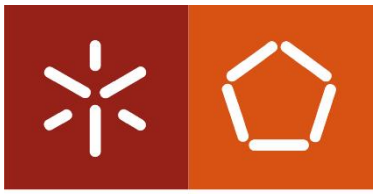

Universidade do Minho

CENTROALGORITMI

T.W.C.V

Three Ways Controlled Vehicle

Project Design

Embedded Systems

MIEEIC

2016/2017

Authors

Ailton Lopes - A72852

Nuno Afonso - A65354

Professor:

Adriano Tavares

José Mendes

Index

Figure Index

Introduction

1. Motivation

The number of people around the world that requires the help of artificial means to move around it's continually growing and according to the website ***kdsmartchair.com*** every year there are 2 million new wheelchair users. The purpose of this project is to develop a system targeted for peoples who are paralyzed, and are unable to control parts of their body enough to physically control a traditional wheelchair.
In this project, it's going to be Designed and implemented a system capable tracking the direction of hand movements, which is intended to be used as a human-robot interaction interface for controlling the vehicle that we pretend control.

2. Aims

The main purpose of the project is to design and implement an embedded system capable of tracking the movements of a hand in real-time using inertial measurement unit sensors (accelerometer) to control a robot that simulate the wheelchair. To avoid accidents, the robot it's going to have sensors to detect objects and be able to stop itself in when near to obstacles.
the system will be able to recognize at least 6 movements:

- Move Forward
- Move Backyard
- Move Right
- Move Left
- Stop

To detect the hand position, it's used a Raspberry PI that gets the data from the accelerometer sensor and according to the angle measured by the accelerometer it´s detected the exactly position of the hand and measuring the hand position continually we can detect the direction of the hand motion. Once the movement it's detected the Raspberry PI attached to the sensors sends instructions to the Raspberry PI on the robot about which movement to perform. The raspberry pi on the robot control the robot motors and the obstacle detections sensors.
The two Raspberry PI will use wireless (Wi-Fi) communication to exchange data with each other.

Theoretical Concepts

1. Waterfall Development Process

The development of this project follows the Waterfall Development Process.
The waterfall model is a sequential design process, used in software development processes in which progress is seen as flowing steadily downwards through the phases of requirements, design, implementation, verification and maintenance. Despite the development of new software development process models, the Waterfall method is still the dominant process model with over a third of software developers still using it. The Waterfall model was first defined by Winston W. Royce in 1970 and has been widely used for software projects ever since.

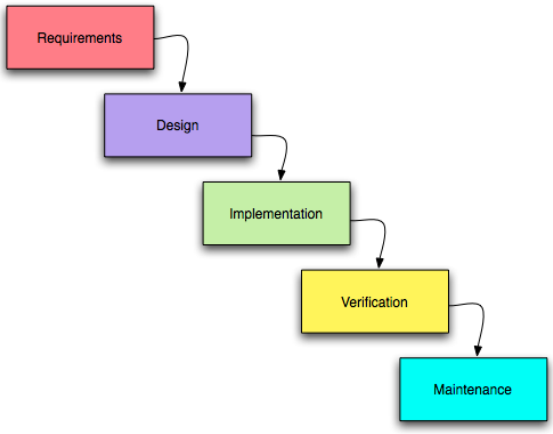

Figure 1: Waterfall Development Process Phases

Although the process flow is downwards in any moment the developers can go back to a previous stage if necessary, for example in case of detecting an error, etc.
Advantages of using the Waterfall model:

- Design errors are captured before any software is written saving time during the implementation phase.
- Excellent technical documentation is part of the deliverables and it is easier for new programmers to get up to speed during the maintenance phase.
- The approach is very structured and it is easier to measure progress by reference to clearly defined milestones.
- The total cost of the project can be accurately estimated after the requirements have been defined (via the functional and user interface specifications).
- Testing is easier as it can be done by reference to the scenarios defined in the functional specification.

2. Speech recognition

Speech recognition (SR) is the subsystem of computational linguistics that develops methodologies and technologies that enables the recognition and translation of spoken language into text by computers. It is also known as "automatic speech recognition" (ASR), "computer speech recognition", or just "speech to text" (STT). It incorporates knowledge and research in the linguistics, computer science, and electrical engineering fields.
The speech recognition models can be classified in two types:

Speaker dependent – SR models that use "training", where an individual speaker reads text or isolated vocabulary into the system. The system analyzes the person's specific voice and uses it to fine-tune the recognition of that person's speech, resulting in increased accuracy. Systems that do not use training are called **speaker independent**.
Speech recognition is a very important from the technology perspective and recently the field has benefited from advances in deep learning and big data.
These speech industry companies include Google, Microsoft, IBM, Baidu, Apple, Amazon, Nuance, SoundHound, IflyTek, CDAC many of which have publicized the core technology in their speech recognition systems as being based on deep learning.

Speech Recognition engines available for the Raspberry Pi.

1. libsprec

A speech recognition library that is developed by H2CO3 (with few contributions by myself, mostly bug fixes).

Pros: It uses the Google Speech API, making it more accurate. The code is more easy to understand (in my opinion).

Cons: It has dependencies on other libraries that H2CO3 has developed (such as libjsonz). Development is spotty. It uses the Google Speech API, meaning processing doesn't take place on the Raspberry Pi itself, and requires an internet connection. It requires one small modification to the source code before compilation to work properly on the Raspberry Pi.

2. Julius

Julius is a high-performance, two-pass large vocabulary continuous speech recognition (LVCSR) decoder software for speech-related researchers and developers. Based on word N-gram and context-dependent HMM, it can perform almost real-time decoding on most current PCs in 60k word dictation task. Major search techniques are fully incorporated such as tree lexicon, N-gram factoring, cross-word context dependency handling, enveloped beam search, Gaussian pruning, Gaussian selection, etc. Besides search efficiency, it is also modularized carefully to be independent from model structures, and various HMM types are supported such as shared-state triphones and tied-mixture models, with any number of mixtures, states, or phones. Standard formats are adopted to cope with other free modeling toolkit such as HTK, CMU-Cam SLM toolkit, etc.

The main platform is Linux and other Unix workstations, and also works on Windows. Most recent version is developed on Linux and Windows (cygwin / mingw), and also has Microsoft SAPI version.

Features

- An open-source software (see terms and conditions of license)

- Real-time, hi-speed, accurate recognition based on 2-pass strategy.
- Low memory requirement: less than 32MBytes required for work area (<64MBytes for 20k-word dictation with on-memory 3-gram LM).
- Supports LM of N-gram, grammar, and isolated word.
- Language and unit-dependent: Any LM in ARPA standard format and AM in HTK ascii hmmdefs format can be used.
- Highly configurable: can set various search parameters. Also, alternate decoding algorithm (1-best/word-pair approx., word trellis/word graph intermediates, etc.) can be chosen.
- Full source code documentation and manual in Engligh / Japanese.
- List of major supported features:
- On-the-fly recognition for microphone and network input
- GMM-based input rejection
- Successive decoding, delimiting input by short pauses
- N-best output
- Word graph output
- Forced alignment on word, phoneme, and state level
- Confidence scoring
- Server mode and control API
- Many search parameters for tuning its performance

Pros: It can perform almost real-time speech recognition on the Raspberry Pi itself. Standard speech model formats are adopted to cope with other free modeling toolkits.
Cons: Spotty development. The recognition is also too inaccurate and slow for my usage. Long installation time

3. **Pocketsphinx** - PocketSphinx is a speech recognition engine, specifically tuned
for handheld and mobile devices, though it works equally well on the desktop, it can be used in embedded systems (e.g., based on an ARM processor).

Features:

- State of art speech recognition algorithms for efficient speech recognition. CMUSphinx tools are designed specifically for low-resource platforms
- Flexible design
- Focus on practical application development and not on research
- Support for several languages like US English, UK English, French, Mandarin, German, Dutch, Russian and ability to build models for others
- BSD-like license which allows commercial distribution
- Commercial support
- Active development and release schedule
- Active community (more than 400 users on Linkedin CMUSphinx group)
- Wide range of tools for many speech-recognition related purposes (keyword spotting, alignment, pronunciation evaluation)

Pros: Under active development and incorporates features such as fixed-point arithmetic and efficient algorithms for GMM computation. All the processing takes place on the Raspberry Pi, so it is capable of being used offline. It supports real time speech recognition

Cons: It is complicated to set up and understand for beginners. All the processing takes place on the Raspberry Pi, making it a bit slower.

The Speech recognition engine used in this project is the **PocketSphinx** for being the one more suitable for the Raspberry Pi and because the entire recognition process happens in the board. More information about PocketSphinx can be found in the CMU Sphinx website.

Design Phase

1.Requirements

The Requirements is the first phase of the Waterfall model.
In this section, it's going to be determined the goals, functions and constraints of the software system and the representation of this aspects in forms agreeable to modeling the analysis. The goal is to create a list of requirements that can provide insight to the costumers to make sure the product under development meet their needs and expectation and at the same time gives the developer a complete representation of functions and constraints of the system.

1.1. Functional requirements

- Perform the commands (move forward, move backward, turn right and turn left) sent by the user from the desktop application or de controller system.

- The robot system is able to work without the controller system receiving the commands from the Remote desktop system.

- The robot system is able to work without the Remote desktop system (in this case there is no feedback from the robot to the user about the robot status).

- Gives the user the possibility of choosing the robot controller mode (hand or head movements, voice commands or through the desktop system application)

1.2. Non-Functional Requirements

- High performance.
- Easy to upgrade adding others control mechanisms.
- Friendly user interface.
- The controller system is wearable
- Low cost, low power consumption and high performance.

2. Constraints

In this project, there are two different types of constraints: **Technical Constraints** and **Non-Technical Constraint** specifying the rules and limitations in the project development

2.1. Technical Constraints

- Use the cross-compiling toolchain provided by Buildroot.
- Use the C++ language to the development of the application
- Use the board Raspberry Pi.
- Use embedded Linux operating system.

2.2. Non-technical Constraints

- Group is composed by two elements.
- Budget for the project under 100€.
- Minimum of 6 hours of work a week for each group member.
- Finish within the specified deadlines.
- Meet the milestones specified by the group (in the Gantt Diagram)
- Use and gain experience in new software used in the software and hardware development.
- Lowes budget possible**.**

3. System overview

The system is divided in three parts: The Controller, the Robot and the Remote desktop.

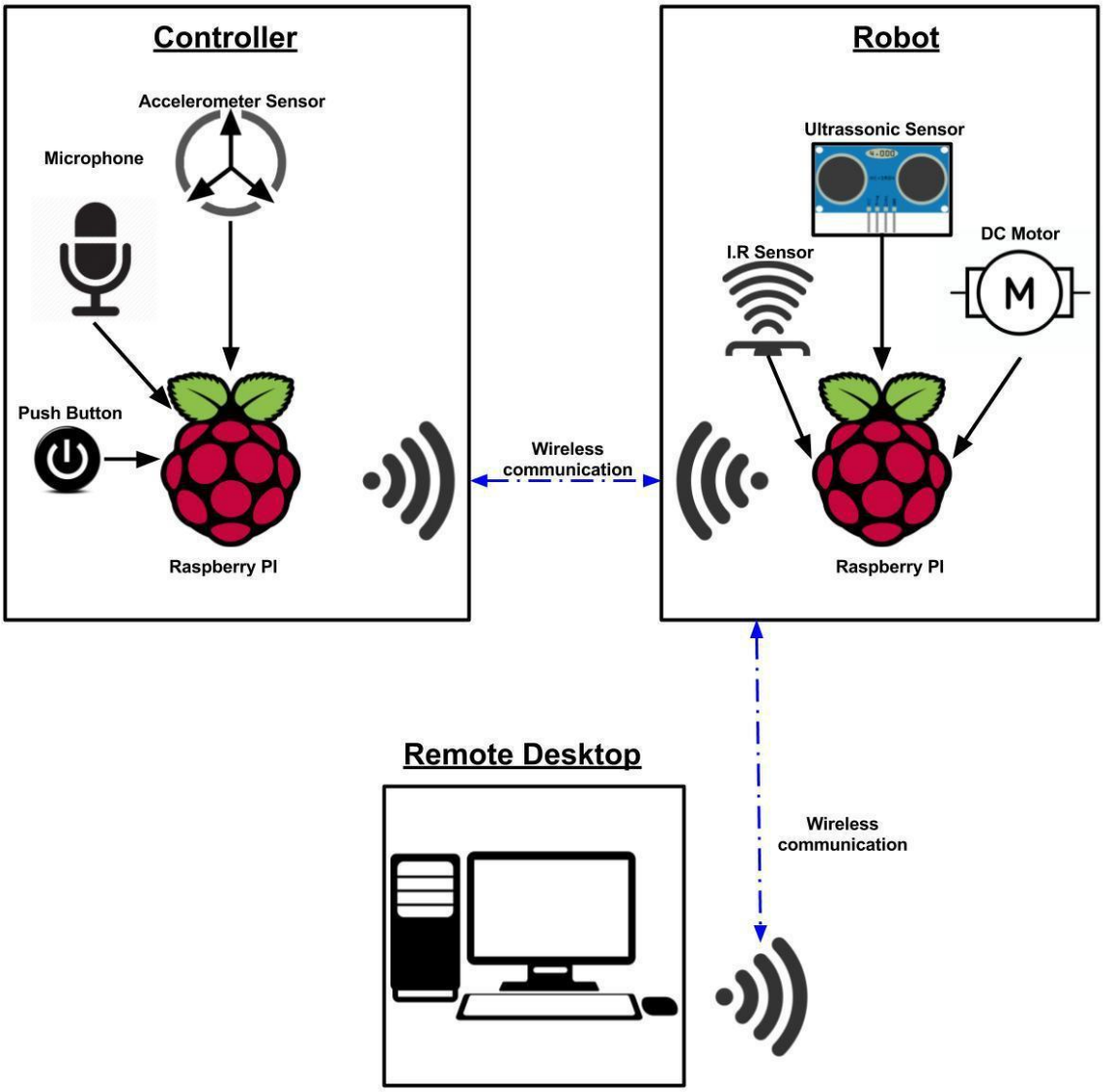

figure 2 – System Hardware architecture

The controller System it's composed by the Raspberry PI 3 connected to an accelerometer sensor, push buttons and a microphone. The controller system gives the user two different scenarios to control the Robot: the first one is using the accelerometer sensor to detect the user hand or head position and control the Robot based on the hand or head position, and the second it's through a microphone connected to the Raspberry PI send voice commands to control the Robot

In the Robot system, we have another Raspberry PI that control the movements of the robot and uses infrared and ultrasonic sensor to sense obstacle around it. The Robot performs his movements according to the commands sent by the controller system or the application from the Remote Desktop the Remote desktop system presents in real-time the robot status.
The board in the Controller system communicates with the board on the Robot system through a TCP IP socket, the Robot system also uses the same communication to communicate with the Desktop Application

4. Resources

In this part of the design Phase is specified all the resources (hardware and software used in the project), the list can change during the project implementation or design.

4.1. Hardware

The hardware resources used in this project are the following:

- Raspberry PI
- IR Sensor
- 3-axis Accelerometer sensor
- Batteries
- DC Motors
- Motor Driver L298N
- Push Buttons
- Microphone
- Ultrasonic Sensor
- LED's
- Buzzer.

4.2. Software

The software that are going to be part of the implementation of the project are the following:

- Embedded Linux
- Buildroot
- C/C++ language
- Pocketspinx speech recognition engine

5. Technology Stack

The system technology stack is described in figure 16 and represents the hardware and software of the system.

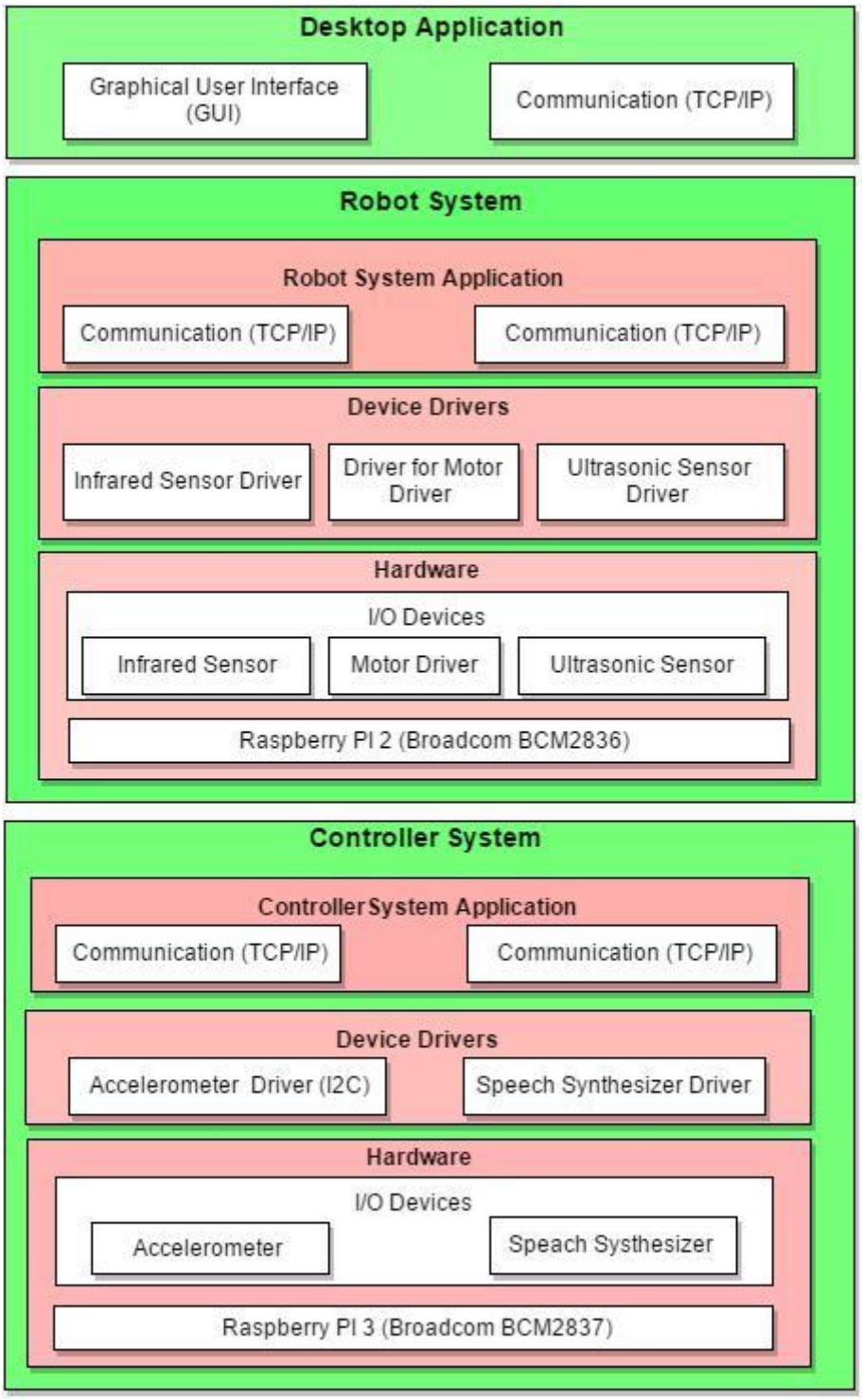

Figure 3: System Technology Stack

In the system Technology Stack image is presented the system software and hardware components and how they interact with each other. On the top, we have the Desktop Application, the application will be developed in C/C++ using the QT, the communication between the Robot system and the Desktop application will be through TCP/IP protocol.
In the middle, it's the Robot system layer, the robot system has three I/O devices (Infrared Sensor, ultrasonic sensor and the motor driver) that communicates with the Robot system application through the device drivers of each Hardware device. the robot system application communicates with the Desktop and the Controller system using TCP/IP protocol. The development board used in this layer is the Raspberry PI 2 (System on chip Broadcom BCM2836).
The segment on the bottom it's the Controller system, the controller system is divided in three segments, the hardware segment composed by the development board Raspberry PI 3 (with the System on chip Broadcom BCM2836) connected to two hardware devices (accelerometer Sensor and Speech synthesizer), the others two segments are the device drivers and the Controller application, the controller application use the device drivers to communicate with the hardware devices. The Controller system communicates with the Robot system through TCP/IP protocol.

6. UML Modeling Diagrams

The Unified Modeling Language(UML) is a standard visual modeling language projected to be used for modeling business and similar processes, analysis, design, and implementation of software-based systems. UML is a common language for business analysts, software architects and developers used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems. It provides guidance as to the order of a team's activities, specifies what artifacts should be developed, directs the tasks of individual developers and the team and offers criteria for monitoring and measuring a project's products and activities.

UML specification defines two major kinds of UML diagram: structure diagrams and behavior diagrams.
Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.
Structure diagrams: Class diagram, object diagram, package diagram, profile diagram, employment diagram, etc.
Behavior diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.
Behavior diagrams: Interaction Diagram, State machine diagram, activity diagram, use case diagram, etc.

6.1. Block Diagram

Figure 4: Generic System Block Diagram

The figure 4 presents the generic global system block diagram with the three main blocks and the corresponding cardinalities. These blocks are the Robot, the Controller and the Desktop application. The Robot is the only imperative block, because it only needs one of the other blocks to make the system work.

6.2. Desktop application UML

In this section is presented the UML diagrams from the Desktop Application.

6.2.1. Use case diagram

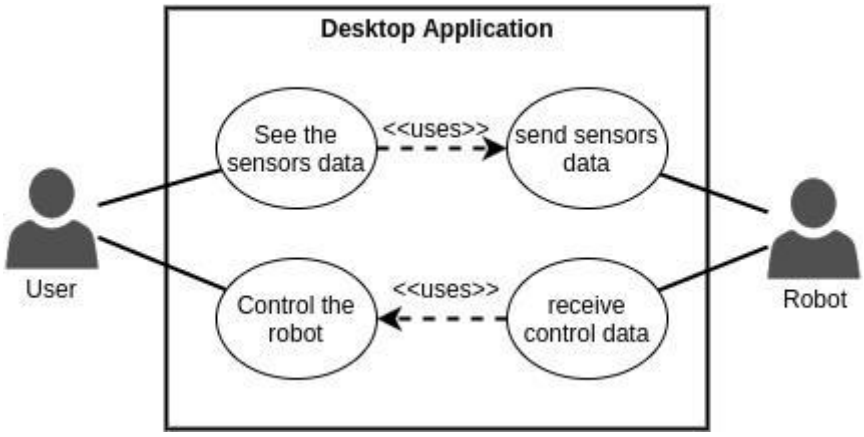

Figure 5: Desktop Application Use Cases

The Figure 5 shows the Desktop Application use cases. We can see all actions performed there and the actors that interact with it. In the Desktop Application, the user can not only see the Robot and Controller sensors data but he can also control the Robot. It is possible because this application will be able to exchange data with the Robot over Wi-Fi connection. If the user chooses the Desktop application as control method, the controls provided by the Controller System is disabled until it is enabled again in the controller button.

6.2.2. Event Table

EVENT	RESPONSE	ACTOR	TYPE
Updated control information	Send control information to robot	Desktop application	Asynchronous
Direction selection	Updates control information	User	Asynchronous
Updated sensors information	Shows the sensors information	Desktop application	Synchronous

Table 1: Events table for Desktop Application

In the previous table is shown the Desktop Application events accordingly to the use cases defined. It also shows the type of the event (synchronous or asynchronous), the actor who triggers the event and the system response when that event occurs.

6.2.3. State diagram

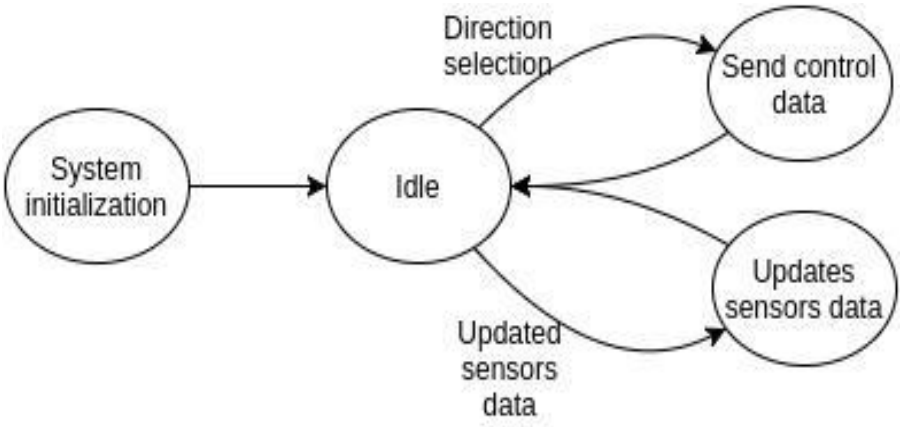

Figure 6: State diagram for Desktop Application

The state diagram presented above (Figure 6) describes the Desktop Application in a very high level. After the Desktop Application be initialized, it will be waiting to the user inputs the controls and send it to the Robot. It will also receive sensors data from the robot and periodically update that data on the desktop screen.

6.3. Controller System UML

In this section is presented the UML diagrams from the Controller System.

6.3.1. Use Case diagram

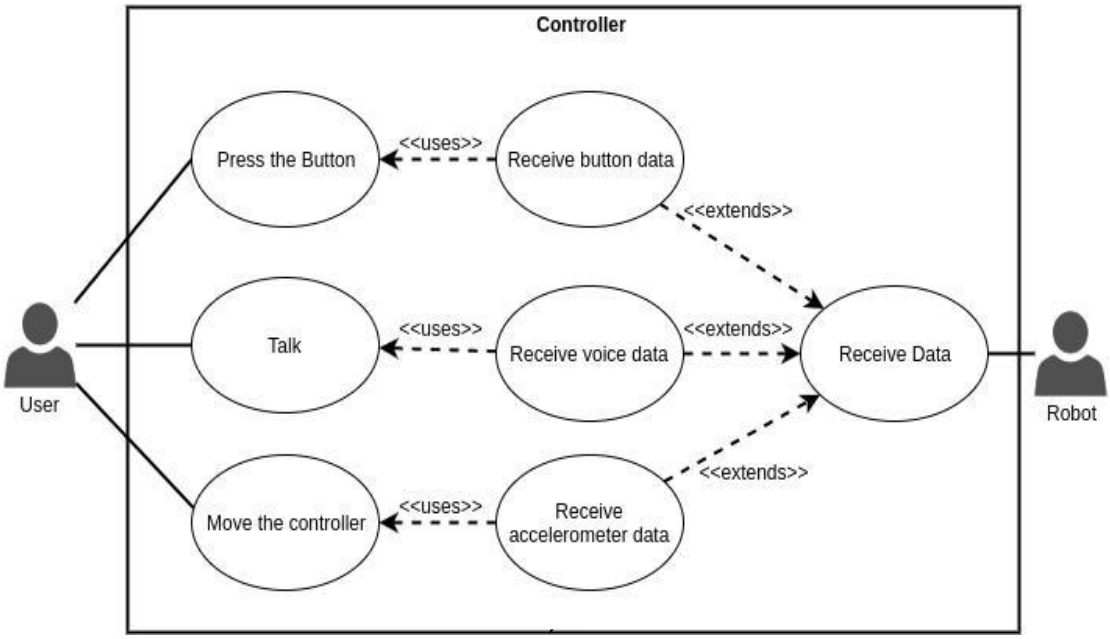

Figure 7: Controller Use Cases

The figure presented above (Figure 7) describes the Controller Use Cases. It presents the actors involved and the actions they perform. The user can input control data in two different ways: by voice commands or my moving the controller. There is one button to select one of this input methods, and another button to start and stop the Robot movements. The Robot interacts with the controller, by receiving the data from the input control methods and the data from the buttons.

6.3.2. Events Table

EVENT	RESPONSE	ACTOR	TYPE
Press Button	Send start/stop information to robot	User	Asynchronous
Updated accelerometer information	Read and Send accelerometer information to Robot	Controller	Synchronous
Say command	Update commands information	User	Asynchronous
Command detected	Send command to robot	Controller	Asynchronous

Table 2: Events table for Controller

The table presented above shows the events that can happen in the Controller, according to the use cases defined. It also shows the type of the events (synchronous or asynchronous), the actor who triggers the event and the system response when that event occurs.

6.3.3. State Diagram

the figure below describes the Controller system in a very high level. After the system initialization the Controller is waiting for the user to input the control data by voice or by motion. The Controller also awaits the button to start or stop the Robot is pressed, and also the button that selects the input control method.

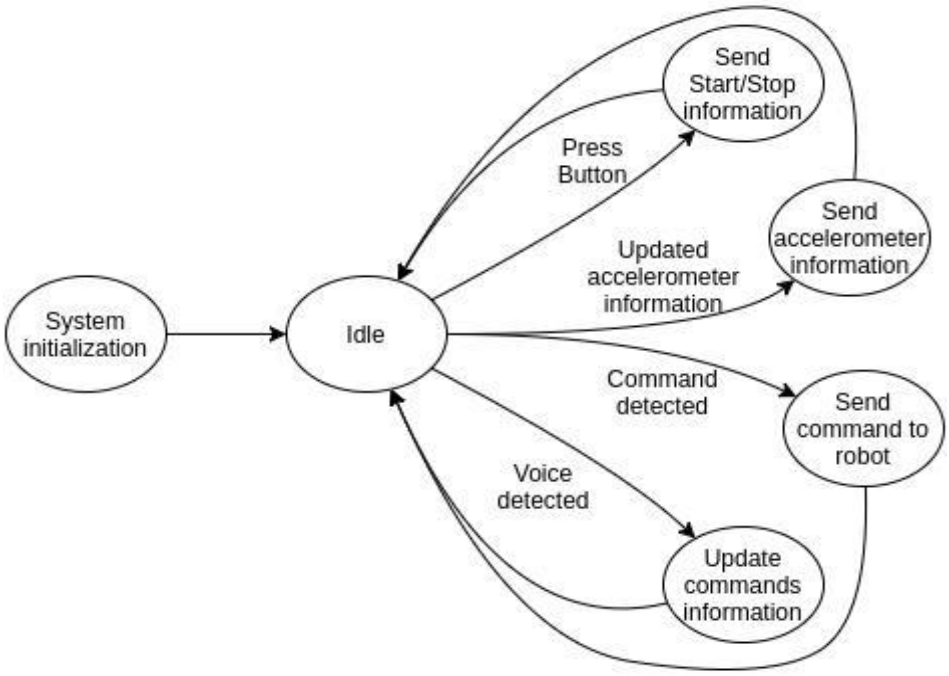

Figure 8: State Diagram for Controller

6.4. Robot System UML

6.4.1. Use Cases Diagram

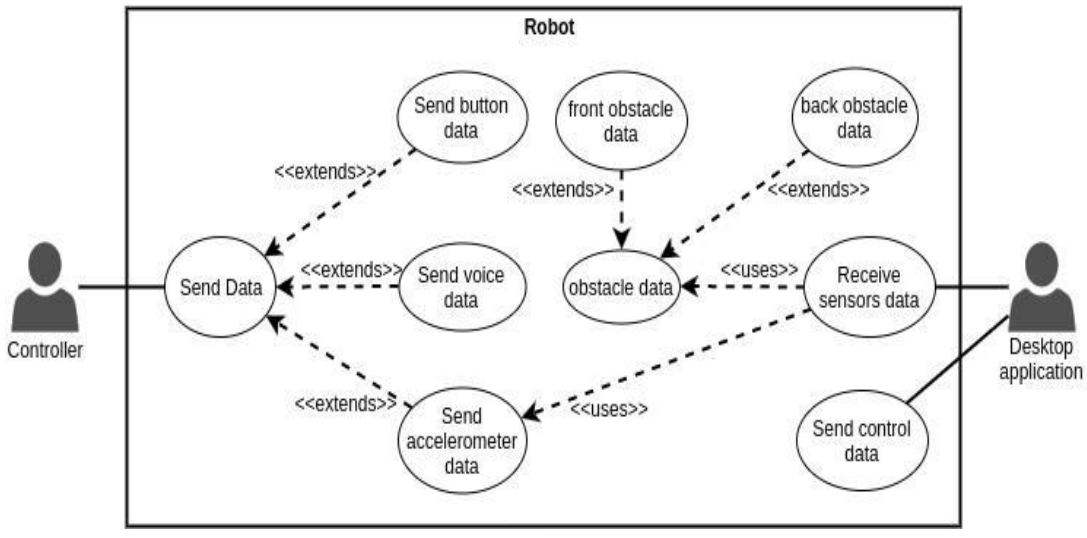

Figure 9: Robot Use Case

The figure presented above (Figure 9) describes the Robot use cases. It presents the actors involved and the action they perform. The User doesn't interact directly with the Robot, since this one only exchanges data with the controller and the desktop application.

The Robot receives data inputs from the controller and from the desktop application, moving accordingly to that data and the data from the obstacle sensors data. The data from all the sensors, including the accelerometer data received by the controller, is sent to the Desktop application, to allow its visualization in a graphical interface.

6.4.2. Event Table

EVENT	SYSTEM RESPONSE	ACTOR	TYPE
Front obstacle detected	Prevent the robot to move forward; Buzzer alert	Robot	Asynchronous
Back obstacle detected	Prevent the robot to move back; Buzzer alert	Robot	Asynchronous
Receive Start/Stop information	Start the robot if it is stopped or stop if it is moving	Controller	Asynchronous
Receive accelerometer information	Move the robot accordingly to accelerometer information	Controller	Synchronous
Receive control data	Move the robot accordingly to control data	Desktop application	Asynchronous
Receive voice information	Move the robot accordingly to voice commands	Controller	Synchronous
Updated sensors information	Send sensors data to desktop application	Robot	Synchronous

Table 3: Events table for Robot

The table presented above (Table 3) shows the events that can happen in the Robot, accordingly to the use cases defined. It also shows the type of the event (synchronous or asynchronous), the actor who triggers the event and the system response when that event occurs.

6.4.3. State Diagram

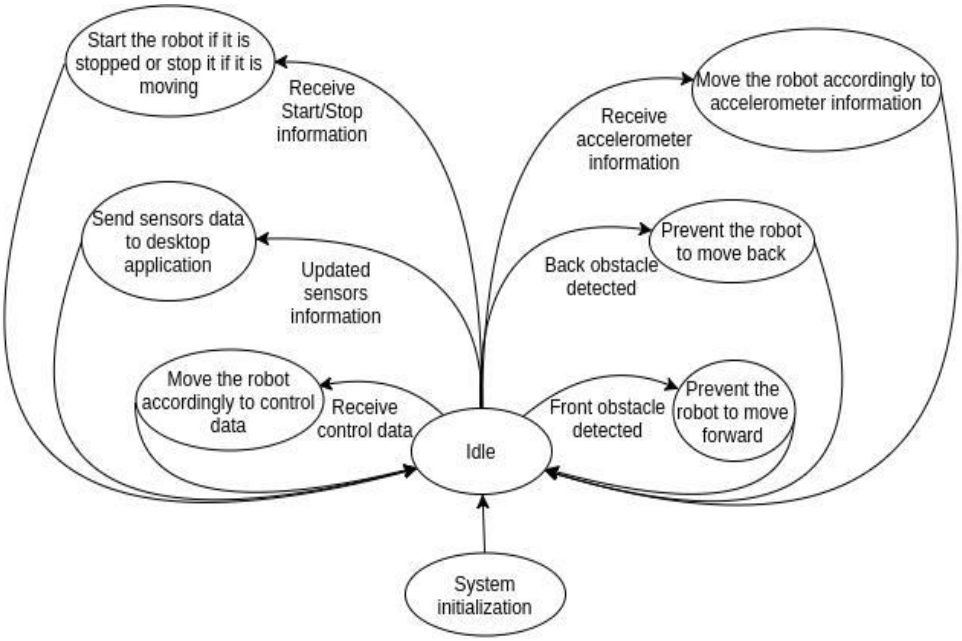

Figure 10: State Diagram for Robot

The state diagram presented above (Figure 10) describes the Robot in a very high level. After the system initialization, the controller is receiving movement information data from the Controller's accelerometer, and also from the Desktop Application. It is also waiting to receive data from the two buttons in the Controller, one to select the input method (voice or motion), and another to start and stop the Robot. The Robot has also sensors to detect front and back obstacles, that are used to limit the Robot movement. The robot sends periodically to the Desktop Application information about the sensors.

6.5. System Sequence Diagram

In order to fulfil the specification defined previously the sequence diagram of the entire system can have three different cases:

6.5.1. Case 1

This system diagram (shown in the figure 11) is applied when the user wants to use the controller as the input of the commands and the remote desktop system as the output of the robot status.

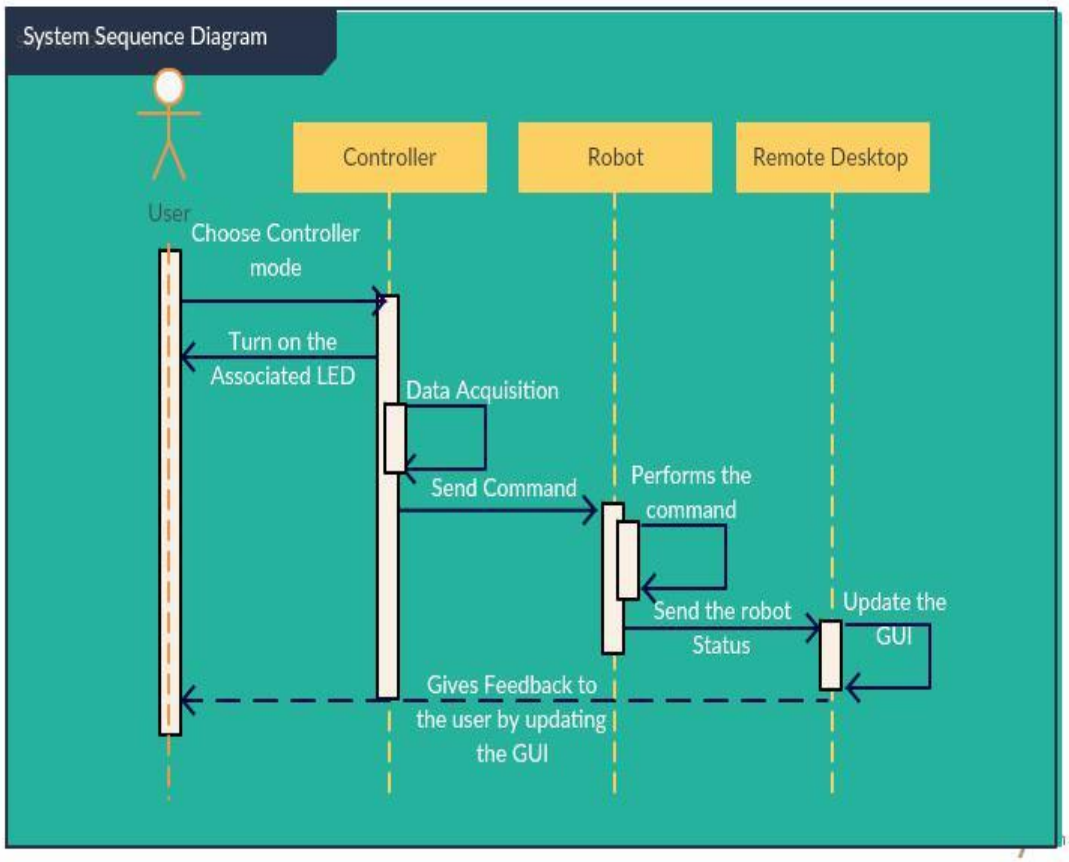

Figure 11: Case 1 System diagram

6.5.2. Case 2

This system diagram (shown in the figure 12) is applied when the user doesn't want to use the controller to input the commands but uses the remote desktop system to input the commands and to do the output of the robot status.

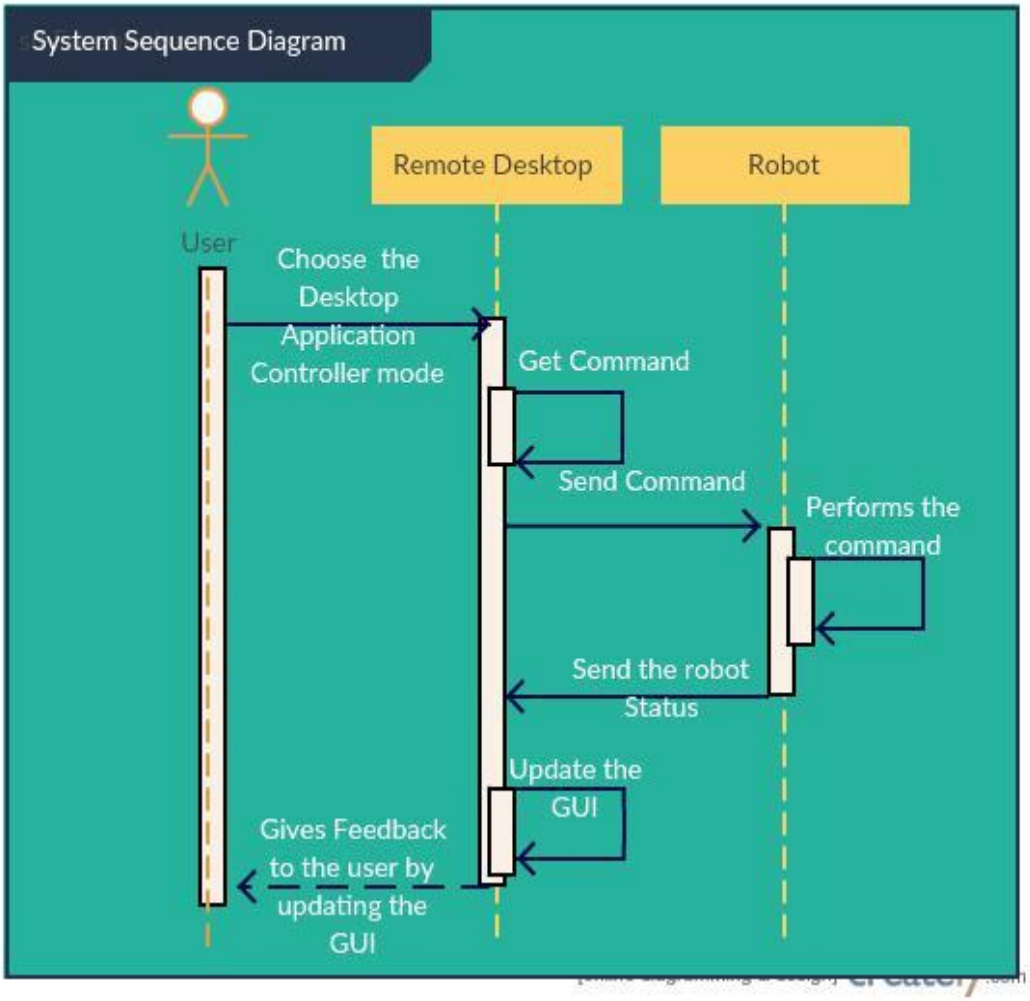

Figure 12: Case 2 System diagram

6.5.3. Case 3

This system diagram (shown in the figure 13) is applied when the user doesn't want to use remote system and only use the controller to input the commands and control the Robot.

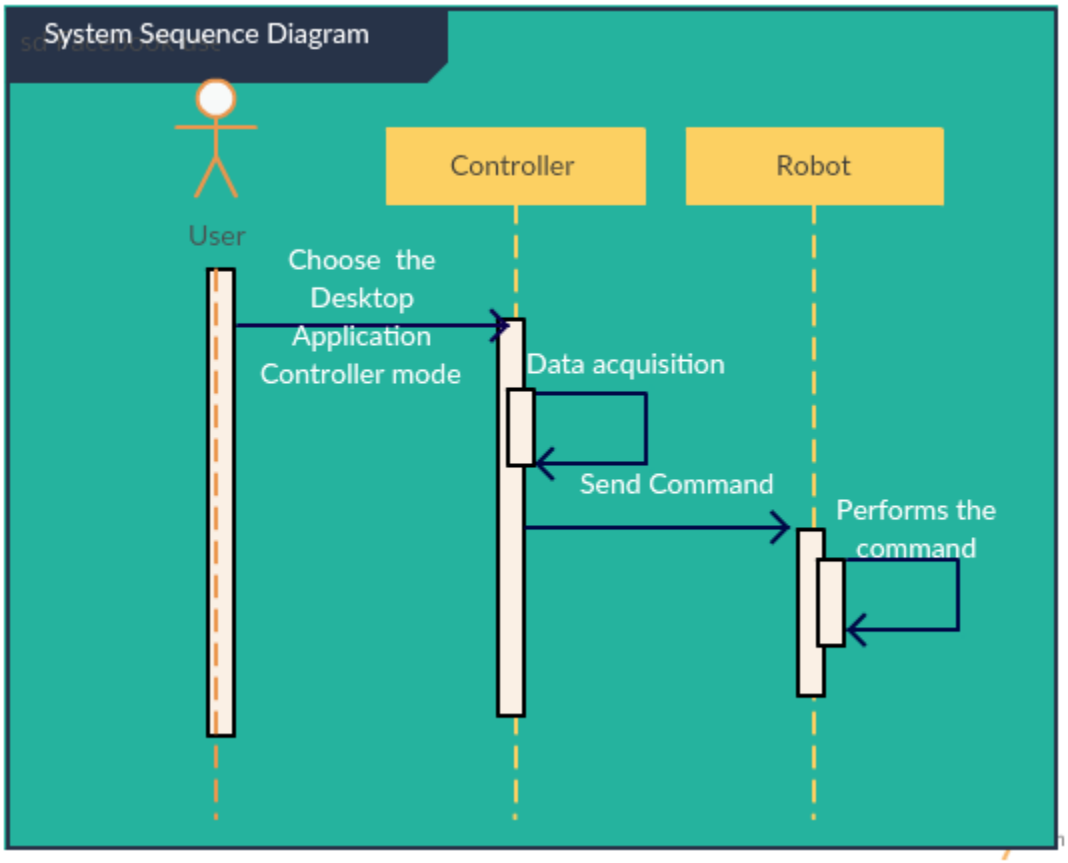

Figure 13: Case 3 System diagram

Design Phase

1. ardware Specification

The hardware used in the project are described in this section of the report

1.1. Hardware Resources

1.1.1. Development Board Raspberry PI 3

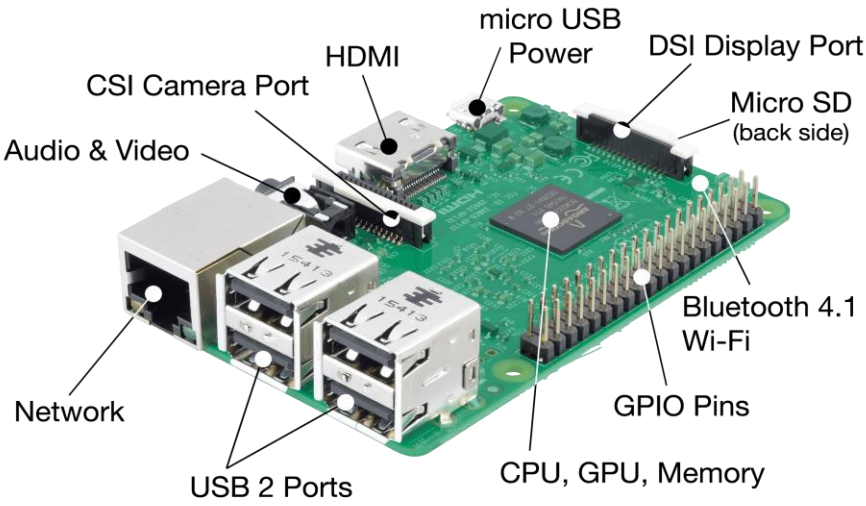

Figure 14: Raspberry PI 3 Overview

Specifications:

- SoC: Broadcom BCM2837
- CPU: 4× ARM Cortex-A53, 1.2GHz
- GPU: Broadcom VideoCore IV
- RAM: 1GB LPDDR2 (900 MHz)
- Networking: 10/100 Ethernet, 2.4GHz 802.11n wireless
- Bluetooth: Bluetooth 4.1 Classic, Bluetooth Low Energy
- Storage: microSD
- GPIO: 40-pin header, populated
- Ports: HDMI, 3.5mm analogue audio-video jack, 4× USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

1.1.2. Development Board Raspberry PI 2

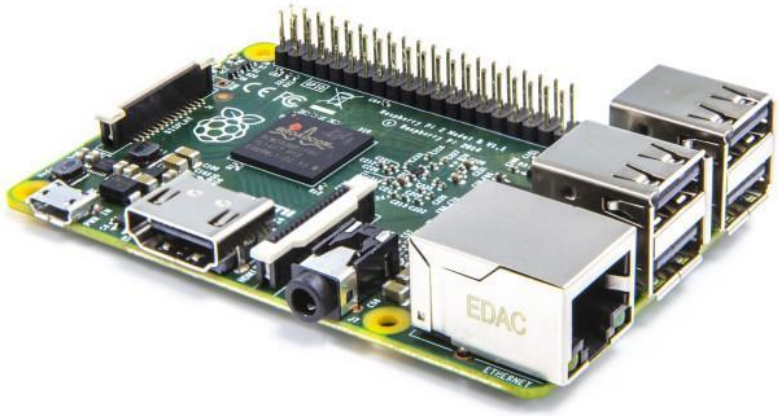

Figure 15: Raspberry PI 2 Model B Overview

Specification:

- SoC: Broadcom BCM2836 (CPU, GPU, DSP, SDRAM)
- CPU: 900 MHz quad-core ARM Cortex A7 (ARMv7 instruction set)
- GPU: Broadcom VideoCore IV @ 250 MHz
- More GPU info: OpenGL ES 2.0 (24 GFLOPS); 1080p30 MPEG-2 and VC-1 decoder (with license); 1080p30 h.264/MPEG-4 AVC high-profile decoder and encoder
- Memory: 1 GB (shared with GPU)
- USB ports: 4
- Video input: 15-pin MIPI camera interface (CSI) connector
- Video outputs: HDMI, composite video (PAL and NTSC) via 3.5 mm jack
- Audio input: I²S
- Audio outputs: Analog via 3.5 mm jack; digital via HDMI and I²S
- Storage: MicroSD
- Network: 10/100Mbps Ethernet
- Peripherals: 17 GPIO plus specific functions, and HAT ID bus
- Power rating: 800 mA (4.0 W)
- Power source: 5 V via MicroUSB or GPIO header
- Size: 85.60mm × 56.5mm
- Weight: 45g (1.6 oz)

1.1.3. ADXL345 Accelerometer Sensor

Description:

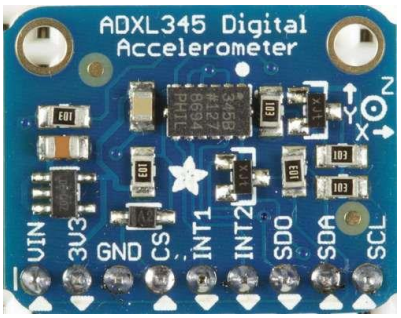

Figure 16: ADXL345 Sensor

The ADXL345 is a small, thin, low power, 3-axis accelerometer with high resolution (13-bit) measurement at up to ±16 g. Digital output data is formatted as 16-bit twos complement and is accessible through either a SPI (3- or 4-wire) or I2C digital interface. The ADXL345 is well suited for mobile device applications. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock. Its high resolution (4 mg/LSB) enables measurement of inclination changes less than 1.0°. Several special sensing functions are provided. Activity and inactivity sensing detect the presence or lack of motion and if the acceleration on any axis exceeds a user-set level. Tap sensing detects single and double taps. Free-fall sensing detects if the device is falling. These functions can be mapped to one of two interrupt output pins. An integrated, patent pending 32-level first in, first out (FIFO) buffer can be used to store data to minimize host processor intervention. Low power modes enable intelligent motion-based power management with threshold sensing and active acceleration measurement at extremely low power dissipation. The ADXL345 is supplied in a small, thin, 3 mm × 5 mm × 1 mm, 14-lead, plastic package.

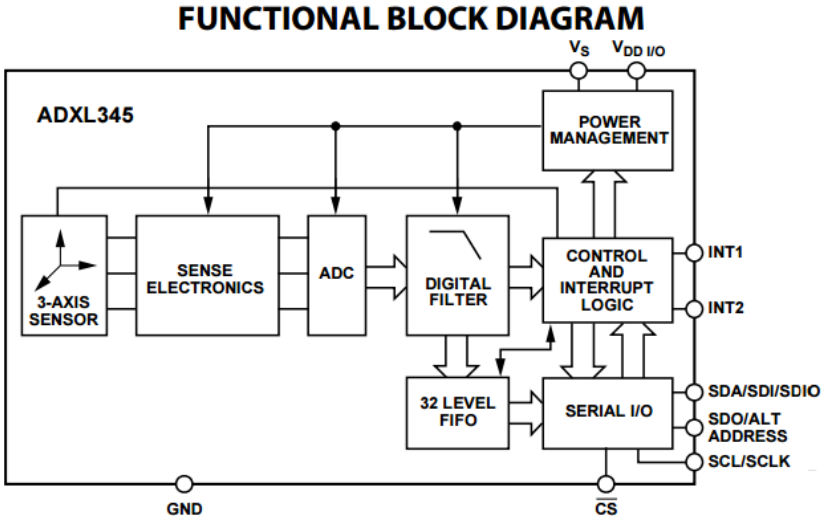

Figure 17: ADXL345 Accelerometer Sensor Block Diagram (Datasheet)

ADXL345 Accelerometer Sensor Pin Description

Pin	Label	Description
1	VDD	Digital interface Supply Voltage
2	GND	Ground voltage
3	VS	Supply Voltage
4	CS	Chip Select
5	INT1	Interrupt 1
6	INT2	Interrupt 2
7	SDO/ALT ADDRESS	Serial Data Output/Alternate I2C Address
8	SDA/SDI/SDIO	Serial data (I2C)
9	SCL/SCLK	Serial Communication Clock

Table 4: ADXL345 Accelerometer Sensor Pin Description

I2C Serial Communication Protocol

With CS tied high to VDD, the ADXL345 is in I2C mode, requiring a simple 2-wire connection (as shown in figure 5). The ADXL345 supports standard (100 kHz) and fast (400 kHz) data transfer modes if the timing parameters given in and are met. Single-or multiple-byte reads/writes are supported.

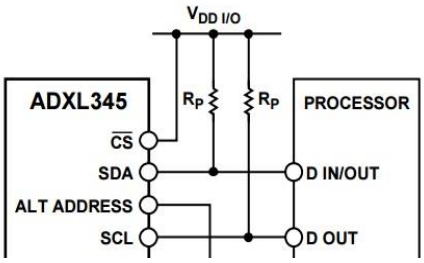

Figure 18: I2C Serial Communication

With the SDO/ALT ADDRESS pin high, the 7-bit I2C address for the device is 0x1D, followed by the R/W bit. This translates to 0x3A for a write and 0x3B for a read. An alternate I2C address of 0x53 (followed by the R/W bit) can be chosen by grounding the SDO/ALT ADDRESS pin (Pin 12). This translates to 0xA6 for a write and 0xA7 for a read.

ADXL345 Accelerometer Sensor Pin Assignment for I2C Communication

label	Connected to:	Description/Type
CS	Pin_17 - 3.3V DC Power	I2C Select
SDA	Pin_03 - GPIO_02 (SDA1, I2C)	I2C Serial Data
SCL	Pin_05 - GPIO_03 (SCL1, I2C)	I2C Serial Clock
INT1	Pin_29 - GPIO_05	General Purpose Input (1)
VS	Pin_17 - 3.3V DC Power	Power Supply
VDD	Pin_17 - 3.3V DC Power	Digital Interface Power Supply
GND	GND	Ground Voltage
SDO	GND	Choose I2C Alternative address (0x53)

Table 5: ADXL345 Accelerometer Sensor Pin Assignment for I2C Communication

1. **-** Each GPIO pin, when configured as a general-purpose input, can be configured as an interrupt source to the ARM. Several interrupt generation sources are configurable (Level-sensitive (high/low), Rising/falling edge and Asynchronous rising/falling edge) Level interrupts maintain the interrupt status until the level has been cleared by system software

IMU Sensor Test Case

Input	Expected Output	Output
Shake the sensor	LED On	

Table 6: IMU Sensor Test Case

1.1.4. Ultrasonic Module HC - SR04

Specifications:

Figure19: Ultrasonic Module

- Power supply :5V
- DC quiescent current: <2mA
- Effectual angle: <15°
- Ranging distance: 2cm – 500 cm
- Resolution: 0.3 cm
- Trigger Input Signal: 10uS TTL pulse
- Echo Output Signal Input TTL level signal and the range in proportion
- Dimension 45*20*15mm

Ultrasonic Sensor Pin Description

Pin	Label	Description
1	VCC	Supply Voltage
2	GND	Ground voltage
3	Trigger	Trigger
4	Echo	Echo

Table 7: Ultrasonic Sensor Pin Description

Timing diagram

The Timing diagram is shown below. It only need to supply a short 10uS pulse to the trigger input to start the ranging, and then the module will send out an 8-cycle burst of ultrasound at 40 kHz and raise its echo. The Echo is a distance object that is pulse width and the range in proportion. The range can be calculated through the time interval between sending trigger signal and receiving echo signal.

Formula: the range = high level time * velocity (340M/S) / 2; Is suggested to use over 60ms measurement cycle, to prevent trigger signal to the echo signal.

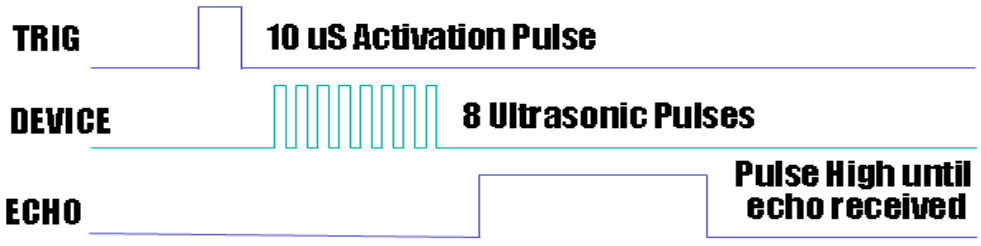

Figure 20: Timing diagram

Ultrasonic Sensor Pin Assignment

Label	Connected to:	Description/Type
VCC	Pin_02 - DC Power 5V	Supply Voltage
GND	Pin_09 - Ground	Ground voltage
Trigger	Pin_12 - GPIO1 PWM0	PWM
Echo	Pin_22 - GPIO_6	Output

Table 8: Ultrasonic Sensor Pin Assignment

Ultrasonic Module HC - SR04 Test Case

Input	Expected Output	Output
Obstacle in the Sensor Range	Receive Echo Signal	

Table 9: Ultrasonic Module HC - SR04 Test Case

1.1.5. Infrared Barrier Sensor Module

Main Features:

- When the signal module detects obstacles ahead, the green indicator will light and the output pin will change to the High state
- Detection distance can be adjusted through the potentiometer knob, it is range from 2cm to 30cm.
- Operating voltage: 3.3-5V

Figure 2: Infrared Sensor Module

Infrared Sensor Pin Description

Pin	Label	Description
1	VCC	Supply Voltage
2	GND	Ground voltage
3	Out	Output

Table 1: Infrared Sensor Pin Description

Infrared Sensor Pin Assignment

Label	Connected to:	Description/Type
VCC	Pin_2 - 5V DC Power	Vcc
GND	Pin_9 - Ground	GND
Out	Pin_18 - GPIO_5	General Purpose Input

Table 2: Infrared Sensor Pin Assignment

Infrared Sensor Test Case

Input	Expected Output	Output
Obstacle in the Sensor Range	Led Turn ON	

Table 3: Ultrasonic Module HC - SR04 Test Case

1.1.6. Motor Driver (L298)

Description:

The L298 is an integrated monolithic circuit in a 15- lead Multiwatt and PowerSO20 packages. It is a high voltage, high current dual full-bridge driver designed to accept standard TTL logic levels and drive inductive loads such as relays, solenoids, DC and stepping motors. Two enable inputs are provided to enable or disable the device independently of the input signals. The emitters of the lower transistors of each bridge are connected and the corresponding external terminal can be used for the connection of an external sensing resistor. An additional supply input is provided so that the logic works at a lower voltage.

Figure 3: Motor Driver

Block Diagram:

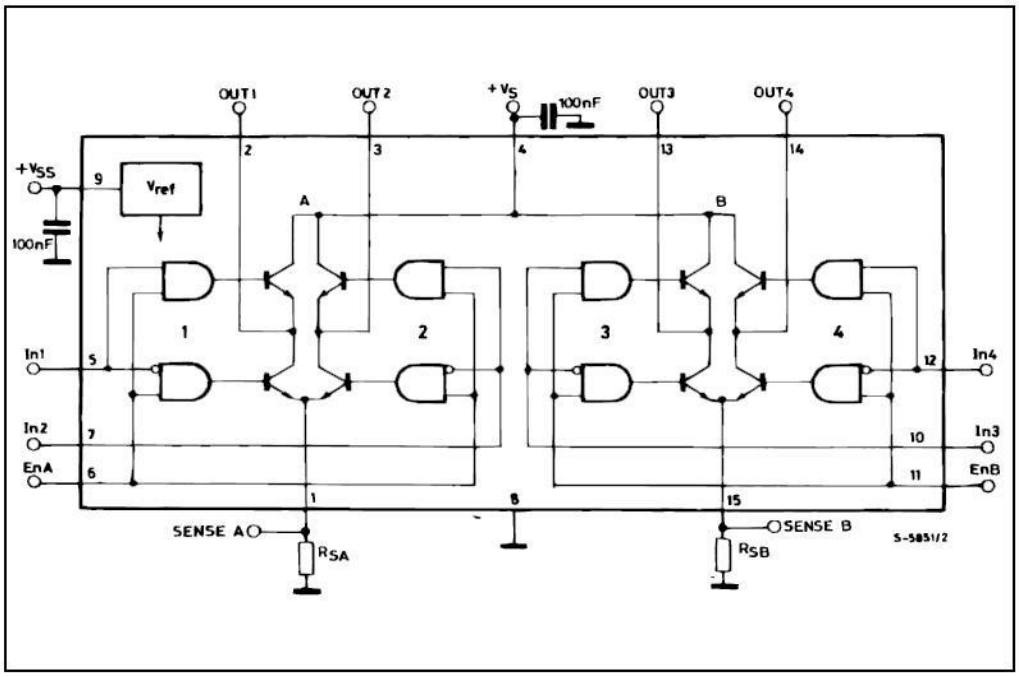

Figure 4: L298N Block Diagram

Motor Driver Pin Description

PIN	Label	Description/Function
1	GND	Ground Voltage
2	Out1	MotorA&C + Voltage
3	Out2	MotorA&C - Voltage
4	Out3	MotorB&D + Voltage
5	Out4	MotorB&D - Voltage
6	In1	Pin to control motor A
7	In2	Pin to control motor A
8	In3	Pin to control motor B
9	In4	Pin to control motor B
10	enA	Enable for Motor A&C
11	enB	Enable for Motor C&D
12	VCC	5 Volt

Table 4: Motor Driver Pin Description

Motor Driver Pin Assignment

Motor Driver L298N	Connected to:	Description/Type
VS	Battery +	-
In1	RPI-Pin_11 - GPIO_0	Input
In2	RPI-Pin_13 - GPIO_2	Input
In3	RPI-Pin_15 - GPIO_3	Input
In4	RPI-Pin_16 - GPIO_4	Input
enA	RPI-Pin_33- GPIO23 PWM1	PWM -Input
enB	RPI-Pin_32- GPIO26 PWM0	PWM -Input
GND	Battery –	-
Out1	Motors A&C M1	PWM - Output
Out2	Motors A&C M2	PWM - Output
Out3	Motors B&D M1	PWM - Output
Out4	Motors B&D M2	PWM - Output

Table 5: Motor Driver Pin Assignment

Motor and Motor Driver Test Case

The following table shows the different combination of the Motor Driver and the expected output/state of the Motor.

Input	Expected Output	Output
ENA = 0	Motor A is disable	
ENA = 1 IN1 = 0 IN2 = 0	Motor A is stopped (brakes)	
ENA = 1 IN1 = 0 IN2 = 1	Motor A is on and turning backwards	
ENA = 1 IN1 = 1 IN2 = 0	Motor A is on and turning forwards	
ENA = 1 IN1 = 1 IN2 = 1	Motor A is stopped (brakes)	

Table 6: Motor Driver Test Case

1.1.7. Wireless Wi-Fi Network Card

Figure 5: Wireless Wi-Fi Network Card

The two Raspberry Pi will use a USB 802.11n Mini wireless network Adapter for the wireless communication, the USB 802.11n Mini Wireless Network Adapter allows almost any USB-enabled desktop, laptop or netbook computer system to connect to the wireless networks. Using a 1T1R (1 Transmitter/1 Receiver) design over the 2.4 GHz frequency in an extremely compact form factor, the USB adapter can reach up to 150 Mbps over an 802.11n network while taking up minimal space. Backward compatible with 802.11b/g networks (11/54 Mbps), this adapter is a versatile wireless networking solution. With support for standard and advanced security options.

Wireless Wi-Fi Network Card Interface

To interface with the network Card is connected to the USB port of the Raspberry PI

1.1.8. Microphone

The microphone will interface with the Raspberry through the USB Port.

Specifications:

- **Input Impedance:** 20 Ohms
- **Sensitivity (headphone):** 115dB +/-3dB
- **Sensitivity (microphone):** -42dBV/PA +/-3dB
- **Frequency response (Headset):** 20Hz to 20kHz
- **Frequency response (Microphone):** 100Hz to 6,500Hz
- **Cord length:** 1.8 m (5.90 ft)

Figure 6: Microphone

1.1.9. Power Bank

Description:

- Input: MicroUSB
- Output: USB
- Voltage: 5V
- Capacity: 2600mAh

Figure 7: PowerBank

1.1.10. Touch Button Module

Main Features:

- When module has detected a key pressed, the corresponding port will output low level signal
- Power supply voltage input range is 3 - 6V DC
- With LED indicator, it will light up after being powered on

Figure 8: Touch Button Module

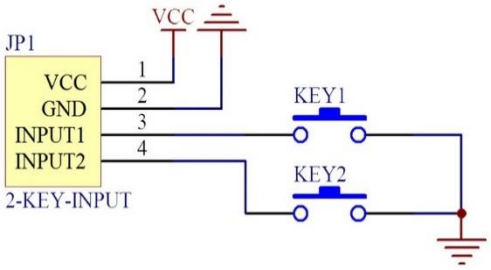

Figure 9: Touch Button Module Schematics

1.2. System Hardware Schematics

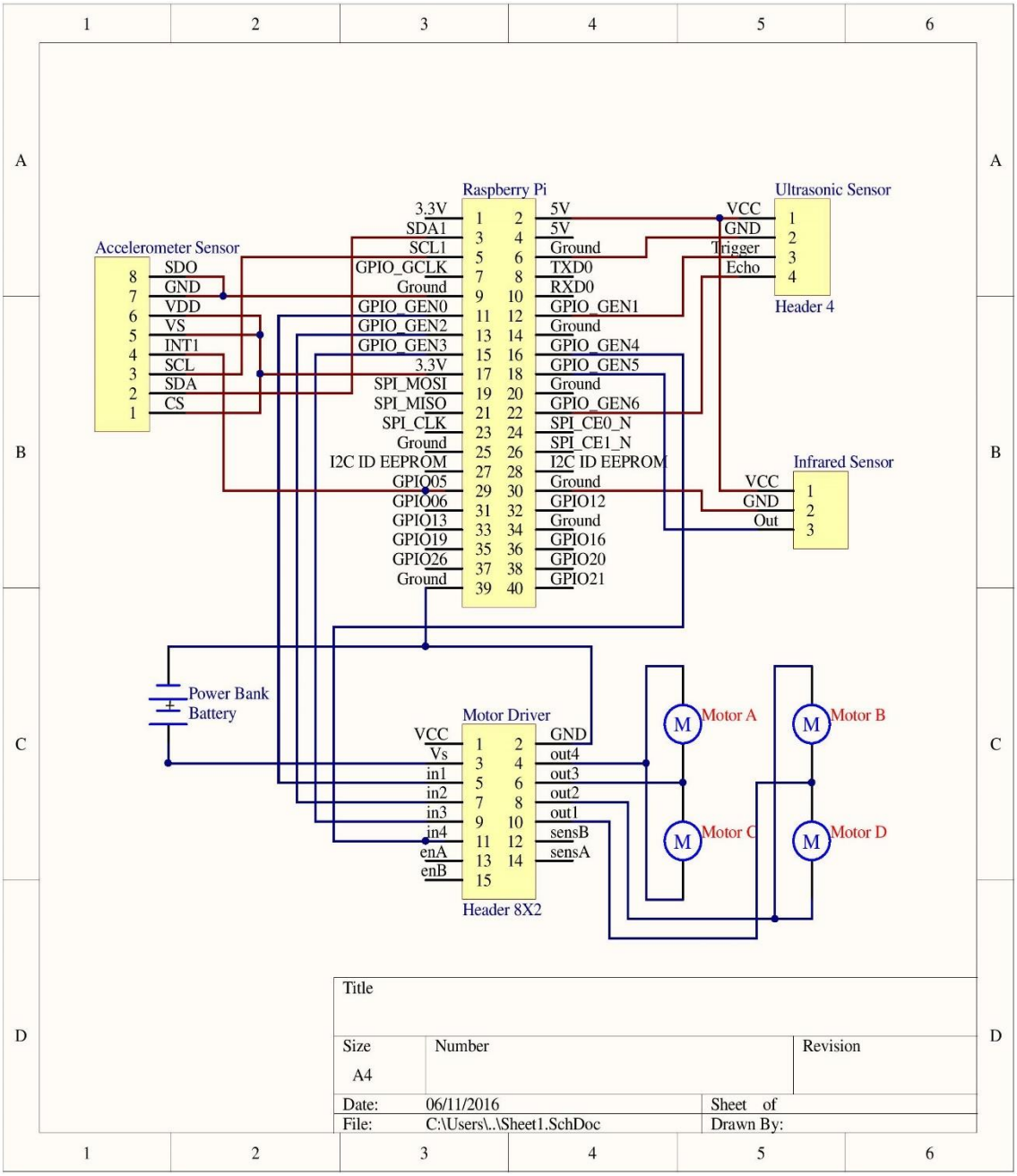

Figure 10: System Hardware Schematics

1.3. Peripherals and Communication Protocols

In this section is presented the peripherals used to interface the Development Board (Raspberry Pi) with the hardware modules.

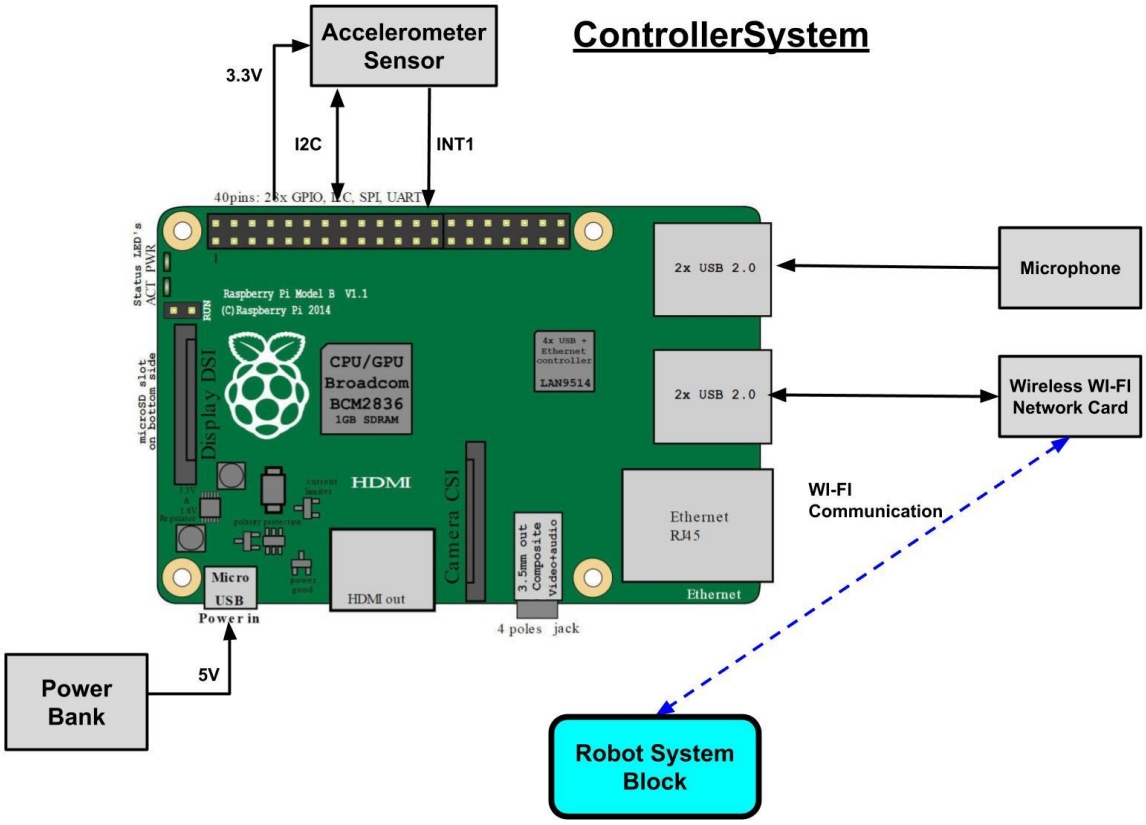

Figure 11: Controller System Peripherals and Communication Protocol

The first figure shows the Peripherals from the development board used to communicate with the hardware modules in the Controller System.
The controller system has four different hardware modules. The power bank that uses is connected to the board micro USB port, the Power Bank is the Power source to the Board. The Wireless WI-FI Network Card and the microphone are connected to the Board through the USB Port. The Accelerometer Sensor interface the board through the GPIO.

In the next figure (figure 18) is presented the peripherals and the protocol communication used to interface the development board and the Robot System. In the robot system, the USB Port is used to communicate with the Wireless WI-FI Network Card, the GPIO to interface the Motor Driver, the Ultrasonic and Infrared sensor and to power the board is used a power bank connected to the Micro USB port.

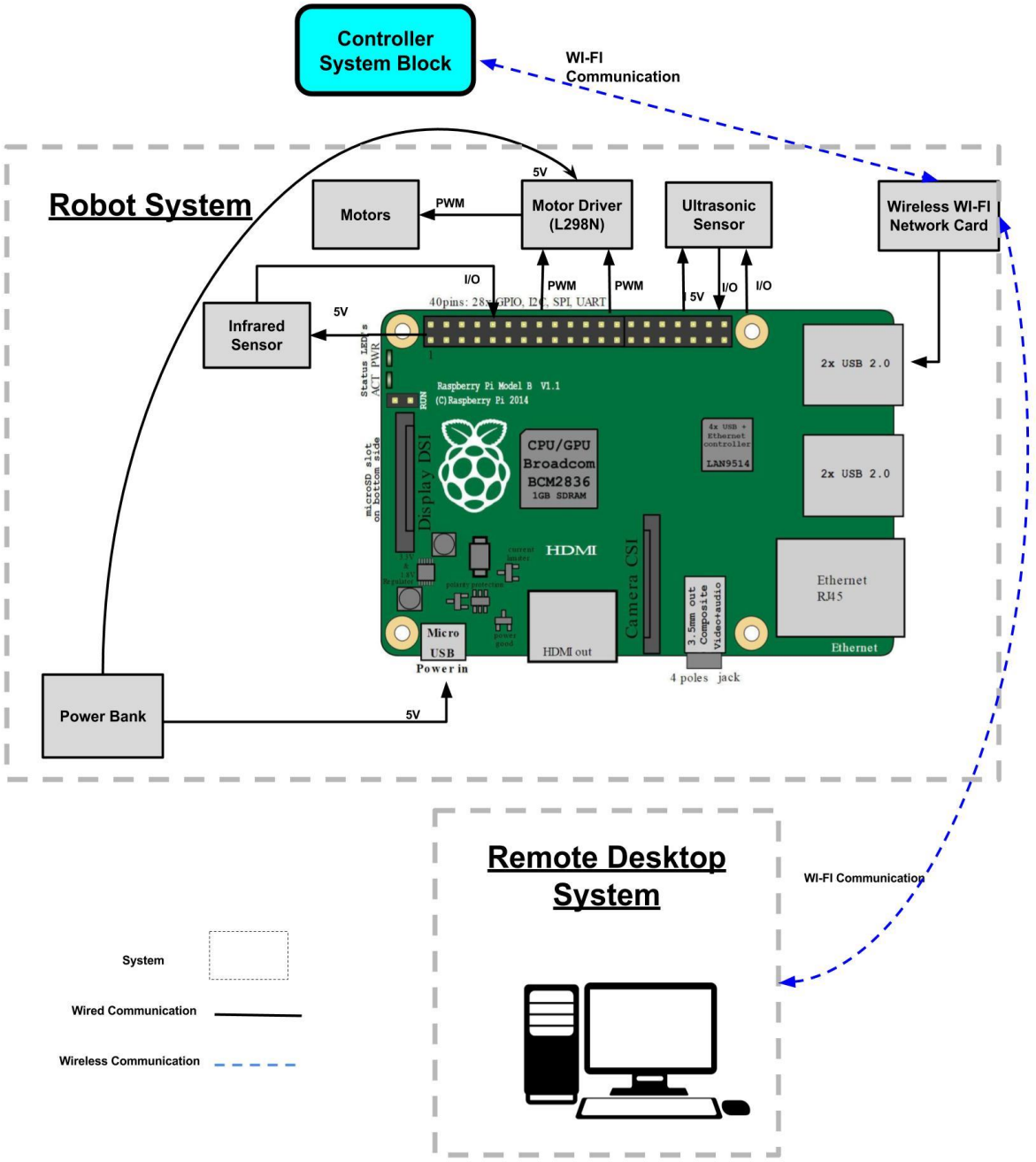

Figure 12: Robot System Peripherals and Communication Protocol

2. Software specification

2.1. Software Tools:

- **Altium Designer**

To design the System Hardware Schematics

- **Qt Creator 4.0.2 Based on Qt 5.7.0**

To implement a graphical user interface in the Desktop Application

- **Buildroot**

To automate the process of building a complete and bootable Linux environment for Robot System and Controller System

2.2 Software Cots:

Pocketsphinx – Pocketsphinx has some dependences as:

- **Pulseaudio**
- **wiringPi**
- **automake**
- **autoconf**
- **libtool**
- **bison**
- **Advanced Linux Sound Architecture (ALSA)**

  The Advanced Linux Sound Architecture (ALSA) provides audio and MIDI functionality to the Linux operating system. ALSA has the following significant features:

  - Efficient support for all types of audio interfaces, from consumer sound cards to professional multichannel audio interfaces.
  - Fully modularized sound drivers.
  - SMP and thread-safe design.
  - User space library (alsa-lib) to simplify application programming and provide higher level functionality.
  - Support for the older Open Sound System (OSS) API, providing binary compatibility for most OSS programs.

2.4. Data Formats

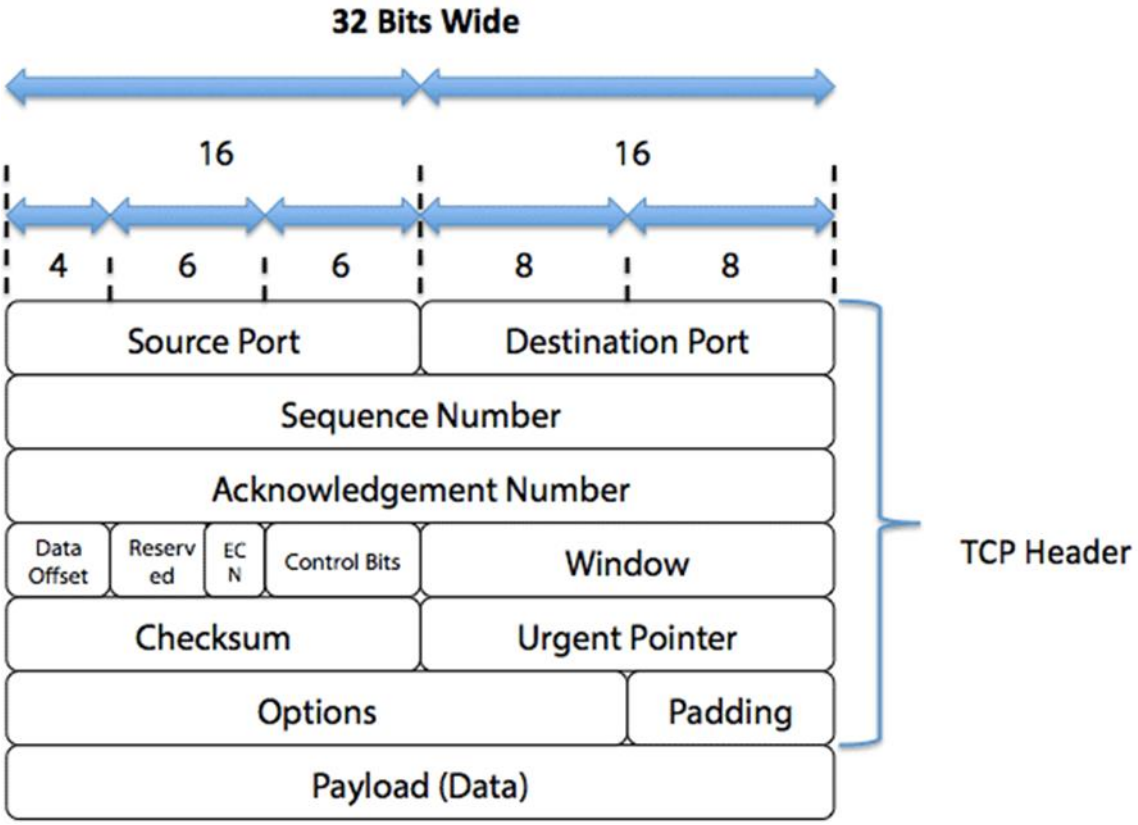

Figure 13: Tcp Data Frame

The messages exchanged between the systems, will be made by TCP. It's Header carries several information fields, including the source and destination host addresses. After the Header comes the Data, whose formats is specified in the next images.

2.4.1. Messages to Robot

- **Accelerometer Data**

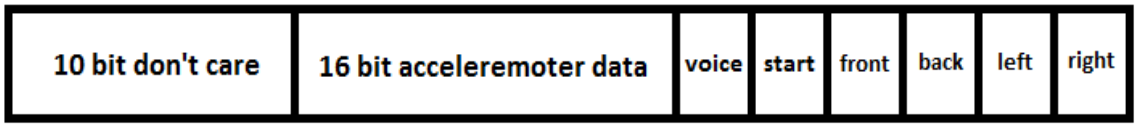

Figure 14: Accelerometer data format

- **Voice or Start/Stop Data**

Figure 15: Voice or Start/Stop data format

2.4.2. Messages to Desktop Application

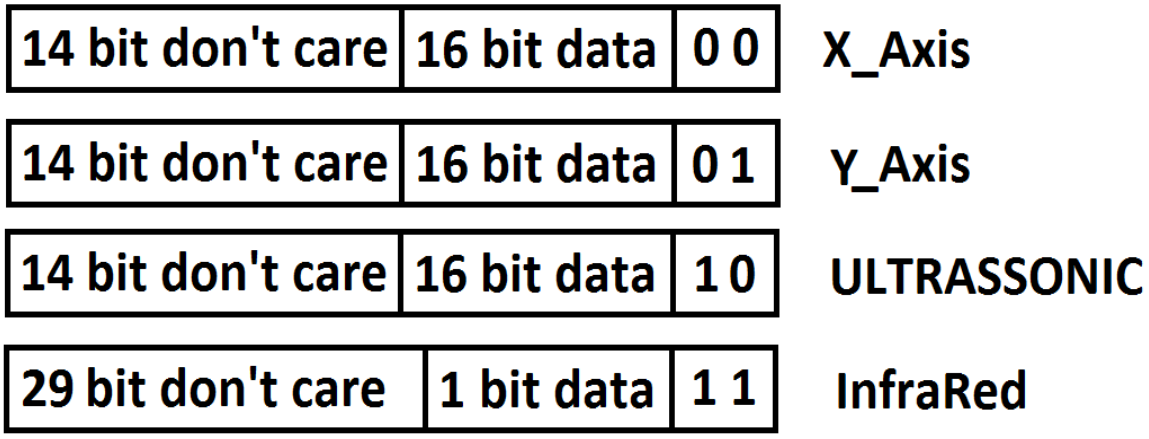

Figure 16: Sensors data format

2.5. Threads Overview

Robot System

- **Send_Wifi -** This thread is responsible to communicate with the Desktop Application. It sends data through a TCP socket to The Desktop Application with the updated data from sensors.

- **Receive_Wifi -** The Robot will receive different TCP frames for each desired direction of movement, Controller buttons data, and also for GUI control option. It will receive frames from two different sources, the Controller and the Desktop Application. This thread is responsible to receive this packages and send the relevant data to the interpreter thread.

- **Command_Handler -** This thread use the data received from the Receive_Wifi Thread and execute/performs the command received.

- **Motor_Controller -** This thread receives the data from the Command Handler and actuates on the motor.

Controller System

- **Buttons_Handler -** This thread waits for changes in the buttons states. After a button is pressed or released, a read operation is performed through the I2C device driver.

- **Send_Wifi -** This thread is responsible to communicate with the Robot System. It sends data through a TCP socket to The Robot System, with the information about direction of movement, voice commands.

- **Voice_Recognition -** This thread is waiting for voice data, interpreting it and moving the commands to the queue to be sent via Wi-Fi.

- **Accelerometer_Handler -** This thread is used to read the accelerometer and sends it to the Robot System.

Desktop Application

- **SendWifi -** This thread is responsible to send (commands) data to through a TCP socket to The Robot System.

- **ReceiveWifi -** The Desktop Application use this thread to receive data (Robot Status) from the Robot System.

- **Update GUI** - This thread is responsible to update the GUI when data is received from the Robot System.

- **Command Handler** – This thread receives commands from the user and send it to the Send_Wifi thread.

2.6. Thread Communication and Synchronization

In order to exchange data between threads is used Semaphore, Queues and Mutex.

2.6.1. Robot System

Semaphore:

- **IR_Obstacle_Sem -** This Semaphore is used to send a signal to the Command_Handler thread when an obstacle is detected by the Infrared Sensor.

- **Ultrasonic_Obstacle_Sem -** This Semaphore is used to send a signal to the Command_Handler thread when an obstacle is detected by the ultrasonic Sensor.

Queue:

- **Send_Alert_Queue** – This message queue is used to send data from the Command_Handler Thread to the Send_Wifi Thread.

- **Receive_Queue** – This message queue is used to send data from the Receive_Wifi Thread to the Command_Handler Thread.

- **MotorPWM_Queue** – This message queue is used to send the PWM for the Motors from the Command_Handler Thread to the Motor_Controller Thread.

Mutex:

- **Distance_Mutex -** This mutex is used to protect the access to the variable that has the distance measured by the Ultrasonic Sensor.

ISRs:

- **IRSensor_ISR** – Interrupt service routine to the Infrared Sensor.

- **UltrasonicSensor_ISR -** Interrupt service routine to the Ultrasonic Sensor.

2.6.2. Controller System

Semaphore:

- **Accelerometer_Sem -** This Semaphore is used to send a signal to the Accelerometer_Handler thread when there is a new data from the Accelerometer.

- **ButtonModule_Sem -** This Semaphore is used to send a signal to the Button_Handler thread when the On/Off button is Pressed or the Button to choose the controller input.

- **ButtonVoice_Sem -** This Semaphore is used to send a signal to the Voice_Recognition thread when to record a new voice command.

Queue:

- **Send_Accelerometer_Queue** – This message queue is used to send data from the Accelerometer_Handler thread to the Send_Wifi thread

- **Send_Voice_Queue** – This message queue is used to send data from the Voice_Recognition thread to the Send_Wifi thread.

Mutex:

- **CommandInput_mutex** – used to protect the access to the variable that has the command Input mode (accelerometer or voice command).

ISRs:

- **ButtonModule_ISR:** Interrupt service routine to the Touch Button Module.

- **ButtonVoice_ISR:** Interrupt service routine to the to start/stop Recording Button.

- **INT_Accelerometer_ISR**: Accelerometer Interrupt service routine.

2.6.3. Desktop Application

Queue:

- **Update_GUI_Queue -** is used to send the command received from the robot System from the Received_Wifi thread to the Update_GUI Thread.

- **New_Commmand_Queue** – Is used to send the data from the Command_Handler thread to the Send_Wifi thread.

2.7. Robot System Threads Interface

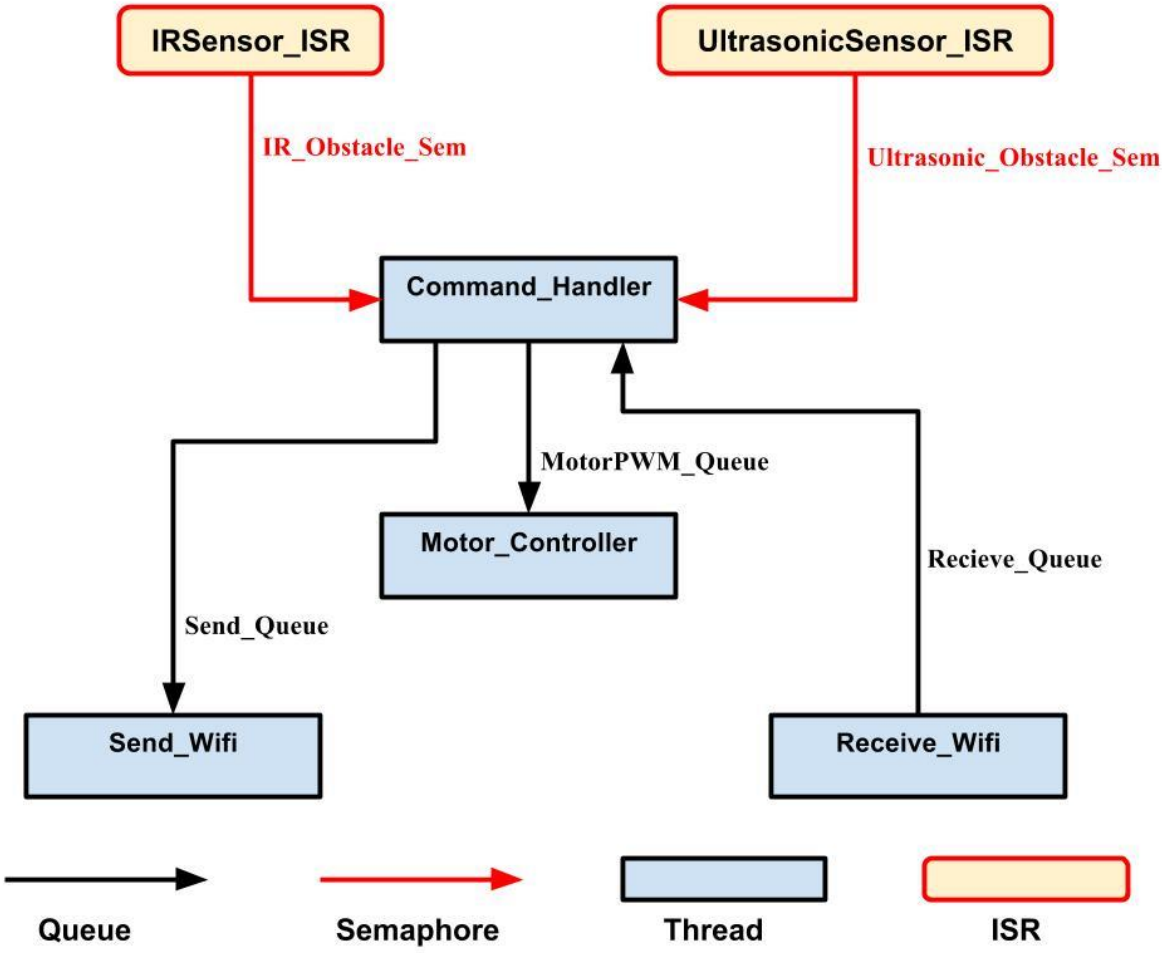

Figure 17: Robot System Threads Interface

2.8. Controller System Threads Interface

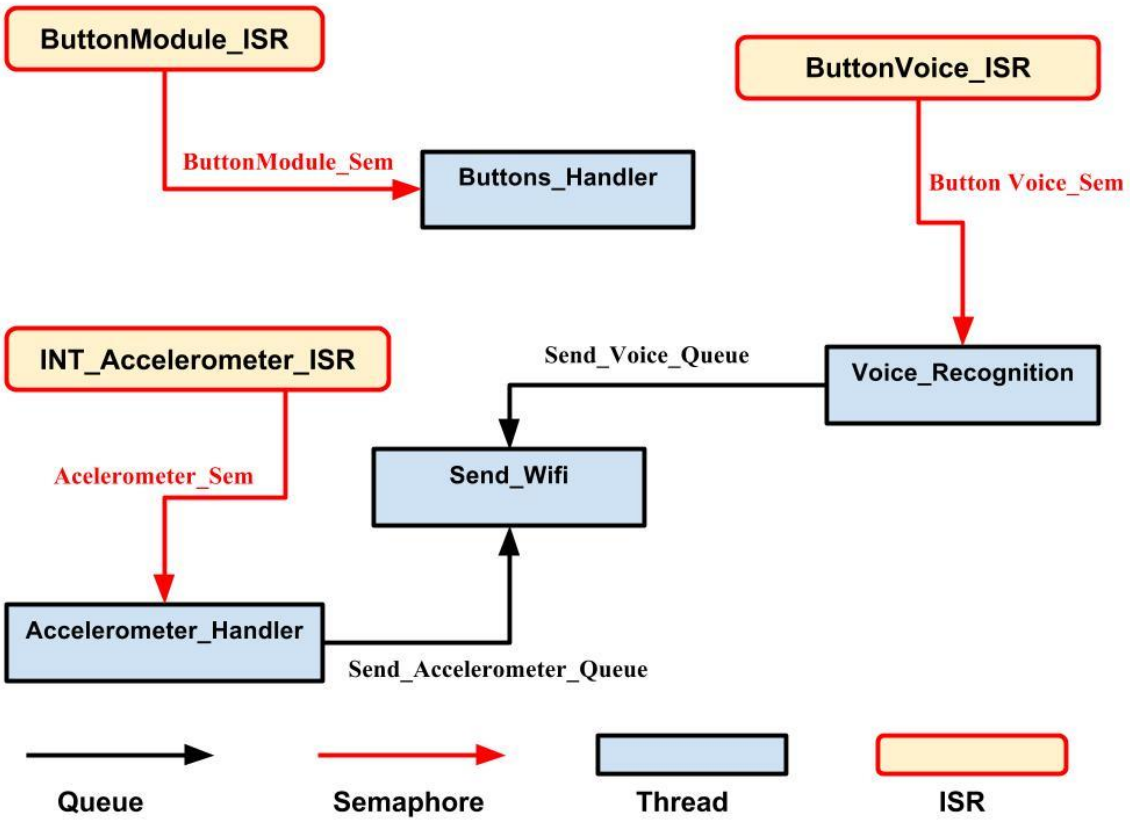

Figure 18: Controller System Threads Interface

2.9. Desktop Application Threads Interface

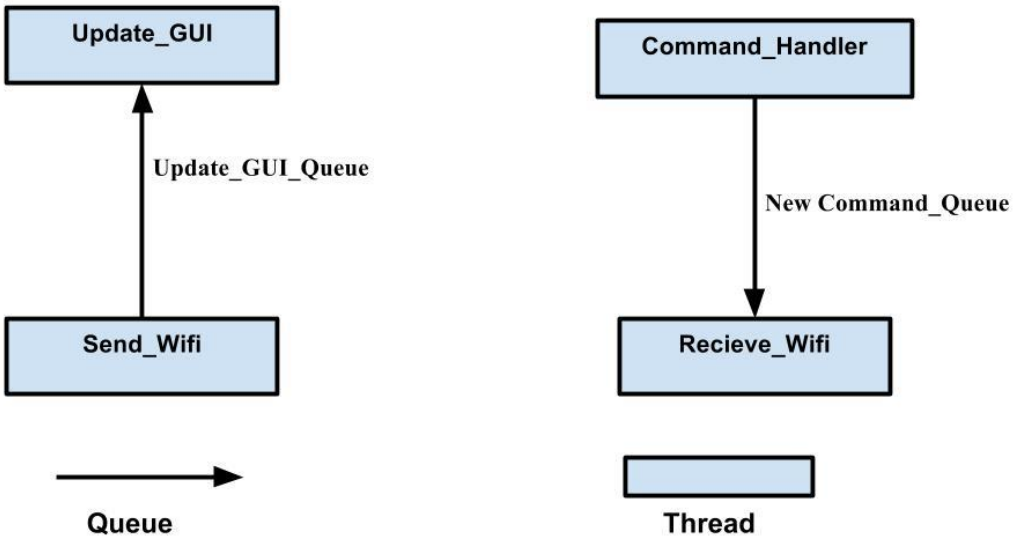

Figure 19: Desktop Application Threads Interface

2.10. System Flowcharts

Flowcharts are used in designing and documenting simple processes or programs. Like other types of diagrams, they help visualize what is going on. There are many different types of flowcharts, and each type has its own repertoire of boxes and notational conventions.

2.10.1. Robot System Flowcharts

Initialization

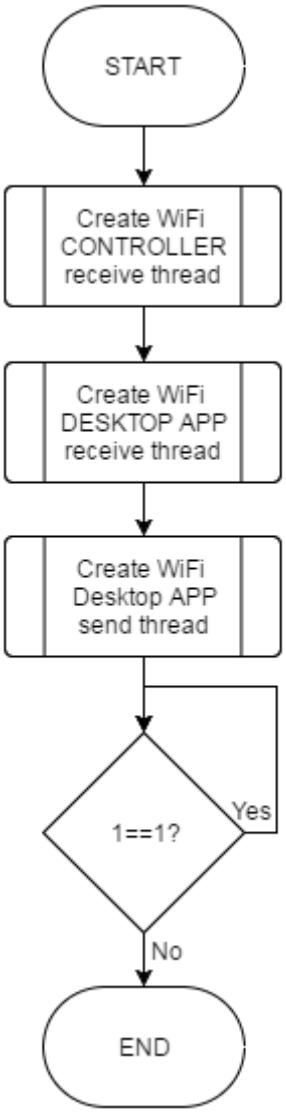

Figure 20: Robot System - Initialization thread

This is the thread that initializes the Robot System. It is responsible to create the "Wi-Fi CONTROLLER receive" thread, the "DESKTOP APP receive" thread and the "DESKTOP APP send" thread. After that, it enters in an infinite loop to avoid the END state.

Wi-Fi Controller Receive

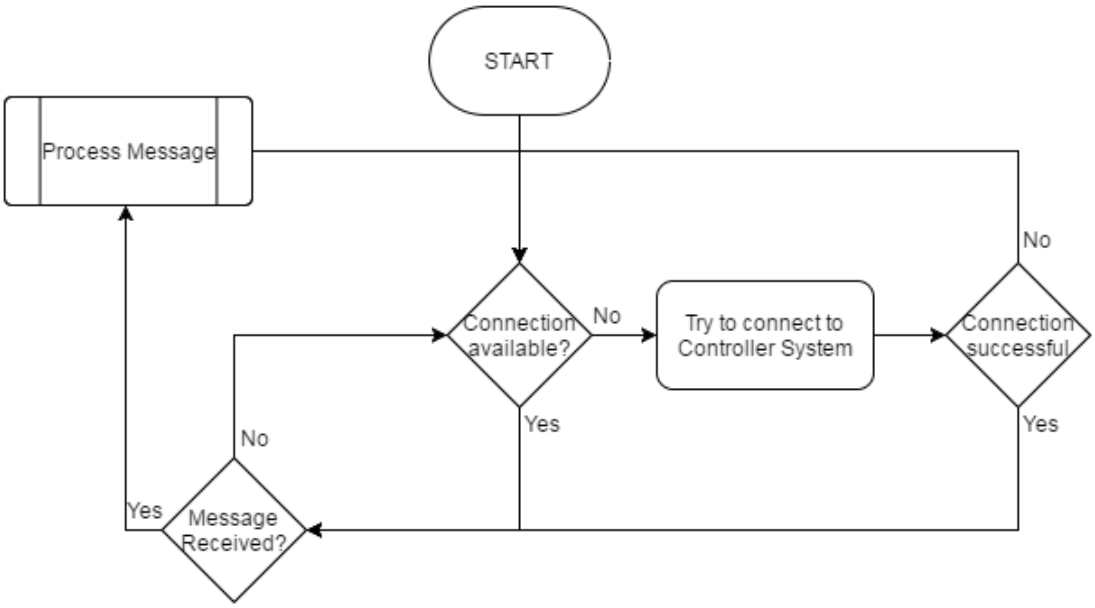

Figure 21: Robot System - WiFi Controller receive thread

This thread is responsible to manage the reception of messages from the Controller System. It checks if a connection is available, and when a message is received, it is processed.

Wi-Fi Desktop Application Receive

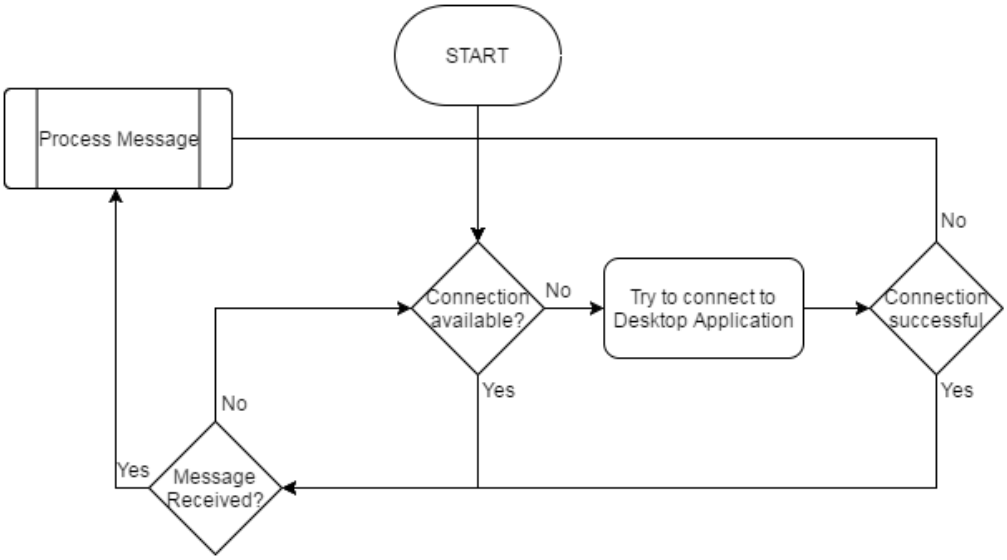

Figure 22: Robot System - WiFi Desktop Application Receive thread

This thread is responsible to manage the reception of messages from the Desktop Application. It tries to connect to the Desktop Application, and when this connection is available and a message is received, it is processed.

Send Wi-fi Thread

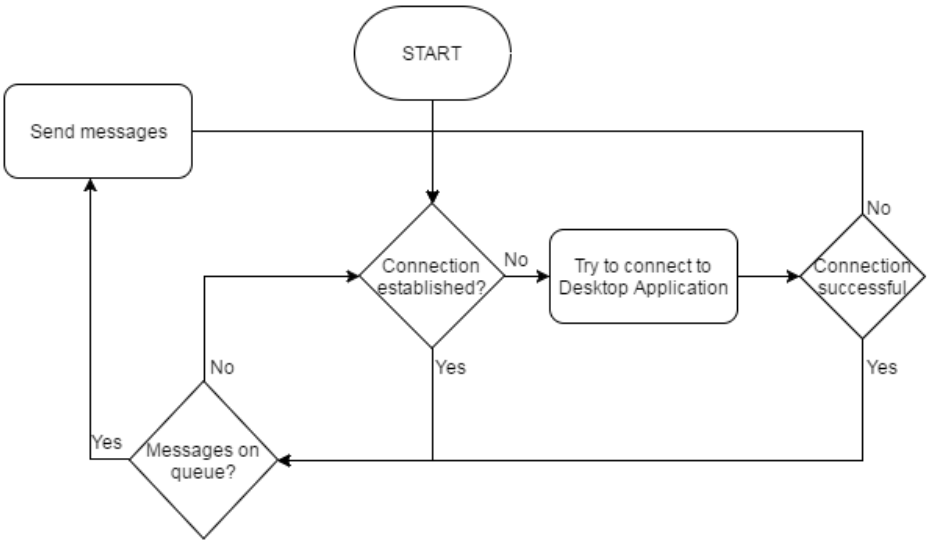

Figure 23: Robot System - Send WiFi thread

This thread is responsible to manage the send of messages to the Desktop Application. It Tries to connect to the Desktop Application, and when a connection is established, it sends the messages in the messages queue.

Command Handler Thread

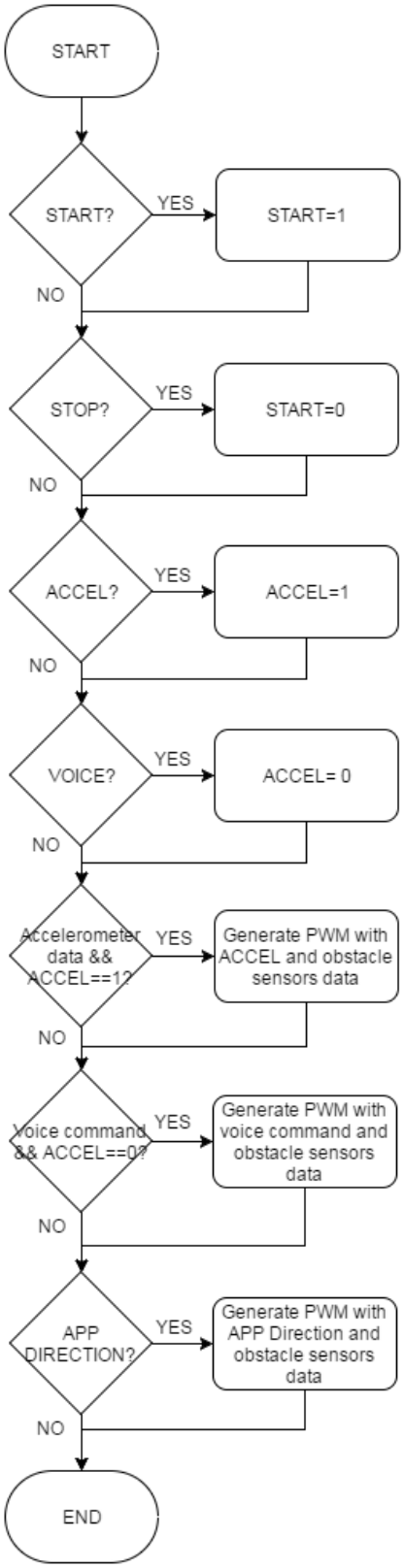

Figure 24: Robot System - Command Handler thread

This thread is responsible to process the messages received. It decodes the message, and performs the corresponding action.

2.10.2. Robot System Thread Priority

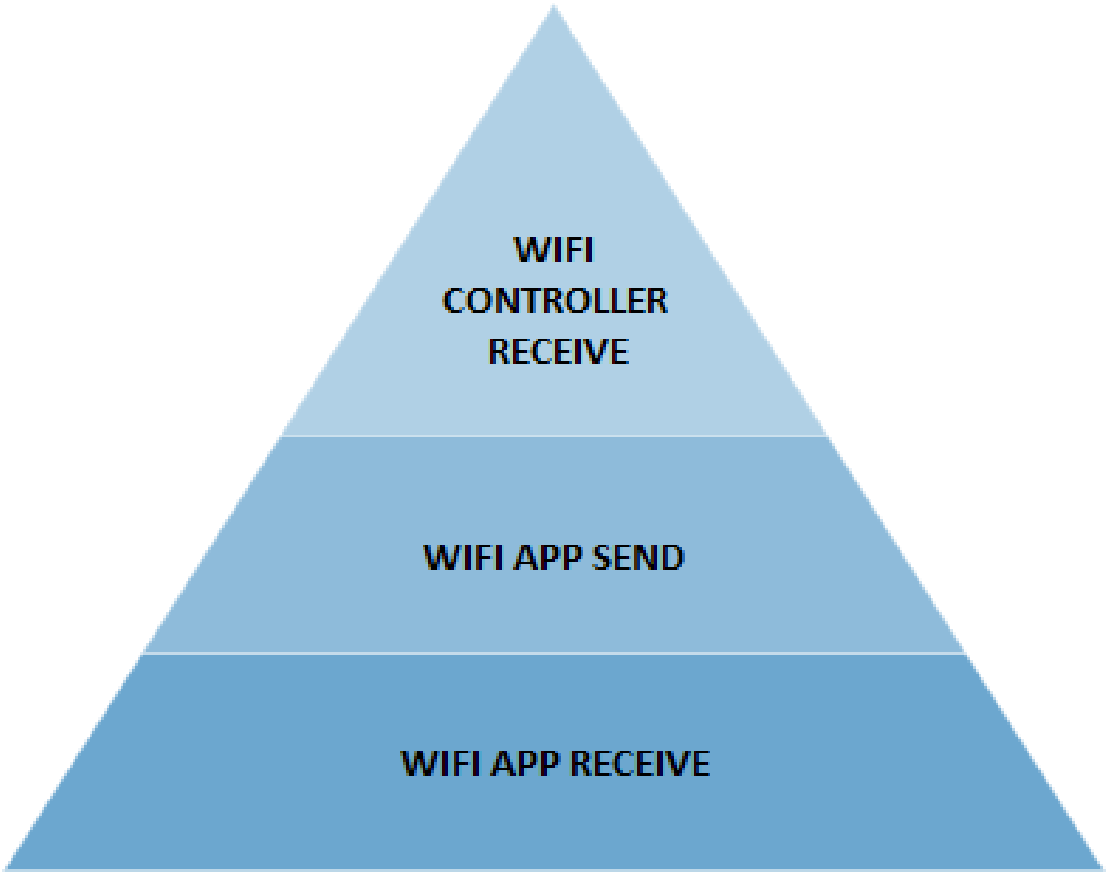

Figure 25: Thread Priority in Robot System

This is the thread priority pyramid for the Robot System. The highest priority thread is the "WIFI CONTROLLER RECEIVE" thread, since the controller is the main controlling device. The medium priority thread is the "WIFI APP SEND" since the main goal of the desktop application is to show the sensors data. The last priority thread is the "WIFI APP RECEIVE", because being able to receive commands from the desktop application isn't very important.

2.10.3. Controller System Flowcharts

Initialization

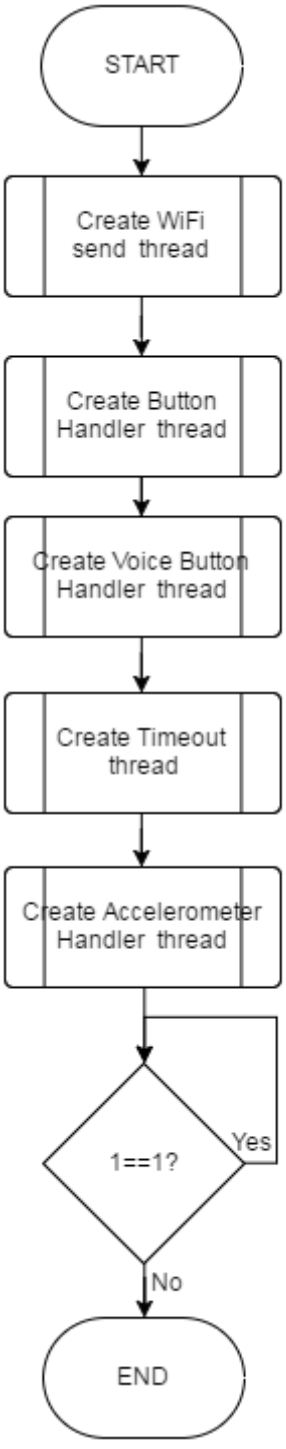

Figure 26: Controller System - Initialization thread

This is the thread that initializes the Controller System. It is responsible to create the "Wi-Fi send" thread, the "START/STOP button" thread, the "ACCEL/VOICE button" thread, the "accelerometer" thread, the "voice"

thread, and the "timeout" thread. After that, it enters in an infinite loop to avoid the END state.

Send Wi-fi Thread

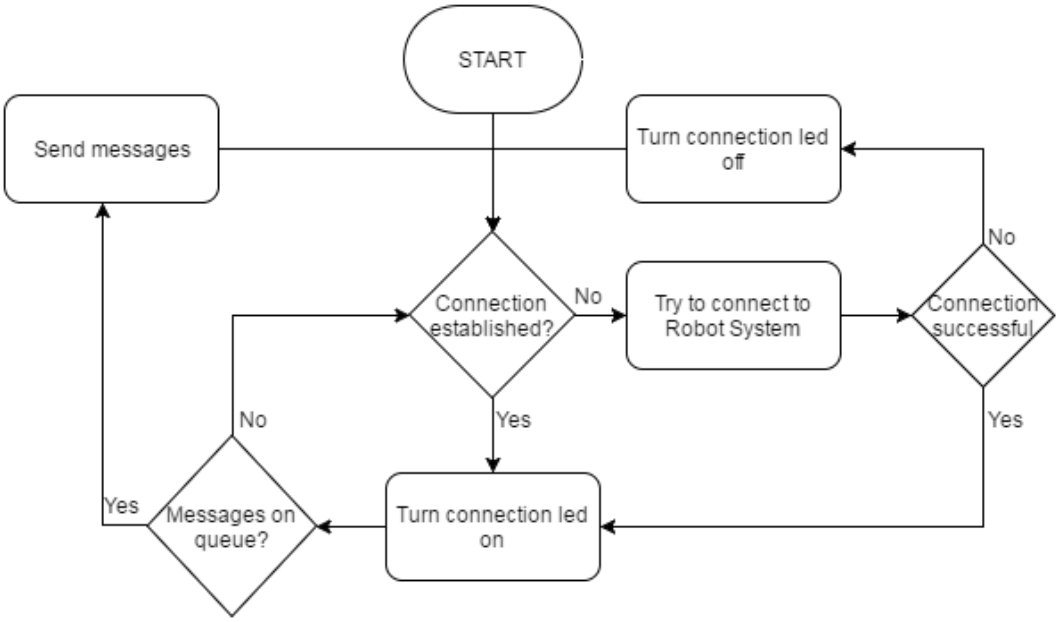

Figure 27: Controller System - Send WiFi thread

This thread is responsible to communicate with the Robot System. It sends data through TCP to The Robot System, when a connection is established and there are messages in the messages queue, with the information about direction of movement and voice commands.

Accelerometer_Handler

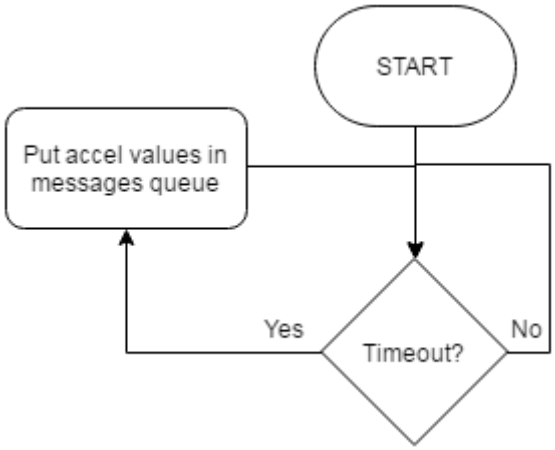

Figure 28: Controller System – Accelerometer Handler thread

This thread waits for the timeout, and when it happens, the accelerometer values are putted in the messages queue, waiting to be send.

Button Handler Thread

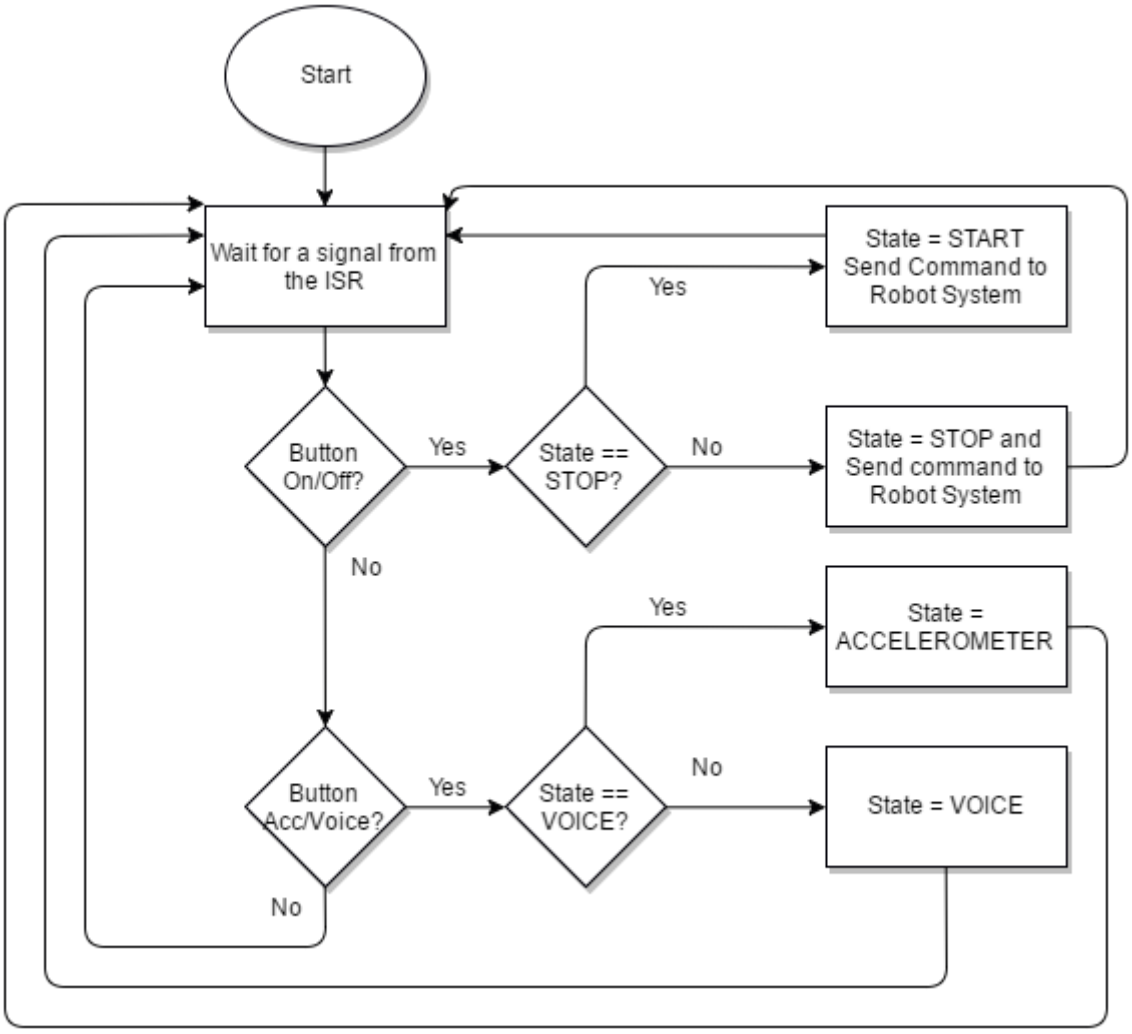

Figure 29: Controller System - Button Handler thread

This thread waits for a signal from the ISR, and according to the button pressed, it will change the state and it will write in the queue que corresponding message

Voice Button Handler

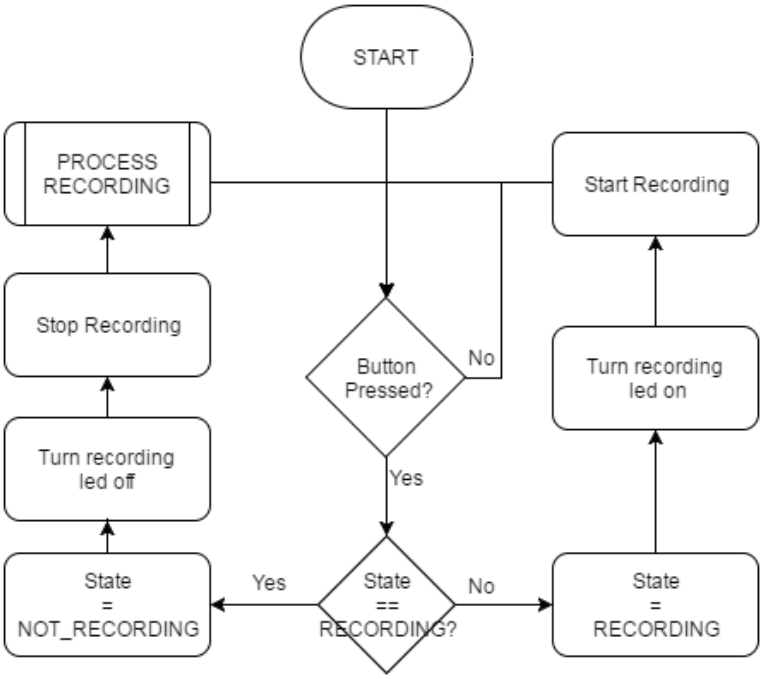

Figure 30: Controller System - Voice Button thread

This thread, waits for the voice button to be pressed to start the recording. Once it is pressed again, the recording stops, and it is processed

PROCESS RECORDING

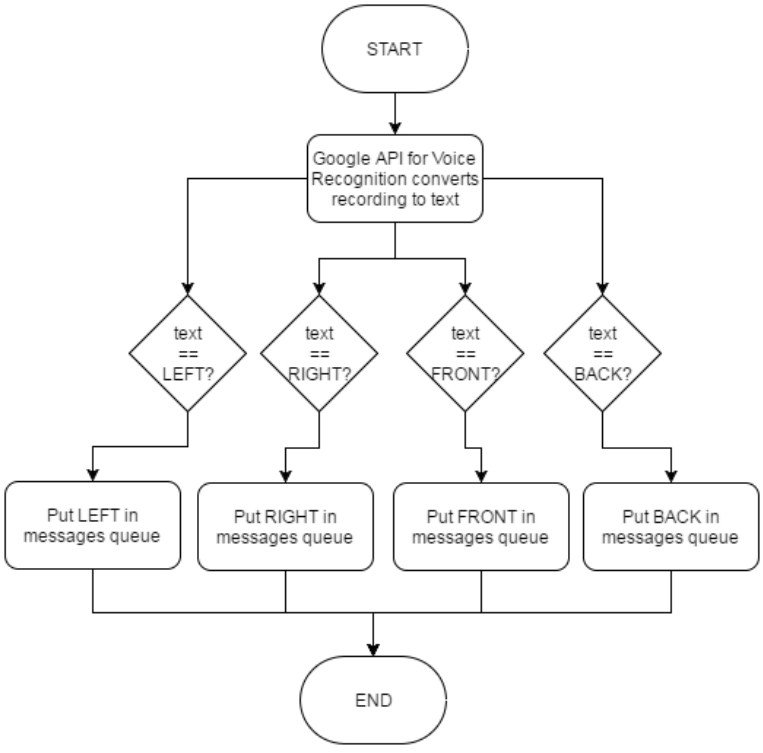

Figure 31: Controller System – Process Recording thread

This thread is responsible for processing the recording using the Google API for Voice Recognition. After the voice being converted to text, if it is a valid command, it is sent in the queue

2.10.4. Controller System Thread Priority

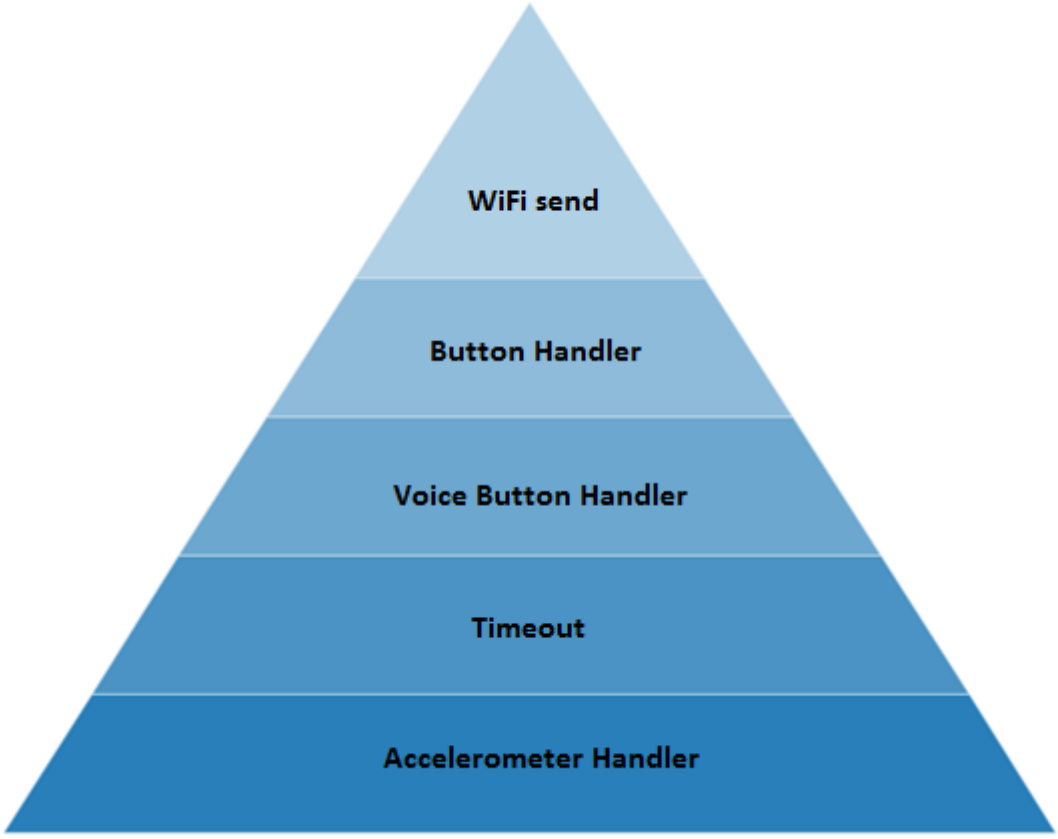

Figure 32: Thread Priority in Controller System

This is the thread priority pyramid for the Controller System. The highest priority thread is the "WiFi send" thread, since without the ability to communicate with the robot, everything else is useless. In second place, comes the "Button Handler" thread, because one of them is like an emergency button to turn the Robot On or Off. In third place, the Voice Button Handler" thread. It has the medium priority. The fourth highest priority is the "Timeout" thread, since it is used for the lowest priority thread "Accelerometer Handler".

2.10.5. Desktop Application Flowcharts

Initialization

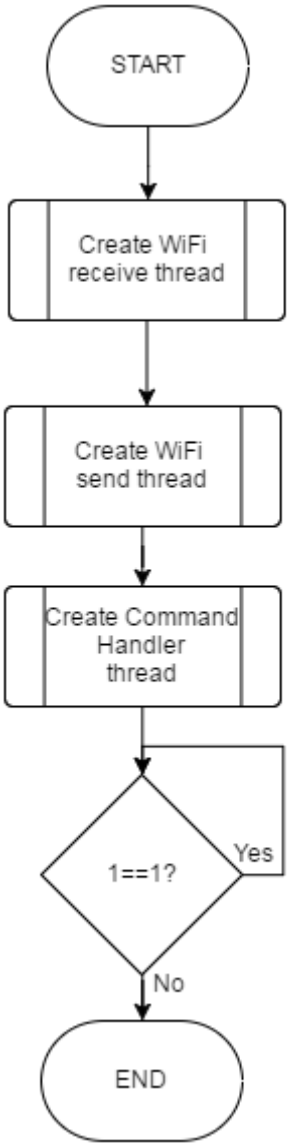

Figure 33: Desktop Application - Initialization thread

This is the thread that initializes the Desktop Application. It is responsible to create the "WiFi receive" thread, the "WiFi send" thread, and the "Command Handler" thread. After that, it enters in an infinite loop to avoid the END state.

Receive Wi-Fi Thread

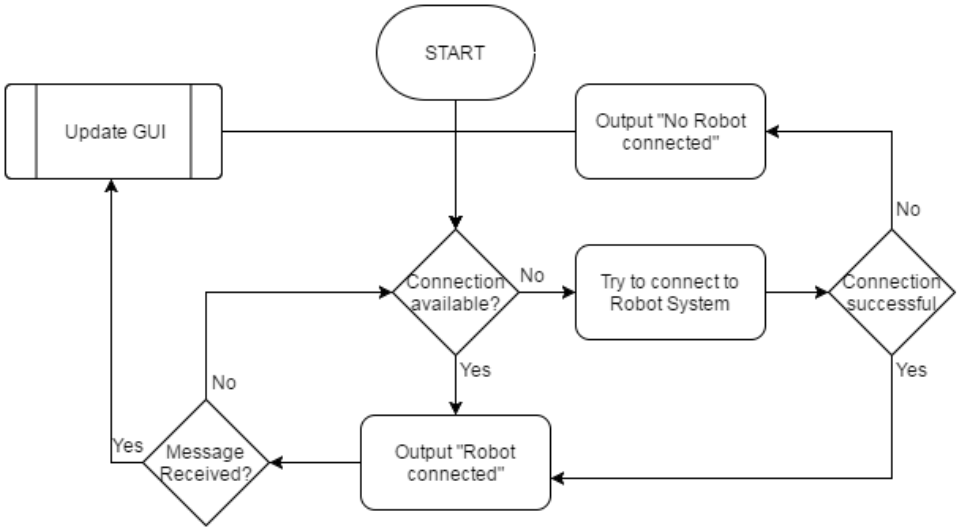

Figure 34: Desktop Application - Receive Wi-Fi thread

This thread is responsible to communicate with the Robot System. It receives data through TCP from the Robot System, when a connection is established it outputs "Robot connected", otherwise it outputs "No Robot connected". This thread is also responsible to update the values of the sensors in the graphical interface, when a message is received from the Robot System.

Send Wi-Fi Thread

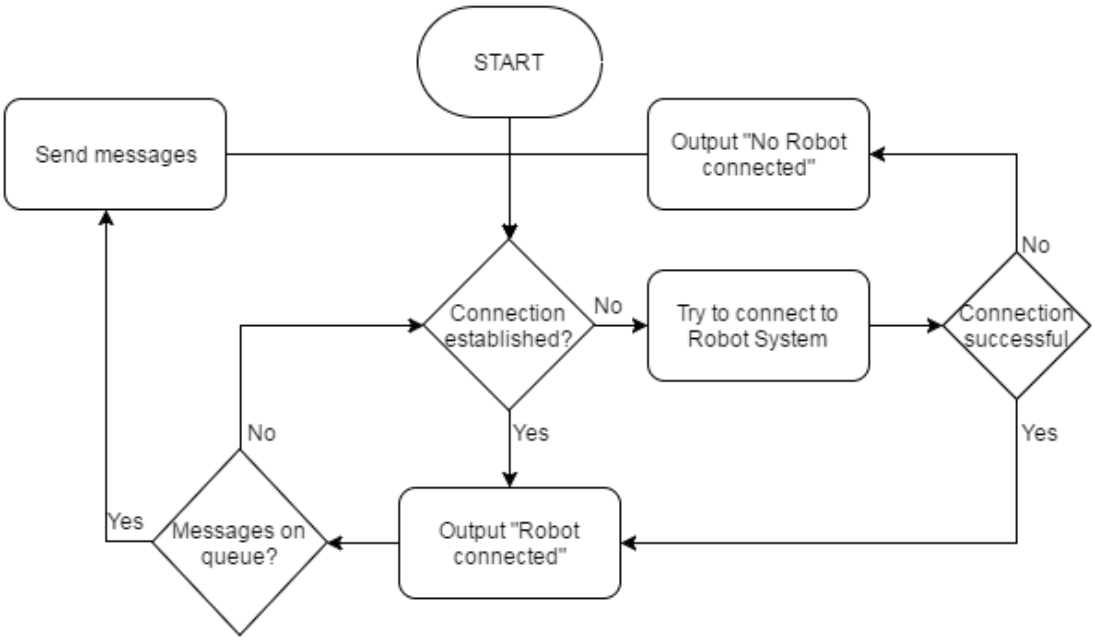

Figure 35: Desktop Application - Send Wi-Fi thread

This thread is responsible to communicate with the Robot System. It sends data through TCP to The Robot System, when a connection is established and there are messages in the messages queue, with the information about direction of movement.

Update GUI

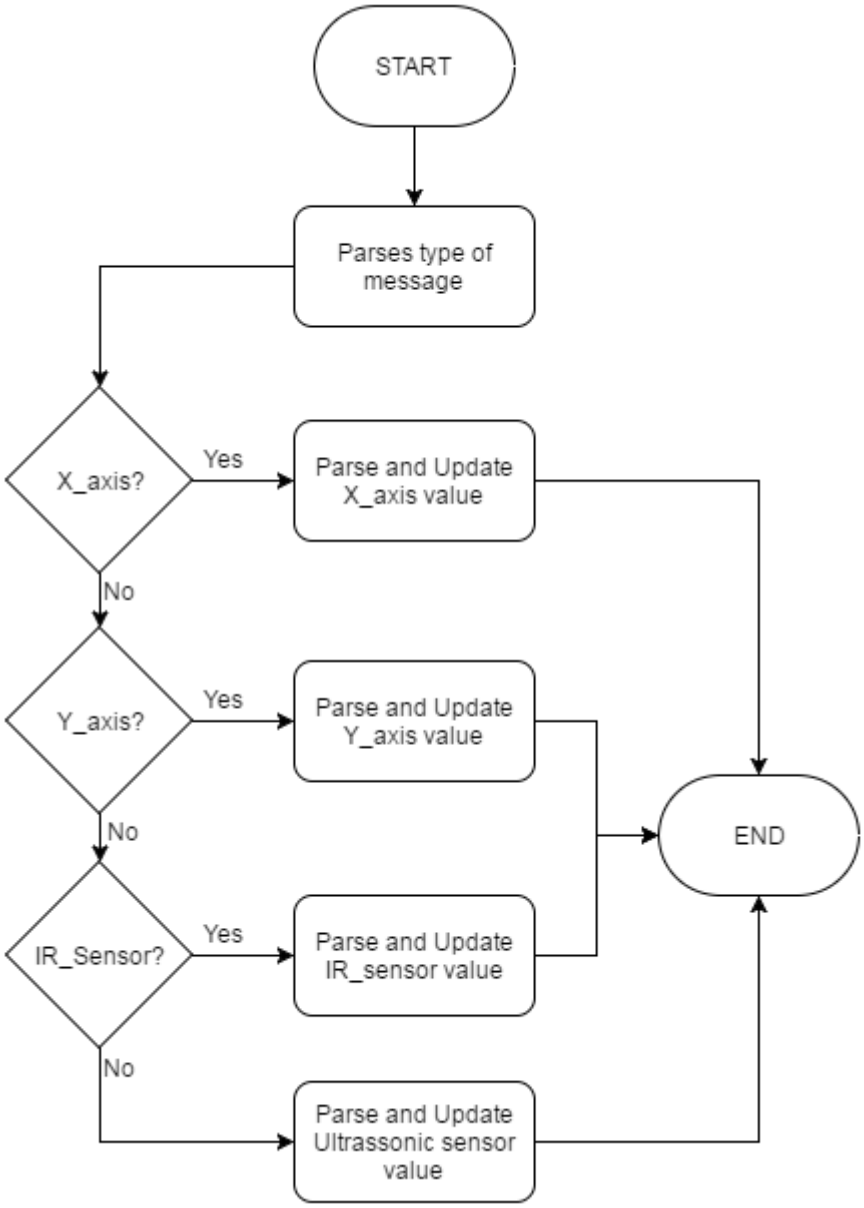

Figure 36: Desktop Application - Update GUI thread

This thread is responsible to update the sensors values. First, the message is parsed to see what kind of information it has. After that the data is parsed, and the Graphical Interface is updated.

Command_Handler

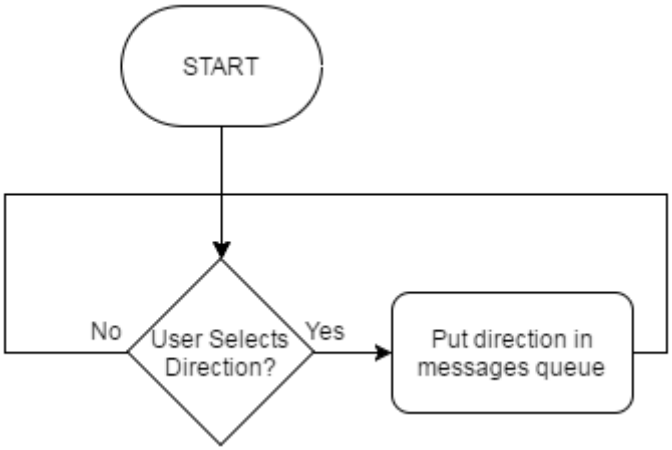

Figure 37: Desktop Application - Command_Handler thread

This thread is waiting for the user to select one of four possible directions, and when he selects one, it is putted in the messages queue, waiting to be send.

2.10.6. Desktop Application Thread Priority

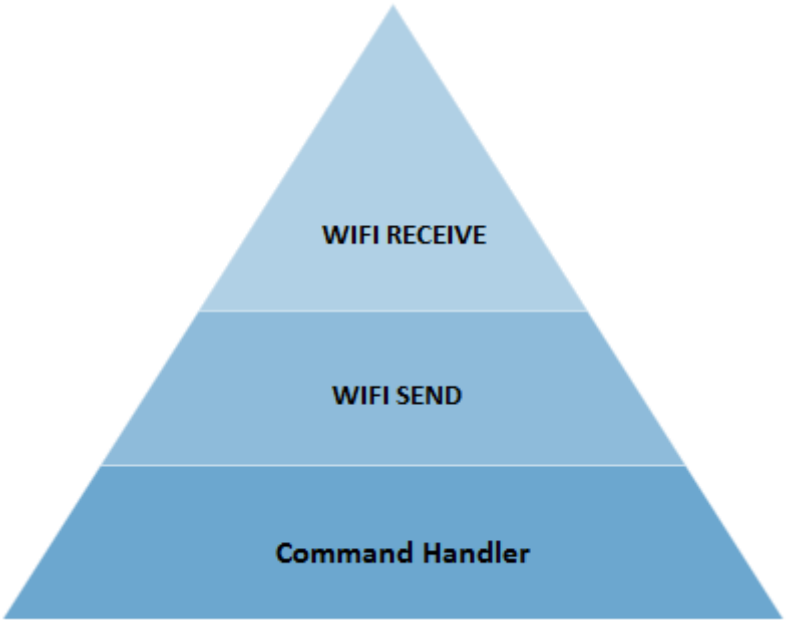

Figure 38: Thread Priority in Desktop Application

This is the thread priority pyramid for the desktop application. The highest priority thread is the "WIFI RECEIVE" thread, since the main goal we want for the desktop application is to show all the sensors data. "WIFI SEND" thread has the medium, priority. The "Command Handler" thread has the lowest priority, since the main controlling mode is using the controller.

2.11. GUI Sketch

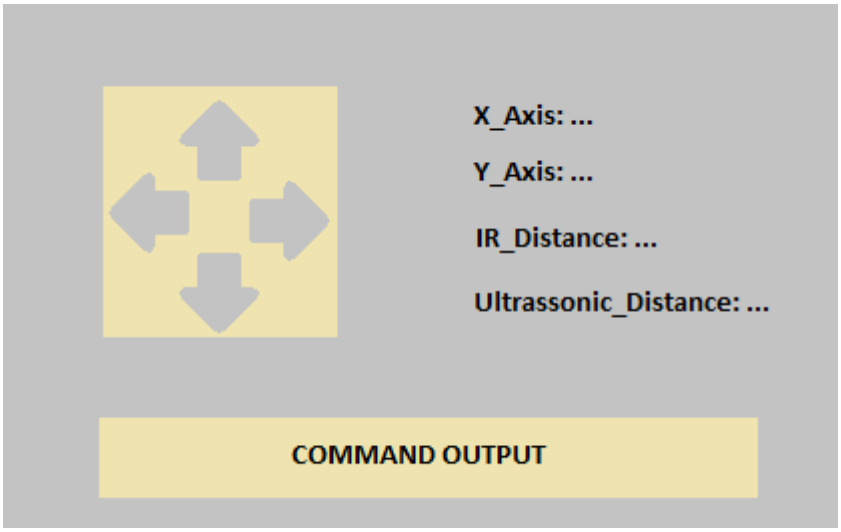

Figure 39: GUI Sketch

This is the preview of the graphical interface in the Desktop Application. The user will be able to Control the Robot System by selecting the moving direction desired. It also will show relevant data relative to the sensors in the Robot System and Controller System, and will output messages about the System, for example, "Front object detected!".

2.12. Gantt Diagram

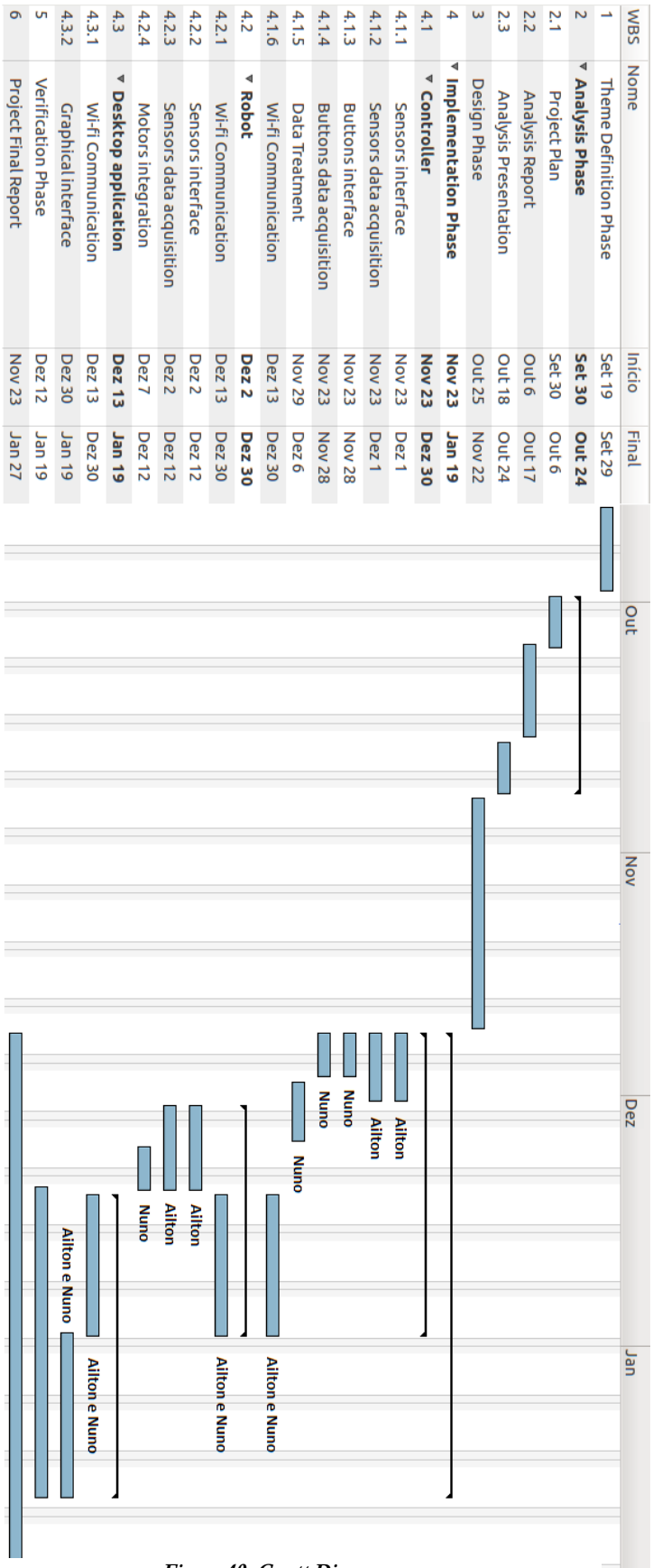

Figure 40: Gantt Diagram

Implementation Phase

1. Robot System

1.1. IR_Interrupt

This interrupt is used in the InfraRed Sensor Pin, and it is fired in both rising and falling transitions.

If the digital read of the Pin is 0, it means that there is a front obstacle, and the robot will be stopped and a sound alert emitted. there is an addictional condition (last), that makes the robot stop only if the car is moving forward.

Otherwhise the buzer sound is turned off.

```
void myInterrupt(void) {

if(digitalRead(IR_PIN)==0) //entrou no obstaculo frontal
{
front_obstacle=1;
if(last)MotorDriver.STOP();
digitalWrite(Buzzer, HIGH);
try{App->send(1);} catch (SocketException &e) {}
}

else //saiu do obstaculo frontal
{
front_obstacle=0;
digitalWrite(Buzzer, LOW);
try{App->send(2);} catch (SocketException &e) {}
}

}
```

Figure 41: IR interrupt

1.2. HandleTCPClient function

This function is used to handle each client connected. It reads the messages received and perform the action respectively.

```
void HandleTCPClient(TCPSocket *sock) {

  cout << "Handling client ";

  try   {cout << sock->getForeignAddress() << ":";}
  catch (SocketException &e) {cerr << "Unable to get foreign address" << endl;}

  try   { cout << sock->getForeignPort();}
  catch (SocketException &e) {cerr << "Unable to get foreign port" << endl;}

  cout << " with thread " << pthread_self() << endl;

  char Buffer[RCVBUFSIZE];
  int recvMsgSize;

  while ((recvMsgSize = sock->recv(Buffer, RCVBUFSIZE)) > 0) { // Zero means end of transmission

  for(int i=0;i<recvMsgSize;i++)
  {
       switch(Buffer[i])
       {
       case '0':            MotorDriver.STOP();                                          break;  //00110000 (48d) ('0') stop
       case '1':            last=1;     if(!front_obstacle) MotorDriver.Move_UP();       break;  //00110001 (49d) ('1') up
       case '2':            last=0;     if(!back_obstacle)  MotorDriver.Move_DOWN();     break;  //00110010 (50d) ('2') down
       case '3':            MotorDriver.Move_RIGHT();                                    break;  //00110011 (51d) ('3') right
       case '4':            MotorDriver.Move_LEFT();                                     break;  //00110100 (52d) ('4') left
       case '5':            App = sock;                                                  break;  //00110101 (53d) ('5') APP connected
       case 128 ... 255:    try{App->send(Buffer[i]);} catch (SocketException &e) {}     break;  //192-255(start)  128-191 (+5)
       }
  }
  }
  // Destructor closes socket
}
```

Figure 42: HandleTCPClient function

1.3. ThreadMain

This thread is used to handle all the clients that try to connect to the robot system. In this case, the Desktop Application and the Controller.

```
void *ThreadMain(void *clntSock) {

  pthread_detach(pthread_self());  // Guarantees that thread resources are deallocated upon return
  HandleTCPClient((TCPSocket *) clntSock);// Extract socket file descriptor from argument

  delete (TCPSocket *) clntSock;
  return NULL;
}
```

Figure 43: ThreadMain thread

1.4. US_Thread

This thread keeps getting the distance from the Ultrassonic Sensor, and if it is lower than 10cm the robot will stop and the buzzer will emit a sound alert. When the distance is bigger than 10cm, the buzzer sound is turned off, unless there is also a front obstacle.

```
void *USThread(void *arg) {

  pthread_detach(pthread_self());  // Guarantees that thread resources are deallocated upon return

  CUltrasonicSensor x;
  pinMode(Buzzer, OUTPUT);

  while(1)
  {
        if(x.Get_Distance_cm(ECHO_PIN,TRIG_PIN)<10)      //back obstacle
        {
        if(back_obstacle==0)MotorDriver.STOP();
        back_obstacle=1;
        digitalWrite(Buzzer, HIGH);
        }

        else                                            //nao tem back obstacle
        {
        back_obstacle=0;
        if (front_obstacle==0) digitalWrite(Buzzer, LOW);
        }

  }

  return NULL;
}
```

Figure 44: USThread thread

1.5. SendUSThread

This thread gets the back obstacle distance and sends it to the Desktop Application every second. We decided to limit the maximum to 63cm, since it is enough, and we have more bits available to send other data.

```
void *SendUSThread(void *arg) {

  pthread_detach(pthread_self());  // Guarantees that thread resources are deallocated upon return

  unsigned char value;
  CUltrasonicSensor x;

  while(1)
        {
        value= x.Get_Distance_cm(ECHO_PIN,TRIG_PIN);
        if (value>63) value=63;
        try{App->send(value+64);} catch (SocketException &e) {printf("No App\n");}
        delay(1000);
        }

  return NULL;
}
```

Figure 45: SendUS thread

3. Controller System

3.1. System Configuration

3.1.1. I2C Configuration

In order to get all the functionalities needed in the project is used the Buildroot.
First since is used the I2C communication to interface the board with the Accelerometer sensor the I2C Device driver was added to the Raspberry Pi kernel build. The image below shows the Configuration made in order to add to the board the I2C device driver.

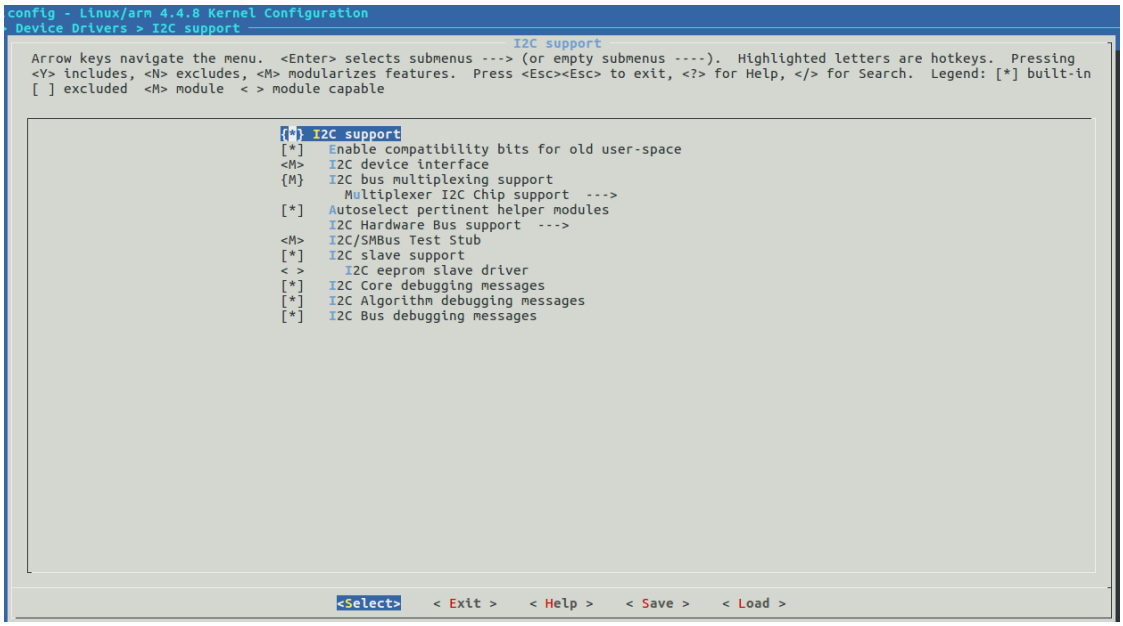

Figure 46: I2C device driver Configuration

After adding the 12C device driver was made a script to load the IC2 device driver to the board removing the need to be manually added every time that the system s started. The image below shows the script added to the to the board to automatically load the I2C device driver. The script was added to the in the etc/init.d directory. The name of the file start with the letter ‘s’ in order for the modules to be added automatically in the system startup.

```
#!/bin/sh
#

modprobe i2c-dev
modprobe i2c-bcm2708
ifup wlan0
```

Figure 47: script to load the I2C drivers

3.1.2. Speech Recognition libraries Configuration

Another Configuration made to the Controller system was adding the library Pocketsphix and Sphinxbase used in the voice recognition. Since the library is not available on buildroot we had to cross compile the library for the board.

Before installing the library its needed to install some dependencies Pulseaudio and ALSA, both of them are available in buildroot as shown in the following figures

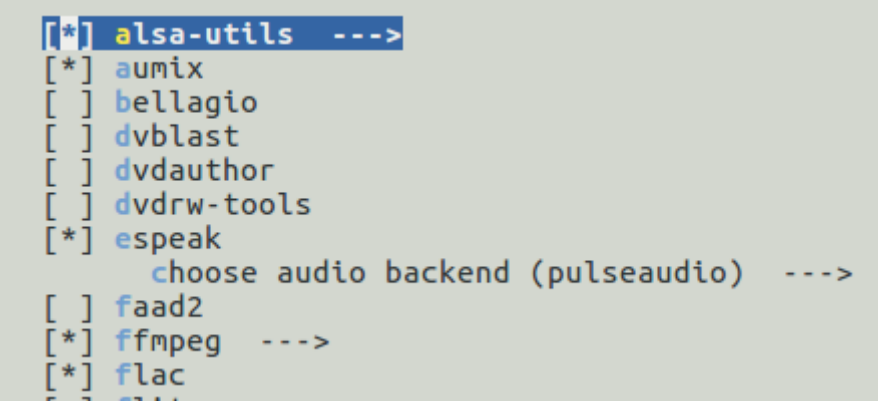

Figure 48: Adding Alsa to the Board Kernel Configuration

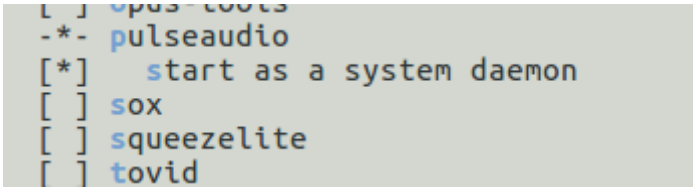

Figure 49: Adding Pulseaudio to the Board Kernel Configuration

After installing the dependences, we are ready to cross compile the library. Since the Pocketsphinx library uses the Sphinxbase library first we cross compile the Sphinxbase library. In oder to cross compile the library we perform the following steps:
Download the latest version of Sphinxbase and Pocketsphinx, we can download it using the CMU website or using the command:

$ wget http://sourceforge.net/projects/cmusphinx/files **/sphinxbase/0.8/ sphinxbase-0.8.tar.gz**

$ wget http://sourceforge.net/projects/cmusphinx/files /pocketsphinx/ **0.8/pocketsphinx-0.8.tar.gz**

After downloading the files, we extract the downloaded files using the following commands:

$ tar -zxvf sphinxbase-0.8.tar.gz; rm -rf sphinxbase-0.8.tar.gz

$ tar -zxvf pocketsphinx-0.8.tar.gz; rm -rf pocketsphinx-0.8.tar.gz

---

After the extraction of the files cd into the Sphinxbase directory and execute the commands shown in the figures below:

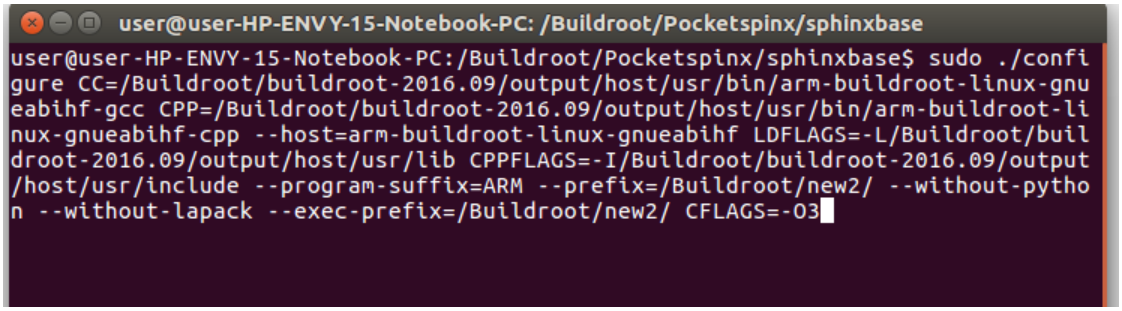

Figure 50: Cross compiling Sphinxbase Library

Normally the command shown in the figure 22 would be **$ ./configure --enable-fixed** but since we are doing the cross compiling of the library we have to give the data about the target system. Looking carefully in the command we can see that it's given all the information of the target system like the gcc compiler used, the g++ compiler, the libraries and includes path, the host etc.

After that the user must execute the following command:

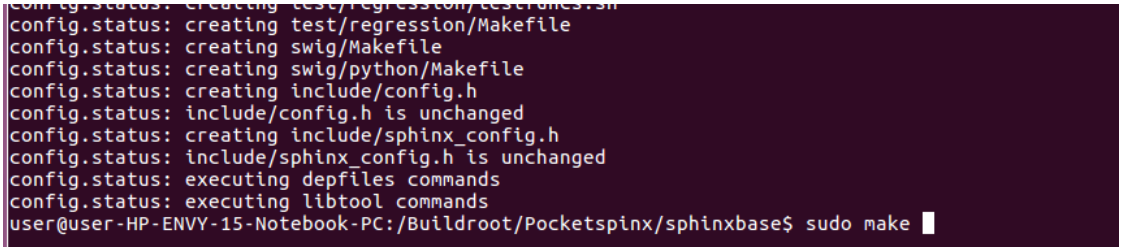

Figure 51: Cross compiling Sphinxbase Library

After making the configure of the installation of the library the user can execute the command make to build the library. If it's not the first time installing the library the user should execute the command $ sudo make clean before making the command in the figute 23.

To finish the installation of the Sphinxbase library the use must execute the command shown in the image below.

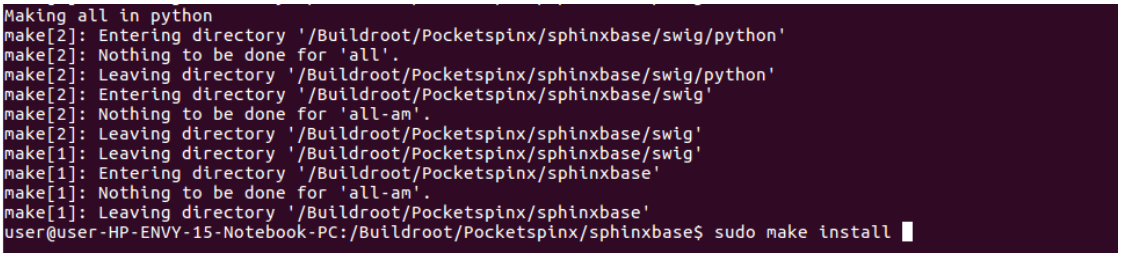

Figure 52: Cross compiling Sphinxbase Library

This command will install the library in the folder usr/loca/lib or in another one if specified by the user in the configure command, for this case the library will be installed in the following path /Buildroot/ARM/lib. The library is installed in the previous path because it was chosen by the user in the configure command.
After the last command the user should repeat the same command to the Pocketsphix library and after that the libraries can be found in the path chosen in the Configuration process. The following image presents the contents of the folder chosen by the user to install the libraries

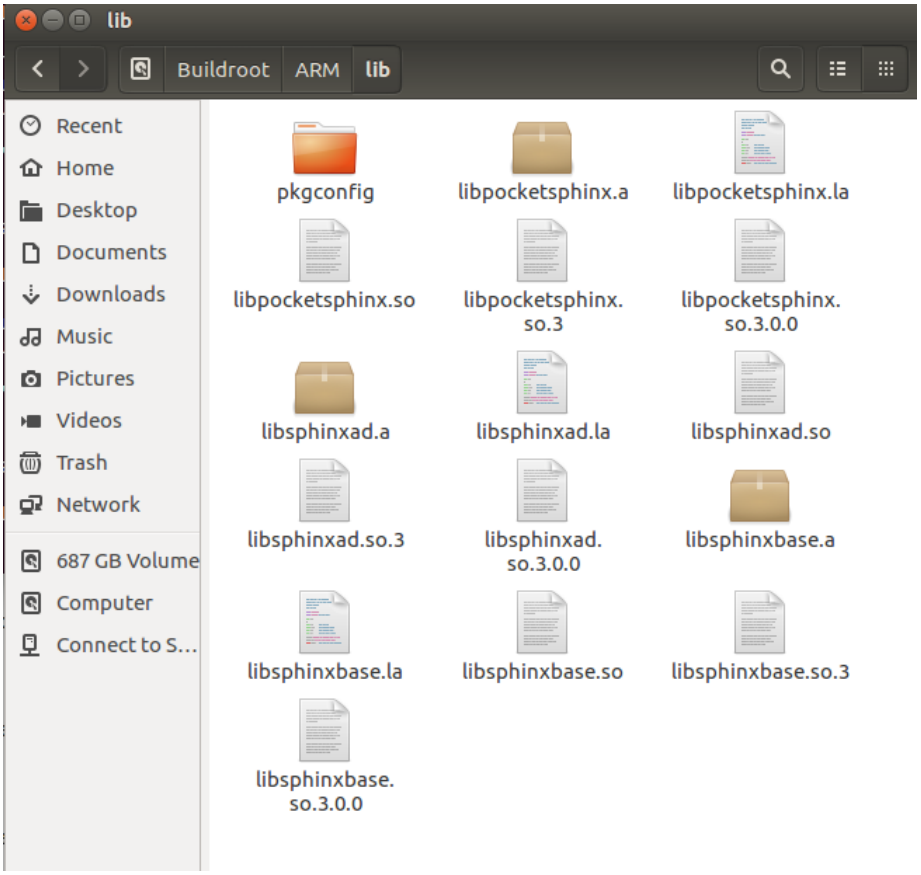

Figure 53: Cross compiled Libraries

Since the libraries installed are shared to allow the execution of a program using the same library we have to install the library in the target system. To install the library in the raspberry pi the we should copy the files .so to the board (usr/lib) and to compile the user should provide the path to the files generated in the cross compiling.
The following image shows how to cross compile using the shared library installed

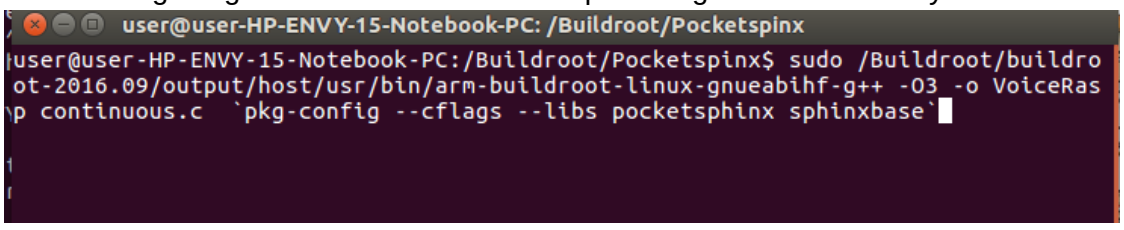

Figure 54: Cross compiling using the shared Libraries installed

3.1.3. Raspberry built-in Wi-Fi Card Configuration

Since the communication between the controller system and the Robot system is through the TCP-IP it's necessary to configure the built-in Wi-Fi card to allow the communication between the two subsystems. In order to configure the wi-fi card we have to install some package using the Buildroot.

First in the buildroot folder we Invoke the Kernel Configuration utility using make linux-menuconfig and had the package shown in the following images:

1- Enable Wireless Networking support under Networking support

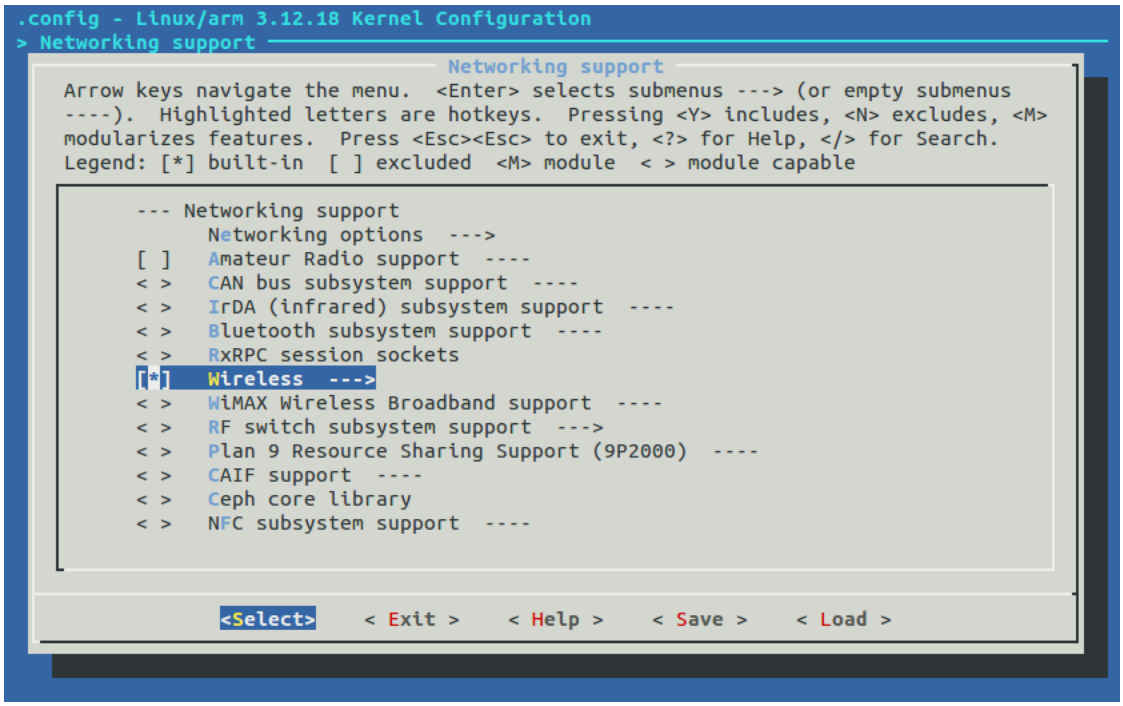

Figure 55: Enable network support in the kernel Configurations

Enable cfg80211 – wireless Configuration API, and Generic IEEE 802.11 Networking Stack (mac80211), under Networking Support, Wireless

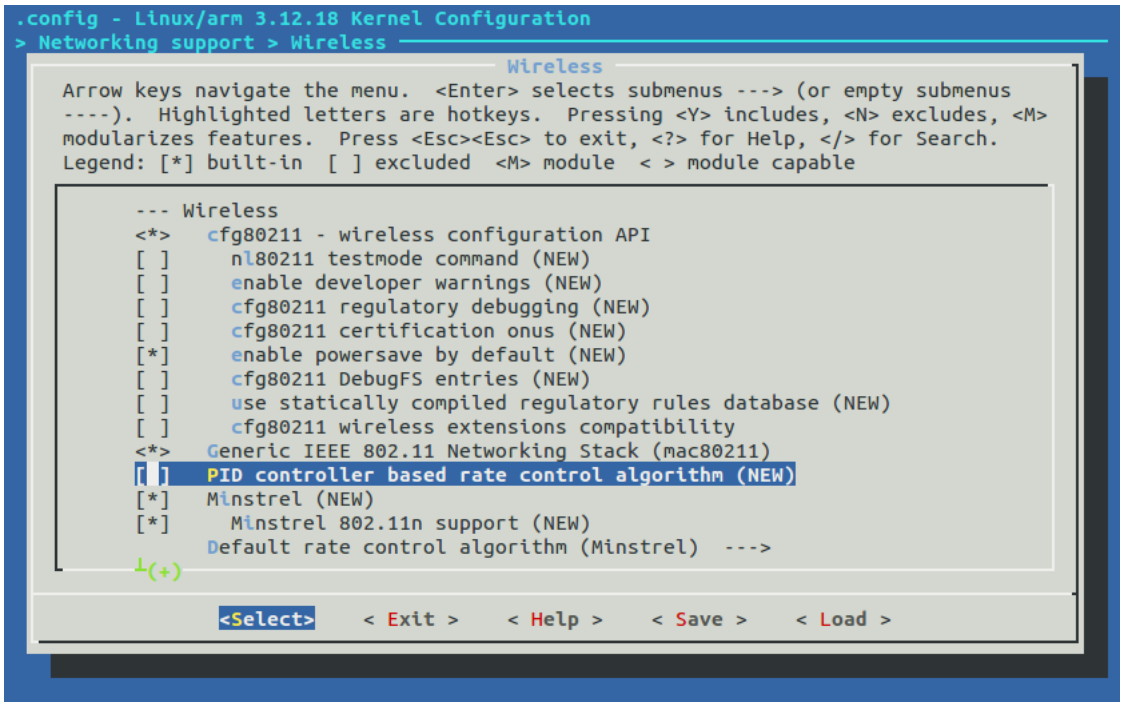

Figure 56: Enable cfg80211 in the kernel Configurations

Enable EEPROM 93CX6 support under Device Drivers, Misc devices, EEPROM support.

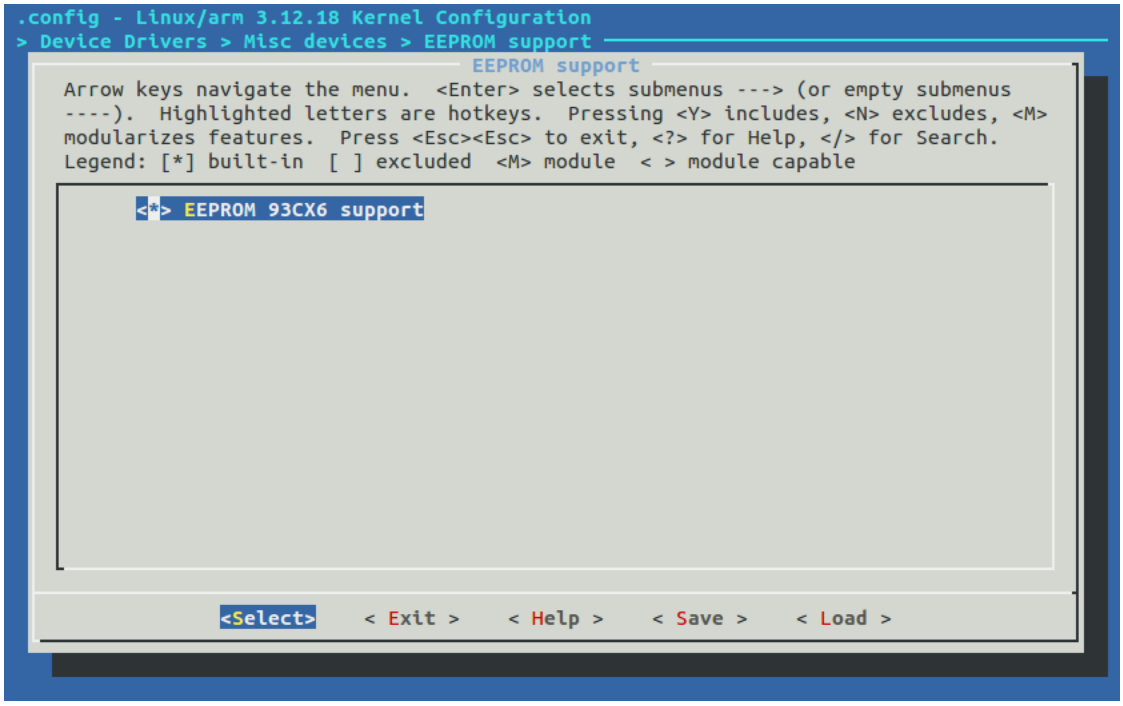

Figure 57: EEPROM 93CX6 in the kernel Configurations

Enable Wireless LAN driver support under Device Drivers, Network device support

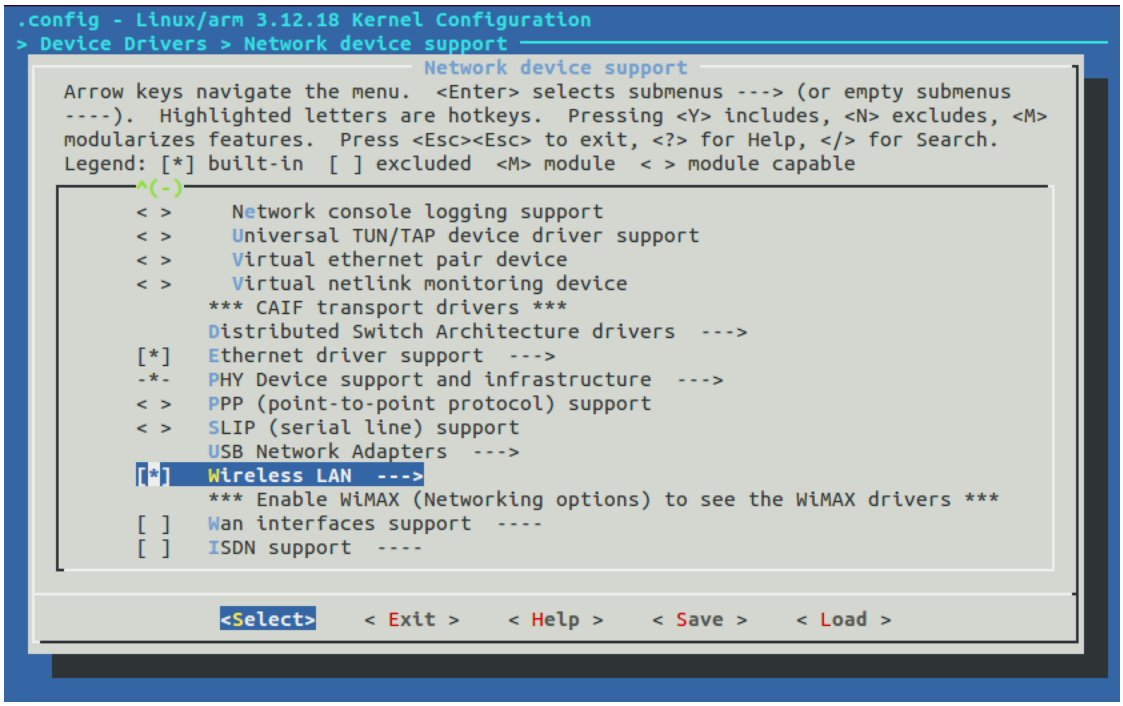

Figure 58: enable Wireless LAN driver support under Device Drivers in the kernel Configurations

Enable Realtek 8187 and 8187B USB support under Device Drivers, Network device support, Wireless LAN. Select the driver appropriate for your adapter

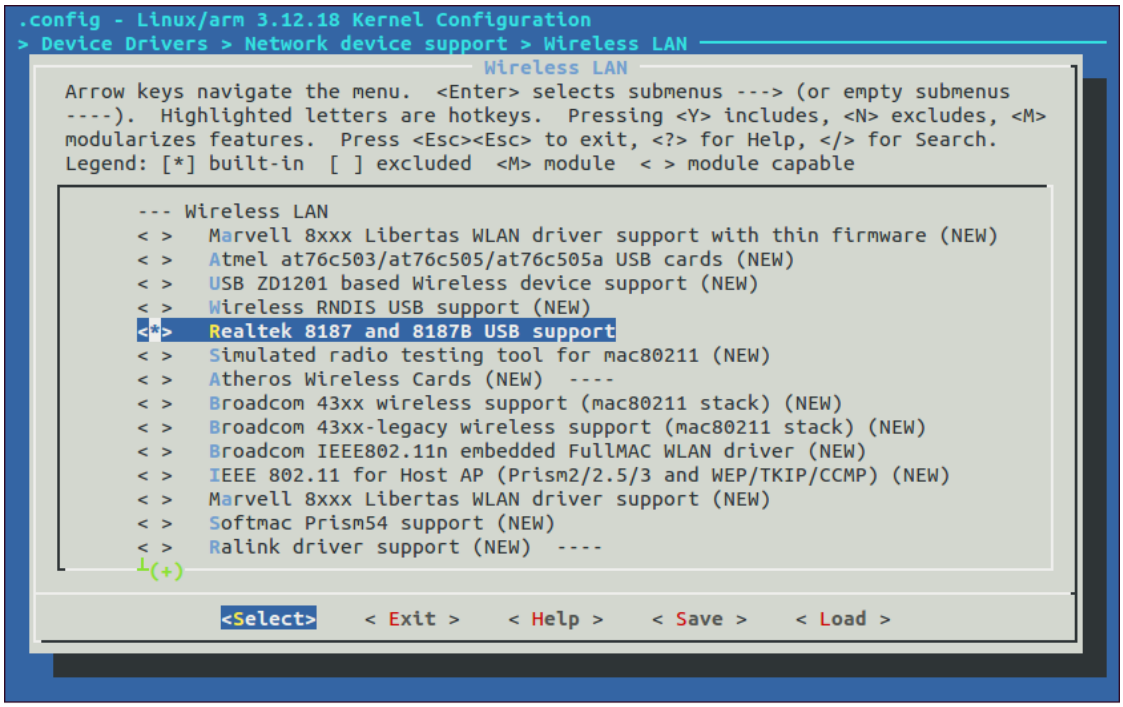

Figure 59: Realtek 8187 and 8187B USB support under Device Drivers in the kernel Configurations

After selecting the previous drivers in the linux-menuconfig we now select some packages in the target packages in menuconfig.
Select package iw required to configure wireless networking, under Target packages, Networking applications. Enable iproute2 if you want to use the ip utility instead of ifconfig.

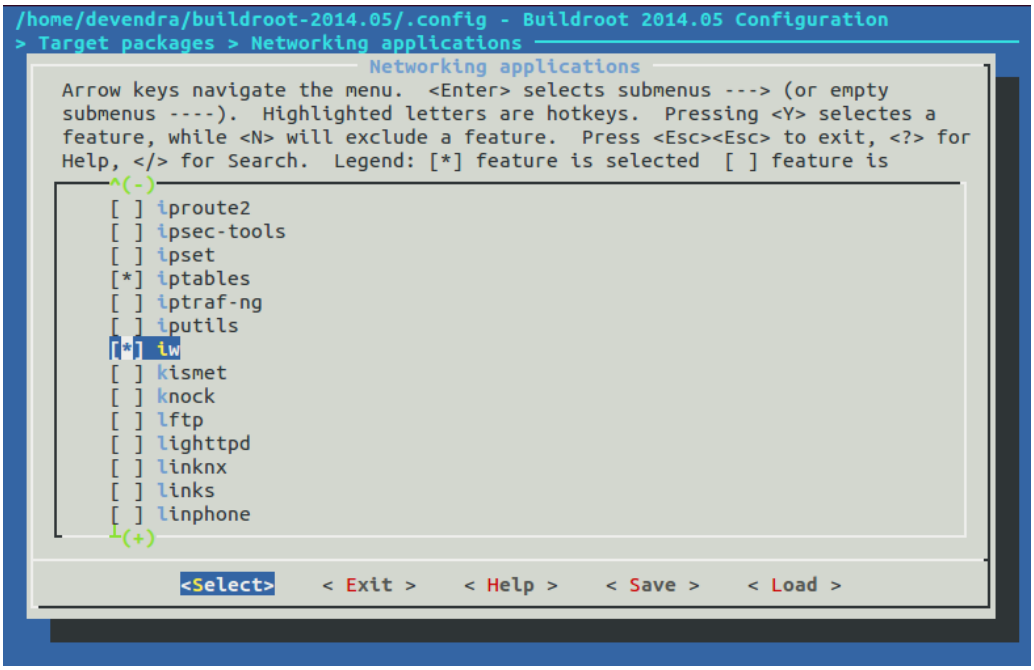

Figure 60: Select package iw the in the kernel Configurations

Select package wpa_supplicant and its sub-packages for WPA/WPA2 support

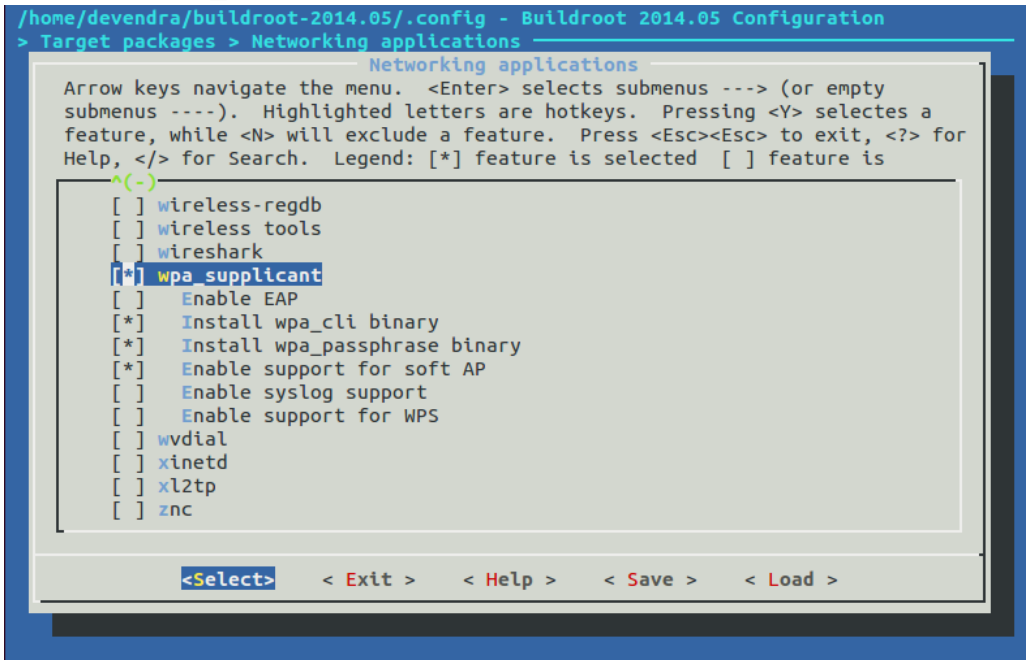

Figure 61: Select package wpa_supplicant the in the kernel Configurations

After selecting the packages perform the build by invoking make, copy the newly minted system to an SD card, and use it to boot up your Raspberry Pi.

After having the SD card done download the firmware from the website **https://github.com/RPi-Distro/firmware-nonfree/tree/master/brcm80211/brcm** and copy it to the folder /etc/network/modules on the raspberry pi.

3.1.4 Wifi dongle Configuration

In this project, we had to use one wi-fi dongle since one of the raspberry's used is version 2B and it doesn't have built in wi-fi board. It was used to establish a wireless connection to exchange data between the three subsystems of our project, the Robot, the Desktop Application and the Controller. It is also useful to send wirelessly the compiled code to the Raspberry board. Since the chipset of our dongle is the RTL8188ucs, we configured the menuconfig the following way.

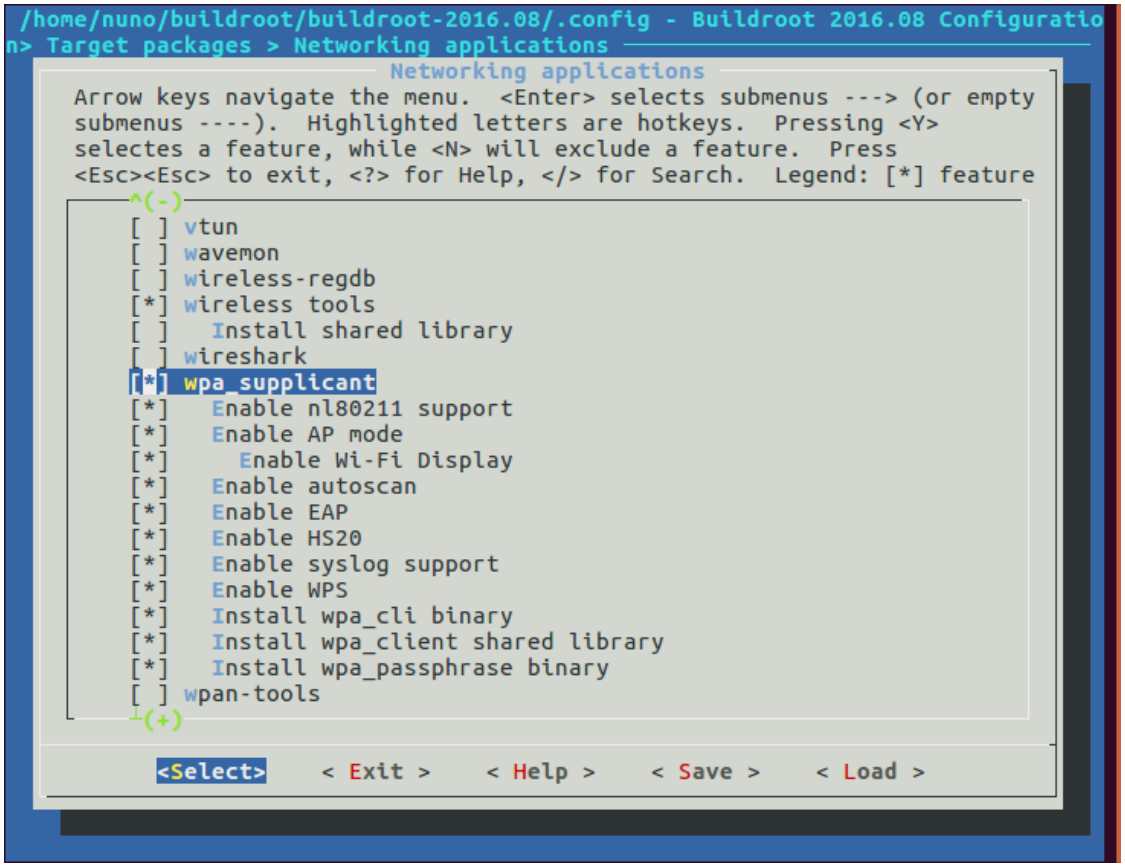

Figure 62: wifi dongle Configuration

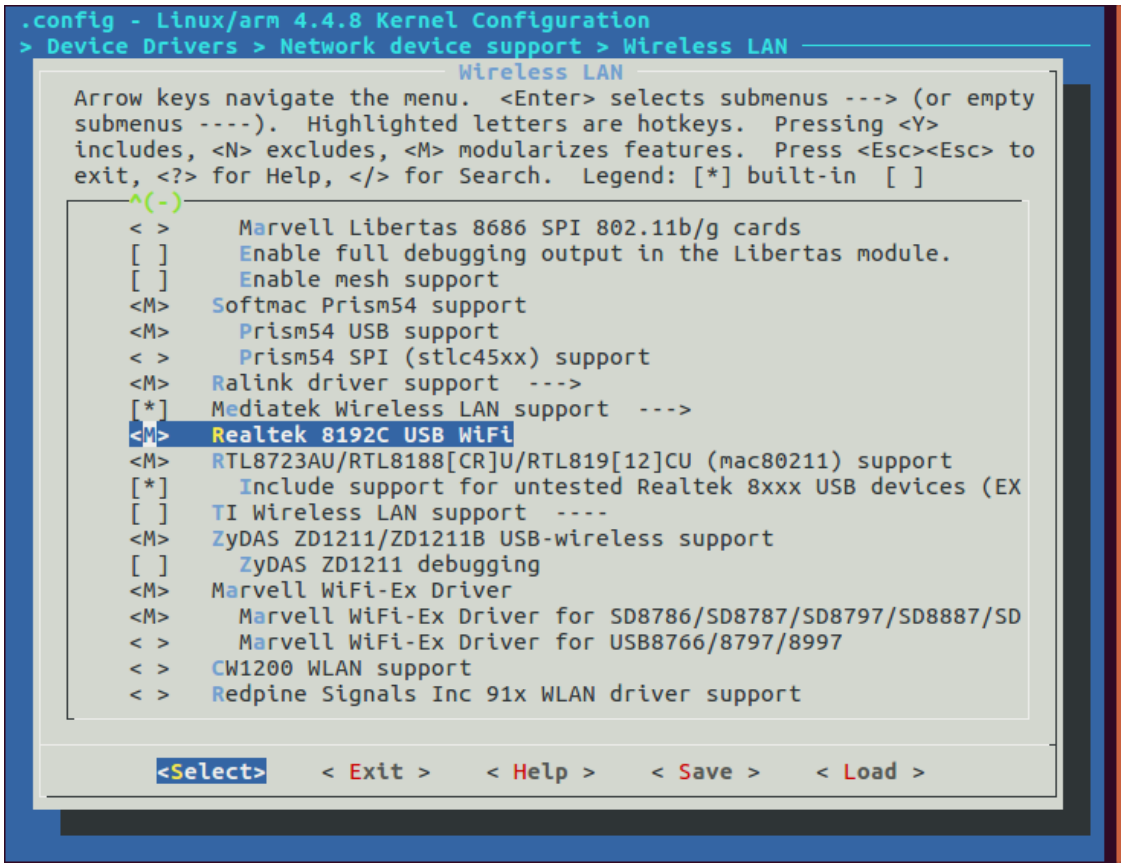

Figure 63: wifi dongle Configuration

After making this Configurations, we need to change the wpa_supplicant.conf file with the SSID and password of the network we want to connect. The interfaces file was also modified to use a static IP address.

```
ctrl_interface=/var/run/wpa_supplicant
ap_scan=1

network={
ssid="myWifi"
scan_ssid=1
proto=WPA RSN
key_mgmt=WPA-PSK
pairwise=CCMP TKIP
group=CCMP TKIP
psk="12345asdq"
#psk=8cfe62b9e1b7fb51e33b593c579076ef556c75b282d1c808f6f400b6427a650b
}
```

Figure 64: wpa_supplicant.conf file

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
        pre-up /etc/network/nfs_check
        wait-delay 15

#wireless

ifup wlan0
auto wlan0
#iface wlan0 inet dhcp
iface wlan0 inet static
address 192.168.43.2
netmask 255.255.255.0
network 192.168.0.0
broadcast 192.168.0.255
gateway 192.168.43.1
```

Figure 65: interfaces file

We also added a script to the etc/init.d directory in our linux partition in order to load the wifi module and connect to the network in the boot.

```
#!/bin/sh
#

modprobe 8192cu.ko
ifup wlan0
```

Figure 66: script to modprobe on boot time

3.1.5 WiringPi configuration

We used the WiringPi library to be able to access the GPIO for several purposes and to use interrupt in the InfraRed pin. Since we don't found this library in our buildroot version, we added it manually.
First, we downloaded from the WiringPi site, the updated library (version 2.32). It comes with the following files.

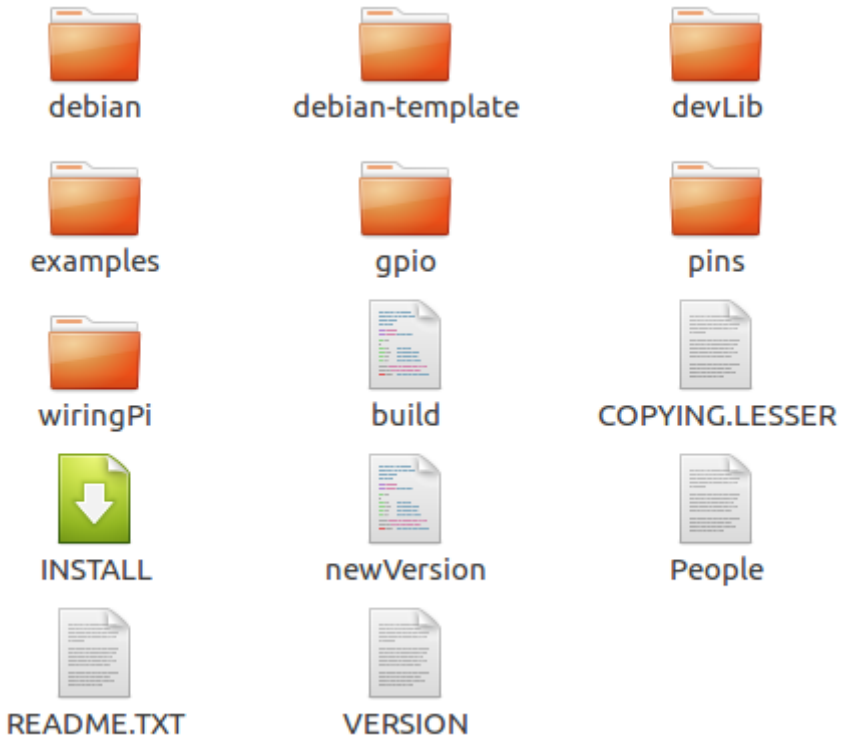

Figure 67: WiringPi files

After this, we create the libwiringPi.a by executing the makefile in the wiringPi folder with the argument static. However, we had to change the compiler in the makefile as following to match the one generated by buildroot.

```
#DEBUG  = -g -O0
DEBUG   = -O2
CC      = arm-buildroot-linux-uclibcgnueabihf-gcc
INCLUDE = -I.
DEFS    = -D_GNU_SOURCE
CFLAGS  = $(DEBUG) $(DEFS) -Wformat=2 -Wall -Winline $(INCLUDE) -pipe -fPIC
```

The same process must be made to generate the libwiringPiDev.a static library in the devLib folder.
Both static libraries are copied to the directory "/home/nuno/buildroot/buildroot-2016.08/output/host/usr/lib/gcc/arm-buildroot-linux-uclibcgnueabihf/4.9.4" and the header files to "/home/nuno/buildroot/buildroot-2016.08/output/host/usr/arm-buildroot-linux-uclibcgnueabihf/sysroot/usr/include"
It is also needed to include the gpio executable file in the Raspberry's /usr/bin folder. This file is generated in the gpio folder, by executing the makefile modified with our compiler.

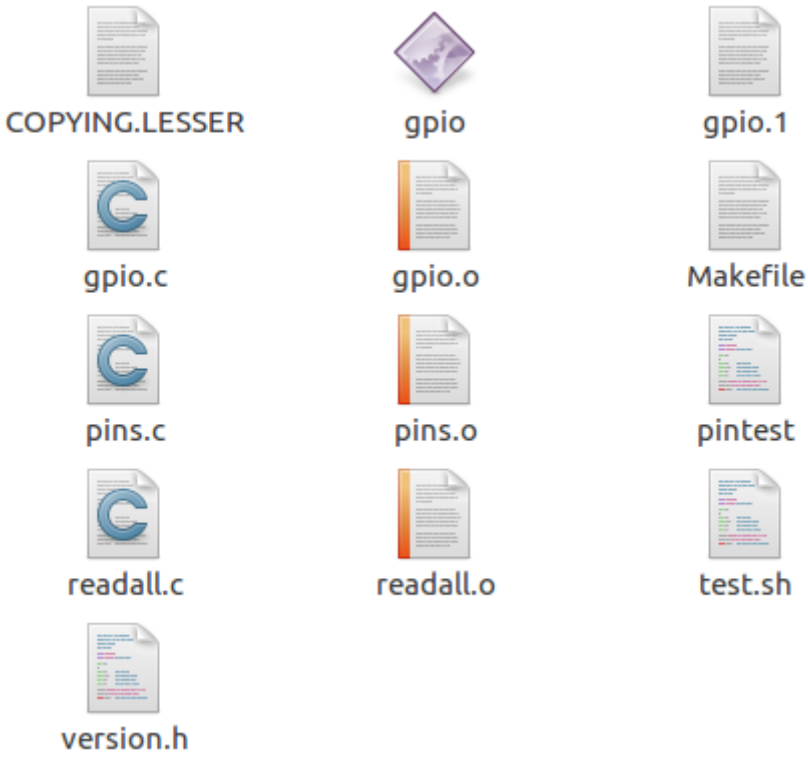

Figure 68: gpio files

3.2. Software Implementation

The software implementation of the Controller system is divided in 4 parts.
The I2C communication, GPIO, ADXL345 and the Voice Command. In this section of the report is presented the implementation of each one of the four subsystems of the Controller system. The following image shows the all the files from the Controller system.

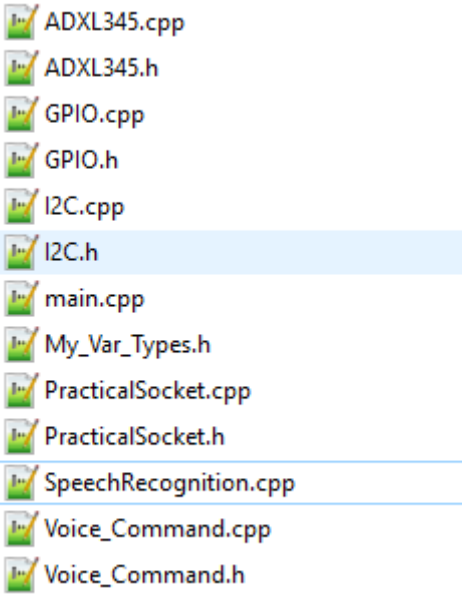

Figure 69: Controller System Files

3.2.1. I2C Communication

The I2C communication is the part of the code that interfaces the Accelerometer sensor using the I2C device driver.
In this section we have the class CI2CCommunication that's is presented below, the class is used to access the Accelerometer sensor using the I2C device Driver.

```
class CI2CCommunication{
    public:
    CI2CCommunication(void);
    CI2CCommunication(char* I2CPort);
    ~CI2CCommunication(void);
    int init(void);
    int getFiledescriptor(void);
    int writeAccelerometer(char * buf, int len);
    int readAccelerometer(short &x , short &y, short &z);
    char* getI2CPort(void);
    int getI2Cfp(void);
    void setI2Cfp (int fp);
    private:
    int fileDescriptor;
    char* I2CPort;
};
```

Figure 70: I2CCommunication class

The class functions members are used to connect, read, and write to the accelerometer sensor using the I2C device Driver, this function will be described below, the function used to connect to the device driver is the int Init(void).
The Init function uses the I2C port ("/dev/i2c-1") to open the I2C device driver, if successful the function returns 0, and if it fails opening the device driver it returns -1.

```
int CI2CCommunication::init(){

    if ((fileDescriptor = open(I2CPort, O_RDWR)) < 0) {
            printf("Failed to open i2c Device Driver\n");
            return -1;
    }
   return 0;
}
```

Figure 71: Init function implementation

Now that we have a way to connect to the device driver, the class also provides another function to read the device driver, the readAccelerometer(short &x , short &y, short &z), the function receives three pointers that are used to return the value of the three axis of

the accelerometer sensor.

```
int CI2CCommunication::readAccelerometer(short &x , short &y, short &z ){
      // check the accelerometer sensor
    if (ioctl(fileDescriptor, I2C_SLAVE, 0x53) < 0){
        fprintf(stderr, "ADXL345 sensor not found3");
        return -1;
    }
    char buf[7];
    buf[0] = 0x32;
    if(writeAccelerometer(buf,2) != 0){
            return -2;
    }
    if (read(fileDescriptor, buf, 6) != 6) {
            printf("Unable to read from ADXL345 Sensor\n");
            return -3;
    }
    else {

        x = (buf[1]<<8) |  buf[0];
            y = (buf[3]<<8) |  buf[2];
            z = (buf[5]<<8) |  buf[4];
    }
   return 0;
}
```

Figure 72: readAccelerometer function implementation

The function first check if the accelerometer sensor is available, if yes, the function writes on the sensors registers and then read the sensor axis, if succeeds the function returns 0, if not the function returns a negative number according to the error. The class also have a member function to write on the accelerometer sensor, the functions writes on the accelerometer sensor, the function implementation is shown below.

```
int CI2CCommunication::writeAccelerometer(char * buf, int len){
    if (write(fileDescriptor, buf, len) != len){
        fprintf(stderr, "Can't write to device\n");
        return -1;
    }
    else
        return 0;
}
```

Figure 73: WriteAccelerometer function implementation

3.2.2. ADXL345 Accelerometer Sensor.

This subsystem of the Controller System it's responsible to read the accelerometer sensor and calculates the hands position and send a command according to the current position to the Robot System.

```
class C3WCR_Accelerometer {
    public:
        C3WCR_Accelerometer(void);
    ~C3WCR_Accelerometer(void);
        C3WCR_Accelerometer(uint8_t address);
    int init(void);
    uint16_t getX(void);
    uint16_t getY(void);
    uint16_t getZ(void);
    int getThreadID(void);
    void printAxis(void);
    int createThread(void);
    private:
        uint8_t devAddr;
        unsigned char AccelerometerX;
    unsigned char AccelerometerY;
    unsigned char AccelerometerZ;
    pthread_t accelerometerHandlerID;
    static void *Accelerometer_HandlerThread(void *I2CPort);
    //CI2CCommunication I2C_1;

};
```

Figure 74: ADXL345 sensor class

The class presented above is composed by two big parts, the int CreateThread (void), that is the function that creates and assign the priority of the Thread Accelerometer_Handler and the Accelerometer_HandlerThread that receives a signal from the timer, reads the sensor and according to the hands position a command is sent to the Robot System.

To send the signal the Accelerometer_HandlerThread is used the function above that interrupt the OS every 10ms

```c
int TimerTick_Configuration(long freqMili){

    timer_t timerid;    struct sigevent sev; struct itimerspec its;
    sigset_t mask;    struct sigaction sa;
    //signal
    sa.sa_flags = SA_SIGINFO;
        sa.sa_sigaction = TimerTick_Handler;
        sigemptyset(&sa.sa_mask);
        if (sigaction(SIG, &sa, NULL) == -1){
            printf("Error Signation!\n");
        return -1;
    }
    //block the signal
    sigemptyset(&mask);
        sigaddset(&mask, SIG);
        if (sigprocmask(SIG_SETMASK, &mask, NULL) == -1){
            printf("Error Blocking the timer signal!\n");
        return -1;
    }
    // Create timer using Real time clock
    sev.sigev_notify = SIGEV_SIGNAL;
        sev.sigev_signo = SIG;
        sev.sigev_value.sival_ptr = &timerid;
        if (timer_create(CLOCKID, &sev, &timerid) == -1){
            printf("Error Blocking the timer signal!\n");
        return -1;
    }
        its.it_value.tv_sec = freqMili / 1000;
        its.it_value.tv_nsec = (freqMili % 1000)*1000000 ;
        its.it_interval.tv_sec = its.it_value.tv_sec;
        its.it_interval.tv_nsec = its.it_value.tv_nsec;
    if (timer_settime(timerid, 0, &its, NULL) == -1){
                printf("Error Setting the timer!\n");
        return -1;
    }
    // Unblock timer signal
    if (sigprocmask(SIG_UNBLOCK, &mask, NULL) == -1){
            printf("Error Unblocking the timer signal!\n");
        return -1;
    }
    return 0;
}
```

Figure 75: Timer Tick function

```
// as the input mode using the buttons
void* C3WCR_Accelerometer::Accelerometer_HandlerThread(void * arg){
    int auxiliar = 0, angle1, angle2;
    short ax, ay, az;
    float AccXangle, AccYangle;
    char send[9];
    while (1){
        // Wait for a signal from the Timer
        sem_wait(&TimerTick_semaphore);

        //check the input mode
        pthread_mutex_lock(&IPC_Thread_Sync.mutex);
        if(IPC_Thread_Sync.value == 2){
            pthread_cond_wait(&IPC_Thread_Sync.cond, &IPC_Thread_Sync.mutex);
        }
        pthread_mutex_unlock(&IPC_Thread_Sync.mutex);

        //printf("**ACC Start**\n");
        I2C_1.readAccelerometer(ax,ay,az);
        AccXangle = (float) (atan2(ay,az)+M_PI)*RAD_TO_DEG;
            AccYangle = (float) (atan2(ax,az)+M_PI)*RAD_TO_DEG;
            if (AccXangle >180)
                    AccXangle -= (float)360.0;

            if (AccYangle >180)
                    AccYangle -= (float)360.0;

        if(SendAngle){
            send [2] = ((ax>>12) & 0xF)  | 0x80; send [1] = ((ax>>6)  & 0x3F) | 0x80;
            send [0] = ((ax)     & 0x3F) | 0xC0; send [5] = ((ay>>12) & 0xF)  | 0x80;
            send [4] = ((ay>>6)  & 0x3F) | 0x80; send [3] = ((ay)     & 0x3F) | 0x80;
            send [8] = ((az>>12) & 0xF)  | 0x80; send [7] = ((az>>6)  & 0x3F) | 0x80;
            send [6] = ((az)     & 0x3F) | 0x80;
            for(int i = 0; i<9; i++){
                //sock.send(&send[i],1);
            }
        SendAngle = false;
        }
        GetMotion(AccXangle, AccYangle);
        printf("**ACC done**\n");
    }
    pthread_exit(NULL);
}
```

Figure 76: Accelerometer_Handler thread

The accelerometer Thread receives a signal from the Timer_Tick function every 10 ms, the signal is sent using a semaphore (**Timer_Tick_Semphore**) calculates the hands position and according to the hands position it sends a a command to the Robot system. To communicate with the Robot system is used a TCP-IP socket. We can see also that is a conditional variable (**IPC_Thread_Sync.cond**) that blocks the thread according to the command input chosen by the user

The IPC IPC_Thread_Sync is and external struct created in the main.cpp function and shared in the ADXL345.cpp and Voice_Command.cpp, the struct is used to synchronize the Accelerometer_Handler thread and the Voice_Command thread

```
struct IPC{
    pthread_mutex_t mutex;
    pthread_cond_t  cond;
    int value;
};
```

Figure 77: Struct IPC

The Function CreateThread is used to create the thread Accelerometer Handler and assign the priority for the thread

```
// the accelerometer thread
int C3WCR_Accelerometer::createThread(void){
    int thread_policy;
    pthread_attr_t thread_attr;
    struct sched_param thread_param;

    pthread_attr_init (&thread_attr);
    pthread_attr_getschedpolicy (&thread_attr, &thread_policy);
    pthread_attr_getschedparam (&thread_attr, &thread_param);

    printf (
        "Default policy is %s, priority is %d\n",
        (thread_policy == SCHED_FIFO ? "FIFO"
        : (thread_policy == SCHED_RR ? "RR"
        : (thread_policy == SCHED_OTHER ? "OTHER"
        : "unknown"))),thread_param.sched_priority);

    SetupThread(60,&thread_attr,&thread_param);

    printf ("Creating thread at RR/%d\n",thread_param.sched_priority);
    pthread_attr_setinheritsched (&thread_attr, PTHREAD_EXPLICIT_SCHED);
    int status_2 = pthread_create (&accelerometerHandlerID, &thread_attr, Accelerometer_HandlerThread, NULL);
    CheckFail(status_2);
    pthread_join (accelerometerHandlerID, NULL);

    TimerTick_Configuration(100);
    sem_init(&TimerTick_semaphore, 0, 0);
}
```

Figure 78: Create Thread function

3.2.3. GPIO - Push Buttons

This section of the controller system controls the buttons input and output. Its responsible to get the users input from the pushbuttons and actuate in the system according to the button pressed.
To be able to accomplish those tasks is used the CButton Class.

```
class CButton{
    private:
    bool m_INPUTMODE;
    bool m_ONOFF;
    pthread_t ButtonHandlerID;
    static void *Button_HandlerThread(void *args);

    public:
    CButton(void);
    ~CButton(void);
    int init(void);
    bool getINPUTMODE(void);
    bool getONOFF(void);
    void changeInputMode(void);
    void sendStartOrStopToRobot(void);
    int  createThread(void);
};
```

Figure 79: CButton Class

The CButton class used a thread to manage the buttons input (Button_Handler Thread)

```
void* CButton::Button_HandlerThread(void* args){
    Button == '0';
    while(1){
        sem_wait(&semaphore);
            switch (Button){
                case '1':{
                    printf("Recieved 1\n");
                    Controller3WCR.sendStartOrStopToRobot();
                    Button = '0';
                    break;
                }
                case '2':{
                    printf("Recieved 2\n");
                    sem_post(&Record_semaphore);
                    Button = '0';
                    break;
                }
                case '3':{
                    printf("Recieved 3\n");
                    Button = '0';
                    pthread_mutex_lock(&IPC_Thread_Sync.mutex);
                    //change the Input mode
                    Controller3WCR.changeInputMode();
                    if(Controller3WCR.getINPUTMODE()){
                        // signal voice Command handler thread
                        IPC_Thread_Sync.value = 2;
                        pthread_cond_signal(&IPC_Thread_Sync.cond);
                    }
                    else{
                        // Signal Accelerometer Handler Thread
                        IPC_Thread_Sync.value = 1;
                        pthread_cond_signal(&IPC_Thread_Sync.cond);
                    }
                    pthread_mutex_unlock(&IPC_Thread_Sync.mutex);
                    break;
                }
                default: Button = '0';
            }
    }
    pthread_exit(NULL);
```

Figure 80: Button_Handler Thread

As the figure above shows the above the Button Handler Thread receives a signal from the ISR when a button is pressed and according to the button pressed it performs a task (ex: change input mode, etc.)
The function create thread, is similar to the one created in the ADXL345 class, it creates a thread and assign it a priority

3.2.3. Voice Command

The voice command section is divided in two parts, the Recognition part and the Voice_CommandHandler thread,

The recognition is done by using the Pocketsphix library and the value is sent to the VoiceCommandHandler thread using a TCP-IP socket.
The voice commandHandler Thread Receives a signal from the button Handler thread when the record button is pressed, and send a message in the socket to the Recognition function, and the recognition function returns the string with the result of the recognition process. When received the string the Voice_CommandHandler uses the CheckSCommand function that will search in the string received the possible command, if found the thread sends the command to the robot system

```
void * CVoiceCommand::VoiceCommandHandler(void *args){
    char send[2];
    char temp[2];
    while(1){
        sem_wait(&Record_semaphore);

        pthread_mutex_lock(&IPC_Thread_Sync.mutex);
        if(IPC_Thread_Sync.value == 1){
            pthread_cond_wait(&IPC_Thread_Sync.cond, &IPC_Thread_Sync.mutex);
        }
        pthread_mutex_unlock(&IPC_Thread_Sync.mutex);

        send [0] = '1';
        send [1] = '1';
        Input_sock.send(&send[0],1);
        memset(&Receive, 0, 32);
        Input_sock.recv(&Receive, 32);
        printf("Received: %s", Receive);
        // find a word in the string related to one of the 5 commands available
        send[1] = checkSCommand(&Received);
        Output_sock.send(&send[1],1);*/
        printf("Voice Command\n");

    }
    pthread_exit(NULL);
}
```

Figure 81: Voice Command Thread

Recognition function

```c
#include <pocketsphinx.h>

int
main(int argc, char *argv[])
{
    ps_decoder_t *ps;
    cmd_ln_t *config;
    FILE *fh;
    char const *hyp, *uttid;
    int16 buf[512];
    int rv;
    int32 score;

    config = cmd_ln_init(NULL, ps_args(), TRUE,
                         "-hmm", MODELDIR "/en-us/en-us",
                         "-lm", MODELDIR "/en-us/en-us.lm.bin",
                         "-dict", MODELDIR "/en-us/cmudict-en-us.dict",
                         NULL);
    if (config == NULL) {
        fprintf(stderr, "Failed to create config object, see log for details\n");
        return -1;
    }

    ps = ps_init(config);
    if (ps == NULL) {
        fprintf(stderr, "Failed to create recognizer, see log for details\n");
        return -1;
    }

    fh = fopen("goforward.raw", "rb");
    if (fh == NULL) {
        fprintf(stderr, "Unable to open input file goforward.raw\n");
        return -1;
    }


     rv = ps_start_utt(ps);

     while (!feof(fh)) {
         size_t nsamp;
         nsamp = fread(buf, 2, 512, fh);
         rv = ps_process_raw(ps, buf, nsamp, FALSE, FALSE);
     }

     rv = ps_end_utt(ps);
     hyp = ps_get_hyp(ps, &score);
     printf("Recognized: %s\n", hyp);
     sock.send(&hyp, sizeof(hyp)); //Send the string in a socket
     fclose(fh);
     ps_free(ps);
     cmd_ln_free_r(config);

     return 0;
}
```

4. Desktop Application

3.1. ConnectToHost

This simple function, is used only to connect to the Robot system, passing the address and port as arguments.

```
bool Client::connectToHost(QString host, int port)
{
    socket->connectToHost(host, port);
    return socket->waitForConnected();
}
```

Figure 82: connectToHost function

3.2. GUI_Update

This function is used to update each field of the interface of the Desktop Application.

```
QString Client::gui_Update(int sensor){

    switch(sensor)
    {
        case 1: return X_Axis;      break;
        case 2: return Y_Axis;      break;
        case 3: return Ultrassonic; break;
        case 4: return Infrared;    break;
    }

}
```

Figure 83: GUI_Update function

3.3. ReadData

This function receives the data from the Robot system and updates the variables received. For the X_Angle and Y_Angle there is a more complex code, since its values are divided in multiple bytes.

```
void Client::readData()
{
  MessageReceived = socket->readAll();


  for(QChar & i : MessageReceived){

    unsigned char val=i.toLatin1();                int x,y,z; float AccXangle=0,AccYangle=0;

    if (val==1)Infrared=" FRONT OBSTACLE!";
    if (val==2)Infrared="";
    if (val>=192) {VALUE=val-192;           count=1;}
    if (val>=64 && val<128) {if(val==127) Ultrassonic= ">=63"; else Ultrassonic=QString::number(val-64);}
    if (val>=128 && val<192) switch (count)
        {
        case 1: VALUE+=(val-128)*64;     count=2;                                  break;
        case 2: VALUE+=(val-128)*4096;   count=3;   x=VALUE-1000;                  break;
        case 3: VALUE=(val-128);         count=4;                                  break;
        case 4: VALUE+=(val-128)*64;     count=5;                                  break;
        case 5: VALUE+=(val-128)*4096;   count=6;   y=VALUE-1000;                  break;
        case 6: VALUE=(val-128);         count=7;                                  break;
        case 7: VALUE+=(val-128)*64;     count=8;                                  break;
        case 8: VALUE+=(val-128)*4096;   count=0;   z=VALUE-1000;
        AccYangle = (float) (atan2(x,z)+M_PI)*RAD_TO_DEG;
        AccXangle = (float) (atan2(y,z)+M_PI)*RAD_TO_DEG;
        if (AccXangle >180) AccXangle -= (float)360.0;
        if (AccYangle >180) AccYangle -= (float)360.0;
        Y_Axis=QString::number(AccYangle);X_Axis=QString::number(AccXangle);    break;
        }

        qDebug() << "x="<<x<<"y="<<y<<"z="<<z<< "val=" << val << "count"<< count << "VALUE" << VALUE;/*i=256;*/
     }
}
```

Figure 84: ReadData function

3.4. WriteData

This function is responsible to write data to the socket, to enable the sending of messages to the robot.

```
bool Client::writeData(QByteArray data)
{
    if(socket->state() == QAbstractSocket::ConnectedState)
    {
        socket->write(data); //write the data itself
        return socket->waitForBytesWritten();
    }
    else
        return false;
}
```

Figure 85: WriteData function

Test Cases

Front obstacle detected, 25cm back obstacle.

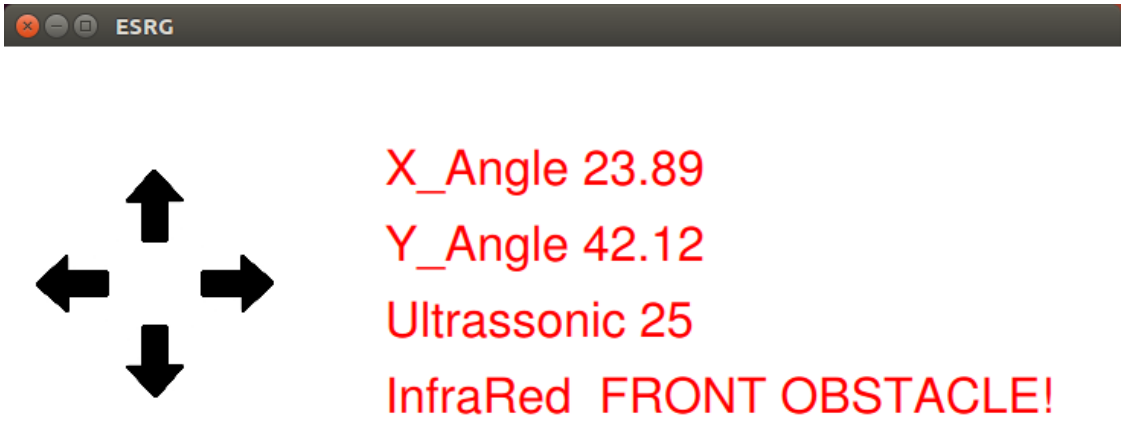

Figure 86: Front Obstacle Test Case

Front obstacle not detected, 25cm back obstacle.

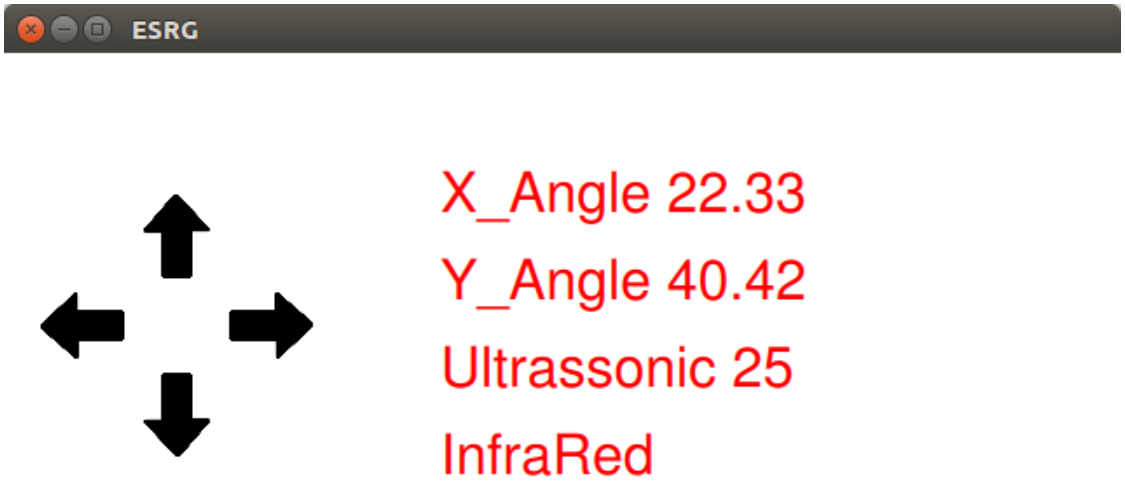

Figure 87: No Front Obstacle Test Case

Robot Final Prototype

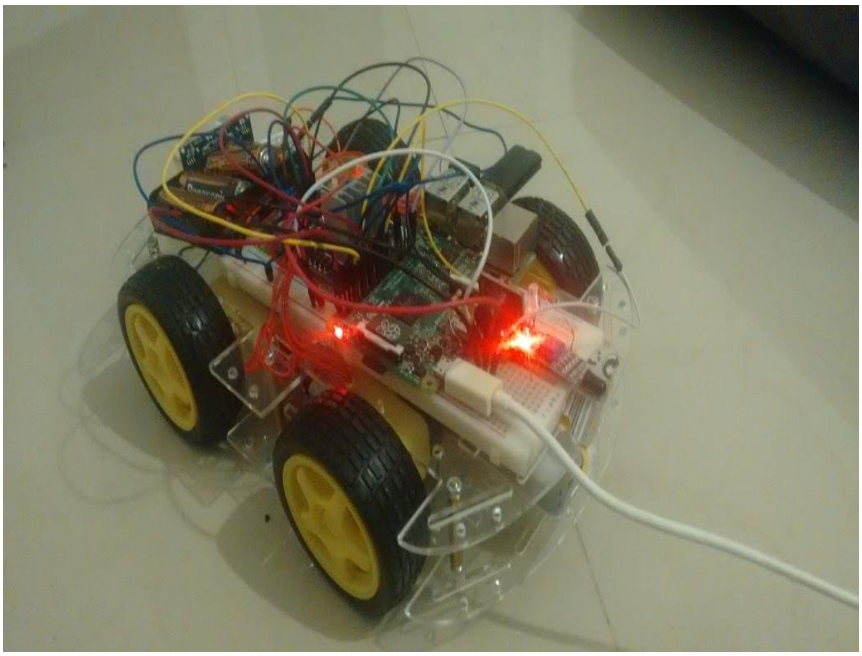

Figure 88: Final Robot Prototype

Conclusion

1. Issues during the development of the project

During the development of the project, we had an issue with the **Pulseaudio** when trying to access the USB microphone. When trying to run the speech recognition the code fails always because the **Pulseaudio** it's not capable of opening the microphone what makes impossible for us to be able to control the Robot system. We tried the same code (compiled with a library for the PC) in the PC and it works fine, but in the raspberry the code is able to run but it fails in the middle of the recognition process.
Another problem faced is when trying to use different speech recognition engine (ex: Julius) we couldn't make to code run doing the high number of dependences and some of them are not even present in the Buildroot.
Lack of information about cross compiling the speech engines, most of the speech engine used the commands **$ sudo apt-get** to install the libraries on the raspberry, we found out that there's not much information in those speech engines documentation about cross compiling the library.

2. Conclusions and future prospective

We were able to accomplish almost all of the milestones proposed the can be controlled using the accelerometer sensors and the desktop application, the car was able to stop in an emergency (obstacle detection) and alert by emitting sound.
We were able to fulfil almost all the constraint and requirements specified in the previous stages of the Development except use the speech recognition engine, but even not being able to get the speech engine working the entire process of cross compiling the libraries, trying different types of speech recognition engine gave as a bigger understanding about some concepts like, different types of libraries, advantage and disadvantages, cross compiling, etc. Also, we were able to get a bigger understanding of how the Linux operating system works, and the design of a software for an OS using multitasking.

Using the Waterfall model allowed us to have a much small margin of error during the project implementation doing to the fact that the project has been studied in the earliest stages of the waterfall model allowing is to have less error in the implementation stage.
Overall the entire process had a great impact in the group understanding about the design of an embedded system and the Linux Kernel

References:

CMU Sphinx - Project by Carnegie Mellon University
Website: http://cmusphinx.sourceforge.net/

Raspberry Pi Stack Exchange
Website*:* http://raspberrypi.stackexchange.com/questions/10384/speech-processing-on-the-raspberry-pi

Howchoo
Website*:* https://howchoo.com/g/ztbhyzfknze/how-to-install-pocketsphinx-on-a-raspberry-pi

Robot Rebels, an online robot maker community.
Website*:* http://www.robotrebels.org/index.php?topic=220.0

Speech Recognition with Pocketsphinx - ***Building application with pocketsphinx Installation***
Website: https://www.raspberrypi.org/forums/viewtopic.php?f=37&t=9487

Jamesrobertson Blog
Installing PocketSphinx on Raspbian (Jessie)
Website:http://www.jamesrobertson.eu/blog/2016/jan/18/installing-pocketsphinx.html

Julius
Open-Source Large Vocabulary CSR Engine Julius
Website: http://julius.osdn.jp/en_index.php

hertaville
Development Environment for the Raspberry Pi using a Cross Compiling Toolchain and Eclipse
Website: http://www.hertaville.com/development-environment-raspberry-pi-cross-compiler.html

Pulseaudio
pulseaudio Documentation
Website: https://www.freedesktop.org/wiki/Software/PulseAudio/