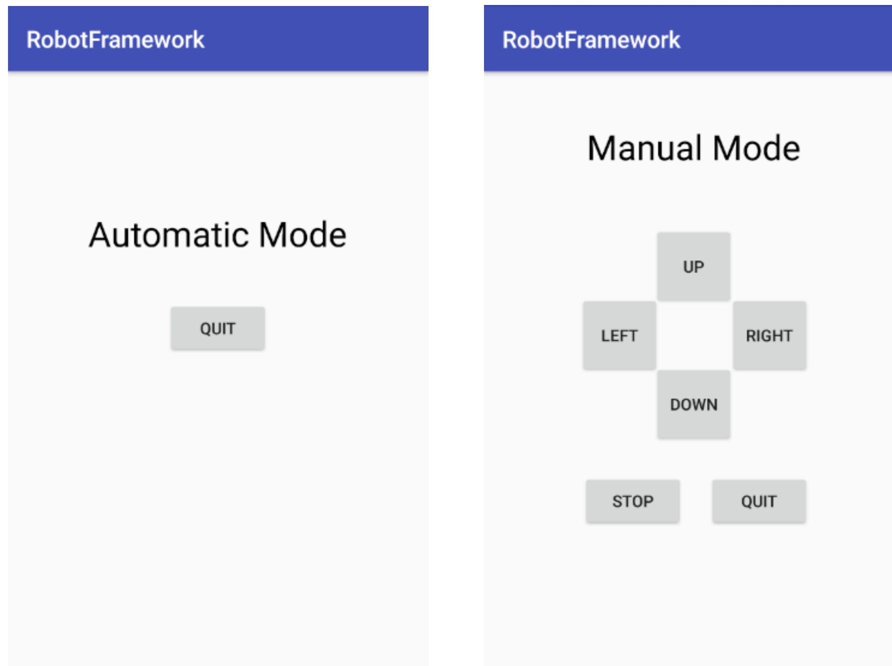


Figure 5.7: Main screen of the android application that controls the robot.



(a) Automatic mode screen. (b) Manual mode screen.

Figure 5.8: Android application screens after selecting a mode.

When we look at the code, file *robotWifi.ino* in appendix A, that's going to be used by the two development boards, we can see that the algorithms to automate and manually control the robot, doesn't suffer any changes. But there is a big difference in the code, when we want to connect to a WiFi network, as we can see below.

```
1  #ifdef __RPi__
2  Motor6 motor(2, 3, 4, 17, 27, 22, 0.95, 1.0);
3  Sonar leftSonar(25, 8), frontSonar(23, 24),
4  rightSonar(15, 18);
5  #endif
6
7  #ifdef ARDUINO_ESP8266_NODEMCU
8  const char *ssid = "";
9  const char *password = "";
10 Motor6 motor(4, 13, 0, 14, 2, 12, 0.95, 1.0);
11 Sonar leftSonar(16, 5), frontSonar(15, 3),
```

```
12 rightSonar(1, 10);
13 #endif
```

As we can see, the GPIO pins of the motors and sensors will always be different but we can't change that, since it's an hardware problem related to the design of the boards. The big difference will be when we want to use the WiFi connection on the *NodeMCU* compared to the *RPi*. Since the *NodeMCU ESP8266* is a microcontroller, to program that board to connect to a WiFi network, we always need to do it in the code of the program we want to upload. To do so, on lines 8 and 9, we declare the *ssid* and *password* variables, and later on we use the function *WiFi.begin(ssid, password)* to connect to that WiFi network. Because the *RPi* is a micro-computer which uses a Linux OS based, if we want to program on that board, we either need to be already connected via WiFi and use a SSH connection, or we need to manually use it as a computer, and in this case since we are already using it as a computer, we can just manually connect it to the WiFi network just like any other computer.

Summarizing, if we are using a microcontroller and want to use any shield or module to connect to a WiFi network, we always need to do it by programming code. When using a micro-computer like the *RPi* board, we need to manually connect it like we do it on any computer.

6 Conclusions and Future Work

6.1 Conclusions

Looking back at both case studies in chapter 5, we can conclude that if we want to program a software framework that specifically utilizes different development boards to control a robot, unless we programmatically treat all GPIO connections equally, to specifically be the same across all development boards, there must be low level differences that the framework must handle. Since all development boards are designed differently and also work differently, from each other, in terms of hardware the GPIO connections are unlikely to be equal. Even though we knew that beforehand, it was decided that if a developer wants to connect some device to a development board, it should always be able to choose which GPIO pin he wants to connect to, meaning that the user can be limited by hardware but should not be limited by software. Thus the framework must allow us to chose at the application level the GPIO pins required to connect the devices, and part of the code is most likely to be different for different boards. Anyway its a configuration issue that can be handled simply with conditional compilation directives, leaving the rest of the code of the application equal across platforms.

As we saw in case study 2, the code of the microcontroller is a bit different than the code in the microcomputer. This problem is persistent because of how they were designed to work internally. And because of this, to code a framework to equally work with any development board in the same language, becomes an exhaustive and hard task when we need to address every single issue we encounter. To program a microcontroller we usually need to use a piece of software that is already prepared for those types of boards, just like using the *Arduino* boards or the *NodeMCU* with *Arduino Software*. These basic development environments already integrate a compiler and a bunch of preset libraries that allow the control of a microcontroller through the programming language they were designed for, which makes the programming in the microcontrollers very easy to use when compared with microcomputers. The microcomputers were designed to work like an actual computer, meaning being all purpose devices and not mainly designed for digital control. One of the differences between an actual computer like the *RPi* and a microcontroller is that they simply were attached a GPIO pin extender to it's PCB. This makes us have to choose a programming language that have support for the GPIO pins, or we need to

add some support for the GPIO to the language, either with a third party software module or developed by us. In this thesis that wasn't the case, because we actually found a framework, for the *RPi* that was already prepared to access the GPIO pins from the C programming language, which we then can use with the C++ language.

Also, before we start programming in any development board we must choose one that meets our needs. For instance, if we want to use multiple digital devices and multiple analog devices, we are better off choosing a board that has enough analog and digital pins for the current/future project, or if we want to only use digital devices we better get a board with a lot of digital pins. With this in mind, later on, we don't need to resort to external hardware that will eventually complicate the programming of that development board.

That was the case of some of the boards that these framework supports, where it was needed to add external multiplexers and ADCs. However the framework was extended to handle them as it was shown.

6.2 Future Work

For future development of this project, this software framework can be extended to work with more devices and other projects, rather than just 2 wheels like robots. Also the framework can be used to implement more advanced applications of wheeled robots, than the 2 study cases presented in this thesis. Such projects must benefit from the use of new development boards, new sensors and other important electronic devices. Since this software framework was developed to be programmed with C/C++ language, the future researcher must be wary of this restriction, and take that into account when choosing a new development board for the new project.

References

Arduino (Ed.). (2018a). *Arduino uno wifi rev2.* Retrieved from https://store.arduino.cc/arduino-uno-wifi-rev2

Arduino (Ed.). (2018b). *Servo library.* Retrieved from https://www.arduino.cc/en/reference/servo

Arduino (Ed.). (2018c). *Servowrite.* Retrieved from https://www.arduino.cc/en/Reference/ServoWrite

Arduino (Ed.). (2018d). *Servowritemicroseconds.* Retrieved from https://www.arduino.cc/en/Reference/ServoWriteMicroseconds

arduino.cc (Ed.). (2018a). *Arduino.* Retrieved from https://www.arduino.cc/

arduino.cc (Ed.). (2018b). *Arduino uno rev3.* Retrieved from https://store.arduino.cc/arduino-uno-rev3

arduino.cc (Ed.). (2018c). *Language reference.* Retrieved from https://www.arduino.cc/reference/en/

arduino.cc (Ed.). (2018d). *What is arduino?* Retrieved from https://www.arduino.cc/en/Guide/Introduction

CytronTechnologies (Ed.). (2013). *Hc-sr04 user manual.* Retrieved from http://web.eece.maine.edu/~zhu/book/lab/HC-SR04%20User%20Manual.pdf

DFRobot (Ed.). (2018). *Devastator tank mobile robot platform.* Retrieved from https://www.dfrobot.com/product-1219.html

ElectronicWings (Ed.). (2017). *Pwm in avr atmega16/atmega32.* Retrieved from http://www.electronicwings.com/avr-atmega/atmega1632-pwm

Espressif. (2018). *Esp8266ex datasheet.* Retrieved from https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

Grusin, M. (2018). *Serial peripheral interface (spi).* Retrieved from https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

ISLProductsInternational (Ed.). (2018a). *Dc motor / dc gear motor basics.* Retrieved from https://www.islproducts.com/designnote/dc-motor-dc-gearmotor-basics/

ISLProductsInternational (Ed.). (2018b). *Servo motor fundamentals.* Retrieved from https://www.islproducts.com/designnote/servo-motor-fundamentals

Jameco (Ed.). (2018). *How do servo motors work?* Retrieved from https://www.jameco.com/jameco/workshop/howitworks/how-servo-motors-work.html

kaaproject.org (Ed.). (2018). *Iot 101: What is an iot platform?* Retrieved from https://www.kaaproject.org/what-is-iot/

Microchip (Ed.). (2018). *2.7v 4-channel/8-channel 10-bit a/d converters with spi serial interface.* Retrieved from https://cdn-shop.adafruit.com/datasheets/MCP3008.pdf

Nodemcu connect things easy. (2018). Retrieved from http://nodemcu.com/index_en.html

Nodemcu devkit v1.0. (2018). Retrieved from https://github.com/nodemcu/nodemcu-devkit-v1.0

opensource.com (Ed.). (2018). *What is a raspberry pi?* Retrieved from https://opensource.com/resources/raspberry-pi

pigpio (Ed.). (2018). *The pigpio library.* Retrieved from http://abyz.me.uk/rpi/pigpio/

raspberrypi.org (Ed.). (2018a). *Noobs.* Retrieved from https://www.raspberrypi.org/documentation/installation/noobs.md

raspberrypi.org (Ed.). (2018b). *Raspberry pi 3 model b+.* Retrieved from https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

RobotRus (Ed.). (2018). *Sharp gp2y0a41sk0f analog distance sensor 4-30cm.* Retrieved from https://www.robot-r-us.com/sensor-infrared/sharp-gp2y0a41sk0f-analog-distance-sensor-4-30cm.html?keyword=GP2Y0A41SK0F

SHARP (Ed.). (2018). *Sharp gp2y0a41sk0f.* Retrieved from https://www.pololu.com/file/0J713/GP2Y0A41SK0F.pdf

TexasInstruments (Ed.). (2017). *Cd405xb cmos single 8-channel analog multiplexer/demultiplexer with logic-level conversion.* Retrieved from http://www.ti.com/lit/ds/symlink/cd4052b.pdf

Vidarte, L. (2017). *Esp8266 nodemcu workshop.* Retrieved from https://github.com/lvidarte/esp8266/wiki/NodeMCU

Yuan, M. (2017). *Getting to know nodemcu and its devkit board.* Retrieved from https://www.ibm.com/developerworks/library/iot-nodemcu-open-why-use/index.html

Appendix A - Framework and Case studies

To access the full code of each library created to be used by the framework and the full code of both case studies, please refer to “case studies” and “lib” folders, inside “robotFramework” folder in the digital support (CD-ROM).