CENTROALGORITMI

Universidade do Minho
Escola de Engenharia

BALANCING ROBOT ROBBY

Authors:
Ailton Lopes A72852
José Oliveira A65308

Project Report
Project 1 MIEEICOM
2016/2017

Professor:
Adriano Tavares
José Mendes

Abstract

With the acceleration of industrial automation and the progress of computer science, the self-balancing robot technology has experienced rapid development, this kind of robots it's widely used in the control System in universities or the industry and there are a lot of commercial products that uses the same principles in in the market. This paper presents Robby Analysis, design and Implementation. A two wheeled Robot running the Real-Time Operating System FreeRTOS, controlled by an Android Application using Bluetooth Communication.

The paper covers the entire development process of the Robot and the Android application as the Issues and conclusions.

Acknowlegment

We would like to show our gratitude to the professor **Adriano Tavares**, and **José Mendes** for sharing their pearls of wisdom with us during the course of the semester and giving us all the support and help needed, and also the entire 2016/17 embedded System class for their contribution in one way or another for this project.

Key Abbreviations

IMU -	inertial measurement unit
RTOS –	Real Time Operating System
PID –	Proportional-Integral-Derivative
DSP –	Digital Signal Processor

Contents

Figure Index

Table Index

1. Introduction

The system to be implemented in this project is similar to the inverted pendulum. The inverted pendulum system is an example frequently used in control systems, the pendulum is mounted on a platform and is balanced by controlling the movement of the platform, this setup can be varied in many ways to make the system more complex and it can be controlled in many ways.

A Self-balancing robot it's a two wheels' robot that balance itself on the vertical axis using their motors movements according to the angle and position where it is falling, the robot angle and position it's calculated in real-time using the information measured for the inertial measurement unit (IMU). The inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and the magnetic field surrounding the body, using a combination of accelerometers, gyroscopes and magnetometers. Since it's only needed the measurement of the angle in one axis (x axis) in this project it's only used the Accelerometer and the Gyroscope to calculate the angle of the robot.

1.1. Aims

This main purpose of the project is to design and implement an embedded system (Self-balancing robot) with the capability of balancing itself on the vertical axis, capable of resisting outside forces and performing commands sent by the user. The commands can be to move forward, move backward, turn in one direction, among others.

To control the Robot, it's going to be developed an android application where the user can send different commands to the Robot using Bluetooth communication.

2. Theoretical Concepts

2.1. Real Time Operating System (RTOS)

RTOS is an operating system (OS) intended to serve real-time applications (RTA). Simple applications are designed to run following a sequence of steps pre-defined by the developer(s), but that approach it's not suitable when the system is subjected to a "**real-time constraint**", for example giving a response to an event. Real-time programs must guarantee response within specified time constraints, often referred to as "**deadlines**". The correctness of these types of systems depends on their temporal aspects as well as their functional aspects, in other words a real-time system not only has to be functional but also fulfil the specified deadline.

Multitasking and Concurrency

multitasking is the ability of performing one or multiple tasks over a certain period of time by executing them concurrently. Multitasking does not necessarily imply parallel execution, but it does mean that more than one task can be part-way through execution at the same time, and that more than one task is advancing over a given period of time. The Illusion of simultaneous execution of task is created because of the capability of the tasks to interrupt the execution of the other already started before the complete execution of the running task (Context switch).

Basic Multitasking Models:

- Round-Robin
- Round-Robin with time-slice
- Preemptive

For an embedded system, the best model is the Preemptive where the most important task will run until the end or interrupted by another one with higher priority.

The operating system used in this project is the **FreeRTOS.**

FreeRTOS

FreeRTOS is a popular real-time operating system kernel for embedded devices that is designed to be small enough to run on a microcontroller. FreeRTOS provides methods for multiple threads or tasks, mutexes, semaphores, message queue and software timers.

Figure 1 – FreeRTOS logo

FreeRTOS Key features:

- Free pdf book and reference manuals.
- Very small memory footprint, low overhead, and very fast execution.
- Tick-less option for low power applications.
- Equally good for hobbyists who are new to OSes, and professional developers working on commercial products.
- Scheduler can be configured for both preemptive or cooperative operation.

2.2. 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.

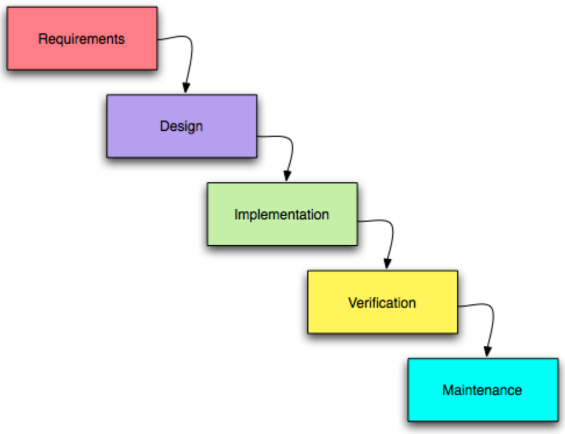


Figure 2 – Waterfall model phases

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.

2.3. PID Controllers

proportional-Integral-Derivative (PID) is the o of the most widely used control algorithm in the industry and has been used worldwide for industrial control systems. The popularity of PID controllers can be attributed doing to their robust performance over a wide range of operating conditions and in part to their functional simplicity, which allows engineers to operate them in a simple and straightforward manner.

The controller name is doing to the coefficients that composes it, Proportional, Integral and Derivative, the desired output can be obtain by tuning the controller Coefficients

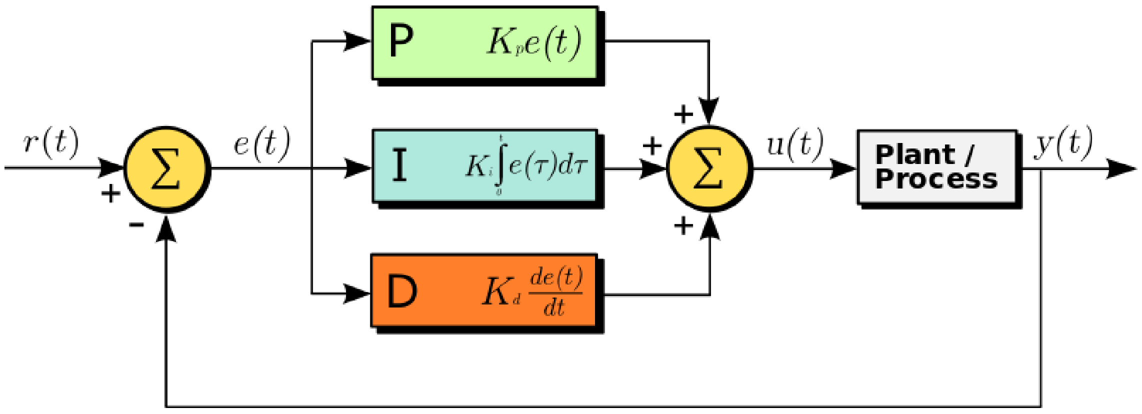


Figure 3: PID Block Diagram

A PID controller continuously calculates an error value **e(t)** as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms. The controller attempts to minimize the error over time by adjustment of the Output variable **u(t).** The output of the controller **u(t)** can be calculated using the following expression:

$$\boldsymbol{u(t) = Kp * \ e(t) + \ Ki \int_0^t e(\tau) * d\tau + Kd * \frac{de(t)}{dt}}$$

$$\boldsymbol{e(t) = Setpoint - y}$$

Where **y** is the measurement of the process variable, **Ki** is the Integral gain, **Kp** proportional gain and **Kd** derivative gain.

Proportional Response

The proportional component depends only on the difference between the setpoint and the process variable (error e(t)). The proportional gain (Kp) determines the output response rate for the error signal. For example, if the error term has a magnitude of 10, a proportional gain of 5 would produce a proportional response of 50. In general, increasing the proportional gain will increase the response speed of the control system. However, if the proportional gain is too large, the process variable will begin to oscillate.

If Kp is increased further, the oscillations will become larger and the system will become unstable and may oscillate even out of control.

Integral response

The integral component sums the error term over time. The result is that even a small error will cause the integral component to increase slowly. The integral response will increase over time unless the error is zero, so the effect is to drive the steady-state error to zero. A phenomenon called integral windup occurs when the integral action saturates the controller.

Response Derivative

The derived component causes the output to decrease if the process variable is increasing rapidly. The response derivative is proportional to the rate of change of the process variable. Increasing the derivative gain (Kd) will cause the control system to react more strongly to changes in the error parameter by increasing the speed of the overall system control response. In practice, most control systems use very small derivative time (Kd) because the response derivative is very sensitive to noise in the process variable signal. If the sensor feedback signal is noisy or if the control loop rate is too slow, the response derivative may make the control system unstable.

PID Tuning

The process of changing the values of the PID parameters (Kp, Ki and Kd) to get the best controller response it's called **PID Tuning.** There several methods to tune the PID controller and the most effective methods require the development an algorithm.

PID Tuning methods:

Methods	Advantage	Disadvantages
Manual	No math required	Require experience
Ziegler-Nicholas	Proven Method	Process upset required
Cohen-Coon	Good process models	Good only for first order processes
Software tools	Consistent tuning, support non-steady state tuning	Cost and training required
Algorithmic	Consistent tuning, support non-steady state tuning, very precise	Very slow

Table 1: PID Tuning methods

2.4. Kalman Filter

The Kalman filter is the best possible (optimal) estimator for a large class of problems and a very effective and useful estimator for an even larger class, it uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more precise than those based on a single measurement alone,

The Kalman filter has numerous applications in technology. A common application is for guidance, navigation, and control of vehicles, particularly aircraft and spacecraft.

Furthermore, the Kalman filter is a widely-applied concept in time series analysis used in fields such as signal processing. Kalman filters also are one of the main topics in the field of robotic motion planning and control, and they are sometimes included in trajectory optimization.

Filter Algorithm

The algorithm consists in two Steps: Prediction and Update/correction.

In the **prediction** step, the Kalman filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement is observed. In the **Update** step these estimates are updated using a weighted average ([1]), with more weight being given to estimates with higher certainty. The algorithm is recursive. It can run in real time, using only the present input measurements and the previously calculated state and its uncertainty matrix; no additional past information is required.

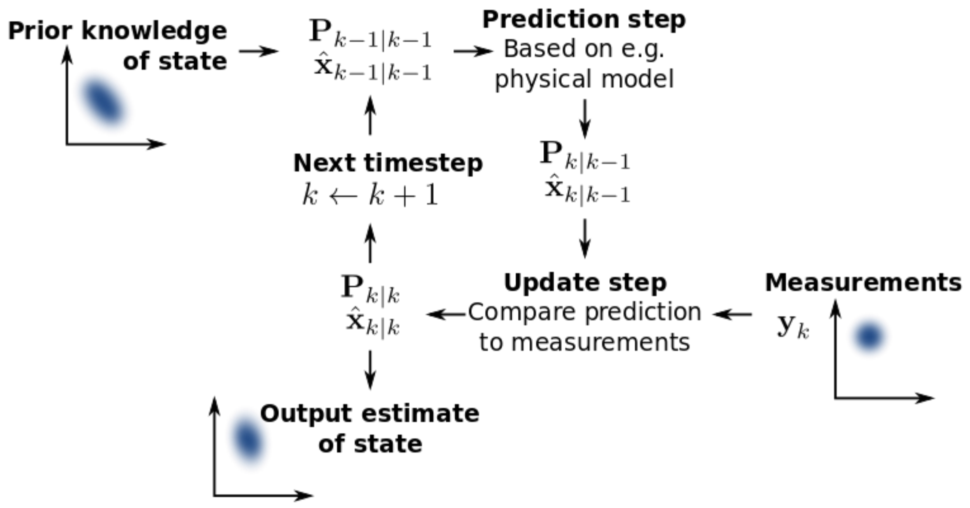

Prior knowledge of state
$\mathbf{P}_{k-1|k-1}$ $\hat{\mathbf{x}}_{k-1|k-1}$
Prediction step
Based on e.g. physical model
$\mathbf{P}_{k|k-1}$ $\hat{\mathbf{x}}_{k|k-1}$
Next timestep
$k \leftarrow k+1$
Update step
Compare prediction to measurements
Measurements
$\mathbf{y}_k$
$\mathbf{P}_{k|k}$ $\hat{\mathbf{x}}_{k|k}$
Output estimate of state

Figure 4: Kalman Filter algorithm overview

([1]) - The weighted average is where it's decided how much trust to give to the sensors measurement and the estimator, if the sensor is precise the sensor is given more weight to the sensor measurement if not it's given more weight to the estimator.

3. Requirements

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.

3.1. Functional requirements

- Perform the movement command (move forward, move backward, turn right and turn left) sent by the user from the android application.
- Support automatic tuning of the PID controllers.
- Enable the user to choose the PID tune Mode (manual or Automatic) in real time using the android application.
- Enable the user to see the current value of the PID gains.
- Use the wheel's movement to balance the robot in the vertical.
- Acquire data from the inertial sensors (Accelerometer and Gyroscope) to measure the current angle of the Robot.
- Use the encoders sensors mounted on the motors to measure the distance traveled by each motor to compensate de motors differences.

3.2. Non-functional requirement

- Low cost, low power consumption and high performance.
- Friendly user interface

4. 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

4.1. Technical Constraint:

- Use the development Board STM32F4-DISCOVERY.
- Use the development environment Keil MKD-ARM.
- Use the Board to make to tuning of the PID Controller.
- Implement the auto-tuning of the controller gains.
- Use at least three Sensors.
- Use the real-time operating System FreeRTOS.
- Use the Digital Signal Processor (DSP)

4.2. Non-technical Constraint

- Two elements for the group.
- Fulfil the specified deadlines
- Meet the milestones specified by the group (in the Gantt Diagram)
- Minimum of 6 hours of work a week.
- Use and gain experience in new software used in the software and hardware development.
- Lowes budget possible

5. Resources

5.1. Hardware Resources

One of the constraint of the project is use the development board STM32F4 – Discovery and use the DSP. Another constraint is to use at least three sensors in the project and the following list will specify all the hardware resources selected by the developers:

- STM32F4-Discovery,
    - I. Microcontroller STM32F407VGT6
    - II. DSP.
    - III. Push Button.
- IMU MPU6050 (3-axis accelerometer; 3-axis gyroscope)
- Batteries
- Bluetooth module HC – 06,
- DC Motors with encoder.
- Motor Driver L298N.
- Android Phone

5.2. Software Resources

Similar to the Hardware Resources there are also some software resources that was imposed in the project constraint.

One of them is use the FreeRTOS operating system that is a popular real-time operating system kernel for embedded devices, FreeRTOS is designed to be small and simple. The kernel itself consists of only three C files. To make the code readable, easy to port, and maintainable, it is written mostly in C, but there are a few assembly functions included where needed (mostly in architecture-specific scheduler routines).

The other one is use the Keil uVision IDE that combines project management, run-time environment, build facilities, source code editing, and program debugging in a single powerful environment. µVision is easy-to-use and accelerates the development of an embedded software. µVision supports multiple screens and allows to create individual window layouts anywhere on the visual surface. The IDE also offers debugging tool that provides a single environment to test, verify, and optimize the application code. The debugger includes traditional features like simple and complex breakpoints, watch windows, and execution control and provides full visibility to device peripherals.

It's used others software that are not a constraint of the project but will be required in the development of the project, this software are:

- MATLAB,
- Android Studio.
- Star UML
- STM Studio
- Altium

6. System Overview

The system its divided in two main segments, the Robot system and the Android application.

The robot system is powered by the STM32F4 - Discovery board with the STM32F407VGT6 microcontroller featuring 32-bit ARM Cortex® - M4. Connected to the board is an Inertial measurement sensor(IMU) **MPU6050** that is an integrated 6-axis Motion Tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, and a Digital Motion Processor™ (DMP), the MPU6050 communicate with the board using the I2C communication protocol. The Inertial measurement sensors is used to measure the robot current angle in real-time that is used as input to the PID controller used to balance the robot. In order to control the Robot´s position according to the output of the PID controllers is used the dual H-Bridge motor controller **L298N,** the L298N is required doing to the fact that the motors work with a much higher voltage (12v) than the maximum given by the Development board, so to solve this problem is used four Li-Ion MR18650 – 3,7V 3000mAh to power the motors and by using the L298N we are able to control the motors using the Development board. In order to isolate the command circuit to the power circuit the board power supply is a Power Bank

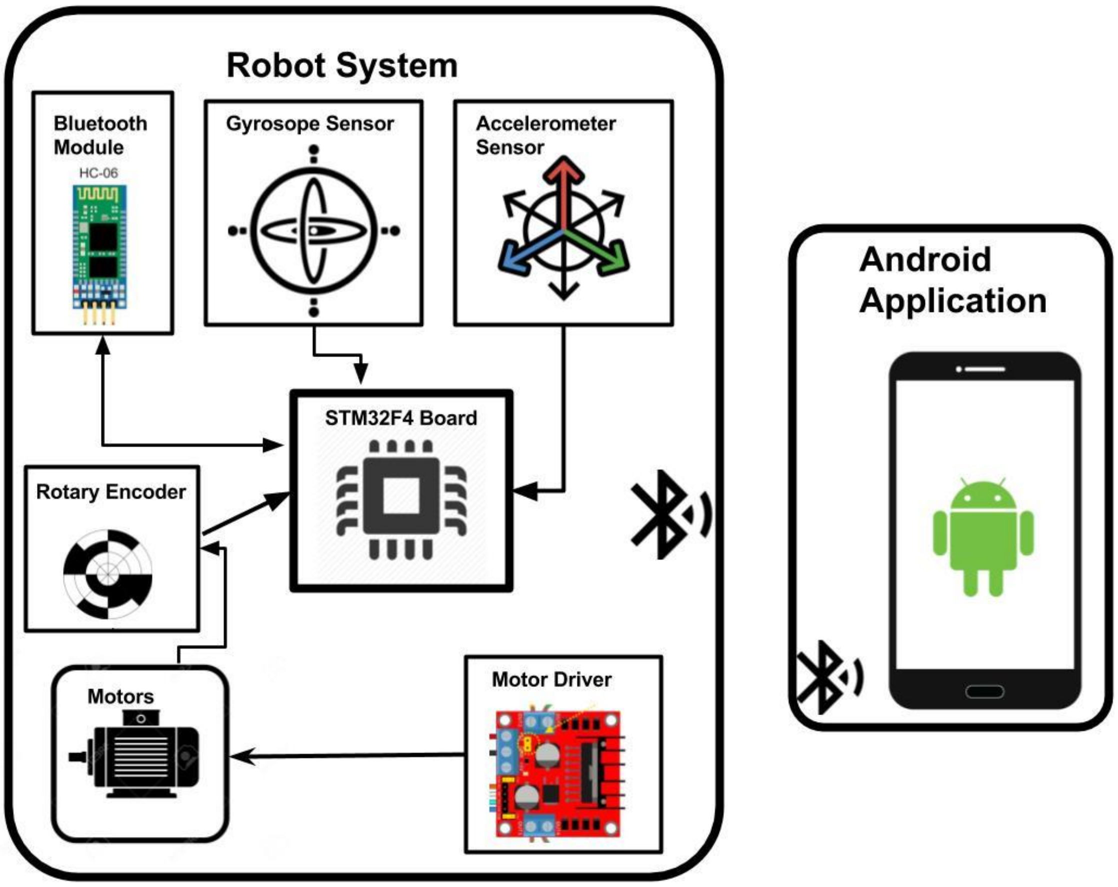


Figure 5: System Overview

To allow the communication between the Robot System and the Android application is used the Serial Bluetooth Module HC-06 that communicate with the board through USART.

The Android application provides the user several functionalities to control the Robot, the user can send a movement command (forward, backward, right or left), the user can manually tune the PID controllers or choose the Auto-Tuning mode and the user can also stop or start the robot

7. System architecture

The System Stack architecture of the system is divided in two parts: The Robot System and the Android Application

The android application is composed by three segments:

- The user interface that provides de output of the data received from the Robot system and gives the user the input tools tool necessary to use all the functionalities of the application.
- The Command Handler that reads the input chosen by the user and provides the output associated to that command

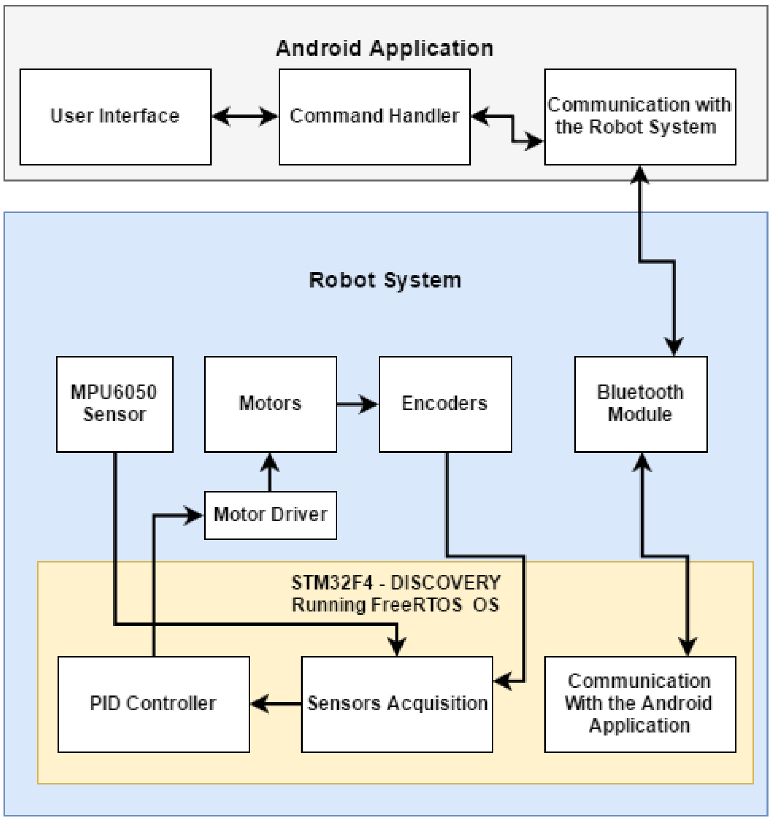


Figure 6: System Stack Architecture

- The Communication section that is responsible to exchange data with the Robot system using Bluetooth communication.

The Second part of the System Stack Architecture is the Robot system that is composed by many sections:

The section The STM32F4 – Discovery Running the FreeRTOS OS has three subsections that communicates and exchange data with the sensors and actuators in the robot system. The Communication with android Application subsection is responsible to exchange data with the Android application using the Bluetooth module, the sensor acquisition subsection is responsible to acquire the data in real time from the sensors and sent it to the PID controller section that will produce an output to the motor driver according to the input data, in order to keep the robot Standing (balancing).

8. 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.

8.1. Use Cases Diagram

The use case Diagram represents the user's (actors) interaction with the system.

Due to their simplistic nature, use case diagrams can be a good communication tool for presentations. The drawings attempt to mimic the real world and provide a view for the stakeholder to understand how the system is going to be designed.

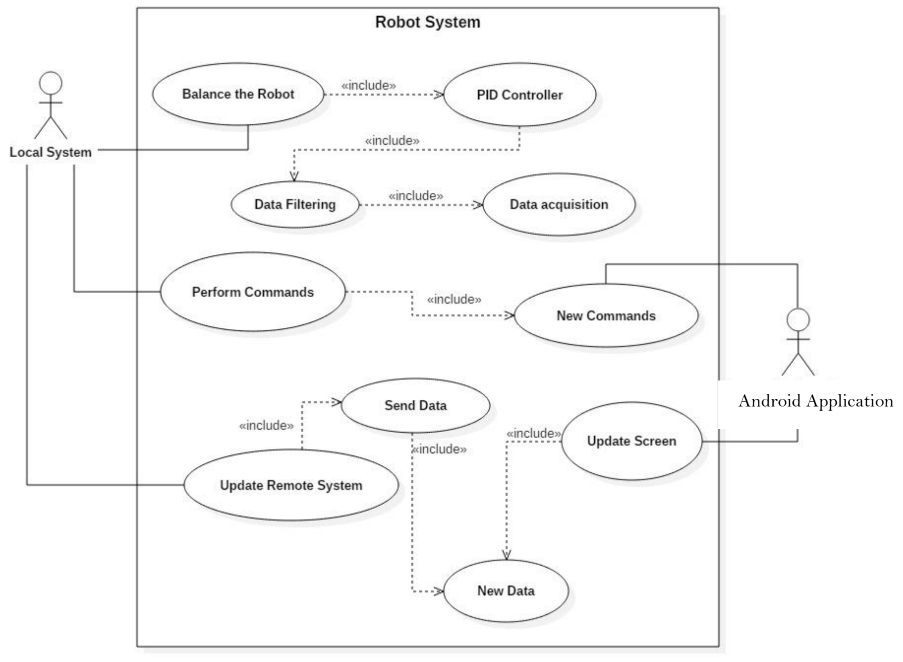


Figure 7 - Robot System Use Cases Diagram

The Robot System has two actors, the local System and the Android Application.

The Android application is an actor in the Robot system because it has the ability of perform action like send a command for the robot system to perform and it can also receive data from the Robot system to Update the user Interface (GUI).

The other actor is the Robot System that can perform three different actions: Balance the robot, perform a command or send new data to the Android Application. Balance the Robot is the main action of the Local System and is done by first reading the sensor, filtering the data and used the PID controllers to Calculate the Robot wheel's output. To perform a command the Robot System receive a command from the Android application and according to the command received the local System actuates in the Robot wheel's. The Local System can also update the Android application of the current State of the Robot System.

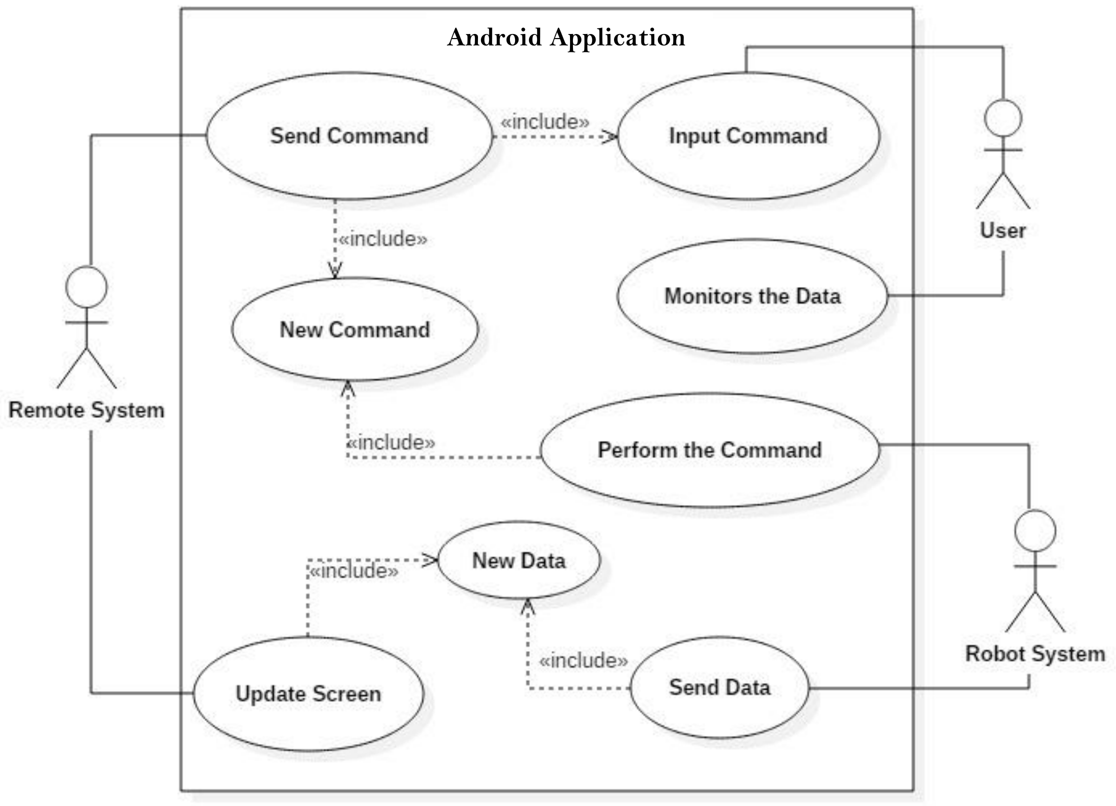


Figure 8 – Remote system event cases

In the Android Application use cases diagram we have three actors:

The Robot System that can perform actions like send a new data to the Remote system to Update the user interface or performs a new command according to the input received form the Remote System (Android Application.)

The user is also an actor in the Android Application and it can send a new command to the Robot system through the Android Application and also it can monitor the data from the Robot System over the GUI.

The third actor is the Local system that can Perform command as Update the screen with the new data received from the Robot system or Send a new command to the Robot system

8.2. Event Table

The following table have different events that can occur in the Robot System, the type of event, trigger, system response and the source.

8.2.1. Robot System Event Table

Events	Trigger	System response	Source	Type
Start or Stop	Turn on or off command received via bluetooth	Balance or stop the Robot	Android Application	Asynchronous
Movement Command	Movement Command Received via bluetooth	Move According to the direction received	Android Application	Asynchronous
PID gains manual Tuning	New PID Gains received via bluetooth	Update the variables that holds the PID Gains	Android Application	Asynchronous
Change PID Tuning mode	Character ‘a’ received via bluetooth	Auto-tuning Flag = 1	Android Application	Asynchronous
Bluetooth connection request	Character ‘p’ received via bluetooth	Send the value of the PID Gains to update the GUI	Android Application	Asynchronous

Table 2 – Robot System event table

8.2.2. Android Application Event Table

Events	Trigger	System response	Source	Type
Connect to Robot Requested	Bluetooth button pressed	Send Start or stop command to Robot System	Android Application	Asynchronous
Movement Command	Movements arrows buttons Pressed	Send movement command to Robot System	Android Application	Asynchronous
PID Manual Tuning	Send button pressed	Send the gains to the Robot System	Android Application	Asynchronous
Change PID Tuning mode	Auto-Tuning Button pressed	Send character ‘a’ via Bluetooth to Robot system	Android Application	Asynchronous

Table 3 – Android Application event table

8.3. State Diagram

State diagrams are used to give an abstract description of the behavior of a system. This behavior is analyzed and represented as a series of events that can occur in one or more possible states.

State diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. So, the most important purpose of state diagram is to model life time of an object from creation to termination.

8.3.1. Robot System State Diagram.

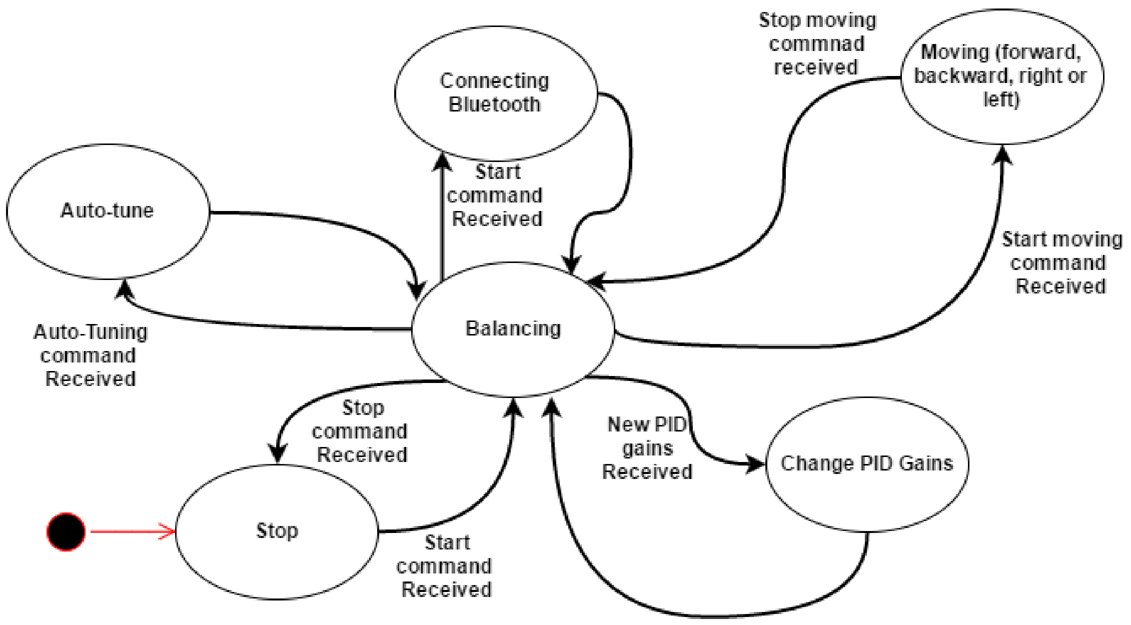


Figure 9 – Robot system state diagram

As shown in the Figure 13 the Robot system is when started is in the Balancing state and it can change to Auto-Tuning state if it receives Auto-tuning command from the Android application, the same happens when It changes to the Change PID Gains or connecting Bluetooth. Those states described before are states that happens for a short period of time, for example the Change PID gains it only changes the value of the variables that holds the value of the PID gains and it get back to the Balancing state, they don't need any event to happens in order for them to finish. The moving state is different form the others because when the Robot system receives a command to start movement it only stops when it receives a new command to stop. The Robot can go to the stop state when the system begins or if it receives a stop command from the android application.

8.3.2. Android Application State Diagram

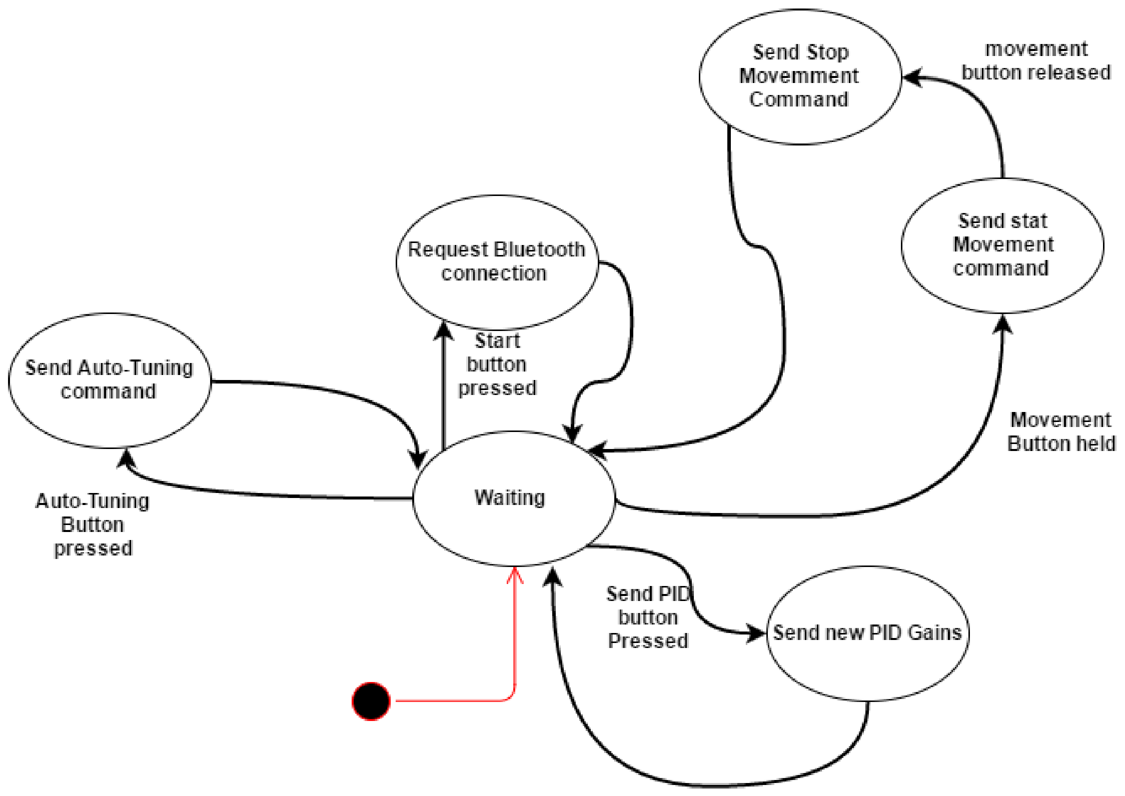


Figure 10 – Android Application State Diagram

The figure 14 shows the State Diagram of the Android Application, similar to in the Robot System in the Android Application there are some states that only have a start condition and after it finishes the execution it gets back to the starting point (waiting) but there are othere like the Send Start Movement Command that when the used press and holds a movement button it sends the start movement commanto to the Robot System and only when the user release the button it sends the stop movement command and get back to the waiting state.

As said before only by looking at the figure representing the differents states and the events that can trigger a State switching it's possible to understant the System flow of control.

8.4. Sequence Diagram

A Sequence diagram is an interaction diagram that shows how objects operate with one another and in what order.

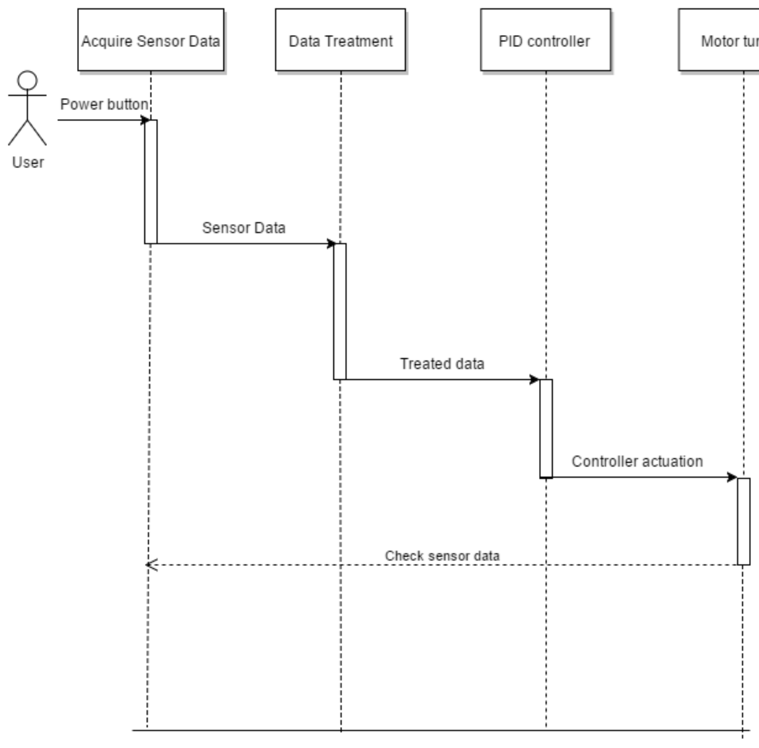


Figure 11 – Robot System Sequence Diagram

In the figure 15 it's presented the Sequence diagram of the Robot System showing the system execution flow.

In the Robot System first its acquire the data from the sensors (MPU6050) , after reading the sensor it's calculated the Robot current angle using the Kalman or the complimentary filter, after calculating the Robot's angle the angle is used as input in the Balance PID controller to calculate the PWM for the motors and the last part of the sequence is use the PID output to update the motors inputs.

9. Hardware Specification

9.1. Hardware Resources

9.1.1. Development Board STM32F4-Discovery

The STM32F4DISCOVERY Discovery kit allows users to easily develop applications with the STM32F407 high performance microcontroller with ARM® Cortex®-M4 32-bit core. It includes everything required either for beginners or for experienced users to get quickly started.

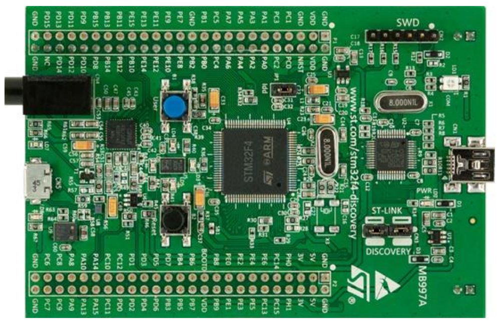


Figure 12 - STM32F4-Discovery

Features:

- STM32F407VGT6 microcontroller featuring 32-bit ARM Cortex® -M4 with FPU core,
- 1-Mbyte Flash memory, 192-Kbyte RAM in an LQFP100 package
- On-board ST-LINK/V2 on STM32F4DISCOVERY or ST-LINK/V2-A on
- STM32F407G-DISC1
- ARM® mbed™ -enabled (http://mbed.org) with ST-LINK/V2-A only
- USB ST-LINK with re-enumeration capability and three different interfaces:

1. virtual com port (with ST-LINK/V2-A only)
2. mass storage (with ST-LINK/V2-A only)
3. debug port

a. Board power supply:

4. Through USB bus
5. External power sources (3 V and 5 V)

- LIS302DL or LIS3DSH ST MEMS 3-axis accelerometer
- MP45DT02 ST MEMS audio sensor omni-directional digital microphone
- CS43L22 audio DAC with integrated class D speaker driver
- LEDs
- Two push buttons (user and reset)
- USB OTG FS with micro-AB connector
- Extension header for all LQFP100 I/Os for quick connection to prototyping board and easy probing.

STM32F4 Board Schematics

In the schematics in in figure 5 is only present the pins of the board that are going the be used in the project

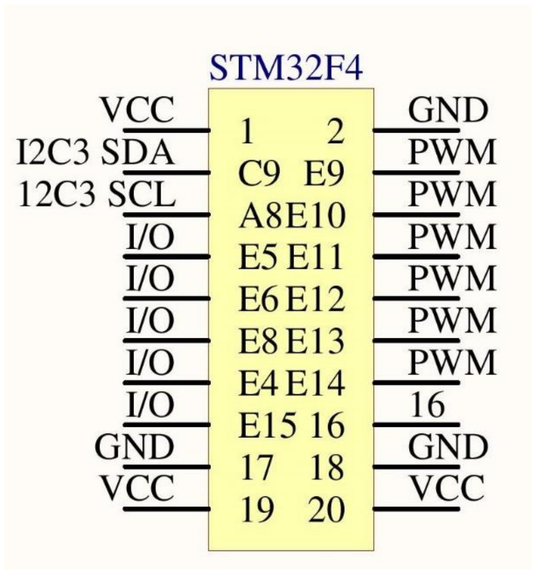


Figure 13 – STM32F4 Board Schematics

Pin description

Pin	Name	Description
1	PC9 I2C3 SDA	Input from the IMU sensor (SDA)
2	PA8 I2C3 SCL	Input from the IMU sensor (SDA)
3	PA3 USART2 RX	Input from the Bluetooth Module
4	PA2 USART2 TX	Output from the Bluetooth Module
5	PE5	Input2 from the encoder B
6	PE6	Input1 from the encoder B
7	PE8	Input2 from the encoder A
8	PE4	Input1 from the encoder A
9	GND	Ground voltage
10	VCC	5 V
11	PE9 (PWM)	PIN to control the Motor
12	PE10(PWM)	PIN to control the Motor
13	PE11(PWM)	PIN to control the Motor
14	PE12(PWM)	PIN to control the Motor
15	PE13(PWM)	PIN to control the Motor
16	PE14(PWM)	PIN to control the Motor

Table 4 - STM32F4 Board Pin Description

9.1.2. Bluetooth Module HC-06

HC-06 module is an easy to use Bluetooth SPP (Serial Port Protocol) module, designed for transparent wireless serial connection setup. Serial port Bluetooth module is fully qualified Bluetooth V2.0+EDR (Enhanced Data Rate) 3Mbps Modulation with complete 2.4GHz radio transceiver and baseband. It uses CSR Bluecore 04-External single chip Bluetooth system with CMOS technology and with AFH (Adaptive Frequency Hopping Feature).

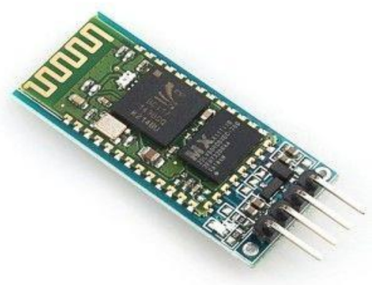


Figure 14 – Bluetooth module HC-06

Hardware features:

- Typical -80dBm sensitivity.
- Up to +4dBm RF transmit power.
- Low Power 1.8V Operation ,1.8 to 3.6V I/O.
- PIO control.
- UART interface with programmable baudrate.

Software features:

- Default Baud rate: 38400, Data bits:8, Stop bit:1Parity: No parity, Data control: has supported baud rate: 9600, 19200, 38400, 57600, 115200, 230400, 460800.
- Given a rising pulse in PIO0, device will be disconnected.
- Status instruction port PIO1: low-disconnected, high-connected.
- PIO10 and PIO11 can be connected to red and blue led separately. When master and slave are paired, red and blue led blinks 1time/2s in interval, while disconnected only blue led blinks 2times/s.
- Auto-connect to the last device on power as default.
- Permit pairing device to connect as default.
- Auto-pairing PINCODE:"1234" as default
- Auto-reconnect in 30 min when disconnected as a result of beyond the range of connection.

Purpose

The Bluetooth Module HC-06 is used in the Robot system to communicate with the Android Application (send data or receive command).

Bluetooth module Pin assignment

PIN	Connected to:	Description
TX	STM32F4 – PA3 (USART2 RX)	Serial communication
RX	STM32F4 – PA2 (USART2 TX)	Serial communication
VCC	STM32F4 – VCC	Power Supply to the module

Table 5 – Bluetooth module Pin assignment

Bluetooth Module Test Case

The following table shows the input and the expected output for the Bluetooth module

Input	Expected Output	Output
"Hello World"	"Hello World"	

Table 6 – Bluetooth Test Case

9.1.3. Sensor MPU6050

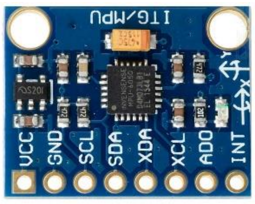


Figure 15 –MPU6050

Gyroscope Features

- Digital-output X-, Y-, and Z-Axis angular rate sensors (gyroscopes) with a user-programmable full-scale range of ±250, ±500, ±1000, and ±2000°/sec and integrated 16-bit ADCs
- Digitally-programmable low-pass filter
- Gyroscope operating current: 3.2mA
- Sleep mode current: 8µA

- Factory calibrated sensitivity scale factor
- Self-test

Accelerometer Features

- Digital-output triple-axis accelerometer with a programmable full-scale range of ±2g, ±4g, ±8g and ±16g and integrated 16-bit ADCs
- Accelerometer normal operating current: 450µA
- Low power accelerometer mode current: 8.4µA at 0.98Hz, 19.8µA at 31.25Hz
- Sleep mode current: 8µA
- User-programmable interrupts
- Wake-on-motion interrupt for low power operation of applications processor

Pin Description

Pin	Name	Description
1	VDD	Power supply (3.3V or 5V)
2	GND	Ground voltage
3	SCL	I2C serial Clock
4	SDA	I2C Serial Data
8	INT	Interrupt Digital Output

Table 7 – IMU sensor pin description

IMU Sensor Schematics

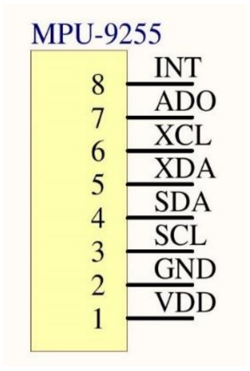


Figure 16 – IMU sensor schematics

I2C Serial Interface

In this project is used the I2C protocol to interface the Development board with the accelerometer sensor.

I2C is a two-wire interface comprised of the signals serial data (SDA) and serial clock (SCL). In general, the lines are open-drain and bi-directional. In a generalized I2C interface implementation, attached devices can be a master or a slave. The master device puts the slave address on the bus, and the slave device with the matching address acknowledges the master. The MPU-60X0 always operates as a slave device when communicating to the system processor, which thus acts as the master. SDA and SCL lines typically need pull-up resistors to VDD. The maximum bus speed is 400 kHz. The slave address of the MPU-60X0 is b110100X which is 7 bits long. The LSB bit of the 7-bit address is determined by the logic level on pin AD0.

I2C Communications Protocol

START (S) and STOP (P) Conditions

Communication on the I2C bus starts when the master puts the START condition (S) on the bus, which is defined as a HIGH-to-LOW transition of the SDA line while SCL line is HIGH (see figure below). The bus is considered to be busy until the master puts a STOP condition (P) on the bus, which is defined as a LOW to HIGH transition on the SDA line while SCL is HIGH (see figure below).

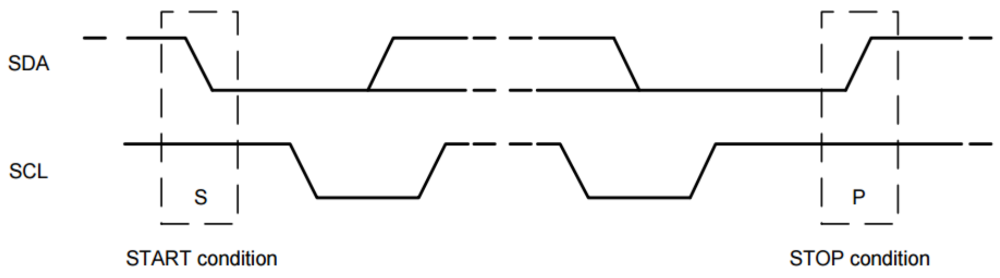


Figure 17 – I2C start and stop condition

After beginning communications with the START condition (S), the master sends a 7-bit slave address followed by an 8th bit, the read/write bit. The read/write bit indicates whether the master is receiving data from or is writing to the slave device. Then, the master releases the SDA line and waits for the acknowledge signal (ACK) from the slave device.

Purpose

The MPU-9255 is used also as input to controller that balance the robot (detect in real time the position of the Robot using the accelerometer and gyroscope sensors).

IMU Sensor Pin assignment

PIN	Connected to:	Description/Type
VDD	STM32F4 – VCC	Power Supply (5V)
GND	STM32F4 – GND	GND
SCL	STM32F4 – PC9	I2C3 SDA (Serial Data)
SDA	STM32F4 – PA8	I2C3 SCL (Serial Clock)

Table 8 – IMU sensor pin assignment

IMU Sensor Test Case

The following table shows the input and the expected output for IMU Sensors

Input	Expected Output	Output
Shake the sensor	LED on	

Table 9 – IMU sensor test case

9.1.4. L298N Motor Driver

The L298N 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 together 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 18 – Motor driver

Block Diagram:

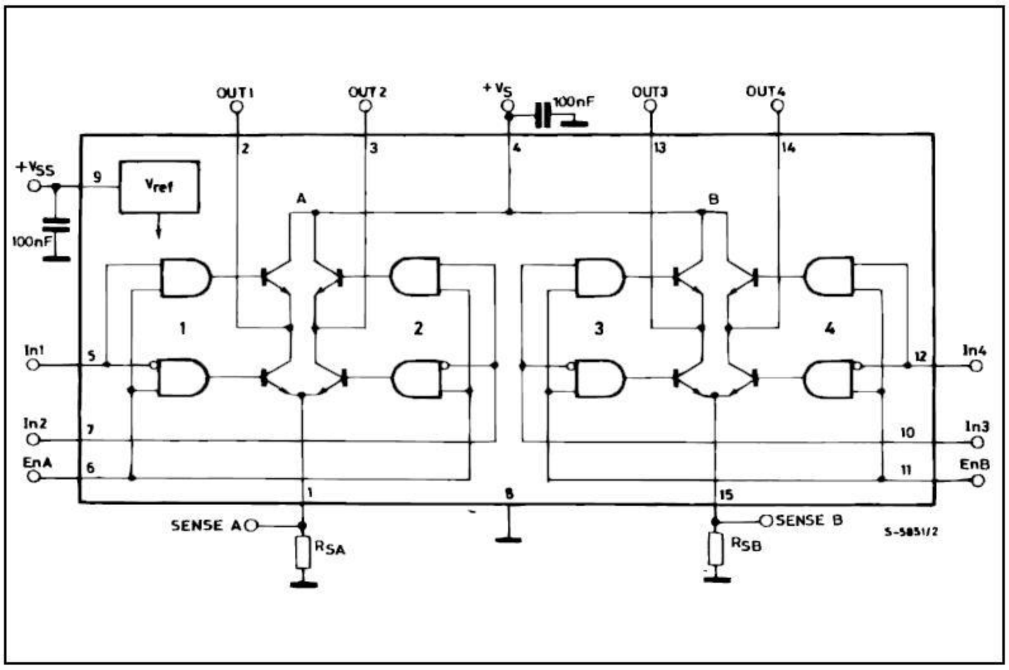


Figure 19 - L298N Block Diagram

Pins Description

Pin	Description
VCC	5V
VS	Input Voltage (12V)
In1	Input to control Motor A
In2	Input to control Motor A
In3	Input to control Motor B
In4	Input to control Motor B
enA	Input to enable Motor A
enB	Input to enable Motor B
Out1	Motor A output +
Out2	Motor A output -
Out3	Motor B output +
Out4	Motor B output -
GND	Ground voltage

Table 10 – Motor driver pin description

Motor Driver L298N Schematics

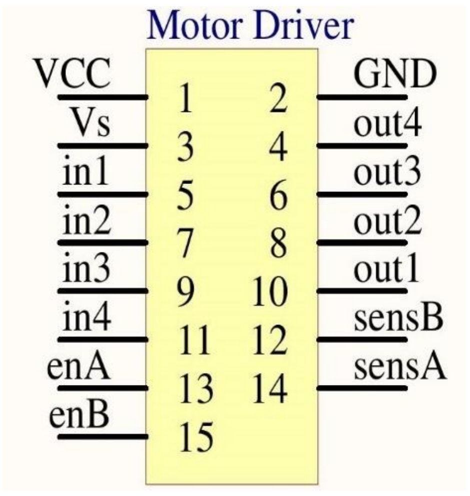


Figure 20 – Motor with encoder schematics

Purpose:

The L298N is used to interface(control) the motors using the development board.

Motor Driver Pin assignment

Motor Driver L298N	Connected to:	Description/Type
VS	Battery + (12V)	-
In1	STM32F4 - PE14	PWM
In2	STM32F4 - PE13	PWM
In3	STM32F4 - PE12	PWM
In4	STM32F4 - PE11	PWM
enA	STM32F4 - PE10	Input
enB	STM32F4 – PE9	Input
GND	Battery –	-
Out1	Motor A M1	PWM
Out2	Motor A M2	PWM
Out3	Motor B M1	PWM
Out4	Motor B M2	PWM

Table 11 – Motor driver pin assignment

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 12 – Motor driver test case

9.1.5. JGA25-371 DC Gearmotor with Encoder

Figure 21 – JGA25-371 DC gearmotor with encoder

Overview

This JGA25-371 DC gearmotor features an integrated encoder which provides a resolution of 12 counts per revolution, ensuring an accurate control of motor's speed.

Specifications

- Operating voltage: between 6 V and 24 V
- Nominal voltage: 12 V
- Free-run speed at 12 V: 126 RPM
- Free-run current at 12 V: 46 mA
- Stall current at 12 V: 1 A
- Stall torque at 12 V: 4.2 kg·cm
- Gear ratio: 1:34
- Reductor size: 21 mm
- Weight: 85 g

Pin description

Pin	Label	Description
1	OUTA	Hall sensor output A
2	OUTB	Hall sensor output B
3	M1	Motor +
4	M2	Motor -
5	VCC	Hall sensor VCC
6	GND	Hall sensor GND

Table 13 – Motor with encoder pin description

Motor with Encoders schematics

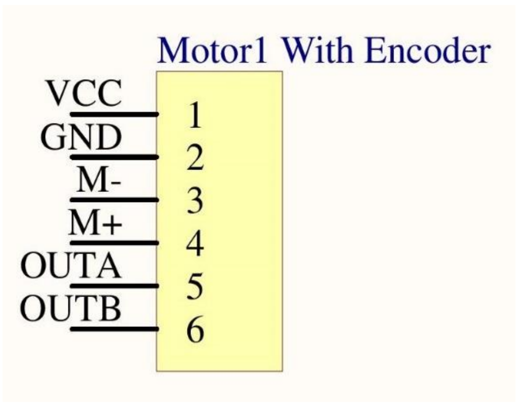


Figure 22 – Motor with encoder schematics

Purpose:

The two motors are used to control the robot movements, and the encoder sensor is used as input to the controller system by measuring the velocity of the robot.

Motor 1 with encoders Pin assignment

PIN	Connected to:	Description/Type
OUTA	STM32F4 – PE4	Output
OUTB	STM32F4 – PE8	Output
M1	Motor Driver – Out1	PWM
M2	Motor Driver – Out1	PWM
VCC	STM32F4 – VCC	Power Supply (5V)
GND	STM32F4 – GND	GND

Table 14 – Motor 1 pin assignment

Motor 2 with encoders Pin assignment

PIN	Connected to:	Description/Type
OUTA	STM32F4 – PE6	Output
OUTB	STM32F4 – PE5	Output
M1	Motor Driver – Out1	PWM
M2	Motor Driver – Out1	PWM
VCC	STM32F4 – VCC	Power Supply (5V)
GND	STM32F4 – GND	GND

Table 15 – Motor 2 pin assignment

9.1.6. Battery Li-Ion MR18650 – 3,7V 3000mAh

Specification:

- **Manufacturer:** GP
- **Rechargeable battery type**: Li-Ion
- **Battery size**: MR18650
- **Rated voltage**: 3.78V
- **Capacity:** 3000mAh
- **Body dimensions**: Ø18.3 x 65.4mm
- **Maximum current**: 6A

Figure 23 – li-Ion Baterry

9.1.7. Power Bank

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

Figure 24 – Power Bank

9.2. System Hardware Schematics

The figure below shows how the hardware components are connected with each other.

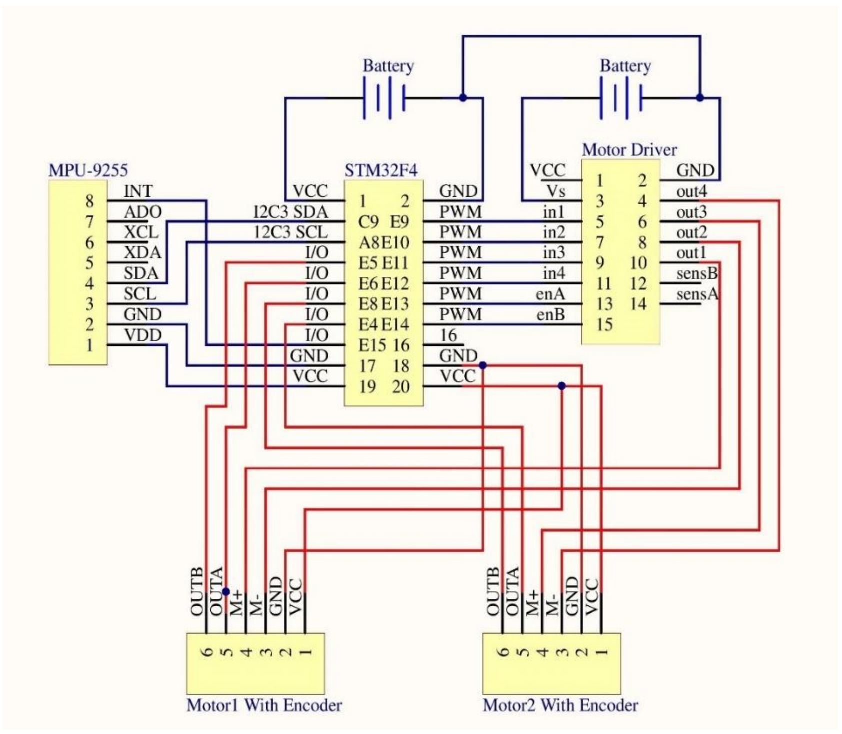


Figure 25 – System schematics

9.3. Peripherals and Communication Protocols

The Figure presents the peripherals used in this project and the communication protocol used.

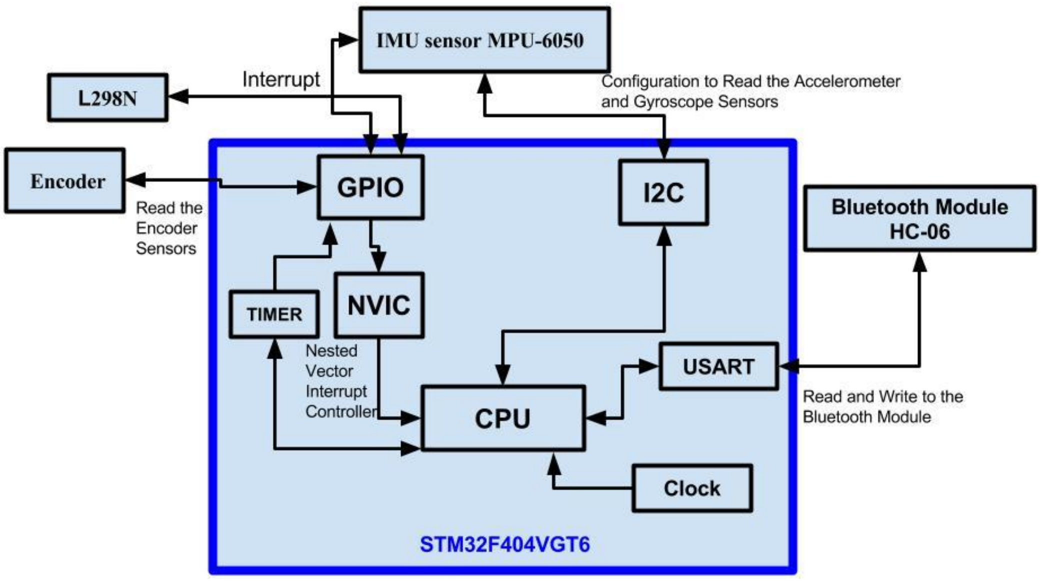

IMU sensor MPU-6050
L298N
Interrupt
Configuration to Read the Accelerometer and Gyroscope Sensors
Encoder
Read the Encoder Sensors
GPIO
I2C
Bluetooth Module HC-06
TIMER
NVIC
Nested Vector Interrupt Controller
USART
CPU
Read and Write to the Bluetooth Module
Clock
STM32F404VGT6

Figure 26 – Peripherals and Communication Protocols overview

9.4. Software specification

9.4.1. Tools:

- Altium
- MATLAB
- StarUML

9.4.2. Data Format

The Robot System and the Android application are constantly exchanging data using the Bluetooth communication. The following Table shows the structure of the Data format used.

Type	Command	Source	Response	Description
PID Write	0x-val1-val2-val3	Android App	0xS	(1)
PID Read	1x	Android App	1x-val1-val2-val3	(2)
Status	2G	Android App	2-val-x	(3)
Movement	3x	Android App	3S	(4)

Table 17: Data Format

Description:

(1)

The first command is used when the used wants to write the value of the Gains (Kp, Kd and Ki) of one of the PID controller in the robot system.

The command sent by the Android application has the Format showed in the previous Table (0-x-val1-val2-val3).

The fist character [**0**] is to informs the Robot system that the command is the type PID Write, the second character [**x**] tells the Robot System which PID controller the User wants to change the gains (x = 1 – Balance Controller, x=2 Steering Controller and x = 3 – Distance Controller), and the values of the gains are the **val1** – Kp new value, **val2** – Kd new value and **val3** - Ki new value.

The Response from the Robot System to the PID Write command first character [**0**] informs the Android system that the commands is a response to the PID Write

Command, the second one indicates witch PID Controller was Updated, and last character **[S]** tells that the gains were successful updated to the new values.

Example: Android Application send **11-0.4-10-14**, this commands tells the Robot System Update the Gains of the **Balance Controller** to **Kp = 0.4, Kd = 10 and Ki = 14**, after performing the command the Robot System will answer with **01S,** telling the Android Application the command was successful performed.

(2)

The second command first character **[1]** tells the Robot system that is a PID Read Command and the second (x) specify witch Controller the user want to read the value of the gains (x = 1 – Balance Controller, x=2 Steering Controller and x = 3 – Distance Controller).

The Robot System answer to this command has the format similar to the PID Write Command (1x-val1-val2-val3). The first character **[1]** informs the Android Application that this command is an answer to a previous Read Command sent by the Android Application, the second **[x]** character tells the App which PID Controller is the value received from (x = 1 – Balance Controller, x=2 Steering Controller and x = 3 – Distance Controller), and the others are the value of each Gain (**val1** – Kp value, **val2** – Kd value and **val3** - Ki value) separated by the character **[-].**

Example: The User Send the command **11** (command PID Read the gains of the Steering controller) to the Robot System, the Robot System will answer with **11-Kp-Kd-Ki** and Kd, Kp and Ki are the actual value of the gains of the Steering Controller.

(3)

The command number three is used when the used wants to know in that particular moment what is the Robot Current Status, the first character specify the type of the command, in this case is **[3]** for Status command, the second character **[G]** tells the Robot System that the User what to Get the Robot System status.

The Robot System answer to this command with **2-val-x,** where [2] is tells the App that is an answer to a Get Status command, [val] is the actual angle of the Robot system and the [x] character specify what the Robot System is performing at the moment (x = 1 -Just Robot Balancing, x=2 Robot Moving forward, x=3 Robot Moving Backward, x=4 – Robot Turning Right and x= 5 Robot Turning Left).

Example: User send the command **2G** asking the Robot System to send the Robot Status, the Robot System Answer with **2-5-4** telling the used that the Robot Actual Angle is 5 degrees and that the Robot is Performing the Turning Right Command.

(4)

This type of command is sent when the used what to change the Robot System current Position, the command format is **3x,** where [**3**] specify the type of command (movement) and [**x**] specify the type of movement (1 forward, 2 backward, 3 turn right and 4 to turn left).

The Robot System answer with **3S**, where [**3**] tells that is an answer to the movement command, and [**S**] tells that is Successful performed the command.

Example: the user Send **31** telling the Robot System to move forward, after the Robot System perform the command it answer with **3S** telling that the command was successful performed.

9.4.3. Controller Algorithm

To control the Robot is going to be used a PID controller. The use of the PID controller is one of the constraint of the project and for this project is going to be used three PID controllers to control the Robot.

PID Controller

As said before the robot system has three PID Controllers:

Distance PID Controller

This controller is used to make to robot move forward or backward.

The output of this controller (desired angle) is used in the Balancing PID controller, the input of this controller is based on the movement chosen by the user.

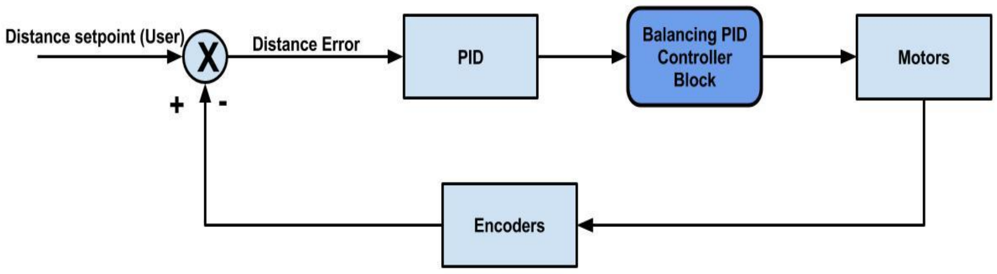


Figure 27: Distance PID controller

The distance can be using the following equation: $\boldsymbol{dist} = \frac{\boldsymbol{dr+dl}}{\boldsymbol{2}}$, where **dr** is the distance measured by the right encoder and **dl** the left encoder.

That way the **error** is equal to $\boldsymbol{dist_{Set} - dist}$, where **dist**$_{\text{Set}}$ is the desired distance/position chosen by the user.

Balancing PID Controller

After the user desired angle has been calculated that value is used as input in this PID controller to calculate the input for the motor, this PID controller uses as input the robot current angle measured by the IMU sensors (accelerometer and gyroscope).

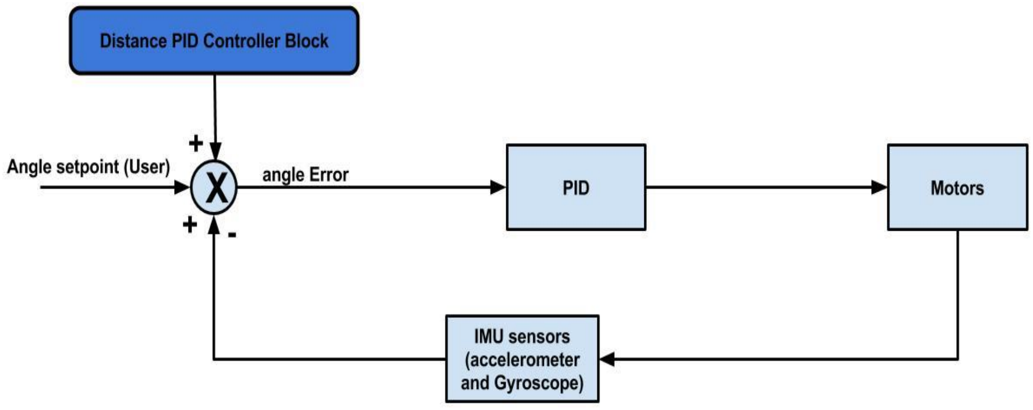

Distance PID Controller Block
Angle setpoint (User)
+
X
+
-
angle Error
PID
Motors
IMU sensors (accelerometer and Gyroscope)

Figure 28: PID controller for the Robot

Steering PID Controller

This PID controller allows the used to control the robot maneuvers (turning right or left), the controller's input are the desired setpoint (turning angle) defined by the user and the actual yaw angle (heading) of the robot measured using the Encoders. Different from the Balancing PID controller this controller output will be applied different in the two motors as shown in the figure 6.

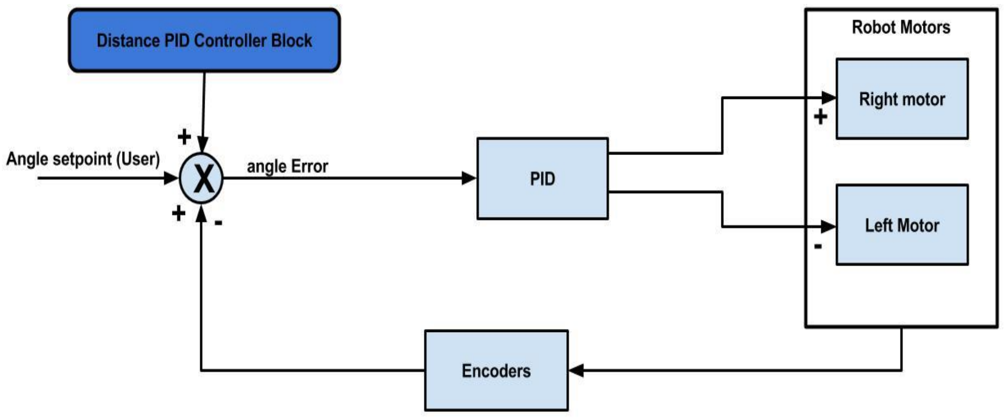


Figure 29: PID controller for the Robot

9.4.4. Angle measurement Algorithm

To calculate the Robot angle is used three sensor, two accelerometers and a gyroscope sensor, is used two acceleromenter because the IMU sensor used in the project include an accelerometer and the the development board also has an accelerometer, the two accelerometers readings will be used to give a better measurement by doing the fusion of the data from both accelerometer.

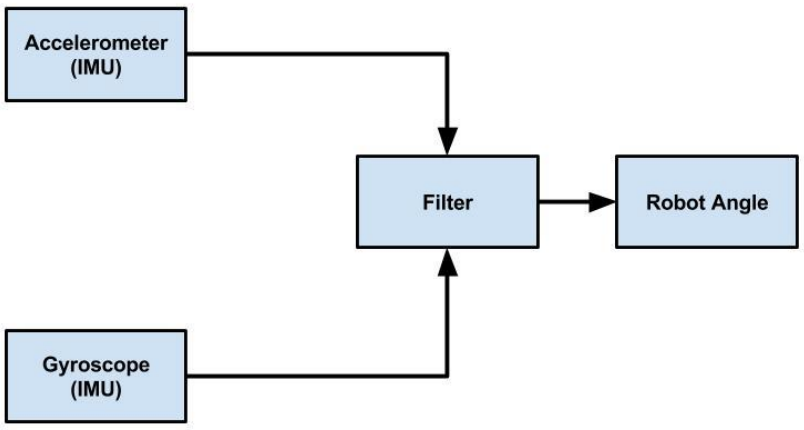


Figure 30: Angle measurement algorithm

It's used two different filter to make the fusion of the Accelerometer Data and the Gyroscope data, the complimentary filter and the Kalman Filter

Complementary Filter

The complementary filter fuses the accelerometer and integrated gyro data by passing the former through a first order low pass and the latter through a first order high pass filter and adding the outputs.

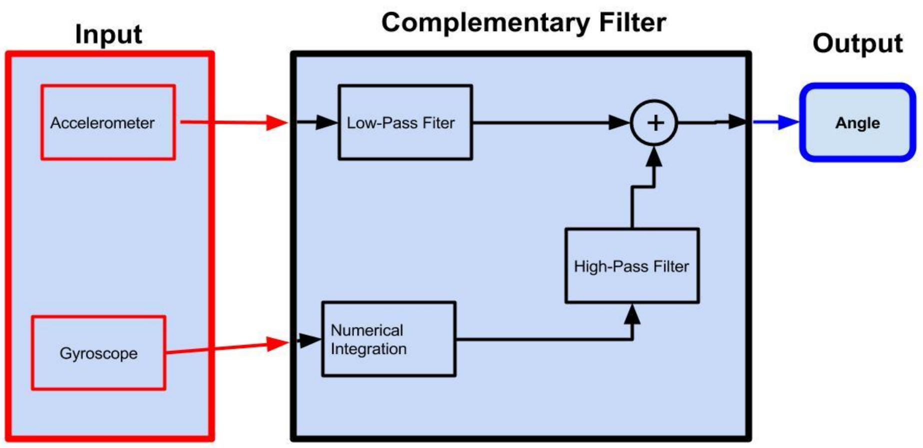


Figure 31: Complementary Filter Overview

9.4.5. Tasks Specification

Here it's the tasks specified in the Design stage of the Waterfall model.

- **Acquisition_Task:** Is responsible to read the data from the accelerometers and gysroscope calculate Gyroscope Rate, the angle using both filters (complimentary or Kalman) and send the data to the BalancingPIDTask
- **Update_EncoderTaks:** this task receive an character from the External interrupt ISR and according to the caracter received it increments or decrement the variable that holds the value of the encoders transitions.
- **BalancingPIDTask**: Receive the Robot current angle and the Gyroscope Rate from the AcquisitionTask use the PID matematical equasion calculate the input for the motors, the value calculated is added to that value the output of DistancePIDControlerTask and sent to the Motor_Controller task.
- **DistancePIDTask**: Receives a signal from the timer ISR calculate the distance traveled by the wheels and used that as input to the PID controller and sends the Output tho the BalancingPIDTaks in a queue
- **MotorControllerTaks:** Recieves the direction and the new PWM values from the BalancingPIDTask and change the motor driver input pins acording to direction received and update the PWM timer with the new values received.
- **USARTReceiveTask :** Receives the data from the Android application via Bluetooth calls the command Handler function.

9.4.6. Task Communication and Synchronization

In onder to exchange data between tasks is used Semaphore, Queues and Mutex.

Semaphore: in the project is used three semaphores to notify task about an event.

- **Timer_ISR_Encoders_Semaphore**: Notifies the task Acquisition_Encoder that a new data is available.
- **Timer_ISR_Semaphore:** Sends a signal to the Acquisition Task to Start a new Sample.

Queue: queue is going to be used to exchange data between tasks, the queues used in this project are:

- **Receive_USART_queue:** used to exchange date from the USART ISR with the USARTReceiveTask
- **Angle_queue:** is used to send the Angle and the Gyroscope Rate from the acquisition task to the Distance_PIDTask.

- **Encoder_queue:** used to send data from the Receive_BluetoothTask to the CommandTask. .
- **PWM_queue:** is used to send the PWM and the direction for both motors from the Acquisition_Task to the Motor_ControllerTask.

Mutex: The mutex are used to protect the access to the critical section of the code, since we are going to use some global variables that are accessed by different tasks is used mutex to control the access of thoses variables. The Mutex used in this project are:

- **s_DistancePID_mutex:** This mutex is used to protect the Struct that has the Parameters for the Balancing PID controler.
- **s_BalancePID_mutex:** This mutex is used to protect the Struct that has the Parameters for the Distance PID controler.
- **Output_Distance_PID_mutex**: since the output of the Distance PID is required in both PID's tasks the access to the variable that holds the value of the Distance PID output is protect by a mutex
- **BalancingRobotEncoder_mutex:** used to protect the acces to the struct that holds the value of the Encoders transitions.
- **USART_Data_mutex :** used to protected the acces to the variables that can be changed by the user using the Android application.

ISRs:

- **ISR_Timer_SamPeriod:** Since the time is really important for this system to be able to work is defined a Sampling Period that signal the program when to start a cicle, The control system is is better for a shorter cicle. For this project the cicle will be under 20ms, that means that the the entire cicle is done at least 50 times every second, to manage the Sampling Period is used a timer that will signal the program to start a new Cicle.
- **ISR_Timer_Encoders:** this ISR sends a signal to the Distance PID Task to calculate the output of the Distance_PID controller, the period of this timer is higher than the Sample Period.
- **Interrupt_EncoderM1:** this ISR sends a caracter in the queue to the Update_EncoderTask or increments the variable that holds the value of the encoder in the right motor transition.
- **Interrupt_EncoderM2:** this ISR sends a caracter in the queue to the Update_EncoderTask or increments the variable that holds the value of the encoders in the left motor transition.

Communication and synchronization overview

The figure 32 shows how the different task of the system interface/communicates with each other and the communication mechanisms used. The figure shows the shared resources and the mutex used to protect the acces to the shared resource. In the figure is also presented the Interrupt Service Routines (ISRs) used to communicate with the task of the Robot System.

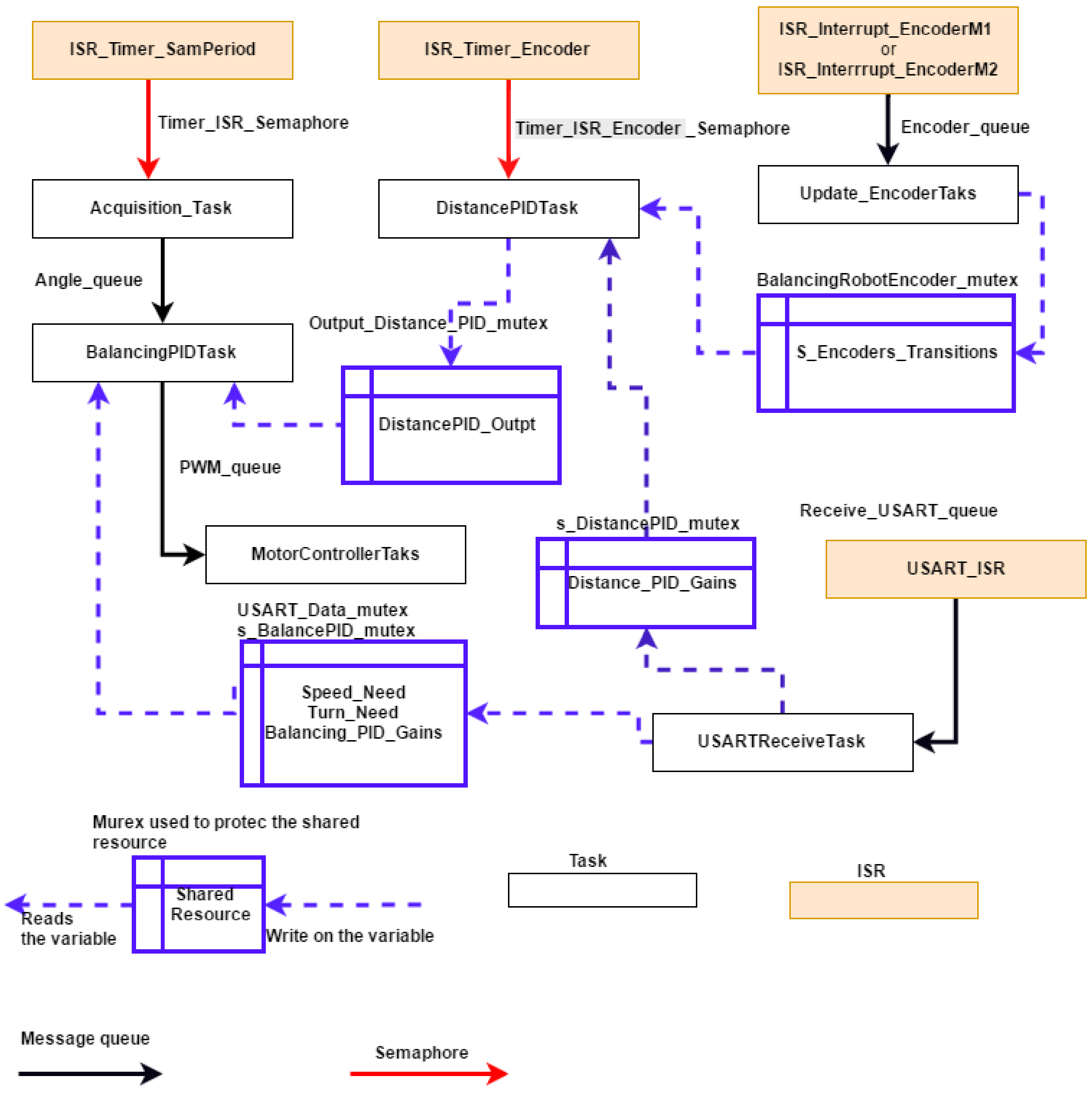


Figure 32: Robot System Tasks Communication and Sincronization Overview

9.4.7. Task Priorities

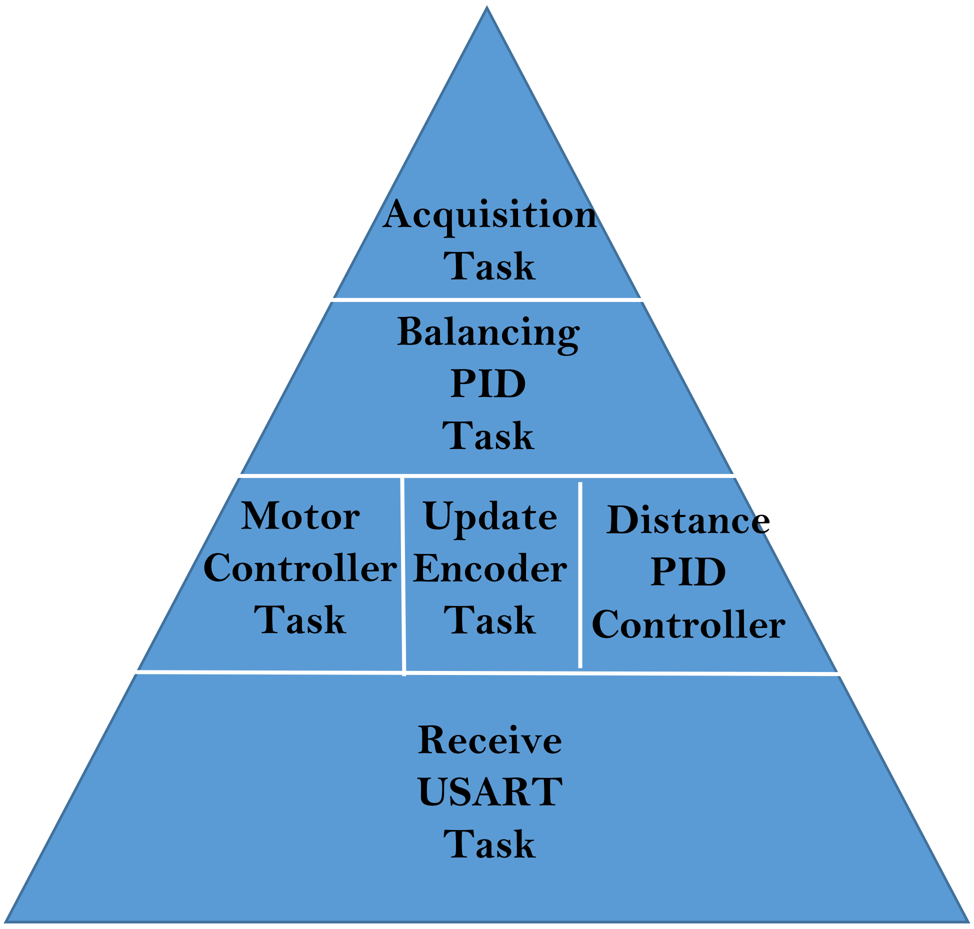


Figure 33: Robot system Task Diagram

Since the most important feature of the system is being able to balance itself the tasks used to perform the balance of the Robot system have higher priority, that's why the Data acquisition tasks and the Balance controller tasks have the highest priority.

The second layer has the second higher priority because they are the tasks that provide data to the Balancing Controller Task

The bottom layer has the lowest priority tasks, the tasks for communication since the communication is less important than the balancing.

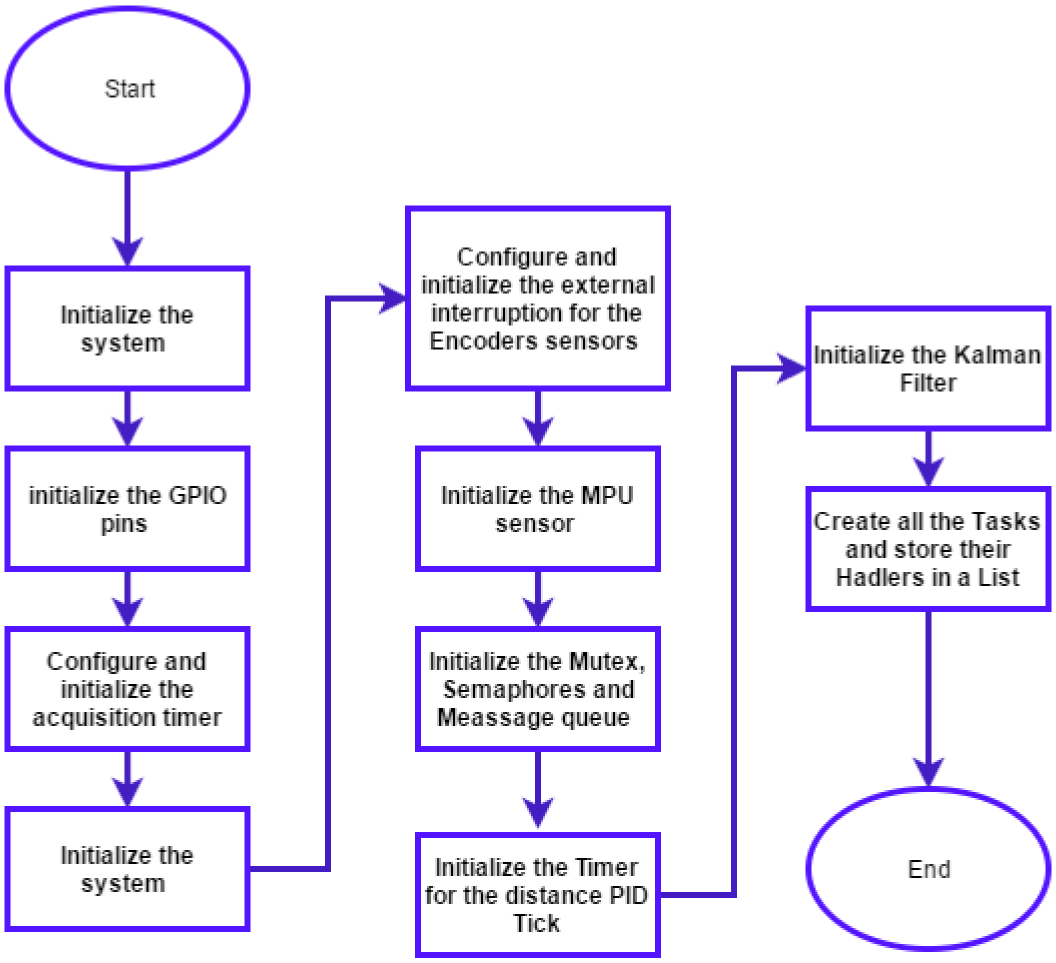
The Android application will not run an Operating System so it doesn't have tasks but functions, the functions in the Android application will be explained in the Flowcharts

10. Flowchart

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.

10.1. Robot System Flowchart

10.1.1. Initializer Function Flowchart

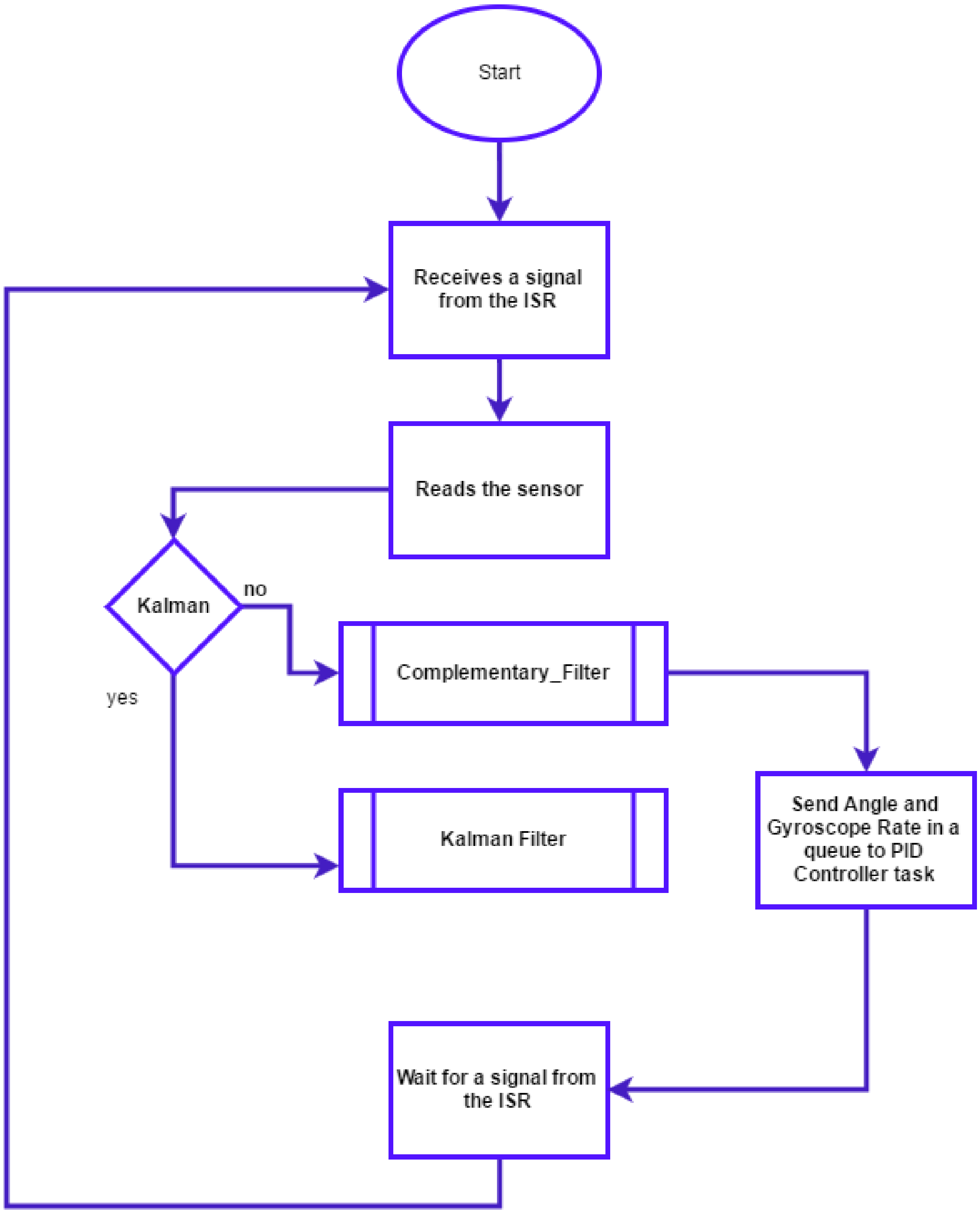


Start
Initialize the system
initialize the GPIO pins
Configure and initialize the acquisition timer
Initialize the system
Configure and initialize the external interruption for the Encoders sensors
Initialize the MPU sensor
Initialize the Mutex, Semaphores and Meassage queue
Initialize the Timer for the distance PID Tick
Initialize the Kalman Filter
Create all the Tasks and store their Hadlers in a List
End

Figure 34: Initializer Flowchart

10.1.2. Acquisition_Task behavior Flowchart

In the acquisition task first the Task is blocked waiting for a signal from the Timer ISR, when It receives the signal the it's going to read the sensors calculate the Robot angle and the gyroscope rate and sends it to the Balancing PID task



Start
Receives a signal from the ISR
Reads the sensor
Kalman
no
yes
Complementary_Filter
Kalman Filter
Send Angle and Gyroscope Rate in a queue to PID Controller task
Wait for a signal from the ISR

Figure 36: Acquisition Task Flowchart

10.1.3. Complimentary and Kalman filter functions Flowchart

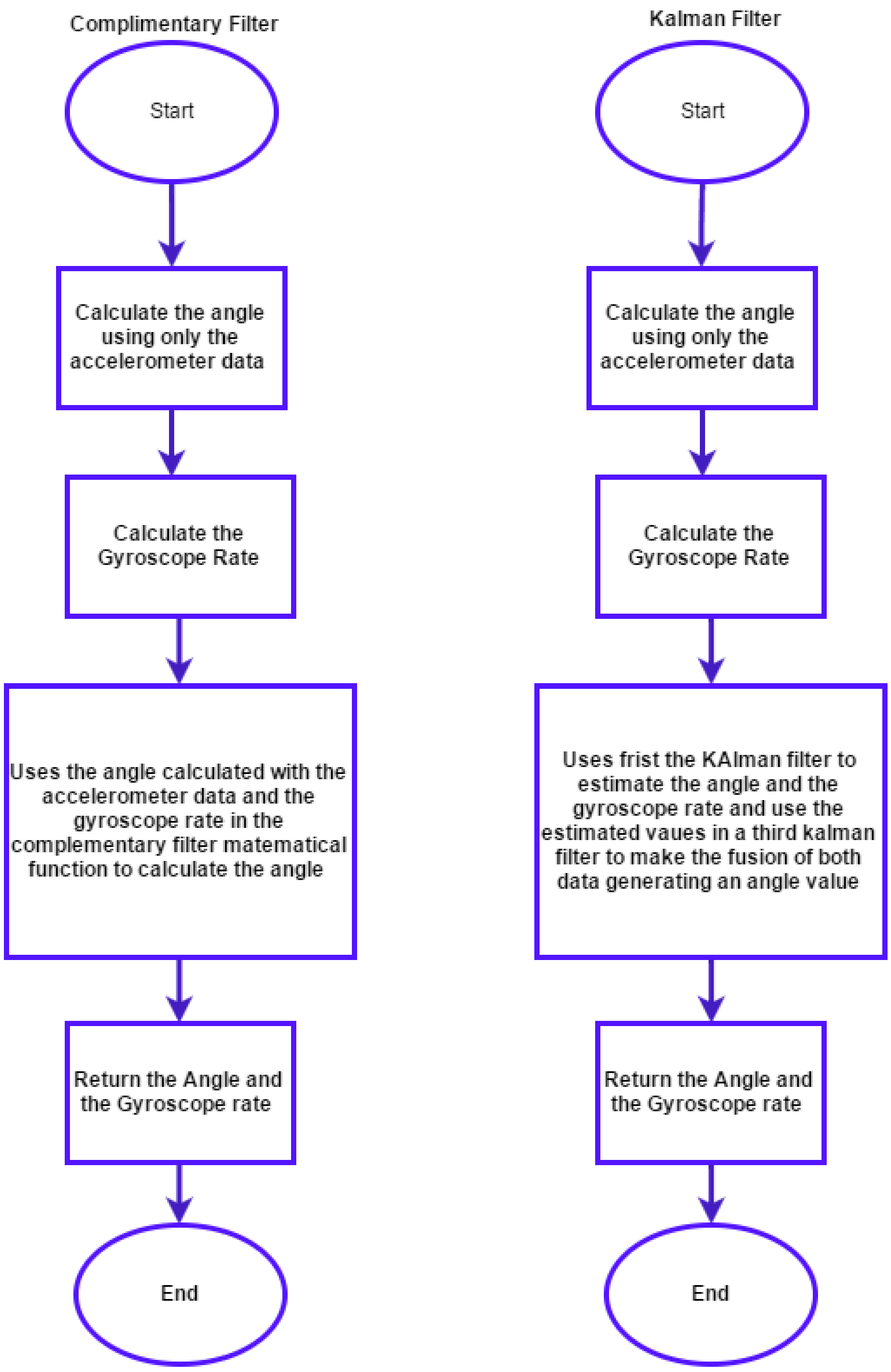
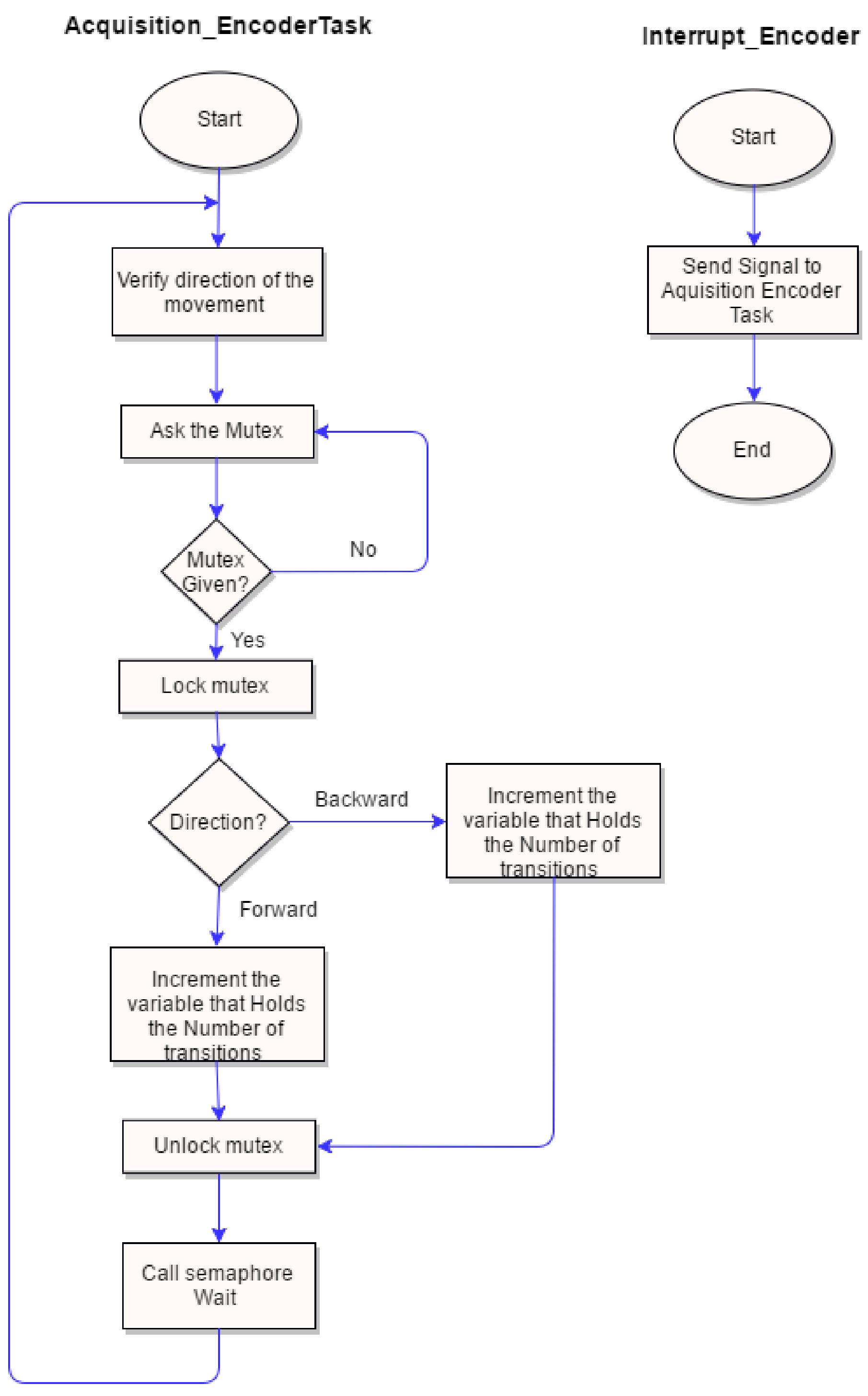

Complimentary Filter
Start
Calculate the angle using only the accelerometer data
Calculate the Gyroscope Rate
Uses the angle calculated with the accelerometer data and the gyroscope rate in the complementary filter matematical function to calculate the angle
Return the Angle and the Gyroscope rate
End
Kalman Filter
Start
Calculate the angle using only the accelerometer data
Calculate the Gyroscope Rate
Uses frist the KAlman filter to estimate the angle and the gyroscope rate and use the estimated vaues in a third kalman filter to make the fusion of both data generating an angle value
Return the Angle and the Gyroscope rate
End

Figure 37: Complimentary and Kalman filter Flowchart

10.1.4. Acquisition_EncoderTask behavior Flowchart

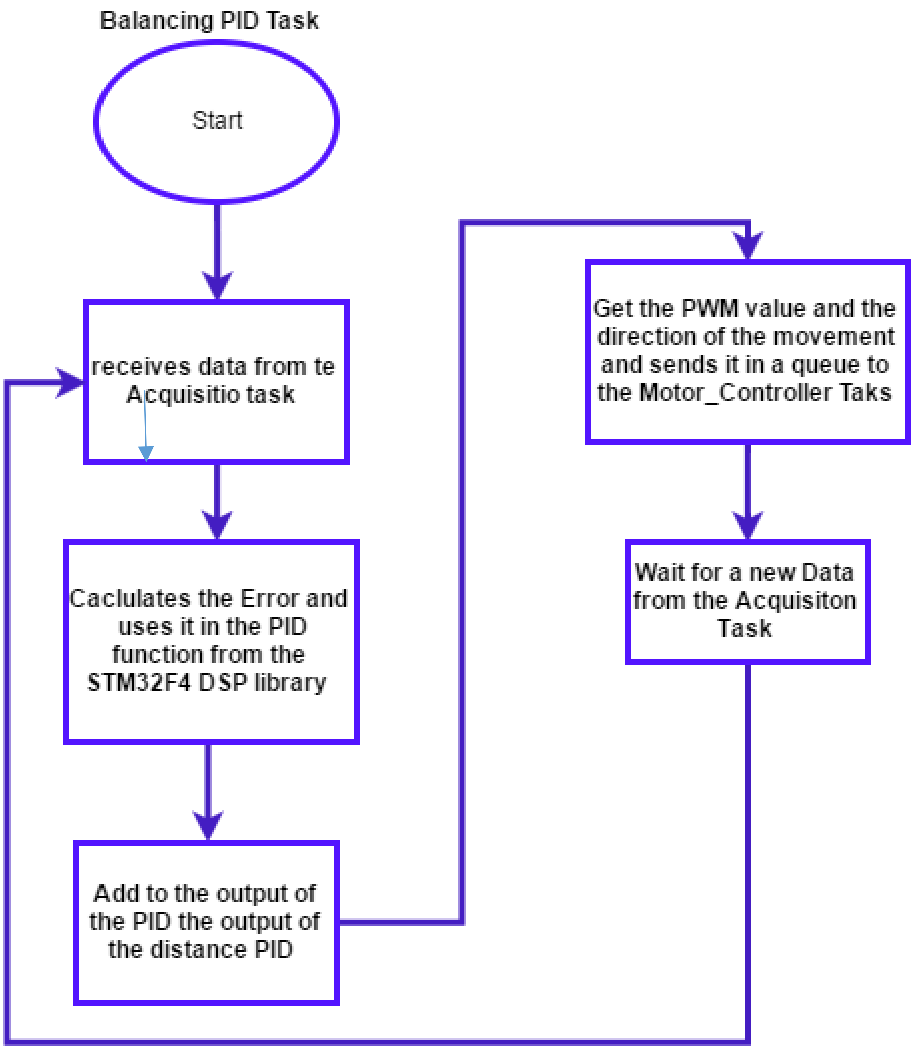
This tasks receives an external interruption from the encoders, first it sees the direction of the movement and according to the direction it increments or decrement the value of the variable that holds the mutex transitions value



Acquisition_EncoderTask
Start
Verify direction of the movement
Ask the Mutex
Mutex Given?
No
Yes
Lock mutex
Direction?
Backward
Increment the variable that Holds the Number of transitions
Forward
Increment the variable that Holds the Number of transitions
Unlock mutex
Call semaphore Wait
Interrupt_Encoder
Start
Send Signal to Aquisition Encoder Task
End

Figure 38: Acquisition Encoder Flowchart

10.1.5. Balance_PIDTask behavior Flowchart



Balancing PID Task
Start
receives data from te Acquisitio task
Caclulates the Error and uses it in the PID function from the STM32F4 DSP library
Add to the output of the PID the output of the distance PID
Get the PWM value and the direction of the movement and sends it in a queue to the Motor_Controller Taks
Wait for a new Data from the Acquisiton Task

Figure 39: Balance PID Task Flowchart

10.1.6. Distance PID Task behavior

This task reads the values of the transitions in the encoders in each motor and calculate the PID using the average of the distance traveled by both motors.

This PID is really important because it helps in the balancing of the Robot by compensating the robot that traveled less or compensating both wheels when the Robot travel more than the setpoint, in other word if the Robot is balancing where the setpoint is zero cause the Robot don't need to move if the robot move in one direction this problem is corrected by adding more value to the motors in the opposite direction

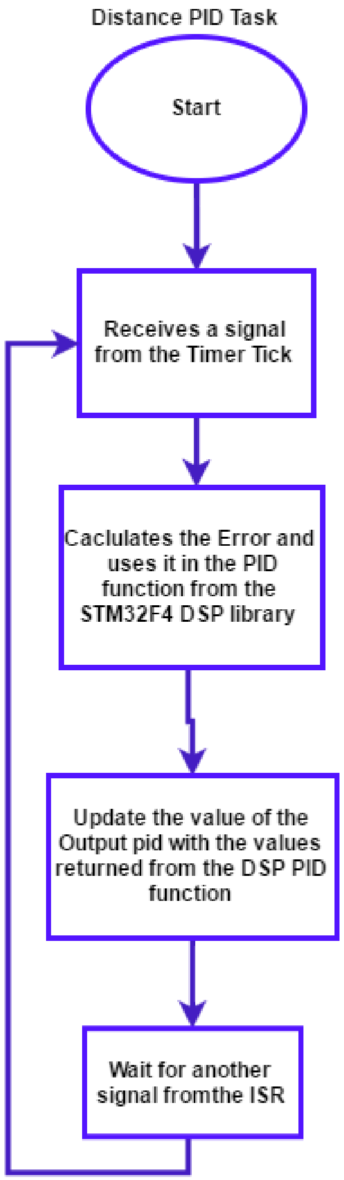
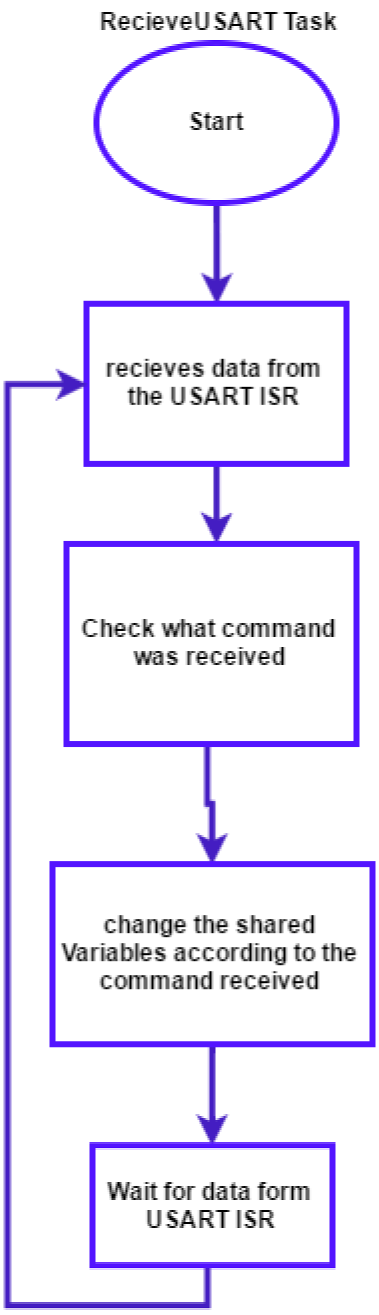

Distance PID Task
Start
Receives a signal from the Timer Tick
Caclulates the Error and uses it in the PID function from the STM32F4 DSP library
Update the value of the Output pid with the values returned from the DSP PID function
Wait for another signal fromthe ISR

Figure 40: Distance PID Task Flowchart

10.1.7. Receive_USARTTask behavior Flowchart



RecieveUSART Task
Start
recieves data from the USART ISR
Check what command was received
change the shared Variables according to the command received
Wait for data form USART ISR

Figure 41: Send and Receive USART Tasks Flowchart

10.1.8. Command Handler Function Flowchart

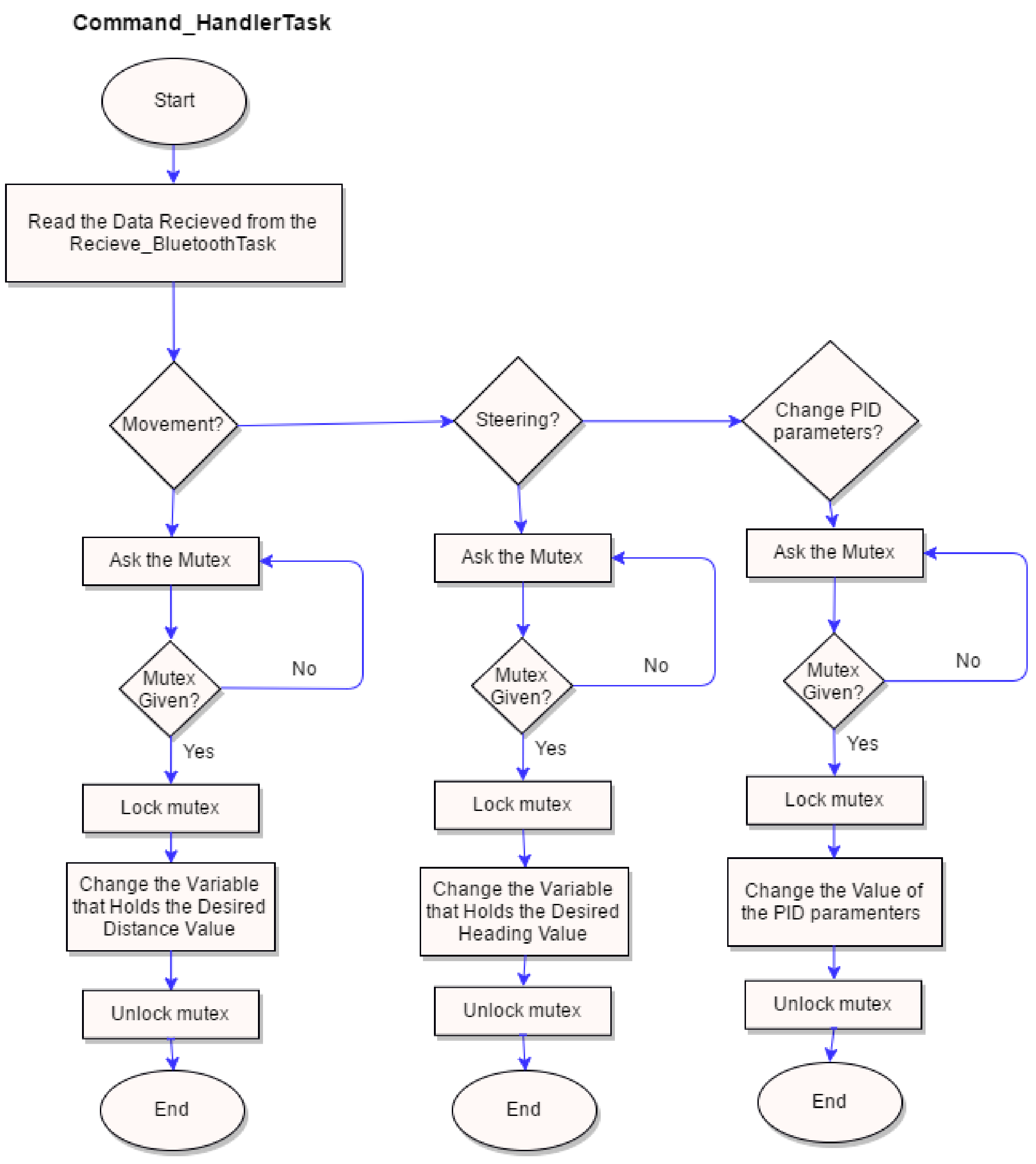

Command_HandlerTask
Start
Read the Data Recieved from the Recieve_BluetoothTask
Movement?
Steering?
Change PID parameters?
Ask the Mutex
Mutex Given?
No
Yes
Lock mutex
Change the Variable that Holds the Desired Distance Value
Change the Variable that Holds the Desired Heading Value
Change the Value of the PID paramenters
Unlock mutex
End

Figure 42: Command Handler Tasks Flowchart

10.2. Android Application Flowchart

This section describes all the functions used in the Android Applications.

10.2.1. Communication (Send and Receive Bluetooth functions)

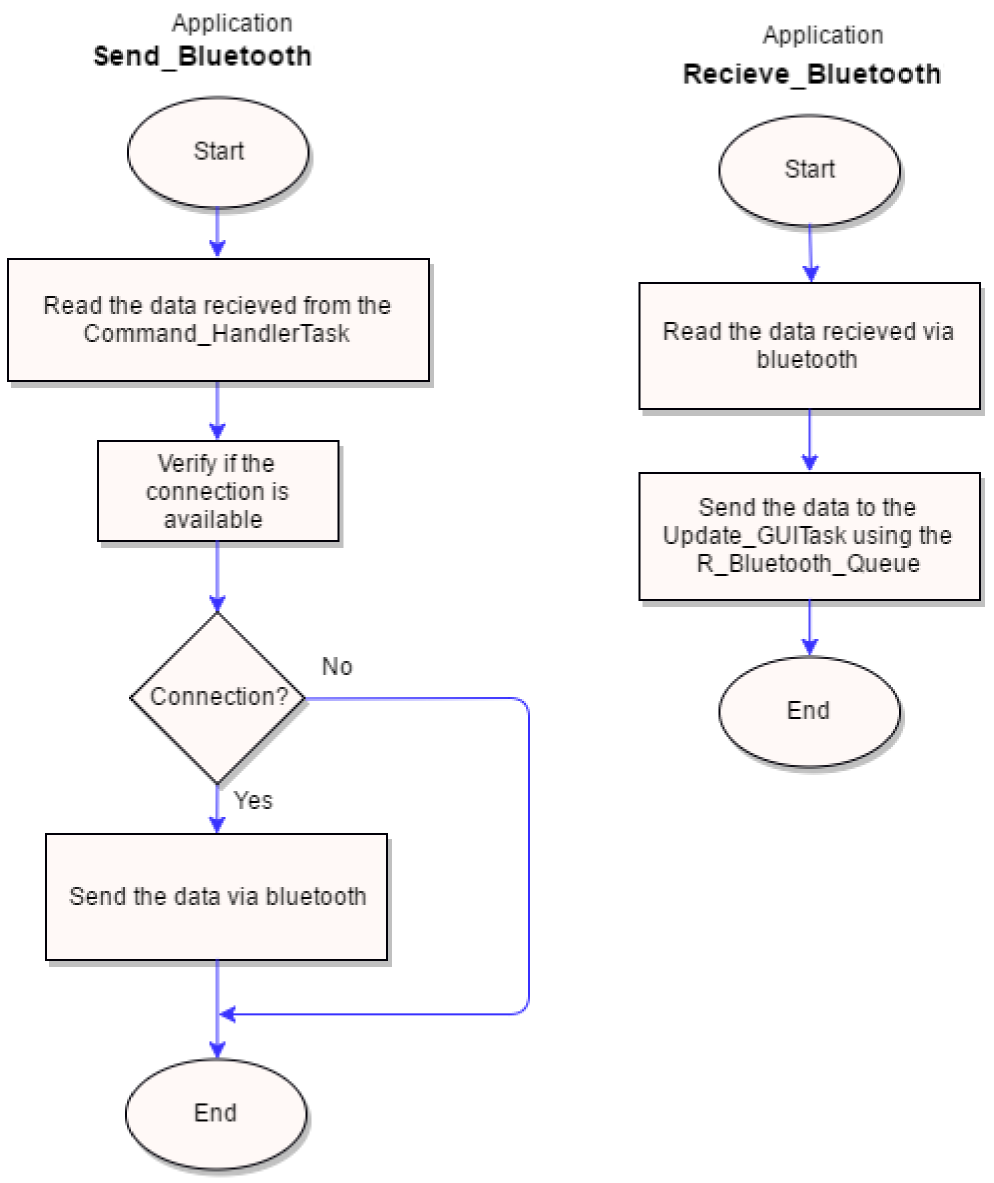
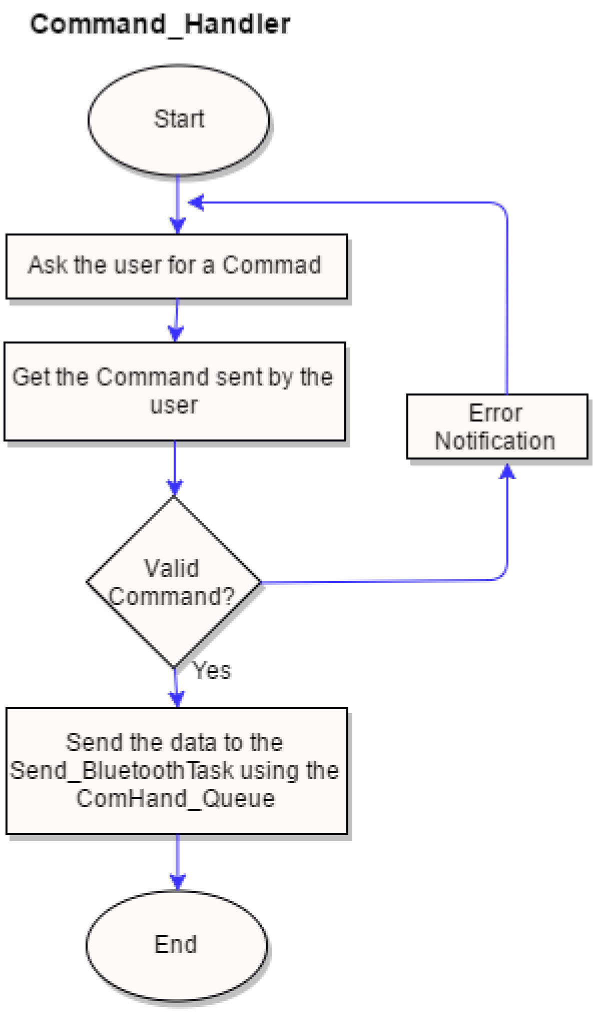

Application
Send_Bluetooth
Start
Read the data recieved from the Command_HandlerTask
Verify if the connection is available
Connection?
No
Yes
Send the data via bluetooth
End
Application
Recieve_Bluetooth
Start
Read the data recieved via bluetooth
Send the data to the Update_GUITask using the R_Bluetooth_Queue
End

Figure 43: Bluetooth Communication Tasks Flowchart (Android Application)

10.2.2. Command_Handler function Flowchart



Command_Handler
Start
Ask the user for a Commad
Get the Command sent by the user
Error Notification
Valid Command?
Yes
Send the data to the Send_BluetoothTask using the ComHand_Queue
End

Figure 44: Command Handler Function Flowchart (Android Application)

11. Implementation

11.1. Local System

11.1.1. Overview

Our code has as base the FreeRTOS real-time operating system and we make use of the STM32F4 DSP and Standard Peripherals Library.

We begin by initializing the system, configuring the Timer 2 to set the acquisition period of the IMU and setting up the external interrupts for the encoders. We initialize the GPIO for the LED that is toggled when we start the data acquisition from the IMU, to signal that the acquisition has started. If we are not getting data from the IMU the code will block, since the accelerometer and gyro data are essential for our system.

Next, we initialize the mutexes, semaphores and message queues that we will use. We setup the PWM for the motors and put them working. The encoder acquisition begins and we initiate the USART to receive data from the application.

The tasks are then initialized and their name and handler stored in a struct. Finally, we define the priorities of the interrupts and begin the task scheduling.

```
int main(void) {

    SystemInit();
    Initialize_Pin();
    Timer2_Init_AcquisitionTick();
    Encoder_Left_Configuration();
    Encoder_Right_Configuration();
    MPU_LED_Init();
    while(MPU_Init());
    FreeRTOS_Init();
    TimerGPIO_Init();
    TIMER_Init();
    PWM_Init();
    //Kalman_init() ;
    MotorController_GPIOInit();
    Timer5_Init_EncodersTick();
    init_USART1(9600);
    BalancingRobotTaskList.i = 0;
    now = 0;
    TaskHandle_t * pxCreatedTask;
    Create_Task((void*)PID_Controller_Function, "BalancingPIDTask",  500, ( void * ) 1, 29 , pxCreatedTask );
    Create_Task((void*)PID_Distance_Function,   "DistancePIDTask",   128, ( void * ) 2, 10 , pxCreatedTask );
    Create_Task((void*)Acquisition_Function,    "AcquisitionTask",   128, ( void * ) 3, 15 , pxCreatedTask );
    Create_Task((void*)Motor_Controller,        "MotorControlerTask",128, ( void * ) 4, 10 , pxCreatedTask );
    Create_Task((void*)Update_Encoder,          "UpdateEncoderTask", 50,  ( void * ) 5, 10  , pxCreatedTask );
    Create_Task((void*)USART_Receive_Function,  "USARTReceiveTask",  128, ( void * ) 6, 6   , pxCreatedTask );
    Create_Task((void*)Test,                    "TestTask",          8,   ( void * ) 7, 0  , pxCreatedTask );

       NVIC_SetPriority(EXTI0_IRQn, 9);
              NVIC_SetPriority(EXTI3_IRQn, 9);
              NVIC_SetPriority(TIM2_IRQn,  6);
              NVIC_SetPriority(TIM5_IRQn,  5);
              NVIC_SetPriority(USART1_IRQn,6);
      vTaskStartScheduler();

}
```

Listing 1 – Main function

11.1.2. Data structures

The Encoder struct is used to store the counters for the encoders, each motor, left and right, has a positive and negative counter that will be incremented accordingly.

```
typedef struct{
	int ENCODER_RIGHT_POSITIVE;
	int ENCODER_RIGHT_NEGATIVE;
	int ENCODER_LEFT_NEGATIVE;
	int ENCODER_LEFT_POSITIVE;
}Encoder;
```

Listing 2 – Encoder struct

The PWM struct contains values for both right and left PWM and their direction.

```
typedef struct{
	int R_PWM;
	int L_PWM;
	short R_Direction;
	short L_Direction;
}PWM;
```

Listing 3 – PWM struct

In the MPU_Data struct we can store the angle obtained by the accelerometer and the gyro rate.

```
typedef struct{
	float Accelerometer_Angle;
	float Gyroscope_Rate;
}MPU_Data;
```

Listing 4 – MPU_Data struct

The Distance_PID struct has the output value for the distance PID as well as the counter differential between both motors.

```
typedef struct{
	float Output;
	float Diferential;
}Distance_PID;
```

Listing 5 – Distance_PID struct

We use a PID_Struct struct to store the gains values of the balance PID.

```
typedef struct{
	float Kp;
	float Ki;
	float Kd;
}PID_Struct;
```

Listing 6 – PID_Struct struct

FromUSART_ISR struct stores the index and gain of the PID parameter to alter, received from the application.

```
typedef struct{
       char Index;
       float Gain;

       }FromUSART_ISR;
```

Listing 7 – FromUSART_ISR struct

In the Tasks_List struct the name and handler of the tasks are stored.

```
typedef struct Node{
            char *  Task_Name[10];
            TaskHandle_t *Task_Handle[10];
                       int i;
}Tasks_List;
```

11.1.3. Global Variables

- **DistancePID**: Struct of type Distance_PID. Used By both PID Tasks to exchange Data (Distance PID output and Encoders Differential).
- s_**BalancingRobotEncoder**: Struct of type Encoder. Used to store the encoders transition by the ISR. This variable is also used by the Distance PID task, as such, in order to synchronize the tasks and the ISR's all interruptions are turn off when the distance PID reads this variable.
- **Output_DistancePID_mutex:** This semaphore is used to synchronize the access to the DistancePID Struct (between both PID Tasks)
- **Timer_ISR_Semaphore:** this Semaphore is used by the ISR for the TIMER2 to signal the Acquisition task when to read the IMU data.
- **Timer_ISR_Encoders_Semaphore:** this semaphore is used by the ISR for the TIMER5 to signal de Distance PID task to calculate the output for the distance PID and the Differential in the encoders.
- **Angle_queue:** this message queue is used to send the Angle and the Gyroscope Rate from the Acquisition Task to the Balance PID Task.
- **Receive_USART_queue:** this message queue is used to send the data received via Bluetooth to the Receive_USART Function.
- **PWM_queue:** To send the output of the Balancing PID (PWM) to the motor controller task we use this queue.
- **f_Read and f_Read2:** We use this variable to make sure the ISR doesn't give the semaphore before the semaphore is created. Normally it doesn't happen but we still didn't understand why when we initialize the timer the execution flow goes to the ISR.
- **MPU6050_Data0:** This variable is defined in the TM MPU6050 library, and is used to store the value of the accelerometer and gyroscope Axis, the variable is used first

in the initialization of the MPU sensor and after it is only used in the Acquisition task, so there is no need to synchronize the access.

- **BalancingRobotTaskList:** this variable is used to Store the handle and the name of all the tasks created, this struct will be used in the cleanup to delete all the tasks.
- **s_BalancePID:** Struct of type PID_Struct. This variable has the Gains Kp, Ki and Kd for the Balance PID.
- **s_BalancePID_mutex:** Since we can update the PID gains we are going to use this mutex to protect the access to the s_Balance_PID.
- **s_DistancePID:** Struct of type PID_Struct. This variable holds the gains for the Distance PID
- **s_DistancePID_mutex:** Similar to the s_BalancePID, the PID gains can be updated so we need to use a mutex to protect the access to the s_DistancePID.

11.1.4. Initialization functions

11.1.4.1. Timer 2

In our system, Timer2 is used to set the acquisition period of the IMU, which we setup as 10 milliseconds.

```
void Timer2_Init_AcquisitionTick(void){

  NVIC_InitTypeDef NVIC_InitStructure;
       TIM_TimeBaseInitTypeDef timerInitStructure;
  /* Enable the TIM2 gloabal Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /* TIM2 clock enable */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
  /* Time base configuration */
  timerInitStructure.TIM_Period = 40-1;  //
  timerInitStructure.TIM_Prescaler = 42000-1;
  timerInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
  timerInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM2, &timerInitStructure);
  /* TIM IT enable */
  TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
  /* TIM2 enable counter */
  TIM_Cmd(TIM2, ENABLE);
}
```

11.1.5. Thread Synchronization and Communication

We initialize every IPC and synchronization in a single function that we called FreeRTOS_Init.

```
void FreeRTOS_Init(void){
	//Semaphore
	Timer_ISR_Semaphore = xSemaphoreCreateBinary();
	Timer_ISR_Encoders_Semaphore = xSemaphoreCreateBinary();

	//Mutex
	Output_DistancePID_mutex = xSemaphoreCreateMutex();
	BalancingRobotEncoder_mutex = xSemaphoreCreateMutex();
	s_DistancePID_mutex = xSemaphoreCreateMutex();
	s_BalancePID_mutex = xSemaphoreCreateMutex();

	//Message queue
	Receive_USART_queue = xQueueCreate( 1, sizeof( char[MAX_R_STRLEN]));
	Angle_queue = xQueueCreate( 1, sizeof( MPU_Data ));
	Encoder_queue = xQueueCreate( 5, sizeof( char ));
	PWM_queue = xQueueCreate( 5, sizeof(PWM));
}
```

Listing 8 – FreeRTOS_Init function

11.1.6. ISR's

11.1.6.1. Timer ISR

and Timer5 is used to set the acquisition period of the encoders.

With the Timer2 ISR we send a signal, using the *Timer_ISR_Semaphore*, to the Acquisition task every 10 milliseconds.

```
void TIM2_IRQHandler(void){
	if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET){
		TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
			if(f_Read == 1){
				f_Read = 0;
				now++;
				now = (now%1000);
				BaseType_t xHigherPriorityTaskWoken;
				xHigherPriorityTaskWoken = pdFALSE;
				GPIO_ToggleBits(GPIOD,GPIO_Pin_10);
				xSemaphoreGiveFromISR( Timer_ISR_Semaphore, &xHigherPriorityTaskWoken );
//Unblock the task by releasing the semaphore.
				if (xHigherPriorityTaskWoken == pdTRUE)
// if true means that the task unblocked has higher priority than the one running
				portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
 // call the scheduler for the context switching
			}
	}
}
```

Listing 9 -Timer 2 IRQ Handler

The timer 5 ISR sends a signal to the Distance PID task, the signal is sent using the Timer_Encoders_Semaphore

```
void TIM5_IRQHandler(void){
	if ((TIM_GetITStatus(TIM5, TIM_IT_Update) != RESET)){
		TIM_ClearITPendingBit(TIM5, TIM_IT_Update);
		BaseType_t xHigherPriorityTaskWoken;
		xHigherPriorityTaskWoken = pdFALSE;
		xSemaphoreGiveFromISR( Timer_ISR_Encoders_Semaphore, &xHigherPriorityTaskWoken );
//Unblock the task by releasing the semaphore.
		if (xHigherPriorityTaskWoken == pdTRUE)
			portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
  }
}
```

Listing 10 – Timer 5 IRQ Handler

11.1.6.2. Encoder ISR

To count the transitions for the encoders we use two external interrupts on GPIOA pin PA3 for the right encoder and on pin PA0 for the left encoder.

We have setup the external interrupt for a rising or falling pulse in PA3, when that occurs, our handler function checks the state of PA2 pin, which is connected to the encoder as well, and increments the counter variable accordingly.

```
void EXTI3_IRQHandler(void){
	if((EXTI_GetITStatus(EXTI_Line3) != RESET)){
		EXTI_ClearITPendingBit(EXTI_Line3);

		if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_3) == 1 && GPIO_ReadInputDataBit(GPIOA,
		GPIO_Pin_2) == 1){ // 11
			s_BalancingRobotEncoder.ENCODER_RIGHT_NEGATIVE++;
		}
		else if((GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_3) == 0) && GPIO_ReadInputDataBit(GPIOA,
		GPIO_Pin_2) ==1){ // 10
			s_BalancingRobotEncoder.ENCODER_RIGHT_POSITIVE++;

		}
		else if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_2) == 1 && (GPIO_ReadInputDataBit(GPIOA,
GPIO_Pin_3) == 0)){ //01
			s_BalancingRobotEncoder.ENCODER_RIGHT_POSITIVE++;
		}
		else if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_2) == 0 && (GPIO_ReadInputDataBit(GPIOA,
GPIO_Pin_3) == 0)){
			s_BalancingRobotEncoder.ENCODER_RIGHT_NEGATIVE++;

				}
		}
}
```

Listing 11 – External Interrupt 3 IRQ Handler

The same happens for the external interrupt in PA0, being the other connected pin the PA1.

```
void EXTI0_IRQHandler(void){
	if(EXTI_GetITStatus(EXTI_Line0) != RESET){
		EXTI_ClearITPendingBit(EXTI_Line0);

		if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0 == 1) && GPIO_ReadInputDataBit(GPIOA,
GPIO_Pin_1) == 1){ // 11
		s_BalancingRobotEncoder.ENCODER_LEFT_POSITIVE++;
		}
		else if((GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0) == 0) && GPIO_ReadInputDataBit(GPIOA,
GPIO_Pin_1) == 1){ // 10
		s_BalancingRobotEncoder.ENCODER_LEFT_NEGATIVE++;
		}
		else if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0 == 1) && (GPIO_ReadInputDataBit(GPIOA,
GPIO_Pin_1) == 0)){ //01
		s_BalancingRobotEncoder.ENCODER_LEFT_NEGATIVE++;
		}
		else if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0 == 0) && (GPIO_ReadInputDataBit(GPIOA,
GPIO_Pin_1) == 0)){  ///////
		s_BalancingRobotEncoder.ENCODER_LEFT_POSITIVE++;
				}
		}
}
```

11.1.6.3. USART ISR

In the USART ISR we receive the data from the remote app through Bluetooth, the character received is saved in an array until '/n', signaling the end of the command, is received or the maximum length of the array has been reached. Once that happens, the counter is reset and the received command sent to the queue for the *USART_Receive_Task* to receive.

```
void USART1_IRQHandler(void){
	static portBASE_TYPE xHigherPriorityTaskWoken;
	static char received_string[MAX_R_STRLEN];

	 xHigherPriorityTaskWoken = pdFALSE;
	if( USART_GetITStatus(USART1, USART_IT_RXNE) ){

		static uint8_t cnt = 0; // this counter is used to determine the string length
		char c = USART1->DR; // the character from the USART1 data register is saved in t

	/* check if the received character is not the LF character (used to determine end of string)
		 * or the if the maximum string length has been been reached
		 */
		if( (c != '\n') && (cnt < MAX_R_STRLEN) ){
			received_string[cnt] = c;
			cnt++;
		}
		else{ // otherwise reset the character counter and print the received string
			cnt = 0;

	xQueueSendToBackFromISR(Receive_USART_queue,(void*)&received_string,&xHigherPriorityTaskWoken);
		  if (xHigherPriorityTaskWoken == pdTRUE)
			portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
		}
	}
}
```

Listing 12 – USART ISQ Handler

11.1.7. Tasks behavior

11.1.7.1. Acquisition task

When this task receives a signal from the timer2 ISR, the IMU data is acquired and treated in the complementary filter function, where the Robot current angle and gyroscope rate are calculated.

The obtained angle and gyro rate are then sent to the Balance PID task using the *Angle_Queue* message queue.

```
void Acquisition_Function(void * args){
	short vec[3], vec2[3];
	MPU_Data ToPID;


	while(1){
		f_Read = 1;
		if(xSemaphoreTake( Timer_ISR_Semaphore, LONG_TIME ) == pdTRUE ){
			TM_MPU6050_ReadAll(&MPU6050_Data0);
		vec[0] = MPU6050_Data0.Accelerometer_X;
			vec[1] = MPU6050_Data0.Accelerometer_Y;
			vec[2] = MPU6050_Data0.Accelerometer_Z;
			vec2[0]= MPU6050_Data0.Gyroscope_X;
			vec2[1]= MPU6050_Data0.Gyroscope_Y;
			vec2[2]= MPU6050_Data0.Gyroscope_Z;
			ComplementaryFilter(vec,vec2, &ToPID.Accelerometer_Angle, &ToPID.Gyroscope_Rate);
			if( xQueueSend( Angle_queue,( void * ) &ToPID,( TickType_t ) 10 ) != pdPASS ){
			//error
			}
		}
	}
}
```

Listing 13 – Acquisition_Function()

11.1.7.2. Balancing PID task

This is the task for the Balance PID, the task receives the Angle value and the Gyroscope Rate from the acquisition task and calculates the PWM for the motors using the PID mathematical Equation.

To the output of the Balance PID is subtracted the output of the Distance PID and then sent to the Motor Controller task through the *PWM_queue* message queue to change the motor PWM values. In the case a direction command was received it will also be added to the PWM values, using the *speed_need* and *turn_need* variables.

This function also checks if auto-tuning is enabled and in that case, uses the respective function for it.

```
void PID_Controller_Function(void * args){
	PWM s_BalancingRobotPWM;
	MPU_Data  MPU_DataPID;
	float Setpoint = -0, PWM_Input2 = 0, PWM_Right, PWM_Left,Error, Last_Error, RobotCurrentAngle = 0,
Gyro_Rate; // we can use the Gyro angle on the Kd term
	float IMAX = 100, IMIN = -100,P_in, D_in, I_in2;
	s_BalancePID.Kp = 80.0, s_BalancePID.Kd = 0.1, s_BalancePID.Ki = 7.0;
	PID_ATune(&RobotCurrentAngle, &PWM_Input2, &AT);
	while (1){
		if( xQueueReceive( Angle_queue, &( MPU_DataPID ),( TickType_t ) 10 )){
			if( xSemaphoreTake( s_BalancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
				if(!Stop_Flag){
					RobotCurrentAngle = MPU_DataPID.Accelerometer_Angle;
					Gyro_Rate = MPU_DataPID.Gyroscope_Rate;
					//PID Calculation
					Error = Setpoint - RobotCurrentAngle;
					if(Auto_Tunning){
						int a = PID_ATune_Run(&AT);
						if(a != 0) {
							s_BalancePID.Kp = PID_ATune_GetKp(&AT);
							s_BalancePID.Kd = PID_ATune_GetKd(&AT);
							s_BalancePID.Ki = PID_ATune_GetKi(&AT);
						}
					}
					P_in = s_BalancePID.Kp * Error;
					D_in = s_BalancePID.Kd * Gyro_Rate; // (Error + Last_Error);
					I_in2 += (s_BalancePID.Ki*Error);
					if (I_in2 > IMAX) I_in2 = IMAX;
					else if (I_in2 < IMIN) I_in2 = IMIN;

					xSemaphoreGive(s_BalancePID_mutex);
					PWM_Input2 = P_in + D_in + I_in2;
					PWM_Right = (PWM_Input2);
					PWM_Left = (PWM_Input2);
					Last_Error = Error;
				}
				if( (xSemaphoreTake(BalancingRobotEncoder_mutex,(TickType_t) 10 ) == pdTRUE)){
					if(!speed_need)PWM_Input2 -= DistancePID.Output/100;
						PWM_Right=(PWM_Input2-DistancePID.Diferential+turn_need-speed_need);
						PWM_Left=(PWM_Input2+DistancePID.Diferential-turn_need-speed_need)*1.2;
						xSemaphoreGive(BalancingRobotEncoder_mutex);
					}
				}
				else{
					PWM_Right = 0;
					PWM_Left = 0;
				}
				if( PWM_Right < 0){
					PWM_Right = - PWM_Right;
					s_BalancingRobotPWM.R_Direction = 0;
				}else{
					s_BalancingRobotPWM.R_Direction= 1;
				}
				if( PWM_Left < 0){
					s_BalancingRobotPWM.L_Direction = 0;
					PWM_Left = - PWM_Left;
				}else{
					s_BalancingRobotPWM.L_Direction = 1;
				}
				s_BalancingRobotPWM.R_PWM = ((int)(PWM_Right*1000))/1000 ;
				s_BalancingRobotPWM.L_PWM = ((int)(PWM_Left*1000))/1000  ;
				// send the PWM to the Motor controller task
				if(xQueueSend(PWM_queue,(void *) &s_BalancingRobotPWM,(TickType_t) 10 ) != pdPASS );
				}
		}
}
```

Listing 14 – PID_Controller_Function()

11.1.7.3. Distance PID task

This task is responsible to calculate the output of the Distance PID according to the encoders value, the task use two global variables (DistancePID and s_BalancingRobotEncoder).

```
void PID_Distance_Function(void * args){
	//float Kp = 20.00, Ki = 4.00 ;
	s_DistancePID.Kp = 20.00, s_DistancePID.Ki = 4.00, s_DistancePID.Kd = 0.0;
	float E_Right = 0.0, E_Left = 0.0;
	float Error = 0.00, Total_Distance = 0.00, Total_Distance_Dif = 0.0;
	int Imax = 100,Imin = -100;
	char StopFlag = '1';
	while (1){
		if(xSemaphoreTake( Timer_ISR_Encoders_Semaphore, LONG_TIME ) == pdTRUE ){
			E_Right =s_BalancingRobotEncoder.ENCODER_RIGHT_POSITIVE -
		s_BalancingRobotEncoder.ENCODER_RIGHT_NEGATIVE;
			E_Left  =  s_BalancingRobotEncoder.ENCODER_LEFT_POSITIVE  -
		s_BalancingRobotEncoder.ENCODER_LEFT_NEGATIVE;
			s_BalancingRobotEncoder.ENCODER_RIGHT_POSITIVE = 0;
			s_BalancingRobotEncoder.ENCODER_RIGHT_NEGATIVE = 0;
			s_BalancingRobotEncoder.ENCODER_LEFT_NEGATIVE  = 0;
			s_BalancingRobotEncoder.ENCODER_LEFT_POSITIVE  = 0;
			// End critical state

			Error = (E_Right + E_Left)*0.5;
			Total_Distance += Error;
			Total_Distance_Dif += (E_Right - E_Left);
			if (Total_Distance > Imax)
				Total_Distance = Imax;
			else if(Total_Distance < Imin)
				Total_Distance = Imin;
			if( xSemaphoreTake( BalancingRobotEncoder_mutex, ( TickType_t ) 10 ) == pdTRUE ){
				DistancePID.Output = Error*s_DistancePID.Kp + Total_Distance*s_DistancePID.Ki;
				DistancePID.Diferential = (E_Right - E_Left);
				xSemaphoreGive(BalancingRobotEncoder_mutex);
			}
		}
	}
}
```

Listing 15 – PID_Distance_Function()

11.1.7.4. Motor Controller task

This task receives the PWM output for the two motors from the Balance PID task and configures the Motor Driver Input pins according to the values received on the queue.

```
void Motor_Controller(void *args){
       PWM s_PWM;

       while(1){
       if( xQueueReceive( PWM_queue, &( s_PWM ),( TickType_t ) 10 )){
              if(s_PWM.R_Direction == 1){
                     GPIO_SetBits(GPIOE, GPIO_Pin_12);
                     GPIO_ResetBits(GPIOE, GPIO_Pin_11);
              }
              else{
                     GPIO_SetBits(GPIOE, GPIO_Pin_11);
                     GPIO_ResetBits(GPIOE, GPIO_Pin_12);
              }
              if(s_PWM.L_Direction == 1){
                     GPIO_SetBits(GPIOE, GPIO_Pin_13);
                     GPIO_ResetBits(GPIOE, GPIO_Pin_14);
              }
              else{
                     GPIO_SetBits(GPIOE, GPIO_Pin_14);
                     GPIO_ResetBits(GPIOE, GPIO_Pin_13);
              }
              if (s_PWM.R_PWM > 255)
                     s_PWM.R_PWM = 255;
              if (s_PWM.L_PWM > 255)
                     s_PWM.L_PWM = 255;
              UpdatePWMDutyCycle((s_PWM.R_PWM*11),1);
              UpdatePWMDutyCycle((s_PWM.L_PWM*11),2);
              GPIO_ToggleBits(GPIOD,GPIO_Pin_10);
              }
       }
}
```

Listing 16 – Motor_Controller() function

11.1.7.5. USART Receive Task

Receives the command string through the *Receive_USART_queue* queue from the USART ISR handler and treats it in the *commandHandler* function.

```
static void USART_Receive_Function(void* args){
		/** Pid gain index:
		1 - Kp - Balance PID
		2 - Kd - Balance PID
		3 - Ki - Balance PID
		---------------------
		4 - Kp - Distance PID
		5 - Kd - Distance PID
		6 - Ki - Distance PID*/
	static char received_string[MAX_S_STRLEN];

	while(1){
	f_Read2 = 1;
		if( xQueueReceive(Receive_USART_queue,&(received_string),( TickType_t ) 10 )){
		commandHandler(received_string);
		}
	}
}
```

Listing 17 – UART_Receive_Function()

11.1.8. Functions

11.1.8.1. Complementary Filter

This function uses the accelerometer and gyro data acquired and treats it to obtain the angle and gyro rate.

```
float ComplementaryFilter(short accData[3], short gyrData[3], float *Angle_Average, float *Gyro){
	float Gyro_X = 0.00, X_Angle_1 = 0.00,Y_Angle_1  = 0.00,X_Angle_2 = 0.00, Y_Angle_2  = 0.00;
	static float X_Angle_3 = 0.00;
	static float Y_Angle_3 = 0.00;
	//Gyro rate
	Gyro_X = gyrData[0] / GYROSCOPE_SENSITIVITY; // º/s datasheet pagina 12
	float Gyro_Y = gyrData[1] / GYROSCOPE_SENSITIVITY; // º/s datasheet pagina 12

	//using only the ACC
	X_Angle_1 = atan2((float)accData[1], (float)accData[2]) * 180 / PI;
	Y_Angle_1=atan((float)-accData[0]/sqrt((float)accData[1]+(float)accData[2]*(float)accData[2]))*180/PI;

	double rol = atan2(accData[1], accData[2]) * RAD_TO_DEG;
	double pitch= atan(-accData[0]/sqrt(accData[1]*accData[1] + accData[2] * accData[2])) * RAD_TO_DEG;
	double roll2= atan(accData[1]/ sqrt(accData[0] * accData[0] + accData[2] * accData[2])) * RAD_TO_DEG;
	double pitch2 = atan2(-accData[0], accData[2]) * RAD_TO_DEG;

	 //using a low pass filter on the acc
	X_Angle_2 = X_Angle_2*0.75 + 0.25*X_Angle_1;
	Y_Angle_2 = Y_Angle_2*0.75 + 0.25*X_Angle_1;

	//complementary filter
	float a = 0.98*(X_Angle_3 + Gyro_X * 0.011);
	float b = 0.02*(X_Angle_1);
	X_Angle_3 = a + b;

	float c = 0.98*(Y_Angle_3 + Gyro_Y * 0.011);
	float d = 0.02*(Y_Angle_1);
	Y_Angle_3 = c + d;

	//Kalman Filter
	float aux  = (float)getAngle(X_Angle_1,Gyro_X ,0.011);
	float aux2 = (float)getAngle(Y_Angle_1,Gyro_Y,0.011);

	*Angle_Average =   X_Angle_3;//(2*aux );//+ X_Angle_3 + X_Angle_1)/3;
	//*Angle_Average = (aux2 + Y_Angle_3 + Y_Angle_1)/3;
	*Gyro = Gyro_X;
}
```

Listing 18 – Complementary Filter Function

11.1.8.2. Command Handler

The command handler function treats the command string received. If the first character of the string is '0' it is a change values command, we check the following characters which are then saved in the character array *command* until the separator '-', signaling the end of this value, is found. The command is then used in the *changePIDValues* function. The code then goes to the next value and does the same thing, until it reaches the '+' character, which signals that there are no more values to change.

If the first character received is 'p', the current PID values will be sent to the remote app. If the first character isn't any of the previous ones it means it is a movement command. 'u' for up, 'd' for down, 'l' for left and 'r' for right, 's' for stop each has a different index used on the *speed* function. The 'o' character means the app requested for the motors to turn off, if the character is received again it means the user wants the buttons to turn on so the *Stop_Flag* is toggled.

```
void commandHandler(char* received_string){
	int i = 1;
	int count = 0;
	char command[8];

	if (received_string[0] == '0'){ //0-received values;
		while(received_string[i] != '+'){
			if(received_string[i] != '-'){
				command[count] = received_string[i];
				count++;
			}
			else {
				command[count] = ' ';
				changePIDValues(command);
				count = 0;
			}
			i++;
		}
		sendPIDValues();
	}
	  else if (received_string[0] == 'p'){ //App requested the PID values (sent
	                                                             //after connection)
			sendPIDValues();
	}
	else
		switch (received_string[0]){
			case 's': speed(0); break;
			case 'u': speed(1); break;
			case 'd': speed(2); break;
			case 'r': speed(3); break;
			case 'l': speed(4); break;
			case 'o': {Stop_Flag++;
			           Stop_Flag = (Stop_Flag%2);
			          } break;
		}
	}
```

Listing 19 – Command Hander

11.1.8.3. changePIDValues()

With this function, we change the gain values of both PID structs. Which are protected by mutexes. The first character from the command is the index of the PID gain value to change. From the first character after the index until reaching ' ' the value of the gain is converted to float. Then depending on the index the corresponding PID gain is altered.

```c
void changePIDValues(char* command){
	FromUSART_ISR FromISR;

	char *start = &command[1];
	FromISR.Gain = strtof(start, NULL);
	FromISR.Index = command[0];

	switch (FromISR.Index){
		case '1':{
			if( xSemaphoreTake( s_BalancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
				s_BalancePID.Kp = FromISR.Gain;
				xSemaphoreGive(s_BalancePID_mutex);
			}
		} break;
		case '2':{
			if( xSemaphoreTake( s_BalancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
				s_BalancePID.Kd= FromISR.Gain;
				xSemaphoreGive(s_BalancePID_mutex);
			}
		} break;
		case '3': {
			if( xSemaphoreTake( s_BalancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
				s_BalancePID.Ki = FromISR.Gain;
				xSemaphoreGive(s_BalancePID_mutex);
			}
		}break;
		case '4':{
			if( xSemaphoreTake( s_DistancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
				s_DistancePID.Kp = FromISR.Gain;
				xSemaphoreGive(s_DistancePID_mutex);
			}
		} break;
		case '5':{
			if( xSemaphoreTake( s_DistancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
				s_DistancePID.Kd = FromISR.Gain;
				xSemaphoreGive(s_DistancePID_mutex);
			}
		} break;
		case '6':{
			if( xSemaphoreTake( s_DistancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
				s_DistancePID.Ki = FromISR.Gain;
				xSemaphoreGive(s_DistancePID_mutex);
				}
			} break;
		default: ;
	}
}
```

Listing 20 – changePIDValues() function

11.1.8.4. sendPIDValues()

With this function we check every PID gain value, protected by mutexes, convert them from float to a character array with values up to two decimal cases and send them to the remote application, separated by '-' and adding the carriage return and end of line characters to the last value sent.

```
void sendPIDValues(){
	char send_string[MAX_S_STRLEN+1];
	float value;

	if( xSemaphoreTake( s_BalancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
		value = s_BalancePID.Kp;
		xSemaphoreGive(s_BalancePID_mutex);
		snprintf(send_string, 10, "1%.2f-", value);
		USART_puts(USART1, send_string);
	}
	if( xSemaphoreTake( s_BalancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
		value = s_BalancePID.Kd;
		snprintf(send_string, 10, "2%.2f-", value);
	  USART_puts(USART1, send_string);
		xSemaphoreGive(s_BalancePID_mutex);
	}
	if( xSemaphoreTake( s_BalancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
		value = s_BalancePID.Ki;
		snprintf(send_string, 10, "3%.2f-", value);
		USART_puts(USART1, send_string);
		xSemaphoreGive(s_BalancePID_mutex);
	}
	if( xSemaphoreTake( s_DistancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
		value = s_DistancePID.Kp;
		snprintf(send_string, 10, "4%.2f-", value);
		USART_puts(USART1, send_string);
		xSemaphoreGive(s_DistancePID_mutex);
	}
	if( xSemaphoreTake( s_DistancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
		value = s_DistancePID.Kd;
		snprintf(send_string, 10, "5%.2f-", value);
		USART_puts(USART1, send_string);
		xSemaphoreGive(s_DistancePID_mutex);
	}
	if( xSemaphoreTake( s_DistancePID_mutex, ( TickType_t ) 10 ) == pdTRUE ){
		value = s_DistancePID.Ki;
		snprintf(send_string, 10, "6%.2f\r\n", value);
		USART_puts(USART1, send_string);
		xSemaphoreGive(s_DistancePID_mutex);
	}
}
```

Listing 21 – sendPIDValues() function

11.1.8.5. USART_puts()

This function is used to send strings with the USART, which are transmitted via Bluetooth to the remote application.

```
void USART_puts(USART_TypeDef* USARTx, volatile char *s){

	while(*s){
		// wait until data register is empty
		while( !(USARTx->SR & 0x00000040) );
		USART_SendData(USARTx, *s);
		*s++;
	}
}
```

Listing 22 – USART_puts() function

11.2. Remote System

11.2.1. Overview

Our android application has two activities, the first one only shows up if Bluetooth is not enabled on the device, in that case, the activity will request for the user to enable it. Then it goes to our other activity where it enables the Bluetooth and where most of our application functionalities are.
For the application layout we have 3 buttons in the upper bar, in order: Auto-Tuning enable; Bluetooth connection; Power Motors. Then we have 6 input boxes and 6 text boxes (invisible with no values) to display and input the PID gain values. The Send button, to send the values written. Finally, the arrow buttons for the direction commands. For better responsiveness, these buttons are highlighted when pressed.

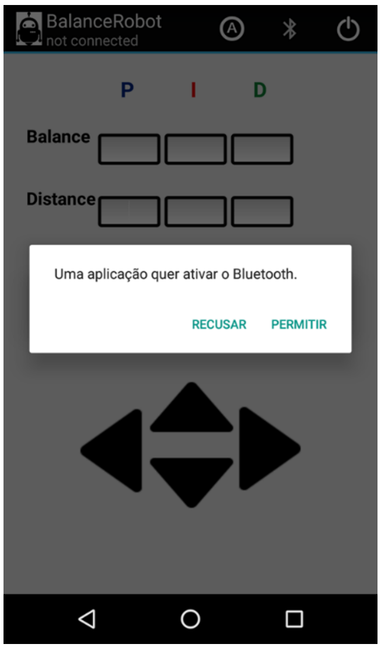


Figure 46 – Remote app Bluetooth request

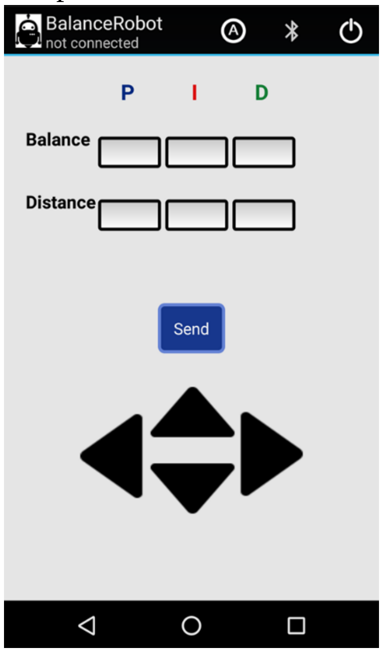


Figure 47 – Remote App layout

11.2.2. Connection to Local System

To do anything in the remote application it must be connected with the robot system, if there is no connection pressing any button will just result in a “Not connected” message. To establish the Bluetooth connection one must press the Bluetooth symbol in the upper bar, then a list of already paired devices will appear, though one can search for devices and pair it in the app as well. Our local system is denominated Tinyos in the device list.

After connected the application, the name of the local systems will appear in the upper bar, replacing the “not connected” text. Also, the application requests the current PID gain values from the local system, once received the data will be treated and displayed in the corresponding text boxes for each value.

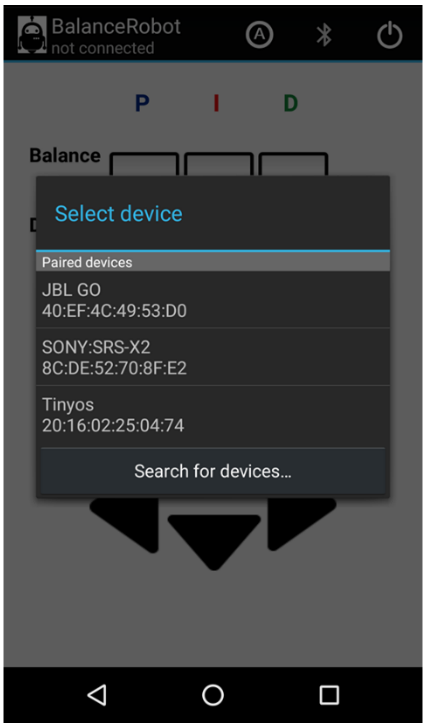


Figure 48 – Bluetooth devices list

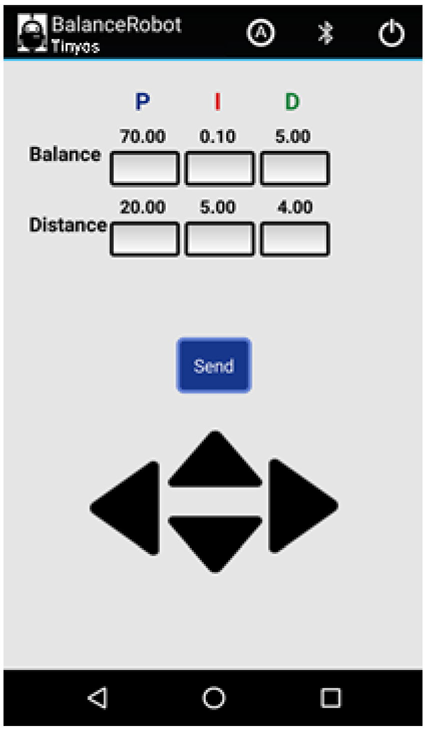


Figure 49 – Application layout after connection, with values of PID gains

11.2.3. Send and Receive Values

Using the input boxes the user can write the desired PID gain value, which can be a decimal value limited by 5 characters and with the maximum value of 200. even if the user inputs a number with more than 2 decimal places the application will only store up to 2.

The user can write one or more values and then press the send button to send it to the local system. The application will check which input boxes have values in them and send a command accordingly. If there are any values above the limit, it will not be added to the command and a message saying which value is it will appear. In the case that there is nothing in the input boxes a "Nothing to be sent message" will be displayed.

Once the values have been received and altered, the local system will send the current values, which the application will receive and display in the respective text boxes.

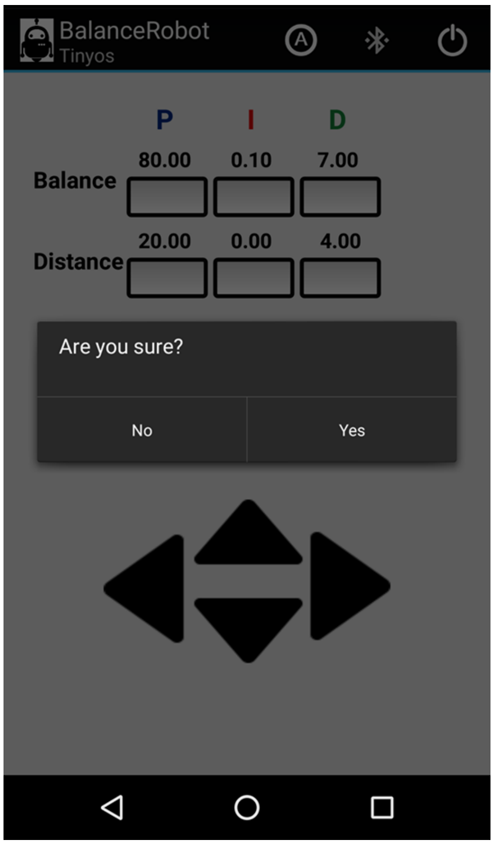
11.2.4. Arrow buttons

Each arrow button sends a different movement command when pressed, once released a "stop" command, signaling the end of the command, is sent. To prevent any errors on this procedure, every other button is disabled while an arrow button is pressed, so only one can be pressed at the same time.

11.2.5. Power and enable Auto-tuning

When the Power button is pressed, a warning box will appear, to ask if the user is sure on his decision, if "Yes" is pressed the command will be sent, otherwise the application will do nothing.

The Auto-tuning button sends a command when pressed and if sent successfully a "Automatic tuning requested" message will appear.



	P	I	D
Balance	80.00	0.10	7.00
Distance	20.00	0.00	4.00

Are you sure?

Figure 50 – Power button pressed

12. Final Prototype

The final prototype of the function is shown in the figure below with all the components assembled.

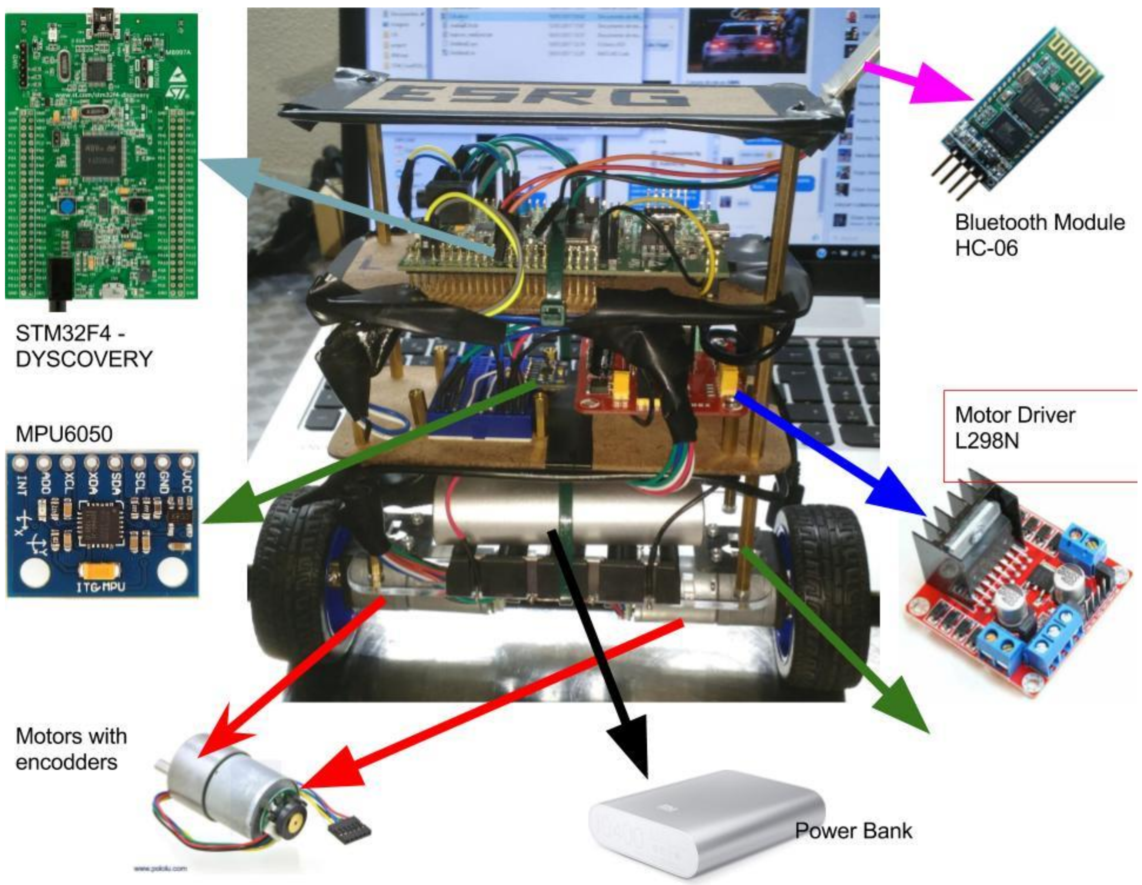


Figure 51: Final Prototype

13. Conclusion

13.1. Issues during the development of the project

During the development of the project, in the implementation stage we had an issue with the Robot's motors that even without actuating in the motors they could move by applying a small amount of force using the hands what means that even if the robot is in the setpoint position the robot will tend to fall to one direction because the gravity force on the robot is enough to make the wheels to move. Because of that problem the Robot can't stay still in the setpoint position but always oscillating around the setpoint position.

Another problem regarding the motors was the position of the Encoders sensors, the motors used in this project had the encoders built-in and as professor José Mendes explained the group the position of the encoders sensors is great to measure the motors velocity but not very good to measure how much the wheel has traveled because the encoder is mounted on the motors and not all movement in the motors generate movement in the wheels and also not all the movement in the wheels is generated from the motors, a better solution would be use another encoder that measure the wheels movement.

13.2. Conclusion and future prospective

All the milestones proposed in the analysis were archived with success, the Robot was able to balance itself and move according to the commands received from the Android Application. The hardware selection was good because other that the motors the rest of the hardware selected had a good performance.

We were able to implement two different Auto-Tuning algorithms, the first one the twiddle that is always checking how the errors is changing using that increase or decrease the gains, we tried to use other algorithms more precise but with higher performance comes more complexity. We tried the Ziegler–Nichols method that is a good one form this project since the output is oscillating, the method was implemented but the group didn't had enough time to attempt to use it in the project and that's one of the future prospective, implement more sophisticated Auto-Tuning methods to tune the PID Controller.

Overall the implementation of this project was really important because the group gain a lot of new knowledge about new concepts like Using for the first time a real-time operating system (FreeRTOS), work with a new and powerful board (STM32F4) and the best of all it's learning the correct way of implement a software/hardware.

The Video of the project can be seen in the link below:

https://www.youtube.com/watch?v=ib41xwgPFGg&t=3s

14. References

MathWorks. 2012. Control Tutorials for MATLAB and SIMULINK, Inverted Pendulum. Available at:
http://ctms.engin.umich.edu/CTMS/index.php.exampleInverted Pendulumsection-SystemModeling.

Jin, D. 2015. Development of a Stable Control System for a Segway. Available at: http://www.raysforexcellence.se/wp-content/uploads/2013/01/Dennis-Jin-Developmentof-a-stable-control-system-for-a-segway.pdf.

Pratical Approuch to Kalman Filter.
http://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/

Kalman Filter
https://en.wikipedia.org/wiki/Kalman_filter

STM32F4 Datasheet
http://www.st.com/content/ccc/resource/technical/document/user_manual/70/fe/4a/3f/e7/e1/4f/7d/DM00039084.pdf/files/DM00039084.pdf/jcr:content/translations/en.DM00039084.pdf

An, Wei, and Yangmin Li. "Simulation and Control of a Two wheeled Self-balancing Robot." Robotics and Biomimetics (ROBIO), 2013 IEEE International Conference on. IEEE

MIP Balancing Robot.
from http://www.wowwee.com/mip.