

POLITECNICO DI TORINO

Collegio di Ingegneria Elettronica, delle Telecomunicazioni e Fisica (ETF)

Corso di Laurea Magistrale in Electronic Engineering

Master Thesis

**Development of IoT Low-Cost Embedded
Board to Enable Autonomous Navigation on
a Cloud Robotics Platform**



Supervisors

Prof. Bona Basilio

Prof. Chiaberge Marcello

Author

Orso Eric

Supervisors at HotBlack Robotics

Ph.D. Ludovico Orlando Russo

Ph.D. Gabriele Ermacora

April 2018

Abstract

Designers who engage in robotics are called Robot Makers. Robotics is an interdisciplinary field, requiring robot makers to have knowledge of mechanics, electronics and information technology, meaning that robotics has a significant entry barrier.

Robots are usually designed with an application in mind, but Robot makers often have to wrestle with the same hardware, interface and software configuration issues over and over. Robot Makers would rather focus on the final application.

The aim of this thesis is to provide an hardware configuration and a software framework that streamline and simplify the design of a robotic platform. This is achieved by combining the custom board developed in this thesis with a Raspberry Pi and with the HotBlack Software Framework.

The custom board provides features that are highly desirable to robot makers. It is cheap, easy to mount and easy to integrate. It powers the robot from a single power source. It allows for a large variety of different power sources to be used. It provides protections against common mishaps. It natively support two DC motors with encoders and four servomotors. It can be expanded through the use of Arduino Shields. It is compatible with the Arduino Software Framework.

There are countless Arduino Shields, providing all sort of functionality for which a large amount of open source code exists. This allow robot makers to easily expand the functionality of their robot.

The HotBlack Software Framework consists of a remote cloud infrastructure and a modified Linux OS Figure with ROS installed. The Figure is loaded on a SD card and works on a Raspberry Pi 3B. The user programs the robot through a cloud interface designed to easily share and import code. This framework allows the user to get a robotic system up and running in a matter of minutes.

ROS is itself a framework that provide abstraction layers and many functionalities aimed at robot makers, like software for navigation, mapping, vision, and more.

The hope is for the hardware and software framework presented in this thesis to become widespread and lower the entry barrier of robotics itself, allowing for an ever growing user base to develop novel robotic applications, forming a virtuous cycle.

Acknowledgements

To my family. Because you always believed in me and supported me through the years. Because I love you. To my Mother. Because you take on the day with a smile. To my Father. Because you help me out when building devices.

To the Guasco Brothers. Because it is easy to be friends when everything is going smoothly. It's when things get rough that the bond of friendship is truly tested. Over the years, many friends came and went, but you stayed, believing in our projects. To an everlasting friendship.

To Andrea Guasco, Because of your leadership and your vision. Your fated call many years back gave me the opportunity to walk the path of robotics. A path we never strayed from ever since.

To Matteo Guasco. Because of your inventiveness and your hunger. Because you never give up and keep trying, learning and adapting.

To late professor Pierluigi Civera. Because your brilliance, experience and insight in so many diverse fields of knowledge were awe inspiring. Because you always believed in our projects and gave us the means to progress. I will always remember you fondly. May you rest in peace.

To HotBlack Robotics. Because you are a Startup with a great idea and with the means to pull it off. May this thesis help you in your endeavours. You will have at least one user for your system. Me.

To Ludovico Russo. Because you are one of the smartest software engineer I ever had the pleasure to meet. Because of your patience and your method. One wonders how many more will engage in robotics because of you.

To Gabriele Emarcora. Because of the vision you have for your company. Because of a project involving robotic street workers.

To Marco Cipriani. Because of your vision for Arol. Because of your many experiences. and because you bet on me when scaling up your team.

To professor Basilio Bona. Because of your patience. Enjoy your retirement, you earned it.

To professor Dario Demarchi. Because you picked up Civera's mantle and support us to this very day.

To professor Marco Mazza and the research team at the EPFL. Because of the opportunity you gave me in Switzerland.

To many professors at the polytechnic of Turin and other universities. Because of knowledge you shared.

To Gianpaolo Macario's smarthphone. Because, as it turns out, mobile connectivity truly has its uses!

To Dario Trimarchi. Because of your skills and advices. Because your warnings where founded indeed.

To my current company, Arol. Because the place is nice. Because you feed me.

To my coworkers at Arol. Because your experience is allowing me to grow. I'm lucky, because you can't choose who your co-workers are.

To the many people that helped me over the years, one way or the other. There are too many names to fit, so, if you are reading this sentence and believe your name should be in the list, it is in the list after all! Just execute this C++ snippet in your mind and there you have it:

```
cout << "Thanks to " << HelperList.GetName(Thesis.GetReaderId()) << " for helping.\n";
```

To the human civilization, and the wonders we achieved. Because the tools and knowledge our ancestors amassed through the millennia with sweat and blood are what made this thesis possible in the first place.

With us, evolution made a bet on brain size and opposable thumbs. How far have we come by making full use of those gifts. I can see greatness ahead for humanity. There is no doubt in my mind that strong AIs and robotics will be the foundation for the shining runway that will lead us to the stars. I hope my humble contribution will bring the stars a little closer.

Table of Contents

1. Introduction.....	1
1.1 Cloud Robotics.....	1
1.2 ROS Robot operating System.....	2
1.3 Arduino.....	2
1.4 HotBlack Robotics.....	3
1.5 Objectives.....	3
2. Research.....	5
2.1 Small Mobile Robotic Platforms.....	5
2.2 Problems Encountered by Robot Makers.....	7
2.3 Existing Raspberry Pi Shields.....	7
2.4 Conclusions.....	9
3. Specifications.....	10
3.1 Design Philosophy.....	10
3.2 Pareto Principle.....	10
3.3 Dropped Features.....	11
3.4 High Level Architecture: Cloud Architecture	12
3.5 High Level Architecture: Custom Board Architecture.....	13
3.6 SBC - Single Board Computer.....	13
3.7 Power System.....	15
3.8 Arduino Shield Connector.....	15
3.9 Form Factor, Design Rules.....	16
3.10 Motors.....	16
3.11 Encoders.....	16
3.12 Recap.....	17
4. Architecture.....	18
4.1 Power System.....	18
4.2 Protections.....	20
4.3 DC Motors and Encoders.....	21
4.4 Microcontroller.....	22
4.4.1 Architecture: Microcontroller Pin Assignment.....	22
4.4.2 Analog to Digital Converter ADC.....	23
4.4.3 PWM Generators.....	23
4.4.4 Atmel In-Circuit Programming (ISP) and Serial Peripheral Interface (SPI).....	24
4.4.5 Universal Asynchronous Receiver Transmitter interface (UART).....	25
4.4.6 Encoders.....	25
4.4.8 Servomotors and Generic Digital I/O pins.....	26
5. Test Boards.....	27
5.1 Design of the Power Regulator Test Board.....	27
5.1.1 Choice of the SEPIC Controller.....	27
5.1.2 Design of the SEPIC Regulator.....	28
5.1.3 Linear Regulators.....	30
5.1.4 Supply Board Layout.....	32
5.1.5 Supply Board BOM.....	34
5.2 Design H-Bridge Board.....	35
5.2.1 Component Choice: H-Bridge.....	35
5.2.2 Mini Arduino Shield.....	36
5.2.4 H-Bridge Test Board Design.....	37
5.2.5 H-Bridge Test Board Layout.....	38
5.2.6 H-Bridge Test Board BOM.....	40

6. Robotic Platform.....	41
6.1 Seeker of Ways.....	41
6.1.1 Seeker of Ways: Architecture.....	43
6.1.2 Motor Controller.....	44
6.1.3 Rosserial Slave.....	47
6.1.4 Differencies between SoW and the custom shield.....	48
6.2 Software and Firmware.....	49
6.2.1 Firmware Architecture.....	49
6.2.2 Rosserial Driver.....	50
6.2.3 ROS Topics.....	55
6.2.4 Custom Board Messages.....	57
6.3 HotBlack Software Framework.....	65
7. Test Boards.....	69
7.1 Supply Board: Testing.....	69
7.2 Test H-Bridge Board.....	72
8. Custom Raspberry Pi Shield Design.....	75
8.1 Custom Shield Schematics.....	75
8.1.1 Battery connector and Protections.....	76
8.1.2 SEPIC Regulator.....	78
8.1.2 Linear Regulators and Filters.....	79
8.1.3 H-Bridges and DC Motors connectors.....	80
8.1.4 Microcontroller Services.....	81
8.1.5 Microcontroller.....	82
8.1.6 Raspberry Pi GPIO.....	83
8.1.7 Arduino Shield Connector.....	84
8.1.8 Servomotors.....	85
8.1.9 Encoders.....	86
8.2 Layout and Routing.....	88
8.2.1 Form Factor.....	88
8.2.2 Components Placing.....	89
8.2.3 Layout.....	90
8.2.4 Gerber and BOM.....	92
8.3 Shield Assembly.....	95
9. Applications.....	97
9.1 Motor Testing.....	97
9.2 Camera Image Processing.....	100
9.3 FPV First Person View Control.....	102
10. Conclusions.....	111
10.1 Future Development.....	111
10.2 Example Applications.....	112
10.2.1 Autonomous Navigation.....	112
10.2.2 Drones.....	112
10.2.3 UAV.....	112
10.2.4 Remote IoT Node.....	112
10.3 Final Remarks.....	113

1. Introduction

Robotics is an interdisciplinary field with a significant entry barrier. Robot makers must possess knowledge in mechanics, electronics and information technology in order to successfully build a robot that performs a task.

Building a robot requires:

- A mechanical structure
- A power source, a power regulator and a charger
- Motors and motor controllers
- Sensors
- Intelligence
- An interface and/or a data link
- The application software

Robots are often created with a specific application in mind. More often than not, the added value of a robotic application comes from the way the robot uses sensor data to interact with the environment and the way the robot answers to user commands.

Robot makers often have to solve the same hardware, interface and software configuration issues over and over. They would rather focus on developing the application.

The aim of this thesis is to design a custom board that handles common hardware related issues routinely faced by robot makers. The custom board will provide quality of life improvements and features that are desirable to robot makers, and will be easy to use.

The custom board is meant to be paired with a powerful software framework that streamline and simplify the configuration and the programming of the robot.

The software framework chosen put a strong emphasis on code sharing and code reuse, so that the robot makers only have to write code directly involved with the application.

1.1 Cloud Robotics

A mobile robotic platforms can be limited in terms of memory, data storage and processing power.

IoT, Internet of Things, is a paradigm in which devices can be individually accessible from everywhere in the world through a cheap internet connection.

The Cloud is an IT infrastructure featuring remote servers accessible through the internet. Those servers can store data and execute computationally intensive applications. Tasks that would be slow or outright impossible to run locally.

Cloud Robotics is a paradigm that combines Cloud technology and IoT. It allows workload to be offloaded from the robotic platform to remote servers through an internet connection.

Cloud Robotics is attractive to robot makers as it can allow a robot to perform tasks far in excess of the specifications of the local hardware.

The Cloud can be used to access large amount of data, like images and maps. It can be used to offload processor heavy tasks, like Figure recognition. It allows for remote control o the platform and supervision of entire fleets of robots.

The most significant drawback of Cloud Robotics is the added latency of the communication link. A latency in excess of several hundreds of milliseconds will severely impair remote control the robot.

1.2 ROS Robot operating System

ROS [1] is a framework. It provides a message exchange mechanism, abstraction layers, cloud support and more. It promotes the reuse of code in robotics and distributed software architecture.

ROS is based on the concept of node. A node can be either a piece of hardware, like a camera or even a robot, or a piece of software, like an navigation algorithm. A node can be either local or remote. Nodes exchange ROS messages between each others. The map of all nodes is called Graph.

ROS natively supports distributed architectures in which some nodes are locals while others are running on a remote server.

ROS defines protocols like Rosserial [2], that defines the exchange of ROS messages over a serial interface to a board, like a motor controller.

Rosserial is implemented as a library for many microcontrollers. The custom board developed for this thesis will communicate with a ROS installed on the Linux operating system (OS) running on the Raspberry Pi [3] using a serial UART interface and the Rosserial protocol.

The custom board will act like a ROS node. It will support several ROS messages, like messages that will physically move motion axis or messages that control the I/O of the Arduino Shields.

1.3 Arduino

Arduino developed a cheap but flexible board, the Arduino Uno [4], and paired it with the Arduino IDE [5], a powerful framework that promote code sharing, code reuse and open source software.

User designed boards, the Arduino Shields, can used to expand the functionality (ex. GPS) of the base Arduino board. Sketches (Arduino IDE project) made by the community exists and allow users to quickly make use of the new hardware.

Arduino became the de-facto standard in the field of prototyping electronics thanks to this business model. There is a lot to be learned from their success case.

Success factors include:

- Cheap open source hardware
- Arduino Shields provide easy optional hardware expansion
- IDE that simplifies the programming phase
- IDE that allows for easy share and import of open source sketches
- Bootloader that simplifies the firmware uploading phase
- Humongous amount of open source code and projects for the platform
- A large community that produces large amount of sketches, code and shields

1.4 HotBlack Robotics

HotBlack Robotics [6] is a start-up based in the Polytechnic University of Turin. This thesis was developed closely with their collaboration. Their inside knowledge of what robot makers proved invaluable for understanding what robot makers would consider as a desirable features in the custom board.

The vision of HotBlack Robotics is to create a framework take advantage of cloud technologies to simplify the setup and programming phase of a robot and promote code sharing, code reuse and open source software. This lowers the entry barrier faced by would-be robot makers, allowing for more people to engage in robotics and contribute with their own solutions, in a virtuous cycle.

HotBlack has developed a clever software framework that simplify the setup phase and the programming of a Raspberry Pi based robot through the use of a custom Linux Figure with ROS installed and a cloud infrastructure. Details of the HotBlack Framework will be explored in greater details later.

A similar model was used to great effect by Arduino.

The HotBlack Framework takes advantage of the Raspberry Pi, the cheapest most popular ARM based SBC today. The Raspberry Pi is a remarkable device. It has many features that are highly desirable for a mobile robotic platform.

The Raspberry Pi is not perfect. The most significant shortcoming is the lack of native support for motors. At least one additional board (a motor controller) is required to build a robot out of a Raspberry Pi. The additional board has to be able to drive at least two motors in order to make the robot useful.

A short research showed there are many boards in the market that allows to do just that. Existing Raspberry Pi shields will be explored in greater details later.

Since an additional board is required anyway, there is an opportunity. The custom board could do more than just drive two motors. It could also handle problems routinely encountered by robot makers, while also providing quality of life improvements and add desirable features.

Learning from the Arduino success case, the custom board should be useful, cheap and simple to integrate for it to become widespread.

With greater adoption, robot makers that use this framework would be able to draw from an ever increasing pool of open source code shared by fellow robot makers. A virtuous cycle.

The hope is for the custom board developed in this thesis paired with the framework developed by HotBlack Robotics to become the de-facto standard for Raspberry Pi based robots.

1.5 Objectives

The objective for this thesis is to develop a custom board that interfaces with a Raspberry Pi and provides useful functionalities that would be desirable for robot makers.

A research was conducted on what boards are already available on the market, what problems robot makers had with them, and, if there was space for a custom board more attuned to the needs of robot makers. A list of existing boards will be explored in details later.

A research was conducted to find out what problems were commonly encountered by robot makers, and by extension, the features that robot makers would like to have on the custom board.

Small robots are popular amongst robot makers. They are cheap, convenient and allows for all kind of algorithms to be implemented and tested.

Small robots are intrinsically safe as their engines are comparatively weak and can't hurt people. Small robots have low hardware specifications and can take advantage the most from a cloud approach. The decision was taken to focus on small robots. Target form factor is 15cm.

A popular way to power a Raspberry Pi based robot is to use two batteries: A power bank and a separate battery for the motor controller. This approach invites for all kind of problems.

The decision was taken to have a flexible on-board power regulator that allows for both the engines and the Raspberry to be powered from the same power source. The decision was taken to allow for many popular power sources to be used, from single cell LIPO to AAA batteries.

Learning from the Arduino success case, the custom board should be cheap, flexible, easy to mount and easy to use. The board should be paired with a powerful framework that simplify the setup phase and the programming phase while encouraging code sharing and code reuse.

2. Research

A research was made to see what kind of robotic platforms are already present on the market, on what Raspberry Pi compatible boards already exists and on what problems are encountered by robot makers.

It's imperative to understand the state of the art before moving forward with the project, otherwise the risk of developing a solution for a problem that is not widespread is high.

2.1 Small Mobile Robotic Platforms

Robots comes in every shape and form. Robots perform as many diverse functions as the imagination, skills and resourcefulness of the robot maker allows. Robots range from tonnes heavy anthropomorphic arms to swarms of miniaturized insect like robots to autonomous flying military combat drones.



Figure 1: Robokind [7][8][9]

The objective of this thesis is to create a custom board and a software framework that make the life of robot maker easy. This requires a large community that share open source code. This in turn requires a wide adoption. The prime factors in adoption are:

- Usefulness
- Cost
- Ease of use

The kind of robots that best fit those factors are small mobile platforms. A market search shows that there are countless small mobile platform sold on the market.

Most of them features two DC motors with wheels for motion, a SBC Single Board Computer for intelligence, two power sources and some additional board.

Advancement in technology means that significant computational power is now available at low cost and low power consumption, making the prospect of a small cheap platform attractive.

SBC are still limited to this day and more demanding application that involve neural network and artificial vision are still out of reach. This is where the IoT component could make the difference, allowing to offload computational heavy tasks on remote servers. There is an opportunity.

Some of the small mobile robotic platforms considered in the research can be seen in Figure 2. They are cheap, small, and mobile. Most feature some additional sensors and board.

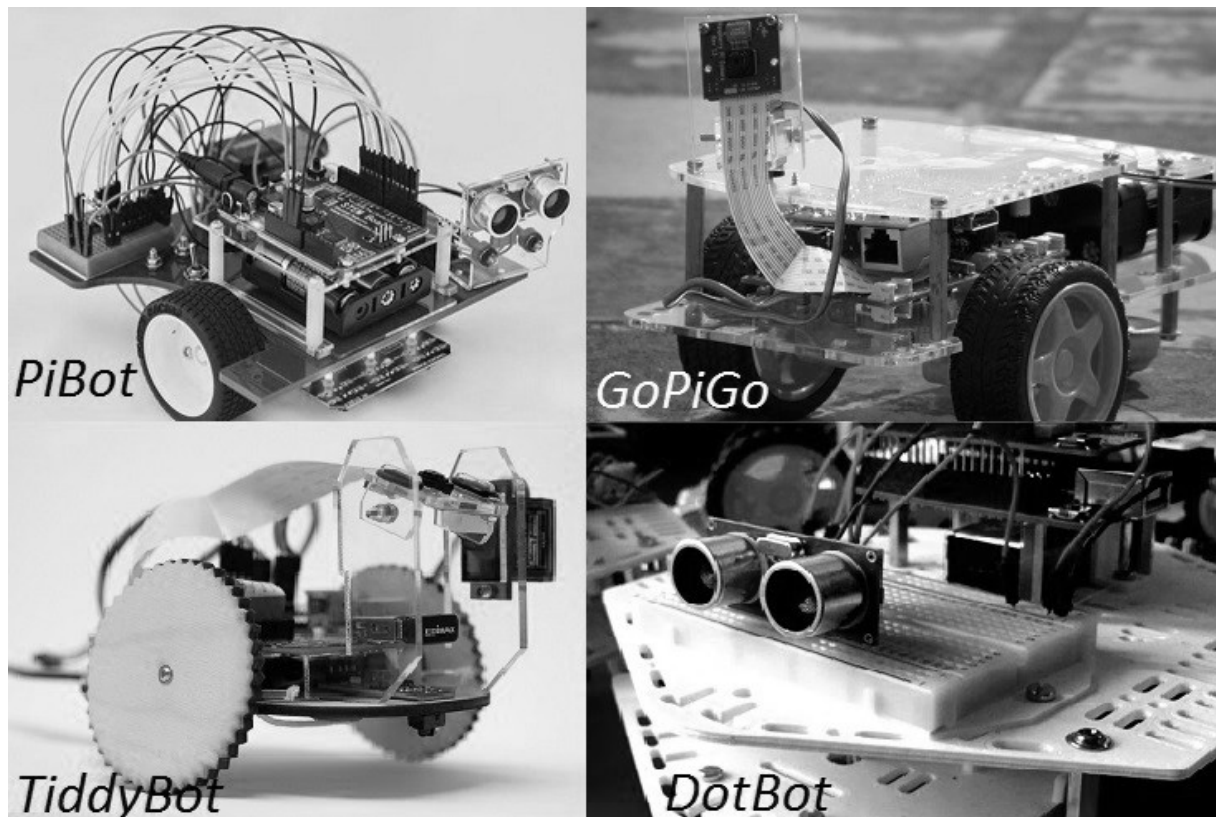


Figure 2: Small Mobile Robotic platforms based on Raspberry Pi [10][11][12]

Aereo-models and Quadcopters (Figure 3) can be considered small mobile platforms as well. Wheeled robots are convenient as they can sit on a table, flying drones on the other hand can be more fulfilling to work with. Ideally, this thesis should cater to robot makers from both worlds.



Figure 3: Flying Drones [13]

Common features of small mobile robotic platforms are:

- The most popular platforms are also the cheapest
- Most wheeled platforms use two small DC motors or servomotors to move around
- Servomotors are popular for manipulators. Flying drones overwhelmingly use servomotors for their control surfaces and brushless motors with a servo interface for the propellers.
- A wide range of sensors are used for navigation, from camera, to ultrasound, to infrared, to IMUs

- Wi-Fi is the most common remote data link used. Bluetooth comes second.
- A huge variety of power sources are used. AAA batteries, power banks, LIPO with varying number of cells and more
- Usually two power sources are used. A Power bank for the Raspberry and a battery for the motors

2.2 Problems Encountered by Robot Makers

A research has been conducted to discover the kind of problems robot makers routinely encounter. HotBlack Robotics has significant experience when it comes to such robots, their expertise proved invaluable in narrowing out the research.

On top of regular research, participation in events like the mini-maker faire [14], hackathons [15], workshops and more, allowed to see first hand what aspects of robotics proved more problematic to robot makers.

Common problems identified include:

- Using multiple power sources invites for often irreversible damage on the robot
- Noise caused by DC motors and servomotors induces bugs and instabilities in the Raspberry Pi. Something that can be very hard to debug and solve
- Voltage, current and capacity are all things robot makers must consider. Making the wrong choice can cause irreversible damage to the robot
- With so many Raspberry Pi operating systems, it's hard to choose the right one for the application
- The ROS framework is as useful as it is hard to learn and make use of
- Robot makers often have an Arduino Board and Arduino Shields. Using them with a Raspberry Pi is not trivial and involves wiring and programming both at firmware level and at software level inside the Raspberry
- Robot makers often spend time solving hardware related issues and software bugs. They would rather focus on the application
- The Raspberry Pi can become unstable when powered through the USB port in some load configurations. Powering Servos from the Raspberry Pi only exacerbates this problem
- Shields usually lack an hole to let the Raspicam flex cable through

There is a big opportunity for a framework that solves all the hardware and software problems identified in the place of the robot makers. The objective must be to make something useful, easy to use and cheap that allows robot makers to focus on the application.

2.3 Existing Raspberry Pi Shields

The first consideration to be done is that mobile robotic platforms require additional boards anyway to pilot the engine, so there is space for a custom board.

A quick research showed that there are countless Raspberry Pi shield on the market that perform a wide variety of functions:

- Shields that allows the raspberry to pilot DC motors, brushless motors, and more.
- Shields that allow for an Arduino Shield to be used by the Raspberry Pi.
- Shields that add a micro controller and some sensors.
- Shields that allow the Raspberry to be powered through a buck regulator.
- No Raspberry Shield was found that powered the Raspberry from below it's nominal operating voltage of 5V

While shields that handles one or two of the problems identified are plentiful, no shield that handled all of them could be found. It was the impossibility to find such hardware that prompted the development of this thesis with HotBlack Robotics in the first place.

Some of the Raspberry Pi Shields considered in the research can be seen in Figure 4.

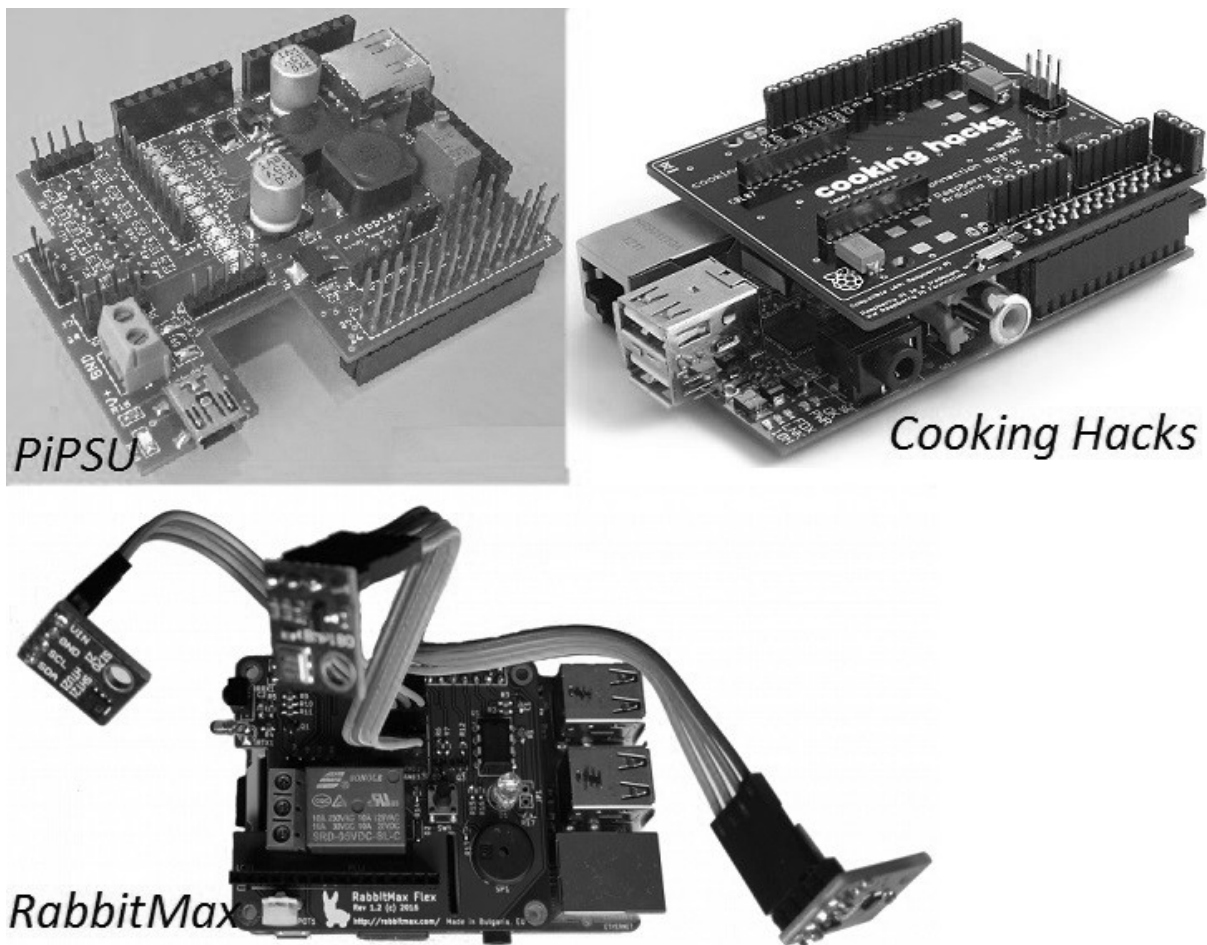


Figure 4: Raspberry Pi Shields [16] [17] [18]

2.4 Conclusions

To be desirable to robot makers, the custom board should have the following features:

- Low cost
- Easy to assembly, easy to integrate, easy to use
- Support at least two DC motors
- Support servomotors
- Power the whole robot from a single power source
- Allow a wide range of input voltages
- Keep the Raspberry Pi stable regardless of noise generated by the motors and power source used
- Protect robot from common mishaps like inverting power cables
- Have an hole to let the Raspicam flex cable through
- Support Arduino Shields

A custom board with the proposed specifications would be highly desirable to a robot maker. If such board were to be combined with a software framework that provides IoT functionalities like remote computing or a simplified online IDE and allow for easy sharing and integration of open source code, widespread adoption of this solution becomes a real possibility.

3. Specifications

Drafting the specifications is probably the most important part of any project. Significant effort has been put into providing only useful and/or cheap features in order to maximize the usefulness of the board to robot makers.

3.1 Design Philosophy

An open source platform is successful only when there is a thriving community sharing and reusing code for it. Adoption is driven by three main parameters:

- Cost
- Ease of use
- Usefulness

Taking into account the main parameters to optimize for, the overall principles that guided the drafting of the specifications are as follow:

- Design a custom board that interfaces with the Raspberry Pi
- The custom board interfaces with the HotBlack framework
- Configuration and setup of the system should be quick and easy
- The HotBlack Framework should promote code sharing and code reuse
- The HotBlack Framework should simplify the programming phase
- The custom board and the HotBlack Framework should solve problems, not create more
- The custom board should be cheap, easy to mount and easy to use
- The custom board should provide features that are desirable to robot makers
- The custom board should give quality of life improvements
- The custom board should be protected against common mishaps and mistakes on the part of the robot maker
- The custom board should be compatible with the popular power sources
- The custom board should be compatible with popular open source software
- The custom board should be able to control at least two motion axis
- The custom board should support popular motors

3.2 Pareto Principle

The Pareto Principle is an empirical observation that emerges from many different domains in different ways. It basically states that the majority of an observed output quantity will be the result of a minority of an observed input quantity.

When applied to a design, it can be stated as follow:

- A solution that covers 100% of the cases will cost 100% of the resources.
- A Pareto Solution will cover 80% of the cases at the cost 20% of the resources.

- A Pareto Solution might not exist in a given domain.

It is an empirical observation, not a law and not always a Pareto Solution will exist. For example, in a specific field a solution exists that costs 90% of the resources but covers only 10% of the cases, making it virtually useless.

Following the spirit of the Pareto Principle, the specifications will be drawn in a way to maximize the user cases covered by the overall solution while minimizing overall costs, thus maximizing usefulness and desirability.

3.3 Dropped Features

Features that would rarely be used, or that would be overly costly, be it in terms of PCB area, component cost, processing power or power requirements have been dropped in favor of features that are cheap to implement and would be desirable for most robot makers.

This is a list of features that were considered, but dropped:

- No On-board battery charger. Extremely useful but overly complex since the board is to be compatible with a wide range of different power sources.
- No IMU. Fairly useful, but not useful enough. User that want an IMU can plug in an Arduino Shield with their favorite IMU.
- No native support for brushless or stepper motors. The cheapest most commonly used motors are Servomotors and DC motors. Arduino Shields can be used to support other types of motor.
- No native support for more than four servomotors. Having four servos allow for quad-copters, already. Any more would result in diminishing returns. Arduino Shields can be used to support more servomotors.
- No standalone I2C and UART connectors. Useful and cheap. But not useful and cheap enough. The pins can be taken from the Arduino Shield connector by robot makers who needs them at only a slightly higher cost, while lowering the base cost for everyone.
- Optic Flow sensor. Useful but costly and clunky.
- Radio (Bluetooth, Wi-Fi, etc...). Useful but costly. Wi-Fi is already inside the Raspberry Pi Model 3.

3.4 High Level Architecture: Cloud Architecture

The overall architecture of the system can be seen in Figure 5.

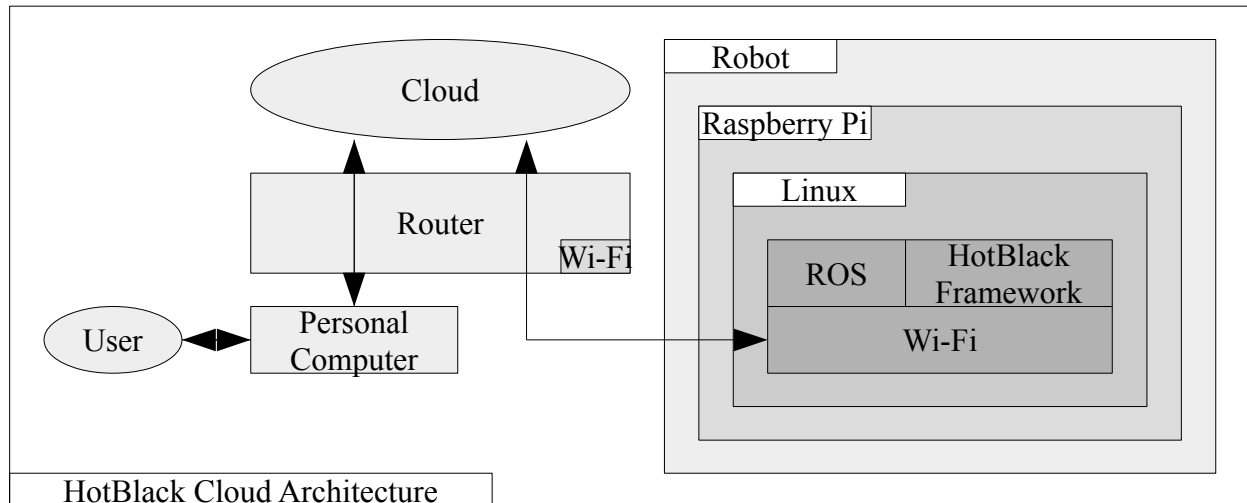


Figure 5: HotBlack Cloud Architecture

The user needs the following pieces of hardware to get started:

- A computer
- A router and an internet connection
- A Raspberry Pi
- An SD card with the HotBlack HBrain [19] image installed

User and robot do not directly communicate with each others, instead, the user communicate with a remote server that handles the IDE, and the server itself take care of all the intricacies involved in configuring and controlling the robot. This architecture has several advantages:

- The user only need a basic browser. The HotBlack IDE on the remote server provides every tool needed to program and control the robot
- Setup of the system can be done in a matter of minutes with the following steps:
 - Configuration of the router. A default wi-fi network name is required for the robot to connect the first time
 - Power up the Raspberry Pi
 - Register on the HotBlack website and accessing HotBlack robotics web page on a common browser

Once those steps are taken care, the user can have the system ready and an example project up and running in only a few minutes after power up by following the tutorial on the site.

3.5 High Level Architecture: Custom Board Architecture

The architecture of the hardware of the custom board can be seen in Figure 6.

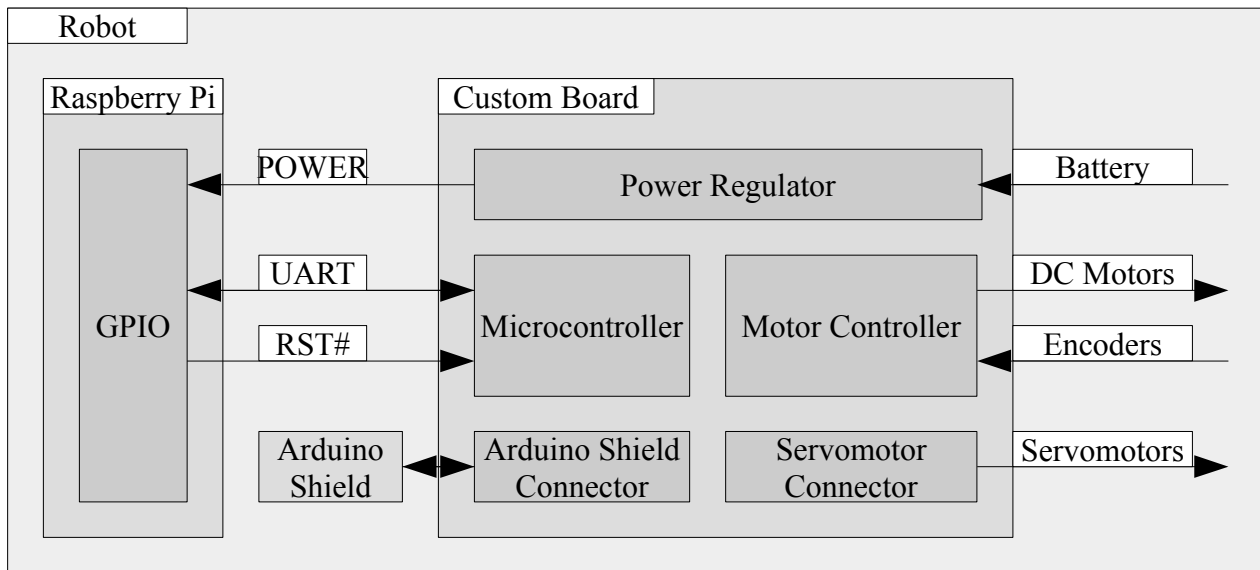


Figure 6: Custom Board Architecture

The board features:

- A voltage regulator that powers the whole robot from a single power source
- Communication with the Raspberry Pi through the UART interface
 - The protocol used is the Rosserial, defined by ROS.
- Programming of the microcontroller through the UART interface
 - The bootloader used is the Arduino Bootloader
 - The board is compatible with the Arduino Framework and can be programmed through the Arduino IDE.
- The board features an Arduino Shield connector for optional hardware expansion
- DC Motor controller for up to two DC motors with encoders
- Four Servomotor connectors

3.6 SBC - Single Board Computer

There are any number of ways to give the robot some intelligence. The Raspberry Pi was chosen because it has several remarkable features that makes it highly desirable for robot makers:

- It is the cheapest most popular SBC available on the market
- It is reliable
- It is powerful
- It requires little power to work. About 10W

There are many other ARM based SBC on the market. Alternatives that were considered include:

- Arduino YUN [20]

- Marsboard [21]
- UDOO [22]
- BeagleBone Black [23]

The Raspberry Pi stood out because of its cost and its popularity, which are key metrics for maximizing adoption.

The development of a custom board with an ARM microcontroller on board was considered, but was quickly dropped because of the scale economy involved. It is cheaper to make a custom board that interfaces with an existing SBC than making a custom SBC. On top of that, the disadvantage of not having a large community developing custom Linux distributions for the custom SBC cannot be overstated.

The Raspberry Pi is not perfect, and does have a few drawbacks:

- The ARM architecture lacks of the X86 instruction set. It limits the choice for the OS
- The system is unstable under some power and load configurations
- It can become fairly hot in certain load configurations

There are several models of Raspberry Pi

- Raspberry Pi
- Raspberry Pi 2. More powerful. More interfaces, (USB, Camera flat and HDMI).
- Raspberry Pi model 3B. On-Board Wi-Fi radio. More powerful 64bit arm architecture.

Raspberry Pi 2 and Raspberry Pi model 3B have the same board layout. The HotBlack Framework is compatible with both of them.

The Raspberry Pi Model 3 is advised as platform because the additional power consumption and cost is worth the wireless connectivity and the additional computational power. Raspberry Pi 2 is more attractive for systems that do not need wireless connectivity.

3.7 Power System

Powering up a mobile robot is not trivial. It is a problem that every robot maker has to face. A research conducted on Raspberry Pi based robots shows that the most common configuration is to have two separate power sources:

- A USB power bank that powers the Raspberry Pi itself
- An additional power source, usually a number of AAA batteries, to power a motor controller, connected through wires to the GPIO.

The power system offers many opportunities to simplify the robot, by removing batteries and wires while providing protections and quality of life features.

Specifications for the power system:

- The custom board will enable the robot to be powered by single power source
- The custom board will be compatible with variety of popular power sources
 - Input Voltage Range: 3.5[V] ~ 15 [V]
 - Single cell LIPO: 3.5[V] discharged ~ 4.2[V] overcharged
 - Two cells LIPO: 7.0[V] discharged ~ 8.4[V] overcharged
 - Three cells LIPO: 10.5[V] discharged ~ 12.6[V] overcharged
 - NiMH AAA battery: 1.1V per cell
 - Alkaline AAA battery: 1.5V per cell
 - USB power bank: 5V
 - Portable power supply: 3.5[V] ~ 15 [V]
- The custom board will provide protections against common mishaps, like connecting the wires in the opposite polarity or protecting the raspberry against noise generated by motors

The power source must be able to provide enough power for both the Raspberry Pi and the motors. The Raspberry Pi 3 Model B alone draws about 10[W]. No amount of electronics can output more steady state power than what a power source can provide.

3.8 Arduino Shield Connector

An Arduino Shield is a board that can be mounted on top of an Arduino board and adds some kind of hardware functionality to the system. Shields by design are easy to mount and cheap to produce.

There exists Arduino shields that adds all kind of functionality. From GPS receivers to Geiger detectors. From IR sensors to loudspeakers. Sketches often exists that allow users to quickly make take advantage of the new hardware.

There is an opportunity. The custom board can be designed to be compatible with Arduino Shields. This will allow robot makers to draw from a large pool of existing desirable robot hardware in the form of Arduino Shields.

There is another opportunity. The board can be made compatible not only with Arduino hardware, but also with the Arduino Framework, allowing the robot maker to also draw from a large pool of existing software written for Arduino and Arduino Shields.

3.9 Form Factor, Design Rules

The custom board will be a shield for the Raspberry Pi 2 or the Raspberry Pi 3. It will have about the same physical dimensions of the Raspberry Pi, it will be mounted on top of it and connects with its GPIO.

The board will be easy to mount, it will have just two levels of interconnect and will use a design rule suitable for a TQFP packages, the package of the microcontroller. Such design rule makes the design of the custom board harder, but lowers the production costs.

3.10 Motors

There are many types of motors. Adding support for all of them would only increase the complexity of the board. The custom board will focus on providing support for the most popular motors used in small robotic application. Servomotors and DC Motors.

Since the custom board is compatible with Arduino Shields, it can be optionally expanded to drive potentially any kind of motor.

In order to be useful, a robot needs at least two axis. The custom board will feature at least two axis of DC motors and two axis of servomotors. A reasonable power limit for a small robotic platform is 3[A]. Often much smaller DC motors are used 500 [mA], meaning that the drives will not overheat in a typical use case.

The servomotors are somewhat cheaper to support. The footprint of their connector is small and has a scalable layout. Four axis of servomotors allows for the board to be used as controller for quad-copters and UAV, which is the reason four was chosen as the number of axis of servomotors to be supported.

Motor specifications so far:

- Four axis of Servomotors. 5V
- Two axis of DC Motors. 3A each

3.11 Encoders

Having two encoder connectors enables closed loop position and speed control on the DC motors. Having encoders also enables odometry, the ability to estimate the current position from the angular position of the wheels. Both are staple features used in robotics.

3.12 Recap

The specifications of the custom board so far:

- The custom board will be designed as a Raspberry Pi 2/3B Shield
- The custom board will be cheap, easy to mount and easy to use
- The custom board will be mounted on top of the Raspberry Pi and connect to its GPIO
- The custom board will allow for a single power source to powers the whole robot
- The custom board will be compatible with many popular power sources
- The custom board will feature protections against common mishaps, like inverting the power wires
- The custom board will feature an Arduino Shield connector and will be compatible with Arduino Shields
- The custom board will support the following protocols:
 - The Arduino Bootloader
 - The Rosserial protocol
- The custom board will be compatible with the following frameworks:
 - The Arduino Framework
 - The HotBlack Framework
 - ROS
- The custom board will be able to control four axis of servomotors natively
- The custom board will support two axis of DC motors natively
- Input Voltage range: 3.5[V] ~ 15.0[V]

Limitations:

- The power source must be able to provide whatever power the system requires
- The Raspberry Pi itself draws about 10[W]
- The DC motors will be powered directly from the input voltage. 3[A] maximum
- The total current used by the Raspberry, the Arduino Shield, and all Servomotors combined cannot exceed 5[A]

4. Architecture

Having drawn the specifications, it's time to delve deeper into the architectural design of the individual blocks of the custom board.

4.1 Power System

The first stage in the power system must be an efficient regulator that take whatever voltage is in input and generate an intermediate voltage. After careful considerations, it was chosen to use the following specifications for the power regulator:

- Input Voltage: 3.5[V]~15[V]
- Output Voltage: 5.7[V] 5 [A]

Considerations include the drop-out needed by the linear regulators, the voltage drop at higher current, the input voltage range for servomotors and the safety margin needed for the Raspberry Pi to remain stable in high load configurations.

Since this regulator needs to be efficient, the only solution is to use a switching regulator. The output voltage can be either greater or lower than the input voltage, this limit the switching topology that can be used. Possible switching regulator topologies are:

- Buck-Boost. Inverting Regulator.
- Flyback. Requires a transformer and a more complex feedback network.
- SEPIC (Single Ended Primary Inductor Converter). Requires two inductors.

The Buck-Boost has an inverted output, which is fine for a stand alone board, but causes all sort of problems when the robot makers wishes to use the battery for something else. Making things harder for the user would go against the design philosophy of the board. The Buck-boost is not suitable.

The Flyback is the optimal choice when an insulation barrier is required, as insulation comes for free thanks to the transformer used. In this case, insulation is not needed, and the additional component required to close the feedback loop can be seen as an overhead.

The topology of choice for this board is the SEPIC topology (Figure 7).

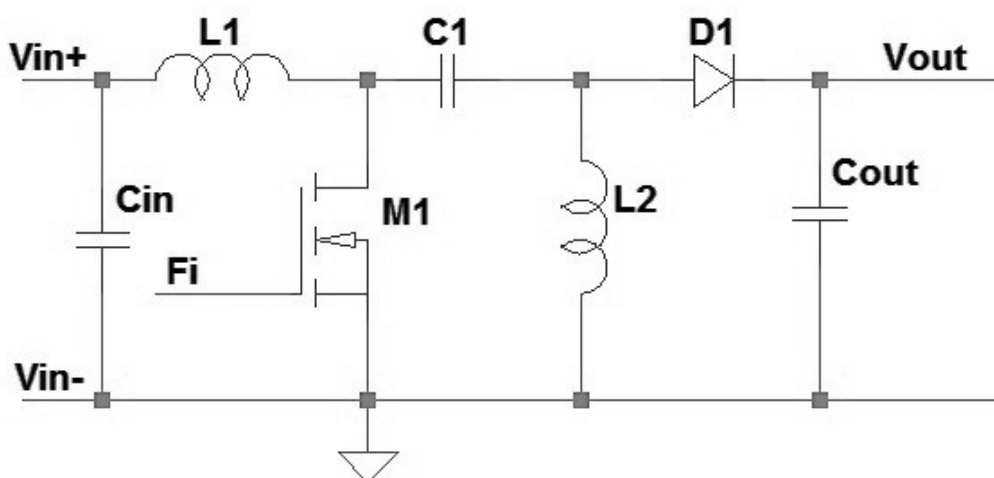


Figure 7: SEPIC Topology

In a SEPIC regulator, the energy is transferred from input to output through a capacitor (C1). Two inductors are required for the regulation (L1 and L2). A single low side switch S1 is used. A diode suffice as second switch D1.

The intermediate voltage 5.7[V] 5[A] is used by the other peripherals.

The Raspberry Pi need a clean power line. A LDO is used to clean up the 5.7[V] line and generate the 5[V] 2[A] required.

The Arduino Shield, the microcontroller and the encoders require another clean 5[V] line. Another smaller LDO generates their supply line.

Servomotors can make use of the intermediate voltage directly. Robot Makers will enjoy an higher torque from the servomotor with no drawback other than greater power dissipated.

Powering the DC Motors is trickier. When the input voltage is greater than the intermediate voltage ($>5.7[V]$), it's convenient to power the DC motors directly from the power source. When the input voltage is lower than the intermediate voltage ($<5.7[V]$) it would be convenient to use the intermediate voltage, but that would vastly increase the required power rating of the switching regulator and finding an H-Bridge which can work from 5.7[V] to 15[V] would be a challeng in itself.

A compromise was made. The decision was taken to power the DC motors directly from the input voltage. After careful consideration, the architecture of the block schematics of power system can be summarized as in Figure 8.

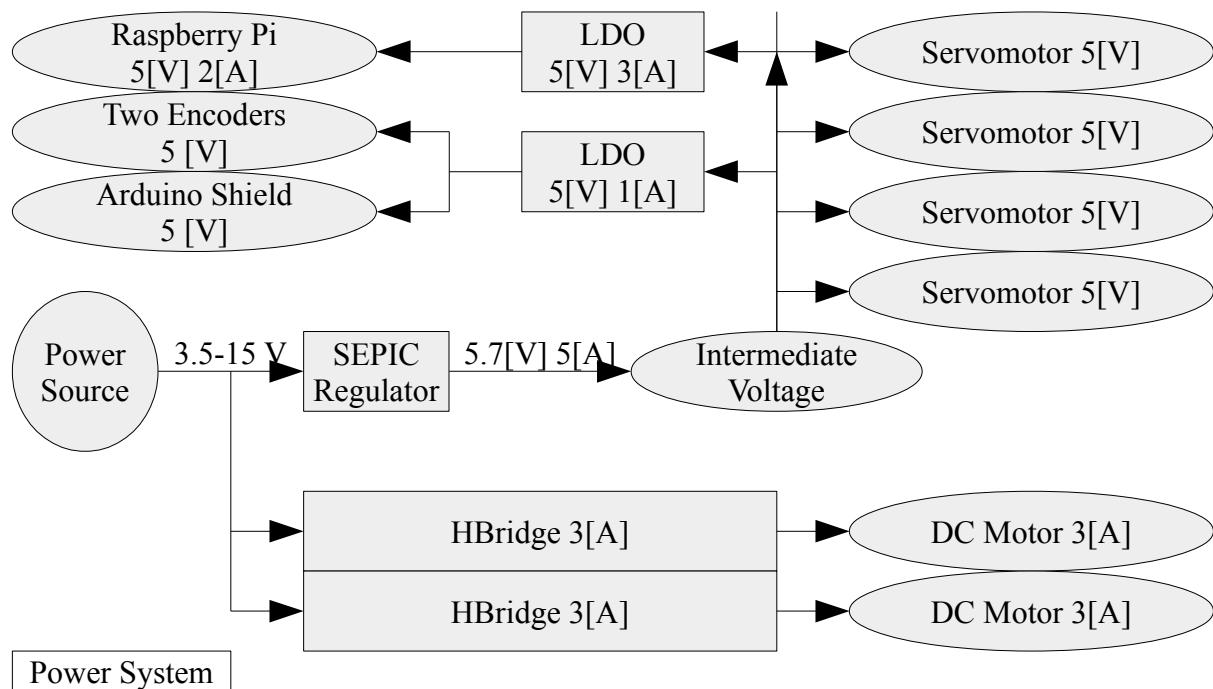


Figure 8: Architecture of the power system

4.2 Protections

The aim is to handle common mishaps on the part of the robot maker. Placing reasonable protections on the power system is not only a good practice, but goes a long way in helping out robot makers who may be inexperienced with safe electronic practices.

The overall architecture of the protections is shown in Figure 9.

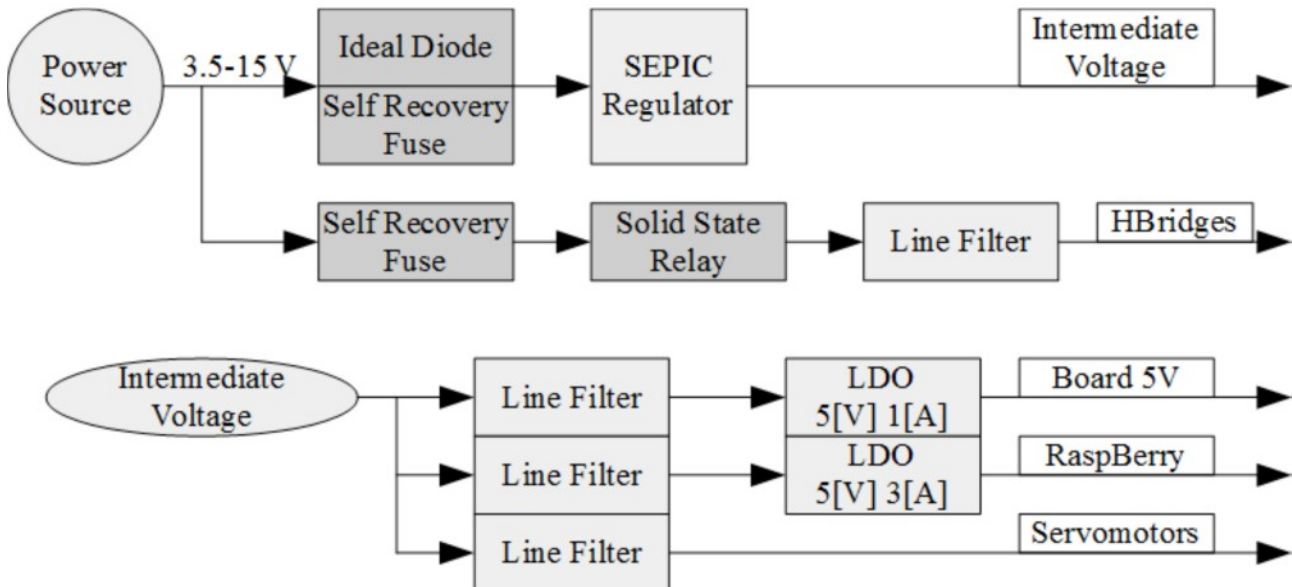


Figure 9: Architecture of the protections

Connecting power wires the wrong way is the most common mistake. The system should not be destroyed by reversing the power wires. Diodes and ideal diodes ensure this does not happen.

Self recovery fuses are used both to prevent over currents and to avoid the hassle of changing a fuse when an over-current situation does happens.

Logic reset due to noise inside the power lines is a problem that is as common as it is subtle. Robot makers might find it very hard to find the source of the instability. Line filters inside the board and separation of the power domains ensure the noise insulation needed to mitigate the problem. The added value of handling noise related problems cannot be overstated.

The user may be in a situation when the DC Motors are either not present or not needed. To selectively disconnect the DC motors, a solid state relay is required.

Power injected by DC motors in the power lines during break is another potential cause of problems. The H-Bridge used features internal recirculation diodes. Another layer of recirculation diodes has been placed near the motor connector to move the power dissipation of the braking away from the bridge.

Power injected by the DC motor during break is recirculated on the input voltage rail. A zener could be inserted to prevent over-voltage situations, but since the input voltage has an wide range, it's impractical, so this protection was not inserted.

4.3 DC Motors and Encoders

The block schematics for the DC Motors and the encoders is straightforward (Figure 10). The microcontroller generates the control signals for the H-Bridges, moving the motors. Encoders can be sampled by the microcontroller in interrupt both to have an odometry reading, or to close a PID position or speed loop on the motors. Both odometry and PID can be implemented at the same time.

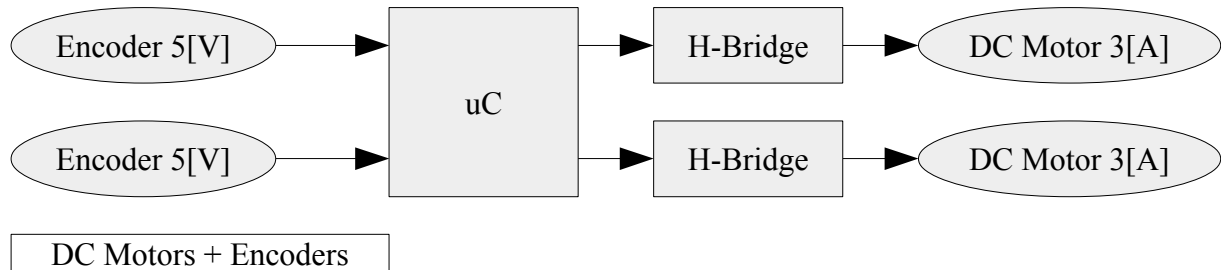


Figure 10: DC Motors Architecture

What is hidden from the block schematics are all the nuances that have to be handled in order to make the system reliable.

- Noise from the motors must be filtered out.
- H-Bridges work at a different voltage from the microcontroller.
- H-Bridges can be cut-off by a solid state relay driven by the microcontroller itself.
- Inputs from the encoder should be filtered.
- Protections should be in place to protect the microcontroller from failure on the encoders
- Encoders can have all sorts of supply voltages and electrical levels. Open drain 12V, TTL, etc...

The choice of the right component is of paramount importance. usually an H-Bridge can work at a low voltage or at an high voltage. One, or the other. Significant effort has been put in making the right choice.

4.4 Microcontroller

An important step in defining the architecture of the board is the choice of the microcontroller. The specifications for the microcontroller so far are as follow:

1. Compatibility with Arduino Shields
2. Enough pins
3. Compatible with Arduino Bootloader
4. Compatible with Arduino Framework
5. Programmable through the Raspberry

The specifications impose many constraints. The first obvious choice would be to use an Atmel ATmega 328P [24], the same microcontroller on board of the Arduino Uno. It satisfies every specification except #2. It doesn't have enough pins.

Specification #1 implies either a working voltage of 5V or the use of level shifters. It was decided to use a 5V microcontroller.

Specification #3 implies the use of an Atmel microcontroller or the porting of the bootloader to another family. It was decided to stick to Atmel.

Specification #4 implies the use of microcontrollers already supported by the Arduino IDE.

Specification #5 implies either two hardware UART communication peripherals or a lot of headaches.

After sifting through many Atmel microcontroller and after careful consideration, the choice was made to use an Atmel ATmega 644 [25]. It features:

- 5V operation at 20MHz
- Package TQFP44. Just enough pins for everything on the board
- Two hardware UART. One for programming and Rosserial communication with the raspberry, one for communication with the Arduino Shield
- Enough pins for everything, barring some limitations
- Level shifter are required for communication with the Raspberry Pi (3.3V)

4.4.1 Architecture: Microcontroller Pin Assignment

The assignment of board functions to pins on a microcontroller is somewhat flexible. There is some leeway as there are multiple instances for many of the embedded peripherals.

There are also many constraints for the pin assignment as well due to limitations about the layout and the routing and limitations due to board space, electromagnetic compatibility of switching tracks, routing for the power lines and more.

Features:

- Device: ATmega 644 5V 20MHz
- Package: TQFP44
- Arduino Shield Connector

- Four Servo Motors
- Two H-Bridges that drive two DC motors
- Two Encoders
- Two LEDs
- Raspberry Pi GPIO

The microcontroller of choice is the ATmega 644, it has 32 usable pins divided in four banks of eight. Many pins are reserved as interfaces with the Arduino Shield.

4.4.2 Analog to Digital Converter ADC

The assignments for the ADC interface are set. The pins must be wired directly to the Arduino Shield ADC pins (Table 1).

Pin Number	Description	Device	Device Description
37	PA0, ADC0	Arduino Shield	PA0, ADC0
36	PA1, ADC1	Arduino Shield	PA1, ADC1
35	PA2, ADC2	Arduino Shield	PA2, ADC2
34	PA3, ADC3	Arduino Shield	PA3, ADC3
33	PA4, ADC4	Arduino Shield	PA4, ADC4
32	PA5, ADC5	Arduino Shield	PA5, ADC5

Table 1: Microcontroller pin assignments for the ADC

4.4.3 PWM Generators

The assignment for the PWM generators are somewhat trickier. The microcontroller has three dedicated PWM peripherals, each generating two PWM outputs for a total of six. All of those six PWM outputs need to be routed to the Arduino Shield connector.

The board also features two H-Bridges that drive two DC motors. Each of them requires two PWM lines for a total of four. Now, there is some overlap, but also some leeway. There are four possible user configurations:

1. The user need neither the DC motors neither a Shield that requires PWM signals.
2. The user need DC motors
3. The user need a shield that make use of up to six PWM signals
4. The user need the DC motors and a shield that uses two PWM signals
5. The user need both DC motors and a Shield that uses six PWM signals.

Now, options 1, 2 and 3 are easy to cater to. Four PWM signals can be forked, sending them to both the Arduino Shield and the DC motors at the same time.

Option 4 is trickier. If a shield uses PWM signals that are already used by the DC motors, then, there will be a soft collision. The user won't be able to control all axis independently.

Option 5 is the trickiest, as the user will have to slave two of the five axis to each others.

The design philosophy comes in our aid. The board should covers all reasonable and cheap options, while avoiding costly ones, to maximize usefulness while minimizing costs.

The solution is to fork four PWM lines. Users will be able to deploy options 1, 2 and 3 with ease, but will be unable to deploy option 4 and 5 without significant effort on their part. It's a reasonable limitation as the user can expect to be able to use either the on-board motor controllers or a shield motor controller. Just not both at the same time.

Microcontroller		Board	
Pin Number	Description	Device	Device Description
43	PB3, OC0A	Arduino Shield	D6, OC0A
44	PB4, OC0B	Arduino Shield	D5, OC0B
13	PD5, OC1A	Arduino Shield	D9, OC1A
14	PD4, OC1B	Arduino Shield	D10, OC1B
15	PD7, OC2A	Arduino Shield	D11, OC2A
16	PD6, OC2B	Arduino Shield	D12, OC2B
13	PD5, OC1A	HBridge A	PWM+
14	PD4, OC1B	HBridge A	PWM-
15	PD7, OC2A	HBridge B	PWM+
16	PD6, OC2B	HBridge B	PWM-

Table 2: Microcontroller pin assignments for the PWM generators

4.4.4 Atmel In-Circuit Programming (ISP) and Serial Peripheral Interface (SPI)

A Microcontroller can be programmed two different ways. Either through an internal communication interface with the help of a bootloader, or with the dedicated Atmel ISP programmer interface.

Using a bootloader is often more convenient for the user, as it allows to program the board through an interface that is already user, like a USB port.

The Atmel ISP interface is the only way to burn the bootloader firmware in the first place therefore is required in order to for the board to be primed for use. In order to make use of the ISP programming interface, the user needs a dedicated piece of HW. The AvRisp MkII [26] was the programmer used in this thesis. It is the cheapest option, but there are any numbers of alternatives on the market.

The ISP make use of the SPI hardware embedded inside the microcontroller. The SPI pin must also be routed to the Arduino Shield, resulting in a limitation. The user will need to disconnect the ISP cable of the programmer when they make use of an Arduino Shield that needs the SPI communication. The assignment for the SPI related pins can be seen in Table 3.

Microcontroller		Board	
Pin Number	Description	Device	Device Description
1	PB5, MOSI	Atmel ISP	ISP MOSI
2	PB4, MISO	Atmel ISP	ISP MISO
3	PB3, SCK	Atmel ISP	ISP SCK
14	RST#	Atmel ISP	ISP RST#
1	PB5, MOSI	Arduino Shield	D11
2	PB4, MISO	Arduino Shield	D12
3	PB3, SCK	Arduino Shield	D13

Table 3: Microcontroller pin assignments for the ISP and the SPI interfaces

4.4.5 Universal Asynchronous Receiver Transmitter interface (UART)

The microcontroller features two hardware UART interfaces. One is used for in-circuit programming and communication with the Raspberry Pi board, the other is used for communication with the Arduino Shield.

The Raspberry Pi works at 3.3V and the microcontroller works at 5V. A level shifter ensures the electrical levels are be compatible between the two.

The UART peripheral is a point to point communication interface. The RX input pin of the first device must be connected with the corresponding TX output pin of the second device, and vice versa. Assignment for the UART pin can be seen in Table 4.

Microcontroller		Board	
Pin Number	Description	Device	Device Description
9	RXI0	Raspberry Pi GPIO	TXO
10	TXO0	Raspberry Pi GPIO	RXI
11	RXI1	Arduino Shield	TXO
12	TXO1	Arduino Shield	RXI

Table 4: Microcontroller pin assignments for the UART interface

4.4.6 Encoders

The board supports two differential encoders with index. They are meant to be used in combination with the H-Bridges and DC motors to close position and speed control loops inside the microcontroller and provide odometry readings.

There are any number of possible electrical configurations for the encoder interface. They can be open collector or open drain, they can be TTL, they might work with 5V, 12V, or higher voltages still. The complexity of the encoder interface increases significantly if compatibility with multiple configurations is given.

In accordance with the design philosophy for the custom board, a single configuration was chosen, 5V TTL encoders, the cheapest most common encoders used in small robotic applications.

If the user wishes to use encoders with different electrical levels, they must take care to place appropriate level shifter electronics outside the custom board. It's a reasonable limitation. Pin assignment for the encoders can be seen in Table 5.

Microcontroller		Board	
Pin Number	Description	Device	Device Description
25	PCINT22	Encoder A	Channel A
26	PCINT23	Encoder A	Channel B
31	PCINT6	Encoder A	Channel Z
40	PCINT8	Encoder B	Channel A
41	PCINT9	Encoder B	Channel B
30	PCINT7	Encoder B	Channel Z

Table 5: Microcontroller pin assignments for the encoders

Any meaningful incremental encoder firmware implementation requires for the encoder reading to be handled by an Interrupt Service Routine (ISR).

The requirement is for the encoder signals to be able to trigger such a response inside the microcontroller. Atmel allows for ISR to be triggered by any pin, so the specification is easy to

meet.

The programming of an encoder reading ISR is made easier and more efficient if both A and B channels of the encoder can trigger the same ISR.

4.4.8 Servomotors and Generic Digital I/O pins

Finally, it's time to route all remaining pins that do not have other specifications. This include the Servomotors, as they do not require any hardware peripheral. The assignment for the remaining pins can be seen in Table 6.

Microcontroller		Board	
Pin Number	Description	Device	Device Description
42	PB2	H-Bridge	Shutdown
2	PB6	Leds	Led 0
3	PB7	Leds	Led 2
9	PC0	Servomotors	Servo 0
10	PC1	Servomotors	Servo 1
11	PC2	Servomotors	Servo 2
12	PC3	Servomotors	Servo 3
9	PC0	Arduino Shield	D2
10	PC1	Arduino Shield	D4
11	PC2	Arduino Shield	D7
12	PC3	Arduino Shield	D8
23	PC4	Arduino Shield	D12
24	PC5	Arduino Shield	D13

Table 6: Microcontroller pin assignments for Servomotors and generic digital I/O

The control signal for a servo motor is a Pulse Width Modulated (PWM) signal with a period of 20mS and usually the 'zero' at 1.5mS width.

It would be possible to use an hardware PWM generator to generate the control signal for the servomotors, but those interfaces are already assigned, furthermore, a servomotor signal can be efficiently generated using an hardware timer and an ISR.

The shutdown signal for the H-Bridges can be just another general I/O as there are no hard specifications tied to it. It can be wired to just about every pin.

Lastly, are the remaining general digital I/O required by the Arduino Shield connector. There are not enough pin left to have individual connections. Some of the digital I/O lines are shared between the Arduino Shield and the Servomotors. This introduces a limitation. The user will not be able to use both a servomotor slot and a Shield that use the shared I/O independently.

The custom board features two LEDs. They are connected to the ISP programmer pins. This introduces a limitation. SPI and LEDs cannot be used independently.

5. Test Boards

Chances are that some mistakes will slip through the design phase. It is cheaper, easier and quicker to individually test parts of the design that are most likely to fail, rather than make with the complete design only to have to redo it again once mistakes are discovered.

Two individual boards test two critical sections of the design:

- The Power Regulator
- The H-Bridge Driver

5.1 Design of the Power Regulator Test Board

The Power Regulator is a critical part of the design. A test board allows to test the Power System independently from all other components.

Design flow:

- Choice of the SEPIC Controller
- Design of the SEPIC Regulator
- Design of the Linear Regulators
- PCB layout for the test board
- BOM Bills of Materials and production of the PCB

5.1.1 Choice of the SEPIC Controller

The topology of the switching regulator was decided to be a SEPIC. Specifications are:

- Input: 3.5[V] ~ 15 [V]
- Output: 5.7[V], 5[A]

The critical part of the design of the switching regulator is the choice of the controller. There are two kinds of controllers:

- Integrated MOS Switch
- External MOS Switch

A quick research showed that controllers featuring an integrated switch cannot meet the specifications. A controller with and external MOS switch is required.

Several manufacturers were considered. The tightest specification is the wide range of the input voltage. Many controllers were found that were able to work with low voltages (3V) using an internal charge pump to drive the MOS. Many controllers were found that could handle higher voltages (15V). Finding a controller able to work over the full range proved challenging. In the end a candidate was found: LT1619[27].

Other controller candidates that were considered were: LM3488, LT1624, LT1871, LT3757, LTC3806, TPS51363. Of those, the LM3488 [28] seemed the most promising.

5.1.2 Design of the SEPIC Regulator

With the topology and the controller decided, the next step was to design the switching regulator itself and calculate the value for all components.

The schematics of the regulator (Figure 11) was derived from the datasheet of the component LT1619 with the help of the well written Application Note AN-1484 [29] from Texas Instrument about SEPIC regulators.

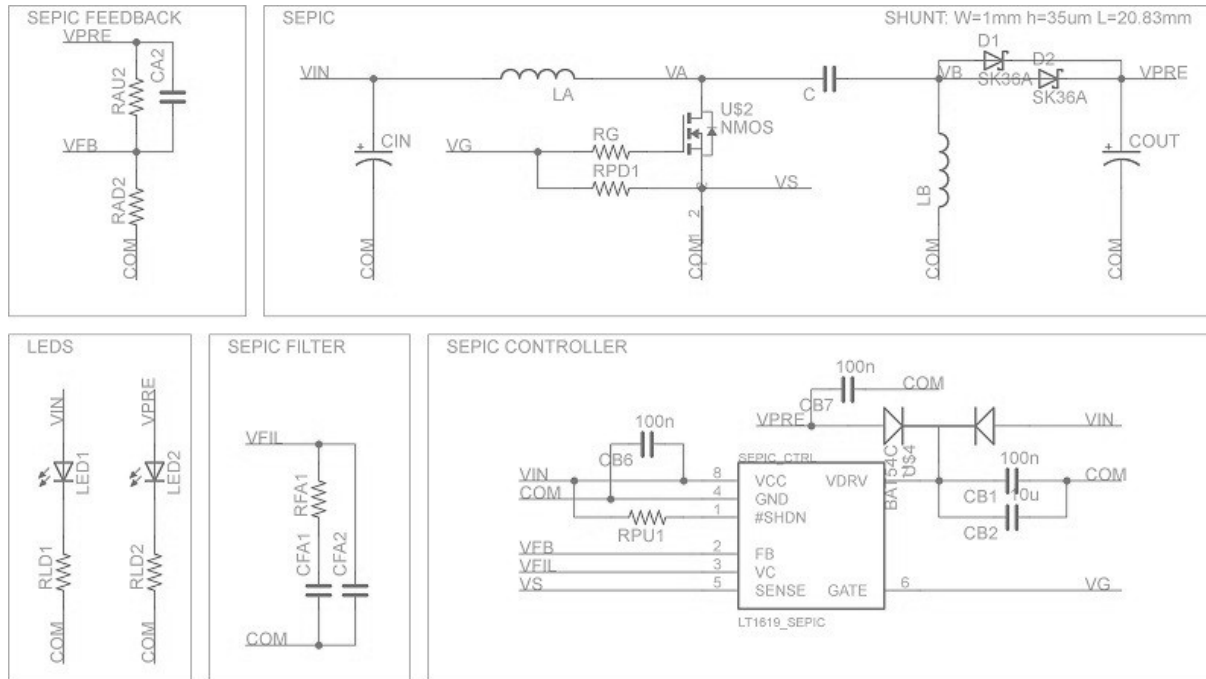


Figure 11: Schematics of the SEPIC regulator

As a low side switch, the obvious choice was to use a N-channel MOS. The switch need a low voltage threshold, since the chosen controlled does not have a charge pump and need a nominal current greater than 10.93[A].

Many candidates were found that satisfied the specification. The transistor chosen was the FDD6635 which has a threshold voltage of 3[V] and a nominal current of 15[A].

The choice of the SEPIC diode is not critical. There are many candidates as specifications are not tight for this component. A Schottky diode was chosen to improve efficiency. The final choice was the B560C with 42[V] of blocking reverse voltage and 5[A] of nominal current.

The choice of the capacitor is critical since it's the component that handles the energy transfer between input and output. If an electrolytic capacitor is used, it must be rated for an high enough RMS current. If a ceramic capacitor is used, the problem becomes finding one that has both enough capacity. The choice was to use a ceramic capacitor.

The input capacitor is not critical components as the SEPIC is continuous in input. A 220uF 35V electrolytic capacitor was chosen for the job.

The output capacitor has tighter specifications as the SEPIC is not continuous in output. Care has to chose one that can handle the RMS current. A 220uF 16V electrolytic capacitor was chosen for the job.

The feedback network is what decides the value of the steady state output voltage. The output voltage is 5.7V. A feed forward capacitor is used to improve transient response, its value was calculated according to the datasheet of the component.

The controller LT1619 requires the design of several filters. The datasheet of the component does an excellent job in explaining how to choose the value of the components for the filters.

The controller requires both the output voltage and the switch current to close the control loop. A shunt resistor is required near the source of the SEPIC switch. It was decided to try and use a shaped PCB track in place of the shunt resistor.

Linear Technology provides the SPICE model for the SEPIC controller LT1619. Before routing the PCB and ordering the components, LT Spice was used to test a simulated version of the regulator in various load configuration and input voltages. The feedback from the simulation was used to iteratively correct the value of the components, until a combination was found to provide the best performance (Figure 12).

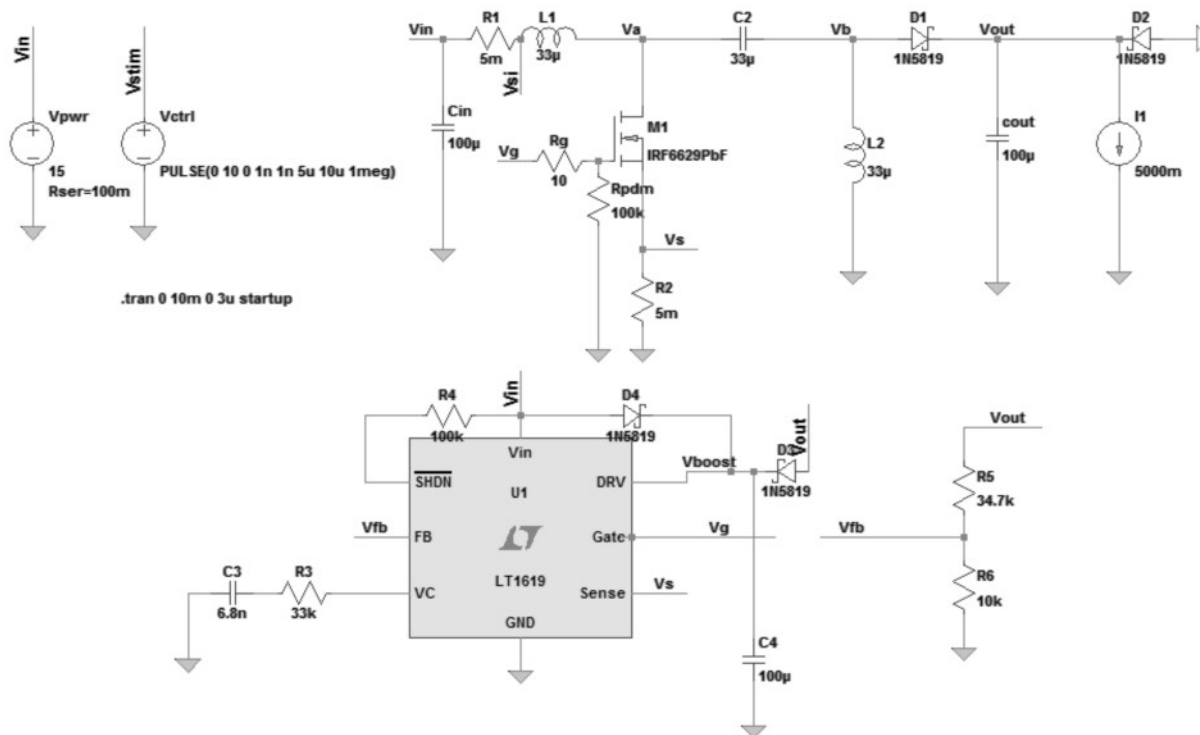


Figure 12: LT Spice SEPIC Simulation Schematics

After the LT Spice simulation, the final value for the components of the SEPIC regulator in Figure 11 are as follow:

SEPIC Power Stage:

- Switch: FDD6635
- Diode: B560C
- LA, LB: EPCOS B82477G4333M000 (33uH)
- C: TDK CKG45NX7S2A106M500JH (Ceramic 10uF)
- Cin: 220uF 35V
- Cout: 220uF 16V
- Rg: 10 Ω SMD0805
- Rpd1: 100 K Ω SMD0805

SEPIC Controller:

- Rpu1: 100 K Ω SMD0805
- Cbypass (3): 100nF 50V SMD0805
- CB2: 10uF 16V SMD1210
- Schottky barrier: BAT54C

SEPIC Feedback network:

- RAD2: 34.8 K Ω SMD0805
- RAU2: 10 K Ω SMD0805
- CA2: 10nF 50V SMD0805

SEPIC Filter network:

- CFA1: 6.8nF 50V SMD0805
- RFA1: 34.8 K Ω SMD0805
- CFA2: Not used

The simulated output of the SEPIC regulator in a typical use case, the robot maker will power the board with a three cell LIPO battery giving 11.1V in input and the load will be about 2A with the Raspberry Pi drawing about 1A two servomotors drawing each 500mA. Transient and steady state response of the regulator can be seen in Figure 13.

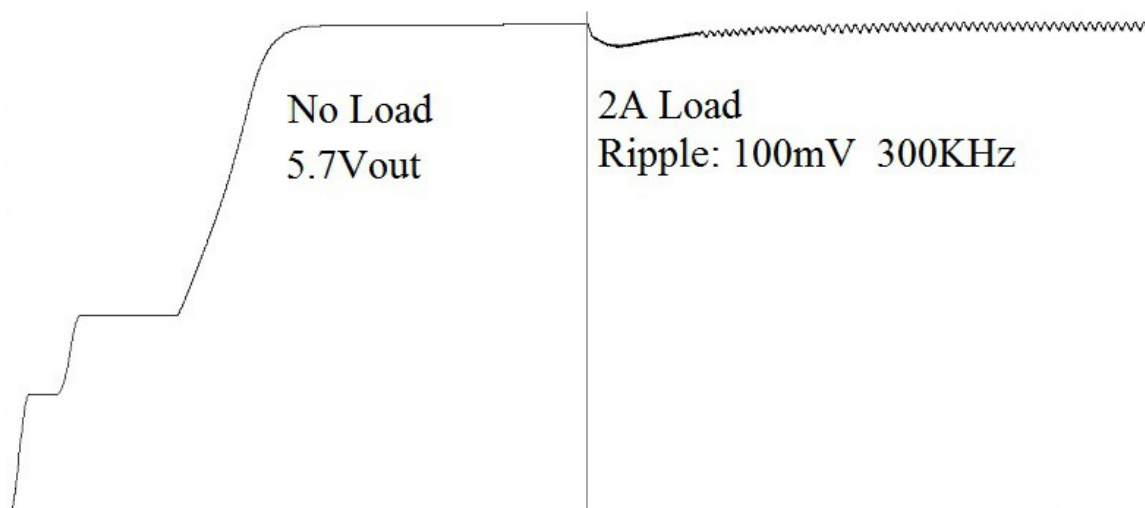


Figure 13: LT Spice Simulation. 11.1V_{in}, 2A_{out}

5.1.3 Linear Regulators

Switching regulators are efficient but noisy. Linear regulators have clean outputs but are inefficient. I combine the two to have the best of both worlds.

The bulk of the voltage swing between input and output is handled by the efficient SEPIC regulator. The remaining voltage swing is handled by linear regulators. This way I do most of the regulation efficiently, and I handle what's left in a clean way. LDO have about 700mV of drop-out voltage across them.

Two linear regulators are used:

- A 5V 3A LDO for the Raspberry Pi

- A 5V 1A LDO for the Arduino Shield, the microcontroller and encoders.

The design of the linear regulators is straightforward. They need to be able to work at their rated current with only about 700mV of drop-out. It all comes down to choosing the right component.

The schematics used for the linear regulator can be seen in Figure 14.

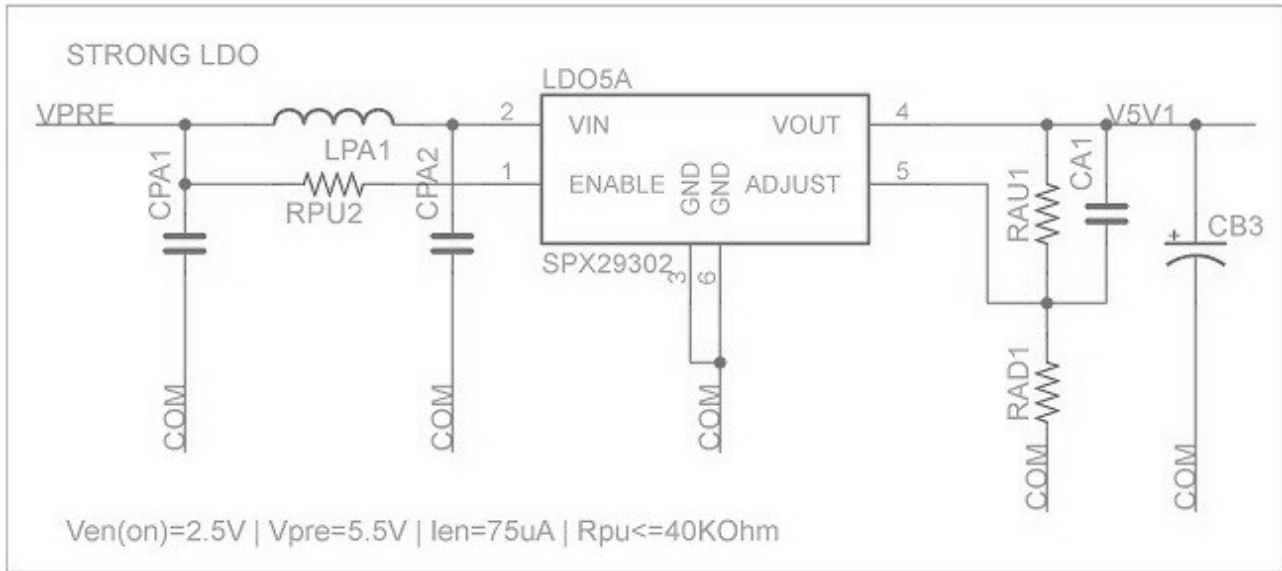


Figure 14: Schematics for the LDO Regulators

The choice for the linear regulator for the raspberry was the EXAR SPX29302 [30]. It provides adjustable output, high precision and only need about 600mV of drop-out at 3A. A perfect choice to powers the raspberry. The schematics is in Figure 11.

There is a variant that has a fixed voltage, and requires fewer components. The reason the adjustable version was chosen was to have the ability to increase the raspberry voltage from 5.0V to 5.2V, increasing the stability of the device in high load configurations.

A line filter is placed at the input of all linear regulators to further clean up noise and reduce cross talk between power lines. This is an important feature, as there are DC motor controllers on board that are known to inject dangerous high frequency noise in the power lines.

The design of the other linear regulators is straightforward. It's all about choosing a component that works with a low enough drop-out voltage.

5.1.4 Supply Board Layout

The power regulator is a critical part in the design of the board. A test board was created with only the power regulator section on it in order to test the power section independently.

Eagle cad was used to design the footprint of the components, design the layout of the PCB and generate the Gerber files.

The design philosophy when it comes to power tracks was to use copper areas to make carry as much current as possible.

The board measures 24 by 61 mm. The thickness of the copper layer will be about 22um, Maximum current flow will be about 8A.

The board (Figure 15) include a SEPIC controller, power components for the SEPIC regulator, linear regulators and line filters, along with two LEDs that light when the board is regulating power.

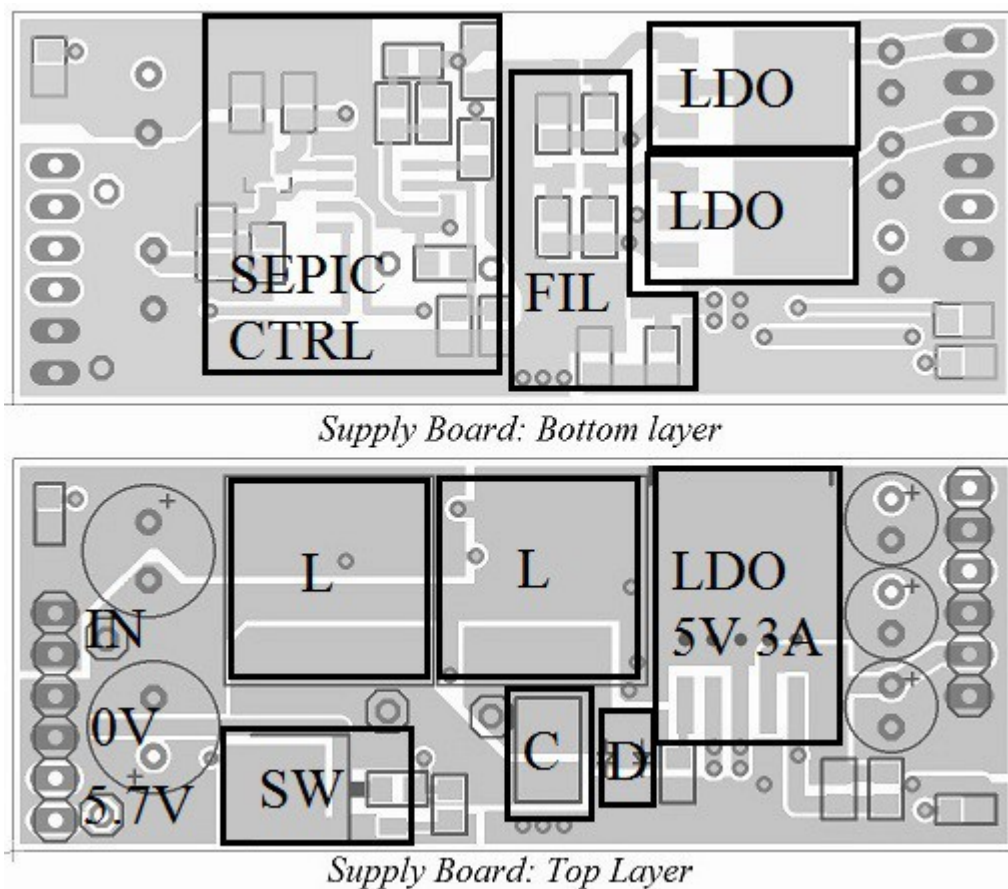


Figure 15: Schematics PCB layout

The board interfaces with the outside world through two six pin wide 2.54mm screw connectors. The board takes from 3.5V to 15V in inputs and generate several outputs: 5.7V@5A, 5V@3A, 5V@1A and 3.3V@1A. Not all output can be used at their rated power at the same time.

A more detailed description of details for the layout of the board can be seen in Figure 16.

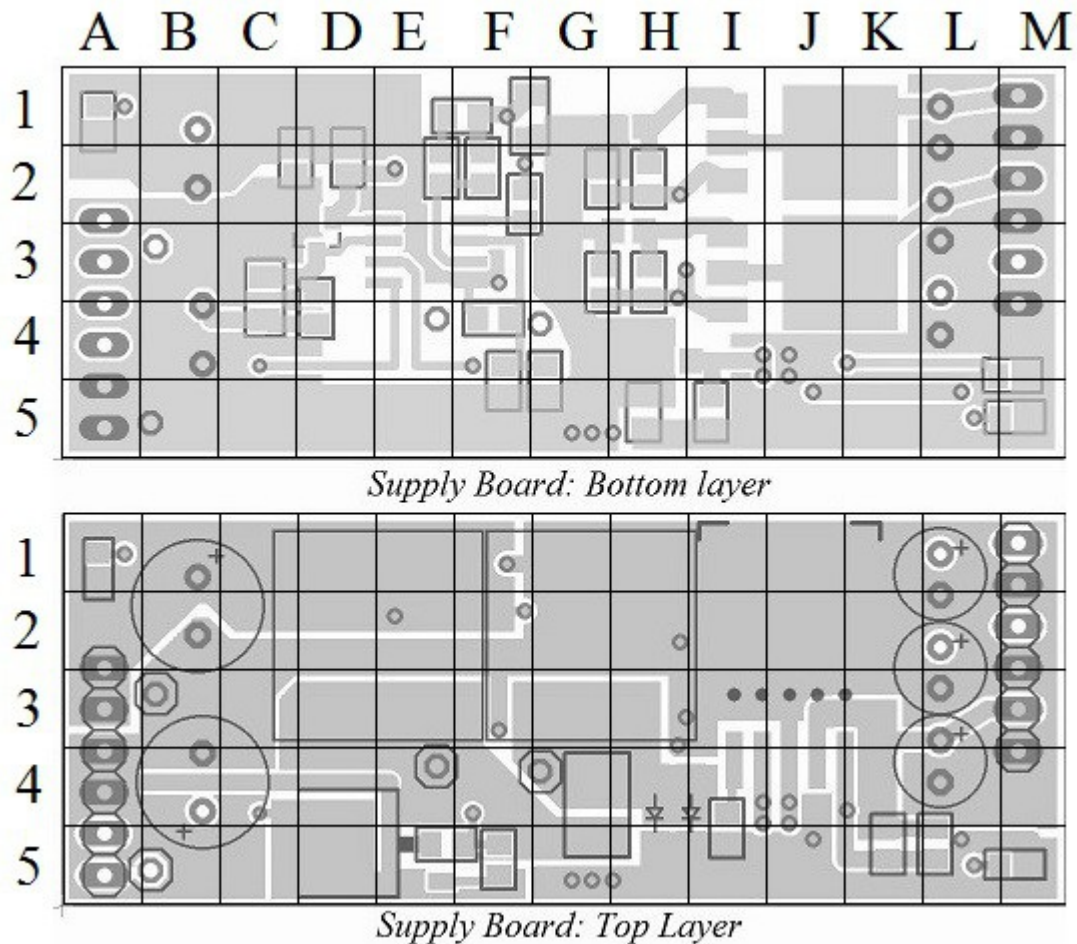


Figure 16: Test Power Regulator Board Layout Sectors

PCB Layout Details:

- (Top) B4, C4, D4, E4, E5: Shunt track. A shaped track that presents a controlled resistance, used by the SEPIC controller to sense the current on the switch.
- (Bot) E3: SO8 package of the SEPIC controller LT1619
- (Top) A1: LED, Input power active
- (Top) M5: LED, Output 5.7V active
- (Bot) From A1 to F2: Copper area carries input current
- (Top) From A4 to M1: Copper area for ground voltage
- (Bot) From A3 to G2 to G1: Copper area for intermediate voltage
- G5, H5: Multiple parallel vias carry intermediate voltage to LDO regulators
- (Bot) From A1 to E2: Copper area for internal SEPIC node
- (Top) K5, L5 and (Bot) M4: feedback network for 3A LDO regulator

Eurocircuit [31] was chosen as manufacturer for the boards. A preview of the final aspect of the board can be seen in Figure 17.

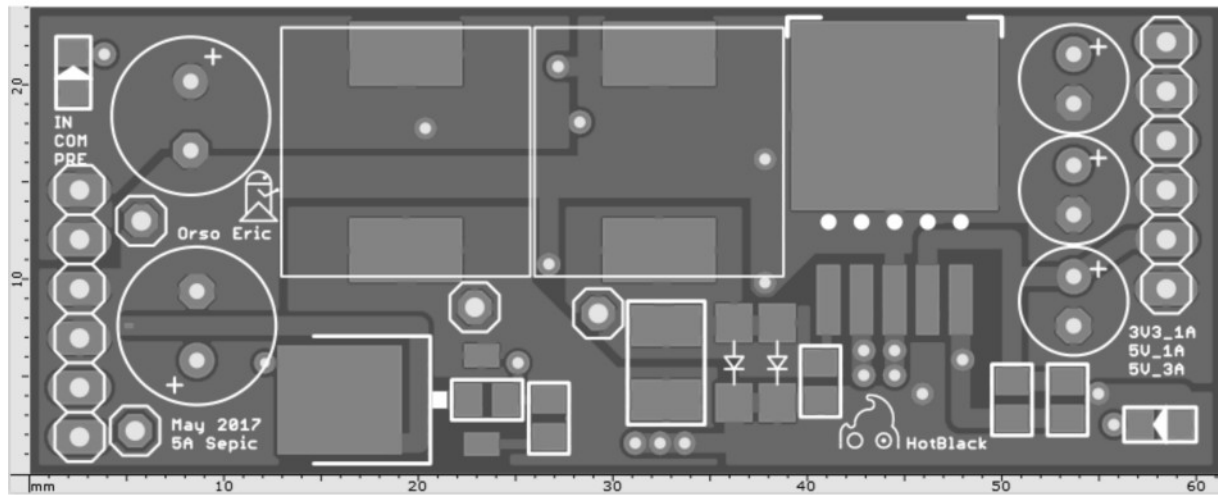


Figure 17: Power Regulator Test Board Preview

5.1.5 Supply Board BOM

The BOM Bills of Materials for the Power regulator test board can be seen in Table 7. Components were supplied by RS Components.

<i>Description</i>		<i>Code</i>	<i>Quantity</i>
Led Verde 2.2V SMD0805		LG R971	2
Low Vth NMOS		FDD6635	1
LDO 5V 800mA sot223		REG1117-5	1
Electrolitic Capacitor 220uF 35V 755mA 8mm	35ZLH220MEFC8X11.5		1
Inductor 33uH 3A	B82477G4333M000		2
Twin Scottky 200mA sot23	BAT54CW		1
Diodo Schottky 60V 0,75V 3A	SK36A R2		2
Sepic Controller SOIC8	LT1619ES8#PBF		1
Resistenza 1K SMD0805	CR0805-FX-1001ELF		2
Resistenza 10K SMD0805	CR0805-JW-103ELF		2
Resistenza 33.8K SMD0805	CPF-A-0805B34K8E1		2
Resistenza 100K SMD0805	RMC1/10K104FTP		3
Condensatore 6.8nF V SMD0805	08055F682KAZ2A		1
Condensatore 100nF 50V SMD0805	08055C104KAT2A		4
Capacitor Sepic 10uF 100V SMD1812	CKG45NX7S2A106M500JH		1
Condensatore 10uF 25V SMD1206	GRM31CB31E106KA75L		2
Ferrite 240nH 26mO 3.2A SMD0806	74479876124C		3
Electrolitic Capacitor 100uF 6,3V 480mA 6,3mm	63ZLG100M63X7		4
Morsetto a vite 5.12mm 1x3			4
Resettable Fuse PTC 85mO	MF-NSMF200-2		1

Table 7: BOM for the Test Power Regulator Board

5.2 Design H-Bridge Board

The H-Bridge is a problematic component. The difficulty lies in finding one that fits the specifications. A test board has been used to test it independently from other components and validate the choice.

5.2.1 Component Choice: H-Bridge

Driving DC motors involves the use of a circuit known as H-Bridge, a structure in which four solid state switches are commuted in a PWM modulation.

The rotor of the engine has a significant inertia and act as a low pass filter. The result is that the speed of the DC motor will be directly proportional to the duty cycle of the PWM control signal. An H-Bridge allows to control both the speed and the direction of rotation of a DC motor.

There are any number of ways to implement an H-Bridge. Low power motors (<50mA) can be driven by digital I/Os. High power motors (>10KW) are driven by high power IGBT transistors at high voltages and currents.

Target robotic platforms use at most 3A DC Motors at low voltages (<15V) usually. There exists fully integrated H-Bridge components that makes the control as easy as powering the component and wiring in a digital PWM signal.

The problem arise from the intended specifications. The board allows for a wide input range (3.5-15V). H-Bridge that works at lower voltages, have a low maximum voltage as well. H-Bridges that works at 15V will cut-off power below a certain threshold (9V usually). An H-Bridge is limited by the gate driver and its charge pump.

One option is to make the H-Bridge from discrete components. This option was quickly discarded because of the area required. It just can't fit the board.

Another option is to power the H-Bridge from the internal intermediate voltage of 5.7V, but it was discarded as well because it would limit the choice for the DC Motor and it would further tax the already burdened SEPIC regulator.

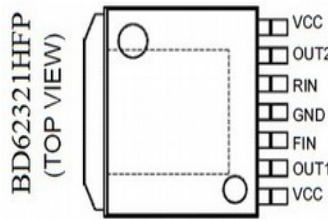
The design philosophy for this board is not provide costly features. Robot makers can take advantage of servo motors when working with a low input voltage, so it is a reasonable limitation for the DC motors to be inoperable when using a small battery.

A through market research was conducted to see if there were H-Bridges available on the market that had a small footprint and a reasonably low cut off threshold. Ideally, the H-Bridge should be able to operate from two LIPO cells to maximize usefulness.

Again, it is about choosing the right component for the job. H-Bridge Specifications:

- Maximum voltage: 15V
- Cut off voltage greater than 6.6V (Two LIPO cells at low charge level)
- Small enough footprint for two to fit the custom board

A research showed that an H-Bridge existed that satisfied the tight specifications. The BD62321 [32] from ROHM (Figure 18).



*Figure 18: ROHM
BD62321*

Unlike other H-Bridges, like the LMD18200 [33], the BD62321 has just seven pins. No external charge pump capacitors, or filters are needed at all. Only power, motor pins and PWM signals are routed. The package seems also good from a thermal point of view. The component features:

- 36Vmax, 6Vmin, 3Amax, 100KHz PWM
- HRP7 Package (9.4 by 10.5 mm)

5.2.2 Mini Arduino Shield

Testing an H-Bridge component requires a way to generate PWM signals.

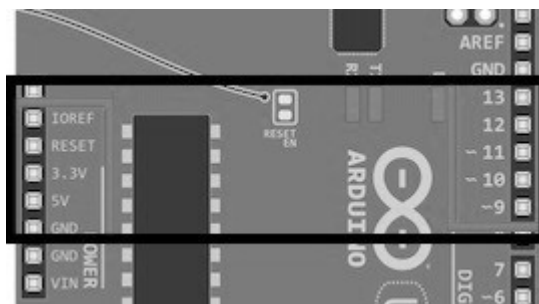
An option would be to include a microcontroller inside the board, but that seems overkill as this board is designed to just test the component.

Since the PWM signal will not be generated internally, it must come from the outside. An option would be to just place some screw connectors and make a simple breakout board.

A more attractive proposition is to use an Arduino Uno board as controller, and looking at the Arduino Shield connector, there is an opportunity: The PWM pins are exactly on the opposite side of the power pins.

The test board makes use of a minimal number of Arduino Connector pins and is designed as the smallest Arduino Shield.

The outline of the mini Arduino Shield to test the H-Bridge component is shown in Figure 19.



*Figure 19: Outline of the mini Arduino
Shield*

Some pins are unused, and there is some free space on the board. The space host a couple of LEDs, a linear regulator to power the Arduino Uno and an encoder connector to allow for a PID control loop.

5.2.4 H-Bridge Test Board Design

The schematics for the H-Bridge test board can be seen in Figure 20.

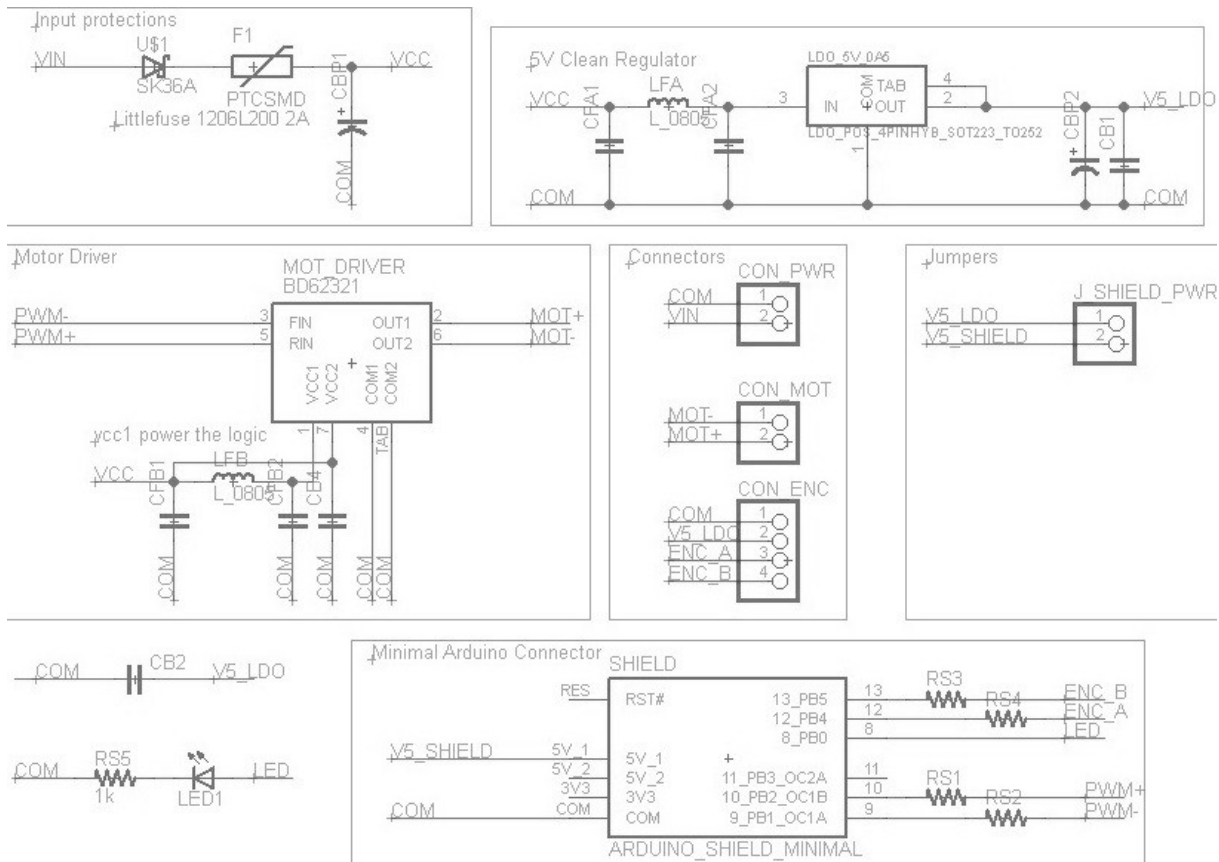


Figure 20: Schematics for the H-Bridge Test Board

This board is made of several sections:

- Input Protections: Protect against reverse voltages and over-currents
- 5V Regulator: Powers the Arduino shield board
- Motor Driver: The H-Bridge under test along with line filters against motor noise
- Connectors

As for the microcontroller pin assignment, the PWM pins for the H-Bridge must be routed first as they require dedicated HW. Pin D9 and D10 were chosen.

Remaining pins have no hard constraint to them. Any generic I/O would do. D8 is wired to a LED, D12 and D13 are wired to the encoder channels.

DC motors are known to inject high frequency noise in the power lines and can easily make microcontroller unstable. A line filter is inserted to protect the system.

5.2.5 H-Bridge Test Board Layout

The layout of the H-Bridge Test Board (Figure 21) is straightforward. Special care was taken in getting the footprints of the components right, in giving enough area to the power tracks and in the design of the Arduino Uno connector footprint.

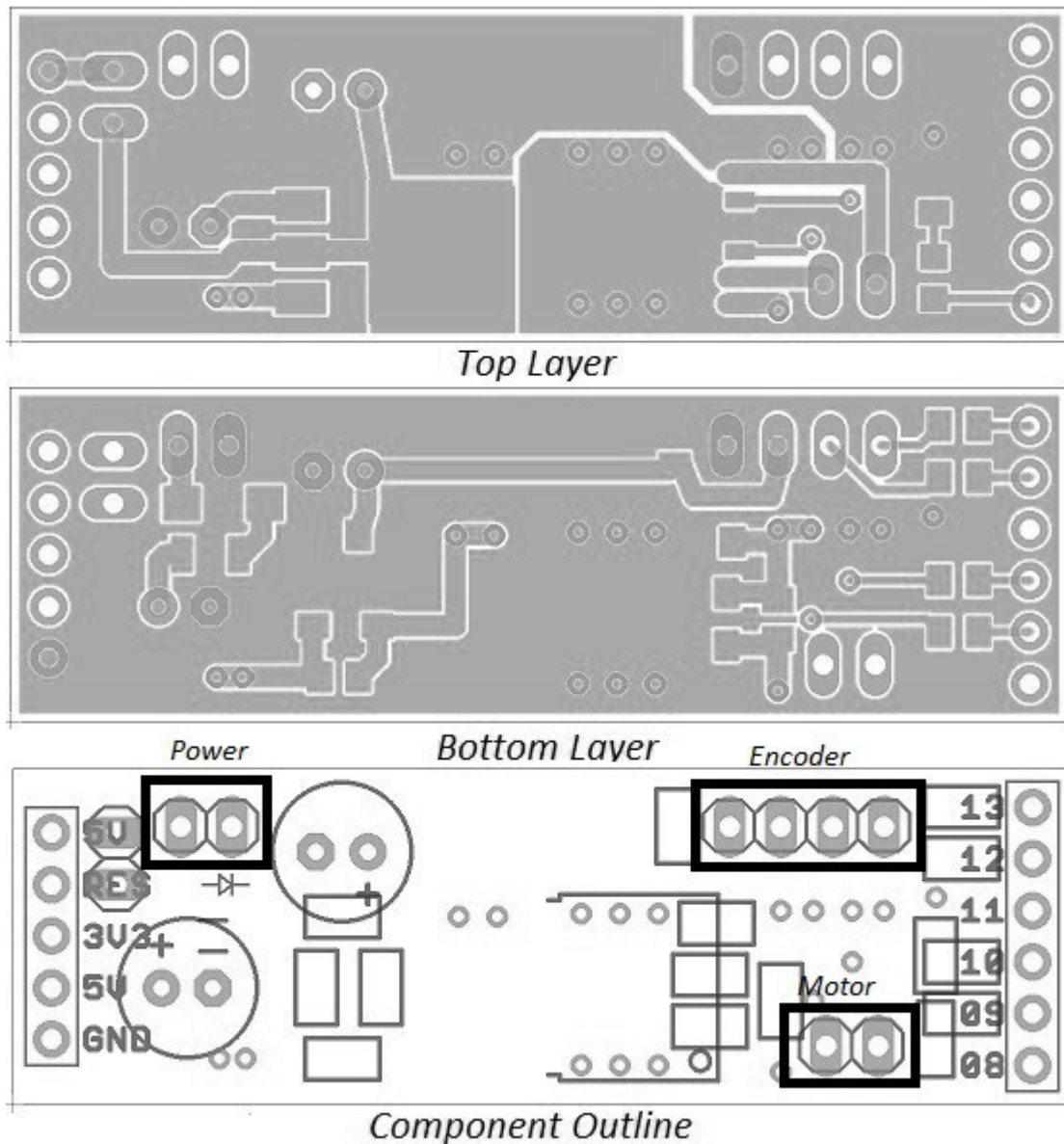


Figure 21: Layout for the H-Bridge test board

In Figure 22 the area allocation for the various sections of the board can be seen.

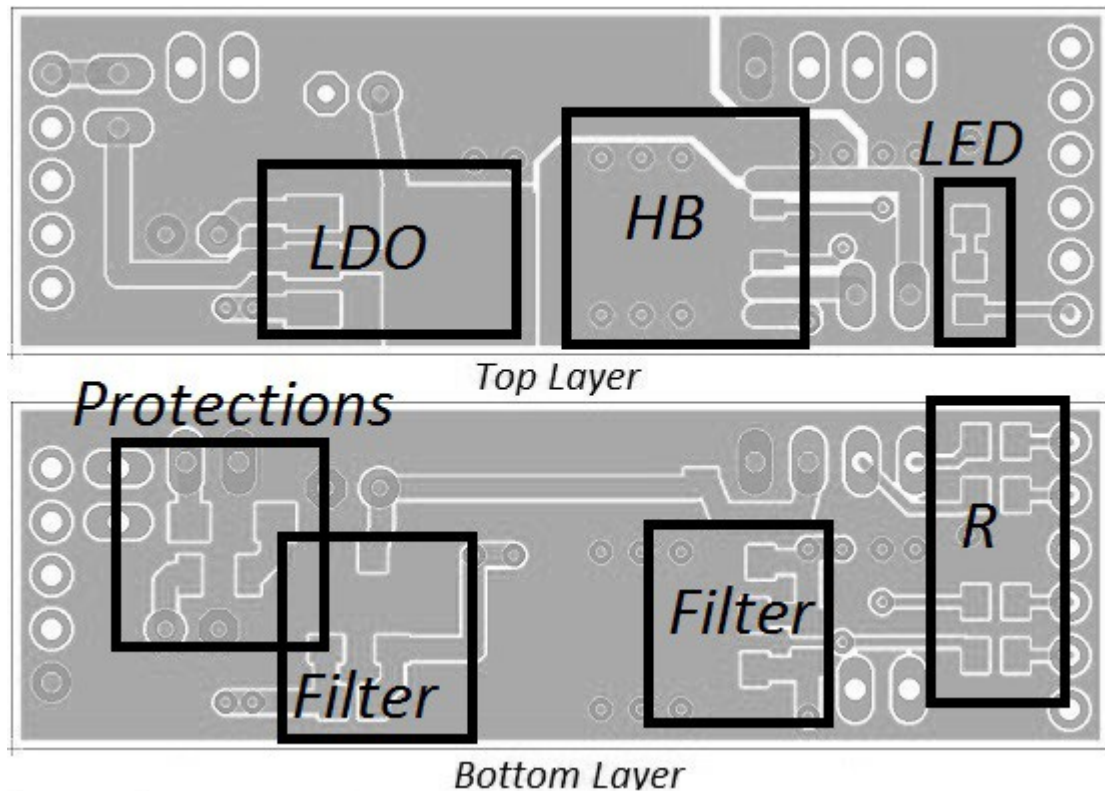


Figure 22: Area allocation for the H-Bridge test board

The board has been manufactured by Eurocircuits. In Figure 23 the preview of the board can be seen. The final dimensions are 16.5x52[mm]. It's about the smallest a practical Arduino Shield can be.

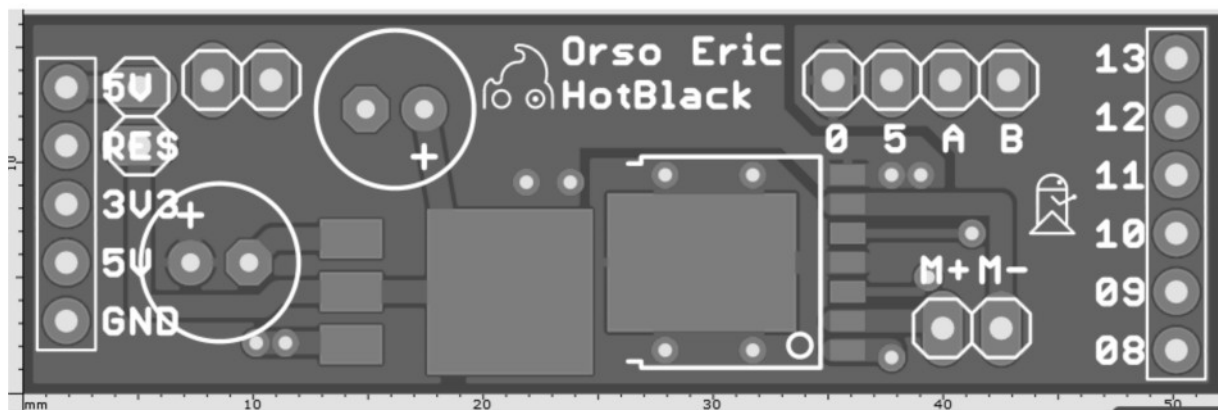


Figure 23: Preview of the H-Bridge test board

5.2.6 H-Bridge Test Board BOM

The BOM, Bill Of Materials can be seen in Table 8. The components were ordered from RS Components.

Description	Manufacturer Code	Quantity
LDO 5V 800mA sot223	REG1117-5	1
Morsetto a vite 2.54mm 1x4	1725672	1
Morsetto a vite 2.54mm 1x2	1725656	2
H-Bridge. 3A 6Vmin 36Vmax	BD62321HFP-TR	1
Ferrite 240nH 26mO 3.2A SMD0806	74479876124C	2
Led Verde 2.2V SMD0805	LG R971	1
Condensatore 100nF 50V SMD0805	08055C104KAT2A	7
Resistenza 1K SMD0805	CR0805-FX-1001ELF	5
Diodo Schottky 60V 0,75V 3A	SK36A R2	1
Electrolitic Capacitor 100uF 6,3V 480mA Ir 6,3mm	63ZLG100M63X7	1
Electrolitic Capacitor 220uF 35V 755mA 8mm	35ZLH220MEFC8X11.5	1
Resettable Fuse PTC 85mO	MF-NSMF200-2	1

Table 8: BOM for the H-Bridge Test Board

6. Robotic Platform

The objective for this thesis is to develop an useful framework for small robotic platforms. Building one such platform comes natural as it allows to test and experience the features as they are being developed.

It takes time to design, route and manufacture the custom shield. Boards already developed by the candidate for past robotic platforms have been used in a configuration close enough to the desired final result provide meaningful feedback.

6.1 Seeker of Ways

The small mobile robotic platform built for this thesis is named Seeker of Ways (Figure 24). It's a good sounding and apt name for a robot that focuses on navigation and remote control from the cloud. This platform has been used extensively to write and test the firmware for the board and to develop example applications using the HotBlack software framework.

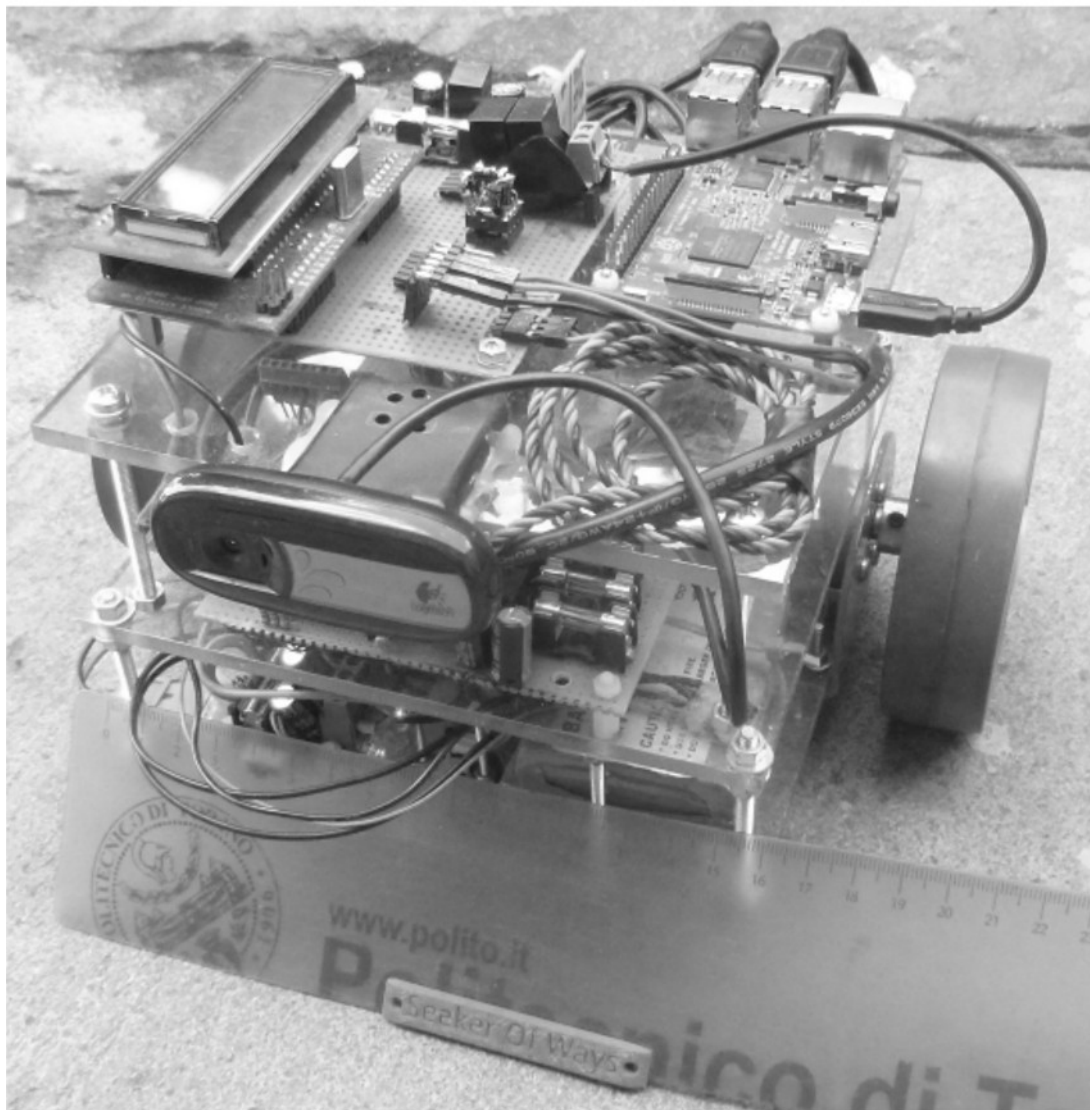


Figure 24: Seeker of Ways

The platform is made of the following components:

- Three levels of polycarbonate as support structure.
- Commercial DC motors with wheels and integrated encoders for motion
- Three LIPO cells in series (3700mAh)
- LIPO battery charger
- Power Switch / Protections
- Motor controller
- USB Camera

6.1.1 Seeker of Ways: Architecture

The architecture of the robotic platform can be seen in Figure 25.

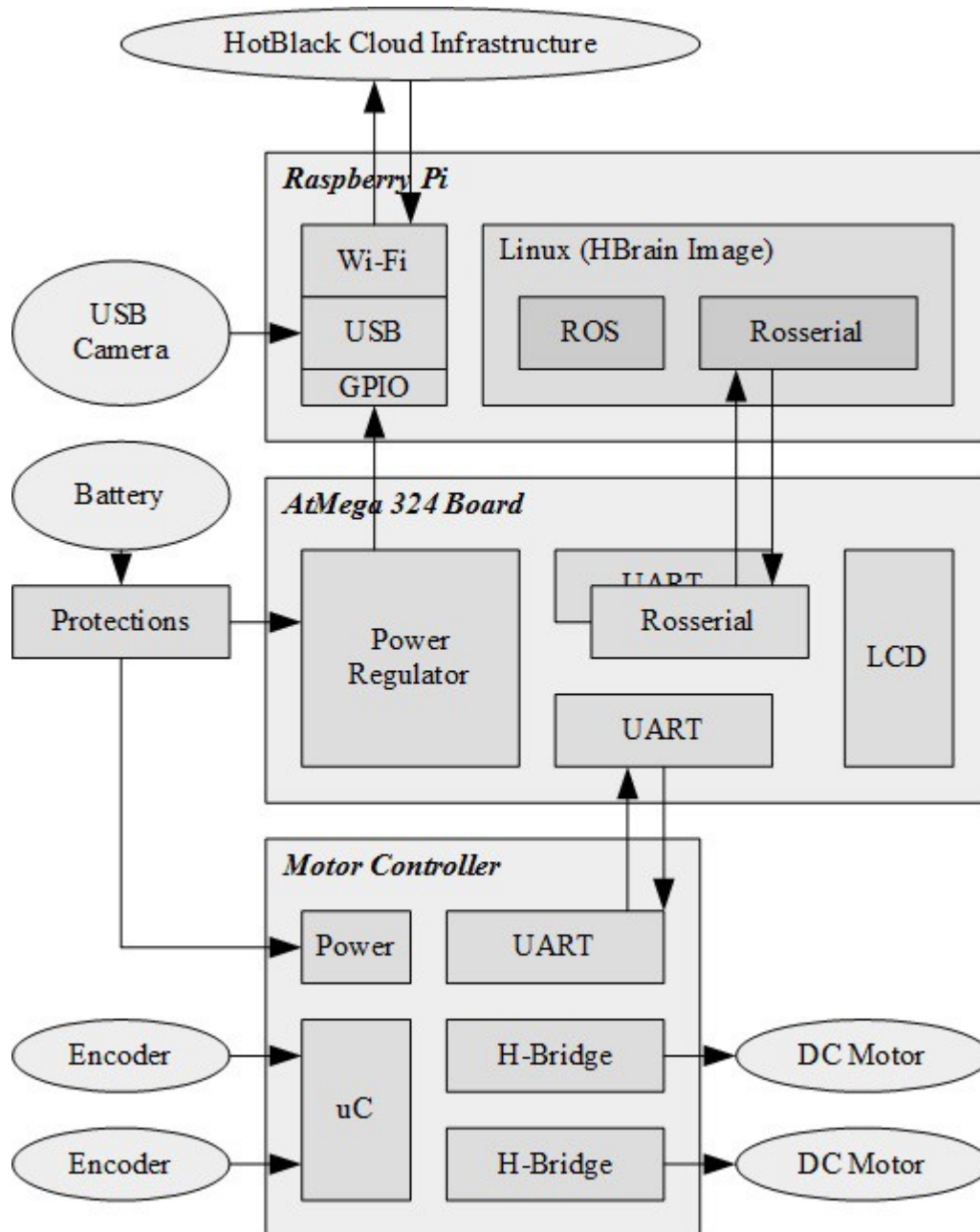


Figure 25: Architecture of Seeker of Ways

The architecture works as follow:

At the topmost level there is the HotBlack cloud infrastructure that runs the HotBlack software framework. The user programs and controls the robot through a remote interface.

The Raspberry Pi runs an operating system Figure developed by HotBlack Robotics. The user does not need to modify the Figure. Te user need a wireless network named DotBot for the first use. The name of the network can be reconfigured once inside.

The ROS Framework through an hidden but complex chain of events searches for ROS compatible devices, including the motor controller connected through the UART interface using the

Rosserial protocol.

An ATmega 324 board runs the firmware that will be run by an ATmega 644 once the custom shield is complete. By default the board recognizes custom made topics that allow to move the motors and read the encoders.

The ATmega324 board connects through a second UART interface to a motor controller with a simple protocol. Once the shield is complete, the motor controller too will be on-board, and there will be no need for a second board.

The Raspberry Pi is powered through the GPIO by the ATmega324 board, just like it will be in the final configuration. In this version the motor controller draws power directly from the battery.

6.1.2 Motor Controller

The motor controller (Figure 26) is one that was built a while ago. It was designed exactly for this kind of robotic applications and it can be considered an early precursor to the custom board being developed.

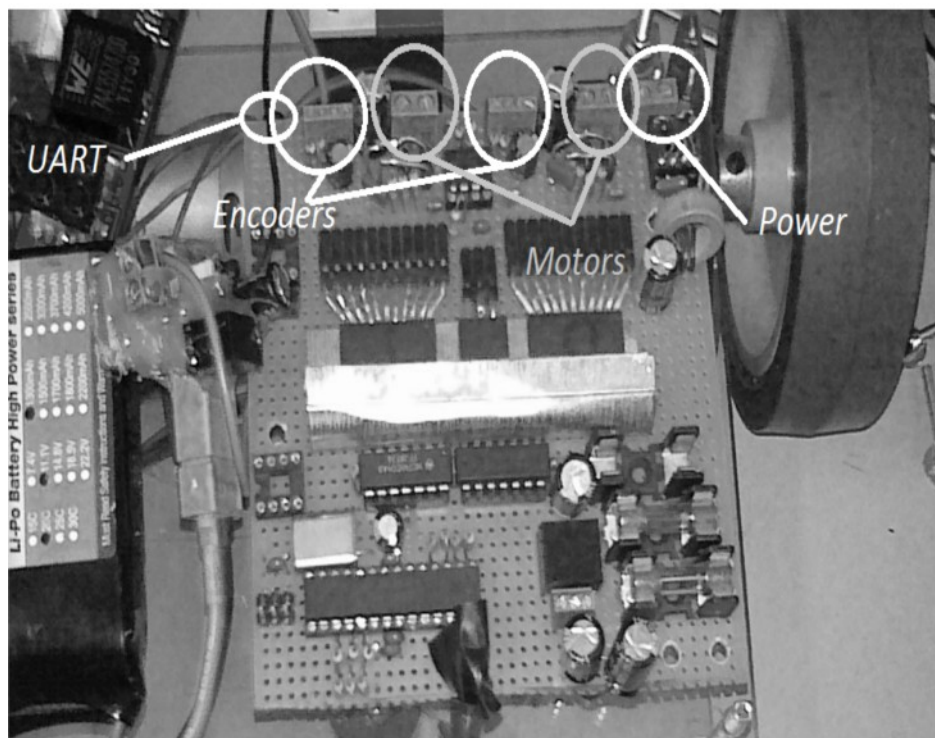


Figure 26: Motor Controller of Seeker of Ways

This motor controller is soldered on a prototype board. It communicates through an UART interface and controls two DC Motors with Encoders. PID can be closed in both speed and position.

The motor controller uses the LMD18200, an integrated H-Bridge with MOS switches, itself an inferior version to the newer better motor controller chosen for the custom board.

The microcontroller is an ATmega328, the same microcontroller used by an Arduino. The firmware is written in C using Atmel's own IDE and cross compiler, the AVR Studio 7. Actually, the previous firmware was developed on AVR Studio 4, the old code has been ported to the new IDE version to take advantage of the better cross compiler.

Most of the firmware was already done, and the board was fully tested and reliable. Despite gathering dust for a few years, it still worked like a charm when powered up as great care was taken

when bundling the wire-wrap cables on the back to avoid malfunction.

The firmware is made of several modules. A block schematics of the architecture of the firmware can be seen in Figure 27.

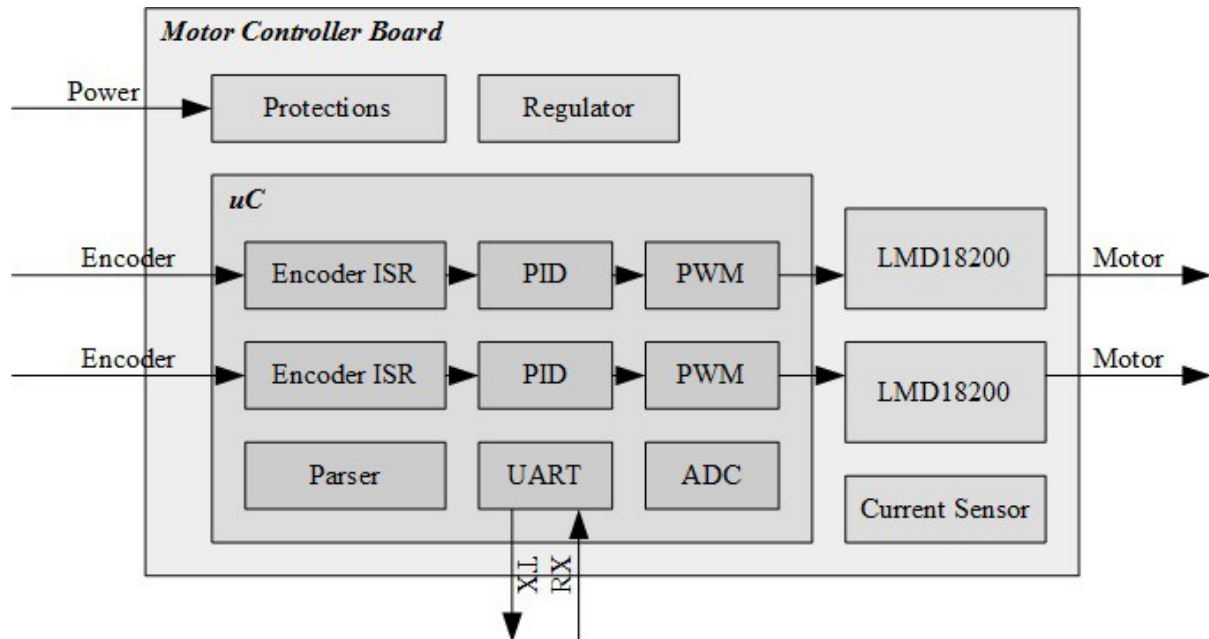


Figure 27: SoW Motor Controller Firmware Architecture

The command parser is a firmware library I'm quite proud of. I found myself writing parsers over and over, so I made a library that matches characters and handle special parametric symbols like the printf function, but limited in scope in order to minimize resource requirement. The result is that one just need to define the commands in a dictionary and feed it with a byte stream. The parser library will handle all the nuances.

The definition of the dictionary can be seen in the code snippet “main.c” - *Parser Dictionary*. Four commands were defined:

- UART_CMD_PING: reset the connection timeout, a safety feature
- UART_CMD_SIGN: asks for the board signature. A string
- UART_CMD_SETVELXY: sets the target speed for the wheels
- UART_CMD_GET_ENCREL: Get the change in encoder reading since last call

Code Snippet “main.c” - Parser Dictionary

```

///-----
///  PARSE COMMANDS
///-----

//Ping command
#define UART_CMD_PING          1
//Sign command. Board is expected to answer with the signature string on UART
#define UART_CMD_SIGN          2
//Set desired forward and sideway velocity

```

```

#define UART_CMD_SETVELXY      10
//Get the relative encoder position since last reading. Cap to S16
#define UART_CMD_GET_ENCREL    21
//Command dictionary. Command IDs 0 and 255 are forbidden
U8 uart_cmd[] =
{
    //Ping: No action. Effect is to reset the connection timeout
    UART_CMD_PING                , 'P', '\0',                                //Sign:
Ask for board signature
    UART_CMD_SIGN                , 'F', '\0',
    //Set X Y: X=forward motion. Y=sidway motion
    UART_CMD_SETVELXY            , 'V', 'X', '%', 'd', 'Y', '%', 'd', '\0',
    //Get relative encoder reading since last call
    UART_CMD_GET_ENCREL          , 'G', 'E', 'T', 'E', 'N', 'C', 'R', '\0',
    //Dictionary Terminator
    '\0'
};
//Board Signature
U8 *board_sign = (U8 *)"TwinDrive";

```

6.1.3 Rosserial Slave

The control board (Figure 28) too was built a while ago to make practice with routing PCBs and to have a breakout board for Atmel microcontrollers with a slot for a LCD. It's a board that served well over the years. A couple of dozens of them have been used over for various projects.

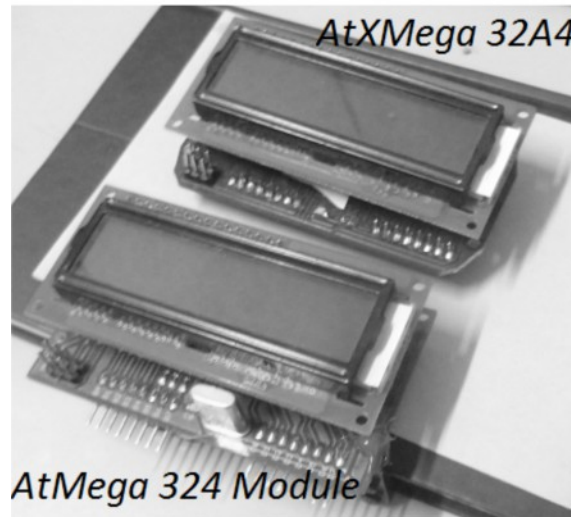


Figure 28: Modules

The board used in SoW features an ATmega 324 and an LCD display. The DIL40 ATmega324 and the TQFP44 ATmega644 belongs to the same family, making porting the code as easy as changing the target platform on the IDE.

The ATmega324 module is mounted on top of a prototyping board (Figure 29) which features connectors, protections and the power regulator for the Raspberry Pi.

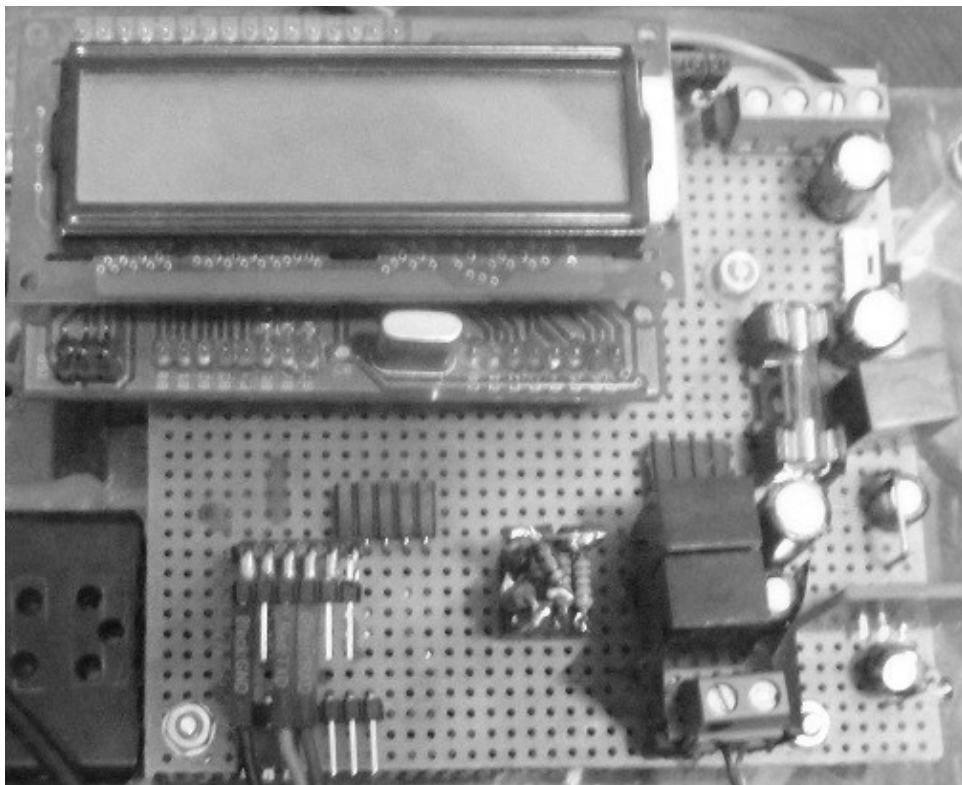


Figure 29: SoW Rosserial Slave Board

The firmware on board of the rosserial Slave board performs most of the functions that would be expected of the custom shield being developed. It act as rosserial Slave, it powers the system. A block schematics of the board and the firmware can be seen in Figure 30.

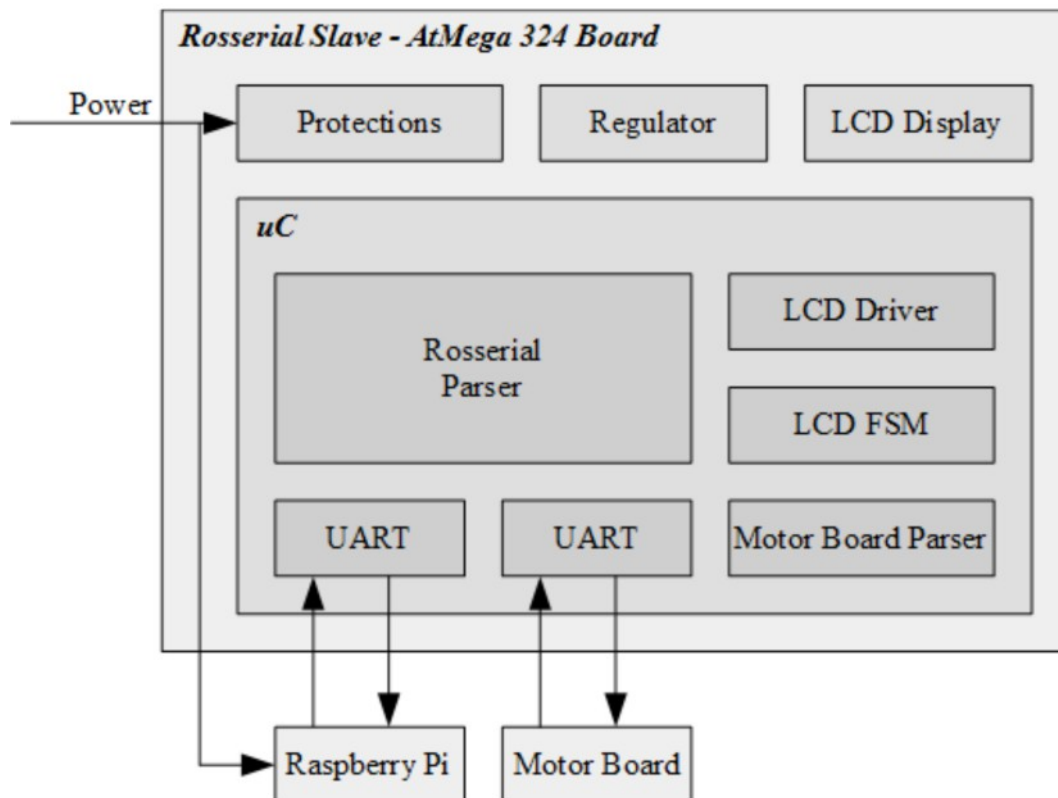


Figure 30: Block Schematics of SoW rosserial Slave Board

The architecture differs from the final intended configuration of the custom shield. A native LCD display is present. The second UART peripheral is used to communicate with the motor board rather than being free. The rosserial slave board lacks an Arduino shield connector and remote bootloader capability.

6.1.4 Differencies between SoW and the custom shield

Seeker of Ways architecture features many of the functions of the custom shield, but has significant differences as well:

- The functions of the motor controller, are performed by a dedicated motor board controlled through the second UART peripheral
- No Bootloader support. It lacks a controllable reset line
- No Arduino Shield connector. Lacks expansion capability
- LCD Display. It is installed on the microcontroller module, and provides an effective runtime debug and monitor tool.

Despite the differences, Seeker of Ways proved to be an invaluable tool to develop the firmware and the applications. The platform has a Long battery life, is sturdy and can cross small obstacle, is fast, and is reliable, having never broken down.

6.2 Software and Firmware

The previous chapters focused on specifications, vision for the project, architecture, component choice and HW implementation. This section describes the code that runs on the microcontroller inside the shield and the high level software framework and application.

6.2.1 Firmware Architecture

Seeker of Ways, the mobile robotic platform built for development, has a different hardware configuration compared to the custom board that is going to be developed. As long as the choice of microcontroller is made correctly, minimal work is required to port the code.

The motor board is based on an ATmega 328 and uses two LMD18200 as H-Bridges. It was built using a prototyping soldered board for the purpose of driving two DC motors with encoders and run PID control loops and communication.

The motor board was chosen because it was available and tested, with the firmware already written. Only minimal work was required to add a few messages to the command parser.

The roserial driver runs on an existing custom PCB based on an ATmega 324, basically a version of the ATmega 644 with less memory. Porting the code can be as simple as changing the target of the cross compiler in the Atmel Studio 7 IDE [34].

Two new firmware modules have been implemented inside the roserial slave:

- The roserial driver that communicates with the Raspberry Pi
- The bootloader.

The custom board is seen by the Raspberry Pi as a ROS node. The communication between the two is done using the roserial protocol. over a UART communication channel.

On top of the driver for the communication protocol, ROS messages themselves that allows communication between the Raspberry Pi and the custom board need to be defined in this phase.

Several baud rates were considered. The roserial UART runs at 128[Kb/s]. Discretization of internal prescalers of the microcontroller means that the true baud rate is 125 [Kbs]. This discretization causes no meaningful data loss.

6.2.2 Rosserial Driver

This thesis requires implementing the roserial protocol inside an ATmega644.

An open source C++ implementation already exists for Arduino. Rather than rewriting the driver from scratch, the existing code has been ported to AVR C++ using AVR Studio 7 as cross compiler.

Porting the library involves rearranging the folder structure of source and header folders, refactoring the names of the .h files and writing a low level class that interfaces the roserial driver with the physical peripherals of the microcontroller.

The low level hardware interface methods that needs implementing in order to port the roserial driver are:

- Ros_driver::read. Get data from the UART
- Ros_driver::write. Send data through the UART peripheral
- Ros_driver::time. Get the number of milliseconds passed since power on
- Subscriber and publisher topics callback functions. This code physically execute ROS messages on the hardware (get encoder position, set motor speed, etc...)

The *read* method from the Ros_driver class, can be seen in the *ros_driver_at324.cpp* snippet. It executes a blocking read from the UART peripheral, reading a character from the UART interface.

Code Snippet "ros_driver_at324.cpp" - read

```
int Ros_driver::read( void )
{
    ///-----
    ///    STATIC VARIABLES
    ///-----

    ///-----
    ///    LOCAL VARIABLES
    ///-----

    //return value
    int ret;

    ///-----
    ///    CHECKS
    ///-----

    ///-----
    ///    INITIALIZATIONS
    ///-----

    ///-----
    ///    BODY
    ///-----

    //Blocking read from UART interface
    ret = get_ch();

    ///-----
```

```

    ///    FINALIZATIONS
    ///-----

    ///-----
    ///    RETURN
    ///-----

    return ret;
}    //end method: read

```

The *write* method from the *Ros_driver* class, can be seen in the *ros_driver_at324.cpp - write* snippet. This method is non blocking. It transfers the output string to a memory. The microcontroller will handle transmission through the UART interface when the system is idle to save resources.

Code Snippet "ros_driver_at324.cpp" - write

```

void Ros_driver::write(uint8_t* data, int length)
{
    ///-----
    ///    STATIC VARIABLES
    ///-----

    ///-----
    ///    LOCAL VARIABLES
    ///-----

    //temp fast counter
    register int t;
    //temp var
    U8 txd;

    ///-----
    ///    CHECKS
    ///-----

    ///-----
    ///    INITIALIZATIONS
    ///-----

    ///-----
    ///    BODY
    ///-----

    //For: scan string
    for (t = 0;t < length;t++)
    {
        //While: the Uart1 HW buffer is not empty
        while (UART1_TX_BUSY());
        //
        txd = data[t];
        cnt_tx++;
        //Write on UART tx buffer.
        UDR1 = txd;
    }

    ///-----
    ///    FINALIZATIONS

```

```

    ///-----
    ///-----
    ///    RETURN
    ///-----

return;
}    //end setter: write | uint8_t *, int

```

The *time* method from the *Ros_driver* class, can be seen in the *ros_driver_at324.cpp - time* snippet. This method returns the number of milliseconds the microcontroller has been up. The variable is stored as a global variable and updated by the main.

Code Snippet "ros_driver_at324.cpp" - time

```

unsigned long Ros_driver::time( void )
{
    ///-----
    ///    STATIC VARIABLES
    ///-----

    ///-----
    ///    LOCAL VARIABLES
    ///-----

    unsigned long ret;

    ///-----
    ///    CHECKS
    ///-----

    ///-----
    ///    INITIALIZATIONS
    ///-----

    ///-----
    ///    BODY
    ///-----

    //fetch global millisecond time
    ret = g_time_ms;

    ///-----
    ///    FINALIZATIONS
    ///-----

    ///-----
    ///    RETURN
    ///-----

    return ret;
}    //end method: time

```

A subscribed topic will execute a user defined callback function. For each topic subscribed, a callback function must be defined that execute the hardware level instructions required. As example, the code snippet *msg_empty* shows a callback function that toggle a led connected to the port PC0 when the message “empty” is received.

Code Snippet "main.cpp" - msg_empty

```
void msg_empty( const std_msgs::Empty& toggle_msg )
{
    ///-----
    ///    STATIC VARIABLES
    ///-----

    ///-----
    ///    LOCAL VARIABLES
    ///-----

    ///-----
    ///    CHECKS
    ///-----

    ///-----
    ///    INITIALIZATIONS
    ///-----

    ///-----
    ///    BODY
    ///-----

    TOGGLE_BIT( PORTC, 0 );

    ///-----
    ///    FINALIZATIONS
    ///-----

    ///-----
    ///    RETURN
    ///-----

    return;
} //end function: msg_empty
```

Making use of the roserial driver inside the microcontroller is remarkably simple. The user need to do as follow:

- Call the constructor for the roserial driver class
- Call the roserial initialization method
- Call the constructors for the subscriber topic classes and link the topics to the callback functions.
- Subscribe the topic classes inside the roserial node class
- Inside the infinite for call the *ros_node.spinOnce* method. It's the runtime method.

The code snippet “*main.cpp*” listed below shows a the minimum code that execute the Rosserial driver. This code registers a topic called toggle_led of type Empty and toggles a LED when the Raspberry Pi sends such message.

Code Snippet “main.cpp”

```
#define EVER (;;)

int main( void )
{
    ///-----
    ///    STATIC VARS
    ///-----

    ///-----
    ///    LOCAL VARS
    ///-----

    //ROSnod class.
    ros::NodeHandle  ros_node;

    ///-----
    ///    VARS INIT
    ///-----

    ///-----
    ///    Rosserial init
    ///-----

    ros_node.initNode();

    ///-----
    ///    Subscriber
    ///-----

    //Construct subscriber topic class
    ros::Subscriber<std_msgs::Empty> sub_empty("toggle_led", &msg_empty );
    //Subscribe topic
    ros_node.subscribe(sub_empty);

    ///-----
    ///    MAIN LOOP
    ///-----

    //Main Loop
    for EVER
    {
        ///Operative user code...

        ///-----
        ///    ROS Handler
        ///-----

        //If: Handle ros messages. Flag generated by interrupt timer.
        if (flags.ros == 1)
        {
            //clear flag
            flags.ros = 0;
            //
        }
    }
}
```

```

        ros_node.spinOnce();
    }
} //end for: for EVER

return 0;
} //end main

```

6.2.3 ROS Topics

A ROS Node can be either a piece of software or hardware. Nodes are seen by ROS as black boxes that sends or receive messages to one another. Messages in ROS are called Topics. The publish subscribe paradigm can be seen in Figure 31.

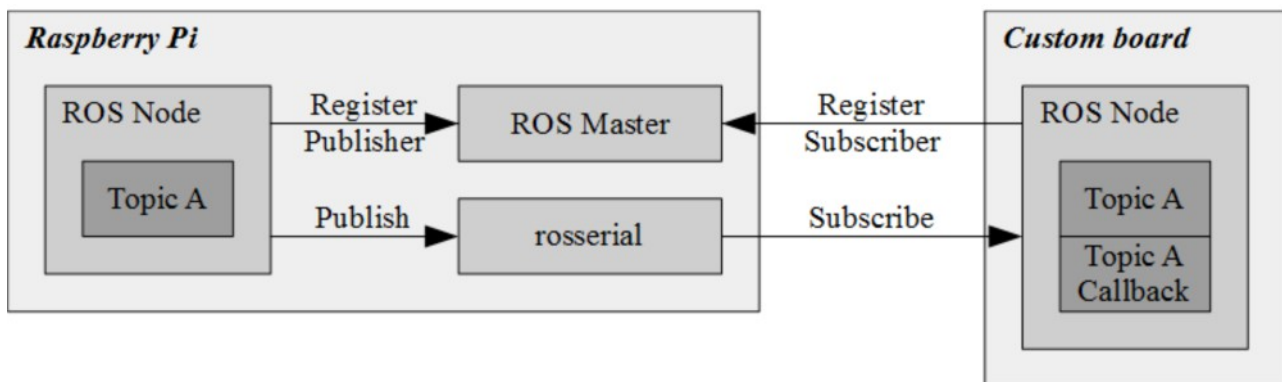


Figure 31: ROS Publish-Subscribe Paradigm

A ROS Topic is a message that is implemented as a C++ class. It provides standardized methods as interface (::init, ::read, ::write, ::time). Nodes that make use of a Topic must individually include the definition to make use of it.

A node that sends a message is called Publisher. A node that receives a message is called Subscriber. ROS API calls allow nodes to register, send and receive.

ROS defines a set of standard messages that can be used out of the box for communication, like the *UInt8* topic that exchange an unsigned 8bit number or an *Figure* topic meant to exchange images. The header definition for the message *UInt8* can be seen in the code snippet "*UInt8.h*" below.

Code Snippet "*UInt8.h*" - *UInt8*

```

namespace std_msgs
{
    class UInt8 : public ros::Msg
    {
    public:
        uint8_t data;

        UInt8():
            data(0)
        {
        }

        virtual int serialize(unsigned char *outbuffer) const
        {
            int offset = 0;
            *(outbuffer + offset + 0) = (this->data >> (8 * 0)) & 0xFF;

```

```

        offset += sizeof(this->data);
        return offset;
    }

    virtual int deserialize(unsigned char *inbuffer)
    {
        int offset = 0;
        this->data = ((uint8_t) (*(inbuffer + offset)));
        offset += sizeof(this->data);
        return offset;
    }

    const char * getType()
    {
        return "std_msgs/UInt8";
    };
    const char * getMD5()
    {
        return "7c8164229e7d2c17eb95e9231617fdee";
    };
};
}

```

A custom user Topic is defined by writing a C++ header file that must be included by both the subscriber and the publisher node. In a similar fashion to the ROS driver, this header must provide standardized interface methods.

ROS is a framework that has been adopted by a large community, and many open source software tools exist that automatically generate the .h definition file for a Topic by providing the description of the payload fields of the message.

The topic has a data field that hold the payload, a serialize and deserialize function that pack/unpack the data into a 8bit vector for communication, a getType() function that generate a string with the name of the message, and a getMD5() functions that saves a precalculated MD5 CRC hash in string form used by the subscriber to check the integrity of the message.

There are many namespaces to chose from when adding a new topic. Commonly used namespaces are:

- `std_msgs`: It holds many basic topics that exchange a single variable (float, UInt16, etc...). UInt8 message belongs to this namespace
- `geometry_msgs`: This namespace is home to topics that describe coordinates. Twist.h is part of this namespace and contains a 3D angle and position
- `sensor_msgs`: used to exchange sensor data. Figure.h belongs to this namespace and exchanges an image, usually snapped from an on-board camera

6.2.4 Custom Board Messages

The design philosophy for the system described in this thesis is to make the life of the robot maker easier. Providing a set of ROS Topics ideally suited for small robotic mobile platforms is one way in which this is accomplished.

Advanced users will of course be able to add their own custom topics to cater to their specific application, but this requires knowledge of the intricacies of ROS.

A set of useful messages and example open source projects have been provided that allow an inexperienced robot maker to make the HotBlack powered platform to move out of the box, just by running the example project on the browser based HotBlack IDE.

First and foremost, messages are needed that allows to move the robot. Depending on the structure of the platform, there are several kind of motions:

- Wheeled robots: Move in a 2D space. Require two Cartesian positions and one angle.
- Aeromodels: Move in a 3D space. Require three Cartesian positions and three angles.
- Robotic arms. End effector move in a 3D space. Require three Cartesian positions and three angles.

A standard message exists in ROS that already describes a 3D Euler position featuring three Cartesian positions and three angles, *Twist.h*. This Topic belongs to the *geometry_msgs* namespace. Its code can be seen in the code snippet “*Twist.h*” down below.

Code Snippet “Twist.h”

```
namespace geometry_msgs
{
    class Twist : public ros::Msg
    {
    public:
        geometry_msgs::Vector3 linear;
        geometry_msgs::Vector3 angular;

        Twist():
            linear(),
            angular()
        {
        }

        virtual int serialize(unsigned char *outbuffer) const
        {
            int offset = 0;
            offset += this->linear.serialize(outbuffer + offset);
            offset += this->angular.serialize(outbuffer + offset);
            return offset;
        }

        virtual int deserialize(unsigned char *inbuffer)
        {
            int offset = 0;
            offset += this->linear.deserialize(inbuffer + offset);
            offset += this->angular.deserialize(inbuffer + offset);
            return offset;
        }
    }
```



```

const char * getType()
{
    return "geometry_msgs/Twist";
};
const char * getMD5()
{
    return "9f195f881246fdfa2798d1d3eebca84a";
};
};
}

```

The Twist Topic covers Aeromodels and Robotic Arms quite well, but the message is unsuited for wheeled platforms.

Twist exchanges six floating point numbers. A wheeled platform would need at most three, and on top of that, an 8bit microcontroller like the ATmega 644 lacks the hardware to natively handles floating point calculations. Using floats involves significant overheads. On top of all of that, exchanging six floating point numbers of four byte each one hundred times a second would use 19.2Kb/s of bandwidth of the 128Kb/s.

A more efficient set of custom ROS messages is needed for wheeled platforms. The messages have been collected in a custom namespace named after the HotBlack framework: *hb_core_msgs*.

A total of four custom ROS messages have been defined:

- Velocity2D: Controls the speed of two wheels
- Pose2D: Identify the position of the robot in two dimensions
- Odom2D: Returns the relative position of the robot in two dimensions
- Odom2DExtended: Returns the absolute position of the robot in two dimensions

The first message is one that allows to set the speed of the wheels. After careful consideration the *Velocity2D* message has been crafted. it uses two signed 16bit integers in fixed point to control the speed of two wheels. The definition of the Topic can be seen in the code snippet "*Velocity2D.h*" down below.

Code Snippet "Velocity2D.h"

```

namespace hb_core_msgs
{
class Velocity2D : public ros::Msg
{
    public:
        int16_t linear;
        int16_t angular;

        Velocity2D():
            linear(0),
            angular(0)
        {
        }

        virtual int serialize(unsigned char *outbuffer) const
        {
            int offset = 0;
            union
            {

```

```

        int16_t real;
        uint16_t base;
    } u_linear;
    u_linear.real = this->linear;
    *(outbuffer + offset + 0) = (u_linear.base >> (8 * 0)) & 0xFF;
    *(outbuffer + offset + 1) = (u_linear.base >> (8 * 1)) & 0xFF;
    offset += sizeof(this->linear);
    union
    {
        int16_t real;
        uint16_t base;
    } u_angular;
    u_angular.real = this->angular;
    *(outbuffer + offset + 0) = (u_angular.base >> (8 * 0)) & 0xFF;
    *(outbuffer + offset + 1) = (u_angular.base >> (8 * 1)) & 0xFF;
    offset += sizeof(this->angular);
    return offset;
}

virtual int deserialize(unsigned char *inbuffer)
{
    int offset = 0;
    union
    {
        int16_t real;
        uint16_t base;
    } u_linear;
    u_linear.base = 0;
    u_linear.base |= ((uint16_t) (*(inbuffer + offset + 0))) << (8 * 0);
    u_linear.base |= ((uint16_t) (*(inbuffer + offset + 1))) << (8 * 1);
    this->linear = u_linear.real;
    offset += sizeof(this->linear);
    union
    {
        int16_t real;
        uint16_t base;
    } u_angular;
    u_angular.base = 0;
    u_angular.base |= ((uint16_t) (*(inbuffer + offset + 0))) << (8 *
0);
    u_angular.base |= ((uint16_t) (*(inbuffer + offset + 1))) << (8 *
1);

    this->angular = u_angular.real;
    offset += sizeof(this->angular);
    return offset;
}

const char * getType()
{
    return "hb_core_msgs/Velocity2D";
};
const char * getMD5()
{
    return "51fd6b987aa2eb4fc97aebca8aaf424e";
};
};
}

```

The second message defines the absolute position of the robot. It uses three 32bit signed int in fixed point for the X and Y position and one for the absolute heading. This message allows the Raspberry Pi to move the robot to a position or to receive the estimated position of the platform in space. The definition of the Topic can be seen in the code snippet “*Pos2D.h*” down below.

Code Snippet “Pos2D.h”

```
namespace hb_core_msgs
{
    class Pose2D : public ros::Msg
    {
    public:
        int32_t x;
        int32_t y;
        int32_t theta;

        Pose2D():
            x(0),
            y(0),
            theta(0)
        {
        }

        virtual int serialize(unsigned char *outbuffer) const
        {
            int offset = 0;
            union
            {
                {
                    int32_t real;
                    uint32_t base;
                } u_x;
                u_x.real = this->x;
                *(outbuffer + offset + 0) = (u_x.base >> (8 * 0)) & 0xFF;
                *(outbuffer + offset + 1) = (u_x.base >> (8 * 1)) & 0xFF;
                *(outbuffer + offset + 2) = (u_x.base >> (8 * 2)) & 0xFF;
                *(outbuffer + offset + 3) = (u_x.base >> (8 * 3)) & 0xFF;
                offset += sizeof(this->x);
            } union
            {
                {
                    int32_t real;
                    uint32_t base;
                } u_y;
                u_y.real = this->y;
                *(outbuffer + offset + 0) = (u_y.base >> (8 * 0)) & 0xFF;
                *(outbuffer + offset + 1) = (u_y.base >> (8 * 1)) & 0xFF;
                *(outbuffer + offset + 2) = (u_y.base >> (8 * 2)) & 0xFF;
                *(outbuffer + offset + 3) = (u_y.base >> (8 * 3)) & 0xFF;
                offset += sizeof(this->y);
            } union
            {
                {
                    int32_t real;
                    uint32_t base;
                } u_theta;
                u_theta.real = this->theta;
                *(outbuffer + offset + 0) = (u_theta.base >> (8 * 0)) & 0xFF;
                *(outbuffer + offset + 1) = (u_theta.base >> (8 * 1)) & 0xFF;
                *(outbuffer + offset + 2) = (u_theta.base >> (8 * 2)) & 0xFF;
                *(outbuffer + offset + 3) = (u_theta.base >> (8 * 3)) & 0xFF;
                offset += sizeof(this->theta);
            }
        }
    };
}
```

```

        return offset;
    }

virtual int deserialize(unsigned char *inbuffer)
{
    int offset = 0;
    union
    {
        int32_t real;
        uint32_t base;
    } u_x;
    u_x.base = 0;
    u_x.base |= ((uint32_t) (*(inbuffer + offset + 0))) << (8 * 0);
    u_x.base |= ((uint32_t) (*(inbuffer + offset + 1))) << (8 * 1);
    u_x.base |= ((uint32_t) (*(inbuffer + offset + 2))) << (8 * 2);
    u_x.base |= ((uint32_t) (*(inbuffer + offset + 3))) << (8 * 3);
    this->x = u_x.real;
    offset += sizeof(this->x);
    union
    {
        int32_t real;
        uint32_t base;
    } u_y;
    u_y.base = 0;
    u_y.base |= ((uint32_t) (*(inbuffer + offset + 0))) << (8 * 0);
    u_y.base |= ((uint32_t) (*(inbuffer + offset + 1))) << (8 * 1);
    u_y.base |= ((uint32_t) (*(inbuffer + offset + 2))) << (8 * 2);
    u_y.base |= ((uint32_t) (*(inbuffer + offset + 3))) << (8 * 3);
    this->y = u_y.real;
    offset += sizeof(this->y);
    union
    {
        int32_t real;
        uint32_t base;
    } u_theta;
    u_theta.base = 0;
    u_theta.base |= ((uint32_t) (*(inbuffer + offset + 0))) << (8 * 0);
    u_theta.base |= ((uint32_t) (*(inbuffer + offset + 1))) << (8 * 1);
    u_theta.base |= ((uint32_t) (*(inbuffer + offset + 2))) << (8 * 2);
    u_theta.base |= ((uint32_t) (*(inbuffer + offset + 3))) << (8 * 3);
    this->theta = u_theta.real;
    offset += sizeof(this->theta);
    return offset;
}

const char * getType()
{
    return "hb_core_msgs/Pose2D";
};
const char * getMD5()
{
    return "2bffa1127cd10ca02349c9c72c72be56e";
};
};
}

```

The last messages are meant to exchange telemetry data. Rather than make a one fit all message, two topics have been conceived. One optimized for bandwidth, one optimized for completeness of information. This allows the control program to optimize bandwidth usage.

Telemetry data are used to receive either the differential encoder readings or absolute encoder readings with a timestamp. They can be used to estimate the current position and heading of the platform.

The Odom2D message is a short message meant to exchange a differential encoder or position reading in respect to the previous transmission.

Odom2D reuses the Velocity2D topic to encapsulate the differential telemetry reading. A timestamp is needed since the time jitter of the roserial driver is unknown and varies depending on the workload of the microcontroller, the residual bandwidth of the communication channel and a random jitter due to the discretization of the internal timers.

Accurate timing of the position is required by many navigation algorithms. The definition of the Topic can be seen in the code snippet “*Odom2D.h*” down below.

Code Snippet “Odom2D.h”

```
namespace hb_core_msgs
{
    class Odom2D : public ros::Msg
    {
    public:
        uint32_t time_stamp;
        hb_core_msgs::Velocity2D velocity;

        Odom2D() :
            time_stamp(0),
            velocity()
        {
        }

        virtual int serialize(unsigned char *outbuffer) const
        {
            int offset = 0;
            *(outbuffer + offset + 0) = (this->time_stamp >> (8 * 0)) & 0xFF;
            *(outbuffer + offset + 1) = (this->time_stamp >> (8 * 1)) & 0xFF;
            *(outbuffer + offset + 2) = (this->time_stamp >> (8 * 2)) & 0xFF;
            *(outbuffer + offset + 3) = (this->time_stamp >> (8 * 3)) & 0xFF;
            offset += sizeof(this->time_stamp);
            offset += this->velocity.serialize(outbuffer + offset);
            return offset;
        }

        virtual int deserialize(unsigned char *inbuffer)
        {
            int offset = 0;
            this->time_stamp = ((uint32_t) (*(inbuffer + offset)));
            this->time_stamp |= ((uint32_t) (*(inbuffer + offset + 1))) << (8 *
1);
            this->time_stamp |= ((uint32_t) (*(inbuffer + offset + 2))) << (8 *
2);
            this->time_stamp |= ((uint32_t) (*(inbuffer + offset + 3))) << (8 *
3);
            offset += sizeof(this->time_stamp);
            offset += this->velocity.deserialize(inbuffer + offset);
        }
    };
}
```

```

        return offset;
    }

    const char * getType()
    {
        return "hb_core_msgs/Odom2D";
    };
    const char * getMD5()
    {
        return "f7b5df72ac17f89fc59d9683e7ee53e2";
    };
};
}

```

Odom2DExtended allows the Raspberry Pi, to get all telemetry data with a single topic. It includes a timestamp, the Odom2D with an added Pose2D payload. This message is complete in terms of information provided, but expensive in terms of bandwidth used. The definition of the Topic can be seen in the code snippet “*Odom2DExtended.h*” down below.

Code Snippet "Odom2DExtended.h"

```

namespace hb_core_msgs
{
    class Odom2DExtended : public ros::Msg
    {
    public:
        uint32_t time_stamp;
        hb_core_msgs::Velocity2D velocity;
        hb_core_msgs::Pose2D pose;

        Odom2DExtended():
            time_stamp(0),
            velocity(),
            pose()
        {
        }

        virtual int serialize(unsigned char *outbuffer) const
        {
            int offset = 0;
            *(outbuffer + offset + 0) = (this->time_stamp >> (8 * 0)) & 0xFF;
            *(outbuffer + offset + 1) = (this->time_stamp >> (8 * 1)) & 0xFF;
            *(outbuffer + offset + 2) = (this->time_stamp >> (8 * 2)) & 0xFF;
            *(outbuffer + offset + 3) = (this->time_stamp >> (8 * 3)) & 0xFF;
            offset += sizeof(this->time_stamp);
            offset += this->velocity.serialize(outbuffer + offset);
            offset += this->pose.serialize(outbuffer + offset);
            return offset;
        }

        virtual int deserialize(unsigned char *inbuffer)
        {
            int offset = 0;
            this->time_stamp = ((uint32_t) (*(inbuffer + offset)));
            this->time_stamp |= ((uint32_t) (*(inbuffer + offset + 1))) << (8 *
1);
            this->time_stamp |= ((uint32_t) (*(inbuffer + offset + 2))) << (8 *

```

```

2);
    this->time_stamp |= ((uint32_t) (*(inbuffer + offset + 3))) << (8 *
3);
    offset += sizeof(this->time_stamp);
    offset += this->velocity.deserialize(inbuffer + offset);
    offset += this->pose.deserialize(inbuffer + offset);
    return offset;
}

const char * getType()
{
    return "hb_core_msgs/Odom2DExtended";
};
const char * getMD5()
{
    return "830beb884a3d5949d8e4c94af38fff87";
};
};
}

```

The messages described above are topic for which example projects and open source firmware has been released. They are meant for a robot maker to get started with mobile robotics in a matter of minutes and to be able to control the robotic platform even without a deep understanding of the architecture of the frameworks involved. An extremely valuable feature that promotes adoption.

Robot makers experienced with ROS and firmware programming can make their own custom messages to cater their own custom application or add code and existing topics developed by other ROS developers.

The added value of the HotBlack software framework is the ability to easily share code and projects, so that other robot makers can make use of snippets and application developed by more experienced designer, and built their own application on top of that.

6.3 HotBlack Software Framework

The custom board developed in this thesis is only one of the components required to realize the overall vision for the project.

A framework that provides IoT functionalities, easy code sharing and a streamlined online IDE is required in order to make the development of a robotic platform as fast and easy as possible.

HotBlack Robotics, the start-up this thesis has been developed alongside with, has developed the software and cloud infrastructure. The HotBlack Software Framework.

To engage in robotics using the HotBlack Software Framework, the user only need three things to get started:

- An SD card with the HBrain image burned on
- A Raspberry Pi model 2 or 3 and its power source
- A wireless router connected to the internet

To prime the system, the user first set up the Wi-Fi network to a predefined SSID and Password. The HBrain image always tries to connect to this network the first time. Wi-Fi configuration is as follow:

SSID: dotbot

PWD: dotbot@polito

Once the Wi-Fi is setup, the user opens a common web browser and goes to the following web address:

<http://cloud.hotblackrobotics.com/cloud/index>

There, the web page interfaces with the HotBlack cloud infrastructure. The user registers themselves on the site.

The Raspberry Pi should take a few tens of seconds to boot up, connect to the Wi-Fi network and establish a connection with a remote server. Once the handshake is complete, the robot will show up on the web page as shown in Figure 32.

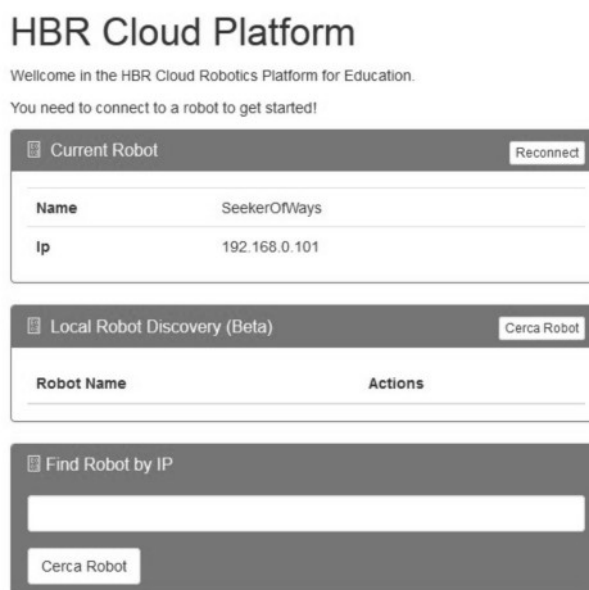
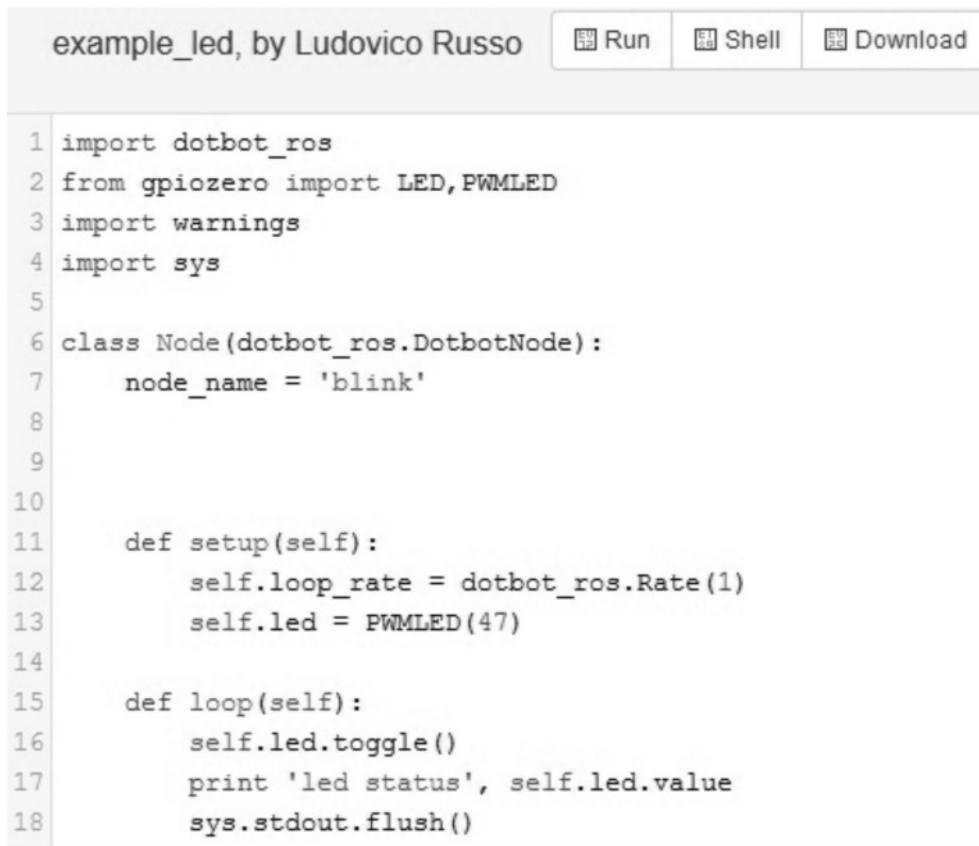


Figure 32: HotBlack Cloud Robot Find

It can be seen that Seeker Of Ways was found in the local network. The robot has a default name initially. Both the default Wi-Fi network and the name of the platform can be changed by accessing a configuration page on the website. No editing of configuration a file is required.

The user can now proceed on the 'Sketches' tab on the web page. There, the user can program sketches using Python and run them.

A few templates and example projects are provided by HotBlack developers for first timers. One such project is example_led, that toggles pin 47 of the Raspberry Pi GPIO. The source code of the sketch can be seen in Figure 33. By pressing 'Run', the user can get the first program running.

The image shows a web interface for a robot's sketch editor. At the top, there's a header bar with the title "example_led, by Ludovico Russo" and three buttons: "Run", "Shell", and "Download". Below the header is a text area containing Python code for an LED blink sketch. The code is numbered from 1 to 18 on the left side. The code imports dotbot_ros, gpiozero, warnings, and sys. It defines a class Node that inherits from dotbot_ros.DotbotNode. The class has a node_name attribute set to 'blink'. It has two methods: setup(self) and loop(self). The setup method sets the loop_rate to 1 Hz and initializes the LED on pin 47. The loop method toggles the LED, prints its status, and flushes the stdout buffer.

```
1 import dotbot_ros
2 from gpiozero import LED, PWMLED
3 import warnings
4 import sys
5
6 class Node(dotbot_ros.DotbotNode):
7     node_name = 'blink'
8
9
10
11     def setup(self):
12         self.loop_rate = dotbot_ros.Rate(1)
13         self.led = PWMLED(47)
14
15     def loop(self):
16         self.led.toggle()
17         print 'led status', self.led.value
18         sys.stdout.flush()
```

Figure 33: HBR Example Sketch LED blink

The HotBlack framework provides a powerful tool that maps the complete Node and Topic list of the ROS network. The user can then see a map of ROS nodes currently active by going into the 'node' tab (Figure 34).

Ros Console

Nodes List		
Name	Actions	
/rosapi	kill node	
/SeekerOfWays/blink	kill node	
/rosout	kill node	
/SeekerOfWays/hbr_driver	kill node	
/rosbridge_websocket	kill node	
/serial_node	kill node	

Topics List		
Name	Type	Actions
/rosout	rosgraph_msgs/Log	Echo
/rosout_agg	rosgraph_msgs/Log	Echo
/msg_cnt_u8	std_msgs/UInt8	Echo
/SeekerOfWays/driver	std_msgs/UInt8	Echo
/diagnostics	diagnostic_msgs/DiagnosticArray	Echo
/cmd_vel	hb_core_msgs/Velocity2D	Echo

Figure 34: HBR ROS Node Map

After running the sketch, it can be seen that the ROS node */SeekerOfWays/blink* is present in the list of active nodes (Figure 34). That's the same name that was given to the node in the sketch (Figure 33) with the line: `node_name = 'blink'`. 'Kill node' allows the user to terminate a Sketch.

This is it. Just like that, in a matter of minutes from the first boot, the user has been able to execute a program on the platform.

This framework and the abstraction layers it provides, allow the user to get a Python program, running on a ROS framework, running on a Linux operating system, running on a Raspberry Pi, from a common browser, without so much as editing a Linux configuration file.

This is an exceedingly effective approach that massively lowers the entry barrier of

developing on an ARM based Linux/ROS platform. This is as simple as things can get. And ease of use is one of the main drives for adoption.

HotBlack Robotics provides a series of Python libraries and example sketches that perform a variety of functions. Image processing, telegram bots or remote control from on-screen joystick are just a few examples. It is also possible to write html pages to control the robot.

The user can count on a good selection of templates to start from to implement their own applications, and things can only get better as adoption increases and a thriving community shares more and more open source projects.

Sketches can be easily shared and imported. With this, all the software tools required to achieve the objective of simplifying the life of the robot makers are in place.

7. Test Boards

The test board and the components arrived by the time the bulk of the firmware was ready. The next step is to solder the H-Bridge board and the Supply board, test them and find out bugs in the design.

7.1 Supply Board: Testing

The voltage regulator is a critical part of the design. Once soldered, the first test was to power the test board through a small resistor in series with the power supply to ensure that it wouldn't blow up on activation. The test failed. The board displayed a near short circuit in the input.

After some debug the problem became obvious. The power NMOS of the SEPIC regulator has drain and source inverted. It's a mistake that slipped through during the design the layout of the package of the power MOS.

The solution (Figure 35) is to remove the component and use flying wires to connect the power NMOS the right way.

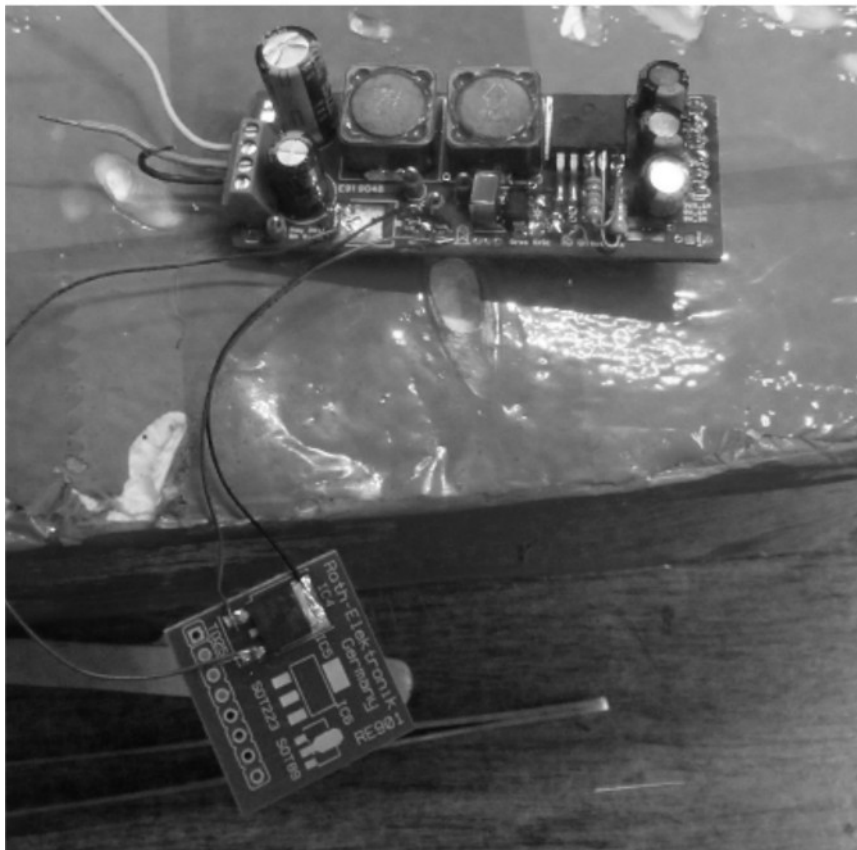


Figure 35: Test Power Board Power MOS Correction

The power test was repeated with the power MOS fixed and this time was a success.

The SEPIC regulator generated an intermediate voltage of 5.6V in output (Figure 36). An intermediate voltage of 5.6V falls short of the 5.7V required. Correcting this problem was easy, as it just required soldering different resistors in the SEPIC feedback network.

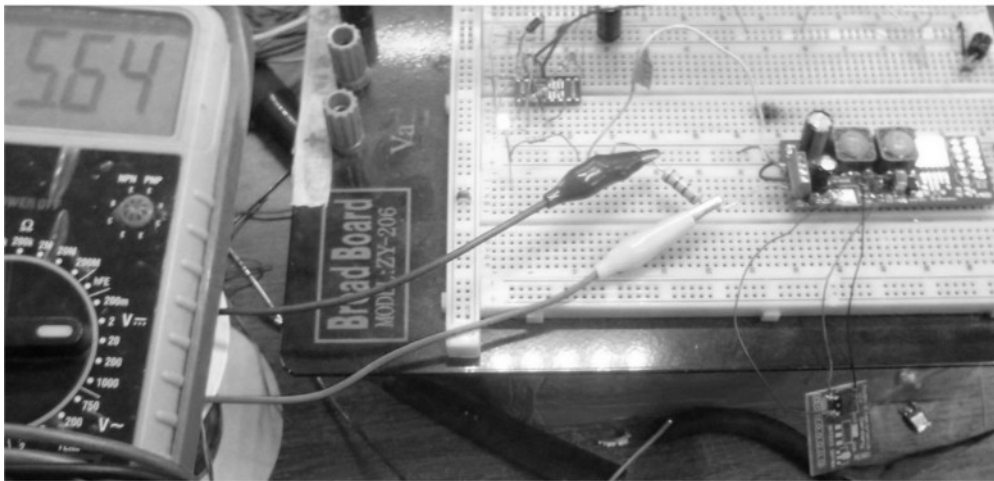


Figure 36: Testing of the Test Supply Board

The next step was to measure the output of the LDO regulators. The 3.3V regulator worked just fine, but the 5V regulator was outputting about 4.7V.

The problem was that the 1117 series regulators used are PMOS style LDO and need 1.1V to make the linear regulation. Solution was to either raise the intermediate voltage or to use a better component.

The output of the 5V 3A regulator is fine. The chosen LDO for the Raspberry Pi is a NMOS style linear regulator that requires only about 600mV at 3A to make the regulation.

The final step was to test the system under load. The supply board was used to power Seeker of Ways, both the Raspberry Pi and the rosserial Slave (Figure 37).

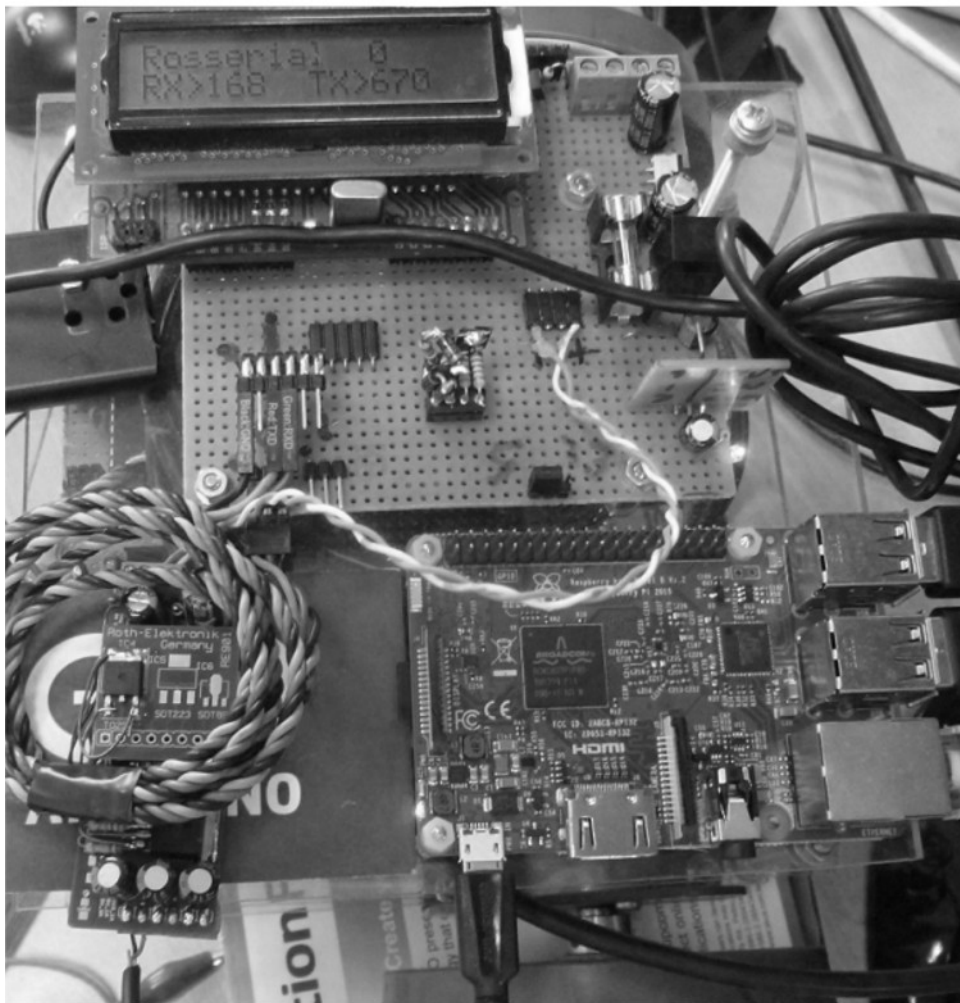


Figure 37: Test Supply Board with Seeker of Ways

The test went without an hitch. The robot worked perfectly while powered by the new voltage regulator.

7.2 Test H-Bridge Board

The assembled test board for the H-Bridge component can be seen in Figure 38.

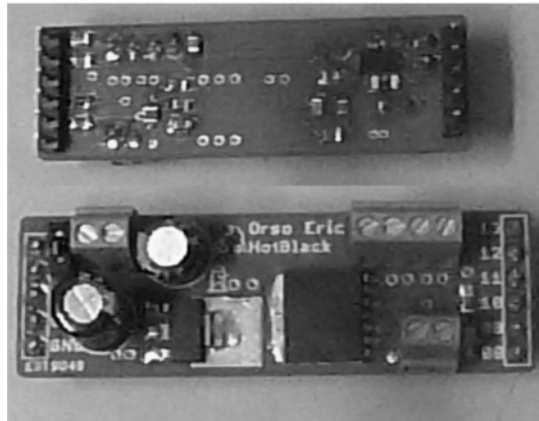


Figure 38: Test H-Bridge Board

The first test, the power test was successful. Electrical levels were right and there were no signs of overheating.

The next step was to plug the board on top of an Arduino Uno (Figure 39). Again, the power test went without an hitch. The Arduino board was able to operate correctly while powered through the test board.

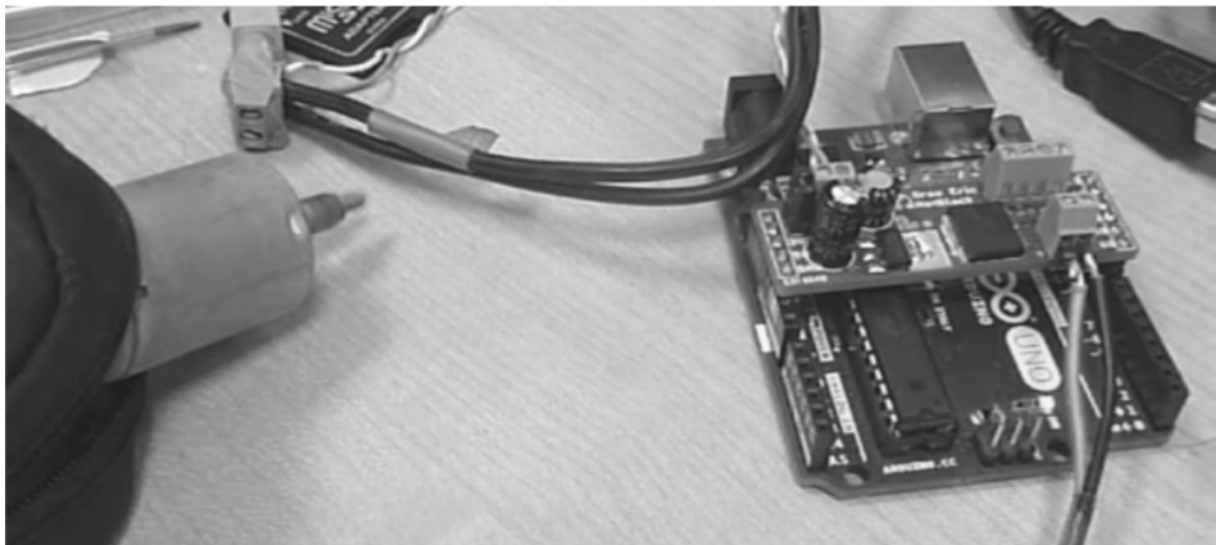


Figure 39: Test H-Bridge board using an Arduino

Finally, an Arduino Sketch was written to control a small DC motor. Even this final test was successful. Both the board and the H-Bridge component work as intended.

The Arduino sketch used to test the board can be seen in the snippet "Test_hb.ino".

```
Code Snippet: "Test_hb.ino"
//HotBlack Minishield Single Motor Controller
//10, OC1B | PWM+
//9, OC1A | PWM-
//8 | LED

#define MIN 5
#define MAX 20

extern void set_pwm( int16_t spd );

bool f_led = 0;

//upcount
bool f_cnt = 0;
int16_t cnt = 0;

uint16_t cnt_led;
uint8_t pwma, pwmb;

void setup()
{
    // put your setup code here, to run once:
    //analogWrite( 9, 127 );
    //analogWrite( 10, 140 );

    set_pwm( +0 );
}

void loop()
{
    // put your main code here, to run repeatedly:
    //Sleep(1);
    delay(10);

    if (f_cnt == 0)
    {
        cnt++;
    }
    else
    {
        cnt--;
    }

    if (cnt > +255)
    {
        cnt = +255;
        f_cnt = 1;
    }
    else if (cnt < -255)
    {
        cnt = -255;
        f_cnt = 0;
    }
}
```



```

set_pwm( cnt );

cnt_led++;
if (cnt_led > 100)
{
    cnt_led = 0;
    digitalWrite( 8, f_led );
    f_led = !f_led;
}

}

void set_pwm( int16_t spd )
{
    uint8_t pwm_plus, pwm_minus;
    bool f_dir;
    //
    if (spd < 0)
    {
        spd = -spd;
        f_dir = 1;
    }
    else
    {
        f_dir = 0;
    }

    if (spd < MIN)
    {
        pwm_minus = 0;
        pwm_plus = 0;
    }
    else if (spd > MAX)
    {
        pwm_minus = 0;
        pwm_plus = MAX;
    }
    else
    {
        pwm_minus = 0;
        pwm_plus = spd;
    }

    if (f_dir == 0)
    {
        analogWrite( 9, pwm_minus );
        analogWrite( 10, pwm_plus );
    }
    else
    {
        analogWrite( 9, pwm_plus );
        analogWrite( 10, pwm_minus );
    }
}

```

The sketch generates two PWM ramps to move the motor at a linearly increasing speed in one direction then, decelerate until the motor reverse direction and reach max speed on the opposite direction. After that, the cycle starts over. PWM signals that are too small are clipped to zero as duty cycles that are too smalls can't fully turn on the high side switch.

8. Custom Raspberry Pi Shield Design

Chapter 1 was about objectives and vision for this project

Chapter 2 was about a research on the problems and existing solutions

Chapter 3 was about the specifications for the custom shield

Chapter 4 was about the overall architecture of the system

Chapter 5 was about schematics and layout for the power system and the motors

Chapter 6 was about building a test platform to write and test the firmware and test the architecture

Chapter 7 was about validating the test boards

By this point the bulk of the firmware is written and tested in conjunction with the HotBlack Software Framework. and the critical part of the shield hardware sections have been tested independently on their own test boards.

What's left is to use the feedback obtained to design the full schematics of the custom shield. To route the PCB, manufacture it and test it. Finally, to develop some meaningful applications that take advantage of the system and the HotBlack framework and cloud infrastructure.

8.1 Custom Shield Schematics

The features, specifications, interfaces, and internal configuration of the single blocks have been explored in details in previous chapters. Using the information gathered from the tests, the final hardware configuration of the board with the hardware components required and the interfaces can be seen in Figure 40.

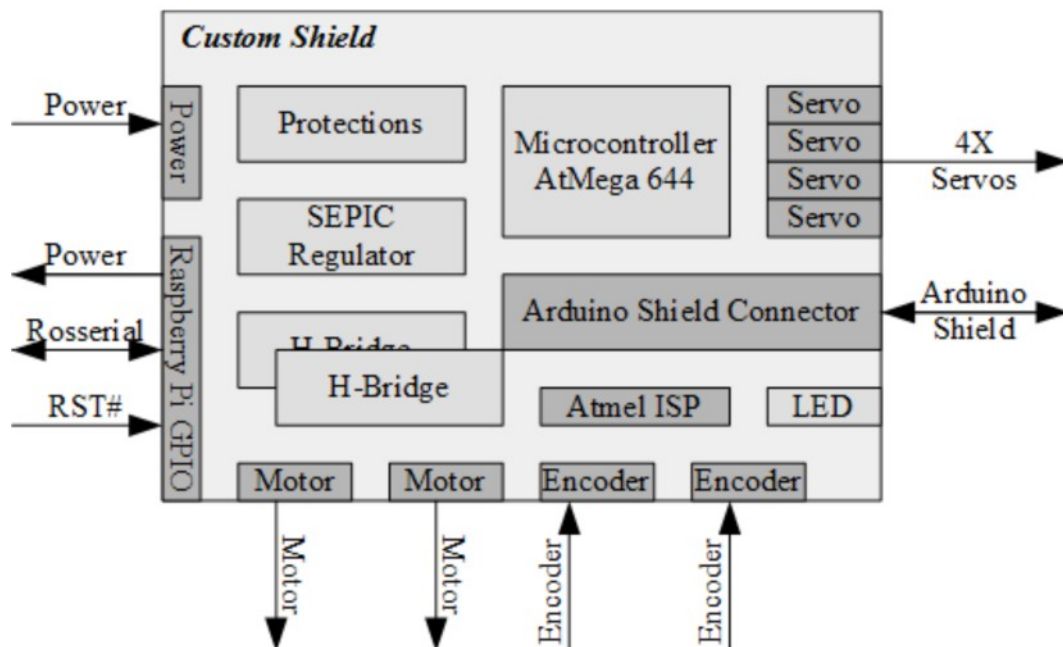


Figure 40: HW Architecture of the custom Raspberry Pi shield

The architecture of the custom shield is comprised of the following sections:

- Battery connector and protection stage
- SEPIC Regulator
- LDO Regulators and filters
- H-Bridges
- Microcontroller, ISP programming connector and LEDs
- Raspberry Shield screw holes and GPIO connector
- Arduino Shield connector
- Servomotor connectors
- Encoder connectors

The schematics is somewhat large, so it will be shown and described in sections rather than as a whole for reading convenience.

8.1.1 Battery connector and Protections

This part of the schematics (Figure 41) deals with the battery connector and with the input protections stage.

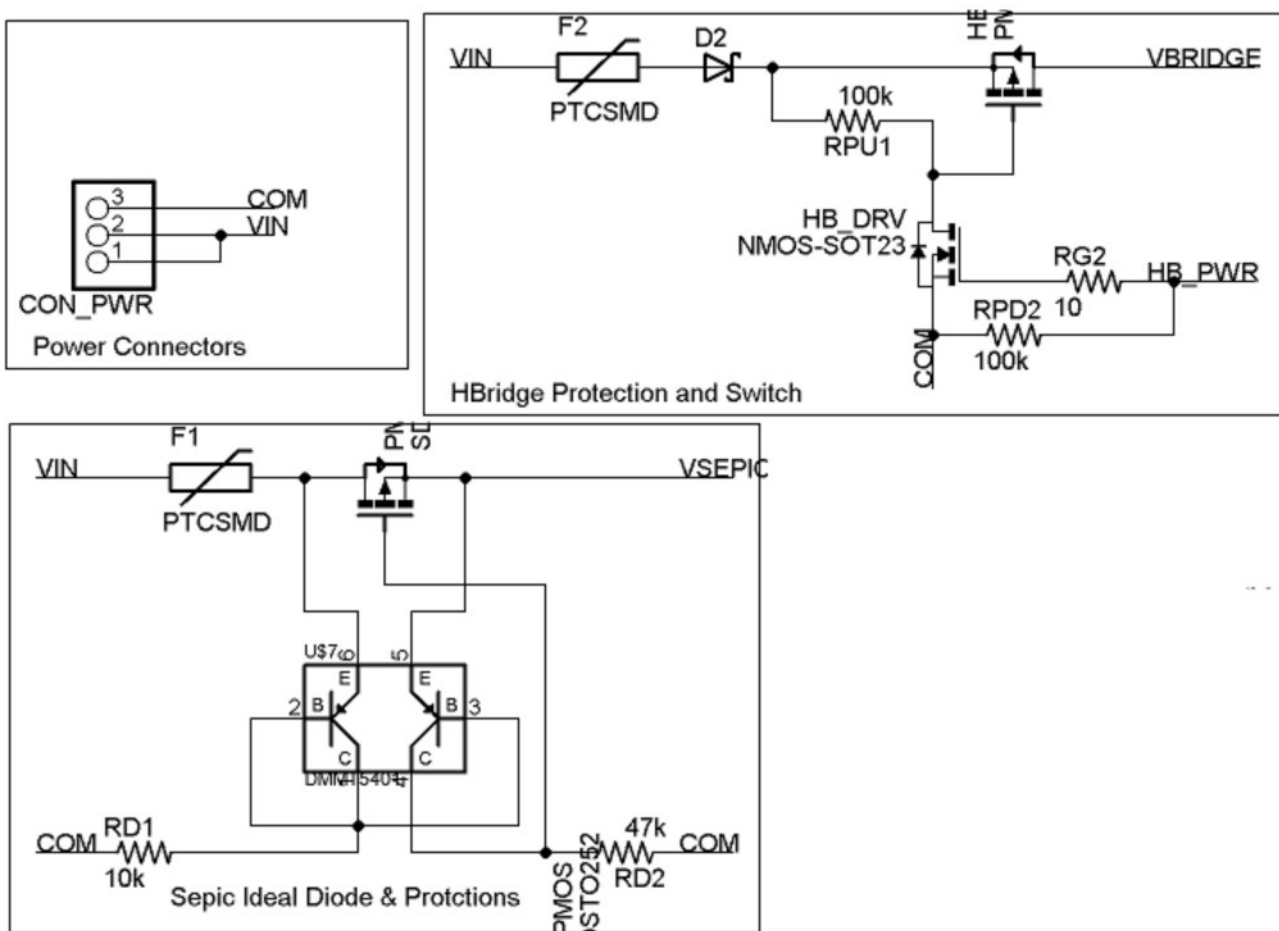


Figure 41: Shield Schematics - Battery Connector and Protections

Inputs:

- CON_PWR: Power source screw connector
- HB_PWR: Logic line from uC. Cut power to H-Bridges.

Outputs:

- COM: 0V. Common voltage sink.
- VSEPIC: Input voltage for the SEPIC regulator
- VBRIDGE: Input voltage for the H-Bridges

A screw terminal connector is used for the wires coming from the power source, usually a battery. The footprint used has three terminal spaced 2.54mm. This allows both 2.54mm screw connectors or 5.12mm screw connectors to be used. Because of the currents involved, using a 5.12mm screw connector is advised.

The H-Bridge features a resettable fuse, a diode that protects against polarity inversion and a solid state relay that allows the microcontroller to cut off power to the H-Bridges altogether through the HB_PWR line. If HB_PWR is left undriven by the microcontroller, the H-Bridge is cut off by default.

VSEPIC is protected by a resettable fuse and by an ideal diode that minimizes voltage drop and power loss in the protection stage. A feature that maximizes efficiency.

8.1.2 SEPIC Regulator

The SEPIC Regulator (Figure 42) has been discussed in details in Chapter 5.1. Minor corrections to the value of components and the schematics have been applied taking into account feedback obtained from the test board.

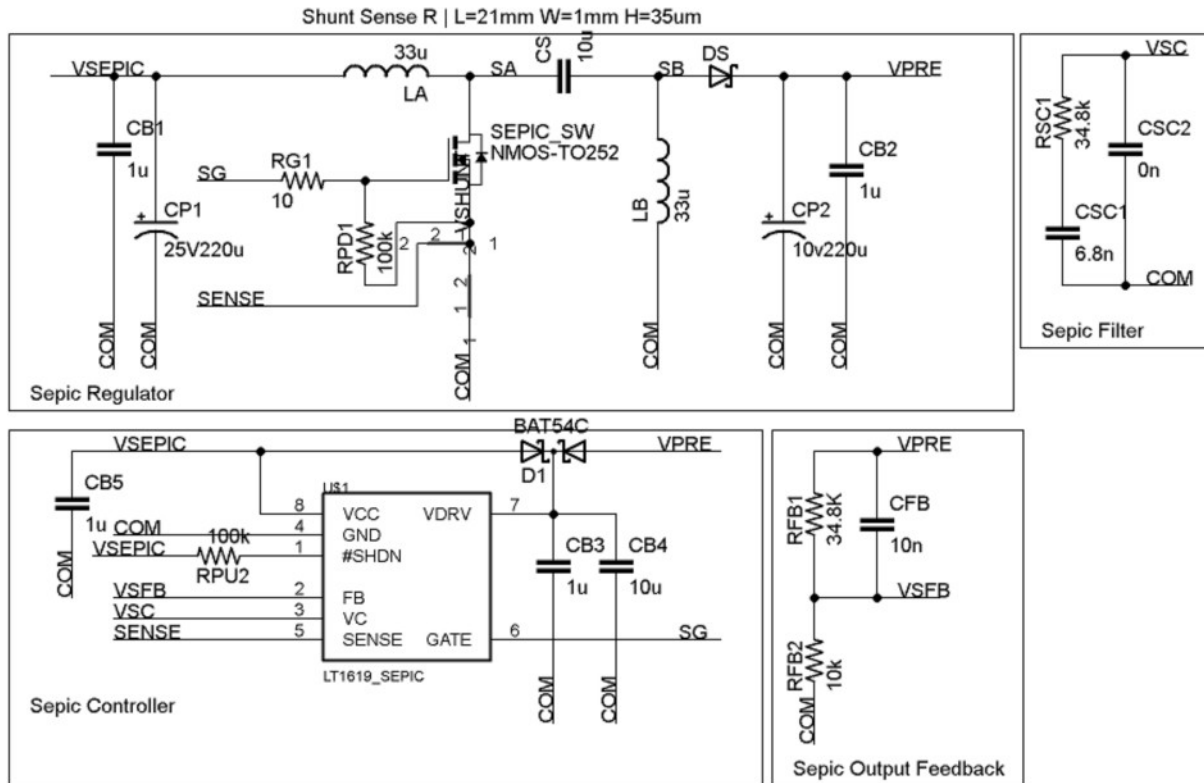


Figure 42: Shield Schematics - SEPIC Regulator

Inputs:

- VSEPIC: Input power from protection stage

Outputs:

- VP: Pre-regulated 5.7V 5A power line. Powers the linear regulators

The SEPIC regulators efficiently generate a 5.7V line from a wide range of input voltages. This line is noisy, and a later section will make this power line usable by other services on the board.

8.1.2 Linear Regulators and Filters

This sections of the schematics (Figure 43) apply further voltage regulation to the 5.7V power line to make it usable by other services of the board. This includes:

- The Raspberry Pi. It needs a clean 5V 2A power line
- The Arduino Shield and the ATmega 644. Both need a clean 5V 1A power line
- The Servomotors use the raw 5.7V line. Care has been taken since they inject noise

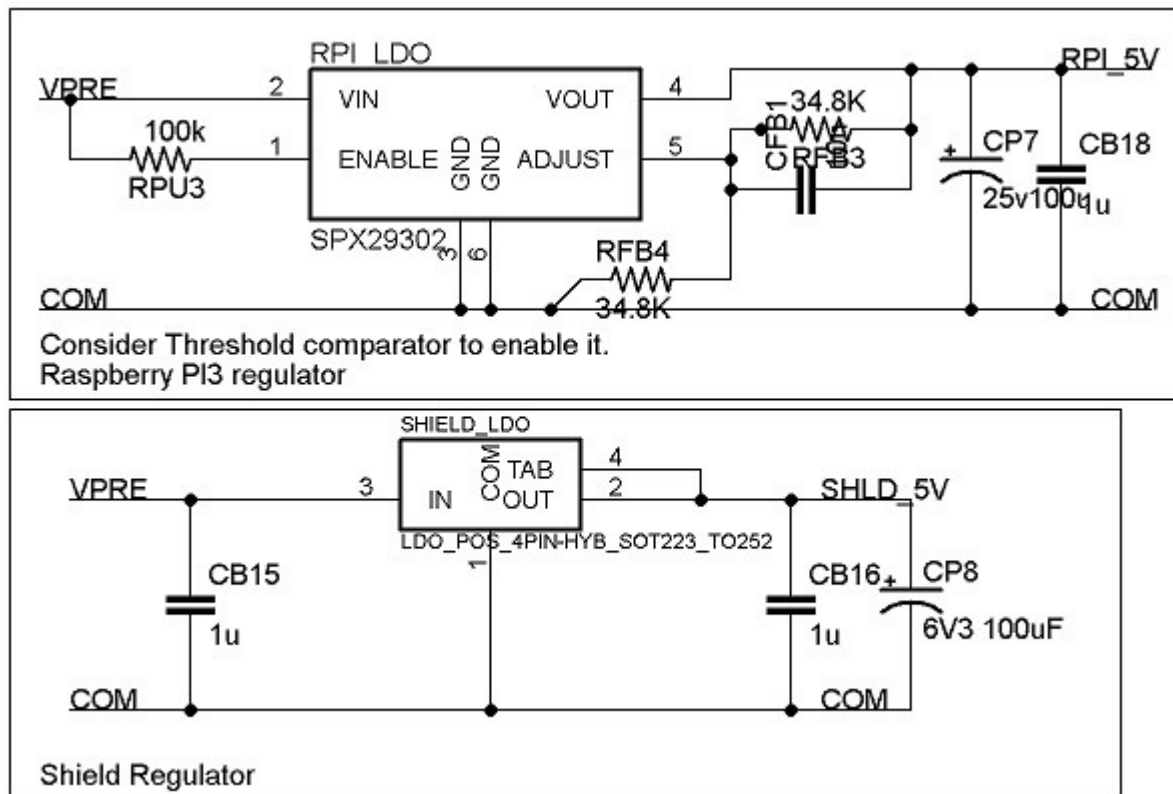


Figure 43: Shield Schematics - Linear Regulators

Inputs:

- VPRE: 5.7V 5A line from SEPIC regulator

Outputs:

- RPI_5V: 5V 3A line for the Raspberry Pi
- SHLD_5V: 5V 1A line for the microcontroller and the Arduino Shield

The 5.7V line is naturally noisy as the SEPIC regulator is not continuous in output. Rather than making the design of the switching regulator more complex than it already is, linear regulators to make the final fine regulation from 5.7V to a very clean and stable 5V. Linear regulators feature a good input noise rejection, and their inherent inefficiency is mitigated by the low voltage dropout they start from.

This paradigm allows the system be robust at the cost of a slight loss in overall power conversion efficiency.

8.1.3 H-Bridges and DC Motors connectors

This section of the schematics (Figure 44) allows the custom shield to make use of DC motors up to 3A. Details of the schematics have been discussed in Chapter 5.2.

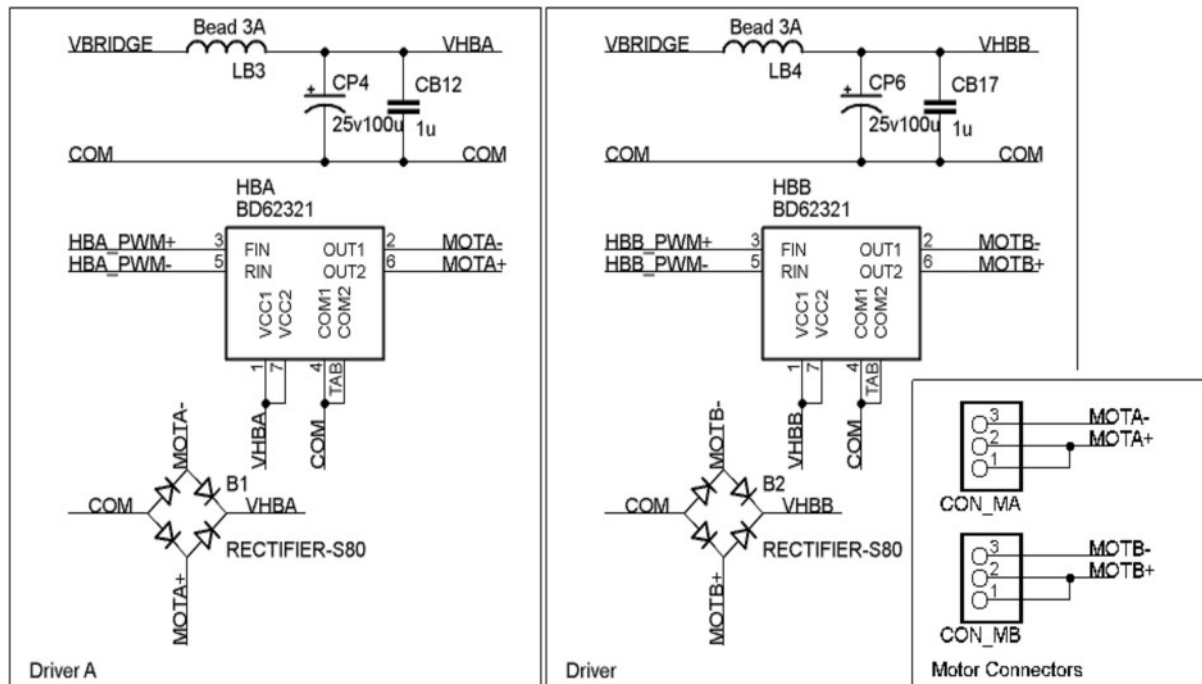


Figure 44: Shield Schematics - H-Bridges

Inputs:

- VBRIDGE: Protected input power line from battery
- HBA_PWM+/-: Digital control lines for Motor A coming from the microcontroller.
- HBB_PWM+/-: Digital control lines for Motor B coming from the microcontroller.

Outputs:

- MOTA+/-: Output power lines to the DC Motor A
- MOTB+/-: Output power lines to the DC Motor B

Line filters (LB3, CB12 and LB4, CB17) are used to clear out harmful switching noise coming from both the H-Bridges and the DC motors.

Rectifiers B1 and B2 are used to recirculate inductive power from the DC motors to the power lines. The H-Bridges have internal recirculation diodes, but having an external rectifier allows to move the power dissipation away from the H-Bridge package, improving reliability.

Two electrolytic capacitors (CP4, CP6) are used to both provide energy reserves for the H-Bridges and to absorb recirculation current that might otherwise increase the voltage on the VBRIDGE line above safe levels during breaking.

8.1.4 Microcontroller Services

The schematics for the microcontroller is shown in two sections. The first section (Figure 45) deals with filters, ISP programming connector, reset and test points. The second section (Figure 46) deals with the pin-out of the microcontroller itself. Pin assignment has been discussed in Chapter 4.4.2.

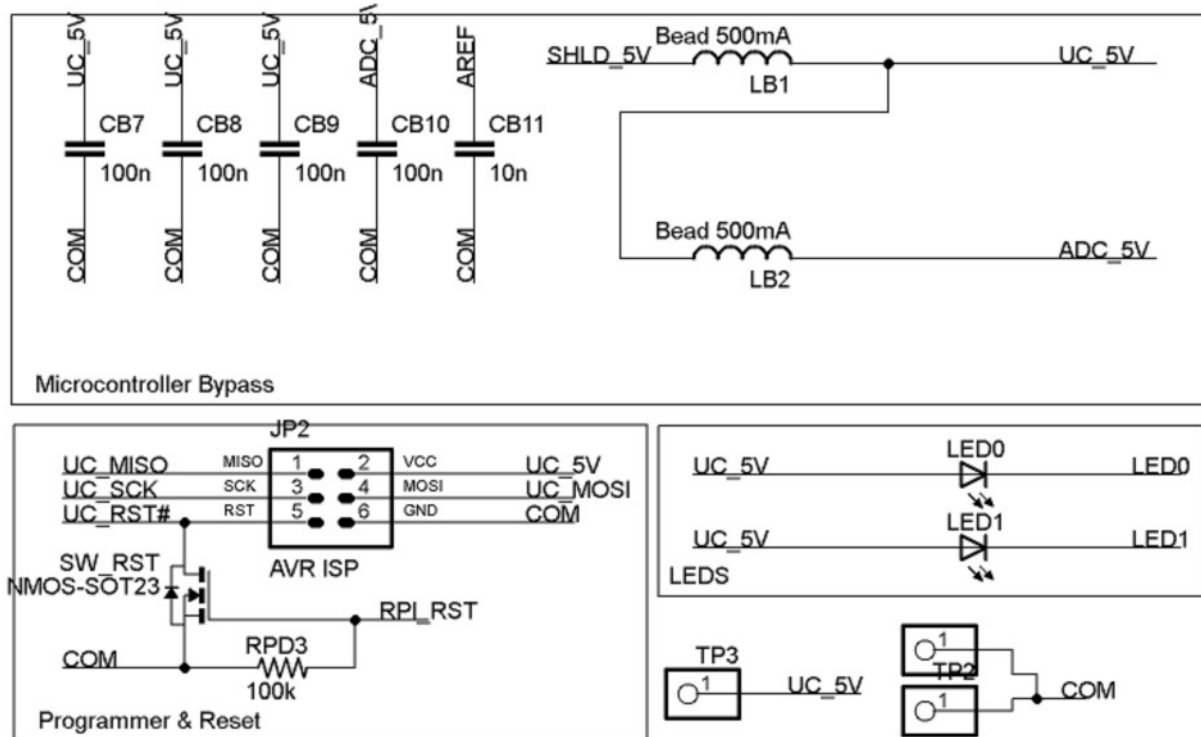


Figure 45: Shield Schematics - Microcontroller Filters and Services

Inputs:

- SHLD_5V: Clean 5V line from LDO
- RPI_RST: Digital reset line from the Raspberry Pi

Outputs:

- UC_5V: Filtered 5V line for the microcontroller
- ADC_5V: Filtered line for the ADC of the microcontroller
- LEDs
- Test Points: Allows to sense the 5V line from the outside
- ISP Connector: Allows the programming of the microcontroller

Line filters and bypass capacitors are used to clean up noise coming in and out of the microcontroller power lines. Digital electronics can be noisy.

The ISP connector allows the microcontroller to be programmed with a dedicated programmer, like the Atmel AvRisp MKII. This is needed to burn the bootloader that allows the microcontroller to be programmed from the Raspberry Pi in the first place.

The microcontroller reset line is controllable from the Raspberry Pi as well. A feature required to allow in circuit programming from Raspberry Pi with the Arduino Bootloader.

Two LEDs can be used at will be the user.

Test points are present to check that the microcontroller is powered right.

8.1.5 Microcontroller

The microcontroller (Schematics in Figure 46) is the nexus for all signals in the board. It can be programmed from the Raspberry Pi. It's seen by the Raspberry Pi as a ROS node and can be controlled through ROS topic messages. It controls the motors, the LEDs, the Servomotors and interfaces with an Arduino Shield. Pin assignment has been discussed in Chapter 4.4.2.

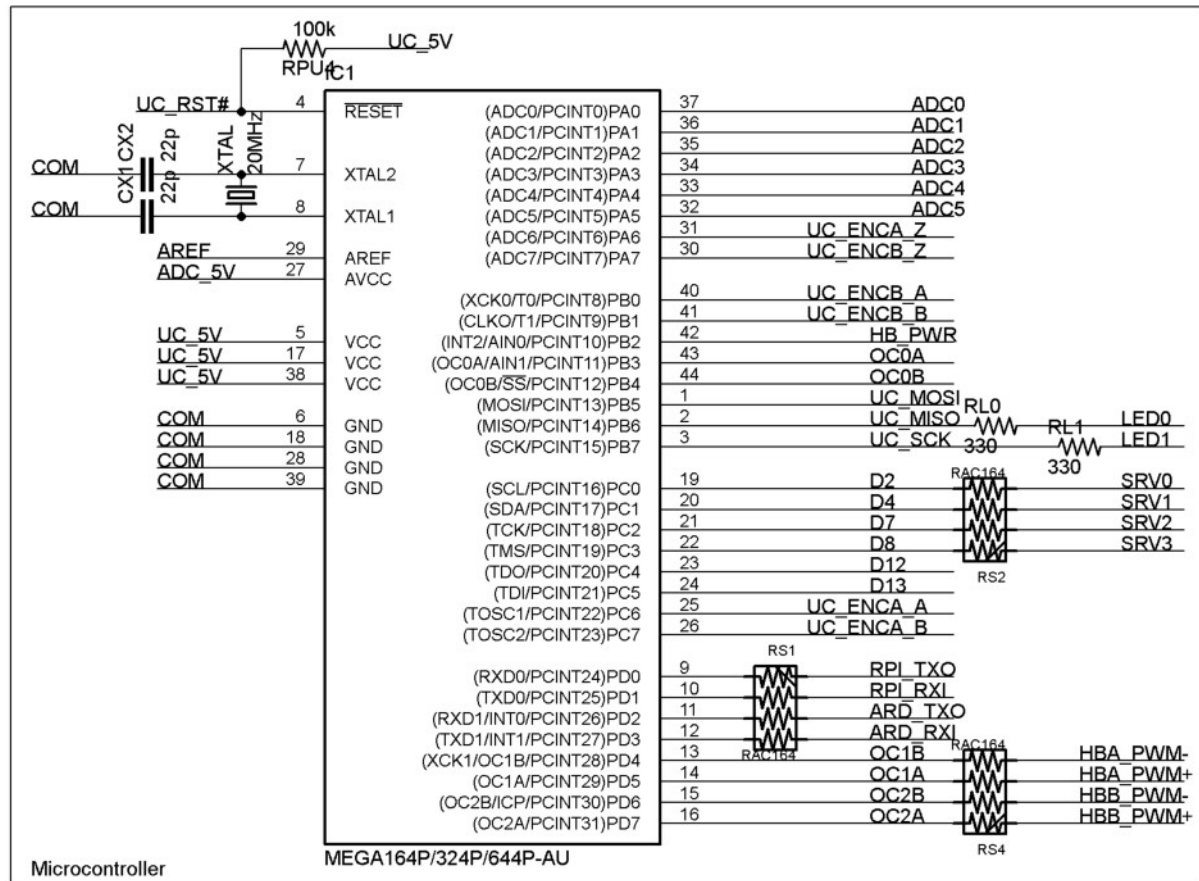


Figure 46: Shield Schematics - Microcontroller

Inputs:

- UC_5V: Clean 5V power line
- ADC_5V: Clean 5V power line
- UC_RST#: Reset signal coming from either ISP connector or from Raspberry Pi

Controls:

- RPI_TXO, RPI_RXI: Communication line with the Raspberry Pi. Rosserial Protocol
- ARD_TXO, ARD_RXI: Communication line with the Arduino Shield.
- HB_PWR: Cut power to H-Bridges and DC Motors
- HBA/B_PWM+/-: Control lines for the H-Bridges and the DC Motors
- UC_ENCA/B_A/B/Z: Encoders. Odometry, Position/Velocity PID with DC Motors

- SRV0/1/2/3: Servomotors control lines
- Dx: General I/O lines for the Arduino Shield
- LED0/1: Generic LEDs
- ADC0 to ADC 5: Analog lines from the Arduino Shield Connector

8.1.6 Raspberry Pi GPIO

The board uses only the first ten pins of the Raspberry Pi GPIO connector (Schematics in Figure 47). There is enough space below the board for the user to access to the other GPIO pins if needed, and using only ten pins to free board area for more useful features.

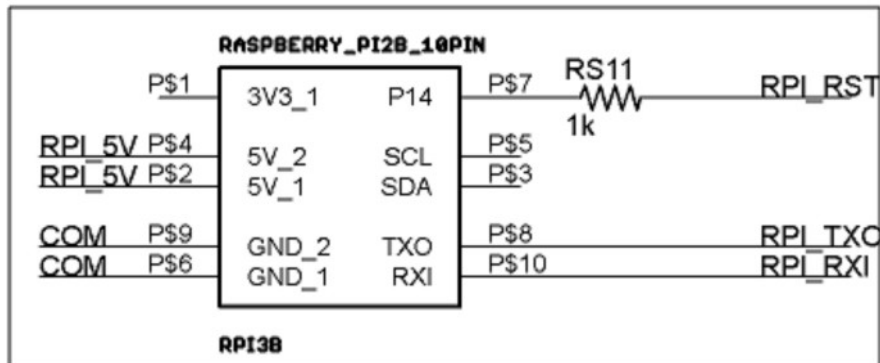


Figure 47: Shield Schematics - Raspberry Pi GPIO

Inputs:

- UC_RST#: Reset signal for the microcontroller

Controls:

- RPI_TXO, RPI_RXI: Communication line with the Raspberry Pi. Rosserial Protocol and programming through the Arduino Bootloader

Outputs:

- RPI_5V: 5V 3A clean power line that powers the Raspberry Pi

The GPIO allows for the Raspberry Pi to be powered by a battery from the custom shield, a feature that makes it that much easier for the robot maker to power the system safely.

The controllable reset line allows for the microcontroller to be programmed by the Raspberry Pi thanks to the RPI_TXO, RPI_RXI communication lines and the Arduino bootloader.

The RPI_TXO, RPI_RXI allows for the Raspberry Pi to control the shield thanks to the roserial protocol and the ROS topic messages.

8.1.7 Arduino Shield Connector

The Arduino Shield Connector (Schematics in Figure 48) allows for Arduino Shields to be used in combination with the custom board. It's an extremely useful feature as countless Arduino Shields exists that offer a wide range of features.

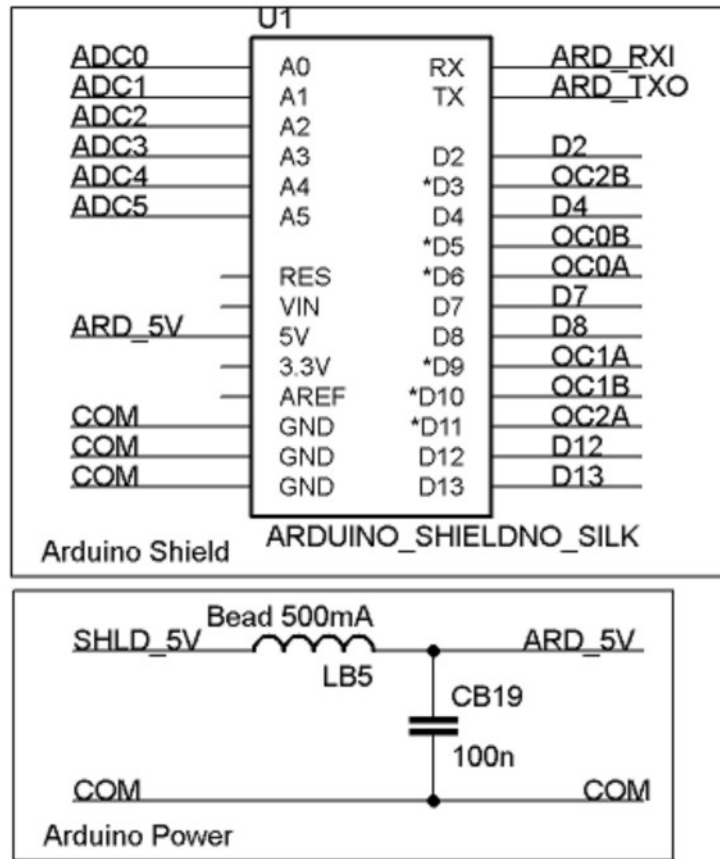


Figure 48: Shield Schematics - Arduino Shield

Inputs:

- SHLD_5V: Clean 5V power line

Controls:

- ARD_TXO, ARD_RXI: Communication line
- OC0/1/2A/B: PWM Lines. Cannot be controlled independently from DC motor lines.
- UC_ENCA/B_A/B/Z: Encoders. Odometry, Position/Velocity PID with DC Motors
- Dx: General I/O lines
- ADC0 to ADC 5: Analog lines

A line filter is used to insulate noise coming from and to the Arduino Shield. The connector simply routes all required lines to the microcontroller, allowing the user to take advantage of existing Arduino Shields.

8.1.8 Servomotors

The servomotor connector (Schematics in Figure 49) is straightforward. A line filter insulate the power line from the noisy servomotors, and an electrolytic capacitor provides an energy storage nearby to protect against current spikes and short brown-out.

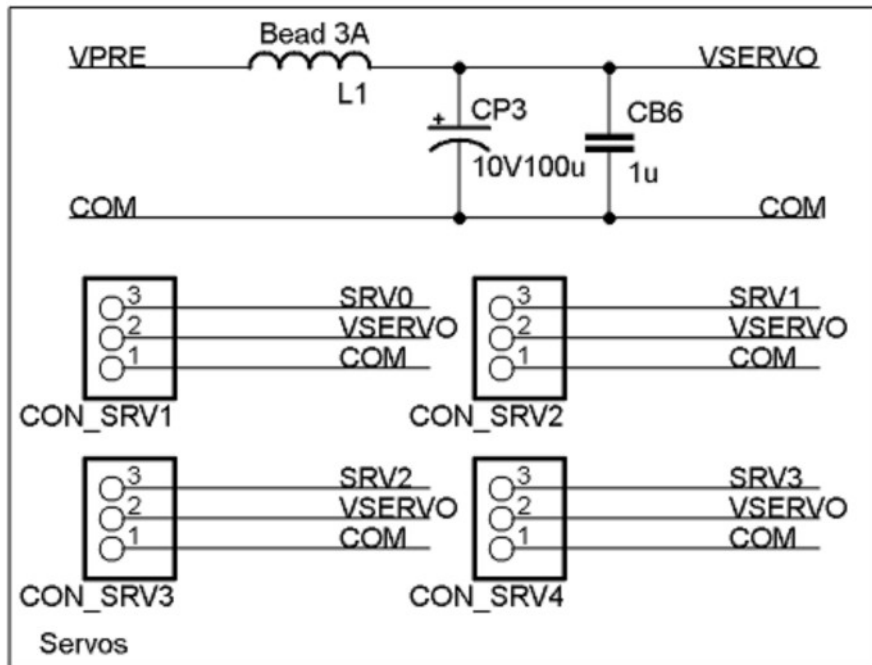


Figure 49: Shield Schematics - Servomotors

Outputs:

- VPRE: 5.7V 5A Noisy pre-regulated power line
- SRV0/1/2/3: Servomotor control signals

Powering the servomotors from the VPRE allows the user to enjoy the full rated torque of the servomotor and allows 5V servomotors to be powered from single LIPO cell 3.7V.

Additionally, noise from the servomotors often causes instability in the system, a problem that can be very hard to debug. It is a significant quality of life feature to handle this noise for the robot maker.

8.1.9 Encoders

The part of the schematics that deals with the encoders (Figure 50) involves some limitations. Several protections are provided to protect the board from failing encoders or encoders with wrong voltage levels.

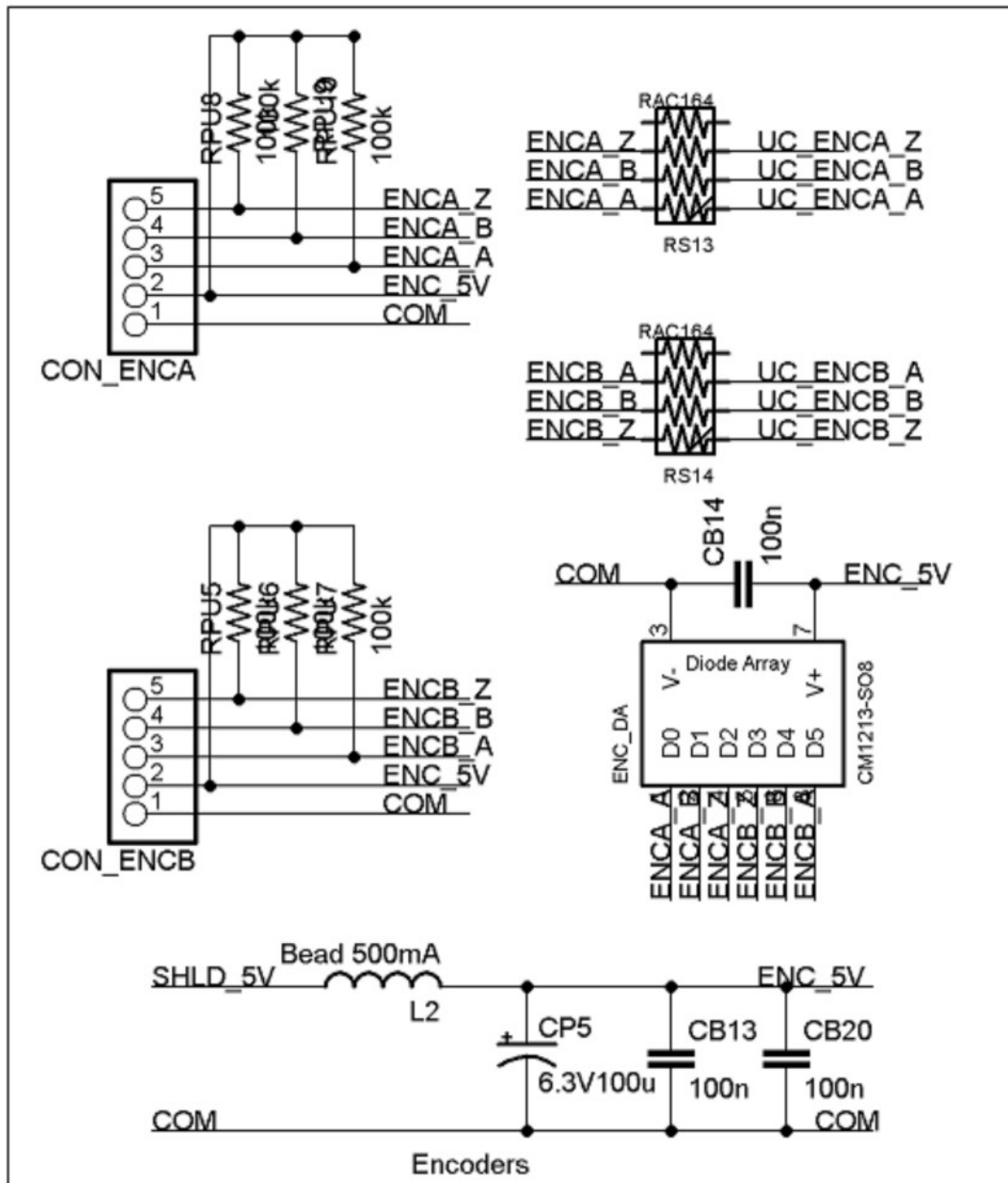


Figure 50: Shield Schematics - Encoders

Inputs:

- ENCA/B_A/B/Z: Encoder signals

Outputs:

- ENC_5V: Clean 5V power line

The encoders are powered by the same LDO that powers the microcontroller and the Arduino Shield. A line filter is used to insulate the board from the noisy encoders and an electrolytic capacitor provides an energy storage nearby to protect against current spikes and short brown-out.

Encoders comes with all kinds of electrical interface. Unfortunately it would take too much area to be compatible with all of them, and the design philosophy is to provide features that are both cheap and popular.

The choice was made to limit compatibility with 5V encoders only. Encoder signals can be either TTL or open collector/open drain outputs. The robot maker is expected to chose the right kind of encoders, or use an external level shifter.

A diode array protects the board if encoders with great logic voltage swing are used (12V/24V encoders).

8.2 Layout and Routing

With the schematics for the shield done, the next step was to decide on the form factor of the board, complete the routing of the layout and generate the gerber files and the BOM for production.

8.2.1 Form Factor

The form factors of the custom board (Figure 51) is the result of several routing attempts and considerations.

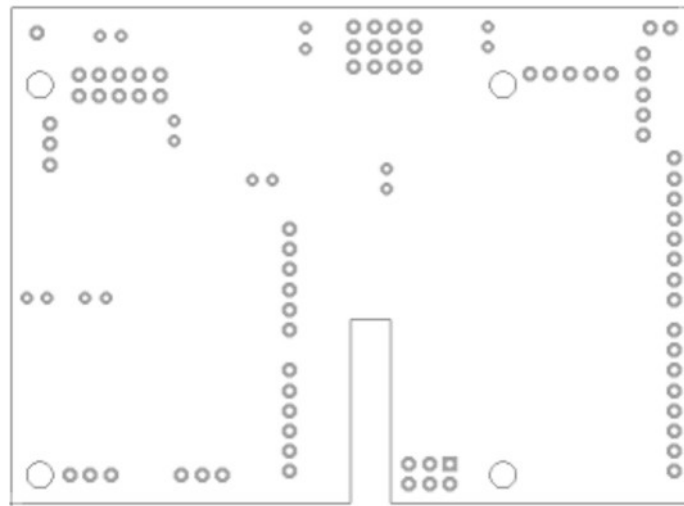


Figure 51: Board - Form Factor

The custom board is 85x62mm, the Raspberry Pi is 85x56mm. The custom board is about 6mm wider. It was considered acceptable to have the board slightly bigger as making the board smaller would have meant dropping features or using a tighter design rule. Both options would have had greater ill effects than just allowing for the added 6mm in width.

The board uses only 10 pins of the Raspberry Pi GPIO. Originally it was thought to use them all, but leave most of them unconnected. The solution chosen allows to reduce costs by having a smaller connector, does not hinder functionality as the user has enough space to access GPIO pins below the board and leaves more area for features.

The board features a rectangular hole in the middle of the lower side. This space allows for the flex cable that connects to a Raspicam [35] to pass through. A feature absent from every Raspberry Shield considered and that gives some real gripes to robot makers.

The position of the connectors was hard thought as the area is relatively small for the amount of features desired. This layout is the results of multiple failed placing and routing iterations.

8.2.2 Components Placing

Placing the components of the board proved to be a major challenge. In Figure 52 can be seen the final outline of the components and the pads on both sides of the custom board.

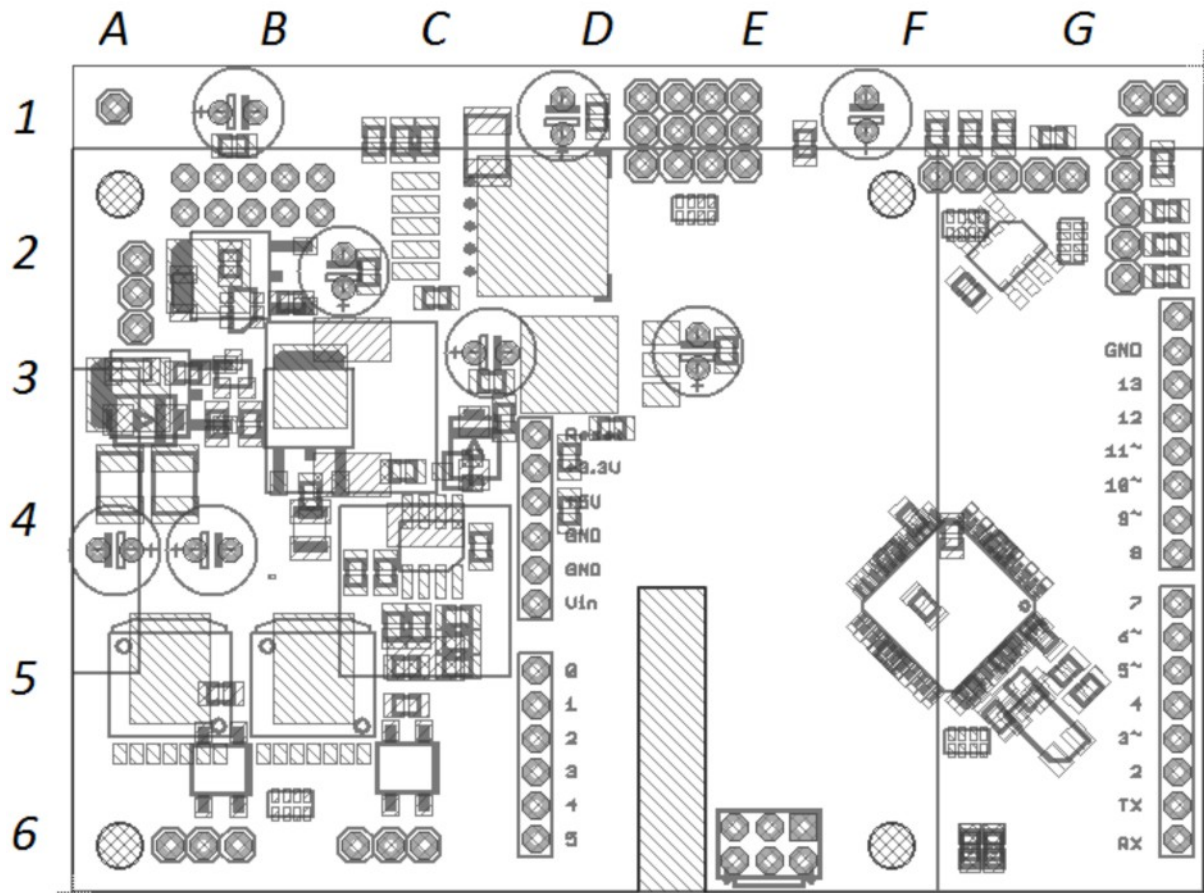


Figure 52: Board - Component Placing

Details of the component placing inside the board:

- A2: The input power connector. Can be either a 2.54mm or a 5.12mm screw connector
- B1/2: GPIO connector of the Raspberry Pi
- A1 to D5: This section is dedicated to the Power supply.
 - B2 (Bot): Ideal diode and resettable fuse for the SEPIC regulator
 - A3 (Bot): Solid state relay and protections for the H-Bridges
 - C2, D2 (Bot): 3A LDO for the Raspberry Pi
 - B3 to D5 (Top): Power components of the SEPIC regulator
 - B3 to B5 (Bot): SEPIC controller
 - D3: 5V LDO for Arduino Shield, microcontroller and Encoders
- A5 to C5 (Bot): H-Bridges
- A6 to C6: Motor connectors and recirculation diodes
- A4 to B4: Capacitors and filters for the H-Bridges

- D4 to D6 and G3 to G6: Arduino Shield connector
- E6: ISP Connector
- D/E 5 to 6: Hole for the Raspicam flex cable
- D1 to E1: Servomotors connector, filter and capacitor
- F4 to G5 (Bot): ATmega 644
- F4 to G5 (Top): Filters, quartz and protections for the Microcontroller
- F1 to G2: Encoders. Connectors, filters, capacitor and protections
- F/G6(Top): LEDs
- E3 to F4: Seemingly empty area. It's very busy with interconnections and vias
- A1 and G1: Test Points

8.2.3 Layout

Routing the board and closing all the nets while keeping an eye to rated currents and interferences was a daunting task. It would have been simpler had a four layer design rule been used, but two layers board are cheaper, so the choice was made to limit the board to two layers if at all possible.

The layout for the top layer can be seen in Figure 53.

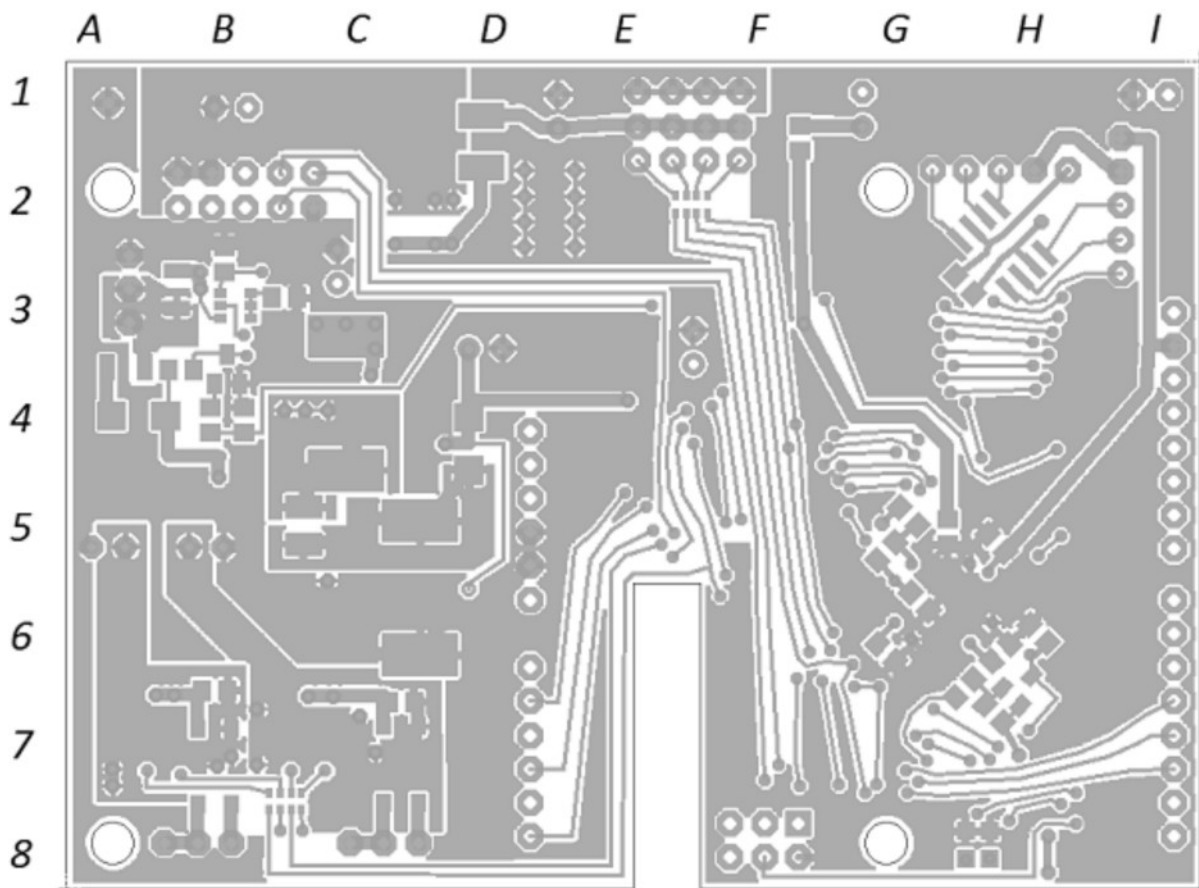


Figure 53: Board - Layout TOP Layer

The layout for the bottom layer can be seen in Figure 54.

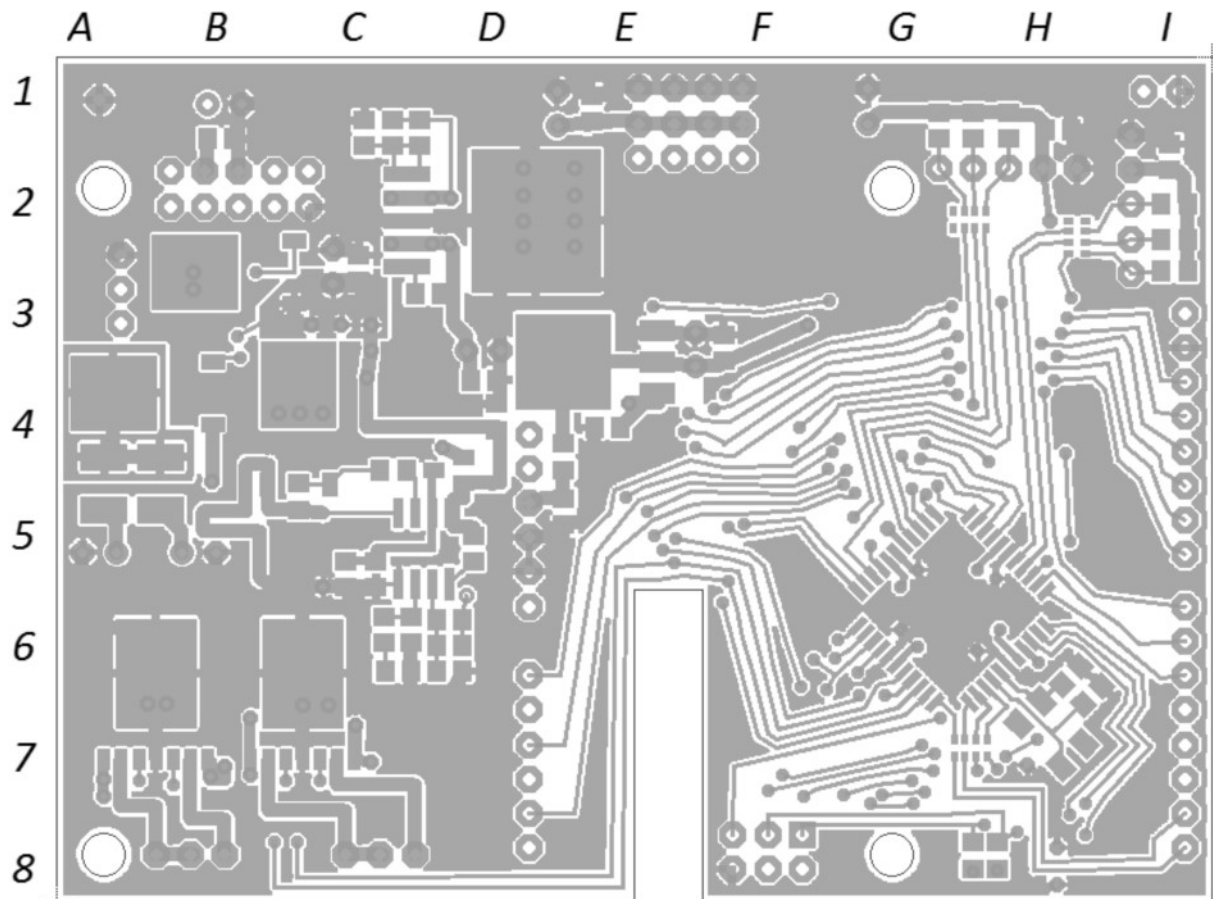
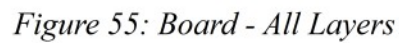


Figure 54: Board - Layout BOT Layer

Details of the layout:

- A1 to E5 and A6 to D8: Large copper areas for the power lines. It was challenging to give the power lines enough areas to allow for their rated current while also leaving space for the smaller control lines to make their way through
- B5: Shaped track that acts as shunt resistor to sense the SEPIC switch current
- F3 to H7: Area very busy with the routing of the microcontroller lines. Multiple iterations were required when after a failed routing, pins on the microcontroller were swapped in the schematics to try and pass another way.
- H7 (Bot): Crystal for the microcontroller
- G6 to H6 (Bot): Microcontroller shield area
- B2 to C2 (Top): Copper area for the 5V line for the Raspberry Pi
- A5 to A7 and B5 to C7 (Top): Copper Area for the supply for the H-Bridges
- C3 to D5: Copper areas for the SEPIC regulator
- By default, all remaining free area is routed to the COM line
- C8 to E8 to E5 to G7: PWM lines for the H-Bridges

The next step in the design is to generate the gerber file for the production of the PCB and prepare a BOM for the components. The full layout of custom shield with all layers visible can be seen in Figure 55.



92/113

In Figure 56 can be seen the final look of the board after the gerber files have been generated and uploaded to Eurocircuit for manufacturing.

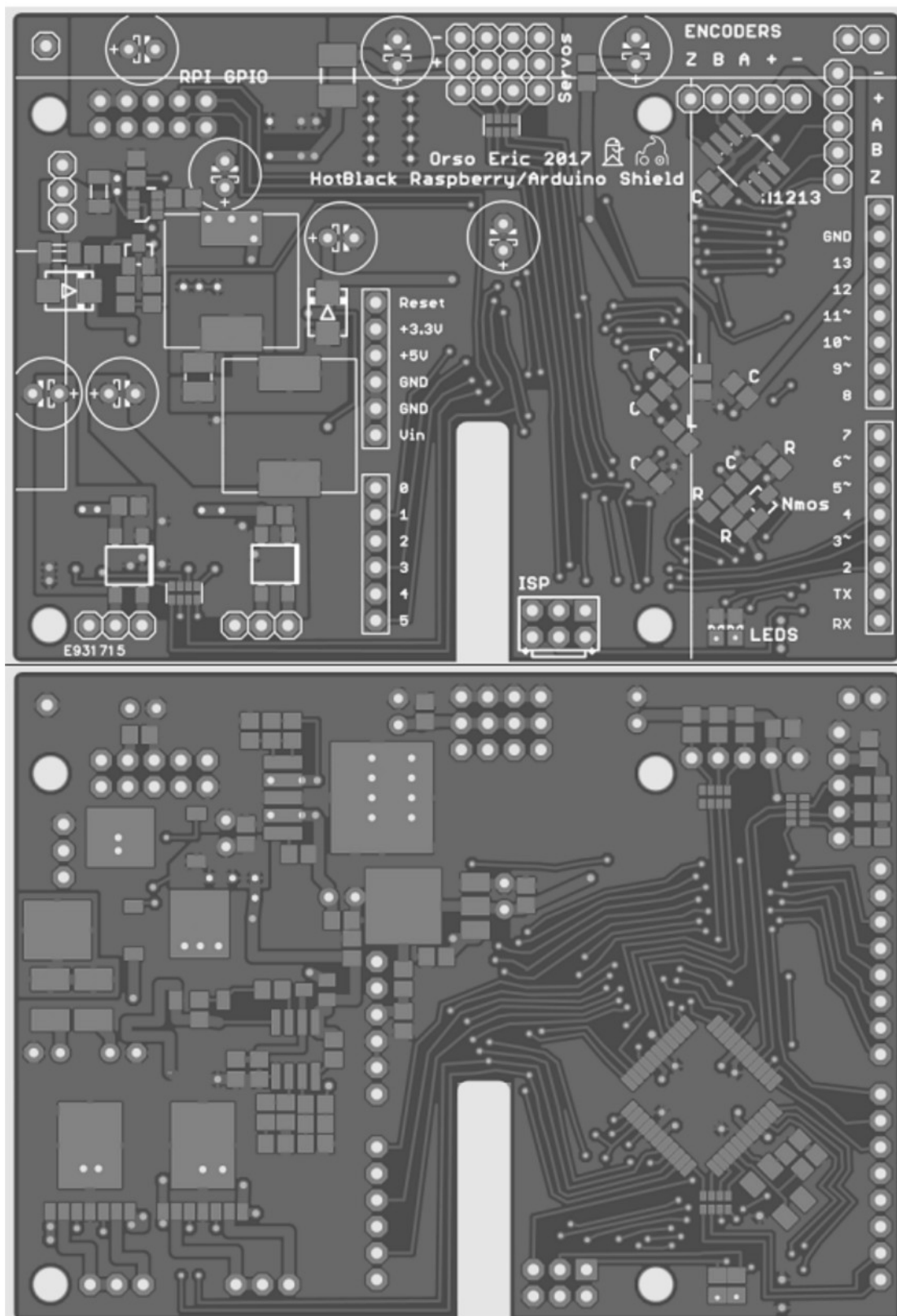


Figure 56: Eurocircuit Board Aspect

In table 9 can be seen the Bill of Materials for the Raspberry Pi Shield. RS Components was chosen as supplier.

<i>Description</i>	<i>Code</i>	<i>Quantity</i>
At Mega 644	ATMEGA644A-AU	1
Distanziali M3 31.8mm	HS 6 10	4
matched bjt	DMMT5401-7-F	1
Raspberry Pi GPIO Female	SH-2x20-L16.5	1
Pin Strip Maschio 2.54mm		2
Pin Strip Femmina 2.54mm	W34420TRC	2
Morsetto a vite 2.54mm 1x5		2
Morsetto a vite 5.12mm 1x2		3
Quarzo 7x5mm 20MHz	LFXTAL026394	1
Led Verde 2.2V SMD0805	LGR971	2
Resettable Fuse PTC 85mΩ	MF-NSMF200-2	2
Electrolitic Capacitor 100uF 6,3V 480mA 6,3mm	63ZLG100M63X7	4
Electrolitic Capacitor 220uF 35V 755mA 8mm	35ZLH220MEFC8X11.5	2
Sepic Controller SOIC8	LT1619ES8#PBF	1
Condensatore 22pF SMD0805		2
Condensatore 6.8nF V SMD0805	08055F682KAZ2A	1
Condensatore 100nF 50V SMD0805	08055C104KAT2A	8
Capacitor Sepic 10uF 100V SMD1812	CKG45NX7S2A106M500JH	1
Condensatore 10uF 25V SMD1206	GRM31CB31E106KA75L	2
Ferrite 240nH 26mΩ 3.2A SMD0806	74479876124C	3
Inductor 33uH 3A	B82477G4333M000	2
Twin Scottky 200mA sot23	BAT54CW	1
SMD rectifier	DB207S	2
Diodo Schottky 60V 0,75V 3A	SK36A R2	2
nmos sot23	IRLML2803TRPBF	2
pmos to252	IPD50P03P4L-11	2
Low Vth NMOS	FDD6635	1
H-Bridge. 3A 6Vmin 36Vmax	BD62321HFP-TR	2
LDO Adjust 3A	SPX29302T5-L/TR	1
LDO 5V 800mA sot223	REG1117-5	1
4X resistors 1K SMD1206	RAC164D102JC	5
Resistenza 1K SMD0805	CR0805-FX-1001ELF	1
Resistenza 10K SMD0805	CR0805-JW-103ELF	2
Resistenza 33.8K SMD0805	CPF-A-0805B34K8E1	3
Resistenza 100K SMD0805	RMC1/10K104FTP	13

Table 9: BOM - Custom Raspberry Pi Shield

8.3 Shield Assembly

The shield PCB that arrived from Eurocircuit can be seen in Figure 57.

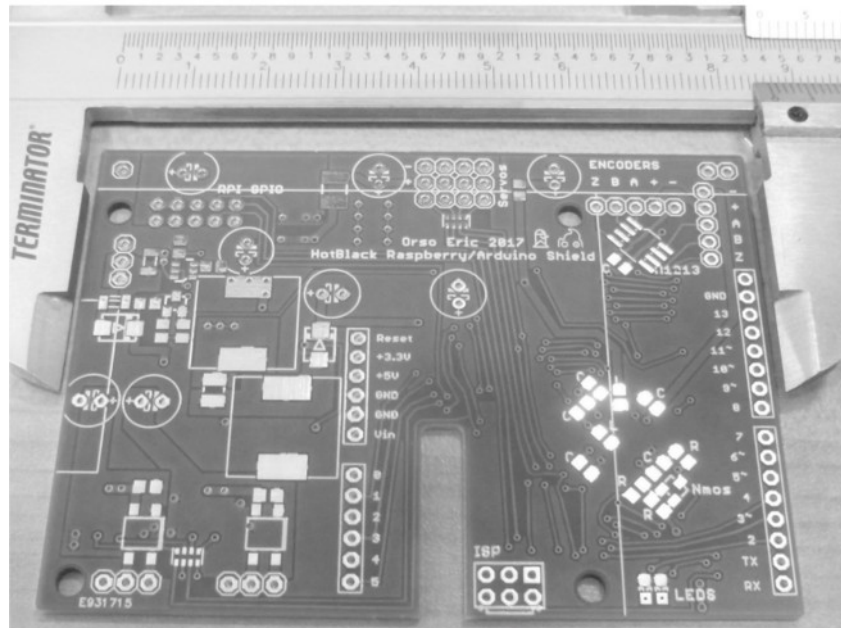


Figure 57: Shield PCB

With the components and the PCB available, the custom shield was finally soldered. In Figure 58 and 59 can be seen the final result after the assembly phase.

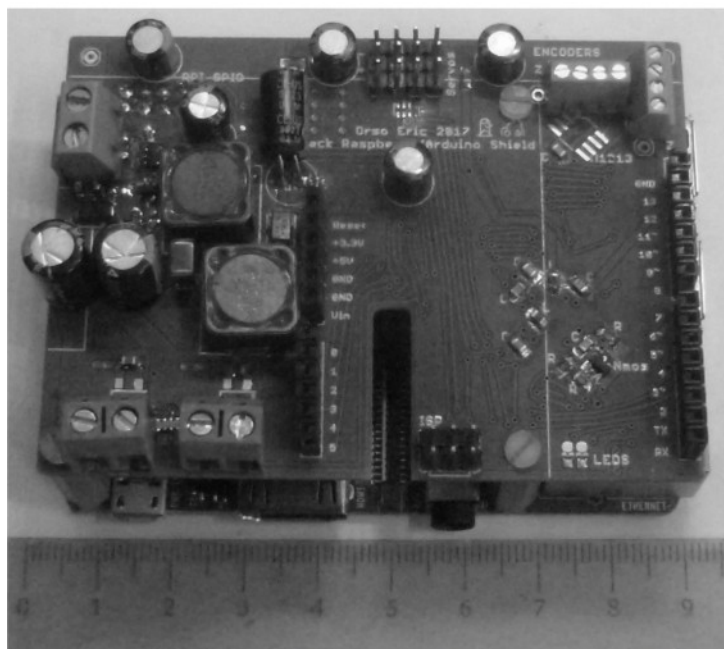


Figure 58: Custom Raspberry Pi Shield Assembled

In Figure 59 can be seen the assembled bottom side of the custom shield.

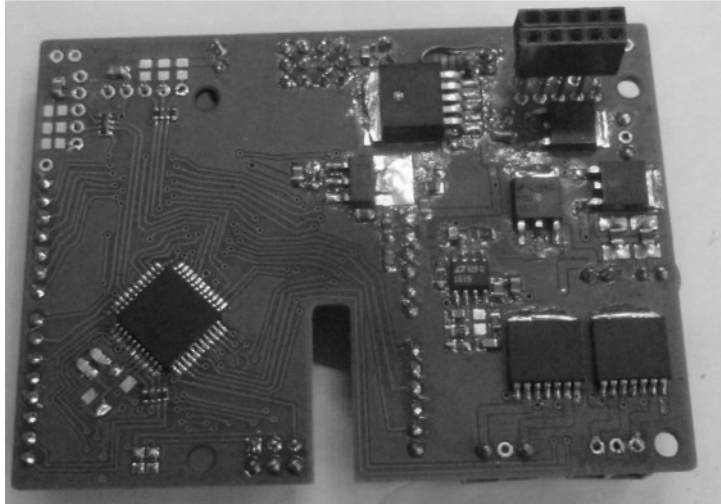


Figure 59: Custom Raspberry Pi Shield Assembled - Bottom View

9. Applications

With the robotic platform Seeker of Ways ready and the custom shield finally assembled, a few example applications have been made making use of the HotBlack Software Framework.

Example applications:

- Motor Testing
- Image Processing
- FPV Robot control with Keyboard

9.1 Motor Testing

This application is written in python and runs on the remote HotBlack software framework as described in Chapter 6.3.

The code described in the Code Snippet *HotBlack Framework Sketch: Test Motors HB Core Messages* move the two motors in opposite directions giving them a linear acceleration ramp followed by a linear deceleration ramp. It's meant to test the motors of the platform.

Code Snippet: HotBlack Framework Sketch: Test Motors HB Core Messages

```
#HotBlack python library
import dotbot_ros
#for print
import sys
from sys import stdout
#for SeekerOfWay print
from std_msgs.msg import UInt8
#for SeekerOfWay control
from hb_core_msgs.msg import Velocity2D

class Node(dotbot_ros.DotbotNode):
    node_name = 'SeekerOfWays_TestMotors'
    #
    def setup(self):
        print 'Seeker Of Way Test Motors'
        sys.stdout.flush()
        #looprate automatically tie to loop. it's in Hz
        self.loop_rate = dotbot_ros.Rate(2)
        #Communication with SeekerOfWays
        self.pub_cnt = dotbot_ros.Publisher('/msg_cnt_u8', UInt8)
        self.cmd_vel = dotbot_ros.Publisher('/cmd_vel', Velocity2D)
```



```

self.cnt = 0
self.fwd = 31
self.turn = 0
self.dir = 0

self.max = +16000
self.min = -16000

def loop(self):
    #C | uint8_t msg | those are data structure ROS based that only hold one
data.
    #those are basic ROS messages
    msg = UInt8()
    #fill data of message with a counter.
    msg.data = self.cnt
    #don't have post increment. increment counter. crash if exceed 255
    if (self.cnt < 254):
        self.cnt += 1
    else:
        self.cnt = 0

    self.pub_cnt.publish(msg)
    #update motor commands
    if (self.dir == 0):
        self.fwd += 250
    else:
        self.fwd -= 250

    if (self.fwd > self.max):
        self.fwd = self.max
        self.dir = 1
    if (self.fwd < self.min):
        self.fwd = self.min
        self.dir = 0

    self.turn = 0;

    #construct the motor message
    cmd = Velocity2D()
    cmd.linear = self.fwd

```

```
cmd.angular = self.fwd
self.cmd_vel.publish(cmd)

print self.fwd, self.turn
sys.stdout.flush()
```

9.2 Camera Image Processing

This application takes the camera stream and generates a topic which contains the stream converted to gray scale. On the Figure it's applied a filter called Orb, which is already implemented in the OpenCV libraries [36] inside ROS. This filter searches for features inside the image (Figure 60).

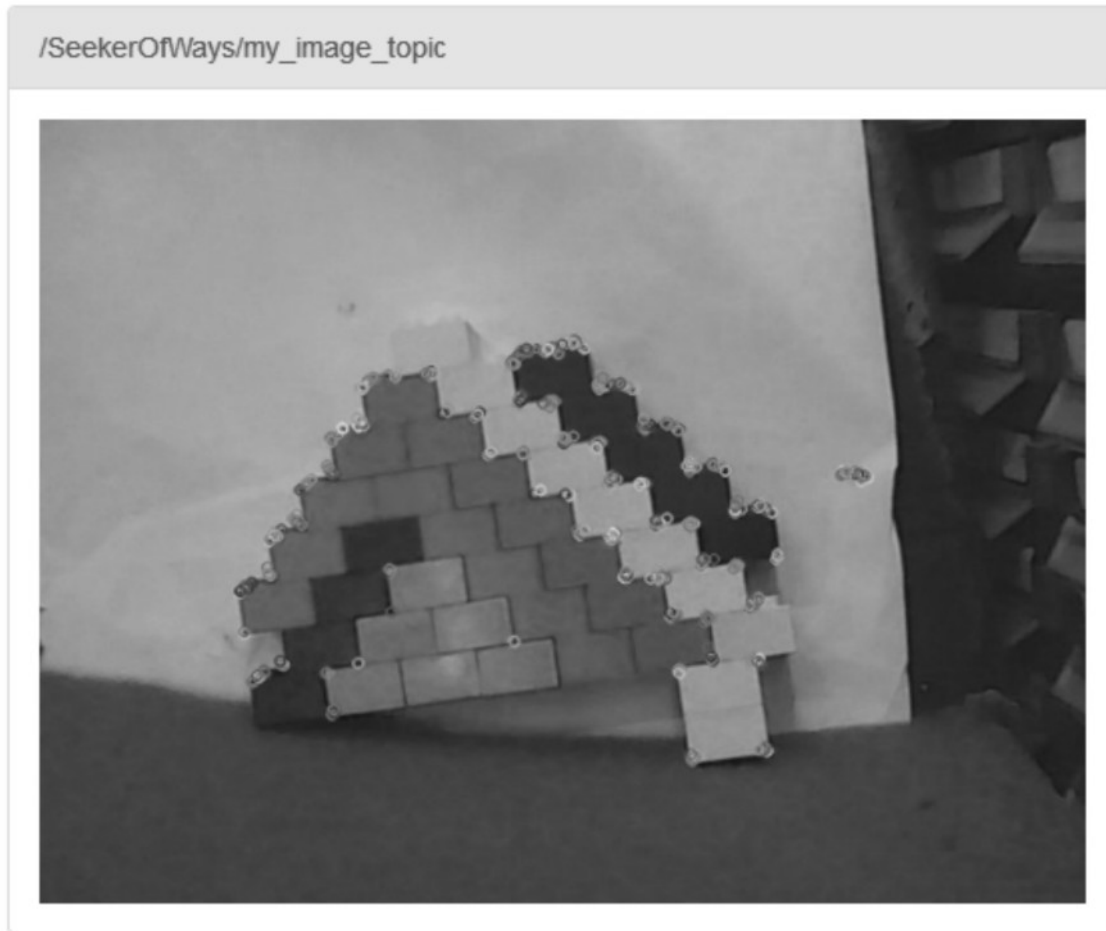


Figure 60: Application - CV Orb

The python code of the application can be seen in the code snippet *Test USB Camera Processing* down below.

Code Snippet: HotBlack Framework Sketch: Test USB Camera Processing

```
import dotbot_ros

from geometry_msgs.msg import Vector3
from std_msgs.msg import Float32, String, UInt8
from gpiozero import DistanceSensor, Button, PWMLED, Robot
import sys
from math import sin
import rospy
```

```

import cv2
import time
from std_msgs.msg import String
from sensor_msgs.msg import Figure
from cv_bridge import CvBridge, CvBridgeError

class Node(dotbot_ros.DotbotNode):
    node_name = 'camera'

    def setup(self):
        self.Figure_pub = dotbot_ros.Publisher("my_Figure_topic", Figure)

        self.bridge = CvBridge()

        self.Figure_sub =
dotbot_ros.Subscriber("/usb_cam/Figure_raw", Figure, self.callback)

        self.orb = cv2.ORB()

    def callback(self, data):
        try:
            Figure = self.bridge.imgmsg_to_cv2(data, "bgr8")
            gray_Figure = cv2.cvtColor(Figure, cv2.COLOR_BGR2GRAY)
            kp1, des1 = self.orb.detectAndCompute(gray_Figure, None)
            Figure = cv2.drawKeypoints(Figure, kp1)
        except CvBridgeError as e:
            print e

        try:
            self.Figure_pub.publish(self.bridge.cv2_to_imgmsg(Figure, "bgr8"))
        except CvBridgeError as e:
            print e

```

9.3 FPV First Person View Control

This application uses a custom made html page, the HotBlack software framework, a piece of javascript code and a robot to make a first person view control application.

This application allows the user to see the streamed Figure from the camera on the custom browser page and control the robot using the WASD keys on the keyboard. It's an application that demonstrates the power and ease of use of this framework.

In Figure 61 can be seen the browser page when the application is running.

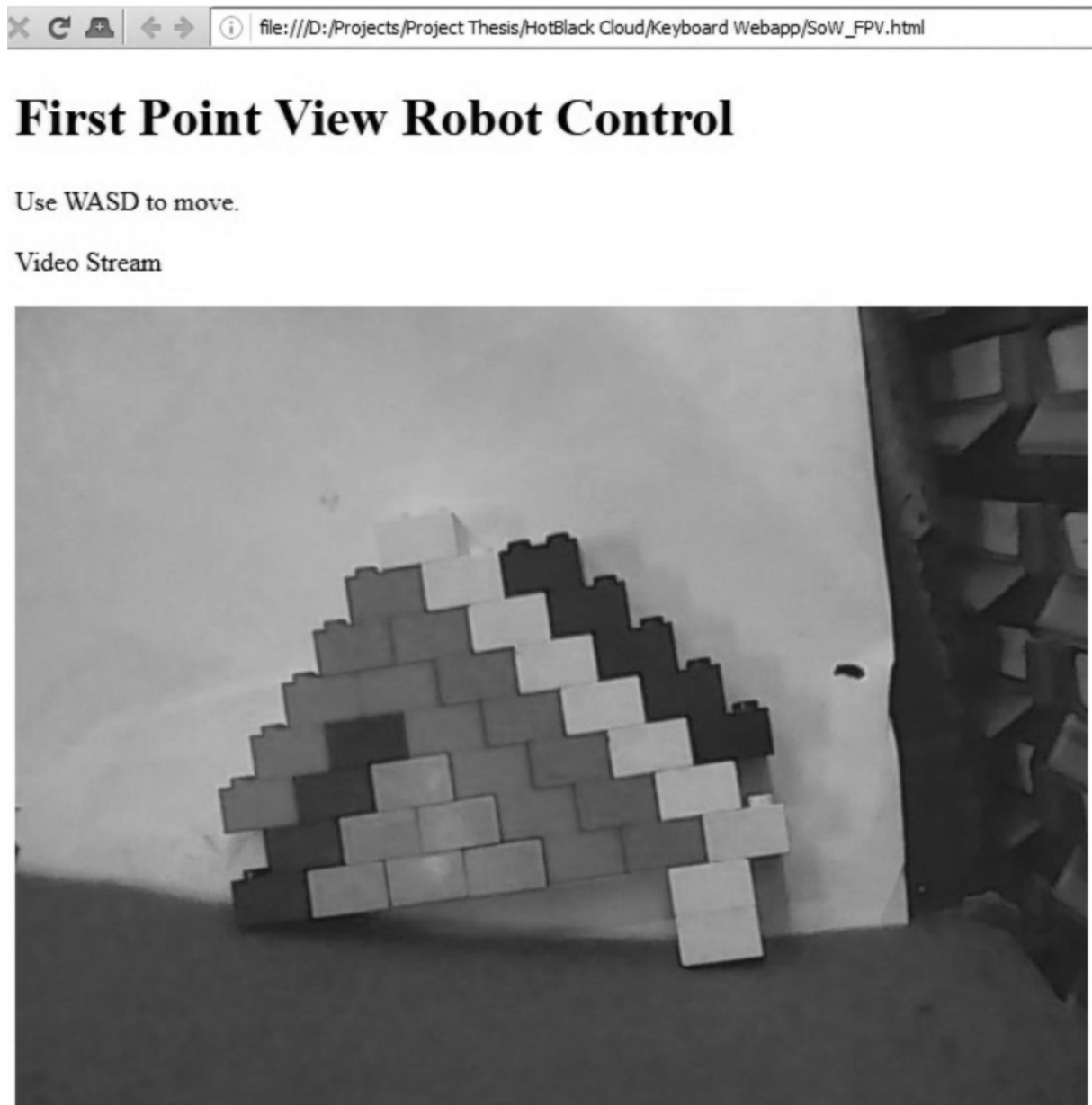


Figure 61: Application - FPV Keyboard Control Browser Page

The following Code Snippet *Keyboard FPV* is the python code that runs on the HotBlack cloud. It takes keystrokes detected with the Keyboard APP and use them to control the robot. The video stream is visible by default by the web browser page thanks to how the HotBlack Software Framework is designed.

Code Snippet: HotBlack Framework Sketch: Keyboard FPV

```
#HotBlack python library
import dotbot_ros
#for print
import sys
from sys import stdout
#for SeekerOfWay print
from std_msgs.msg import UInt8
#for SeekerOfWay control
from hb_core_msgs.msg import Velocity2D
#for keyboard APP
from geometry_msgs.msg import Twist

class Node(dotbot_ros.DotbotNode):
    node_name = 'SeekerOfWays_Keyboard'
    #
    def setup(self):
        print 'Seeker Of Way Keyboard Control'
        sys.stdout.flush()
        #looprate automatically tie to loop. it's in Hz
        self.loop_rate = dotbot_ros.Rate(10)
        #define callback for the keyboard app
        dotbot_ros.Subscriber('/keyboard', Twist, self.keyb_wasd)

        #Communication with SeekerOfWays
        self.pub_cnt = dotbot_ros.Publisher('/msg_cnt_u8', UInt8)
        self.cmd_vel = dotbot_ros.Publisher('/cmd_vel', Velocity2D)
        self.cnt = 0
        self.fwd = 0
        self.turn = 0
        self.dir = 0

        self.max = +25
        self.min = -25

    #handler of keyboard APP
    def keyb_wasd(self, msg):
        self.throttle = 31
```

```

        #update motor variables
        self.fwd = 63*256*msg.linear.x
        self.turn = 31*256*msg.angular.z
        #debug
        print 'Shaka', self.fwd, self.turn
        stdout.flush()

def loop(self):
    #C | uint8_t msg | those are data structure ROS based that only hold one
data.
    #those are basic ROS messages
    msg = UInt8()
    #fill data of message with a counter.
    msg.data = self.cnt
    #don't have post increment. increment counter. crash if exceed 255
    if (self.cnt < 254):
        self.cnt += 1
    else:
        self.cnt = 0

    self.pub_cnt.publish(msg)

    #construct the motor message
    cmd = Velocity2D()
    cmd.linear = -self.fwd
    cmd.angular = self.turn
    self.cmd_vel.publish(cmd)

    #print self.fwd, self.turn
    #sys.stdout.flush()

```

The code for the HTML page can be seen in the Code Snippet *HTML Page: Keyboard FPV* down below. The user just double click on the html file after the python sketch begins running.

A browser page will open, loading the scripts that interface the html page with the HotBlack cloud and with the javascript that reads the keystrokes from the local machine.

A video stream from the robot will also be shown.

Code Snippet: HTML Page: Keyboard FPV

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">

<link rel="stylesheet" type="text/css"
      href="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8/themes/base/jquery-
ui.css" />

<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.0/jquery.min.js"></script>
<script      src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.8.23/jquery-
ui.min.js"></script>
<script
src="http://cdn.robotwebtools.org/EventEmitter2/current/eventemitter2.min.js"></
script>
<script src="http://cdn.robotwebtools.org/roslibjs/current/roslib.js"></script>
<script src="keyboardteleop.js"></script>

<script                                     type="text/javascript"
src="http://www.hotblackrobotics.com/cloud/webgui/static/js/initros.js"></script
>

<script type="text/javascript">
    var robot_address = '192.168.0.101';
    var robot_name = 'SeekerOfWays'
    start_ros('192.168.0.101',          'SeekerOfWays',          '192.168.0.101',
'192.168.0.101/bridge/');

</script>

<script>
/**
 * Setup all GUI elements when the page is loaded.
```



```

*/
function init() {

    // Initialize the teleop.
    var teleop = new KEYBOARDTELEOP.Teleop({
        ros : ros,
        topic : '/TeleOP/keyboard'
    });

}
</script>
</head>

<body onload="init()">
    <h1>First Point View Robot Control</h1>
    <p>Use WASD to move.</p>

    <div id="stream">
        <p>Video Stream</p>
        
    </div>
</body>
</html>

```

The Javascript code that reads keystrokes and generate a ROS topic can be seen in the code snippet *Javascript: Keyboard FPV* down below. This code was written by a fellow robot maker that was tackling the same problem.

Code Snippet: Javascript: Keyboard FPV

```
/**
 * @author Russell Toris - rctoris@wpi.edu
 */

var KEYBOARDTELEOP = KEYBOARDTELEOP || {
  REVISION : '0.4.0-SNAPSHOT'
};

/**
 * @author Russell Toris - rctoris@wpi.edu
 */

/**
 * Manages connection to the server and all interactions with ROS.
 *
 * Emits the following events:
 *   * 'change' - emitted with a change in speed occurs
 *
 * @constructor
 * @param options - possible keys include:
 *   * ros - the ROSLIB.Ros connection handle
 *   * topic (optional) - the Twist topic to publish to, like '/cmd_vel'
 *   * throttle (optional) - a constant throttle for the speed
 */
KEYBOARDTELEOP.Teleop = function(options) {
  var that = this;
  options = options || {};
  var ros = options.ros;
  // permanent throttle
  var throttle = options.throttle || 1.0;

  // used to externally throttle the speed (e.g., from a slider)
  this.scale = 1;
```

```

// linear x and y movement and angular z movement
var x = 0;
var y = 0;
var z = 0;

var cmdVel = new ROSLIB.Topic({
  ros : ros,
  name : '/keyboard',
  messageType : 'geometry_msgs/Twist'
});

// sets up a key listener on the page used for keyboard teleoperation
var handleKey = function(keyCode, keyDown) {
  // used to check for changes in speed
  var oldX = x;
  var oldY = y;
  var oldZ = z;

  var pub = true;

  var speed = 0;
  // throttle the speed by the slider and throttle constant
  if (keyDown === true) {
    speed = throttle * that.scale;
  }
  // check which key was pressed
  switch (keyCode) {
    case 65:
      // turn left
      z = 1 * speed;
      console.log("left");
      break;
    case 87:
      // up
      x = 1 * speed;
      console.log("up");
      break;
    case 68:
      // turn right
      z = -1 * speed;

```

```

        console.log("right");
        break;
    case 83:
        // down
        x = -1 * speed;
        console.log("down");
        break;
    default:
        pub = false;
}

// publish the command
if (pub === true) {
    var twist = new ROSLIB.Message({
        angular : {
            x : 0,
            y : 0,
            z : z
        },
        linear : {
            x : x,
            y : 0,
            z : 0
        }
    });
    cmdVel.publish(twist);
    console.log("publish");

    // check for changes
    if (oldX !== x || oldY !== y || oldZ !== z) {
        that.emit('change', twist);
    }
}

};

// handle the key
var body = document.getElementsByTagName('body')[0];
body.addEventListener('keydown', function(e) {
    handleKey(e.keyCode, true);
}, false);

```

```
body.addEventListener('keyup', function(e) {  
    handleKey(e.keyCode, false);  
}, false);  
};  
KEYBOARDTELEOP.Teleop.prototype.__proto__ = EventEmitter2.prototype;10.  
Conclusions
```

10. Conclusions

This thesis described an approach to robotics made of the following parts:

- A repository of open source code and projects
- A cloud interface and a software framework
- A Raspberry Pi model 2 or 3
- A custom Raspberry Pi Shield

The approach described allows robot makers to develop robotic applications with more ease as the various tools are designed to hide complexity and handle the nuances involved so that the robot maker can focus on the final application.

Within this thesis, a custom board was designed with the needs of robot makers in mind as no board found on the market was free of such problems. A prototype of this board was built along with a robotic mobile platform to test the system first hand.

Within this thesis several robotic applications were developed, taking advantage of the HotBlack Software Framework. Those application proved that the amount of work required by the robot maker to develop a meaningful robotic application can be vastly reduced thanks to the tools provided.

Applications that were developed include:

- Control motors
- Computer Vision filters
- FPV remote control through browser page

It is the opinion of the candidate that this approach has the potential of achieving wide adoption, and by allowing for more people to engage in robotics and focus on the applications, the boundaries of robotic itself can be expanded.

10.1 Future Development

While the results of the hardware and software are more than satisfying, test showed limitations and inefficiencies that can be improved upon:

- Bootloader integration is as yet incomplete. A dedicated tab will be added to the HotBlack Software Framework to quickly upload new firmware to the board. Such tab will allow to import and share code with robot makers in a similar fashion to the python sketches.
- The power system of the board had an ambitious aim. To be able to power the Raspberry through the board from a wide variety of common power sources. While it achieve this objective, it makes the system unstable at heavy loads (many servos) or at low voltages (single LIPO). The regulator has margin for improvements.
- Arduino Shield integration with the HotBlack Software Framework is as yet incomplete. A dedicated tab will be added to support existing Arduino Shield Sketches.
- Board optimization. Reduce area, optimize components, lower costs.

As a robot maker myself, I find this approach exceedingly effective, and I will keep working alongside HotBlack Robotics to ensure the success of this approach to robotic.

10.2 Example Applications

A few example of applications that can be made by taking advantage of the custom shield and the HotBlack Software Framework.

10.2.1 Autonomous Navigation

The system supports many kind of sensors thanks to on-board resources and expansion possibilities opened by the Arduino Shield Connector. Examples of sensors that can be integrated are:

- Odometry: Requires encoders on the wheels
- Absolute Position (open field): GPS receiver on Arduino Shield
- Inertial Navigation: IMU on Arduino Shield
- Visual Navigation: Camera and Image processing. SLAM

Autonomous navigation can be implemented by making full use of this system.

As an added value, image processing can be offloaded from the robot to a remote server, allowing for a more in depth analysis and a faster response. The cloud and the power it provides opens many possibilities.

10.2.2 Flying Drones

A Quad-copter uses four brushless motors that interface with a standard servomotor connector. An aeromodel uses one brushless motor with servomotor connector for the propeller, two servomotor axis for the tail (yaw and pitch) and one servomotor axis for the ailerons on the wings (roll), for a total of four servomotors

The shield developed for this thesis allows for four servomotors to be controlled. The custom shield is ideally suited as control system for flying drones.

The fact that the system can easily integrate a camera and sustain a video stream means that FPV remote control of flying drones is not only viable, but a natural application for this system.

10.2.3 UAV

Flying drones and autonomous navigation can be combined to build an UAV, Unmanned Air Vehicle, by making use of this system.

10.2.4 Remote IoT Node

Not all applications require motors. The HotBlack Software Framework can be used to simply setup the Raspberry as an IoT node. Applications include smart cameras, temperature measurement and more.

10.3 Final Remarks

As a robot maker myself, I find the approach described in this thesis to be exceedingly effective. I have personally been taking advantage of it for my recent robotic platforms and will continue to do so for the foreseeable future.

I will continue working with HotBlack Robotics to improve upon the board that has been designed and build upon the features of the Hot Black Software Framework, with the hope that this system will grow to see widespread adoption from robot makers all around the world.

The code, and gerber for this board can be found on my GitHub [37]. I have many ideas in mind for what to do next and what is to come, and the future looks bright indeed.

Glossary

Atmel: Company that manufactures popular microcontrollers. It was recently acquired by Microchip

Arduino: Company that manufactures a popular prototyping boards and provides a popular online IDE with a thriving open source community

Arduino Shield: A board that expands the functionality of an Arduino base board

ARM: A power efficient microprocessor architecture. Incompatible with X86 instruction set

BOM: Bill of Materials

Cloud: IT architecture involving remote servers and an internet connection

Cloud Robotics: Paradigm in which a Cloud architecture is used for a mobile platform

CRC: Cyclic Redundancy Check. A method to check the integrity of data.

FOM: Figure of Merit. A way to give a number to the fitness of a design to given metrics

FPV: First Person View. A method of control for remote platforms

Gerber: PCB layout production files

HotBlack: A Start-up in the sector of cloud robotics

IT: Information Technology

IoT: Internet of Things

LDO: Low Drop-out linear regulator

Linux: A popular open source Operating System

Linux Distribution: Custom version of a Linux OS. Countless distributions exists of varying degree of adoption, stability and usefulness

Microchip: Company that manufactures microcontrollers and low power ICs.

OS: Operating System

PWM: Pulse Width Modulation

PCB: Printed Circuit Board

ROS: Robot Operating System. A popular framework used in robotics

SEPIC: Single Ended Primary Inductor Converter. A switching regulator topology

UAV: Unmanned Air Vehicle

uC: Microcontroller

X86: An instruction set. X86 processors can run a staggering amount of legacy code

Index of Figures

Figure 1: Robokind [7][8][9].....	5
Figure 2: Small Mobile Robotic platforms based on Raspberry Pi [10][11][12].....	6
Figure 3: Flying Drones [13].....	6
Figure 4: Raspberry Pi Shields [16] [17] [18].....	8
Figure 5: HotBlack Cloud Architecture.....	12
Figure 6: Custom Board Architecture.....	13
Figure 7: SEPIC Topology.....	18
Figure 8: Architecture of the power system.....	19
Figure 9: Architecture of the protections.....	20
Figure 10: DC Motors Architecture.....	21
Figure 11: Schematics of the SEPIC regulator.....	28
Figure 12: LT Spice SEPIC Simulation Schematics.....	29
Figure 13: LT Spice Simulation. 11.1Vin, 2Aout.....	30
Figure 14: Schematics for the LDO Regulators.....	31
Figure 15: Schematics PCB layout.....	32
Figure 16: Test Power Regulator Board Layout Sectors.....	33
Figure 17: Power Regulator Test Board Preview.....	34
Figure 18: ROHM BD62321	36
Figure 19: Outline of the mini Arduino Shield.....	36
Figure 20: Schematics for the H-Bridge Test Board.....	37
Figure 21: Layout for the H-Bridge test board.....	38
Figure 22: Area allocation for the H-Bridge test board.....	39
Figure 23: Preview of the H-Bridge test board.....	39
Figure 24: Seeker of Ways.....	41
Figure 25: Architecture of Seeker of Ways.....	43
Figure 26: Motor Controller of Seeker of Ways.....	44
Figure 27: SoW Motor Controller Firmware Architecture.....	45
Figure 28: Modules.....	47
Figure 29: SoW Rosserial Slave Board.....	47
Figure 30: Block Schematics of SoW roserial Slave Board.....	48
Figure 31: ROS Publish-Subscribe Paradigm.....	55
Figure 32: HotBlack Cloud Robot Find.....	65
Figure 33: HBR Example Sketch LED blink.....	66
Figure 34: HBR ROS Node Map.....	67
Figure 35: Test Power Board Power MOS Correction.....	69
Figure 36: Testing of the Test Supply Board	70
Figure 37: Test Supply Board with Seeker of Ways.....	71
Figure 38: Test H-Bridge Board.....	72
Figure 39: Test H-Bridge board using an Arduino.....	72
Figure 40: HW Architecture of the custom Raspberry Pi shield.....	75
Figure 41: Shield Schematics - Battery Connector and Protections.....	76
Figure 42: Shield Schematics - SEPIC Regulator.....	78
Figure 43: Shield Schematics - Linear Regulators.....	79
Figure 44: Shield Schematics - H-Bridges.....	80
Figure 45: Shield Schematics - Microcontroller Filters and Services.....	81
Figure 46: Shield Schematics - Microcontroller.....	82
Figure 47: Shield Schematics - Raspberry Pi GPIO.....	83

Figure 48: Shield Schematics - Arduino Shield.....	84
Figure 49: Shield Schematics - Servomotors.....	85
Figure 50: Shield Schematics - Encoders.....	86
Figure 51: Board - Form Factor.....	88
Figure 52: Board - Component Placing.....	89
Figure 53: Board - Layout TOP Layer.....	90
Figure 54: Board - Layout BOT Layer.....	91
Figure 55: Board - All Layers.....	92
Figure 56: Eurocircuit Board Aspect.....	93
Figure 57: Shield PCB.....	95
Figure 58: Custom Raspberry Pi Shield Assembled.....	95
Figure 59: Custom Raspberry Pi Shield Assembled - Bottom View.....	96
Figure 60: Application - CV Orb.....	100
Figure 61: Application - FPV Keyboard Control Browser Page.....	102

Bibliography

- [1] : ROS, Robot Operating System, <http://www.ros.org/>
- [2] : ROS, roserial protocol, <http://wiki.ros.org/roserial>
- [3] : Raspberry Pi, Raspberry Pi, <https://www.raspberrypi.org/>
- [4] : Arduino, Arduino Uno Board, <https://www.arduino.cc/>
- [5] : Arduino, Arduino online IDE, <https://create.arduino.cc/>
- [6] : HotBlack, HotBlack Robotics, <http://www.hotblackrobotics.com/>
- [7] : Northrop Grumman, Northrop Grumman X-47B, <http://www.northropgrumman.com/Capabilities/AutonomousSystems/Pages/default.aspx>
- [8] : FANUC, FANUC Arm 2000iB, <http://www.fanuc.eu/it/en/robots/robot-filter-page/r-2000-series>
- [9] : Gilles Caprari, EPFL Autonomous Systems Lab Swarm Robotic Platform, <https://asl.epfl.ch/>
- [10] : PiBot, PiBot, <http://www.pibot.org/>
- [11] : Dexter Industries, GoPiGo, <https://www.dexterindustries.com/gopigo3/>
- [12] : PiBot, TiddlyBot, <http://www.pibot.org/tiddlybot/>
- [13] : HobbyKing, Bixler Plane Model, https://hobbyking.com/en_us/hobbykingtm-bixlertm-v1-l-epo-1400mm-arf.html
- [14] : FabLab, MiniMakerFaire, <https://torino.makerfaire.com/>
- [15] : OGR, Droidcon Torino, <http://it.droidcon.com/2018/>
- [16] : Pridopia, PiPSU, <http://www.pridopia.co.uk/pi-power-bank.html>
- [17] : Cooking-Hacks, Cooking Hacks, <https://www.cooking-hacks.com/>
- [18] : Anavi, RabbitMax, <http://anavi.technology/>
- [19] : HotBlack Robotics, HBrain Image, <https://sourceforge.net/projects/hbrain/>
- [20] : Arduino, Arduino YUN Board, <https://store.arduino.cc/usa/arduino-yun>
- [21] : Marsboard, Marsboard-SOM-RK3066, <http://www.marsboard.com/>
- [22] : UDOO, UDOO X86 SBC, <https://www.udoo.org/>
- 23: Beagleboard, Beagle Bone Black,
- [24] : Atmel, AtMega 328P, <http://www.microchip.com/wwwproducts/en/ATmega328p>
- [25] : Atmel, ATmega 644, <https://www.microchip.com/wwwproducts/en/ATmega644PA>
- [26] : Atmel, AvRisp MKII, <http://www.microchip.com/developmenttools/productdetails.aspx?partno=atavrisp2>
- [27] : Linear Technologies, LT1619, <http://www.linear.com/product/LT1619>
- [28] : Texas Instruments, LM3488, <http://www.ti.com/product/LM3488>
- [29] : Texas Instruments, AN-1484, <http://www.ti.com/litv/pdf/snva168e>
- [30] : Exar, SPX29302, <https://www.exar.com/product/power-management/power-conversion/ldos-and-regulators/linear-ldos/spx29302>
- [31] : Eurocircuit, PCB Manufacturer, <https://www.eurocircuits.com/>
- [32] : ROHM, BD62321, <http://www.rohm.com/web/eu/products/-/product/BD62321HFP>
- [33] : Texas Instruments, LMD18200, <http://www.ti.com/product/LMD18200>
- [34] : Atmel, Atmel Studio 7, <https://www.microchip.com/avr-support/atmel-studio-7>
- [35] : Raspberry Pi, Raspicam, <https://www.raspberrypi.org/products/camera-module-v2/>
- [36] : OpenCV, OpenCV library, <https://opencv.org/>
- [37] : Orso Eric, GitHub Repository, <https://github.com/OrsoEric>