NTNU – Trondheim
Norwegian University of
Science and Technology

Eurobot NTNU 2012

Treasure Island

Are Halvorsen
Sindre Røkenes Myren
Andreas Hopland Sperre

Master of Science in Engineering Cybernetics
Submission date: June 2012
Supervisor: Sverre Hendseth, ITK
Co-supervisor: Snorre Aunet, IET

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Project description

An autonomous robot to compete in Eurobot Open 2012 shall be designed and built.

Assignment Given: 1st of February 2012

Supervisor: Sverre Hensedth
Deparment of Engineering Cybernetics
Co-supervisor: Snorre Aunet
Deparment of Engineering Electronics

Preface

This paper presents the work performed by Sindre Myren and Andreas Sperre, from department of Engineering Cybernetics, and Are Halvorsen from department of Electronics and Telecommunicationsat NTNU. The work has been done as a group, and should weight equally.

We would like to thanks our sponsors, Kongsberg Defence and Aerospace ASA, SINTEF and the department of Engineering Cybernetics at NTNU, for the financial support. Without it we would never have had the chance to build a robot and travel to La Ferté-Bernard in France to compete in Eurobot 2012. We would also like to thank our supervisors Sverre Hendseth and Snorre Aunet for their support and letting us work freely with a practical project.

Further we would like to thank Stefano Bertelli at the department of Engineering Cybernetics for helping us with the financial support and various organizational tasks. We would also like to thank the various workshop employees at NTNU for the advice they have given us. Especially the mechanical workshop at the department of Engineering Cybernetics for producing and giving feedback on all the necessary mechanical components for the robot and the engineers at the Department of Electronics and Telecommunications for helping us with soldering of surface mount packages with their fancy soldering machines.

We would like to thank Elisabeth Kjellmo Nicolaysen for taking the time to paint our robot, it gave our robot a special look that was noticed and praised during the competition in France.

Last, but not least, we would like to thank Steffen Johnsen, Kristian Klausen, Adam Leon Kleppe, Lars Espen Nordhus, John Magne Røe and Leif-Julian Øvrelid. These students were involved in the project through the course Experts in Team at NTNU. All their hard work and vital contribution to the robot design resulted in a robust and easily maintainable robot.

Sindre Myren

Are Halvorsen

Andreas Sperre

Abstract

Eurobot is an annual competition for autonomous robots. Typically two teams compete against each other for 90 seconds on a 2×3m playing area. The main goal is to collect as many points as possible. There are several matches to determine which robot is the best. The rules are different every year. An autonomous robot for this year's competition was designed and built.

The rules for Eurobot 2012 were studied and a design concept was created. In order to implement the design a series of technical pieces of work was carried out. The tasks involved several fields of study including engineering cybernetics, electrical engineering, computer science and mechanical engineering.

A laser-tower positioning system from 2010 was further developed and improved. In addition the robots drive wheels hall-sensors were used to compensate for the robots movement. An *extended Kalman filter* was created to transform these measurements into a position and orientation estimate. Two PID regulators were used to maneuver the robot, one regulator controlling the rotation, the other translation.

A circuit board following the EPIC-plus standard with several efficient power supplies, a micro controller and a circuit board stack was designed, produced and tested. This circuit board was driven by a lithium battery and acted as a power supply for-, and took care of low level interaction with, all motors, servos and actuators on the robot. Firmware was implemented on the circuit board that provided an interface to control all hardware via CAN-bus to a tablet PC.

To implement strategic choices, algorithms and artificial intelligence, an elaborate software system was created. The high-level programing was done in *Go*, a new and exciting programing language from Google. A featured Debug-GUI that presented real-time information and allowed for robot interaction was provided. A strong focus on design and a test driven development, resulted in robust and stable software.

A mechanical design of the robot was created in collaboration with a group of students through the course TTK4850 “Experts in team” at NTNU.

This work lead to a complete robot with a clean implementation hosting advanced technical solutions. The final software allowed strategies to be reprogrammed before each match, and the physical robot was easily maintainable. The positioning system can move the robot to any coordinate on the playing area. In the Eurobot 2012 competition, the robot won three out of five matches and ended at 23rd place out of 43 international teams.

Sammendrag

Eurobot er en årlig konkurranse for autonome roboter, der to lag konkurrerer mot hverandre på et 2 × 3m spillebrett i 90 sekunder. Robotenes mål er å samle så mange poeng som mulig gjennom flere kamper. Reglene er forskjellige fra år til år. En autonom robot er blitt designet og bygget for årets konkurranse.

Reglene for Eurobot 2012 ble studert, og et designkonsept utredet. For å implementere designet måtte en rekke tekniske aspekter utforskes. Oppgaven involverte flere fagfelt, blant annet teknisk kybernetikk, elektronikk, informatikk og maskinteknikk.

Lasertårn-posisjoneringssystemet fra 2010 ble videreutviklet og forbedret. I tillegg ble hallsensorene på robotens drivhjul brukt til å estimere robotens bevegelse. Et *extended Kalman filter* ble designet og implementert for å overføre disse målingene til et posisjon- og rettningsestimat. Deretter ble to PID-regulatorer lagd for å kjøre roboten. En av regulator styrer rotasjon og den andre translasjonen.

Et kretskort med flere effektive strømforsyninger, en mikrokontroller og en kretskort-stack som følger EPIC-plus standard ble designet, produsert og testet. Denne kretsen ble drevet av et litiumbatteri, og fungerte som kommunikasjons-hub samt strømforsyning for alle motorer, servoer og aktuatorer på roboten. Disse enhetene ble styrt fra en tablett-PC over en CAN-bus.

Et programvaresystem ble opprettet for å implementere strategiske valg, algoritmer og kunstig intelligens på tabletten. *Go*, et nytt og spennende programmeringsspråk fra Google, ble brukt for å implementere denne logikken. Et grafisk brukergrensesnitt, som presenterer informasjon om roboten i sanntid, ble lagd for å lette feilsøking samt styre roboten. Et sterkt fokus på design og "test drevet utvikling", resulterte i en robust og sikker programvare.

I samarbeid med en gruppe elever fra TTK4850 "Eksperter i Team", ble et mekaniskdesign utrettet. Dette mekaniske arbeidet gav en robotplattform som var lett å jobbe med og teste på.

Resultatet ble en komplett, fungerende robot, med gode mekaniske løsninger for å støtte opp om avansert elektronikk. Det utviklede styresystemet tillater hurtig re-programmering av strategier, selv like før en kamp. Posisjoneringssystemet kan nå frakte roboten til et hvilket som helst koordinat på spillebrett. Roboten vant tre av fem kamper i *Eurobot 2012* noe som resulterte i en 23. plass utav 43 internasjonale lag.

Contents

Part I

Introduction

Chapter 1

Introduction to Eurobot

This report covers design and implementation of an autonomous robot for use in the competition Eurobot. In order to build an autonomous robot a wide variety of disciplines from several fields of study is required, including but not limited to: electronics, engineering cybernetics, mechanical engineering and computer science.

1.1 What is Eurobot?

Eurobot [15] is an international, annual competition for autonomous robots. It started in 1994 as a national competition in France. In 1998 it became an international competition with nine teams from five countries. In the beginning the competition was known as "Eurobot Open", but it was shortened to "Eurobot". The competition focuses on fair play and sharing of knowledge, as can be seen by this quote from their homepage.

> "Eurobot values fair play, solidarity, technical knowledge, sharing and creativity, both through techniques and project management more than competition. The contest aims at interesting the largest public to robotics and at encouraging the group practice of science by youth. Eurobot and its national qualifications are intended to take place in a friendly and sporting spirit."

Over the years the competition has accumulated quite a few rules. And it is now fairly standardized that two autonomous robots competes for 90 seconds on a 3x2 meter playing area.

1.2 Eurobot-NTNU

Eurobot-NTNU is a Eurobot team from the Norwegian University of Science and Technology. The team consists mostly of students writing master-thesis or pre-master project at the university. As such, each year the team is reconstructed with new people.

The name Eurobot-NTNU reflects on the organization, while each team usually goes under the name "Legend of Norway". The only exception is from 2009, when they were known as "Lost Vikings".

The team first competed in the 2000, and has since taken part every year. The best result was a 4th place in 2009. But the team has since taken a hit, with two years without making the qualifications.

Table 1.1 shows "Legend of Norway's" historical accomplishments. Note that the number of teams is counted after the national finals, limiting it to max three teams per nation.

Year	Location	Theme	№Teams	№Nations	NTNU's result
1998	France	Football	9	5	-
1999	France	Castles	8	5	-
2000	France	Fun Fair	12	7	8.
2001	France	Space Odyssey	19	12	13.
2002	France	Flying Billiards	27	17	17.
2003	France	Heads or Tails	32	19	16.
2004	France	Coconut Rugby	41	21	21.
2005	Switzerland	Bowling	50	22	11.
2006	Italy	Golf	55	23	44.
2007	France	Recycling	50	27	25.
2008	Germany	Mission to mars	39	24	31.
2009	France	Temples of Atlantis	43	26	4.
2010	Switzerland	Feed the world	46	23	DNQ
2011	Russia	Chess'Up!	41	23	DNQ
2012	France	Treasure island	43	17	23.

Table 1.1: List of themes and results

1.3 Disposition of the report

Part I Introduction starts out by explaining what Eurobot is, and what NTNU's role has been in the competition. Chapter 2 outlines work that is not directly related to the robot, but is necessary to succeed as a team. Then this year's Eurobot rules are explained in chapter 3.

Part II Robot Design, starts out with discussion of strategic decisions regarding how the robot should collect points and behave on the playing area. This leads to a series of design choices regarding propulsion systems, positioning, electronics, programming languages and mechanical design. These design choices lead to a complete robot design that is presented in chapter 11. The systems and methods chosen in part II had to be designed and implemented specifically for the competition.

Part III Implementation and Improvements, dwells into three major technical systems designed for the competition. The positioning system, the electronics system referred to as a power card and the software implementation. This work has produced a lot of digital material including source code and design files for mechanical and electrical parts. These can all be found as digital attachments to the report, consult appendix G for more information.

Part IV End Result of the report outlines results for the robot as a whole and discusses them. Each section in part II and III has their own results and discussions. After the discussion, a principal conclusion is given. At the end, suggested future work on the individual modules is presented.

Chapter 2

Management of Eurobot-NTNU

To get the robot described in this article built, a lot of practical work needed to be done. Some of the work may not be very academically relevant or scientifically documented, but nonetheless a considerable amount of time has been spent on it. As such, the most time consuming practical work is listed here.

2.1 Organization management

The control of the nonprofit organization “Eurobot-NTNU” and all of its assets have been transferred to this year’s master students. This includes online banking, web servers, equipment and previous works by Eurobot-NTNU. A budget has been made, and all financial transfers have been accounted for.

A team of six students attending the course TTK4850 “Eksperter i Team - Byggelandsbyen” (EiT) assisted in the development of the robot [26]. The team has been of great support by far outweighing the time spent managing them. The team’s main assignment has been to come up with design concepts regarding how the robot should behave and make the actual mechanical design of the robot.

2.2 Sponsors

Like all larger projects funds are needed to build things. This is especially true for Eurobot teams as they also have to cover the cost of travel to the competition and living expenses during the competition. A good amount of time has been spent on finding sponsors and figuring out what to do if we didn’t find any.

2.3 Promotional work

Eurobot demands that each attending team has to create and print a promotional poster for the robot they built. The poster can be found in appendix F.

The robot and playing area is also used to promote the department of Cybernetics Engineering. The project is shown off during technological events in the local community, to young minds that may become students at NTNU as well as people from the industry attending seminars.

2.4 Creation of test equipment

A physical version of this year's playing area as seen in figure 2.1 was created for testing of robot. The playing area was an integral part of testing the system before the competition. The specifications for the playing area can be seen in figure 3.1.

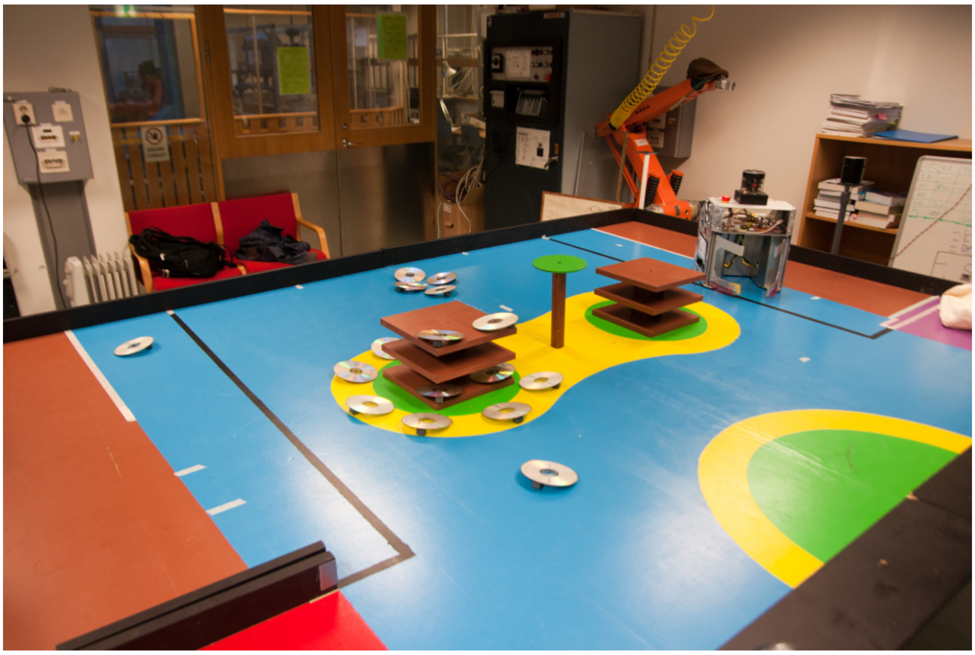


Figure 2.1: Test table

Chapter 3

Eurobot 2012 rules

This chapter will give a brief introduction to the Eurobot 2012 “Treasure Island” rules [52]. The complete set of rules as defined by Planet-Science can be found as a digital attachment to the report, refer to appendix G.

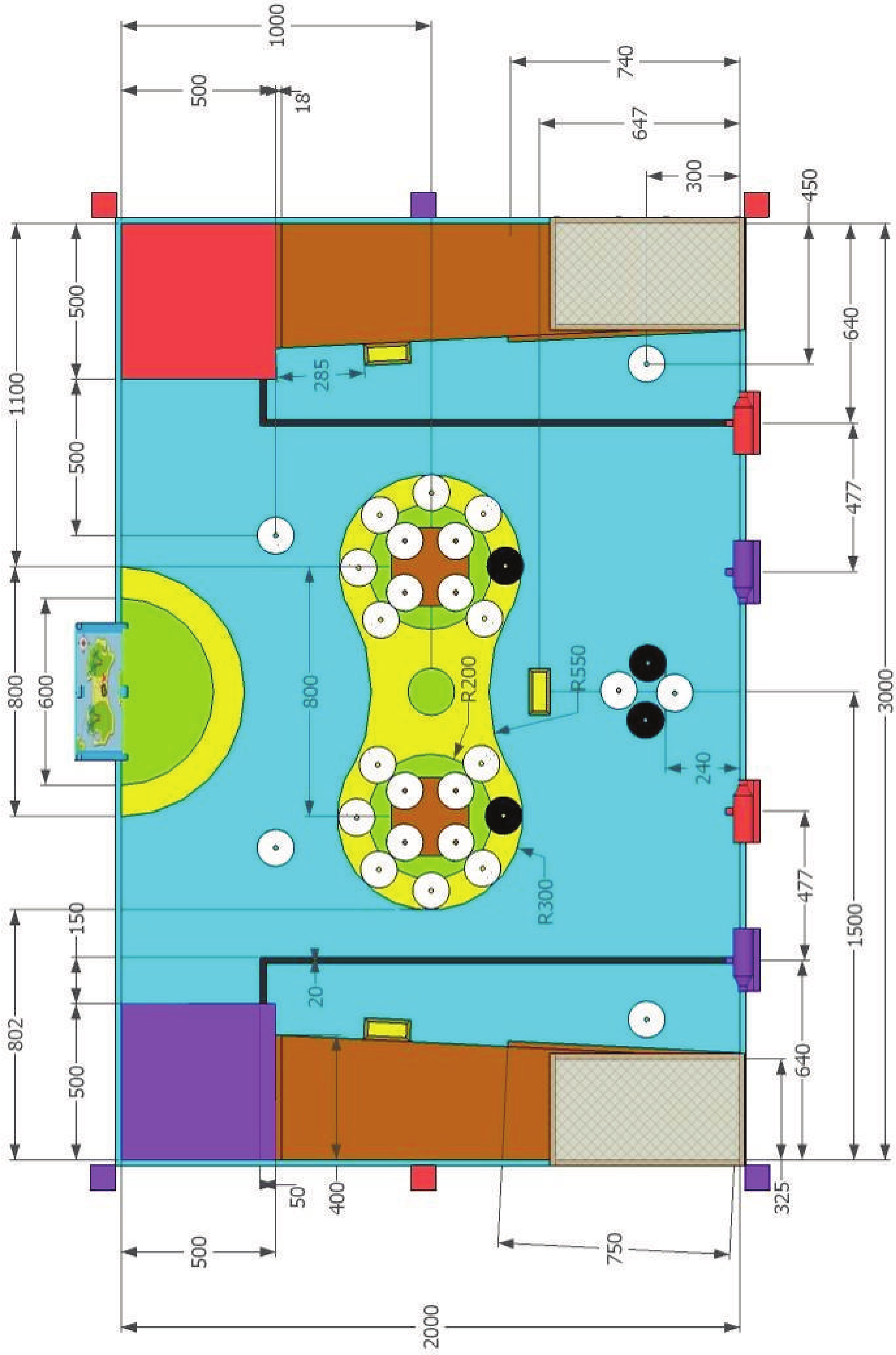


Figure 3.1: Playing area top-view

3.1 Persistent rules

As mentioned in section 1.1 Eurobot has a number of returning “De facto” rules. Each year the rules are based on the previous year’s rules, but with a new theme and new objectives. For instance, table dimensions and safety regulations are typical recurring regulations only undergoing minor changes.

As such this year, just as previous years, a match is 90 seconds long and consists of two autonomous robots competing to collecting the most points. Or rather, this year there are two teams, but up to four robots, as each team are allowed to use a small and a big robot. One team is known as the red team and the other is the violet team, named after the color their starting area.

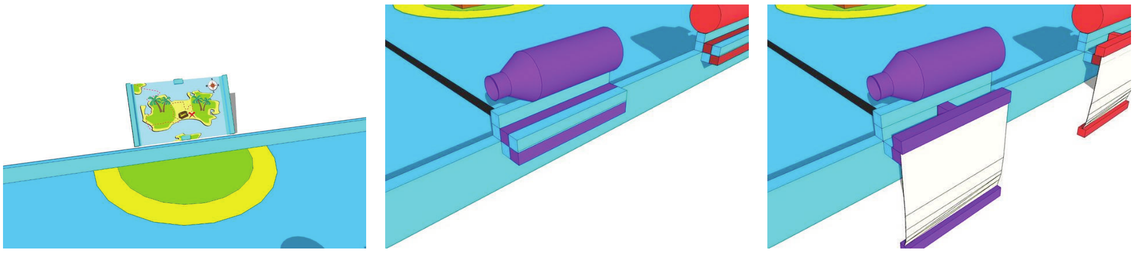
3.2 Playing area layout

This year’s playing area is show in figure 3.1. It is based on the theme “Treasure Island” and as such the blue represents a sea, the green a jungle and the yellow a beach. The violet and red squares on the corners are the starting areas, sometimes referred to as the *captain’s quarters*. In addition to the starting area each team have a *ship*, a *map* and two *bottles*.

A team’s *ship* consists of its starting area and the bordering brown region. The brown region is split into two regions, under the lid in the corner is the *ships hold*, and the rest is called the *loading deck*.

The *map* is painted on the wall seen in figure 3.3. Two pieces of removable fabric covers the map, on for each team. Each piece of fabric is attached to the map by small pieces of Velcro at each corner.

Each *bottle* consists of one big push-button. When completely pressed a flag pops out like in figure 3.2c. Both teams have a black line leading from the starting area leading to their closest bottle.



(a) The map revealed (b) Bottle start configuration (c) Pushed button

Figure 3.2: Points of interest

In the middle of the playing area there are two brown totem poles. Each totem pole has three levels. The top and bottom level holds four coins each, while the middle one holds two gold bars as shown in figure 3.3.

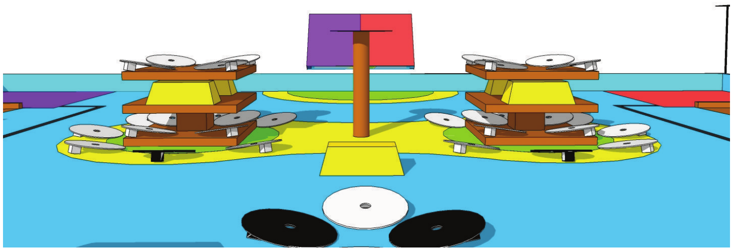


Figure 3.3: The two totem poles, filled with coins and gold bars

In addition to the 4 gold bars and 16 coins in the totem poles, 3 Gold bars and 22 coins are placed as shown in figure 3.1, giving a total of 7 gold bars and 38 coins placed on the playing area each match. Four of the coins are painted black, these coins does not award any points. The position of all the coins is known, but the black coins are placed randomly each match.

3.3 Game objectives

The main objective of the game is to gain more points than your opponent. However in the qualifying rounds it is also important to score as many points as possible, as this will increase your ranking. When the final rounds begin it is all or nothing, one loss and you're out.

Points are awarded accordingly:

- 1 point for every white coin completely inside of the players ship
- 3 points for every gold bar completely inside of the players ship
- 5 points for each button of the players bottles completely pushed in
- 10 points for reviling the players map
- 10 points for victory, 5 for draw, 2 for losing, 0 in case of disqualification

In general it is not illegal to steal or remove points from the opponent. If you choose to do so extra care must be taken not to unnecessary block the opponent. Also, while each team may collect as many coins and gold bars in it ship as it likes, the captain's quarters only holds a maximum of 5 points. In return, it is illegal to steal from the captain's quarters.

It is also important to note the “Fair-play” rule, for instance it is illegal to purposely block the other robot, except in front of your own ship. Penalties will be given for colliding with the enemy, unfair play or destroying elements of the playing field.

3.4 Matches

Initially, there are five rounds of matches where two and two teams compete against each other. Every team accumulates point through these matches by collecting points on the playing area and get bonus points for winning a match by scoring the most points.

After the fifth round the 16 teams with the most accumulated points are transferred to a cup style competition, the playoffs. Here teams are paired up, the winning team advances in the competition while the losing team is out.

3.5 Robot design constraints

Each team may have up to two robots of different maximum dimensions. The biggest robot must have a starting perimeter under 1200mm, and a deployed perimeter below 1600mm. The smaller robot must be less than 800mm at start up, and 1000mm deployed. Both robots must fit inside the starting area square at the same time.

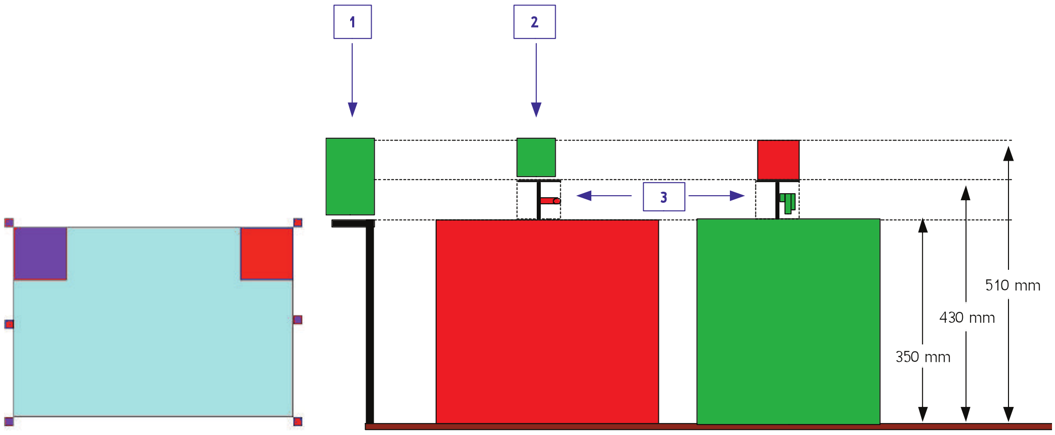
To avoid cheating, the robot must be started through a 500mm long starting cord, and after the match the robots must automatically shut down leaving its actuators limp. The robot must have an obstacle avoidance system, so as to not crash into the other robots. To help simplify this system, any robot cannot be hollow or use colors similar to the ones on the playing area.

3.5.1 Beacon location system

Each team is allowed to have a beacon location system [52, p. 24]. The playing area has six beacon supports, three per team, as shown in figure 3.4a In addition each team may put a beacon on the opponent's robots, as shown in figure 3.4b. All the beacons have to be 80x80x80 millimeters except for the fixed beacons, placed around the playing area, which can be twice as high. Every beacon support has Velcro hooks on them, for quick attachment.

3.6 Safety constraints

In general the robots must conform to national and European laws and specifications. If any device or system is considered potentially dangerous, it will be rejected. For instance there may not be any dangerous protruding sharp parts. There also has to be an emergency stop button located at the top of the robot, this button has to cut the voltage to any actuators/motors on the robot.



(a) Beacon placement

(b) Vertical placement

Figure 3.4: Beacon positioning

3.6.1 Lasers and lights

Lasers used must conform to "EN 60825-1:2007", and be of class 1 or 1M [52, p. 22]. Lasers not projected outside of the playing area may be of class 2 or 2M. High power light sources are allowed even though they may be dangerous for the human eye. Though the general rule "no dangerous parts" still applies.

3.6.2 Energy Source

The Eurobot 2012 rules state that all kinds of energy sources are allowed, unless they involve combustion/pyrotechnic process [52, p. 19]. The voltage inside the robot has to be "low voltage" and may never exceed 48V, unless inside a sealed unmodified commercial product. There are also some new special rules regarding safety of lithium batteries, they have to be contained in fireproofs bag at all times and battery chargers have to be shown during homologation.

All pressure air systems must comply with "Conseil Général des Mines" from 1943. They must not exceed 4 bars, and $pressure * volume$ may not exceed 80 bar.liter.

3.7 Homologation

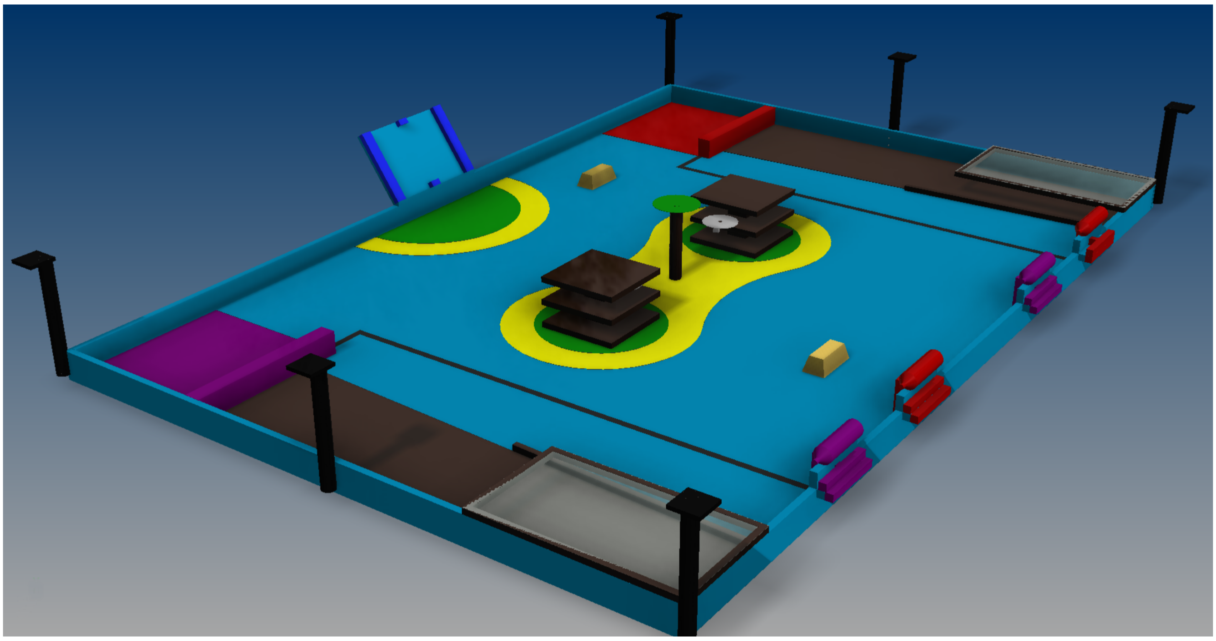
Before any team can play matches it has to pass homologation. A series of tests will be performed by the judges on the robot, they check that the robot conforms to the rules and that it is able to score points in a match. During the homologation match the judges will place a dummy robot on the playing field to test that the real robot won't crash into it.

Part II

Robot Design

Chapter 4

Deciding on a strategy



Playing area for Eurobot 2012

The main goal for the Eurobot competition is to score points. In chapter 3 it was explained that the first matches would be organized as series, whereas the finals would be organized as a cup. The main task to solve for Eurobot 2012 is therefore twofold:

1. In the series, score as many points as possible in 90 seconds

2. In the finals, score more points than the opponent in 90 seconds

4.1 Strategic goals

In 2010 and 2011, Eurobot NTNU was not able to pass the homologation test. This year it was therefore considered more important to be able to compete than it was to score a lot of points. With this in mind, the following goals for the strategy were constructed:

1. Allow the robot to homologate
2. Be simplistic
3. Score safe points
4. Allow reuse of components and concepts from previous robots

4.2 Available concepts

Since the task to solve is new each year, one cannot simply base the strategy on previous year's solutions. There is therefore a certain need for innovation when coming up with the main strategy. It would however still be wise to form a strategy that allows reuse of components and concepts from previous years.

When it comes to the robot's intelligence, the EiT group developed an artificial intelligence implementation, using utility theory. The robot's actions would be grouped into tasks, and all tasks would be given an initial utility. The utilities could be recalculated in-game. When a task is completed, its utility should simply be set to zero. The robot should then choose from a pool of tasks, and perform the task that had the highest utility.

As an alternative to using a utility based artificial intelligence, a more static solution could be chosen.

4.3 Decision

It was the EiT group who decided the main strategy for how the robot should collect its points [26]. However, most of the specific tactics were decided by the master students.

4.3.1 The main strategy

As mentioned in chapter 3, it is allowed to have two robots. However, as we were a rather small team, it was decided to focus on creating just one robot.

It was decided that the robot should be hollow, and that it should have two doors or arms in the front that could be opened and closed. These arms should be able to push coins down from the top and bottom levels of the totem poles. Coins could then be pushed in front of the robot, or inside the hollow center. On each side of the robot, there should also be a wing with a pushing mechanism in the back. These wings could be used to catch gold bars from the middle level of the totem poles during a drive-by action.

As mentioned in chapter 3, the fabric covering the map is worth five points, if collected. The fabric was specified to be a piece of textile attached to a skew board with Velcro in each corner. However the particulars around the task as a hole were very poorly specified. E.g. the exact fabric used and the size of the corner hooks was not specified. It was therefore decided not to implement any tactics or physical devices for collecting these five points.

4.3.2 Game tactics

Artificial intelligence

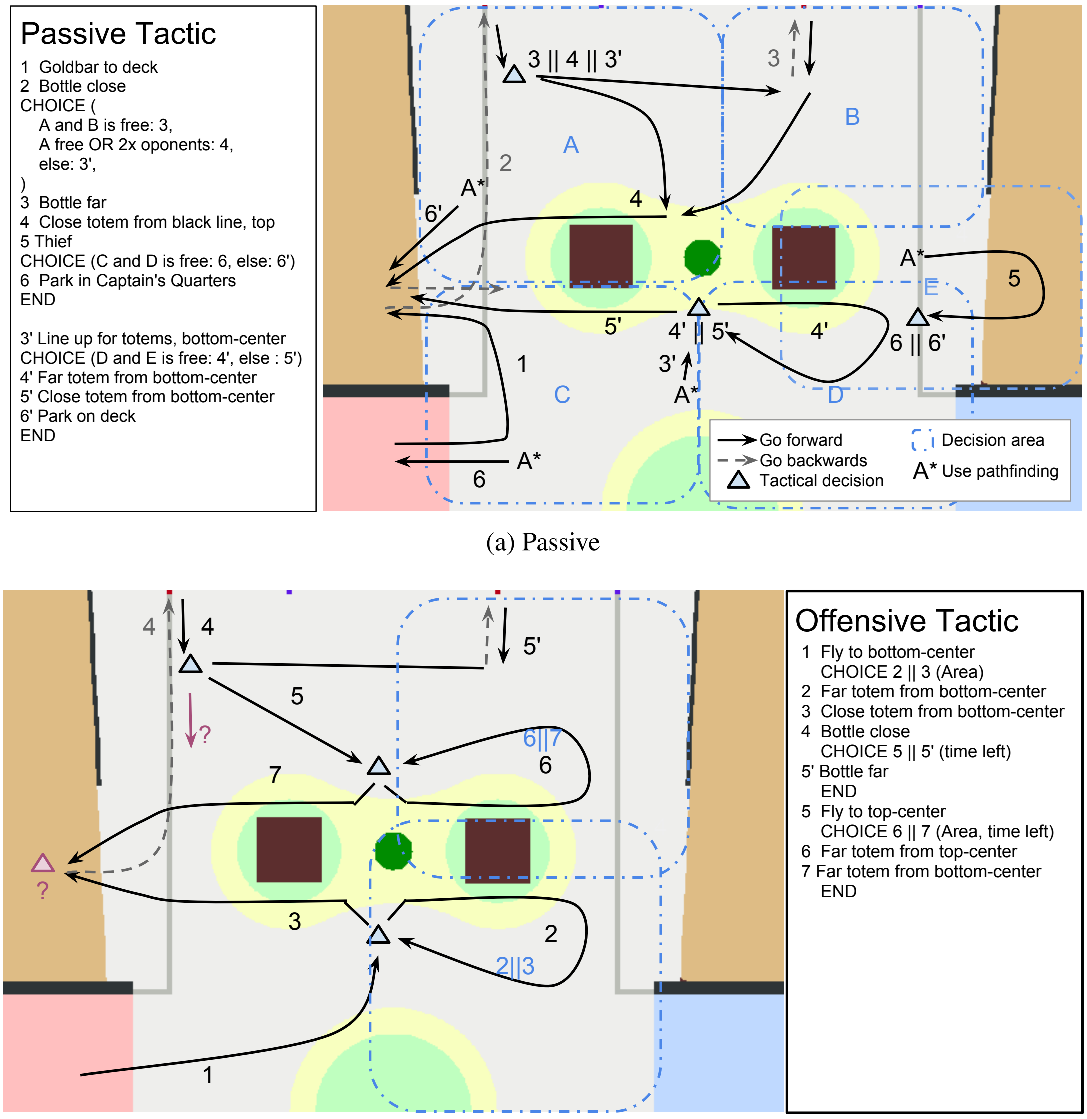
It was decided that a set of pre-defined tasks would be coded, and that each of these tasks would have a defined starting position. Example tasks are *drive-by totem*, *back into button*, *push coins to deck*. What remain to clarify is how these tasks should be connected, and how the robot should choose what task to perform.

The utility based artificial intelligence implementation by the EiT group had some obvious problems. First of all, it would be hard to set the utilities in an intelligent matter. For instance, the utility would vary based on time left, previous performed tasks, the robot's position, the opponent's position, and possibly other factors. More importantly, the utility approach makes it hard to predict what order the tasks are performed, and thus hard to test. A more static solution, where the order of which the tasks are performed is known, would be easier to test. The artificial intelligence developed by the EiT group was thus discarded.

It was decided to instead place the tasks inside a decision tree, as illustrated in figure 4.1. At certain positions in the tree, the robot should do choices. In the figure, a triangle marks the location of these decisions. To decide which task to do next, the programmer should be able to check the remaining time, the position of the opponent robots, or any other accessible data. This way of implementing the tactics, allows a small finite number of possible routes for the robot to follow. Each variant of the tree could be tested and tuned. The robot might still do intelligent choices, but rather than a general utility algorithm, it would be preprogrammed.

Obstacle avoidance

There are two types of obstacles that need to be avoided on the playing area. The first type is static objects, like the totem poles, the walls, etc. The second type is other robots. Being able to stop in case the opponent robot gets in the way is part of the homologation test. In addition,


Passive Tactic
1 Goldbar to deck
2 Bottle close
CHOICE (
A and B is free: 3,
A free OR 2x oponents: 4,
else: 3',
)
3 Bottle far
4 Close totem from black line, top
5 Thief
CHOICE (C and D is free: 6, else: 6')
6 Park in Captain's Quarters
END
3' Line up for totems, bottom-center
CHOICE (D and E is free: 4', else : 5')
4' Far totem from bottom-center
5' Close totem from bottom-center
6' Park on deck
END
3 || 4 || 3'
3
A
B
2
A*
6'
4
A*
5
E
5'
4' || 5'
4'
6 || 6'
1
3'
C
A*
D
A*
6
Go forward
Go backwards
Tactical decision
Decision area
A* Use pathfinding
4
4
5'
5
?
6||7
6
7
?
3
2
2||3
1
Offensive Tactic
1 Fly to bottom-center
CHOICE 2 || 3 (Area)
2 Far totem from bottom-center
3 Close totem from bottom-center
4 Bottle close
CHOICE 5 || 5' (time left)
5' Bottle far
END
5 Fly to top-center
CHOICE 6 || 7 (Area, time left)
6 Far totem from top-center
7 Far totem from bottom-center
END

(a) Passive

(b) Offensive

Figure 4.1: Plan for two different game tactics

a penalty will be given if the robot crashes into the opponent during a match.

To avoid crashing into static objects at the playing area, a pathfinder algorithm can be used. This algorithm should be used to travel between the tasks mentioned in section 4.3.2. Within each task, the use of the pathfinder algorithm should not be necessary.

It was decided that this year's robot would have an opponent avoidance system similar to the one used in 2009 [28, p. 76]. If the opponent gets within the *watch area* shown in figure 4.2, it should slow down. If the opponent is detected inside the *stop area*, the robot should go to a complete halt. For the rest of this section, we will refer to this position as the *halted position*.

If the time frame allows it, a routine to get out of the halted position could be implemented. Given that such a routine is implemented, our robot also needs to figure out what to do next. This is a somewhat complex task, and difficult to test for every location on the playing area that

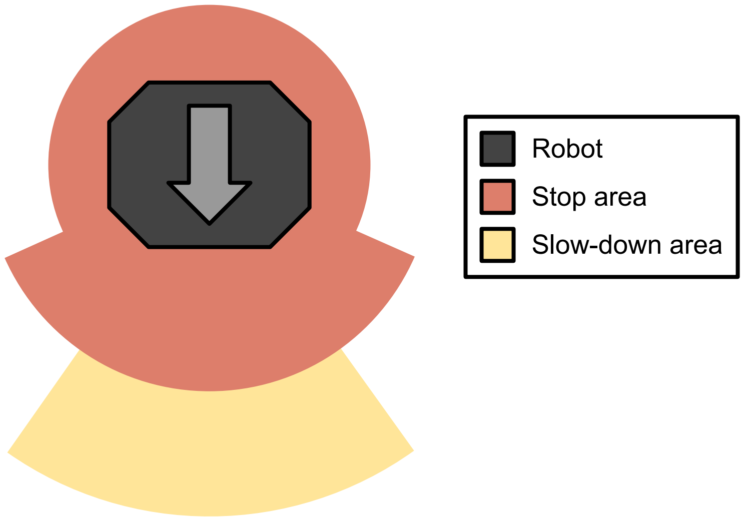


Figure 4.2: Areas for opponent avoidance

such a situation might occur.

It can also be discussed whether backing out is always the right strategic choice. For instance, if the opponent has a way to back out of the halted position, the best action for our robot to perform would be nothing. That way our robot can continue to do it's task as soon as the opponent has moved away.

Also, a better strategy than having to back out from the halted position, might be to not get in the halted position in the first place. It was therefore decided that the implementation of this routine should be of low priority.

4.4 Results

It was decided that only one robot should be built. It should be a hollow robot with two arms and two wings that allowed it to collect gold bars and coins from the totem poles. It should be able to move in curves, and it should use a decision tree to make tactical choices. To avoid static objects on the playing area, a pathfinder algorithm should be implemented.

Chapter 5

Defining a propulsion and navigation system

Figure 5.1: Maxon EC 45 flat

To fulfill the strategic requirements from chapter 4 a way of moving around the playing area was needed. This chapter will perform the necessary study of previous robots to conclude what propulsion system to use.

5.1 Requirements

The strategic goals from section 4.1 was re-tailored for the propulsion system, the result can be seen here:

- Be simple, robust and tested
- Use existing modules
- Run both forwards and backwards (reverse)

5.2 Previous solutions

The articles and datasheets from Eurobot-NTNU were studied to figure out what had previously been done. In addition some other interesting solutions were studied.

1998 Four wheels

In 1998 the robot “Marvin” had 4 wheels and was an articulated vehicle [58, p. 35]. Marvin could be controlled by controlling the pivoting point. The robot used a target seeking algorithm to navigate.

2004 Two wheels

In 2004 a two wheel drive system with two supporting ballpoint wheels was used [14, p. 16]. To control the robot an advanced Line of sight (LOS) algorithm was used [34, p. 17].

New mechanics but the same system was reused in 2005, but with a slightly more advanced regulator.

2006 Belt drive

In 2006 a tank-like belt drive system was used. The robot still used a LOS algorithm, something that worked well except for some over steering [16, p. 77].

2007 Two wheels

In 2007 and 2008 they switch back to a two wheel drive system. The LOS algorithm was still in use, now with two PID-regulators on top controlling the rotation and translation [37, p. 33].

This time the drive wheels were placed towards the front of the robot. In 2009 they moved the drive wheels back to the center of the robot [28, p. 50]. The LOS algorithm was thrown away, characterized as overly complicated, designed for boats prone to drifting [28, p. 128]. Instead, two PID-regulators were used. One to control the heading and one to control the translation.

In 2010 they changed to a new motor controller and new smaller motors [56, p. 39]. The new brushless DC motor, Maxon EC-45 Flat, can be seen in figure 5.1.

In 2011 the electronics module was changed to two Maxon EPOS2 24/2 motor controllers [38, p. 33]. The two PID-regulators from 2009 were re-implemented with only minor differences [55, p. 30]. The robot would turn first, then drive in a straight line, stop rotate and repeat.

Other options

Other solution previously discussed includes Omni and Mecanum -wheels [30, p. 16].

5.3 Decision

5.3.1 Hardware

To be able to have a simple robot that works, and to allow reuse of previous year's hardware, it was decided that the robot's propulsion should be based on a two wheel drive system. The drive axis should be placed as close to the middle as possible, and the wheels as far out as possible.

The current hardware, EC-45 flat motors and EPOS2 controllers, seems to be stable and working [54, p. 37]. Thus there's no need to change the hardware. For optimal functionality each motor controllers requires 24V and up to 2 ampere continuous [33, 32, p. 1,p. 11].

On the software side, given that Linux is used as the operating system, a C interface to control the motors over a CANopen bus exists. Though, it should not be hard to port the software to another operating system or even a micro controller [54, p. 20].

5.3.2 Regulator

The new positioning algorithm, based on two PID-regulators was deemed significantly simpler than the previously used LOS based algorithm. It was also considered to be sufficient to move the robot properly about on the playing area. Therefor it was decided to use the two PID-regulators, one rotation and for translation. To be as fast and agile as possible the robot should be able to drive in curves.

Considering how the existing implementation has to be adapted into the new software ecosystem discussed in chapter 8, and how two PID-regulators aren't that hard to implement. It was found most important to reuse the principle rather than the implementation.

Chapter 6

Deciding on a positioning system

Thorough the existence of Eurobot-NTNU a number of positioning systems have existed. Since the lineage of positioning systems is not linear, there exist a number of deprecated systems one might continue development on. This section will conduct the necessary survey to perform a decision of witch system to continue development on.

6.1 Requirements

A list of requirements needed to meet our regulators needs was constructed. Then a global coordinate system was defined for the output. The system had origin in the red teams corner, x-axis along the long edge and y-axis along the short edge.

- Output x, y, θ in global coordinates
- Smooth, predictable and repeatable output
- Update rate above 10Hz
- Robust and noise tolerant hardware

6.2 Alternatives

The articles and datasheets of previous positioning systems were studied. Any commercial available “Indoor Positioning Systems” was not considered.

1998 - Encoders and stereo computer vision

The robot from 1998 used stereo computer vision in addition to velocity and turning sensors to estimate the position [58].

2002 - Ultrasound

In 2002 a beacon system tailored to the Eurobot rules was constructed [2]. The system used ultrasound to measure the distance to three fixed beacons.

In 2003 gyros and encoders were added to the existing ultrasound positioning system [13]. To join all the acquired data a Kalman filter was used.

2005 - Laser angel meter

In 2005 a new beacon positioning system was constructed. The old system was characterized as: too complex, proven to interference from opponents positioning systems and not observable without a gyroscope [59, p. 33].

The new system had a laser and laser-detector on the robot. The beacons acted as a simple reflector, reflecting the laser beam. The system did not measure the distance to the beacons, only the angle.

To get a quick update rate they used encoders that were decoupled from the drive system. The decoupling was done to ensure that the no-slip assumption held true. The angular measurements and the encoder readings were integrated through a Kalman filter.

2006 - IR angel meter

The laser beacon system was switched with a infrared beacon system [16]. Now all the fixed beacons emit modulated infrared light. The robot beacon rotates, demodulates and registers the angle at which it was hit.

In 2007 the system was reused, but the Kalman filter was switched with a Particle filter [30]. This allows the use of non-linear sensor data, like a color sensor under the robot.

In 2008 a new infrared angle meter was designed. The new meter still used the Particle filter for data joining [37].

2009 - Laser tower and encoders

Improved encoders [29] and a new laser/ultrasound/infrared beacon system was used [28]. In 2009 only used the encoder system for position, the beacons system was only used to detect the enemy.

In 2010 the beacons were reused, but a new robot tower was developed. The tower have two parallel laser rotating with constant speed, doing up to nine rotations per minute measuring both distance and angle [31, p. 89].

6.3 Decision

To conclude there have been three kinds of position systems in use, relative, absolute and a combination of the two. None of the absolute position systems fulfills the fast-update-rate requirement. Moreover relative position systems are proven to drift and require a known start position. A combination of the two seems best but has not been in use since 2008. How to do this data joining is an implementation detail left for chapter 12.3.

For quick-updating sensor it was decided not to use separate encoders, but instead only relay on the motors built in hall-sensors. This decision goes against previous advises [59, p. 33], but was due to the need for new hardware as pointed out in [29, p. 31]. Any new encoders would need to fulfill the “ Robust hardware” requirement, something that has been proven harder than expected [28, 37, p. 130,p. 17]

Due to the robust hardware, fast update rate and already up and running hardware, it was decided to continue using the laser tower from 2010 for absolute positioning. The towers would need updates as explained in [31, p. 93], a task performed in chapter 12. This module would also allow us to track the opponent robot, something that’s necessary for the mandatory obstacle avoidance system.

6.4 Discussion

It should be noted that not all of the position systems studied were in working order. However their methods and principles remains and there is still lots to learn from their faults and errors. Commercial available “Indoor Positioning Systems”, and their working should also have been studied. And given time the hall-sensors should be switched with free-running encoders.

Chapter 7

Defining the electrical platform

There have been several solutions in the area of electronics in Eurobot. It is important to choose an electronics platform that is easy to use and allows further development of robot applications as the rules change every year. This includes where the main logic for the robot is implemented, how low level interaction with motors/buttons/actuators is done and how everything is powered.

7.1 Requirements

The electronics has to support two main aspects: High level implementation of algorithms/strategic choices; and low level management of actuators, digital/analog I/O, propulsion motors and power.

7.1.1 Embedded system

A list of requirements was set up for the embedded system based on the required components for the robot and some requirements from the rules.

- Support the laser positioning system selected in chapter 6
- Support the motor controllers selected in chapter 5
- Control of four servos to control arms and wing as described in chapter 4.3.1
- Support a start cord and an emergency stop button as defined in the rules.

It is also desirable to keep the system simplistic as defined in chapter 4.1. For the embedded system this includes easy maintainability, easy assembly, few wires and standardized connectors.

7.2 Previous systems

Every team from Eurobot NTNU has used a computer as the main processing unit, with an embedded system for low level functionality. Before 2009 a common power supply was used to power all the electronics in the robot, from 2009 to 2011 each embedded module took care of its own power conversion. The robot has been powered by various batteries including lead and lithium types, it was not possible to find detailed information about all of them.

2000-2003

Before 2004 a NOVA-600 x86-motherboard with an integrated processor was used as the main processing unit for the robot. A Barco GC800 micro controller development kit was used for hardware interaction [13]. Each extra Printed circuit board (PCB) communicated with the NOVA-600 motherboard through serial link.

2004-2006

In 2004 the same motherboard was used, but with a PC/104 stack with an I/O card and a motor controller card [14]. The I/O card did not have enough functionality and an extra custom micro controller PCBs was created. This system was used up to and including 2006 [35, 16]. The system was discontinued because the motherboard broke down during the 2006 competition.

2007-2008

In 2007 the NOVA-600 motherboard was exchanged with an x86 mini-ITX motherboard with an Intel Core2Duo processor to increase processing power [27]. This motherboard did not have any direct I/O support and all communication with each extra PCBs were implemented with a USB to CAN-bus adapter.

2009-2011

In 2009 an Acer Aspire one 11.6" laptop was used and most PCBs were made as modules and fitted into a modified 2U rack with passive backplane cards [28]. Each PCB was a standalone module that required a CAN-bus connection and a connection to the battery. The PCB 2U rack was used up to 2011, but the main processing unit was changed back to the mini-ITX motherboard in 2010 and 2011 [56, 38].

2011

An embedded system based on the EPIC [41] standard from the PC/104 consortium was proposed as a pre master project [22]. It contains power supplies for common voltages, a micro controller and a stack following the physical specification of the PC/104-plus standard. Expansion of functionality can be done by creating new PCBs for the stack.

7.3 Decision

7.3.1 The main processing unit

The first main electronics design question was “where should the main processing power be?”. Two solutions were considered; either a single embedded system that implements the high level and low level functionality or a commercial computer that implements the high level functionality with a smaller embedded system to implement the low level functionality. The choice fell on the latter as it supports several operating systems and programming languages and allows programming/debugging directly on the robot. In 2009 it was concluded that a small laptop computer saves physical size, removes the need for a dedicated power supply and reduces the amount of extra gadgets like an external screen, keyboard, mouse and hard drive [29].

To ensure compatibility with several operating systems and programming languages an x86-architecture was desired as opposed to a ARM architecture. To keep the physical size to a minimum tablet computers were looked at, the choice fell on a HP Slate 2 as it was the only tablet computer available with an x86-architecture at the time.

7.3.2 The embedded system

In 2004 they tried to make the electronics more compact by integrating a PC/104 stack in the computer. However all the electronics couldn’t be moved to the stack as some functionality wasn’t available as PC/104 modules.

The rack solution from 2009 is very modular and allows for easy expansion of functionality as extra modules are standalone entities. It also implements a standardized interface to create new modules, however there are some problems inherited by this solution. Since each module requires a CAN-bus and a power connection a lot of wires are connected close to each other, in case of wire failure it may be hard to pin-point where the fault lies. As each module implements its own power supply they take up a considerable amount of space in the rack and possibly decreased the power efficiency of the system if a bad power supply is implemented.

The EPIC based system proposed in 2011 provides modularity through expansion with PC/104-sized circuit boards. Each module in the stack has CAN-bus communication and a range of voltages available through a circuit board stack. Because of this the number of wires needed for a new module is limited to external components like motors/servos etc.

It was decided to continue development of the EPIC system proposed in 2011. This system is compact as it has one relatively small circuit board, it is customizable because of the stack and all wires are connected at a central point.

7.4 Discussion

There are commercial embedded computers available with PC/104 systems. We had already decided that we wanted to use a tablet computer because it allows for easy debugging and is an enclosed system with no or few wires. This fact combined with some points from the standard renders a commercial PC/104 system unfit. Standard voltages in the PC/104 standard are +3.3V, ±5.0V and ±12.0V, further the standard states that "A worst case module could use up to 39W of power" [42]. This is a small amount of power when dealing with motors/actuators and implies that an external power supply has to be built or bought. This also implies that all motors/actuators have to be connected to both a PC/104 module for control signals and a power supply for bulk power. Furthermore there is no guarantee that there are PC/104 modules available that does the kind of control needed in a robot for the Eurobot competition.

The main upside of using a commercial embedded computer is that a lot of time can be saved in the hardware development process. In our case we still need to design a power supply, which takes us back to a solution with several cards spread around the robot with a lot of interconnecting wires.

Chapter 8

Deciding on a programming language

A programming language to implement the strategies mentioned in chapter 4 must be chosen. Using a language that has been used in Eurobot before, would allow heavy reuse of pre-defined software modules. Using a new language, would allow more innovation.

Each programming languages has their own advantages and shortcomings, so the choice would be of great importance for the general software design.

8.1 Requirements

The language should run well on the HP Slate 2 tablet. It should be a high-level language, containing good constructs for concurrency and concurrent communication.

8.2 Alternatives

C++ and POSIX

C++ and POSIX has been used by Eurobot NTNU from 2004 - 2010 [1]. Some of the previous reports are unclear on whether software has been reused or re-implemented. At least it is known that in 2007 a very modular system with support for POSIX messages queues was developed. The implementation was evaluated to be good, but complex.

[1] POSIX was used in: 2004[14], 2005[35], 2006[16], 2007[27], 2009[28] and 2010[56]

Matlab/Simulink auto-generated C code

In 2005, auto-generated C code from Matlab/Simulink code was used [35]. The auto-generated code has been used in combination with manually written C++ and C. The report of 20xx claims that the generated code is hard to interface. When it was finally interfaced, it worked well.

Labview

Labview is a system-design software-environment that provides a drag-and-drop approach to programming [39, p. 5 - 31]. Dataflow is achieved by connecting function-blocks with wires. A system for Eurobot based on Labview was developed and used for Eurobot in 2011. The main report claims that this system worked well [38, p. 53]. However, other sources also claims that the solution lack modularity and limits the programmer significantly [54].

Go

Go is a quite new programming language from Google. Go has built-in concurrency support and good constructs for communication. The real-time capabilities of Go was evaluated in [36]. It was found to have good constructs for concurrent programming and error handling, but the scheduler and garbage collector implementation made it unsuitable for hard real-time. It was recommended used or tested in academic environments. It has not been used for Eurobot before.

8.3 Decision

This year's team was experienced with *writing* code and preferred writing rather than visual programming. It was pointed out that the previous Labview system was not very modular. It was also pointed out that generated C code from Matlab/Simulink was hard to interface. While the later statement might have changed with later versions of Matlab/Simulink, it was decided not to use Matlab/Simulink generated code unless it proved absolutely necessary.

For the high-level programming, it was decided to try the new Go programming language, and see how it compared to previous solutions. The POSIX Queue communication system implemented in previous years should be possible to replace with built-in Go structures. Also, as explained in more detail in chapter 10, it was desirable to use the new Linux CAN kernel module. The reuse of the CAN communication structures from previous years C++ solution, was therefore neither necessary nor desirable.

The drivers for servos, motors, and other devices connected to the robot's computer by a CAN-bus, could theoretically have been written directly in Go. However, this would lock any further development, both software and hardware to Go. Most languages make it easy to write wrappers for C code, whilst wrapping other languages like Go or C++ is not that easy [8].

To make the device drivers for the robot as portable as possible, it was decided that all drivers should compile to normal C libraries. This should provide drivers that can be easily wrapped in Go. Well written C drivers can also be ported to run on different architectures, e.g. microcontrollers.

8.4 Discussion

The most interesting point in the decision, is the choice to use Go for high level programming. This will require most of the software to be written from scratch. It will be interesting to see how this language compares to the C++ POSIX Message Queue system from previous years.

Chapter 9

Deciding on an operating system

Last year, Windows was used as an operating system (OS) for the Eurobot NTNUs robot [38]. Should it continue to be used as an OS this year, or should it be replaced?

To find the best suitable OS for this year's robot, previous years solutions should be evaluated. The OS must be able to run the programming languages that was chosen in chapter 8, and it must run well on the main processing unit selected in chapter 7.

9.1 Requirements

The following requirements were constructed:

1. Run well on the HP Slate 2
2. Make CAN-bus communication easy
3. Be a good environment for running Go and C code
4. Have a sufficient degree of stability and robustness
5. Allow a good environment for software development
6. Be familiar to the developers

9.2 Previous operating systems

VxWorks was used as an operating system for Eurobot Open in 2001 and 2002 [2]. In 2003, Windows, Linux and VxWorks were evaluated as possible operating systems for Eurobot 2004

[13]. For 2004 - 2010, different flavors of Linux have been used [1]. Windows was used as an operating system in 2000, 2003 and 2011 [2].

VxWorks

VxWorks is a proprietary, real-time operating system (RTOS). One of the key features is a multitasking kernel with preemptive and round-robin scheduling and fast interrupts response [62]. The RTOS is said to be tuned for both performance and determinism, and it has a small and configurable memory footprint. The community size seems to be small compared to Linux and Windows, but a simple Google search reveals that it does exist. Compilers and development tools can be bought from Wind River Inc.

Though CAN is obviously possible on VxWorks, there does not seem to be one single standardized CAN API available.

In 2003 it was concluded that the robot did not need the real-time functionality that VxWorks provided. Good device support and a good development environment was considered more important [13, p. 24].

Linux

Linux is an open source operating system with built-in support for a wide range on devices. It runs on a large collection of different processor architectures. There are many different *distributions* available that are based on the Linux kernel [11]. Most Linux distributions has good language support, low system requirements compared to Windows, and provides a powerful terminal. Distributions can often be downloaded directly from the internet free of charge.

Development tools, libraries and compilers can be acquired with ease, and for free. The community is quite large, which again makes it easier for the software developers to find much needed information. The degree of stability and robustness can vary from distribution to distribution, though the Linux kernel itself is considered to be quite stable [24].

Newer versions of the Linux kernel provide the *SocketCAN* kernel module, contributed by Volkswagen Research [54, 47]. SocketCAN provides an API that enables CAN communication over a standard BSD socket interface. The Linux version of the module is often just referred to as the Linux CAN kernel module.

When Linux was first reccomended for use in Eurobot, it was considered the best option due too its relatively low system requirements and good support for USB devices [13].

[1] Linux was used in 2004 [14], 2005 [35], 2006 [16], 2007 [27], 2008 [37], 2009 [28] and 2010 [56]

[2] Windows was used in 2000, 2003 [13, p. 23] and Eurobot 2011 [56, p. 10]

Windows

Windows is one of the most widely used operating system for desktop computers. The system requirements for memory, CPU and storage are considerable higher than they are for most Unix based operating systems [10]. Still, Windows provides a well-defined set of standard libraries, and it has a large development community.

Although Microsoft charges for most of their compilers and development tools, there is free compilers and development tools available. When it comes to CAN communication, Peak System provided a CAN API that could be used [44].

In 2011 it was concluded that the transition to Labview and Windows 7 went without any major problems [38, p. 53]. As LIDAR support and web-cam support existed in Windows, it was claimed that the transition saved the team from a lot of time in driver development. However, it is also stated that that Linux drivers probably did exist [38, p. 53]. The EiT report from the same year states that windows was chosen over Linux mainly due to the team's lack of experience with Linux [55, p. 16].

9.3 Decision

The first requirement for this year's choice of operating system was that it should run well on the HP Slate 2. This tablet PC consists of quite new and specialized hardware, like for instance the touch screen. While it was considered quite likely that both Windows and Linux would run well on this machine, the team did not considered it as likely that everything would work as well with VxWorks. As VxWorks was also mainly unfamiliar, not supported by Go, required quite some financial means, and as we did not need its features, it was excluded as an alternative this year.

For requirement 2, easy CAN-communication, Linux might prove to be a better alternative than Windows, as it provides the SocketCAN kernel module. The EPOS2 drivers from chapter 5 also relies on SocketCAN [54, p. 23].

Both Windows and Linux were considered to meet requirement 3, 4, 5 and 6. However, for a majority of this year's master students, Linux was considered a much more familiar development environment than Windows. For requirement 3, Go was slightly better supported on Linux than it was on Windows [18]. All in all Linux seems to be a stable solution, it has sensible system requirements, a small enough footprint, and good enough real-time capabilities for our use. It was therefore decided that the robot for 2012 should run Linux.

9.4 Discussion

When it comes to stability, Linux has provided a reliable environment for Eurobot in seven years in a row. More importantly, it seems to provide a good environment for software development. However, as the Slate 2 consists of quite new hardware, it’s possible that some components are not yet well supported by all the major Linux distributions.

Chapter 10

Defining the software modules

To be able to implement the software for this year's robot, a rough overview of what modules that are needed, must be composed. The design decisions done in chapter 4 - 9 should give sufficient background for deciding what is needed. In addition to the required modules, it was desirable to have a simulator and debugging-GUI available. The development of these extra modules would require some work, but having them available could prove to speed the development significantly.

In addition to a list off software modules, a rough plan for interaction and communication patterns are needed. To allow reuse in future years, a clear distinction between task-specific- and reusable code should be made.

10.1 Suggested concepts

Producer-consumer problem

The *producer-consumer problem* describes a condition in concurrent programming where you share data over a bounded buffer. There are two threads or processes in this problem. The first one is the *producer*, which task is to put data in the buffer. The second one is the *consumer*, which task is to remove data from the buffer. The solution to the problem involves defining what should happen if the buffer is empty or full, and to avoid race conditions.

The A* pathfinder algorithm

A* is a relative simple and efficient 2D pathfinder algorithm. It has been used in Eurobot before [29, 38, p. 40,]. More detail about A* is given in chapter 14.

10.2 Available software

C EPOS 2 motor drivers

In the autumn of 2011, a C driver for the Maxon EPOS2 motor controllers was written [54]. These drivers rely on Linux CAN and use a subset of the CAN-Open protocol for its communication. Implementation of this subset of CAN-Open and wrappers for the Linux CAN module is included in the driver.

C++ laser tower drivers

C++ drivers, relying on the C++ and POSIX system explained in chapter 8, existed. The CAN communication did not rely on the Linux CAN module.

C++ statistic filter libraries

An extended Kalman filter library called KFilter was found online [63]. As mentioned in chapter 6, both a Kalman filer and a particle filer has been developed for Eurobot in previous years. No pre-written Go library for statistic filters was found. It should be noted that even though it is possible to interface C++ code to Go it is not a straight forward and clean process [8].

Go matrix library

A linear algebra library written in Go was found online [25, 4]. It is called *Gomatrix*, and supported many matrix operations. This was the only linear algebra library found that was written in Go. In addition to this library, it also exist bindings for tools like lapack.

Go pathfinder library

A Go library implementing the A* algorithm was found online [46]. The name of the library was *Gopathfinding*. It was written as a self-educational project and had some performance and implementation issues. The structure of the project was however quite decent. This was the only Go pathfinder library that could be found.

10.3 Defining the modules

Since the electronics mentioned in chapter 7 relay on CAN for communicating with the computer, a layer for CAN-bus communication is needed. Separate device drivers for each device should be developed. To avoid duplicating code, the CAN-bus communication part of the drives should be in a separate module. It is known that the motor drivers use CAN Open, so there will be one module to implement a subset of CAN Open, and one module to wrap the Linux CAN API. The later package should be written so that it is possible to use a drop-in replacement when adapting the drivers for other platforms than Linux.

In the higher level code, there should be a hardware abstraction layer, interfacing all the device drivers to the rest of the code. Having this interface available, should make it easier to implement a simulator.

To control the hardware, a separate controller thread should exist. This controller thread should again receive its orders from an AI thread. These threads should be implemented in separate modules.

To store the current state of the robot, a module containing structures for this should be developed.

To avoid confusion about units, a unit library should be developed. The library should provide mechanisms to prevent physical types and units from being thoughtlessly mixed. E.g. it should be easy to see whether a variable holds millimeters or meters, or whether it stores a distance or a velocity.

According to chapter 3 and 4, a safety mechanism to avoid crashing into the opponent must be implemented. For maximum safety, this functionality should be implemented as its own thread, totally separate from the AI.

Chapter 4 also states that a pathfinder is desirable. The EiT group suggested that the A* algorithm should be used [26]. This is also an algorithm that has been used in Eurobot before. To allow reuse in future years, this module should be implemented as a library. Any task-specific part of the pathfinder should be separated from the library.

Chapter 6 describes the desire to use a combination of relative and absolute positioning. To be able to combine such data, a statistical filter is needed. In chapter 12 it is argued as to why the *extended Kalman filter* was chosen for this job. This filter should be implemented as a general purpose library. Any task-specific parts should be separated from the library.

Chapter 5 explains the regulators that should be used. Two standard PID regulator must be created. It can be implemented directly in the task-specific code, and should not need a separate library.

10.4 Communication patterns

Design proposals for communication patterns between some important software and hardware modules will be explained here. The implementation will be explained in chapter 14.

10.4.1 Hardware communication

Section 10.3 defines that the desire to have a CAN-bus layer, separate device drivers, a hardware abstraction layer and a controller thread. In addition, it is known that the motor drivers rely on Open CAN. Figure10.1 show an illustration of how these different layers should be stacked.

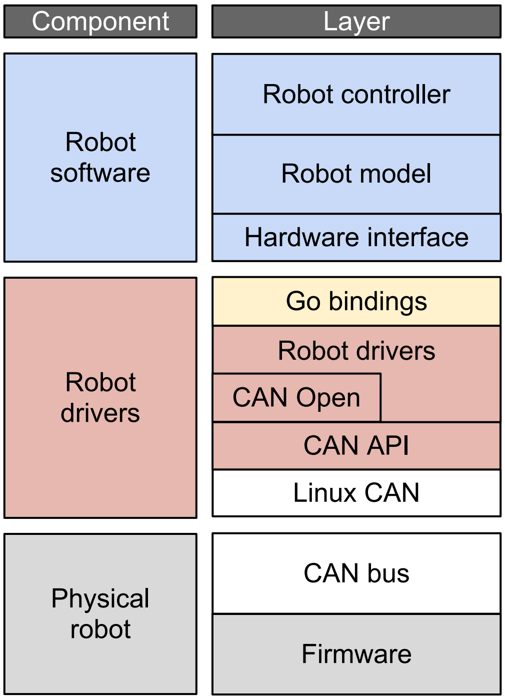


Figure 10.1: Hardware abstraction layer design

When sending data, the robot controller should ask the robot model to perform a hardware action. The command should propagate down to the hardware interface, which should again call the right driver functions. The driver function should generate the necessary CAN messages and send them of to be handled by the Linux kernel. When receiving data, the robot software's hardware interface should have routines for regularly polling the device drivers and store the latest readings for other modules to use.

10.4.2 AI and the controller

To enable easy replacement of the AI thread, the AI and Controller should be designed to have a producer-consumer layout, with the AI as a producer and the controller as a consumer. This makes the AI and Controller form a producer-consumer problem, which can be easily solved by the structures for concurrent communication in Go. Being able to replace the AI, should allow programmers to test different components of the robot, without using the "real" AI. For instance, a custom "AI" for the debug-GUI could be developed.

10.5 Module requirements

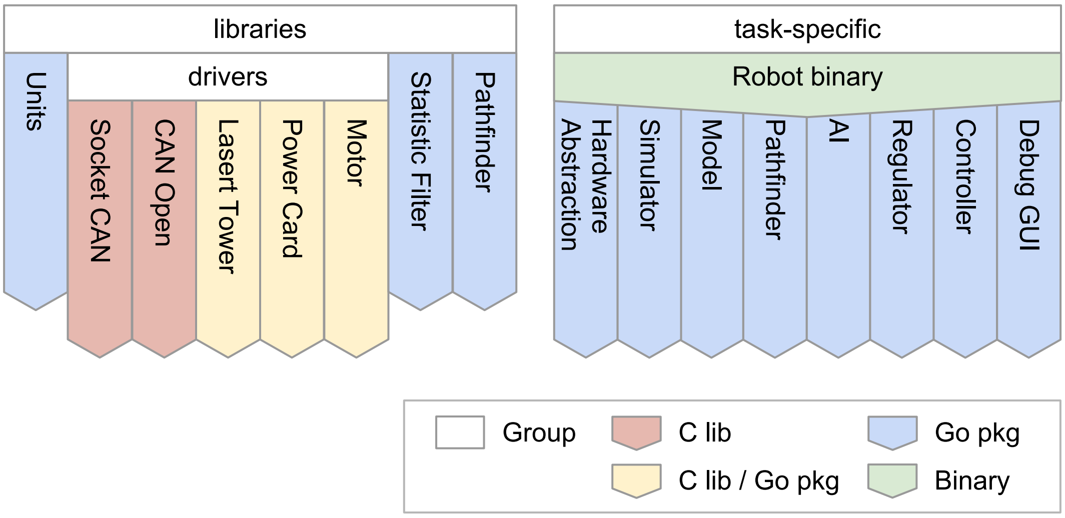


Figure 10.2: Planed software modules

Now that the modules have been defined, the requirement for each module can be defined.

A summary of all the defined modules is available is available in figure 10.2. The color of each module shows whether the module should compile to a Go package, C library or a binary.

10.5.1 Libraries

The CAN modules should compile to C libraries only. The device drivers should be written in C and provide Go wrappers, which would make them compile to both C libraries and Go packages. All other libraries should be implemented in pure Go.

Units: This library should implement variable types and constants for distance, velocity, angle and angular velocities. The goal of the module is to make it very clear what the unit is for any variable that holds one of the four mentioned physical types.

The code should compile to a Go package.

CAN API: This library should define read and write functions for the CAN-bus.

The only CAN backend that need to be supported is the Linux CAN kernel module. It should however be written in such a way that it is possible to support other CAN backends without changing the interface of the library. In this way, drivers depending on this library can be ported to run on other platforms, e.g. on microcontrollers, by changing this one library only.

The source code can be extracted from the C Maxon EPOS 2 drivers written in 2011. The code should compile to a C library.

CAN Open: This library should build on the CAN bus library, and implement a small subset of the CAN Open standard.

The source code can be extracted from the C Maxon EPOS 2 drivers written in 2011. The code should compile to a C library.

Motor driver: This driver should provide a sensible interface for controlling the motors, and read hall sensor data.

It should rely on the CAN Open library only. The code should be based upon the C Maxon EPOS2 drives written in 2011. The code should compile to a C library with a Go wrapper. The wrapper should depend on the units-library to represent physical units.

Laser tower driver: This driver should provide a way to turn the laser tower on and off, as well as a read function.

The code can partially be based on the existing C++ driver. It should only depend on the CAN bus library. The code should compile to a C library with a Go wrapper. The wrapper should depend on the units-library to represent physical units.

Power card driver: This driver should provide functionality to control all robot servos, as well as any other functions that the power card might provide.

The code must be written from scratch. It should only depend on the CAN bus library. The code should compile to a C library with a Go wrapper. The wrapper should depend on the units-library to represent physical units.

Pathfinder: This library should contain a function that will return the shortest non-blocked path from a start to an end coordinate. The function will need to be provided a 2D map of blocked and non-blocked areas.

The code should be based on the Gopathfinding library. It should compile to a Go package.

Extended Kalman filter: The extended Kalman filter library should include a function that takes the latest actuator thrust values u and measurement values y, run a single step of the statistical filter algorithm, and return a new estimate x. This library should then be wrapped by a module that defines the system matrices, and use them to initialize the filter.

As no Kalman filter libraries for Go were found, and since it is hard to wrap C++ code in Go [8], a new Go package that depend on the Gomatrix library, should be developed.

10.5.2 Task-specific

The main task-specific module, called *robot* in figure 10.2, should compile to a binary, but sub-modules that compile to Go packages should be allowed. The following sub-modules should be expected:

Hardware abstraction layer: The hardware abstraction layer should provide "poller-threads" to read measurement data from the drivers of the three devices: motor, laser tower and power card. For other modules, it should look like these "poller-threads" are run in a single thread. A safe shut-down of the main thread is needed so that physical motors and laser tower can be if the software terminates. Wrapper functions for controlling the hardware should be provided.

Simulator: The simulator package should provide exactly the same interface as the Hardware Abstraction Layer, but instead of commanding the hardware, an internal simulator should be stimulated. This way, it should be transparent to the rest of the code whether the simulator is used or not.

Model: The model should include a structure that can store the robot state in a sensible way, and tie the robot to either the hardware abstraction layer, or to the simulator. It should also include structures to store data about the opponents' positions.

Filter: The filter module should provide a thread that periodically collects the latest measurement readings from either the hardware abstraction layer or the simulator, runs the Extended Kalman Filter, and updates the robot's position and angle in within the Model module.

AI: The AI module should provide multiple AI versions, were only one version is to be used at any one time. For other modules, it should look like the AI run as a single thread. Each AI should use a bonded buffer (i.e. a Go channel) to send commands to the controller. The internals of the main AI should implement the strategy and tactics explained in chapter 4. The AI should also use a bounded buffer to read out "Done"-messages from the controller.

Pathfinder: This package should wrap the A* library, and provide a way to convert robot positions into integers that fit into an A* 2D map of blocked and non-blocked areas. The path returned from the library, should be converted back to robot-positions.

Controller: The controller should provide a thread that interacts with the hardware based on commands from the AI. The controller should also be responsible for managing the regulator. For other modules, it should look like the Controller run as a single thread. "Done"-messages should be sent when a command has been completed.

Regulator: The regulator should provide a function that take in the robots current position and a destination, and returns reference values for both the velocity and the angular velocity.

Safety: The safety module should provide a thread that periodically checks whether the opponent is inside our robots slow-down or the stop area, as described in chapter 4.

Chapter 11

Designing the complete system

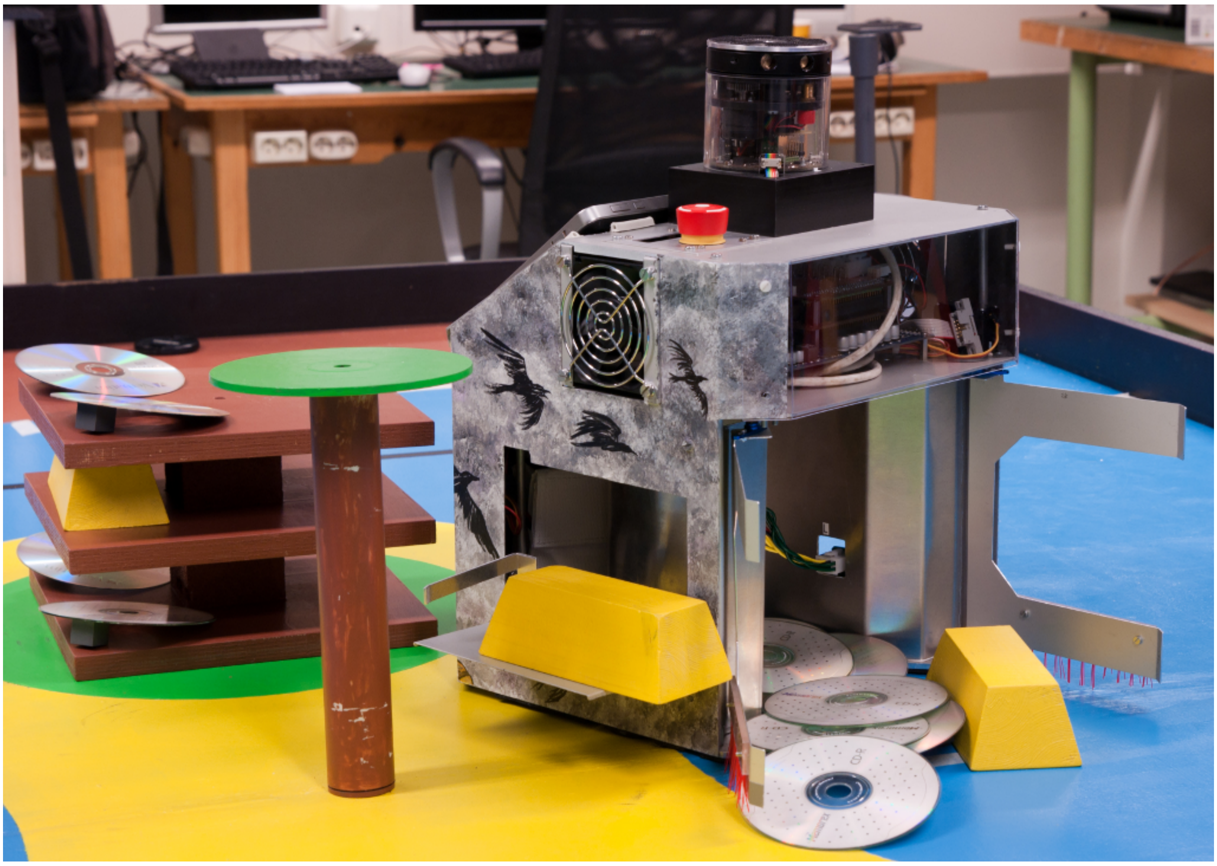


Figure 11.1: Finished Robot

This chapter discusses the final robot design. Just as its components, the complete robot was designed with the strategic goals from chapter 4.1 in mind. These ideals reduce to the mechanical design goals of: simple construction and easy maintainability.

11.1 Currently existing systems

While there are many aluminum sheets from previous years robots laying around, they are all very task specific. The only Eurobot-NTNU robot still functioning, the one from 2011 [38], was critical different in regards to: the PC and embedded-system used, and mostly the mechanical “servo-mechanisms”. Since the Eurobot objectives change from year to year, it is almost never possible to reuse the mechanical components. Due to this, the strategic goal of re-usability was removed.

11.2 Components

The mechanical design of the robot has to realize the strategy choices discussed in section 4, support systems from section 5, 6 and 7, and abide by the rules in section 3. The following components had to be placed in the robot:

- Propulsion Motors
- Laser Positioning System
- Embedded System
- HP Slate 2 Tablet Computer
- Battery pack
- Wings
- Doors
- Servos for Wings and Doors
- Emergency Stop Button

11.3 Component placement

The laser positioning system has to be placed on top of the robot, preferably as close as possible to the center of the robot to simplify the positioning. To further simplify navigation and positioning it was desired to keep the propulsion motors on the same horizontal axis as the laser tower. To make the robot stable the relatively heavy battery package should be placed as close to the ground as possible.

The design of the doors and wings is discussed in the EiT report [26, p. 12]. The doors should be placed in the front of the robot, wings on the sides. The movable axis of the doors and wings has to be mounted to a servo motor.

Due to rules the emergency stop button has to be placed on top of the robot, easily accessible from all angles of the robot. The tablet computer should be placed so the screen is visible during operation for debugging. It is desired to have it high up on the robot to minimize the danger of an opponent crashing into it.

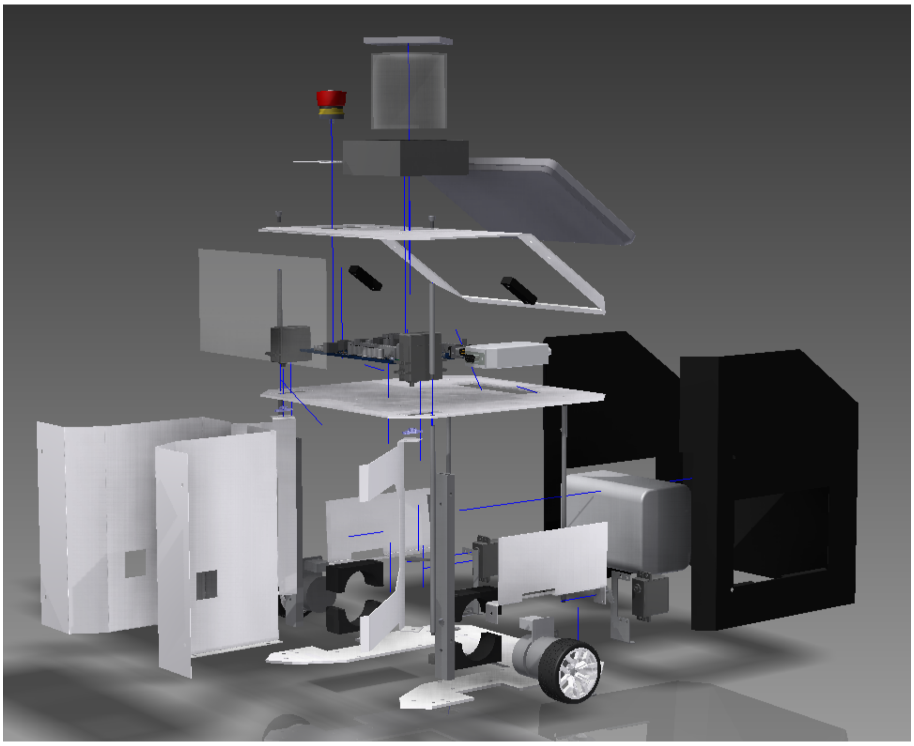


Figure 11.2: Exploded View

11.4 Physical design

The introduction to this chapter states the design goal of a robot is that is should be easy to assemble and disassemble. All the individual mechanical parts of the robot should be easily replaceable with few tools and in a small amount of time. Every part should be easy to reproduce in case something breaks.

These criteria and the component placement was created in collaboration with the EiT group, the mechanical design was done by the EiT group [26, p. 6]. Figure 11.2 shows an exploded view of the final robot design with all the robots components. Each drawn part of the robot was produced by the mechanical workshop at the Institute of Cybernetics Engineering at NTNU according to technical drawings as seen in appendix B.

11.5 Finished robot

The finished robot design was produced and assembled with only two minor modifications. Both of the changes to the design can be seen in figure 11.1 on page 51.

Two fans were added at the top of the robot. The fans increased the mechanical strength to the top of the robot and provided cooling for the tablet computer and electronics. The second modification is that the length of the wings on the side was slightly increased to make it easier to catch gold bars from the totems.

All the mechanical parts, electronics and motors etc. fit straight into the robot. The electronics lives on its own floor, separated from any dangerous movable parts. The HP Slate 2 is placed clearly visible on the top of the robot without interfering with the emergency stop and laser-tower. The laser-tower and drive wheels are all placed on the same axis, while the hollow front is as big as possible.

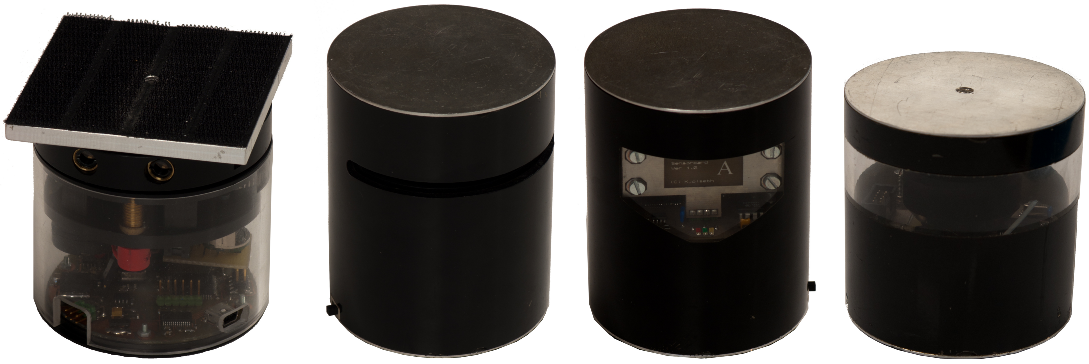
To reduce the need for special tools to assembled and disassemble the robot, it was standardized on ISO M3 screws. All bolts were placed at easily accessible locations. This might seem like a small feature, but it has saved a countless amount of time.

Part III

Implementation and Improvements

Chapter 12

Positioning system improvements



(a) Wall-E (b) Fixed beacon front (c) Fixed beacon back (d) Eve

Figure 12.1: The beacon towers

The absolute positions system developed in 2010 had a list of future work needed to get proper functionality [31, p. 93]. The list is recited here.

1. Calibrate distance and angle measurements
2. Develop an optimal triangulation algorithm
3. Compensate for the robots movements by using encoders

This chapter will cover the work done to fix these points. During this work it was discovered that the minimum range for enemy detection was not good enough, this problem was also corrected.

12.1 Background theory

12.1.1 Existing hardware

The positioning system consists of three different kinds of beacon towers, see figure 12.1. There's a beacon called "Wall-E", it is placed on-top of your own robot. A second beacon called "Eve" or just "E", is placed on the opponent robot. And finally three fixed beacons "A", "B" and "C", which are placed on the edge of the playing area as shown in figure 12.2.

"Eve", and the fixed towers, consists of a laser-detector, a timer and a radio module. They measure the time from the first laser hit to the second and radios the time back to "Wall-E". "Wall-E " spins at a constant rate. Upon receiving a radio message it calculates the distance and angle using equation 12.2.

When functioning, the complete system outputs the distance and angle to three fixed beacons, and a opponent robot, about six times a minute.

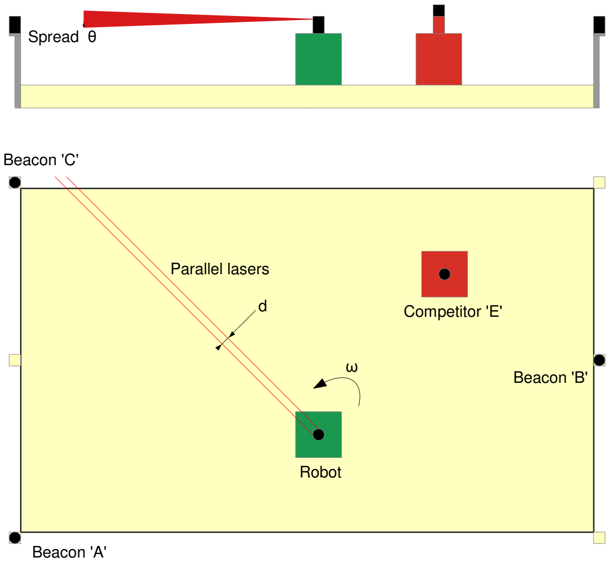


Figure 12.2: Beacon positioning

$$v = \omega R = \frac{d}{t} \tag{12.1}$$

$$R = \frac{d}{\omega t} \tag{12.2}$$

12.1.2 Kalman filter theory

A Kalman filter is a discreet, recursive data processing algorithm [60, 5, p. 92,p. 214]. The Kalman filter has a number of nice properties including being unbiased and minimum variance. The Kalman filter makes a number of assumptions about the system:

1. The process noise and measurement are white and Gaussian
2. The initial state is Gaussian
3. The system is linear
4. The system is observable

In return it has a number of nice properties:

1. The estimate is unbiased and minimum variance
2. The Kalman filter is the optimal state (linear or nonlinear) estimator
3. The Kalman filter is asymptotically stable

It should be noted that generally these mathematically properties does not apply to nonlinear versions of the Kalman filter, like the “Discreet linearized Kalman filter” or the “Extended Kalman filter”.

12.1.3 Particle filter theory

Particle filtering is a sub-optimal filter type that use Sequential Monte Carlo methods (SMC) [12, 5, p. 7,p. 210]. The filter uses a number of virtual points, or particles, to represent the probability density. In each sequence, each of these particles are examined and given a weighting and a survival estimate.

12.2 Calibration

As mentioned in the introduction, and noted in 2011, the hardware needed calibration [38, p. 25]. The laser-tower was left running at the default rotation speed, about six rotations per second, and placed on a table together with one fixed beacon. A number of measurements were carried out. The data can be found in Appendix A.1.

The beacon distance measurements were found to differ from the real world. As seen in figure 12.3 the error grows bigger at longer distances, it can be viewed as a regular non-linear measurement error. This is easy to correct for by running the measurement through the function $f(x)$ defined in figure 12.3.

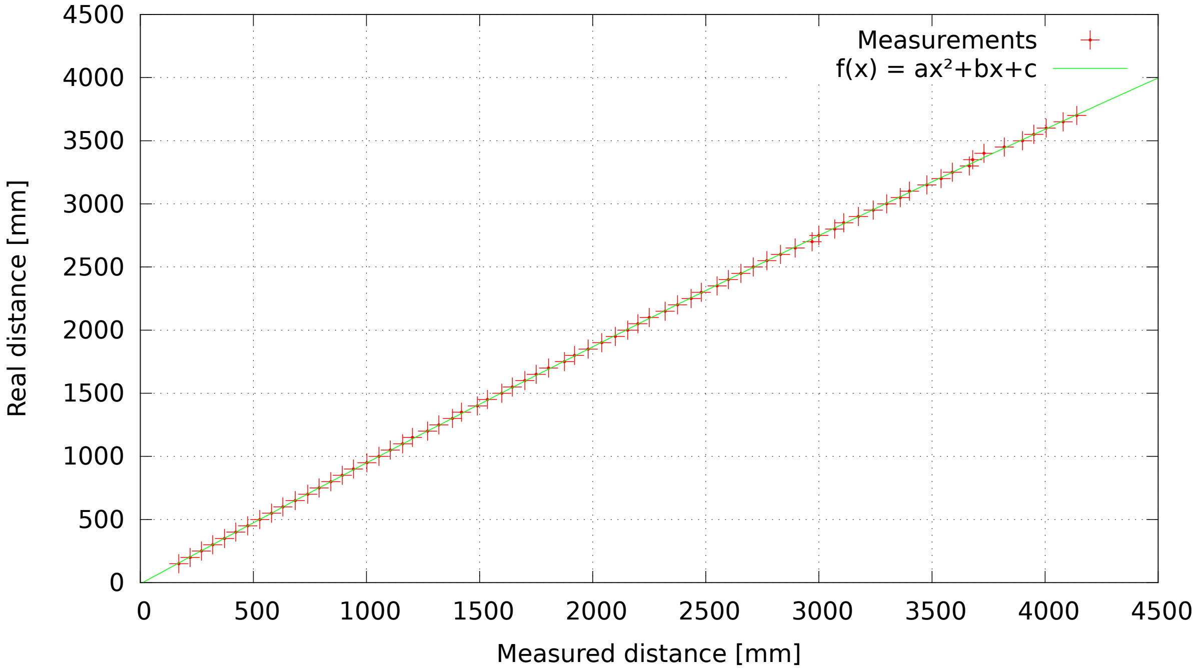


Figure 12.3: Constant measurement error

12.3 Choosing an algorithm

An algorithm to solve the triangulation problem was needed. The historic study performed in section 6.2 showed three main alternatives previously used.

- A direct arithmetic solution [31, p. 58]
- Using a Kalman filter [35, p. 23]
- Using a Particle filter [30, p. 36]

A direct arithmetic solution was quickly dismissed as complicated and error prone. The system is over-defined with six measurements/equations and three variables, position and orientation. The problem with a direct arithmetic solution would be to decide which measurements to trust.

The difference between using Kalman or a Particle filter is much smaller. The Particle filter supports very un-linear input, like a color sensor under the robot. It should also typically produce a more optimal result on nonlinear problems than an extended Kalman filter.

The problem of combining range and angular measurements is not linear, see equation 12.8 and 12.14. As such, the Kalman filter cannot be used as is, but a nonlinear version like the extended Kalman filter can. Parts of the problem at hand remind us of the problem of combining pseudo-range measurements in GPS. This problem turns out to linearize quite well, and is easily solved by an extended Kalman filter [60, p. 62].

In the end an extended Kalman filter was found to be sufficient, and likely simpler to implement than a Particle filter.

12.4 Developing a system model

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t)) + \Gamma \boldsymbol{w} \tag{12.3}$$

$$\boldsymbol{y}(t) = h(\boldsymbol{x}(t)) + \boldsymbol{v} \tag{12.4}$$

The extended Kalman filter needs a system model on the form given in equation 12.3 and 12.4. From here on we will call equation 12.3 the process-equation, and equation 12.4 the measurement-equation. The vectors $\boldsymbol{w}$ and $\boldsymbol{v}$ represent respectively the process and measurement noise, they must be white and Gaussian with zero mean.

12.4.1 Simplified process equations

If we at first imagine the robot to be standing still on the table with no actuator thrust, only a constant position and direction, the process-equation simply becomes:

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{12.5}$$

Since the Kalman filter is a discreet filter, this continuous solution must be made discreet, for instance by using forward Euler. When doing this we do not surprisingly end up with the identity matrix.

$$\boldsymbol{x}_{k+1} = (I - \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}})\boldsymbol{x}_k + \Gamma \boldsymbol{w} \tag{12.6}$$

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{A} \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{\Gamma} \boldsymbol{w} \tag{12.7}$$

This system is still linear and has no process noise. The noise requirement reduces to the beacon measurements having a normal distribution around the correct solution. This assumption is discussed in section 12.5.

12.4.2 Measurement equations

Distance

First we will study the range measurements available these measurements are described by equations 12.8, 12.9 and 12.10 [7]. In these equations x and y is the robots coordinate, A_x is the x-position of beacon tower “A”, and r_A is the measured distance. We note that there are two free variables and three equations, that is the system is over-defined.

$$r_A = \sqrt{(A_x - x)^2 + (A_y - y)^2} \tag{12.8}$$

$$r_B = \sqrt{(B_x - x)^2 + (B_y - y)^2} \tag{12.9}$$

$$r_C = \sqrt{(C_x - x)^2 + (C_y - y)^2} \tag{12.10}$$

The equations must be transformed into a usable form for the extended Kalman filter. That is they must be linearized around the last estimated state. We put the equations into a matrix and use the linearization method explained in [60, p. 62].

$$\boldsymbol{y}(t) = h(\boldsymbol{x}(t)) + \boldsymbol{v} \tag{12.11}$$

$$\begin{bmatrix} r_A \\ r_B \\ r_C \end{bmatrix} = \begin{bmatrix} \sqrt{(A_x - x_k)^2 + (A_y - y_k)^2} + v_A \\ \sqrt{(B_x - x_k)^2 + (B_y - y_k)^2} + v_B \\ \sqrt{(C_x - x_k)^2 + (C_y - y_k)^2} + v_C \end{bmatrix} \tag{12.12}$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} = \underbrace{\begin{bmatrix} \frac{x_k - A_x}{\sqrt{(}(A_x-x_k)^2+(A_y-y_k)^2)} & \frac{y - A_y}{\sqrt{(}(A_x-x_k)^2+(A_y-y_k)^2)} \\ \frac{x_k - B_x}{\sqrt{(}(B_x-x_k)^2+(B_y-y_k)^2)} & \frac{y - B_y}{\sqrt{(}(B_x-x_k)^2+(B_y-y_k)^2)} \\ \frac{x_k - C_x}{\sqrt{(}(C_x-x_k)^2+(C_y-y_k)^2)} & \frac{y - C_y}{\sqrt{(}(C_x-x_k)^2+(C_y-y_k)^2)} \end{bmatrix}}_{H} \tag{12.13}$$

Using the process-equation 12.7, measurement-equation 12.12 and the matrix H from 12.13, we now have a usable model. It assumes a motionless robot, does not use all available data and fails to produce an attitude[1] estimate. But never the less this system was implemented and tested and it does converge to the correct position.

[1] The orientation of an aircraft or other vehicle relative to the horizon, direction of motion

Angle

Given that we know or have estimated the robot's position (x, y) and direction θ, and that we know the location of the fixed beacons (A_x,A_y), it is possible to estimate the beacon angles.

$$atan2(A_y - y_k, A_x - x_k) - \theta_k \tag{12.14}$$

$$atan2(B_y - y_k, B_x - x_k) - \theta_k \tag{12.15}$$

$$atan2(C_y - y_k, C_x - x_k) - \theta_k \tag{12.16}$$

Adding these equations to the measurement-equation $y(t)$, and differentiate them with regards to $\boldsymbol{x}$ to get the matrix $H(\boldsymbol{x})$, the system becomes:

$$\boldsymbol{y}(t) = h(\boldsymbol{x}(t)) + \boldsymbol{v} \tag{12.17}$$

$$\begin{bmatrix} r_A \\ r_B \\ r_C \\ \omega_A \\ \omega_B \\ \omega_C \end{bmatrix} = \begin{bmatrix} \sqrt{(A_x - x_k)^2 + (A_y - y_k)^2} + v_{ra} \\ \sqrt{(B_x - x_k)^2 + (B_y - y_k)^2} + v_{rb} \\ \sqrt{(C_x - x_k)^2 + (C_y - y_k)^2} + v_{rc} \\ atan2(A_y - y_k, A_x - x_k) - \theta_k + v_{ta} \\ atan2(B_y - y_k, B_x - x_k) - \theta_k + v_{tb} \\ atan2(C_y - y_k, C_x - x_k) - \theta_k + v_{tc} \end{bmatrix} \tag{12.18}$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{x_k - A_x}{\sqrt{(}(A_x - x_k)^2 + (A_y - y_k)^2)} & \frac{y - A_y}{\sqrt{(}(A_x - x_k)^2 + (A_y - y_k)^2)} & 0 \\ \frac{x_k - B_x}{\sqrt{(}(B_x - x_k)^2 + (B_y - y_k)^2)} & \frac{y - B_y}{\sqrt{(}(B_x - x_k)^2 + (B_y - y_k)^2)} & 0 \\ \frac{x_k - C_x}{\sqrt{(}(C_x - x_k)^2 + (C_y - y_k)^2)} & \frac{y - C_y}{\sqrt{(}(C_x - x_k)^2 + (C_y - y_k)^2)} & 0 \\ \frac{y_k - A_y}{(A_x - x_k)^2 + (A_y - y_k)^2} & \frac{A_x - x_k}{(A_x - x_k)^2 + (A_y - y_k)^2} & -1 \\ \frac{y_k - B_y}{(B_x - x_k)^2 + (B_y - y_k)^2} & \frac{B_x - x_k}{(B_x - x_k)^2 + (B_y - y_k)^2} & -1 \\ \frac{y_k - C_y}{(C_x - x_k)^2 + (C_y - y_k)^2} & \frac{C_x - x_k}{(C_x - x_k)^2 + (C_y - y_k)^2} & -1 \end{bmatrix} \tag{12.19}$$

In this model the robot's direction θ is unbounded, that is all values from $-\infty$ to ∞ are acceptable. On the other hand, all of the measurements all ways fall within the range $\langle -\pi, \pi]$. This means that the angle θ must be "normalized" to the range $\langle -\pi, \pi]$ before comparing it to a measurement. Not only that, but when calculating the measurement-error internal in the Kalman filter, all angular errors must be "normalized" to $\langle -\pi, \pi]$. That is if we estimate ω_B to be 3.13rad, and the measured ω_B is -3.12rad, the error must be under 0.04rad.

This model was implemented and tested, and after the angles and angle-errors where bounded it works and converges to the correct solution.

12.4.3 Process equations

Now that we have reasonable model for a robot that is standing still, we should give it a sensation of movement. A way achieve this is to add the actuator thrust as an input to the process equation

12.20. The actuator thrust $\boldsymbol{u}$ in our system would be the target velocities of the two motors, V_L and V_R. Since the target velocities might be quite different to the real velocities, and we already have a good estimate of the real velocities from the motors hall-sensors, it would probably be wise to use it.

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t)) + \Gamma \boldsymbol{w} \tag{12.20}$$

$$\boldsymbol{u} = \begin{bmatrix} V_L \\ V_R \end{bmatrix} \tag{12.21}$$

Equation 12.22 and 12.23 from [21, p. 10] describes how a two wheeled robot moves. If the equations are transformed from the body coordinate system to the global coordinate system and made continuous, one gets equation 12.24.

$$\Delta x_{body} = \frac{V_L + V_R}{2R} \tag{12.22}$$

$$\Delta \theta_{body} = \frac{V_L + V_R}{2R} \tag{12.23}$$

$$\begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} \frac{V_L(t)+V_R(t)}{2} cos(\theta(t)) \\ \frac{V_L(t)+V_R(t)}{2} sin(\theta(t)) \\ \frac{V_R(t)-V_L(t)}{2R} \end{bmatrix} \tag{12.24}$$

Deriving and discretizing this equation we get the new process equations for the extended Kalman filter. The new equation incorporates the velocity estimates from the hall-sensors, with the last estimated position, to give the new estimated position.

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} 1 & 0 & -(\frac{V_{Lk}+V_{Rk}}{2} sin(\theta_k)) \\ 0 & 1 & (\frac{V_{Lk}+V_{Rk}}{2} cos(\theta_k)) \\ 0 & 0 & 1 \end{bmatrix} \tag{12.25}$$

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k + h(\frac{V_{Lk}+V_{Rk}}{2} cos(\theta_k)) \\ y_k + h(\frac{V_{Lk}+V_{Rk}}{2} sin(\theta_k)) \\ \theta_k + h(\frac{V_{Rk}-V_{Lk}}{2R}) \end{bmatrix} \tag{12.26}$$

12.4.4 Complete System

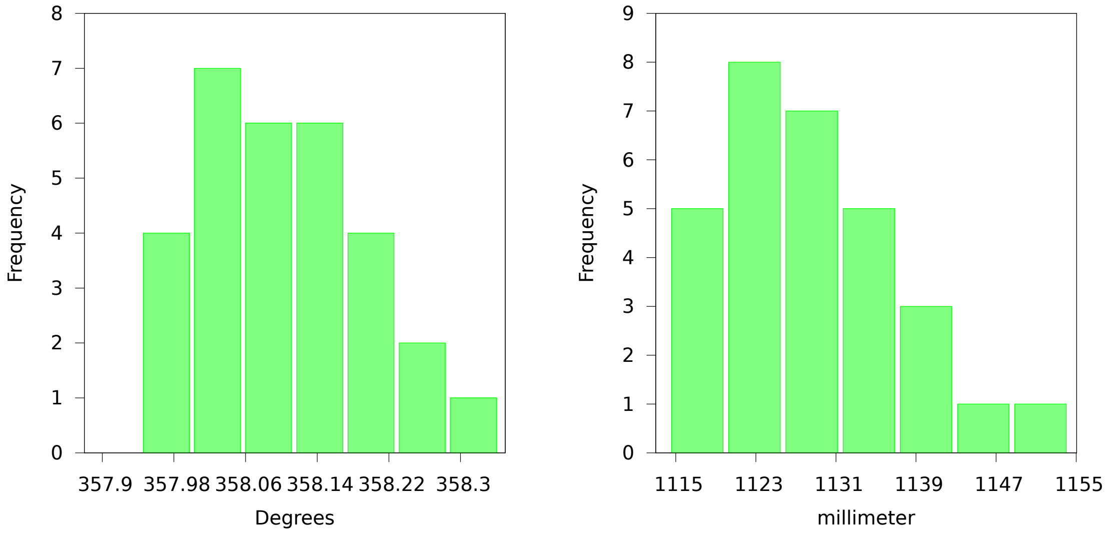
The goal of this section was to find a system model on the form given in equation 12.3 and 12.4. And to transform those equations in to the equations necessary to implement the extended Kalman filter.

If one linearizes away the fact that a rotation of the robot will interfere with the measurements, one can join the process and measurement equations. The system are then given by equation 12.24 and 12.18. Both of the equations have been transformed into the necessary forms to implement the extended Kalman filter.

12.5 Tuning

To get an idea of the variation in the measurements, a number of samples were taken at three different distances. The data can be found in appendix A.2. The measurements were taken at approximately 1, 2 and 3 meters distance. A more extensive testing can be found in [31, p. 77].

Figure 12.4 shows a histogram of the measurements taken at 1 meters. While the data only have 30 samples, it clearly shows a deviation from the normal distribution that is required by the Kalman filter. Accepting these measurements as white and Gaussian would leave us with an non-optimal solution. However the measurements aren't that far off and we have already given up on a mathematical optimal solution by using the extended Kalman filter. Handling the measurements like as if they were Gaussian should still produce a result close to the optimal solution.



(a) Angle distribution

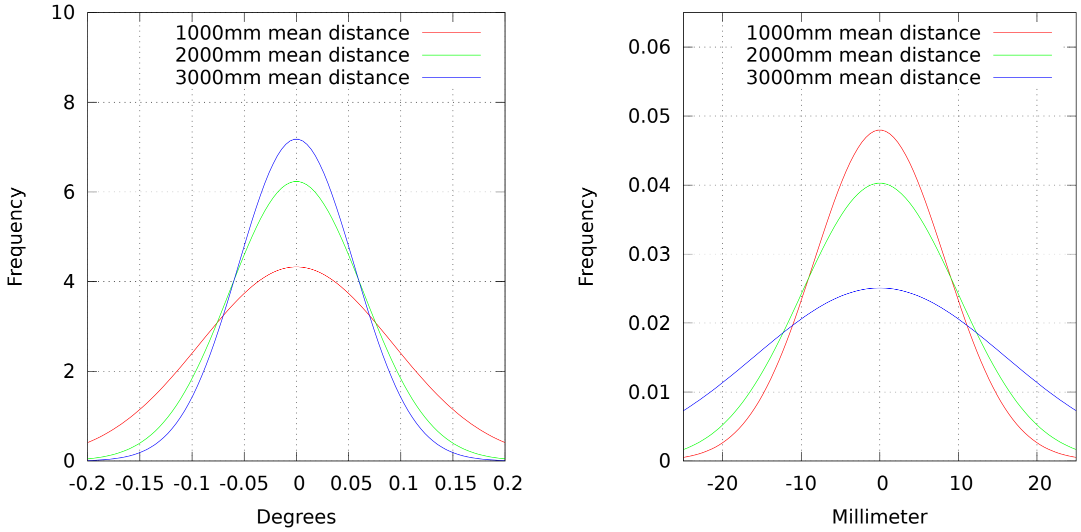
(b) Distance distribution

Figure 12.4: Measurement distribution

The variances of the measurements were calculated and the corresponding normal functions, with zero mean, were plotted in figure 12.5. As seen in figure 12.5a, the angular measurements are pretty accurate, with the measurement only getting slightly noisier at shorter distances. On the other hand, figure 12.5b shows that the distance measurements are fairly noisy and gets worse at longer distances.

12.5.1 Design matrices

The matrices Q and R are known as weighting or design matrices [60, p. 95]. R is the measurement weighting matrix, it holds the size of the measurements variance and covariance, while Q is the process weighting matrix.



(a) Angle deviation

(b) Distance deviation

Figure 12.5: Measurement deviation

To simplify the system, we make the assumption of zero covariance in the process and measurements, leaving Q and R diagonal. This is clearly a wrong assumption. While the process is arguably orthogonal, the measurements are clearly all dependent on correct tower rotation speed. Though as testing will show, it will suffice.

This linearization leaves nine parameters to tune, three process variances x, y and θ, and six measurement variances r_X and ω_X.

In the implementation, distance is always measured in millimeters, and rotation in radians. This unit difference in itself probably leads to a lot of the different variances seen in matrix 12.27. The equation shows the final matrix parameters found through trial and error. It show x and y weighted equally and a higher “trust” on the angular estimate. Another way to argue for the difference seen is to realize that to manipulate the robots x state one might need to change the robot θ state first. While the reverse is not true, changing θ does never require a movement of x first, in other words θ is of a higher differential order than x and y.

$$Q = \begin{bmatrix} 10^{-6} & 0 & 0 \\ 0 & 10^{-6} & 0 \\ 0 & 0 & 10^{-3} \end{bmatrix} \tag{12.27}$$

Looking at figure 12.2, where our starting area is to the left and the opponent to the right, it is easy to conclude that beacon “B” is more likely to temporarily disappear behind the enemy robot. Thus beacon “B ” should be weighted lower, or have a higher variance than “A” and “C”. Remember that the angular measurements have a smaller variance than the distance measurements, thus angular measurements should have a lower variance than distance measurements. Keeping all this in mind, and after some trial and error, the matrix R shown in equation 12.28 was found.

$$R = \begin{bmatrix} 10^{-2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 10^{-1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 10^{-2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 * 10^{-5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 * 10^{-4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 * 10^{-5} \end{bmatrix} \tag{12.28}$$

12.6 Opponent detection

Range

Even though the old beacon system is designed to track the opponent robot in a range from 30cm to 300cm [28, p. 43], testing showed the true minimum distance to be about 55cm. Since the Eurobot rules require an obstacle avoidance system [52, p. 21], it was of great importance to keep track of the opponent robots at shorter distances. Taken into account the biggest legal opponent robots, a collision would be imminent if the opponent beacon was within 30cm. Leaving the minimum opponent detection range at 55cm would force the robot to stop 25cm before a collision would even be possible.

Figure 12.2 shows the vertical position of the opponent beacon on a level above the other beacons. To make the rotating lasers hit the fixed beacons the opponent beacon, a lens spreads the light 30° in the vertical direction. This spreading is not uniform [28, p. 24], and it seems like not enough light is hits the opponent beacon at short distances.

One way to shorten the dead-zone is to minimize the height differences between the omnidirectional-mirror and the lasers source, thus minimizing the need for the light to spread. A quick fix to achieve this would be to turn the opponent-beacon upside down as shown in figure 12.6. Now, since the distance from the mirror to the light-sensor is calibrated according to the angle of the incoming light and the angle of the mirror, just flipping it over won't work. The calculations done in figure 12.7 shows that ideally the mirror angle should be steeper, but that positioning the light-sensor close enough it might just work.

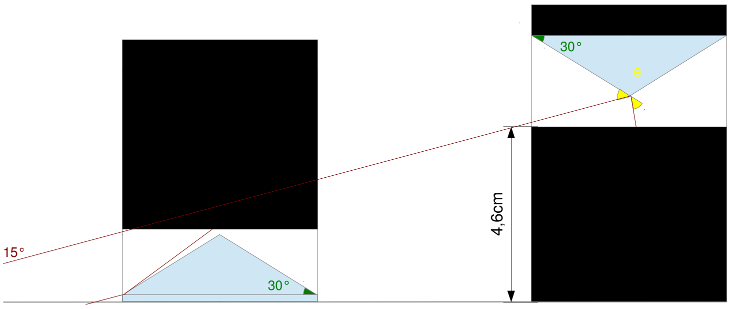


Figure 12.6: Minimizing the height difference

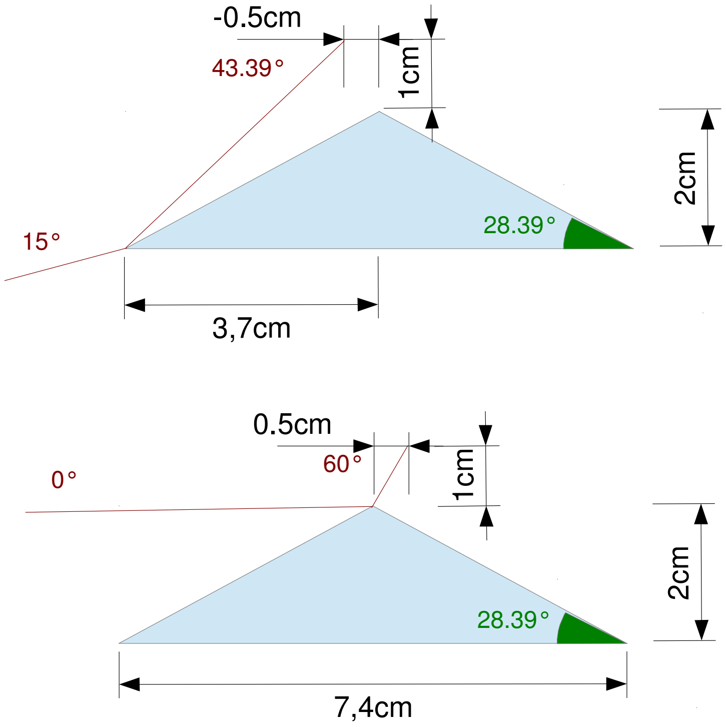


Figure 12.7: Mirror to light-detector distance calculations

The mirrors were adjusted. The tower was flipped over and then tested. Due to time constrains, this was done pretty quickly, something that can be seen in figure 12.8. The new configuration had a range from at-least 20 to 400 centimeters, not stopping before the testing platform hit the robot. On the other hand, the upside-down tower was far more sensitive to tilting especially at longer distances. This is probably due to the low sloping angle of the mirror. Another problem is that the 9V battery, powering the beacon, now lies loose on top of the electronics.

New beacon

Since the new Eurobot rules now allow each team to have two robots, a small and a big one [52, p. 17]. A new opponent-beacon was constructed with the existing design files [28]. A few minor changes were made to the design to integrate all the functionality on a single circuit board, refer to appendix C.1 for the schematics. The new beacon should be exactly equal to the old opponent-beacon except for identifying itself as "F". However due to mistakenly assuming an old omnidirectional-mirror laying around to be equal to the one used in beacon "E", the new beacon losses contact at long range. The cross-section of the new beacon's mirror is not triangular like the old one, but rather slightly curved.

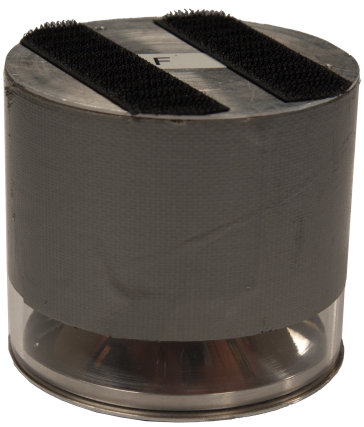


Figure 12.8: The new “upside down” opponent beacon tower

12.7 Result and discussion

The task of calibrating the distance and angle measurements has been performed. A fairly optimal triangulation algorithm has been created and combined with Hall-sensor data. In addition the opponent beacon detection range has been improved, and an extra opponent beacon has been made.

The filter has been implemented and found to work on the entire playing area. Its estimate is slightly skewed, at some places by up to three centimeters, but the output is smooth. In addition, the implementation behaves very deterministically, producing the same offset from run to run, making it easily correctable.

The robots orientation turned out to be the hardest parameter to estimate. Transforming the angular measurement-error to the range $\langle -\pi, \pi]$ turned out to be crucial. It is still possible to throw the orientation estimate off by rotating the robot back and forth with the right frequency.

Sub-centimeter position accuracy is hard to achieve with the system. This is partly due to the problem of placing the fixed beacons in the right position, and to much noise in the beacon measurements. A proper study of the systems accuracy should be performed.

Chapter 13

Producing the power card

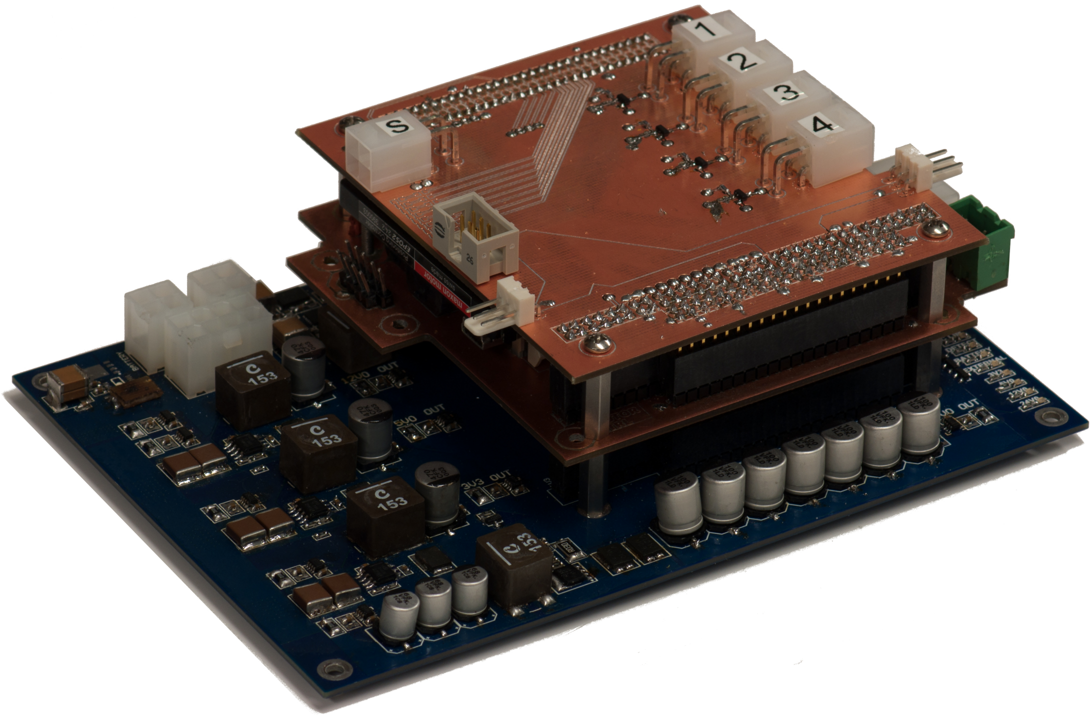


Figure 13.1: The hardware stack

This chapter describes the embedded system used in the robot. The system is a continuation of the work described in Robot Power Management [22].

13.1 Background theory

13.1.1 Linear power supplies

A linear regulator is a voltage-controlled current source (illustrated by figure 13.2), it forces the output voltage to a desired level by generating an appropriate current through a load [53].

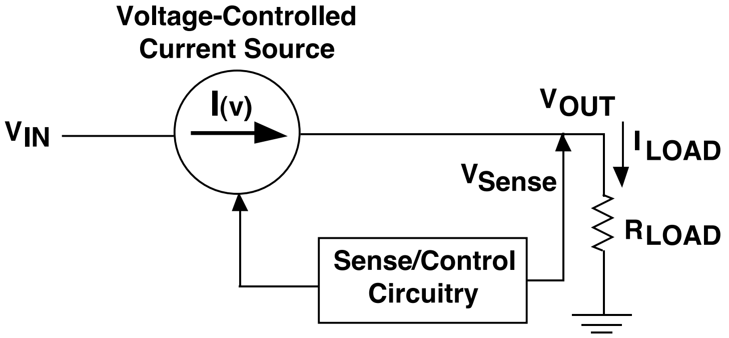


Figure 13.2: Linear Regulator Functional Diagram

The control circuit monitors the output voltage and adjusts the current source to hold the output voltage at a desired value. It is then evident that the limit of the current source defines the maximum load current the regulator can provide while still regulating the voltage.

A linear regulators efficiency is almost entirely dependent on the difference between the input and output voltage, there is also a small power loss due to ground current. Given the property $I_{in} = I_{Out}$ it can be shown that $\frac{P_{in}}{V_{in}} = \frac{P_{out}}{V_{out}}$ using ohms law, the efficiency η and power loss is given by:

$$\eta = \frac{P_{out}}{P_{in}} = \frac{V_{out}}{V_{in}} \tag{13.1}$$

$$P_{loss} = P_{in} - P_{out} = I \cdot \Delta U = I(V_{in} - V_{out}) \tag{13.2}$$

The bigger difference between the input and the output voltage, the less power efficient the regulator becomes.

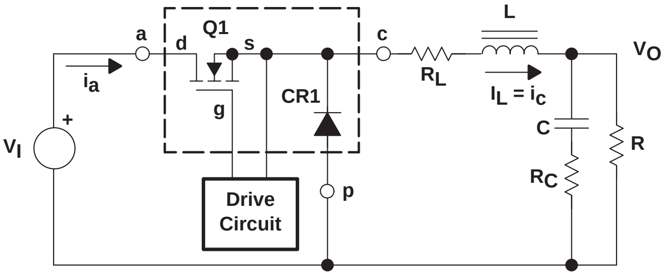
13.1.2 Switch mode power supplies

Switch mode power supplies exploits the characteristics of inductors together with clever switching of a MOSFET transistor to convert an input voltage to a desired output voltage. There are several topologies of switch mode power supplies. The most common types are buck [50], boost [48] and buck-boost [49], which is a combination of the former types. A buck power stage is used when the desired output voltage is lower than the input voltage. A boost power stage is used when the input voltage is lower than the output voltage. If the input voltage is variable and can be either higher or lower than the output voltage, a buck-boost power stage is used. Simplified diagrams of the power stages and their output voltages for buck, boost and buck-boost are shown below; Figure 13.3 and equation 13.5 shows a buck stage, figure 13.4 and equation 13.6 shows a boost stage and figure 13.5 and equation 13.7 shows a buck-boost stage.

All of these power stages use a drive circuit, this circuit implements a Pulse Width Modulated (PWM) signal. A PWM-signal is a train of square pulses with a predetermined period T_S. Every period can be referred to as a switching cycle. The difference between different cycles is the amount of time the signal is "high/on", T_{ON}, consequently the time the signal is "low/off" is called T_{OFF}. The duty cycle D of a PWM-signal is the relation between the on and off time and is defined as:

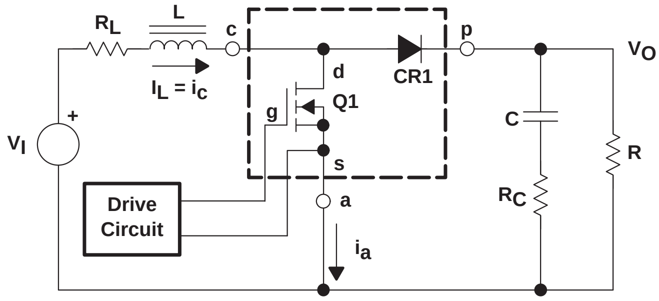
$$D \cdot T_S = T_{ON} \tag{13.3}$$

$$(1 - D) \cdot T_S = T_{OFF} \tag{13.4}$$



$$Vo = V_i \cdot D \tag{13.5}$$

Figure 13.3: Simplified Buck Stage



$$Vo = \frac{V_i}{1 - D} \tag{13.6}$$

Figure 13.4: Simplified Boost Stage

The trick to derivate the equation for the output voltages is to look at the current through the inductor in each of the power stages. The equations for the output voltage holds for continuous current mode (CCM), this mode implies that the current through the inductor never reaches zero. However, if the current through the inductor does reach zero the output voltages will be affected by the load, this is referred to as discontinuous current mode (DCM). To make sure the power stage stays in CCM the inductor has to be adequately large in terms of Henry, this is referred to as critical inductance.. The minimum inductor size is dependent on the input voltage V_i, output voltage V_O, output current I_O and the period of the PWM-signal, T_S. This implies

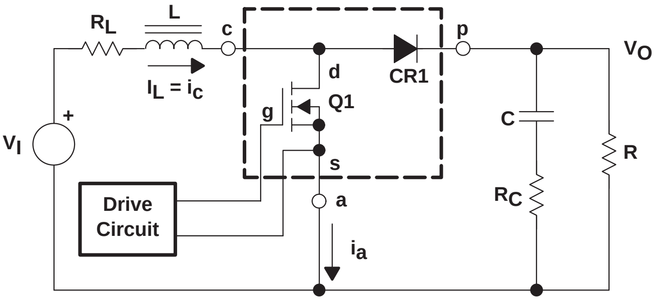


Figure 13.5: Simplified Buck-Boost Stage

$$Vo = -V_i \cdot \frac{D}{1-D} \tag{13.7}$$

that a PWM signal with a high frequency means that the inductor can be smaller in terms of Henry.

13.2 Power source

In light of the rule constraints in section 3.6.2 rechargeable batteries were looked into. Nickel-cadmium and nickel-hydrogen batteries has an energy density of $40-60Wh/kg$ and a typical cell voltage of $1.2V$, lithium batteries on the other hand has an energy density of $100 - 250Wh/kg$ and a typical cell voltage of $3.6 - 3.7V$ [6, p. 18].

Lithium batteries were chosen because of the high cell voltage and energy density. Due to the safety requirements battery packs with a protection module was desired. Traditional lithium ion cells can catch fire, or even explode if short circuited or improperly charged. Lithium cells from A123 systems use a different process that does not have these risks.

The choice fell on custom built battery packages with ANR26650M1-B cells from A123 systems with a protection module from Gylling Teknikk [1][20]. Each cell has a nominal voltage of $3.3V$ and a nominal capacity of $2300mAH$. It is recommended to keep the absolute minimum cell voltage at $1.6V$, and the maximum voltage at $3.8V$. A 12-cell battery pack with two series of 6-cells in parallel was ordered, this pack provides a nominal voltage of $19.8V$ and a capacity of $4600mAH$. The protection module provides cell balancing under voltage, over voltage, short circuit and over current protection and limits the current consumption to $17A$. It limits the minimum cell voltage at $2.0V$ and the maximum cell voltage at $3.8V$, thus making the battery packs operation range from $12.0 - 22.8V$. The cell data sheet shows that each cell will have a voltage higher than $2.5V$ until the cells capacity is almost completely exhausted, this makes the active range of the battery from $15.0V - 22.8V$. It also has to be noted that the voltage is very stable between $90\% - 10\%$ capacity, the can be seen from the cells data sheet [1].

13.3 Power card

In section 7.1.1 it was decided that the embedded system should support the laser positioning system described in section 12, EPOS 24/2 motor controllers described in section 5, servos for arms and wings, a start cord and an emergency stop button required by the rules in section 3.6.

In addition the system was designed with the following primary requirements:

- Reusable.
- Compact design.
- Micro controller as processor
- Standard interface to expand functionality / Modularity.
- Integrated CAN Communication for compatibility with previous Eurobot modules.
- Provide a range of voltages with high current.

And a set of optional requirements for ease of use of the system:

- Battery charger from external DC supply.
- Automatic switching between battery and external DC supply.
- On-board power consumption measurements.
- Ability to turn on/off voltage regulators from the micro controller.
- Few wires.
- Plugs on all wires to/from circuit board (no screw or solder connections).

13.3.1 Major design changes

It was proposed to use two micro controllers, one 32-bit board controller (AT32UC3C0512) to take care of power management and one 8-bit controller to take care of I/O operations [22]. It was also proposed to use a CPLD or an FPGA to be able to dynamically remap all i/o pins from both micro controllers to extension modules.

The same article discusses that a solution with two micro controllers and a CPLD or an FPGA may be overly complex and may discourage future users to use the circuit board. Thus it was decided to use a single 8-bit micro controller. An Atmel AT90CAN32 was chosen because it has hardware support for CAN-bus. This micro controller has been used in earlier Eurobot modules, and thus firmware drivers for the CAN module can be reused [31, appendix B].

When it was decided to not use an AT32UC3C some changes had to be done to the design. AT90CAN32 does not have any analog outputs (DACs) and does not have enough ADC inputs to support the proposed solution for current and voltage monitoring which required 13 ADC inputs. To enable analog outputs an external DAC with SPI interface was added. For current and voltage monitoring an INA219 from TI was chosen, the chip does voltage and current measurements which is readable through an I^2C interface.

Figure 13.6 shows a block diagram of the modified and final system design.

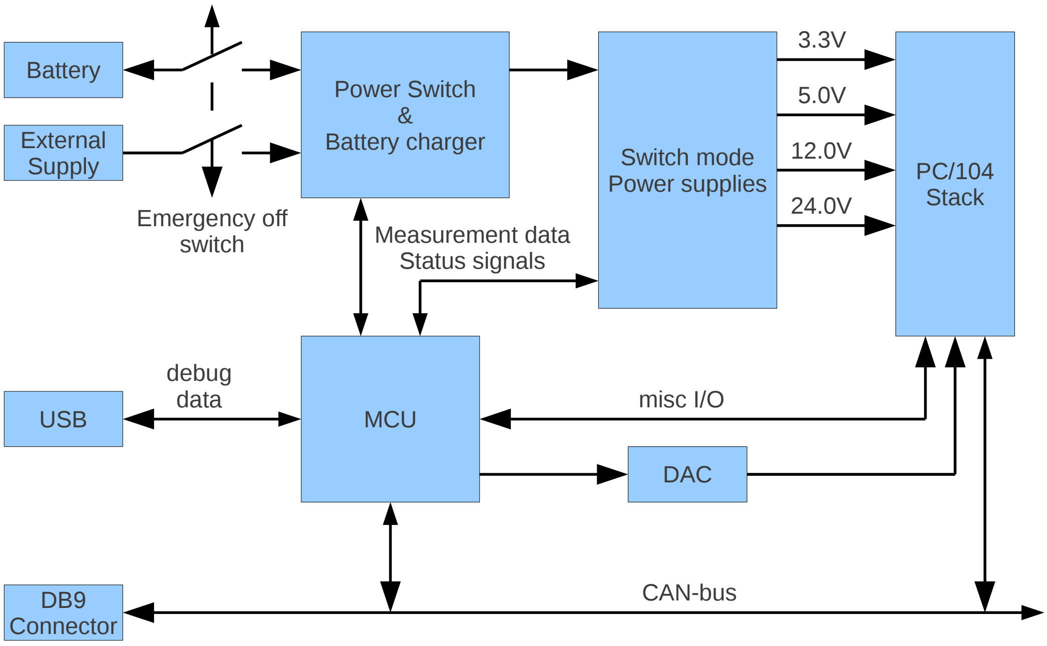


Figure 13.6: System Block Diagram

13.3.2 Battery charger

An on board battery charger was implemented. A charger chip from Linear Technologies was chosen (LTC4009) as the cell voltage of the battery pack can be programmed with external components. The chip monitors the battery voltage and regulates the charge current, it also detects when an appropriate external voltage is connected for charging. It was chosen to charge the battery when an external voltage of $24.0V$ or higher is connected to the circuit board.

A power path controller (LTC4416) was chosen to automatically change between the two main power sources, battery and external DC. When a DC voltage capable of battery charging is detected, the path controller switches the main power source from the battery to the external DC source while the battery is charged.

13.3.3 Power supplies

To support a wide variety of electrical components a wide variety of voltages has to be available. For a robot it is highly likely that actuators of some sort are used, so there also has to be a reasonable amount of current available. It was decided to create four regulated voltages, $24.0V$, $12.0V$, $5.0V$ and $3.3V$, they were specified to be able to deliver $5.0A$ each.

As the input voltage is always higher than $12.0V$, a Buck switch mode regulator chip from Texas Instruments (TPS5450) was chosen. Since the input voltage may vary from $15.0V$ to $22.8V$ from a battery, and a source of $24.0V$+ can be connected when charging a battery, the $24.0V$ regulator had to support input voltages higher and lower than the output. A general switching regulator controller (LM3488) from National Semiconductor was selected and used to design a buck-boost regulator.

13.3.4 Circuit board stack

To provide expansion of functionality the physical standard of the PC/104-plus stack[1]. This standard has a 104 pin ISA connector 2.54mm pitch and a 120 pin PCI connector, 2.00mm pitch. The ISA connector is used to provide power through the stack, 12 pins for each of the regulated voltages and 56 pins for ground. The PCI connector is: connected to the CAN-bus, has some ground pins and almost all of the I/O from the micro controller is connected here. For a complete list of the pin out and the physical dimensions of the stack refer to appendix C.3.

13.3.5 Circuit board

The schematics and circuit board layout was created using Altium Designer 10, figure 13.7 shows a 3D model of the finished circuit board. The complete schematic is available in appendix C.2. The three connectors in the upper left corner are the power input and a connector for an emergency stop button. They are from the Mini-Fit Jr series from Molex. Communication and programming connectors are located on the right hand side, one DB9-connector for CAN, one mini-USB for serial link to a computer and a JTAG header for programming and debugging of the micro controller. The connectors at the top and bottom at the center of the card is the PC/104 stack, these are from the ESQ and ESQT series from Samtec Inc.

As the power supplies can deliver a substantial amount of current it was decided to design a 6-layer board. The top and bottom layer is used for high current transmission and power ground. The inner layers are used for small signal routing of control signals, digital buses etc. The two remaining layers are used as a 3.3V and a ground plane to supply the digital logic on the board.

This circuit board was produced professionally by PCBCART [43] as we did not have any equipment available to make boards with more than two layers. It also gives several other advantages like silk screen print, plated holes, solder-mask and an overall better quality than "home production". All surface mount packages with pads underneath the component were soldered with professional soldering equipment at NTNU with proper temperature control and solder paste. The rest of the components were soldered by hand with a regular soldering iron.

13.4 Extension modules

The following cards were created to extend the functionality of the main circuit board. They both use the PC/104-plus stack. These circuit boards were produced with a circuit board CNC

[1] Note that there was a design error in spacing between the mounting holes for the PC/104-plus stack, the actual dimentions can be seen in appendix C.6.

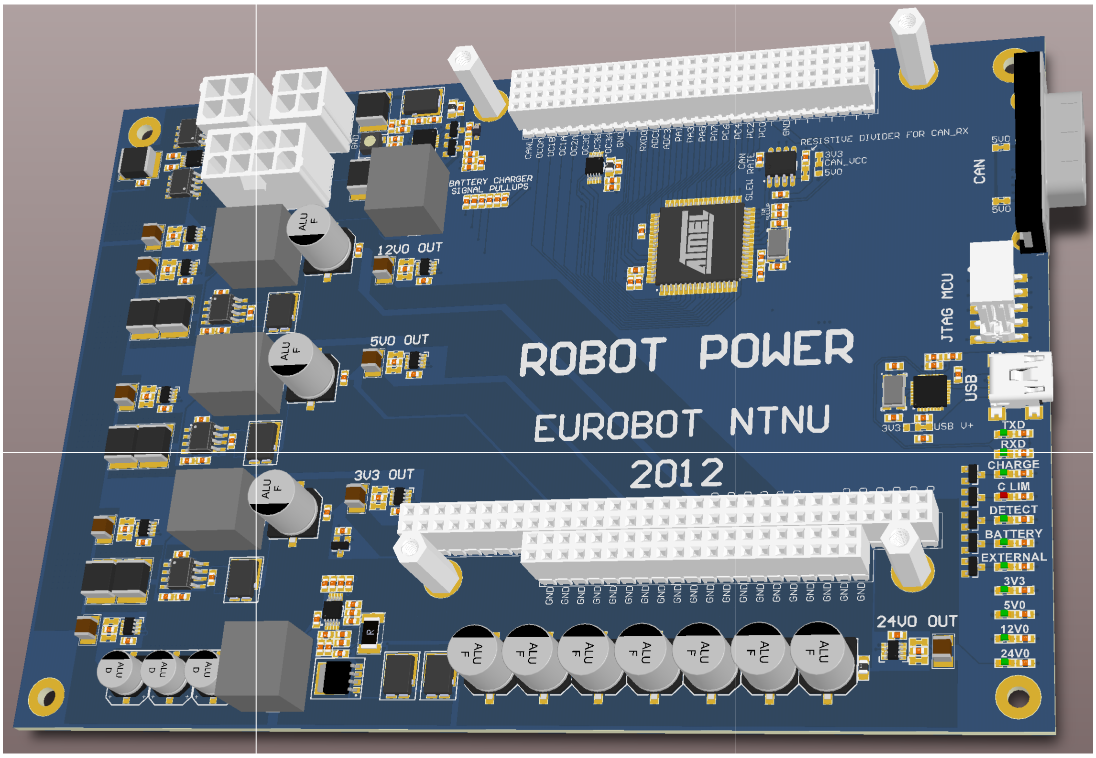


Figure 13.7: Circuit Board

located at the institute of Engineering Cybernetics at NTNU. There was produced a couple of revisions of both cards before the competition. The cards were created with a template design as shown in appendix C.6.

13.4.1 Motor control card

To support motor controllers a circuit board for the stack was reproduced from a previous design [54, p. 19]. It was found that the PC/104 headers didn't fit on the working prototype because the physical drilling was poorly executed, thus a new card was produced with a CNC. The circuit board contains two EPOS2/24 motor controllers from Maxon Motors. The motors are driven by 24.0V from the stack. The motor cards communicates via CAN bus through the stack. The schematic for the design is available in Appendix C.4.

13.4.2 Eurobot 2012 top card

A card was created with plugs for 4 servos, a connection for a start cord, connectors for two fans and a connector for the laser tower. No control logic is implemented here, everything is controlled from the base board. Schematics can be found in Appendix C.5.

13.5 Power card firmware

Firmware for the competition was implemented on the power card. It supports control of 5 servos, a start cord and it keeps track of time so the robot can be turned off after n seconds. Each servo can be turned on/off individually, a position can be set and the speed the servo moves from one position to another can be manipulated. The position is manipulated by setting the duty cycle of a $50Hz$ PWM signal, the duty cycle can be set from 0 to 100%.

Control and configuration of this functionality is done via CAN-bus, the package ID's conform to CANopen[54, p. 4] with node id 0x005. The timer package illustrated in figure 13.8 can be sent to the power card to configure how long the robot should run after the start cord is pulled. It can run from 0 to 255 seconds, where 0 is run forever. 0 is the initial value if packet is not sent. When the start cord is pulled the power card sends a package illustrated in figure 13.9 every time data is updated in the micro controller. The first byte tells if the timer is active, $0 = diabled$ $1 - 255 = enabled$. The second byte tells how long the robot has run for in seconds, $0 - 255$ and the third byte tells how long the robot will run before powering down in seconds, $0 - 255$.

A package illustrated in figure 13.10 can be sent to the power card to manipulate up to five servos. The first byte contains a servo ID: 0x0A to 0x0E, the rest of the bytes in the package will configure this device. The enable byte can be set or cleared to turn power to a servo on/off, $0 = off$, $1 - 255 = 0n$. The third and fourth byte is used to set the servo position, $0 = 0\%$ duty cycle, $65535 = 100\%$ duty cycle[2]. The speed the servo moves can be manipulated with the fifth and sixth bytes. As an example moving the servo from position 400 to 1100 with a speed of 20 would take $\frac{1100-400}{20} \cdot 20mS = 700mS$[3].

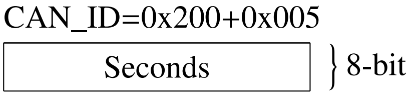


Figure 13.8: Timer RX message

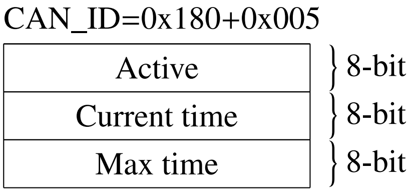


Figure 13.9: Timer TX message

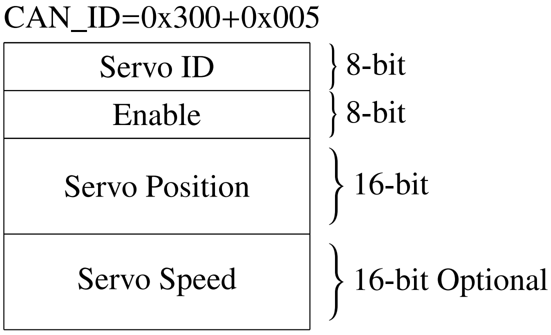


Figure 13.10: Servo RX message

[2] The servos in our robot was found to have an active range from 400-1100

[3] A speed equal to the active duty cycle range will move the servo as fast as the hardware allows it to

13.6 Switch mode power supply test

An efficiency test was carried out on the switch mode power supplies. To test the efficiency of a power supply one looks at the input and output power P_{in} and P_{out}. The efficiency is a ratio between P_{in} and P_{out}, and is defined as:

$$\eta = \frac{P_{out}}{P_{in}} \tag{13.8}$$

The efficiency of a switch mode power supply will vary with input voltage and output current, the output voltage will also vary some based on these parameters. Tests were therefore carried out with the input voltage at the batteries max and minimum voltage. During the test input and output voltage, current and power was logged for different loads from $0A$ to $3.5A$.

Test card

A card for the circuit board stack with separate plugs for GND, 3.3V, 5.0V, 12.0V and 24.0V was created to do power supply tests, its only function is to give access to the power cards voltages via 2.0mm banana-plugs.

Test setup

Figure 13.11 shows a block diagram of the test setup. An adjustable power supply is used to apply maximum and minimum battery voltages to the switch mode power supply that is tested. Input and output voltage, current and power is measured with the on-board current measurement chips. To test the efficiency with different loads automatically an electronically adjustable constant current load is used. A micro controller sets the desired load and requests input and output measurements, then sends the data to a computer. For small currents a variable resistor can be used instead of a constant current load, but variable resistors that can dissipate heat from $125W$ is expensive and the data logging process can't be automated.

Constant current load

An adjustable constant current load with a maximum $5.5A$ drawn has been made with an operation amplifier, one n-channel MOSFET and a 1Ω resistor. The schematic is shown in figure 13.12. When a constant voltage is applied to the positive input of the operation amplifier, in this case a pulse width modulated signal with a low pass filter. The output of the operation amplifier adjusts the gate voltage of the MOSFET to keep the negative input equal to the positive input. This creates a constant voltage equal to the applied voltage across a resistor which will then draw constant current from a supply.

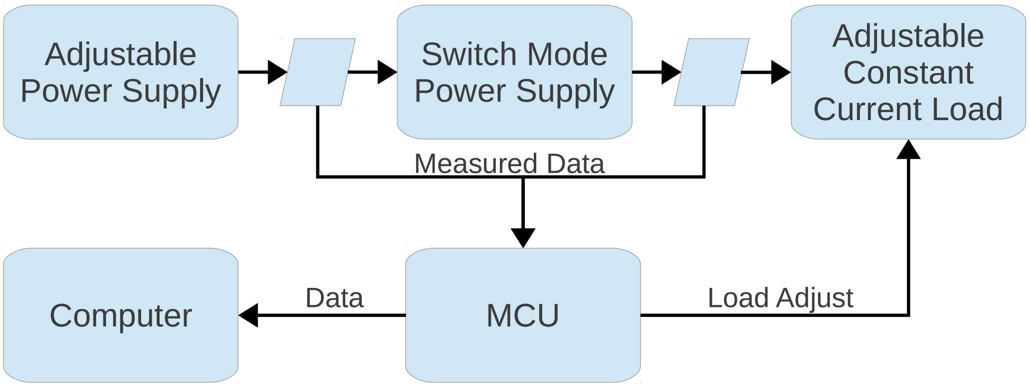


Figure 13.11: Efficiency test setup

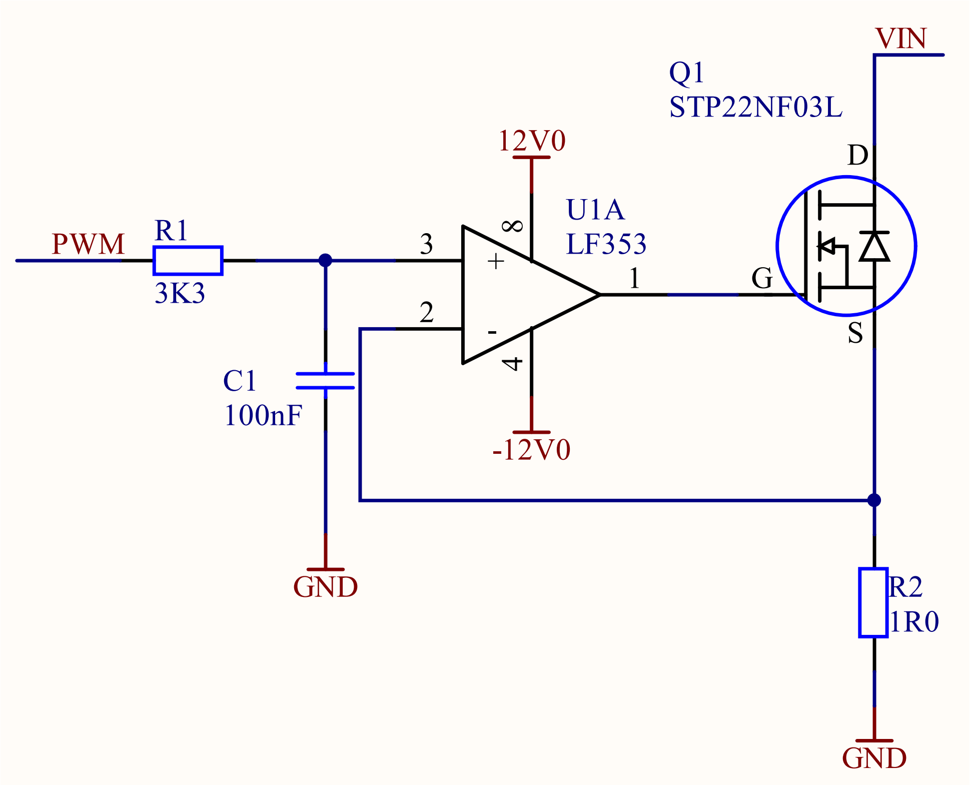


Figure 13.12: Constant Current Load

The majority of the power dissipation is in the MOSFET. Say a $24V$ supply is connected to the MOSFET and we apply $5V$ to the positive terminal of the operation amplifier. The voltage across the MOSFET will then be:

$$V_{ds} = V_{in} - V_R = 24V - 5V = 19V \tag{13.9}$$

The current is determined by the resistor value, as $R = 1\Omega$ the current will be $5A$. This yields the following power dissipation in the resistor and the mosfet:

$$P_R = V_R \cdot I = 5V \cdot 5A = 25W \tag{13.10}$$

$$P_{mosfet} = V_{ds} \cdot I = 19V \cdot 5A = 95W \tag{13.11}$$

For this design $6 \times 5W$ resistors in parallel is used for the 1Ω resistor so it can withstand $30W$. In the MOSFET the exact number of watts dissipated is not the problem, the main constraint is the heat caused by the power dissipation. In this design a STP22NF03L in a TO-220 package is used, this MOSFET has a maximum operation temperature of $175°C$. To withstand lots of power dissipation a heat sink is required, the heat sink has to absorb enough heat to keep the MOSFET within it's operation temperature. When power is dissipated in the MOSFET the temperature increase is relative to the ambient temperature. Heat sinks are specified in thermal resistance $^{\circ C}/_{W}$. Assuming that the package can transfer enough heat to the heat sink a simplified calculation shows that the heat sink needs the following thermal resistance to keep the MOSFET below $175°C$ if the ambient temperature is $20°C$:

$$T_{max} = T_{ambient} + T_{rise} = T_{ambient} + P * \theta_{sink} \tag{13.12}$$

$$\theta_{sink} = \frac{T_{max} - T_{ambient}}{P} = \frac{(175-20)°C}{95W} = 1.72^{\circ C}/_{W} \tag{13.13}$$

Test procedure

Firmware on the micro controller located on the power card implements a test procedure that gets measurements from 0 to 3.5A load. It does 100 measurements on this range in increments of 35mA. For each step it gets measurement data from the input and the output of the power supply under test. Each measurement is done 20 times and averaged, and then the results are sent to a computer via a serial link. Each power supply is tested with different input voltages that are within the main battery range, 16.0V, 19.5V and 22.8V. Appendix C.7 shows the physical test setup. In addition to the power card and current load one 6A lab supply was used as power source, and a second lab supply was used to power the load.

13.7 Results

Figure 13.14, 13.15, 13.16 and 13.17 plots the efficiency of the regulators at different input voltages. See appendix G for the raw measurement data. During the test of the 24V regulator the MOSFET short-circuited at $2800mA$ load, this can be seen in figure 13.17 as the next step is a $6A$ load. It was also noted that the efficiency of the 24V regulator is very high. A plot of the output voltage from all the regulators shown in figure 13.13 reveals that the 24V voltage regulator does not work, the output is equal to the input.

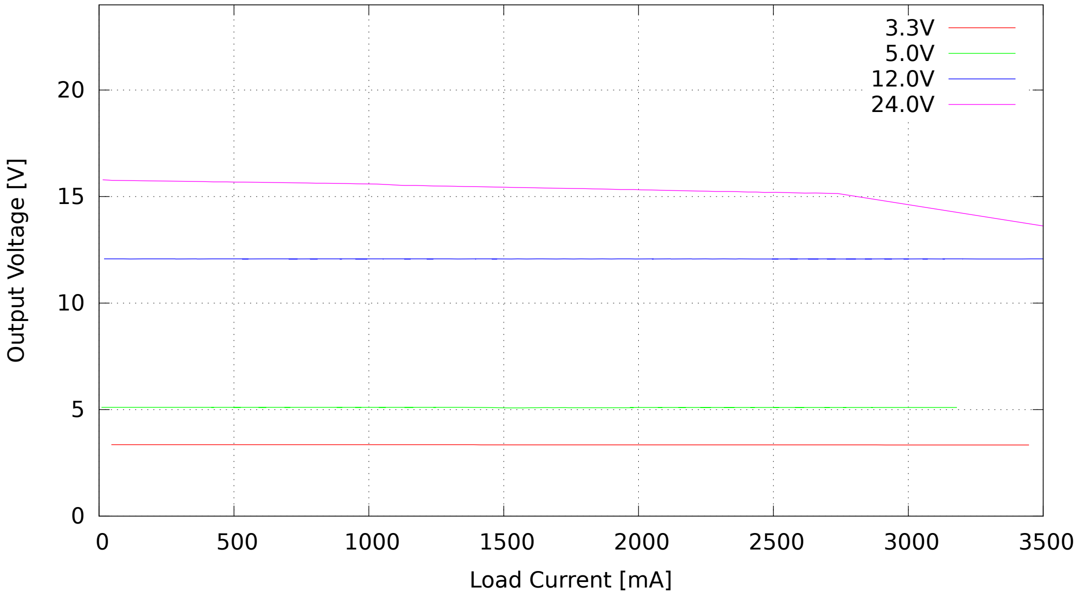


Figure 13.13: Voltage Stability

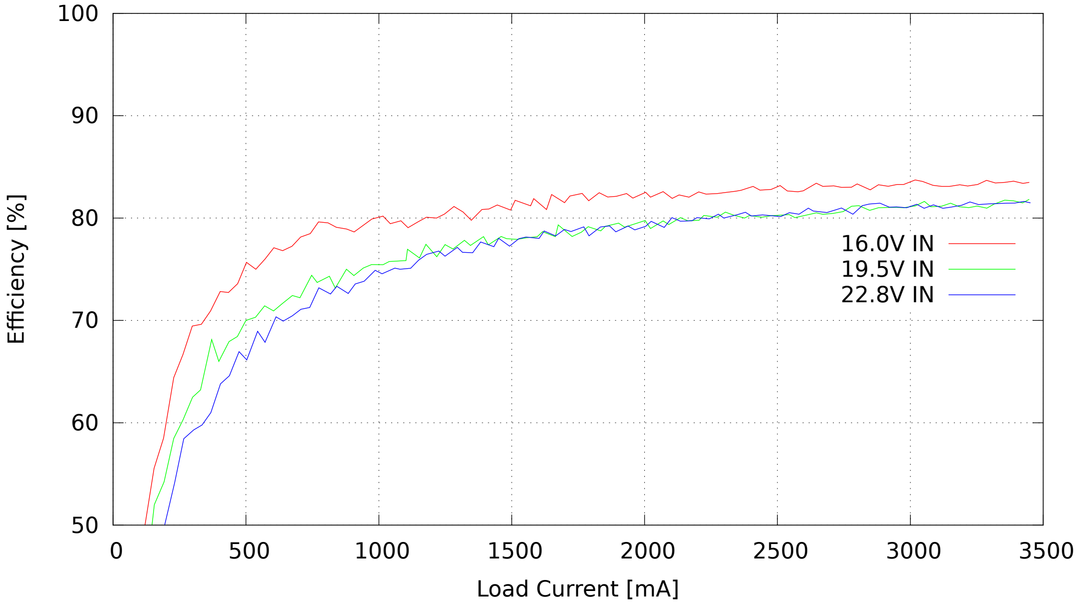


Figure 13.14: $3.3V$ Power Supply Efficiency

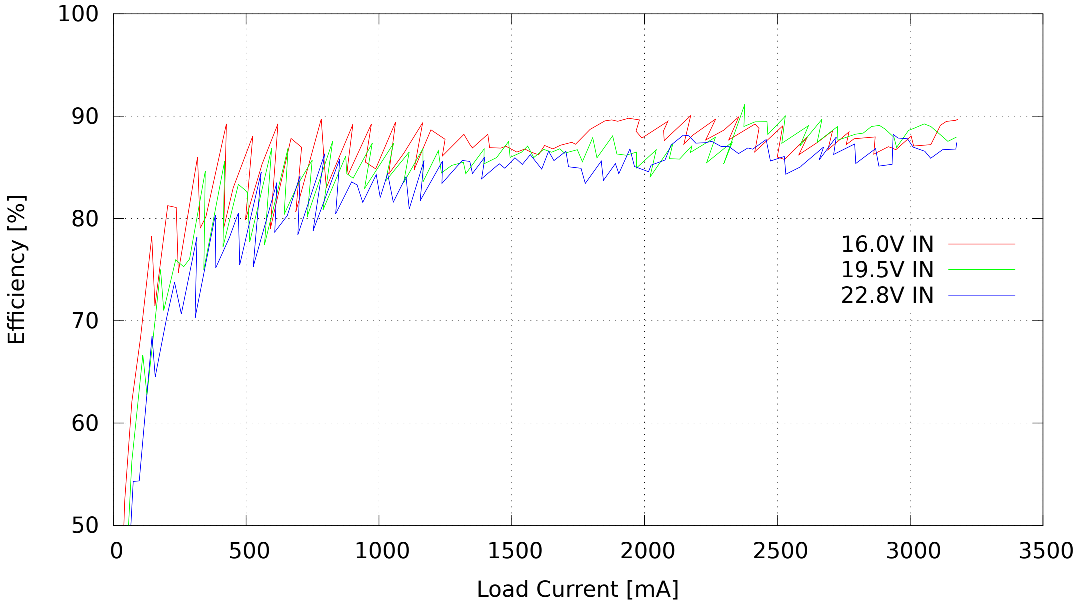


Figure 13.15: 5.0*V* Power Supply Efficiency

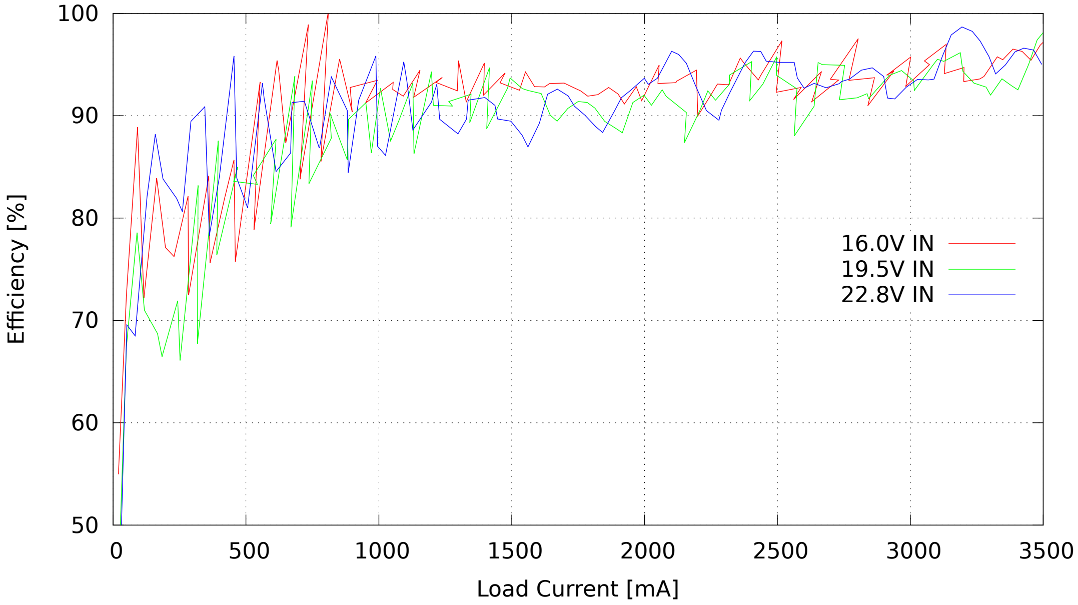


Figure 13.16: 12.0*V* Power Supply Efficiency

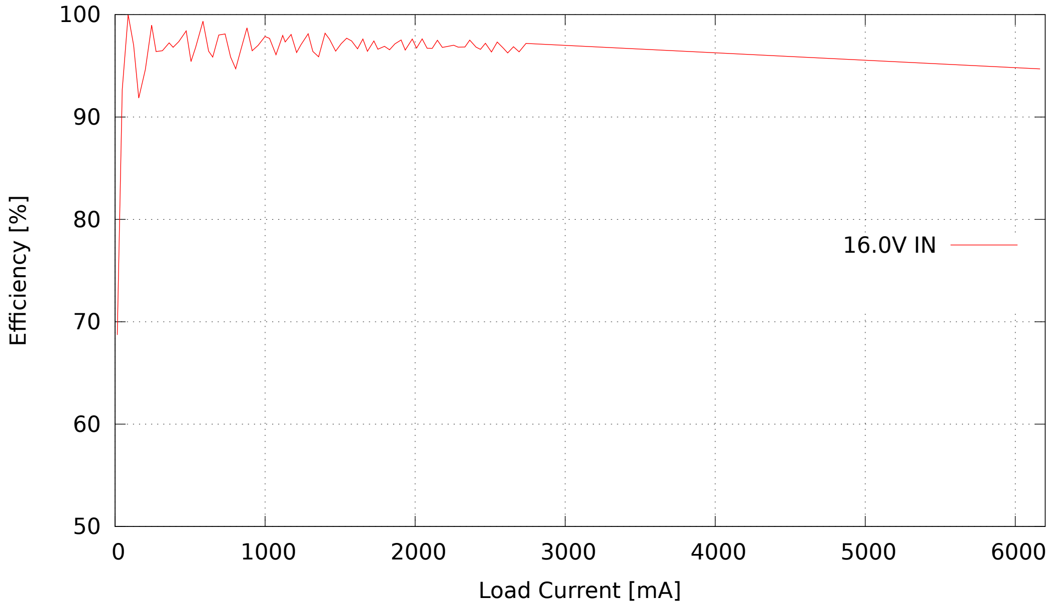


Figure 13.17: 24.0V Power Supply Efficiency

13.8 Discussion

13.8.1 Circuit board stack

The circuit board stack allowed us to create several prototypes for the actual functionality for the robot without having to re-design/re-solder micro controllers or power supplies. This saves a considerable amount of time and cost as these are usually expensive parts of an embedded design, however the connectors required for a PC/104-plus is very expensive. We actively used engineering samples from Samtec Inc to get these for free. If we did not have access to a circuit board CNC a PC/104 solution would be very unpractical as it is very time consuming and hard to drill the 224 required holes by hand.

13.8.2 Battery charger

The battery charger and the power path controller were briefly tested, the battery charger did not work and there were problems when both the battery and an external DC source were connected. It was discovered that the chosen power path controller does not support power switching the way the circuit board was designed. All testing of the battery charger / power path controller was discontinued as it was not an integral part for the system to work.

13.8.3 Switch mode tests

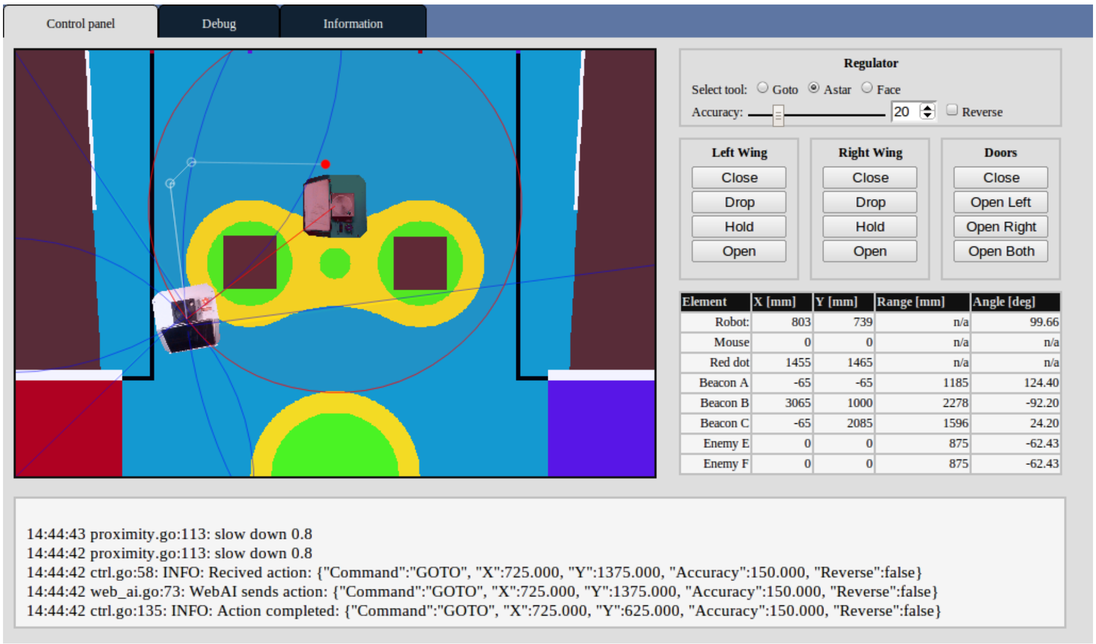
The constant current load was made by available spare parts and a large heat sink was selected without any calculations. When the efficiency tests are done high power dissipation occurs for a few minutes as the measurements are done automatically, this implies that the temperature in the selected heat sink does not have enough time to raise enough to exceed the MOSFETs max temperature. However, when testing the 24V regulator the temperature rise in the MOSFET was higher than the thermal resistance between the MOSFET package and the heat sink which caused it to break down. This could have been solved by using several MOSFETS in parallel on the heat sink, though it was not considered important to create a new load as enough results were logged.

More importantly the results show that the $24.0V$ does not regulate the output voltage correctly, however the robot has worked with this voltage as source for the propulsion system. This is because the main energy source, the battery, has a stable voltage between $90\% - 10\%$ capacity as mentioned in section 13.2.

To get better test results the tests should scan the entire load range of the power supplies to see how they behave close to the maximum load. This was not done due to limits in the prototype load, specifically a reference voltage of 5V was required to draw $5A$ trough the load. The micro controller controlling the tests is driven by $3.3V$.

Chapter 14

Software implementation



Regulator

Select tool: ○ Goto ● Astar ○ Face

Accuracy: 20 □ Reverse

Element	X [mm]	Y [mm]	Range [mm]	Angle [deg]
Robot:	803	739	n/a	99.66
Mouse	0	0	n/a	n/a
Red dot	1455	1465	n/a	n/a
Beacon A	-65	-65	1185	124.40
Beacon B	3065	1000	2278	-92.20
Beacon C	-65	2085	1596	24.20
Enemy E	0	0	875	-62.43
Enemy F	0	0	875	-62.43

```
14:44:43 proximity.go:113: slow down 0.8
14:44:42 proximity.go:113: slow down 0.8
14:44:42 ctrl.go:58: INFO: Recived action: {"Command":"GOTO", "X":725.000, "Y":1375.000, "Accuracy":150.000, "Reverse":false}
14:44:42 web_ai.go:73: WebAI sends action: {"Command":"GOTO", "X":725.000, "Y":1375.000, "Accuracy":150.000, "Reverse":false}
14:44:42 ctrl.go:135: INFO: Action completed: {"Command":"GOTO", "X":725.000, "Y":625.000, "Accuracy":150.000, "Reverse":false}
```

The debugging GUI

Chapter 7 and 9 states that the software should run on a HP Slate 2 tablet on Linux. A choice of the exact Linux distribution, as well as it's installation, needed to be carried out.

At the end of chapter 10, a set of software modules and their requirements are listed. These modules must be implemented.

Chapter 8 states that Go should be used for high-level code, and that C should be used for low-level code.

14.1 Algorithms and data structures

14.1.1 The A* algorithm

A* is an extension of Dijkstra's shortest path algorithm [51, p. 97]. It finds the shortest path from a "start-node" to a "goal-node", beginning at the start-node and expanding from there. All nodes on the edge of the explored are held in a list called the *open set*, while all explored nodes are held in the *closed set*. To decide which node to visit next, it use a heuristics function $f(x)$, and selects the node in the open set with the minimum expected distance.

14.1.2 Advanced data structures

Hash table

A hash table support *insert*, *search* and *delete*, all with expected average time of O(1) [9, p. 221]. A hash table consists of a directly addressable table, or array, and a hash function to map elements into the array.

Priority queue

A heap structure can be used to implement a min-priority queue [9, p. 138]. The min-priority queue supports a number of operations including *insert* and *extract-minimum*, both in $O(\log n)$. A min-priority is a vital component to a dijkstra shortest path implementation [9, p. 598].

14.2 An introduction to Go

This section will give a very brief introduction to the Go Programming language. As the Go language has much inheritance from the C language, it will be a priority to point out where the Go language is different from C. A longer introduction to Go can be found in Myren's prestudy [36] and on the language's official web pages [18]. Information about Go in this section is exclusively gathered from these two cites, though the examples are not.

14.2.1 Basics

Go has a syntax that is similar to C syntax in many ways. One of the largest differences from C syntax lay in variable and function declarations. Go uses the *var* keyword for variable declaration, *const* for constant declarations, and *func* for function declaration. Semicolons at the end of each line are not needed.

Listing 14.1: A simple Go program

```go
package main

import (
	"fmt"
	"errors"
)

func Int32Div(a, b int32) (result int32, err error) {
	if b == 0 {
		err = errors.New("Division by Zero")
		return
	}
	result = a/b
	return
}

func main () {
	var a, b, c int32
	var err error
	defer fmt.Println("Prints when main() returns")

	a = 300
	b = 10
	c, err = Int32Div(a, b)
	if err != nil {
		fmt.Println(err.Error())
		return
	}
	fmt.Printf("%d/%d ~= %d\n", a, b, c)
	return

}
```

Note that unless otherwise stated, all variables in Go are initialized to a default zero value. E.g. the default value of an integer would be *0*, while the default value of a pointer would be *nil*.

A neat feature of Go, is that a function can have multiple return values. Also, function calls can be queued in order to run at a later point in time by using the *defer* keyword. Defer functions are run in a FILO order when the function where the defer keyword was used, runs out of scope.

Listing 14.1 shows the full source code of a simple Go program, demonstrating most of the concepts that just have been explained. The Int32Div function here will return both an integer and an error. If nothing goes wrong, the error value will be nil.

14.2.2 Strongly typed

The Go language is much more strongly typed than C. Except for a very few exceptions, all variable casts must be explicit! E.g. an int64 cannot be implicitly casted to an int32 and vice versa. Even two Go types that have the same underlying type cannot be casted implicitly. The latter is demonstrated in listing 14.2.

Listing 14.2: Type casting

```go
type A int64
type B int64

var a A = 42
var b B

// The following will result in a compilation error:
b = a

// The following will work just fie:
b = B(a)
```

14.2.3 Structs

Listing 14.3: A struct with a member function

```go
type Abc struct {
	a, b int32
	c float64
}

func (t Abc) Float64Sum() float64 {
	return float64(t.a) + float64(t.b) + c
}
```

Struct are built up in much the same way they are in C, and are defined with the *type* keyword. However all Go types, including structs, are allowed to have member functions. Listing 14.3 shows a simple struct with a member function.

14.2.4 Interfaces

Interfaces define a set of functions. Any Go types that implement all these functions, will implicitly implement the interface. An interface with zero functions defined, will be implemented by all Go types.

As an example of an interface in the standard libraries, we have the *io.Writer*. It only defines one function, namely: Write(b []byte)(n int, err error). A pointer to any type that implements this one function can be passed to any function that expects an io.Writer.

14.2.5 Built-in concurrency

In Go, light-weight threads, often referred to as *goroutines*, can be started by prefixing a function call with the *go* command. These threads will be scheduled by Go's internal scheduler, who will multiplex queued goroutines onto a finite number of OS-threads. Listing 14.4 shows, among some other concepts, how goroutines are started.

14.2.6 Communication constructs

Listing 14.4: Solution to the Producer Consumer problem

```go
package main

import (
        "fmt"
        "time"
        "sync"
)

func Producer(c chan<- int, i int) {
        for {
                // Send to channel c
                c <- i
        }
}

// Runs for 5 seconds
func Concumer(c <-chan int) {
        var i int
        var timout chan time.Time = time.Tick(5*time.Seconds)

        for {
                // select switch for receiving from channal c or timeout.
                select {
                i = <-c: // <- the value from the channel is stored in i
                        fmt.Print(i)
                <-timeout; // <- the value from the channel is discarded
                        fmt.Println()
                        return
                }
        }
}

func main() {
        var c chan int = make(chan int, 5)
        var wg sync.WaitGroup

        go Producer(c, 0)
        go Producer(c, 1)
        wg.Add(1)
        go func() {Concumer(c); wg.Done()}() // <- functions can be defined and executed directly.

        wg.Wait() // <- wait until wg equals 0
}
```

For gorotines to become useful, a way to exchange data without risking race-conditions would be required. Standard concurrency constructs like Conditional variables, Mutexe and RWMutex are provided by the Go standard library. To be able to wait for goroutines to finish, the WaitGroup synchronization construct is provided. In addition, there is a more high-level concurrency construct, called a channel.

The channel is mainly used for message passing between different goroutines. When initialized, it can be set up to handle any Go type. The channel can in general be described as a bounded buffer where messages are received by the listener in a FIFO order. Channels are synchronous if they are initialized with a size of zero and asynchronous otherwise. To be able to listen to several channels simultaneously, the *select* keyword can be used. The select keyword can also be used to avoid blocking when trying to either read from an empty channel, or to send to a full channel.

When declaring a channel, an arrow into the channel (chan<- t) means it is a send-only channel, an arrow out (<-chan t) means it is a receive-only channel, and mo arrow either (chan t) way means the channel is bidirectional. Bidirectional channel will be semalessly casted when sent into a function that takes a directed channel.

Listing 14.4 displays a solution to a producer-consumer problem (see chapter 10). The program shows most of the concepts just mentioned regarding concurrency and communication constructs.

14.2.7 Error handling

In Go, there is a distinction between critical a non-critical errors. Normally, error conditions are handled with multiple return values, as shown in listing 14.1.

In case of severe errors, a panic can be raised with the *panic(s string)* function. A panic will end the current function, run any queued defer function, and then continue down the stack. A panic can only be recovered if a call to *recover()* can be found among the queued defer functions in the given goroutine. If a panic is allowed to propagate all the way down the stack of the goroutine where it occurred, the entire program will be terminated.

14.2.8 Package testing

Go provides support for automatic testing of packages. Testing is typically preformed through the use of the *go test* command line tool. Depending on the flags given, *go test* runs through some or all testing and benchmark functions. Testing functions are functions in the package named "*func TestX(*testing.T)*", where X can be any string. The benchmark functions are similarly named "*func BenchmarkX(*testing.B)*".

14.2.9 The time.Duration type

The *Duration* type, in the *time* package, is a type that is implemented to make it easier to handle time. The underlying type is an int64, where it will store the time in nanoseconds.

Listing 14.5 shows a set of convenience constants and member functions defined for the Duration type. All functions Hours(), Minutes(), etc. are used to extract the stored time value as a given time unit. The String() functions provides a way to represent the time in a sensible manner based on the values magnitude.

An example use-case of the convenience constants is shown in listing 14.4, where the *time.Tick* function is called with *5*time.Second* as an argument.

Listing 14.5: The Duration type

```
// Type declaration
type Duration int64

// Constants
const (
    Nanosecond Duration = 1
    Microsecond = 1000 * Nanosecond
    Millisecond = 1000 * Microsecond
    Second = 1000 * Millisecond
    Minute = 60 * Second
    Hour = 60 * Minute
)

// Member functions
func (d Duration) Hours() float64
func (d Duration) Minutes() float64
func (d Duration) Nanoseconds() int64
func (d Duration) Seconds() float64

func (d Duration) String() string
```

14.3 Choice of Linux distribution

In chapter 9, it was decided that Linux should be used on the HP Slate 2. However, the exact distribution to use, still need to be decided. The requirements for the distribution, is equal to the OS requirements; it must run well at the HP Slate 2, be stable and robust, familiar, allow CAN support and provide a good development environment.

The initial plan was to go for one of the big stable Linux distributions. As one of the most widespread and well known distribution, Ubuntu was initially chosen [11]. However, as Ubuntu and Debian had some problems with the consistently booting the Slate 2 and its Intel GMA500 graphics card, this choice had to be re-evaluated. After all, a “stable distribution” cannot be considered stable if it does not properly support the hardware.

It was considered that a more configurable and “cutting edge” distribution would be more likely to properly support the HP Slate 2 hardware. As alternatives familiar to the team, one could use either Archlinux or Gentoo. Archlinux is dedicated to keep the implementation clean and simple and to provide the newest software available [61]. The distribution is highly configurable and agile, but has a greater risk of breaking on system updates than more tested distributions. Archlinux was considered stable enough as long as it wasn’t unnecessary updated at critical times.

Gentoo is a highly configurable and optimized distribution were software packages, in principal, are only provided in source form [17]. Everything thus needs to be compiled by the user. For this task, Gentoo provides a combined build system and package manager that makes package configuration and optimization easy [1]. As the resources of the Slate 2 platform are quite limited, it was decided to not waste too much time on compilation. The team therefore decided to use

[1] Applies to users with a good understanding of software compilation

Archlinux.

14.3.1 Linux configuration

As mentioned in chapter 10, Linux CAN is the kernel module we wanted our software to use for CAN communication. The default Archlinux kernel is compiled without Linux CAN, so the kernel had to be recompiled with the can module enabled. Archlinux makes it easy to recompile the kernel [3]. The kernel was recompiled with support for CAN. In the same go, some modules that were considered unnecessary were striped away to save compilation time. To support our CAN USB-device, drivers from Peak also had to be installed. This procedure is explained in the software's *README* file, which is available as a digital attachment, see appendix G.

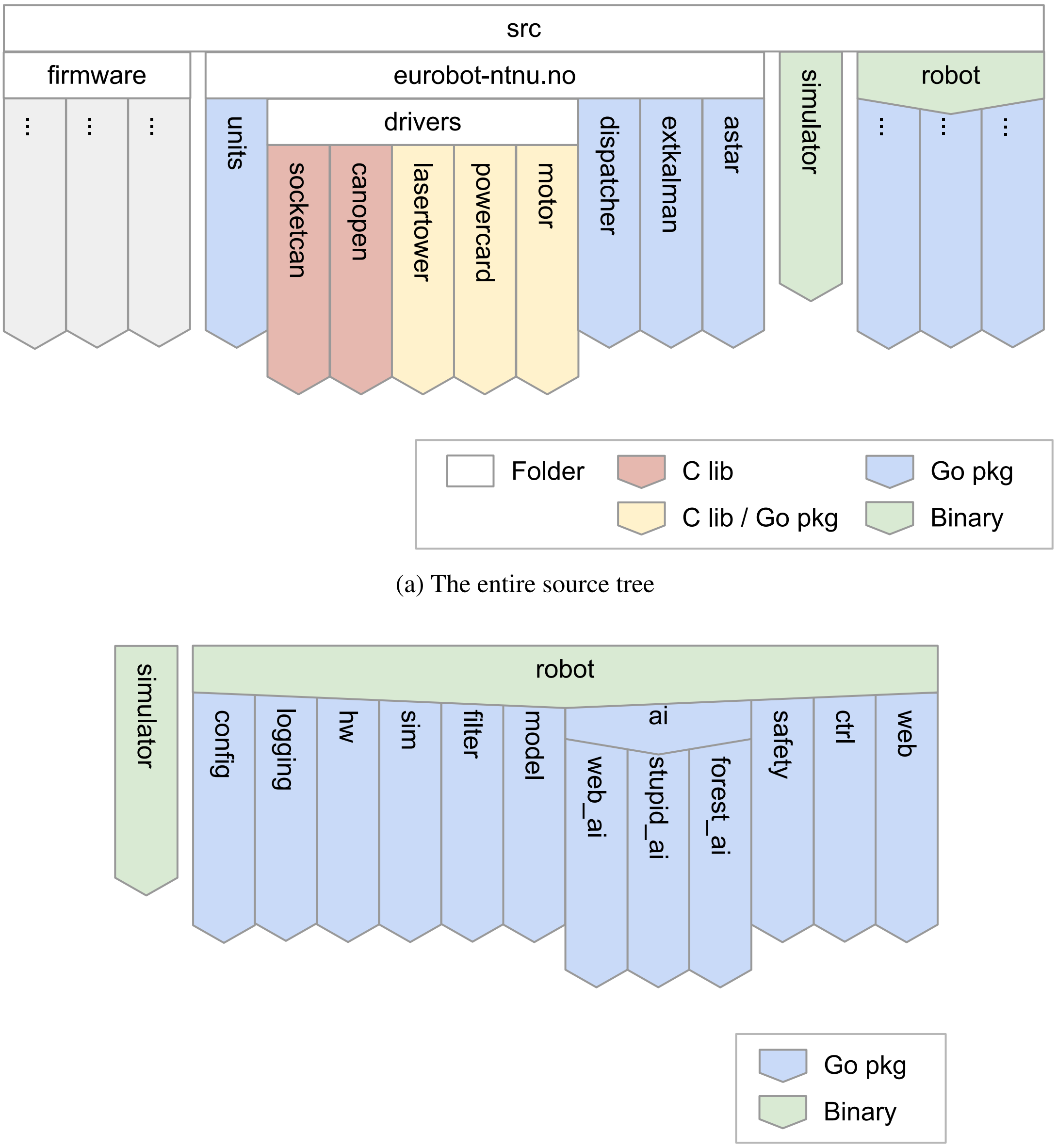
14.4 Software overview

As Go refers to software modules as *packages*, this is the term that we will use when we talk about implemented software modules.

In chapter 10, a set of modules and their requirements were defined. Figure 14.1a and 14.1b shows an overview of what packages were actually implemented, and how they were organized in the filesystem hierarchy. The packages are ordered in such a way that any given package in the figures, only depend on packages to their left. We see that all the general purpose libraries have been placed inside the *eurobot-ntnu.no* folder. The task-specific code is organized as sub-packages of the *robot* package. The firmware folder contains the software that's used on the microcontrollers, and will not be discussed in detail here.

In addition to modules defined in chapter 10, a goroutine dispatcher and a stand-alone C simulator has been added. There are also some additional sub-packages inside the robot package, such as the *logging* and *config* packages. If we compare figure 14.1a and 14.1b with figure 10.2, we see that the names have been changed from conceptual names to actual package names. It also seems like the task-specific pathfinder and regulator modules have vanished. What has actually happened is that the task-specific pathfinder code has been made part of the *ai* package, and the regulator has been made part of the *ctrl* package.

In the following sections there will be a focus on explaining significant implementation details for the various software packages. Any significant addition or aversion from the design will be highlighted.



(a) The entire source tree

(b) Task-specific packages

Figure 14.1: Implemented software packages

14.5 Go libraries

14.5.1 The Units library

Listing 14.6: The Velocity type

```go
// Type declaration
type Velocity float64

// Constants
const (
        NanometerPerSecond = 1e-6 * MillimeterPerSecond
        MicrometerPerSecond = 1e-3 * MillimeterPerSecond
        MillimeterPerSecond Velocity = 1
        CentimeterPerSecond = 1e1 * MillimeterPerSecond
        MeterPerSecond = 1e3 * MillimeterPerSecond
        KilometerPerHour = 1e3 * MeterPerSecond / 60 / 60
)

// Member functions
func (v Velocity) MillimetersPerSecond() float64
func (v Velocity) CentimetersPerSecond() float64
func (v Velocity) MetersPerSecond() float64
func (v Velocity) KilometersPerSecond() float64

func (v Velocity) MultiplyWithDuration(t time.Duration) Distance
```

Listing 14.7: Assorted functions from the Units library

```go
// Returns the distance from c1 to c2.
func (c1 Coordinate) Distance(c2 Coordinate) Distance

// Given a polar-axis parallel to the x-axis, this function
// returns the angle of the vector going from c1 to c2 in range
// -PI to PI.
func (c1 Coordinate) Angle(c2 Coordinate) Angle
```

The implementation of the units-library was heavily inspired by the time.Duration type in the standard libraries (See section 14.2, page 92). Initially, the types *Distance*, *Velocity*, *Angle*, and *AngularVelocity* were defined, all with float64 as the underlying type. Convenience constants and converter functions similar to those defined for time.Duration, was provided for each of these types. In what unit each underlying type is stored, should be irrelevant for the user, as long as the convenience constants are used for initialization, and the converter functions are used to extract the values. As an example, the constants and converter functions for the Velocity type is provided in listing 14.6.

On top of the distance type, a Coordinate type has been defined. The underlying type is a struct containing two Distance objects named X and Y.

To ease some conversion between different types, a set of mathematical/physics operations are provided. As shown in listing 14.6, the Velocity type provides a *MultiplyWithDuration* function that returns a Distance. Likewise, the Distance type provides a *DivideWithDuration* function that returns a Velocity. Some more examples are shown in listing 14.7.

14.5.2 The Dispatcher library

Listing 14.4 on page 91 shows how the WaitGroup type can be used to wait for a goroutine's completion. It does not show how to safely terminate a goroutine. The best way to safely terminate a goroutine would probably be to use a channel. Defer functions can be used to make sure that shut-down routines and last wishes are carried out. However, when multiple goroutines that need safe termination are started, some questions arise:

- Should each goroutine have its own termination channel and wait group, or should they share these constructs?
- How large should the buffer size of this termination channels be?
- What should happen if the buffer is filled?

The task-specific code of this year's robot required numerous goroutines, many of which was believed to need a safe termination. To standardize how these goroutines were started, a general-purpose dispatcher library was developed. It is a small library, less than 40 lines of actual code, but as it was a desire to write it thread-safe, it took several tries to get it just right. The entire source code, excluding package tests, can be found in appendix D.1.

To create a goroutine with the dispatcher library, the *dispatcher.Interface* interface must be implemented. This interface defines only a single function: *Run(termc <-chan bool)*. Implementations should periodically check the termc channel, and return if something is received.

When a type implementing this interface is passed to the *NewGoroutine(...)* function, the Run function is called in a new goroutine, and a pointer to a Routine instance is returened. The Routine instance provide two member functions: *Terminate()* and *Wait()*.

The Wait function blocks until the Run function has safely terminated. It should be possible to call this function from several goroutines simultaneously without causing race-conditions.

When the Terminate function is called, one of two things occurs:

- If the termc channel is not full, the Terminate function sends a value on the termc channel and then waits until the Run function has safely terminated.
- If the termc channel is full, the Terminate function waits until the Run function has safely terminated.

Internally the dispatcher used a Go WaitGroup. If the Run function should terminate before or during a call to the dispatcher's Terminate function, the Terminate function will return at once, since there is nothing to wait for. It should be possible to call the Terminate function from several goroutines simultaneously without causing race-conditions.

Listing 14.8: The ExtKalman struct

```go
// Type definition:

// System equation: v and w are white noise, Q and R are design matrices, P0
// and x0 describe the initial system state.
//
// x(k+1) = f(x,u) + W*w
// y(x) = h(x,u) + v
//
// Q(k) = E[w(k) w(j)^t], where j=k else 0
// R(k) = E[v(k) v(j)^t], where j=k else 0
type ExtKalmanFilter struct {
       // System State
       x matrix.Matrix // State vector
       mP matrix.Matrix // Error covariance matrix

       // Const system matrices
       mW matrix.Matrix // Which states are noisy
       mQ matrix.Matrix // Process noise variance
       mR matrix.Matrix // Measurement noise variance
       cWQWt matrix.Matrix // W*Q*W^t

       // Function pointers
       f func(matrix.Matrix, matrix.Matrix) matrix.Matrix // x = f(x,u)
       dfdx func(matrix.Matrix, matrix.Matrix) matrix.Matrix // F = df(x,u)/dx
       h func(matrix.Matrix) matrix.Matrix // y = h(x)
       dhdx func(matrix.Matrix) matrix.Matrix // H = dh(x)/dx

       // Useful to normalize angle error
       normalize_ydiff func(matrix.Matrix) matrix.Matrix
}

// Initializers:
func Kalman(W, R, Q, x0, P0, A, B, H matrix.Matrix) (k *ExtKalmanFilter)
func ExtendedKalman(
              W, R, Q, x0, P0 matrix.Matrix,
              f, dfdx func(matrix.Matrix, matrix.Matrix) matrix.Matrix,
              h, dhdx, normalize_ydiff func(matrix.Matrix) matrix.Matrix) (k *ExtKalmanFilter)

// Member functions:
func (k *ExtKalmanFilter)Step(y, u matrix.Matrix)(x matrix.Matrix)
```

14.5.3 Extended Kalman filter library

In chapter 10, it was decided that an extended Kalman Filter library should be developed in Go, based on the Gomatrix project. The Gomatrix library provides linear algebra operations for both dense and sparse matrices. The implementation is based on the definitions in “Integrated Satellite and Inertial Navigation Systems” [60, p. 103]. Also, even though code from the KFilter library could not be used directly, it was useful for testing [63]. This made it relatively simple to implement the functionality of the *extkalman* library.

The final implementation uses the *ExtKalmanFilter* struct for storing the system matrices, function-pointers and the last known state and variance. This struct and its associated functions are shown in listing 14.8. Please note that *Gomatrix* has no concept of vectors, as such vectors are implemented as “uni-column” matrices.

The filter is initialized by calling the *ExtendedKalman(..)* function, which takes a set of system matrices and function pointers as input. The function pointers are used to implement the nonlinear behavior and to linearize the nonlinear behavior. When initialized, the filer should be used by calling *Step(...)* periodically. This function takes in the latest available measurements (y) and the actuation (u), and returns the new estimated system state.

Just for good measure, a normal Kalman filter was implemented on top of the extended Kalman filter. The function *Kalman(..)* takes a set of system matrices as input, and generates the necessary function pointers to run the Kalman filter as an extended Kalman Filter.

14.5.4 Optimizing the A* library

As described in chapter 10, the a-star implementation “Gopathfinding” was used. It perfomed the A* search in four direction.

A quick benchmarked was developed that measured the time to find a path from position (0,0) to (99,99) on a 100x100 map with a wall/obstacle line going from (55,20) to (55,80).

The tool parameters, and environment variables, needed to reproduce the results are preserved in the script *run_bench.bash*, found in appendix D.4. The benchmark was run on a *HP Slate 2* tablet [40], resulting in 3.08 seconds per calculation on a 100x100 map. This was deemed too slow for real-time path calculations, and unnecessary so.

Search optimizations

The implementation was profiled to recognize hot spots. The result can be seen in table 14.1. From the result it was concluded that a lot of time, that was spent managing the closed-set. The implementation iterates over the closed set on each step to check if the current node has been visited before. To reduce this overhead a hash table could be used.

The immense slowdown from closed-set problem overshadows any other time consuming parts in the profiling data. However, studying the code shows that a considerable amount of time is spent repeatedly finding the minimum node in the open set. Since the open set is stored as an array, the code has to iterate through the whole list each time to find the minimum node. It would be better if the open set could be stored as a min-priority queue.

As it was considered useful, and had little effect on performance, the A* library was modified to search in eight direction instead of four.

Resulting improvements

After switch to using a hash table for the closed set, and a min-priority queue for the open set, the HP Slate *HP Slate 2* used 0.12 seconds per calculation. Complete benchmarks results are provided in section 14.10.

In function		**Function and callees**		
Hit count	**Percent**	**Hit count**	**Percent**	**Function name**
244	73.5%	245	73.8%	.../gopathfinding.(*Graph).Node
80	24.1%	80	24.1%	.../gopathfinding.hasNode
4	1.2%	4	1.2%	.../gopathfinding.PathCost
1	0.3%	1	0.3%	ReleaseN
1	0.3%	1	0.3%	.../gopathfinding.removeNode
1	0.3%	2	0.6%	runtime.growslice
1	0.3%	1	0.3%	sweep
0	0.0%	246	74.1%	.../gopathfinding.(*Graph).adjacentNodes
0	0.0%	331	99.7%	.../gopathfinding.Astar
0	0.0%	331	99.7%	.../gopathfinding.BenchmarkAstar4Dirs100x100
0	0.0%	1	0.3%	growslice1
0	0.0%	1	0.3%	makeslice1
0	0.0%	1	0.3%	runtime.GC
0	0.0%	1	0.3%	runtime.MCache_ReleaseAll
0	0.0%	2	0.6%	runtime.gc
0	0.0%	1	0.3%	runtime.mal
0	0.0%	1	0.3%	runtime.mallocgc
0	0.0%	332	100.0%	schedunlock
0	0.0%	1	0.3%	stealcache
0	0.0%	332	100.0%	testing.(*B).launch
0	0.0%	332	100.0%	testing.(*B).runN

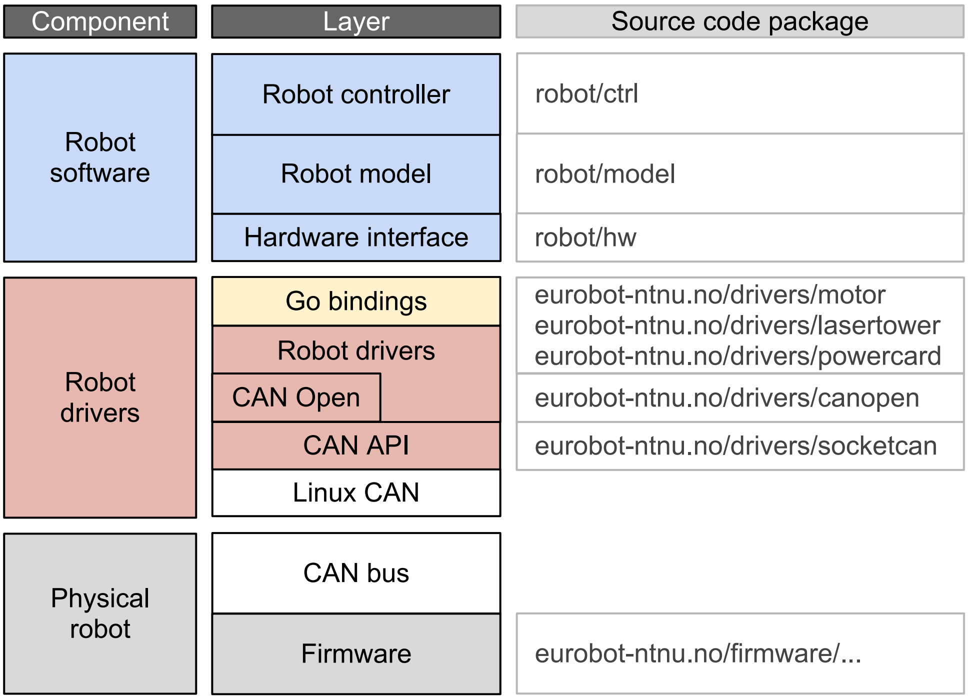
Table 14.1: A* profiling result

A* map generation

A map of known static obstacles on the playing area needed to be created. In the implementation a map is represented as a two dimensional array.

To ease the creation of this map a PNG-image to array converter was created. The converter reads a black and white image and outputs a map usable to the A* algorithm.

14.6 The hardware abstraction layers



Component	Layer	Source code package
Robot software	Robot controller	robot/ctrl
	Robot model	robot/model
	Hardware interface	robot/hw
Robot drivers	Go bindings	eurobot-ntnu.no/drivers/motor eurobot-ntnu.no/drivers/lasertower eurobot-ntnu.no/drivers/powercard
	Robot drivers	
	CAN Open	eurobot-ntnu.no/drivers/canopen
	CAN API	eurobot-ntnu.no/drivers/socketcan
	Linux CAN	
Physical robot	CAN bus	
	Firmware	eurobot-ntnu.no/firmware/...

Figure 14.2: Implemented hardware abstraction layers

14.6.1 Device drivers

Figure 14.2 show which packages that implements the different hardware abstraction layers. This section will discuss the implementation of the *Robot drivers* block.

To allow reuse of the CAN communication code, the two libraries *canopen* and *socketcan* was extracted from the original motor drivers written in 2011 [54]. These libraries can be seen as the *CAN Open* and *CAN API* layers in figure 14.2.

Adapting the motor driver from 2011

The original motor driver used the *ppos* mode of the Maxon EPOS2 Motor Controllers [54]. In ppos mode, the drivers had a function where you could ask any of the two robot wheels to drive a certain distance in a given direction. This function takes the id of the controller commanding either the left or the right motor. It would send the required CAN messages to ask the controller to run a certain number of steps in a certain direction. On top of this one function, there were implemented two functions; one to either drive the robot straight ahead or straight backwards, and one to rotate the robot a certain number of degrees in a given direction.

Due to the navigational choices from chapter 5, it was necessary to change the drivers to use the controller's velocity reference mode. The new drivers export functions that allow the user to set the velocity and angular velocity reference values for the robot. When these functions are used, the driver will calculate the RPM reference value for each of the two motors, and send this data to the controllers via CAN.

Driver tests

Most of the drivers supply one or more source files prefixed by "_test.c". These files would compile into binaries that could be used to test basic functionality for each device driver individually.

Preparation for AVR support

Listing 14.9: Functions exported by the eurobot/socketcan library

```c
int socketcan_open(uint16_t filter[], uint16_t filtermask[], uint16_t num_filters);
void socketcan_close(int fd);
int socketcan_read(int fd, my_can_frame* frame, int timeout);
int socketcan_write(int fd, uint16_t id, uint8_t length, Socketcan_t data[]);
```

Even though the drivers can not yet be compiled to run on an AVR, a structure for compiling the drivers to this platform has been implemented.

The *socketcan* library defines only four functions which signature is shown in listing 14.9. To port any of the robot device drivers to a new platform, these are the only four functions that need to be implemented.

Inside the socketcan package, there are two different .c files that include a definition for the four socketcan functions. The file *socketcan_linux.c* implements the functions by using the Linux CAN API as a backend. The file *socketcan_avr.c* currently only holds dummy functions.

A makefile structure was set up, so that all the device drivers could be compiled to a different target platform by supplying a different make target. By using the command "make install-linux", a dynamically linked .so library would be compiled from socketcan_linux.c and installed to one of *lib/linux_amd64/eurobot* or *lib/linux_386/eurobot*, depending on your systems architecture. By using the command "make install-avr", a statically linked .a library would be compiled from socketcan_avr.c and installed to *lib/arv/eurobot*. The device driver makefiles use the *include* statement to avoid redefining the logic for each and every driver.

Go wrappers

The Go wrappers link to the generated Linux .so files, and makes sure not to export C functions that are written for internal usage only. The Go functions use the Units library for input parameters and returned measurements. This way, the user of the library does not need to worry about what unit scaling or numeric type the C driver operates with. Rather than returning the integer error codes that are returned by the C drivers, the Go wrappers typically operates with Go errors. The one exception is the initialization functions. For these functions a *panic* is raised if something goes wrong. Read functions are blocking but provide the possibility to specify a timeout, just like the C functions.

14.6.2 The stand-alone C simulator

To use the stand-alone C simulator, the Virtual CAN Network driver (vcan) should be installed and a virtual CAN-bus should be set up. Alternativly two computers could be connected by CAN, one running the simulator and the other the software being tested.

Consider that the different hardware abstraction layers shown in figure 14.2. When the simulator is used, a message from the robot controller would propagate all the way down to the Linux CAN kernel module. From there it will be directed onto the virtual CAN-bus, go back up to the CAN API, and then end up within the C simulator. When the C simulator feels like sending a message, it will basically go back the same way via the virtual CAN-bus.

The stand-alone C simulator can be used to test the entire software stack, including the C drivers. It will simulate a single virtual robot, and move it according to detected velocity references. The simulator is mostly stateless. E.g. initialization messages and stop messages from the drivers would mostly be discarded. This meant that the simulator and the robot software could be restarted in an asynchronous pattern during testing.

14.6.3 The robot hardware interface

Listing 14.10: The Hardware interface

```go
type Hardware interface {
        dispatcher.Interface
        GetMeasurements() Measurements
        ServoSetPos(id ServoID, pos ServoPos)
        MotorSetVelocities(movement units.Velocity, rotation units.AngularVelocity)
        MotorHalt()
        MotorEnable()
}
```

This section will discuss the bottom layer in the *Robot software* block in figure 14.2. To simplify the implementation of a Go simulator, a Go interface that delineated how the robot code could interact with the hardware, was defined inside the *robot/hw* package. The code for this interface is shown in listing 14.10.

We see that the Hardware interface extends dispatcher.Interface. This means that implementations must provide a *Run(termc <-chan bool)* function, as explained in section 14.5.2. Typically, this Run function should run hardware pollers and periodically check the termc channel. When something is received on the termination channel, the function should make sure that all hardware is turned off and then return. When that is done, the function can return.

A struct called Measurements was defined to store the most recent data readings from all device drivers in one place. How and when these measurements should be updated, are up to the exact hardware implementation to decide. The Measurements struct looks the same, regardless of how measurement readings are retrieved in the exact hardware implementation.

The struct that implements the real robot hardware for this year's robot is called "RealHardware". It uses defer functions to make sure that all hardware is safely turned off.

14.6.4 The Go simulators

In chapter 10, it was decided that a Go simulator should be implemented. Initially, a simulator to simulate the main robot was implemented. To better test the AI and Oponeont Avoidance system, it was considered useful to also be able to simulate at least one oponent robot. Thus there are two different simulator implementations.

All in all, the hardware interface in the *robot/hw* package is implemented by the following types:

robot/hw.RealHardware struct: This struct interfaces the actual robot hardware.

robot/sim.SimulatedHardware struct: This struct has it's own simulator that simulate a single robot.

robot/sim.Simulator struct: This struct can simulate both our robot and one opponent. It can also be used to only simulate the opponent, and to use real hardware for our robot. Behind the scene, the simulator uses two instances of the *robot/model.Robot* struct that it connects to instances of either RealHardware or SimulatedHardware.

robot/model.Robot struct: This struct should be connected to an instance of one of the three types mentioned above. This struct also implements the Hardware interface. Each time a function defined in the Hardware interface is called on the Robot struct, it stores some reference values within the struct, and calls the function with the same name in the connected hardware instance. As will be explained in section 14.7.5, the Robot's functions may modify velocity references before they are passed on to the hardware instance.

The Go simulators can be activated upon execution of the robot software by setting a flag, as described in the *README* file in the repository (see appendix G). The exact simulator that is started, and the simulators configuration, will depend on the value of this flag. When the *SimulatedHardware* is used, messages from the robot controller will not propagate out of the *robot software* block shown in figure 14.2.

14.7 The robot's main program

Listing 14.11: Robot comand line options and default values

```
\$ robot --help
Usage of robot:
  -ai="forest": AI configuration [web|stupid|forest|passive|offensive]
  -cord=true: Wait on startup cord
  -laser=true: Whether to initialize and start the laser tower
  -motor=true: Whether to initalize and start the motors
  -power=true: Whether to start the power card poller
  -sim="off": Simulator mode [off|us|enemy|all]
  -team="auto": Team configuration [auto|blue|red]
  -web=true: Whether to enable the web server
```

This year's robot software compile to a single binary named *robot*, same as the package name. The porgram can be started with a series of comandline options, as shown in listing 14.11. Different AI and simulator configurations can be chosen. For debuging, the initalization of individual hardware compoents could be left out. If it isn't specified, the team will be auto-detected on start-up based on the position of the fixed beacons.

All code inside the *robot* package or any of it's sub-packages, can be said to be more or less task-specific. An explanation of the internals of the robot packages folows.

14.7.1 Communication constructs

As mentioned in section 14.2, the Go programming language provide *channels* that can be used for both synchronous and asynchronous communication among threads. Though this is a good construct for concurrent communication, it is not always the right choice to use it. Some times more classical synchronization concepts are better suited.

Some requirements for when different constructs should be used, was defined:

Function calls without protection should be used to retrieve data that do not change, and when sending commands that do not need to be processed by another goroutine. E.g. it should be used for retrieving configuration and for commanding the hardware.

Function calls relying on an RWMutex[2] should be used for variable data when only the latest data is of interest. E.g. it should be used for accessing the measurements in the *robot/hw* package.

Synchronous channels should be used when commanding another goroutine to perform a task. E.g. the AI should use this construct to command the controller.

Asynchronous channels should be used when notifying another goroutine about an event, or when it is important that a goroutine does not block. E.g. it should be used when the controller notifies the AI about a completed command.

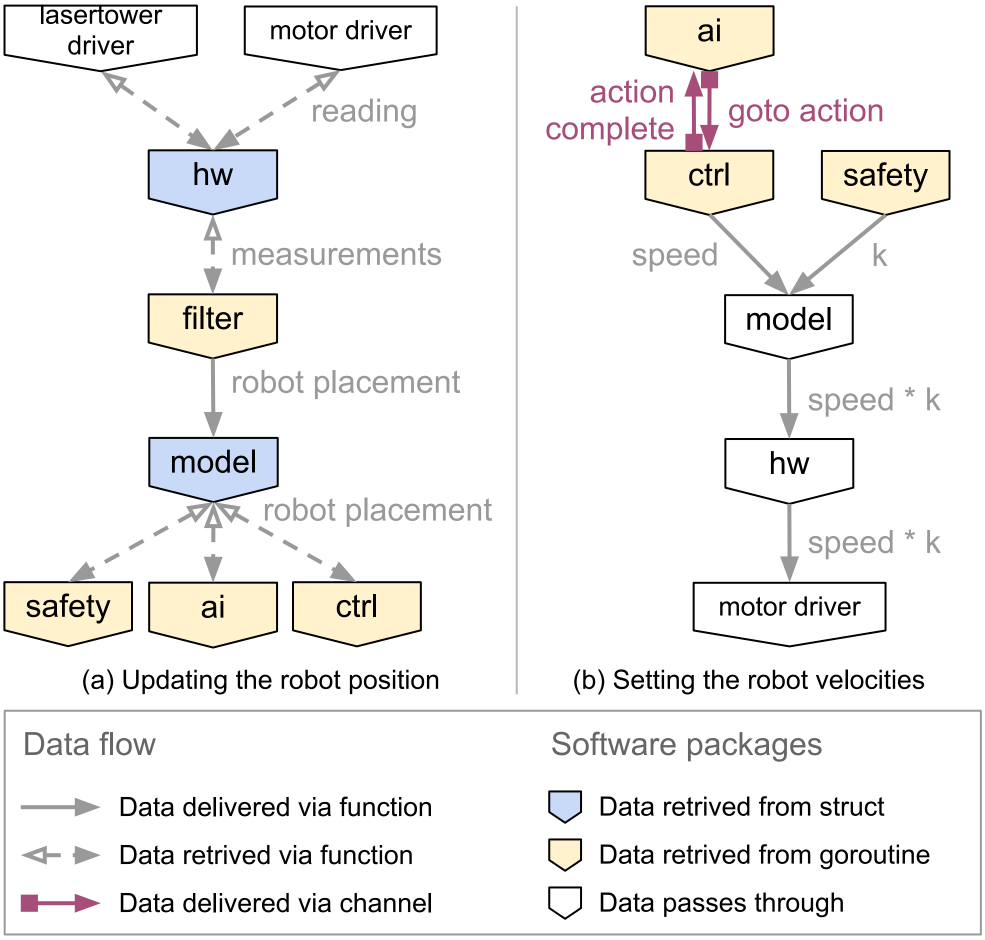


Figure 14.3: Data flow in the robot module

Figure 14.3 shows how data typically flow within the *robot* package. Figure 14.3b show how data flows when the motor velocity references are changed, but the data would flow through more or less the same packages if a servo position was to be set.

14.7.2 Robot positioning

The robot positioning code is implemented inside the *robot/hw*, *robot/filer* and *robot/model* packages. The robot/hw package defines a struct containing the measurements needed to define y in the Extended Kalman filter. The robot/model package defines a struct Robot, where the current robot state can be stored in a logical manner.

The robot/filter package defines an instance of the ExtKalmanFiler struct (Se section 14.5.3) that implements the system equations defined in chapter 12. A goroutine is defined to run the filter instance's Step function every 20th millisecond. The updated system states from this function are stored into instances of the Robot struct.

After each robot position update, a check is made on the time stamp of the opponent beacon readings. If a new reading has arrived, the position of the corresponding opponent is updated based on the beacon reading and the updated robot position. The calculation does not filter the beacon reading at all. Finally, the reading's time stamp is stored, so that the next iteration might do the same check.

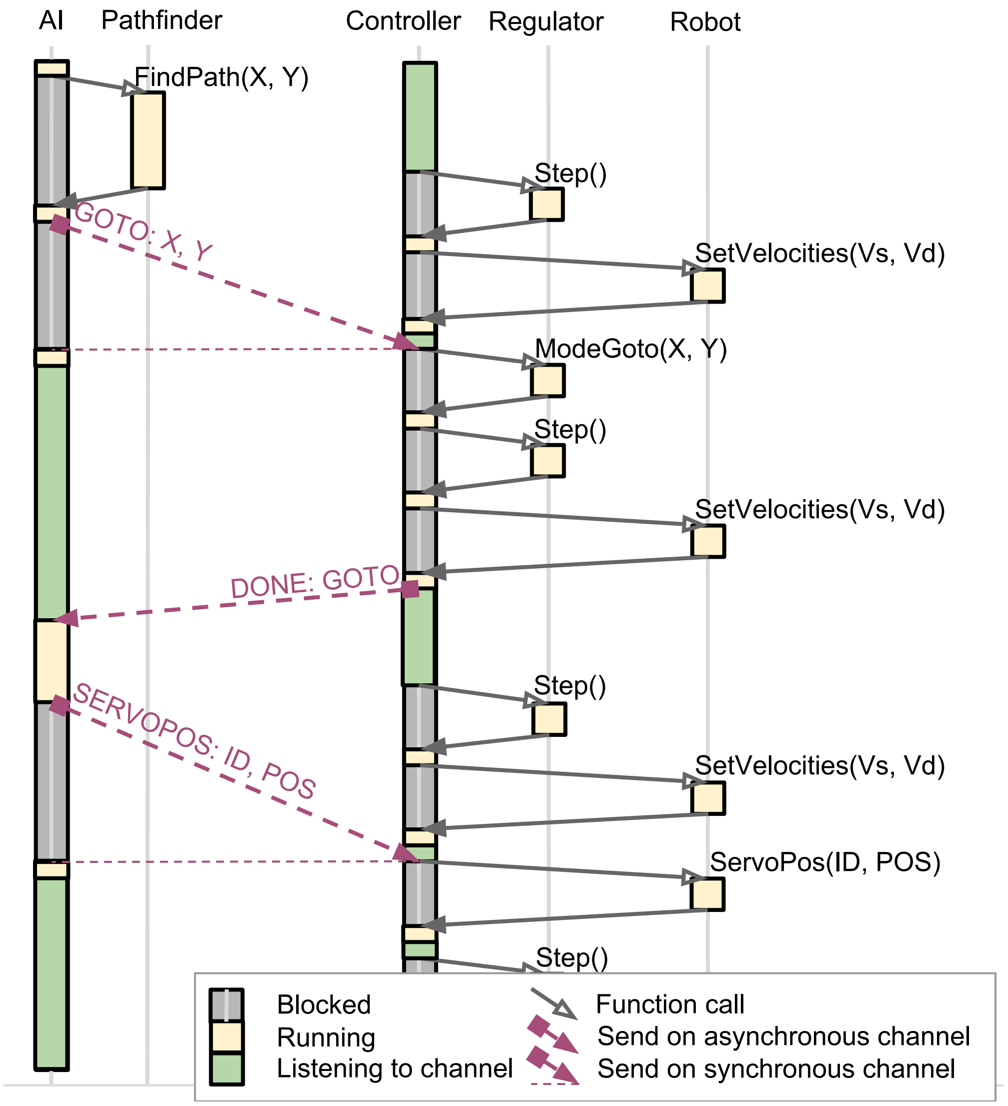


Figure 14.4: Communication diagram

14.7.3 Robot control

Figure 14.4 shows how the AI, the controller and the regulator will typically communicate. Note that since actions like GOTO and SERVOPOS are sent over a synchronous channel, the AI has to wait for the controller to be ready to receive before it can send on the channel. Since asynchronous channels are used to deliver the done message to the AI, the controller never have to wait for the AI.

Listing 14.12: Action types

```
const (
	// Non-regulator actions
	ACTION_RESET = iota
	ACTION_SERVOPOS
	ACTION_WAIT

	// Regulator actions
	ACTION_PAUSE
	ACTION_UNPAUSE
	ACTION_STOP
	ACTION_GOTO
	ACTION_GOTOX
	ACTION_GOTOY
	ACTION_FACE
	ACTION_ROTATE
)
```

Controller Actions

The file *robot/ctrl/actions.go*, define a single struct *Action*. An action can be used to tell the controller what to do. In listing 14.12, all possible action types are shown. For each of these action types, an initializer function is provided to encodes data into an Action struct.

Most of the initializers for the regulator actions would require an accuracy to be configured. E.g. the initializer for the GOTO action, takes X and Y coordinates, and an acceptance radius.

Controller

Listing 14.13: The Controller type

```
// Type declaration
type Controller struct {
	actionc chan *Action // Channel where actions should be received
	donec chan uint8 // Channel where done commands are sent
	robot *model.Robot // A Robot instance
	reg regulator // Regulator instance
}

// Member functions
func (c *Controller) GetActionChannel() (chan<- *Action)
func (c *Controller) GetDoneChannel() (<-chan uint8)
func (c *Controller) Run()
```

The controller has two important tasks. The first task is to listen for actions on the action channel. When an action arrives, it either passes the data onto the regulator instance, or commands the hardware through the *Robot* instance. The second task is to run the regulator's *Step* function with the current robot position and heading as parameters. The Step function returns reference velocities that are again used to update the motor references. How often the regulator should run can be configured. For the most part, a period of 25 microseconds has been used.

Under normal conditions, nothing prevents the controller from running the regulator's Step function at approximately the right times. However there are at least two conditions that can lead to either a live-lock or an unacceptable delay:

- Continuous spamming on the action channel might have an effect.
- If the messages are not removed from the done channel, the buffer will be filled. When this happens the controller goroutine can be blocked, which would prevent the regulator's Step function from being issued.

Regulator

Listing 14.14: The Regulator interface

```go
// The regulator interface
type regulator interface {
	ModeDisabled()
	ModeGoto(x, y, done_accuracy units.Distance, reverse bool)
	ModeRotate(angle, done_accurcay units.Angle)
	ModeFace(x, y units.Distance, done_accuracy units.Angle, reverse bool)
	ModeLineX(x units.Distance, heading units.Angle, done_accuracy units.Distance)
res/discussion.tex
	ModeLineY(y units.Distance, heading units.Angle, done_accuracy units.Distance)
	GetPeriod() time.Duration
	Step(x, y units.Distance, heading units.Angle) (movement units.Velocity, rotation units.
	    AngularVelocity, done bool)
}
```

The regulator was designed to be interchangeable and thus a Go interface has been defined (see listing 14.14). A series of different regulator modes were defined to handle the different possible actions defined in listing 14.12:

DISABLED: The robot should not move

GOTO: The robot should move to a specified coordinate

LINEX: The robot should drive against a line defined by $X = k$

LINEY: The robot should drive against a line defined by $Y = k$

FACE: The robot should rotate to face a specified coordinate

ROTATE: The robot should rotate to reach a specified heading

As decided in chapter 10, the regulator implementation used two separate PID regulators. The first regulator calculates a velocity reference based on an error distance value. The second regulator calculates an angular velocity reference based on an error angle value. After the velocity references have been calculated, a lookup table is used to calculate a *punishment* for each reference. How the errors and punishments are calculated, depend on the regulator's mode. For the GOTO mode, the velocity would be punished if the robot's heading is wrong and the angular velocity would be punished if the robot is very close to the goal.

The mode also defines how an action's accuracy is assessed. E.g. if the regulator is in GOTO mode, the regulator would report to the controller that it has completed when the robot's position is inside an acceptance radius around the destination point. In LINEX or LINEY mode, the regulator will assess the accuracy to specify the distance from the destination line.

Pathfinder

Task-specific code for wrapping the A* library, was defined inside the *robot/ai* package. A scaled pixel map of the playing area was provided. The robot was considered to be one pixel large, and then all obstacles on the playing area were padded to adjust for the robot's radius. Because the robot did not move only in straight lines, it was very hard to predict how close the robot would end up driving to these obstacles. To lower the risk of crashing into obstacles, some of the obstacles on the playing area were given additional padding.

Initially, the path returned by the pathfinder would include a number of points that had about 5*cm* spacing. It was discovered that the robot sometimes oscillated when the pathfinder was used. Most likely this happened due to the combination of measurement noise and an acceptance radius for the A* points that were configure to be too small. Increasing the acceptance radius would result in the robot being more likely to crash into obstacles on the playing area. To improve the situation, a smoothing algorithm was implemented. The final version of the pathfinder thus only returned one point at the end of each straight line.

14.7.4 Artificial intelligence

The Areas interface

Listing 14.15: The Areas interface

```go
type Area interface {
        Holds(p units.Coordinate) bool // is inside
        Mirror() Area
}
```

A simple interface to represent areas on the playing area was defined inside the *robot/ai* package, as shown in listing 14.15. The most essential function is the *Hold* function that takes a coordinate, and checks whether the given coordinate is inside or outside the area. As shown in figure 3.1 on page 10, the playing area is symmetric across the center line between the two totem poles. The *Mirror* function returns a copy of the area that is mirrored across this line.

The following areas have been implemented:

- Square
- Circle
- Arch (or pie)
- Multiplexed-area

The latest type allows the programmer to define complex shapes by combining several area instances.

The Task struct

A set of Action instances (see section 14.7.3) can be grouped into *tasks*. Each task has a start position, a list of actions, and some other parameters.

The *Mirror* function returns a copy of the task where all actions are mirrored across the center line of the playing area. Servo actions are mirrored across the center line of the robot. This way, tasks only need to be defined for *one* team. The tasks for the other team can be auto generated.

The AI interface

Listing 14.16: The AI interface

```go
type AI interface {
	dispatcher.Interface
	// Should return a slice of ctrl.Actions, representing the current plan.
	GetPlan() []*ctrl.Action
	// Should return a channel where actions can be suggested, or nil.
	GetActionChannel() (chan<- *ctrl.Action)
}
```

In chapter 10 it was defined that multiple AI implementations should be allowed. To allow an interchangeable AI, the Go interface shown in listing 14.16 was defined.

The *GetActionChannel* function of an AI implementation may return a channel where actions can be suggested. If the AI implementation is not written to support external suggestions, the function should return nil instead. The function is part of the interface to allow actions to be suggested via the Debug GUI.

The *GetPlan* function should return a list of what actions the AI implementation has planed to do next. The function is meant to provide the Debug GUI with information that can be presented to the user.

The AI interface extends the dispatcher Interface. This means that an implementation must provide a Run function, as explained in section 14.5.2. For an AI implementation to be useful, the

Run function should connect to the action and done channels of the global Controller instance. These channels were explained in section 14.7.3. The importance of keeping the done channel empty should not be forgotten.

An Implementation of the Run function should also periodically check the termc channel, and return immediately if something is retrieved. After a termination, the AI should support to be re-started. Upon a restart the Run function can optionally make a new assessment of which task to perform. However, none of the current implementations does that.

The Stupid AI

The Stupid AI was implemented to hold a single list of Actions. Whenever AI is notified by an action's completion, it sends the next action. This AI was used for testing, and for simulating an opponent.

The Forest AI

The Forest AI implement a platform for running decision trees, as the one defined in figure 4.1 on page 20. When defining a decision tree, a *task* is placed within a *tree node*. The same task definition can be reused in several tree nodes. Each tree node defines a *Next* function that returns the next tree node. A mechanism to mirror each task is given to avoid specifying the same task for both teams.

When the *Next* functions are defined, the programmer stands free to make a decision on what to do next. The decision can be based on many types of available information, especially whether the opponent is in a given area.

Typically a tree implementation would define a number of sectors on the playing area by using the *Area* types. To decide the next node, the Next function would typically check whether a larger area around where the task is to be performed, is free for opponents. Mechanisms are provided to make sure that the areas are mirrored to reflect the current robot team configuration. Multiple trees can and has been defined. It is possible to select what tree to run at start-up by providing command line options.

The Web AI

The Web AI was designed specifiable for the Debug GUI. It is the only current AI implementation that defined an action input channel. Generally, it listens for suggestions from the Debug GUI, and sends them over to the controller when they arrive.

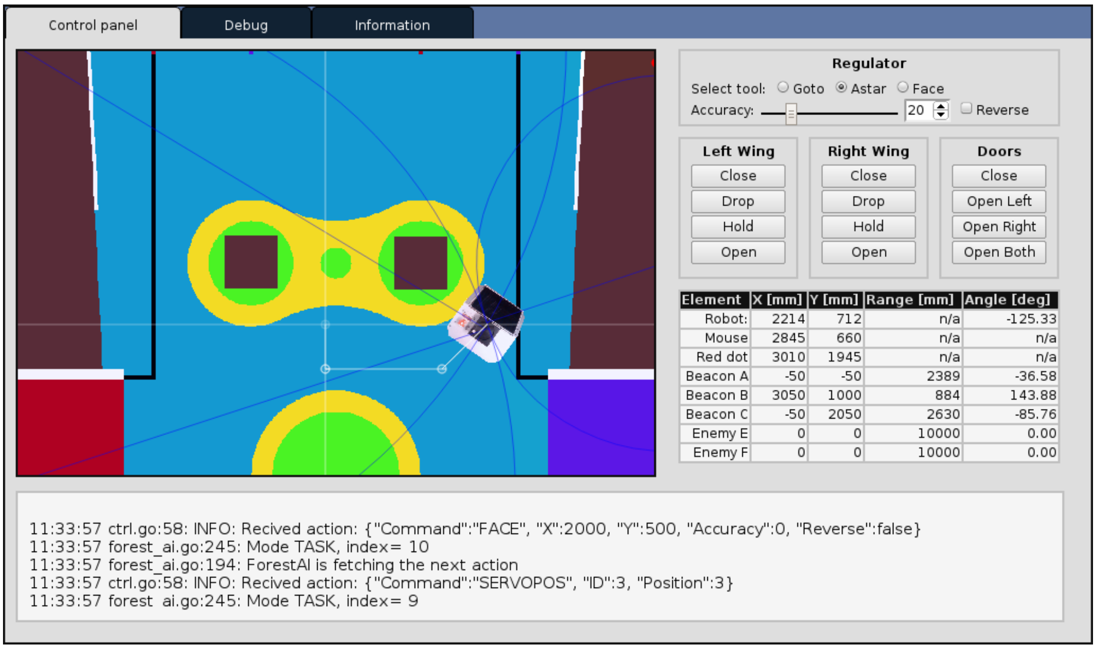
It is called "Web AI" because the Debug GUI is implemented as a web application.

14.7.5 Opponent avoidance system

The opponent avoidance system is implemented inside the *robot/safety* package. It provides it's own goroutine that periodically checks if the opponents are getting to close. The areas shown in figure 4.2 on page 21, is implemented by use of the *Area* types. The figure shows a stop area, and a slow-down area. If an opponent is detected within the slow-down area, a variable within the robot instance is updated based on the opponents proximity. If an opponent is detected within the stop area, the same variable is set to zero. In figure 14.3, it is shown that this variable is multiplied with the speed references before they are sent to the hardware interface.

It was mentioned in chapter 4 and 10 that a routine to get out of a dead-lock situation with an opponent, should be implemented if the time frame allowed it. Unfortunately, such a method has not been implemented.

14.8 The Debug GUI



Select tool: ○ Goto ● Astar ○ Face

Accuracy: 20 □ Reverse

Element	X [mm]	Y [mm]	Range [mm]	Angle [deg]
Robot:	2214	712	n/a	-125.33
Mouse	2845	660	n/a	n/a
Red dot	3010	1945	n/a	n/a
Beacon A	-50	-50	2389	-36.58
Beacon B	3050	1000	884	143.88
Beacon C	-50	2050	2630	-85.76
Enemy E	0	0	10000	0.00
Enemy F	0	0	10000	0.00

```
11:33:57 ctrl.go:58: INFO: Recived action: {"Command":"FACE", "X":2000, "Y":500, "Accuracy":0, "Reverse":false}
11:33:57 forest_ai.go:245: Mode TASK, index= 10
11:33:57 forest_ai.go:194: ForestAI is fetching the next action
11:33:57 ctrl.go:58: INFO: Recived action: {"Command":"SERVOPOS", "ID":3, "Position":3}
11:33:57 forest ai.go:245: Mode TASK, index= 9
```

Figure 14.5: The debugging GUI

14.8.1 Implementation

At the time that this software was written, the Go programming language did not provide any built-in GUI toolkit. However, the standard Go Library made it incredibly easy to program a web server [18]. God mechanisms for JSON encoding of Go structs, also made it very easy to create an API for exposing data to a javascript. A HTML5 canvas[23] was used to implement a virtual representation of the playing area. The code reside within the *robot/web* package.

The Debug GUI collects data from all Robot instances and from the AI. It was an important design decision, that no other robot packages, except for the main robot package, should import or use the *robot/web* package.

To avoid compromising the rest of the robot's stability, the Debug GUI was made fail-safe. This was done by providing a call to the *recover* function inside a defer function of the main goroutine, and by making sure that the web server did not start any additional goroutines. This way, if a panic occurred anywhere inside the web package, the rest of the robot's code should continue to run as if nothing happened.

14.8.2 Features

The following list describes the most essential functionality of the Debug GUI:

- Visualize last known position of all detectable robots
- Visualize the AI's plan of action on the playing area
- Provide a compact table with the most recent measurements and robot positions
- Provide a small window showing the latest log messages
- Allow controlling the robot servos (when the Web AI is used)
- Click-and-drive to any given location on the playing area (when the Web AI is used)
- Provide a debug tab for plotting data
- Allow remote access

In figure 14.5, we see that there are a set of buttons to the right. These control the robot servos when the Web AI is used. If the mouse is clicked on the playing area when the Web AI is used, the robot can either drive to the given position, used A* to find the best path, or face the clicked location, which one of these three tools to use, can be configured to the right. The accuracy of each operation, and whether to drive forwards or in reverse, can also be configured.

When the robot is started with another AI than Web AI, it is no longer possible to control the robot from the GUI. The current AI's plan of action can still be viewed. In figure 14.5, the white lines and circles show where the robot plans to drive. The circles indicate GOTO actions, while the vertical and horizontal lines that go across the entire field, indicate LINEX and LINEY actions (see section 14.7.3).

The blue lines and circles in the same figure, shows the measured angle and distance for each of the stationary beacons. Bellow the control panel to the right, the raw measurements are shown. The same table also shows other useful data like the robots current position and the opponent robots position.

14.9 Software testing

The development of the robot was partially test driven. Small test programs were used to test the device drivers. The Go package testing mechanisms were used to test most of the Go libraries. The Debug GUI was used for testing the complete system.

A global Makefile structure was provided to enable building the entire software stack, including most test programs. The *go test* tool provided mechanisms for running several package tests in one go.

14.9.1 Driver tests

Small test programs were developed for testing the motor, laser tower and power card drivers. They enabled the testing of the most essential driver functionality. In addition, the programs were used for testing and tuning of the physical devices.

The following driver test programs were developed:

Motor test

The test program for the motor driver was used to test the set velocity functions and the initialization/finalization routines. It only required that the motor card was connected to the CAN bus. When the test was used with the physical robot, robot movement was prevented by placing the robot upon a block.

Laser tower test

The laser tower test program tested the read function and the initialization/finalization routines. Any output from the read function would be printed to the terminal in a human readable format. The test required the laser tower to be connected to the CAN bus. In addition to testing, the program was used for tuning the positioning system, as described in chapter 12.5.

Power card test

The test program for the power card driver tested routines for initialization/deinitialization, timer manipulation, and servo control. It required the entire physical robot to be useful.

14.9.2 Go library tests

As mentioned in section 14.2, Go provides a mechanism for designing so called *package tests*. Within a package test, several unit tests[3] and benchmarks can be provided. Package tests were developed for the A*, Units and Dispatcher library. The Extended Kalman Filter was hard to validate with a standard Go package test structure, and thus provided a separate test program.

Units package test

A large amount of unit tests were developed for the Units package. Still, the test coverage[4] is not a 100%. It was however sufficient for validating most of the library's functionality. During development of the library itself, the unit tests were actively used to pin-point bugs.

Dispatcher package test

The Dispatcher package has test coverage of approximately 100%. Still, whether the package is really thread-safe, is not possible to test by the use of unit tests. For assuring this, a manual analysis of the source code was performed. In addition, a quick study of the specification and implementation around the Go channel and WaitGroup construct was performed.

A* package test

The A* package test has a single unit test for testing the map. Some of the unit tests that were provided in the original Gopathfinding library were removed due to a change of the library's interface. All in all, the package test is not sufficient to validate that the library works.

A benchmark was implemented in order to measure the optimizations as described in section 14.5.4.

Extended Kalman Filter test

The Extended Kalman Filter library could not be easily validated by a series unit test. A Go package test could therefore not be developed. Instead, a small test program containing an Extended Kalman Filter implementation was provided. To validate the library, the output of the program was evaluated manually. If the output showed that the system converged against a hard-coded reference value, the filter worked.

[3] Unit testing is a method for testing individual units of source code

[4] A measure of the proportion of a program exercised by a test suite

14.9.3 The Debug GUI

The debug GUI was used for testing the complete system, but was also useful for tuning and validating the functionality of individual components. Some examples of functionality that was validated or tuned through use of the Debug GUI are provided below:

A* library validation

As the test coverage for the A* library was relatively low, the final version of the A* library was validated to work by using the debug GUI in combination with the Go simulators.

Opponent avoidance system validation

The opponent avoidance system was initially validated and tested by use of the debug GUI in combination with the Go simulators. Later on, the system was tuned by using the real robot and a movable pole where the opponent beacon cold be attached.

Pathfinder tuning

As explained on page 110, the Pathfinder object *robot/ai* package wraps the A* library. It provided conversion between types defined in the units library and integer values, as well as the A* map. Padding of the A* map and modification of the acceptance radius of the returned way points where the provided options of Pathfinder tuning.

14.10 Results

Listing 14.17: Benchmarks of the original A* library, run #01

```
PASS
BenchmarkAstar4Dirs100x100 1 3071948000 ns/op
```

Listing 14.18: Benchmarks of the original A* library, run #02

```
PASS
BenchmarkAstar4Dirs100x100 1 3078446000 ns/op
```

Listing 14.19: Benchmarks of the optimized A* library, run #01

```
PASS
BenchmarkAstar4Dirs100x100 10 118449800 ns/op
BenchmarkAstar8Dirs100x100 20 119130000 ns/op
```

Listing 14.20: Benchmarks of the optimized A* library, run #02

```
PASS
BenchmarkAstar4Dirs100x100 10 117770800 ns/op
BenchmarkAstar8Dirs100x100 20 118333500 ns/op
```

14.10.1 Linux distrubution

Archlinux was chosen as the final Linux distribution to use. To get the CAN bus operational, the kernel had to be recompiled. The operating system has worked well on the HP Slate 2.

14.10.2 Go libraries

As described on section 14.5, a total of four general purpose Go libraries were developed. Much thanks to the acquired Gomatrix library, the Extended Kalman Filter library required less human recourses to implement than expected.

Optimization of the A* library

The A* library was based on code from the Gopathfinding library. An optimization of the original code was performed. Benchmarks results of the original library are shown in listing 14.17 and 14.18, while results for the optimized library is shown in listing 14.19 and 14.20. The calculated speedup becomes approximately 26x.

Validation and testing

Each library was tested and proved to work separately. For the A*, Dispatcher and Units libraries, the Go package test construct was used. While the automated tests for the Dispatcher library ensures that the library works it does not test whether it is thread-safe. A manual evaluation of the code suggests that the code is most likely thread-safe. For the Extended Kalman Filer, a separate test program was written.

The most complete test suites are the ones provided for the Dispatcher and Units libraries.

Are the requirements satisfied?

The implementation of A*, Units and the Extended Kalman Filter, should satisfy the requirements defined in chapter 10, and 12. The dispatcher library was not part of the original plan, and is thus an aversion form the design.

14.10.3 Device drivers

A total of three device drivers and two CAN libraries were developed. The CAN libraries were extracted from the existing Maxon EPOS2 motor drivers from 2011. These motor drivers were rewritten to use profile velocity mode rather than position mode.

Platform portability

The CAN API library *eurobot-ntnu/drivers/socketcan* provides a way to abstract away the operating system specifics of CAN communication. Even though the current version of the drivers only support the Linux platform, a Makefile layout was set up to enable compiling the drivers to several architectures.

Validation and testing

Initially, small test programs were used to test each driver individually. For parameter tuning, the robot software and the Debug GUI was used.

14.10.4 Hardware abstraction

A series of hardware abstraction layers were successfully developed. The drivers provide four layers: The CAN API library, the CAN Open library, the C driver and the Go wrapper. The task-specific code provides some additional layers, though the most interesting one, is the bottom one.

The hardware interface

A Hardware interface was defined by use of the Go interface type inside the *robot/hw* package. This interface delineates how the rest of the *robot* code can interfere with the hardware, and enables multiple interchangeable hardware implementations.

Simulators

Two Go simulators were developed by implementing the Hardware interface. They were used for testing software within the task-specific code, and for developing game tactics. In addition, a stand-alone C simulator was developed. This simulator was used for testing the entire software stack, without using the physical robot. All in all, the Go simulators were used much more often than the C simulator.

14.10.5 Debug GUI

The Debug GUI was developed as a Go web server hosting a single web page. The GUI was made fail-safe so that a potential crash should not affect the rest of the robot code. It presented a visual representation of the robot software's world model, and allowed controlling the robot. Because the Debug GUI was developed as a web server, it could be accessed from a remote PC.

All in all, the Debug GUI proved very useful for software debugging and tuning. It also allowed rapid definition of game tactics and strategies.

14.10.6 The robot's main program

The complete robot software was proven to work well, and to be easy to adapt and configure. The software contain of several goroutines[5]. The dispatcher library is used to assure safe shut-down of important goroutines. E.g. the hardware poller routine should always turn of the motor and lasertower before it returns. Data flows have been designed to avoid locking goroutines that have deadlines and to limit the chance of live-locks.

Challenges with the A* algorithm

The combined usage of the A* algorithm and the regulator implementation presented some challenges. Since the robot could move in curves, it was hard to predict how close the robot would drive to objects on the playing area. Some objects therefore needed to be heavily padded when the A* map of the playing area was defined. Whether the robot would run into objects on the playing area or not, varied based on the robot's initial position etc.

[5] Light-weight threads

Aversion from the original design

The implemented robot code mostly fulfills the requirements defined in chapter 10, but some extra sub-packages have been added and a slight reorganization of software packages has been made. In chapter 4 and 10, it was suggested that a way to get out of a dead-lock position with an opponent blocking the robot's way, could be implemented if the time frame allowed it. No such routine was developed. The main reason for not implementing this feature was that the time frame did not allow the task to be implemented properly without lowering the priority of other tasks.

14.10.7 The robot software as a whole

Most of this year's source code has been written form scratch, and has been proven to work well. To accurately illustrate the project size, a detailed count of code lines can be found in appendix D.3.

The software solution is quite robust and well tested. Currently there are no known critical bugs.

14.11 Discussion

Criticism of strategic choices

The combination of an A* algorithm, a robot that can move in curves and a playing area with many obstacles, might not have been the best of combinations. Since the robot was driving in curves, the robot didn't necessarily follow the A* path very accurately. This again made the A* algorithm risky to use.

A* optimization

The optimization of the A* library was quite good, and hopefully the optimized library can be of use to other. However, numerous of A* implementations with good performance probably already exist in other languages. In the game strategic implementations that was used in the actual competition, A* was almost not used at all. In retrospect, it is easy to conclude that the human resources used on A* optimization could have been put to better use.

Opponent avoidance

As mentioned in the results, no routine to avoid a dead-lock when an opponent was blocking our way was implemented. Some of the constructs for implementing this routine were however already in place. For instance, the fact that the AI could be safely killed and restarted could have been used by the opponent avoidance goroutine to tell the AI to make a new choice. The main AI could easily be coded to reevaluate its last decision upon restart. The main challenge was to write a way for the opponent avoidance goroutine to back out of the dead-lock position in a way that would always be safe.

Too many simulators

Simulators were developed in both C and Go. One can ask if this was really necessary.

The Go simulators were very useful for developing Go software components, especially as they allowed simulating the opponent. They can thus be justified.

The stand-alone C simulator could be used for testing the entire software stack. In the end, it wasn't used very much. Thus with Eurobot 2012 in mind, it was probably not necessary to develop, However, for future years students, a stand-alone simulator might prove more useful than one that is tightly integrated with the robot software.

Testing facilities

The amount of time spent on developing unit tests, the two simulators and the Debug GUI, seems to have paid off. Having the vast amount of testing facilities available that this year's solution provides has made debugging become fun and easy. With less time spent on debugging, more time could be spent on design and development.

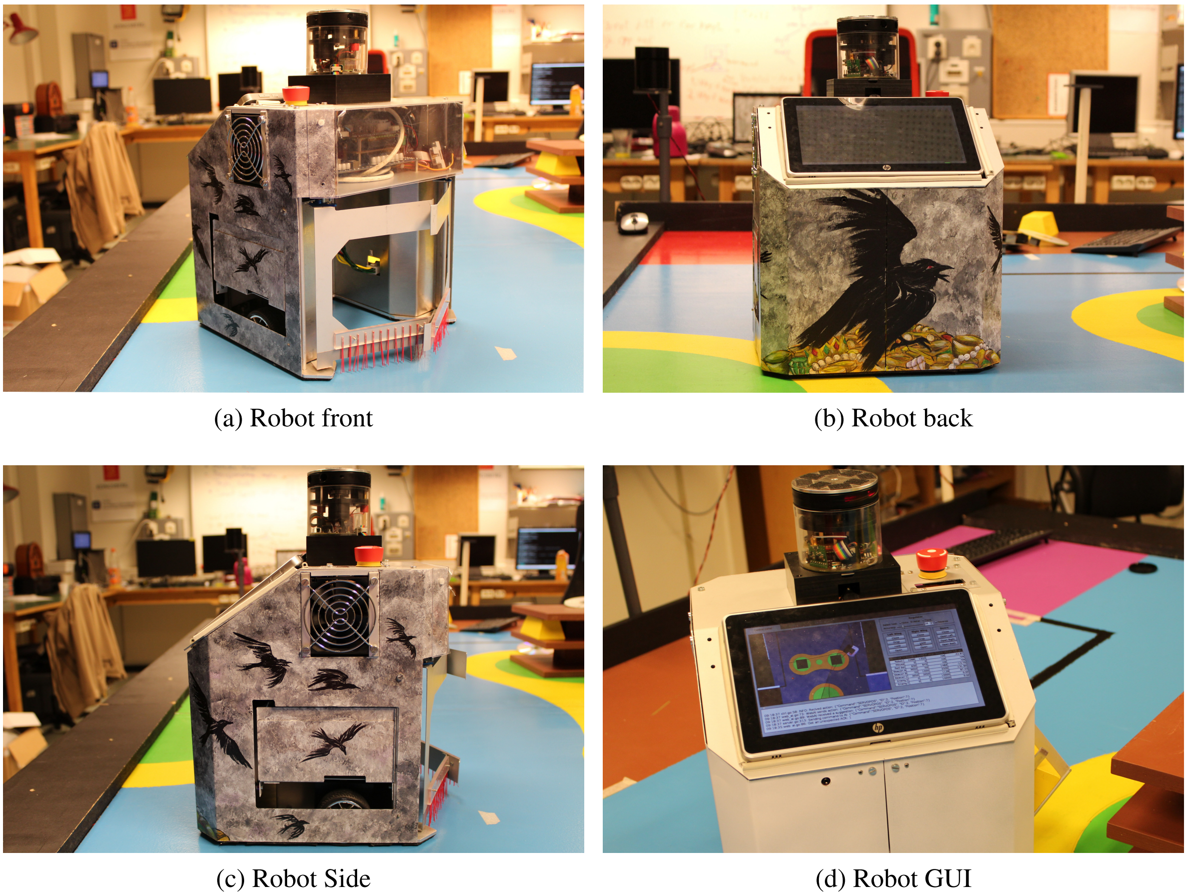
Still, a well functioning and tested software stack is not enough to win Eurobot. The software does however provide a quite decent platform for performing these calibrations.

Part IV

End Result

Chapter 15

Results



(a) Robot front

(b) Robot back

(c) Robot Side

(d) Robot GUI

Figure 15.1: The finished robot

This chapter will describe the result found when combining the findings from the previous chapters. The chapter will try to both describe how well the complete robot functioned and what parts of it that are of most value.

15.1 The complete robot

A functioning robot was built and tested. The mechanical system as a whole worked quite well, and its simple layout and standardized screws made it a joy to work on. The finished robot can be seen in figure 15.1 and a video is digitally attached, se appendix G.

15.2 Competition

Three game tactics were developed for the final competition in France:

Homologation: A simple tactic to score some very easy points and then stop.

Passive: A tactic that allowed the robot to harvest safe points first, and then do more risky tasks later on.

Offensive: A tactic that started with risky tasks that potentially allowed scoring a lot of points.

A conceptual illustration of the plan for the original game tactics is available in figure 4.1. Game tactics were reprogrammed and tuned before each match, so the *exact* implementation shown in the figures was never used. The offensive tactic was never used at all. Information about each match and the exact tactics used, can be found in appendix E.

In the end the robot finished in 23. place [45]. There was a total of 43 teams in the international competition, where 39 qualified. The robot won 3 of 5 matches, and it always started. However, in the last match the emergency stop had to be pressed.

15.3 Position and location

The extended Kalman filter, discussed in chapter 12, worked well on its own. Together with the regulator described in chapter 14.7 they formed the control loop described in figure 16.1. This complete system was tested by running the robot on the playing area in the lab. The final and most important test came when running it during the competitions in France.

During the first tests in the lab the robot moved towards the goals slowly. It worked just fine and reached each goal, but moved with a jerk. Later when the motor-controller acceleration was tweaked, as discussed in chapter 16.2, it ran quite smoothly.

When approaching a target position the angular error, calculated in the regulator, hit a singularity resulting in the robot violently turning close to the target. This artifact was somewhat compensated for by punishing the rotation-regulator close to the target.

The positioning filter was tested for temporary outtakes of beacon towers. It handled the loss of beacon B for an extended period of time. On the other hand, due to the weighting matrix R, the loss of tower A or C can become critical within seconds.

In France the positioning worked fine at least four out of five matches. In the final match the robot crashed into a totem pole. In the second match it cough a totem-pole gold-bar by executing, a task with the current robot is estimated to require about $1cm$ accuracy.

15.4 Power card

A circuit board with several power supplies has been created and tested. It provides many different voltages, a micro controller with digital and analog I/O and a system for expansion of functionality via a stack following the PC/104 plus mechanical specification.

The $3.3V$-, $5.0V$- and $12.0V$-volt regulators works, they have efficiencies of respectively 80%, 90% and 95%. They have been tested to have stable output voltages when supplied with input voltages between $16.0V$ and $22.8V$. The $24.0V$ regulator, battery charger and automatic switching between power sources do not work. The output voltage of the $24.0V$ regulator is slightly lower than the input voltage. The circuitry to switch automatically between power sources for the circuit board is designed wrong and will not work. The battery charger does not charge the battery when an appropriate external DC voltage is connected.

Figure 15.2 shows a picture of the power card mounted in the robot. All wires, except the ones to the battery and the emergency stop button, is connected to expansion cards in the circuit board stack. This shows that the circuit board stack allows expansion of functionality without extra wires. Further each motor/servo/fan is connected to the stack with one labeled and directional connector.

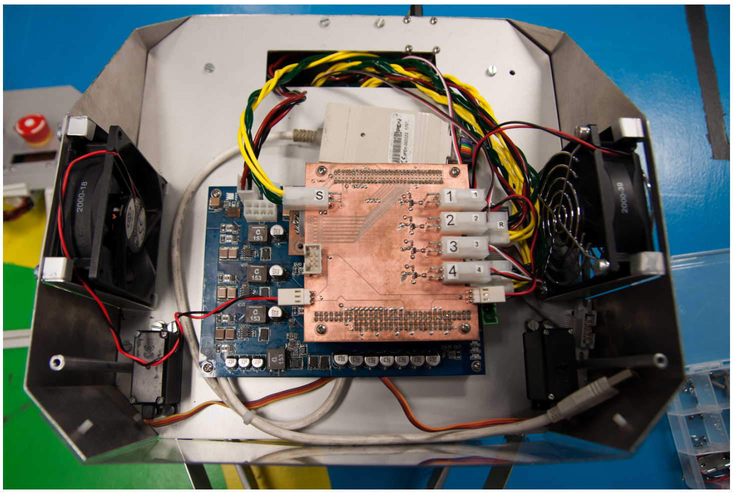


Figure 15.2: Power card mounted in the robot

15.5 Software

A software stack for the robot was written by using the Go and C programming languages. The result for individual software component can be found in chapter 14.10.

The implementation was found to be in compliance with the requirements defined in chapter 14. Some functionality that wasn't part of the requirements were implemented as well. Multiple AI implementations and simulator configurations were provided. The Debug-GUI enabled interactive control of the robot during testing. The software could be configured by specifying different parameters at startup.

The software development process have been partially test driven, and different tools and mechanisms have been used for testing. For the device drivers, small test programs for testing essential functionality was provided. Most of the implemented Go libraries provide so called *package tests* that was used to validate that the libraries worked. A 100% test coverage for these libraries were not provided. Packages inside the main robot program did not provide package tests, but the debug-GUI and simulator implementations could be used for manual validation.

The software worked in all matches during the competition in France. Even in the field, it allowed for calibrations to be performed and for game tactics to be re-programed.

Chapter 16

Discussion

16.1 Mechanical problems

Mechanical errors

The side “wings” required the robot to position itself extremely close to the totem pole and do a drive by. This was known in advance. However if the robot positioned itself slightly to far away it would push the CDs inside the totem pole inwards jamming them. This was not anticipated. Since the doors are not flexible, and the body is especially not flexible, this ended in the CD breaking or the robot spinning and crashing. A video of this phenomenon can be found in online in appendix E.

While the mechanical layout was excellent to work on, the servo-mechanics to manipulate playing element could have been better. The *drive by totem* action, required to use the wings, needs a lot of playing area compared other solutions. This takes time and increase the risk of a conflict with the opponent robot possibly resulting in a deadlock. For example, a solution that drives straight up to the totem-pole and uses servos to manipulate the CDs and gold bars from there could be better.

Design alternatives

The rules allow each teams two robots, a small and a big one. A possible hack for a small team would be to build two small identical robots. If one robot breaks, one would have one in reserve. If two robot breaks, one would have reserve parts. If one robot deadlocks with the other robot during a match, the second robot can gain points, and so on.

16.2 Positioning issues

Design problems found

The main problem with the current position system is the lack of encoders. If the wheels slip, the position estimate will follow suit. Under normal operation this isn't much of an issue, but it will happen for instance during a collision. In addition, due to the material if the robots arms touch anything solid they won't give resulting in the robot spinning.

During fine maneuvering, that is anything sub ±2.5cm, a loss of a positioning beacon was found to be critical, with the result of missing the target. To avoid these problems under fine maneuvering the robot would wait two seconds to let the position system catch up. A better way to fix this problem would be to use more mechanical guided picking-mechanisms that tolerate small positioning errors.

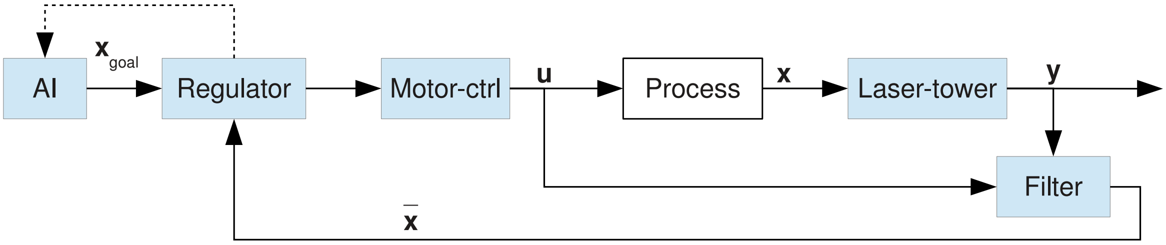


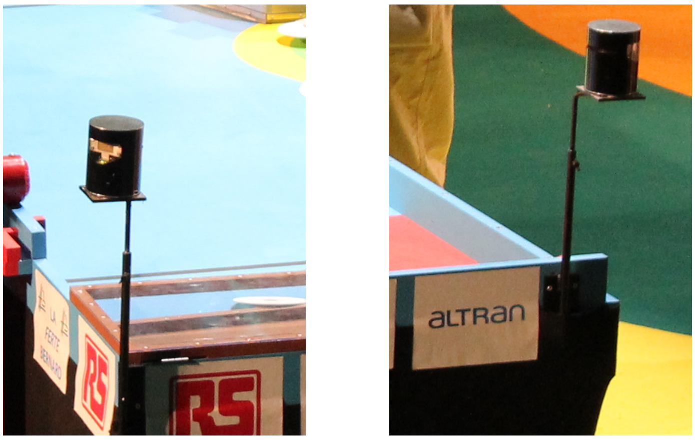
Figure 16.1: Robot control loop

Figure 16.1 shows that a change from the regulator is immediately reflected by the motors, except for the delay presented in the motor-controller. The motor-controller delay can be adjusted by tweaking the *Motor_max_acceleration* parameter in the driver from chapter 14.6. When the robot was configured to have higher acceleration and top speed, the robot ran much quicker and smoother. However, due to the high currents and the big impact of a possible error, the parameters were left at a rather conservative level.

Problems outside the lab

The beacon supports used in the actual competition were found to differ slightly from the specification, and from table to table. Figure 16.2 shows how the supports are mounted differently on each side of the table, resulting in a slight offset. One can also see that the supports are adjustable, even though they should remain fixed at 35*cm* [52, p. 24].

The use of beacon towers as the main positioning method, limits usage to playing areas only. Whilst a robot using only relative-positioning can be setup and run anywhere. In addition, during the competition each beacon is powered by a 9V battery, which must be regularly changed. These small problems make running the robot outside of the lab a hassle.



(a) Corner bracket (b) Straight bracket

Figure 16.2: Beacon supports used in the competition

Alternative systems

The regulator used, demanding an always up to date perfect position estimate, left no room for errors in the Kalman filter. An alternative would be to turn in the right direction, stop, drive in a straight line then stop, rotate and repeat. This seemingly slower system allows the robot to drive very fast in straight lines, with a much less sophisticated positioning system. Chapter 5.2 mentioned that this method was used in 2011, but concluded that it was too slow. This conclusion might have been wrong.

16.3 Power card

Value of results

From the data sheet of the TPS5450 [57], which is used in the $3.3V$, $5.0V$ and $12.0V$ regulator design, it can be seen that the expected efficiency of a design is close to 90%. It can also be seen from the example design in the data sheet that the efficiency of a regulator is higher when the difference between the input and the output voltage is small. The efficiency of the working regulators follows these expectations. If the efficiency is compared to linear regulators, which has been used by previous Eurobot teams from NTNU, this is a great improvement.

Even though the efficiency of the $3.3V$ and $5.0V$ is lower than 90% it is still high compared to a linear regulator. An example where $3.3V$ is needed with an input voltage of $22.8V$ reveals an efficiency of:

$$\eta = \frac{P_{out}}{P_{in}} = \frac{3.3V \cdot I}{22.8V \cdot I} = 15\% \tag{16.1}$$

Design errors

The output voltage of the 24.0V regulator follows the input voltage. The regulator it self does not work, however it was used in the competition. The only components in the robot that used this voltage are the motor controllers which has a working range from 9V to 24V. As discussed in chapter 13.2 the battery package used in the robot has a stable output voltage between 90% to 10% capacity. This is the reason the power card worked in this particular application, if the power card is reused in another application which requires 24.0V the regulator has to be fixed.

The power card was also designed with a battery charger and the possibility of running the robot while charging using automatic switching between energy sources like a laptop computer. It was discovered that the chip chosen to do the switching between sources does not support the mode of operation it was designed to use on the power card. The battery charger did not work during the first few tests either, since the automatic power switching didn't work sufficient testing was not done on the battery charger as the power card could work for this application without charging the battery.

Using the power card

Even though the power card provides efficient regulation of power, the greatest advantage is gained in form of simple usage and modularity. As mentioned in 13.4 both expansion cards for motor control and servo control went through a couple of prototypes. These prototypes could be made without having to design and implement voltage regulators and could be fitted directly on the robot without any extra wires because of the circuit board stack on the power card. Since each servo/motor/fan is connected with a single directional and labeled plug it is relatively fast to assemble/disassemble the electronics without any special tools. This is a good quality, especially for debugging and repairs in a competition situation where time is a factor. Because of the circuit board stack and fewer wires this solution also uses less volume in the robot.

16.4 Using Go

Go provided quite handy concurrency structures that where used in this year's project. The support for synchronous and asynchronous message passing and the support for light-weight threads have proven very useful. Section 14.9 and 14.10 states that Go's facilities for package testing has resulted in a robust software solution. The distinction between critical and non-critical errors in Go have probably also contributed to the software's robustness.

Chapter 14 states that many software modules was developed from scratch. As Go was used for a Eurobot-NTNU robot for the first time, very few modules from previous years could be reused. As the language is also quite young, it was hard to find libraries even for common algorithms. Thus we had to implement our own libraries for A* and extended Kalman.

During the first months of software developments, the Go standard libraries were undergoing major changes in preparation of the Go version 1 release. One month before the competition, version 1 was released [19]. With Go version 1 released, Go provides backwards guarantees compatibility.

With all this extra work one can discuss whether Go was really the right choice. Possibly, more time could have been used on game tactics and other tasks if the existing C++ framework was chosen over Go. However, there was a new hardware stack this year so few of the previous years' drivers could have been reused.

Chapter 15 cites that the new software has worked well in all the robot's matches. Considering that Go has been successfully tested for usage in an autonomous robot, and that the software has proven to work well, the decision to use Go should be well justified.

16.5 Distribution of human resources

Much time was used to develop a robust solution with many advanced features. Much less time was spent at developing the actual game tactics. During the competition week in France, it was discovered that many other teams could get away with much more simplistic robot designs. Robots driving only on encoder and color sensors, crashing into walls every now and then to regain orientation, et cetera. One could thus start to wonder whether our robot has been over-engineered.

With a more simplistic design, more time could have been spent on developing game tactics. In this way it is possible that a better placement in the competition could have been achieved. On the other hand, it could also have resulted in a dull robot, less suitable for a master thesis.

Chapter 17

Conclusion

A robot have been designed and built. It has participated in the Eurobot 2012 competition, winning 3 out of 5 matches. The robot is easily reconfigurable and re-programmable, and can be run in both autonomous and interactive modes.

Its mechanical design is simple, but thoroughly thought through, making it easy to maintain and quickly assemble from parts. On the other hand the servo-mechanisms for grabbing and placing playing-element could more efficient.

A robust power card with a circuit board stack for expansion of functionality has been produced. The power card provides the expansion modules with several voltages created from efficient switch mode power supplies and communication through CAN bus. The power card with expansion modules was used and worked well in the robot. The biggest advantage with the stack is that it has as few wires as possible with polarized and labeled connectors. This makes the solution easy to maintain and fast to reassemble in a competition environment. The new solution has the same modularity as the rack system from 2009, but has fewer wires and takes up less volume in the robot.

The electronic stack connects to a *HP slate 2* tablet PC through a CANopen-bus. The tablet is detachable, making for trivial parallelization of software and hardware updates. During operation the tablet displays real-time information. The real-time information includes both the robot and the opponent robots position on the map, all sensor measurements and AI-commands to execute.

The relatively new programming language *Go* has been used for the first time as the main language on a Eurobot-NTNU robot. To achieve this a lot of software modules have been written from scratch or ported to Go. The new software stack has proven to be both robust and field-reprogrammable. Constructs in the Go language made the software easy to design and test. Continued use of Go in Eurobot can thus be recommended.

An *extended Kalman filter* has been designed to transform the laser-tower and hall-sensor measurements into a position and orientation estimate. The estimates produced are persistent, repeatable and with an accuracy comparable to the inherent variance in placing the fixed beacons.

The filter was used, together with two PID-regulators, to position the robot during the Eurobot 2012 matches.

In conclusion, the robot as a whole works. The fact that it ran consistently during the matches in France should be enough proof of that. It is extremely agile and dynamic. At a push on touch-screen map, it will calculate a clear path and follow it to its end.

Chapter 18

Future work

18.1 Power card

There are some faults on the power card:

- The 24.0V regulator does not work.
- The battery charger does not work.
- Automatic switching between battery power and external DC does not work.

The 24.0V regulator and the battery charger has not been debugged or tested thoroughly and it may be possible to debug fix them on the existing circuit board. The automatic switching between power sources is designed wrong and it can't be fixed with the parts in the design.

If it's not possible to fix the faults with the existing PCB and a new one is going to be designed, it is recommended to:

- Implement the 24.0V regulator as a boost regulator for a simpler design.
- Design a new battery charger / power path control.

If a more compact design is desired a smaller power supply module, with the same output voltages, can be designed on a PC/104 size circuit board with only the ISA connector to fit on the bottom of the stack. A second PC/104 plus card can then be designed with a micro controller with digital/analog I/O.

18.2 Position system

The input to the extended Kalman filter should be changed from the motors hall-sensors to sensors disconnected from the drive wheel. These sensors could be encoders or optical/lasers. A study of the performance before and after this change should be made.

The beacon towers should get longer battery life and proper power management. The new opponent beacons should get a custom made mirror calibrated for its use.

A better regulator for robot movement should be constructed. The new regulator should be capable of easily defining and following paths with minimal error and maximum speed, possibly using a GUI or recording-function to create paths.

In the event of a non-flat playing area the current hardware would be unusable, as the laser-beams would not hit anything. It is therefore recommended to base any completely new position systems on radio rather than laser or ultrasound. A radio system would be resistant to most kinds of disturbances and conceptual equal to GPS.

18.3 Software

The device drivers can be ported to run on other architectures e.g. AVR or Windows. This can be done by adding more backends to *src/eurobot-ntnu.no/drivers/socketcan*.

Work towards expanding the test coverage of the Go libraries should be considered. Developing towards the open source spin-of projects listed in appendix D.1 is recommended.

In general it is advised that software inside the *robot* package is used only as a template for developing new code. If a feature of the robot package is considered highly reusable, the source code should be extracted to forge a new library.

Bibliography

[1] A123 Systems. *ANR26650M1*. http://www.gylling.no/produkter/batterier/a123/ANR26650M1_Datasheet_MARCH_2008.pdf. 2012.

[2] Magnus Andersen. "Ultralydbassert posisjoneringssystem." In: (2002).

[3] Arch Linux Community. *Kernels/Compilation/Arch Build System*. https://wiki.archlinux.org/index.php/Kernels/Compilation/Arch_Build_System. 2012.

[4] John Asmuth. *Gomatrix Github Page*. https://github.com/skelterjohn/go.matrix. 2012.

[5] Robert Grover Brown and Patrick Y.C. Hwang. *Introduction to random signals and applied Kalman filtering - Third edition*. 1997.

[6] Dewald De Bruyn. "Power Distribution and Conditioning for a Small Student Satellite." In: (2011).

[7] Robert Burtch. *CIRCLE-CIRCLE INTERSECTION*. www.ferris.edu/faculty/burtchr/sure215/notes/circle-circle.pdf. 2004.

[8] Stack Overflow Community. *Stack Overflow - Use C++ in Go*. http://stackoverflow.com/questions/1713214/how-to-use-c-in-go. 2011.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms - Second edition*. 2001.

[10] Microsoft Corporation. *Windows 7 system requirements*. http://windows.microsoft.com/en-us/windows7/products/system-requirements. 2012.

[11] DistroWatch. *Distrowatch*. http://www.distrowatch.com/. 2012.

[12] Arnaud Doucet and Adam M. Johansen. "A Tutorial on Particle Filtering and Smoothing: Fifteen years later." In: (2008).

[13] Hans-Christian Egtvedt. "Oppdatering av maskinvare og operativsystem til Eurobot." In: (2003).

[14] Hans-Christian Egtvedt. "Eurobot 2004." In: (2004).

[15] *Eurobot, international robotics contest*. http://www.eurobot.org/eng/. 2012.

[16] Sondre Garsjø and Halvor Platou. "Eurobot 2006." In: (2006).

[17] Inc. Gentoo Foundation. *Gentoo - About*. http://www.gentoo.org/main/en/about.xml. 2012.

[18] Google Inc. *The Go Programming Language*. http://www.golang.org. 2012.

[19] Google Inc. *The Go programming Language Blog - Go version 1 released*. http://blog.golang.org/2012/03/go-version-1-is-released.html. 2012.

[20] Gylling Teknikk. *Battery Protection Modules*. http://www.gylling.no/produkter/batterier/protection-module.shtml. 2012.

[21] Kristin Holst Haaland. "Eurobot 2010, Navigasjonssystem." In: (2010).

[22] Are Halvorsen. "Robot power managment." In: (2011).

[23] Ian Hickson. *HTML Canvas 2D Context, W3C Working Draft 29*. http://www.w3.org/TR/2012/WD-2dcontext-20120329/. 2012.

[24] IBM and SGI. *Linux Test Project*. http://ltp.sourceforge.net/. 2011.

[25] Sean Ogden John Asmuth. *Gomatrix Google Code Page*. http://code.google.com/p/gomatrix/. 2012.

[26] Steffen Johnsen, Kristian Klausen, Adam Leon Kleppe, Lars Espen Nordhus, John Magne Røe, and Leif Julian Øvrelid. "Fagraport Eurobot - Eksperter I Team: Byggelandsbyen." In: (2012).

[27] Gunnar Kjemphol. "Eurobot 2007." In: (2007).

[28] Christian Wegger Kjølseth and Øysten Wergeland. "Eurobot 2009." In: (2009).

[29] Christian Wegger Kjølseth and Øysten Wergeland. "Prosjektoppgave - Eurobot 2009." In: (2009).

[30] Kristian M. Knausgård and Gunnar Kjemphol. "Prosjektoppgave - Eurobot 2007." In: (2006).

[31] Ole Lillevik. "Eurobot 2010, Maskinvare." In: (2010).

[32] Maxon motor AG. *Maxon - EPOS2 24/2 Hardware Reference - Document ID: rel1702*. 2010.

[33] Maxon motor AG. *Maxon - EC 45 flat Ø45 mm, brushless, 50 Watt - 251601*. 2011.

[34] Jonas Aamodt Moræus. "Nonlinear Modeling, Identification and Control of an Underactuated Mobile Robot." In: (2004).

[35] Jonas Aamodt Moræus and Torstein Valvik. "Eurobot 2005." In: (2005).

[36] Sindre Røkenes Myren. "RT capabilities of Google Go." In: (2011).

[37] Odd Eirik Mørkrid and Kristian Ivar Øien. "Eurobot 2008." In: (2008).

[38] Bjørn-Gunvar Nessjøen. "Eurobot 2011." In: (2011).

[39] Tor Onsus. *Instrumenterings-systemer*. NTNU, 2011.

[40] Hewlett Packard. *QuickSpecs HP Slate 2 Tablet - Version 8*. 2012.

[41] PC/104 Embedded Consortium. *EPIC and EPIC Express v3.0*. 2008.

[42] PC/104 Embedded Consortium. *PC/104-Plus Specification v2.3*. 2008.

[43] *PCBCART*. http://www.pcbcart.com/. 2012.

[44] Peak System. *Peak System Home Page*. 2012.

[45] Planet Science. *Coupe de France - Robotique*. `http://www.planete-sciences.org/robot/live/coupe2012/eurobot/`. 2012.

[46] Alex Plugaru. *Gopathfinding Github Page*. `https://github.com/humanfromearth/gopathfinding`. 2011.

[47] *Readme file for the Controller Area Network Protocol Family (aka Socket CAN)*. 2005.

[48] Everett Rogers. *Understanding Boost Power Stages in Switchmode Power Supplies*. `http://www.ti.com/lit/an/slva061/slva061.pdf`. 1999.

[49] Everett Rogers. *Understanding Buck-Boost Power Stages in Switchmode Power Supplies*. `www.ti.com/lit/an/slva059a/slva059a.pdf`. 1999.

[50] Everett Rogers. *Understanding Buck Power Stages in Switchmode Power Supplies*. `http://www.ti.com/lit/an/slva057/slva057.pdf`. 1999.

[51] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[52] Planete Science. *Eurobot 2012 rules*. `http://www.swisseurobot.ch/images/2012/e2012_rules_en_final.pdf`. 2011.

[53] Chester Simpson. *Linear and Switching Voltage Regulator Fundamentals part 1*. `http://www.national.com/assets/en/appnotes/f4.pdf`. 2007.

[54] Andreas Hopland Sperre. “Eurobot 2012 - Propulsion and location.” In: (2011).

[55] Andreas Hopland Sperre, Are Halvorsen, Kim Værner Soldal, Geir Josten Lien, Marin Martinsen, and Øystein H. Holjelm. “Eurobot EiT report 2011.” In: (2011).

[56] Stian Juul Søvik. “Eurobot 2010, Fremdriftssystem.” In: (2010).

[57] Texas Instruments. *TPS5450 - 5-A, Wide Input Range, Step-Down SWIFT Converter*. 2007.

[58] Åsmund Torja. “Navigation for selvgående kjøretøy.” In: (1998).

[59] Torstein Valvik. “Navigationssytem for Eurobot 2005.” In: (2004).

[60] Bjørnar Vik. *Integrated Satellite and Inertial Navigation Systems*. 2011.

[61] Judd Vinet and Aaron Griffin. *Archlinux - About*. `http://www.archlinux.org/about/`. 2012.

[62] Wind River. *Wind River’s VxWorks*. `http://www.windriver.com/products/vxworks/`. 2012.

[63] Vincent Zalzal. *KFilter Sourceforege Page*. `http://kalman.sourceforge.net/`. 2008.

Part V

Appendix

Appendix A

Positioning system

A.1 Laser-tower accuracy

Real distance [mm]	Measured distance [mm]
150	170
200	220
250	270
300	320
350	372
400	422
450	475
500	528
550	580
600	630
650	685
700	740
750	790
800	842
850	893
900	942
950	1002
1000	1055
1050	1105
1100	1160
1150	1203
1200	1270
1250	1320
1300	1380
1350	1420
1400	1490
1450	1535
1500	1598
1550	1645
1600	1700
1650	1750
1700	1805
1750	1875
1800	1920
1850	1980
1900	2040
1950	2100
2000	2155
2050	2200
2100	2250
2150	2320
2200	2375
2250	2435
2300	2480
2350	2550
2400	2600
2450	2655
2500	2710
2550	2770
2600	2830
2650	2895
2700	2970
2750	3000
2800	3070
2850	3110
2900	3175
2950	3240
3000	3300
3050	3360
3100	3400
3150	3477
3200	3540
3250	3590
3300	3665
3350	3680
3400	3730
3450	3820
3500	3900
3550	3950
3600	4005
3650	4080
3700	4140

Table A.1: Distance measurements

A.2 Laser-tower distribution

Dist [mm]	Angle [deg]	Time [ms]
1122	358.15	1084
1119	358.11	1248
1125	357.96	1412
1129	358.13	1576
1127	358.11	1740
1127	358.22	1904
1133	357.96	2072
1134	357.98	2236
1116	358.18	2400
1143	358.00	2564
1124	358.15	2732
1134	358.09	2896
1142	358.02	3060
1127	357.98	3228
1119	358.04	3392
1119	358.26	3560
1122	358.13	3724
1132	358.02	3892
1131	358.07	4056
1129	358.24	4224
1122	358.13	4388
1142	358.00	4556
1119	358.11	4720
1124	358.18	4888
1121	358.22	5052
1123	358.29	5220
1139	358.02	5384
1150	358.15	5552
1134	358.02	5716
1126	358.07	5884

Table A.2: 1000mm mean

Dist [mm]	Angle [deg]	Time [ms]
2142	357.14	1084
2138	357.08	1248
2149	357.14	1412
2117	357.39	1576
2128	357.14	1740
2152	357.14	1908
2145	357.10	2072
2135	357.17	2236
2138	357.19	2400
2138	357.14	2568
2145	357.17	2732
2159	357.21	2896
2138	357.19	3064
2142	357.12	3228
2138	357.14	3396
2135	357.23	3560
2139	357.21	3724
2145	357.12	3892
2149	357.21	4056
2159	357.17	4224
2152	357.21	4388
2142	357.10	4556
2142	357.12	4720
2125	357.28	4888
2149	357.12	5056
2160	357.08	5220
2160	357.08	5388
2142	357.17	5552
2135	357.08	5720
2146	357.14	5884

Table A.3: 2000mm mean

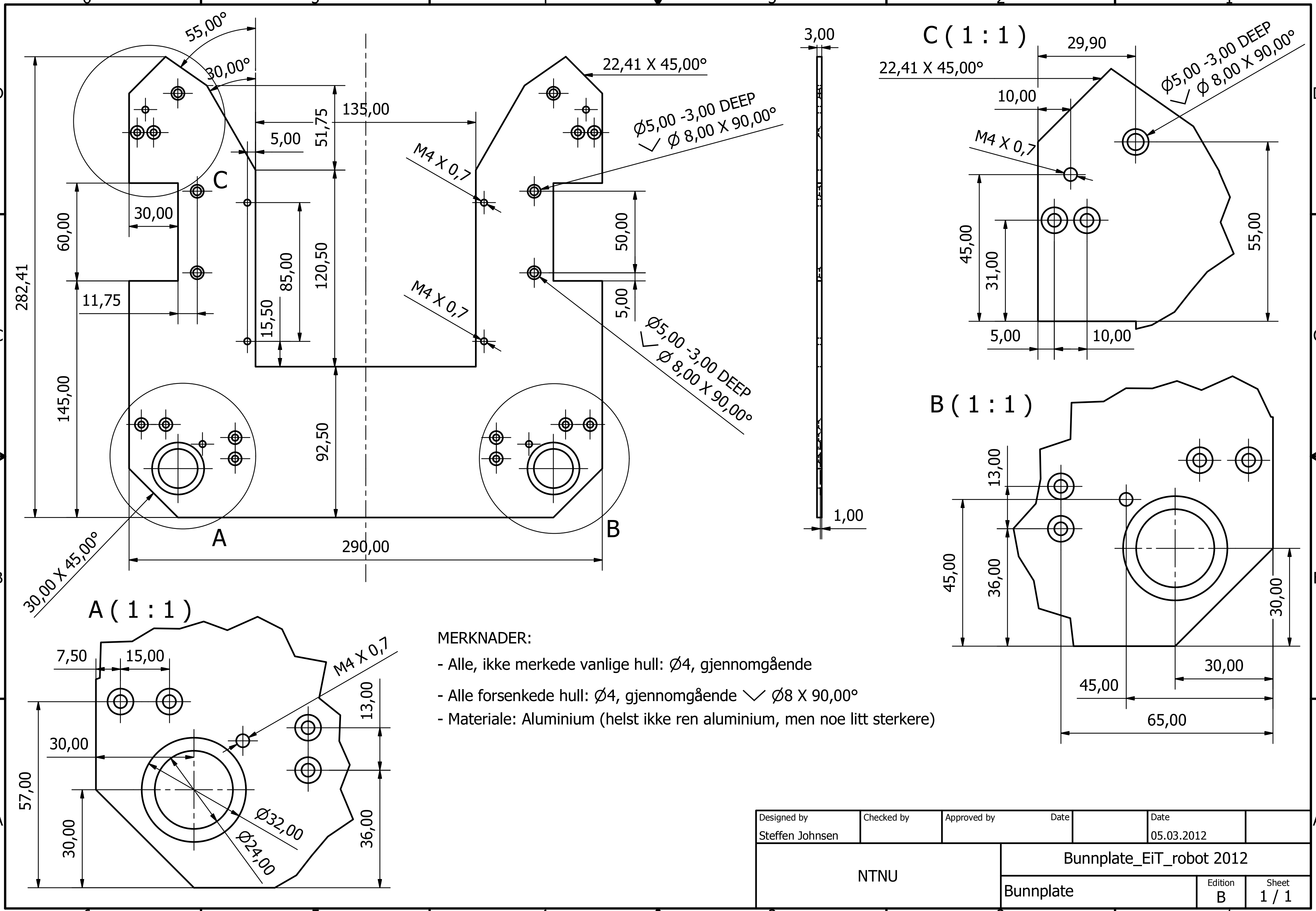
Dist [mm]	Angle [deg]	Time [ms]
3253	357.23	984
3269	357.25	1144
3258	357.21	1308
3291	357.19	1472
3272	357.25	1636
3277	357.10	1804
3250	357.21	1968
3253	357.19	2132
3299	357.21	2296
3263	357.23	2460
3285	357.19	2628
3253	357.23	2792
3247	357.25	2956
3277	357.23	3124
3239	357.23	3288
3271	357.17	3456
3266	357.14	3620
3296	357.12	3788
3266	357.19	3952
3277	357.08	4120
3282	357.21	4284
3282	357.32	4452
3274	357.14	4616
3301	357.08	4784
3274	357.12	4948
3261	357.23	5116
3258	357.21	5280
3280	357.14	5448
3293	357.21	5616
3285	357.23	5780

Table A.4: 3000mm mean

Appendix B

Mechanics

The next page shows the production drawing of the base plate in this years robot. All of the mechanical parts on the robot have drawings like these and are available as digital attachments. Refeer to appendix G for more information.



MERKNADER:

- Alle, ikke merkede vanlige hull: Ø4, gjennomgående
- Alle forsenkede hull: Ø4, gjennomgående ⌵ Ø8 X 90,00°
- Materiale: Aluminium (helst ikke ren aluminium, men noe litt sterkere)

Designed by	Checked by	Approved by	Date	Date
Steffen Johnsen				05.03.2012
NTNU		Bunnplate_EiT_robot 2012		
		Bunnplate	Edition B	Sheet 1 / 1

Appendix C

Electronics

All the schematics/designs presented in this appendix is available on the attached CD or attached files where this paper is published, see appendix G.

C.1 Enemy beacon

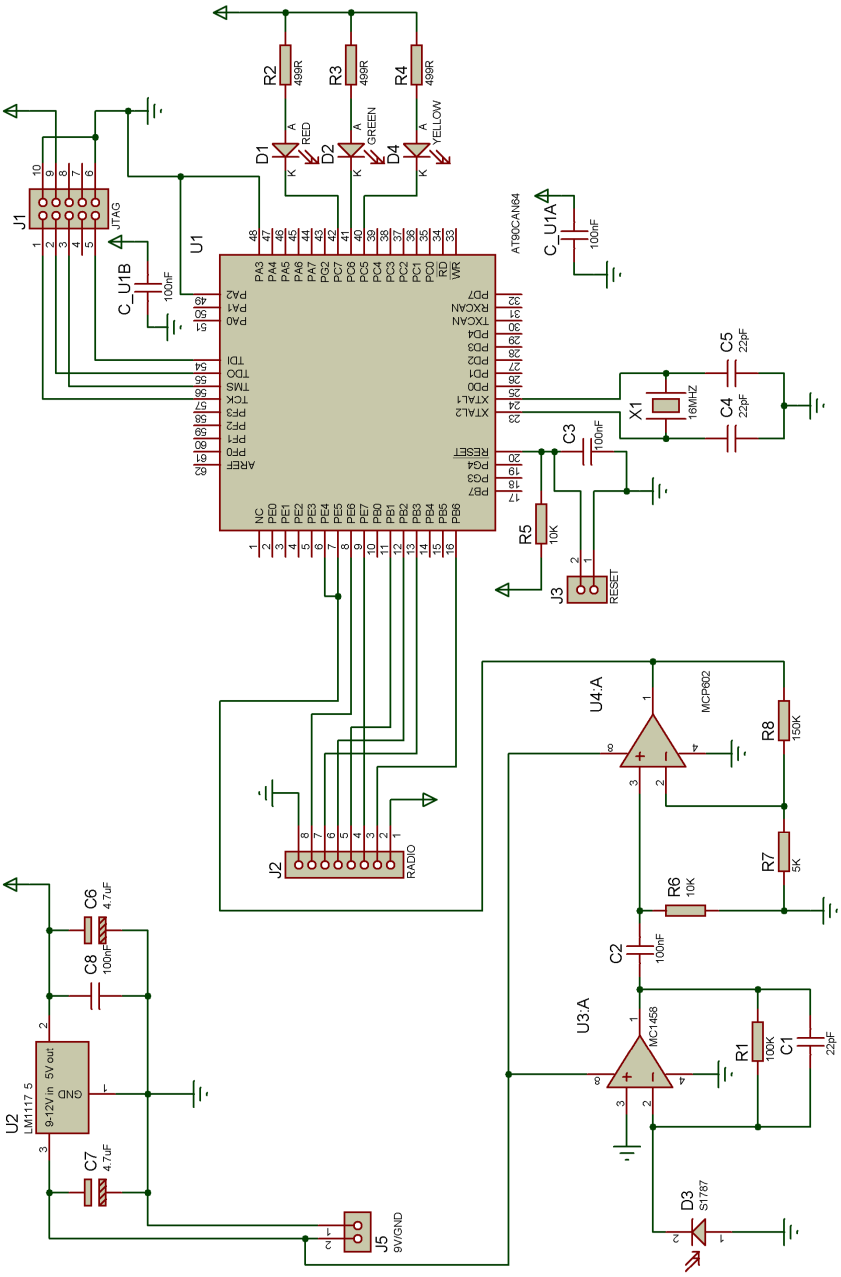


Figure C.1: Enemy Beacon Schematic

C.2 Power card

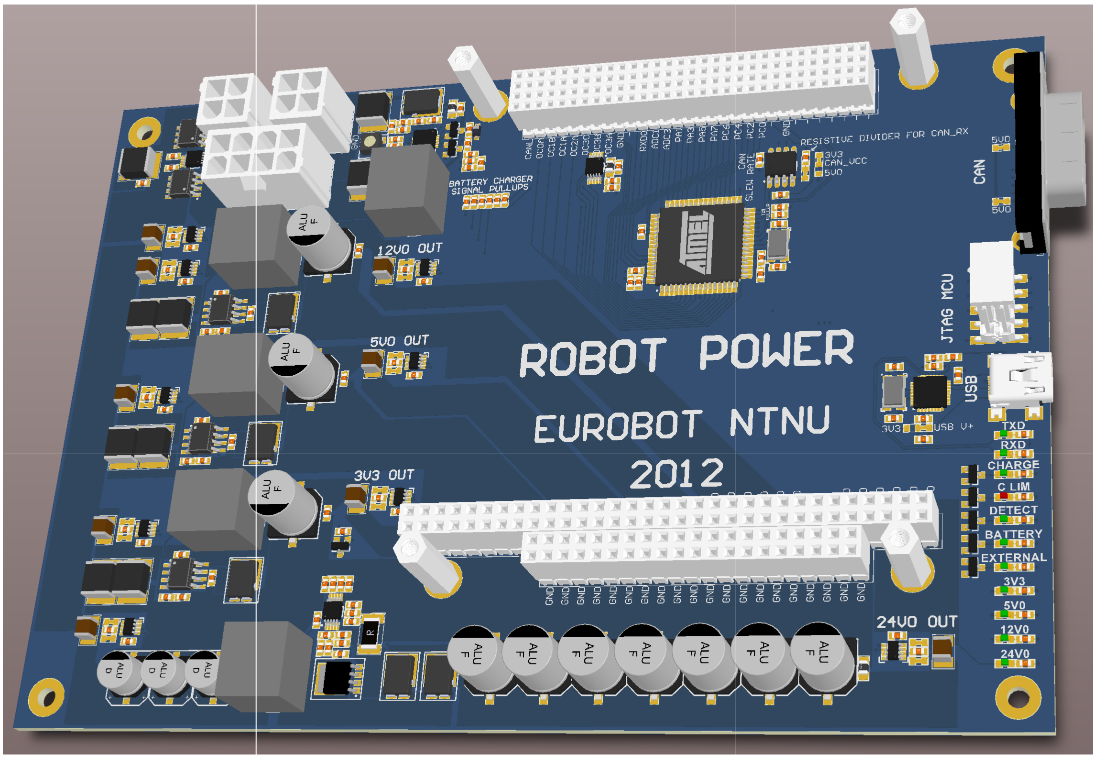
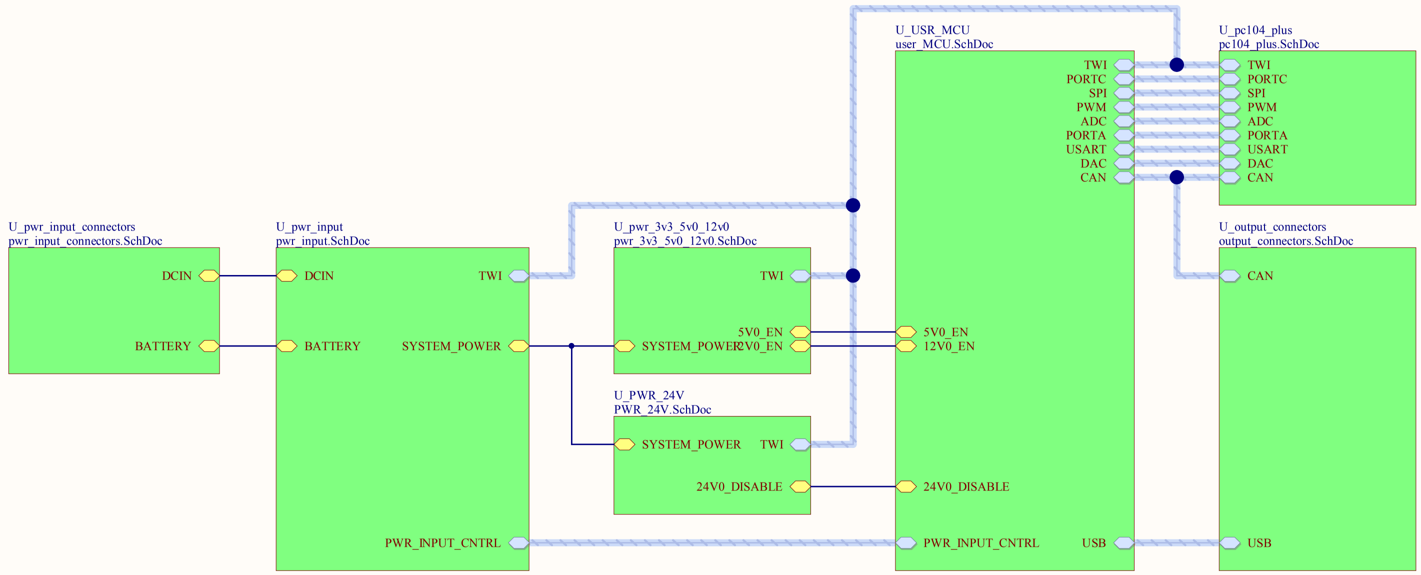
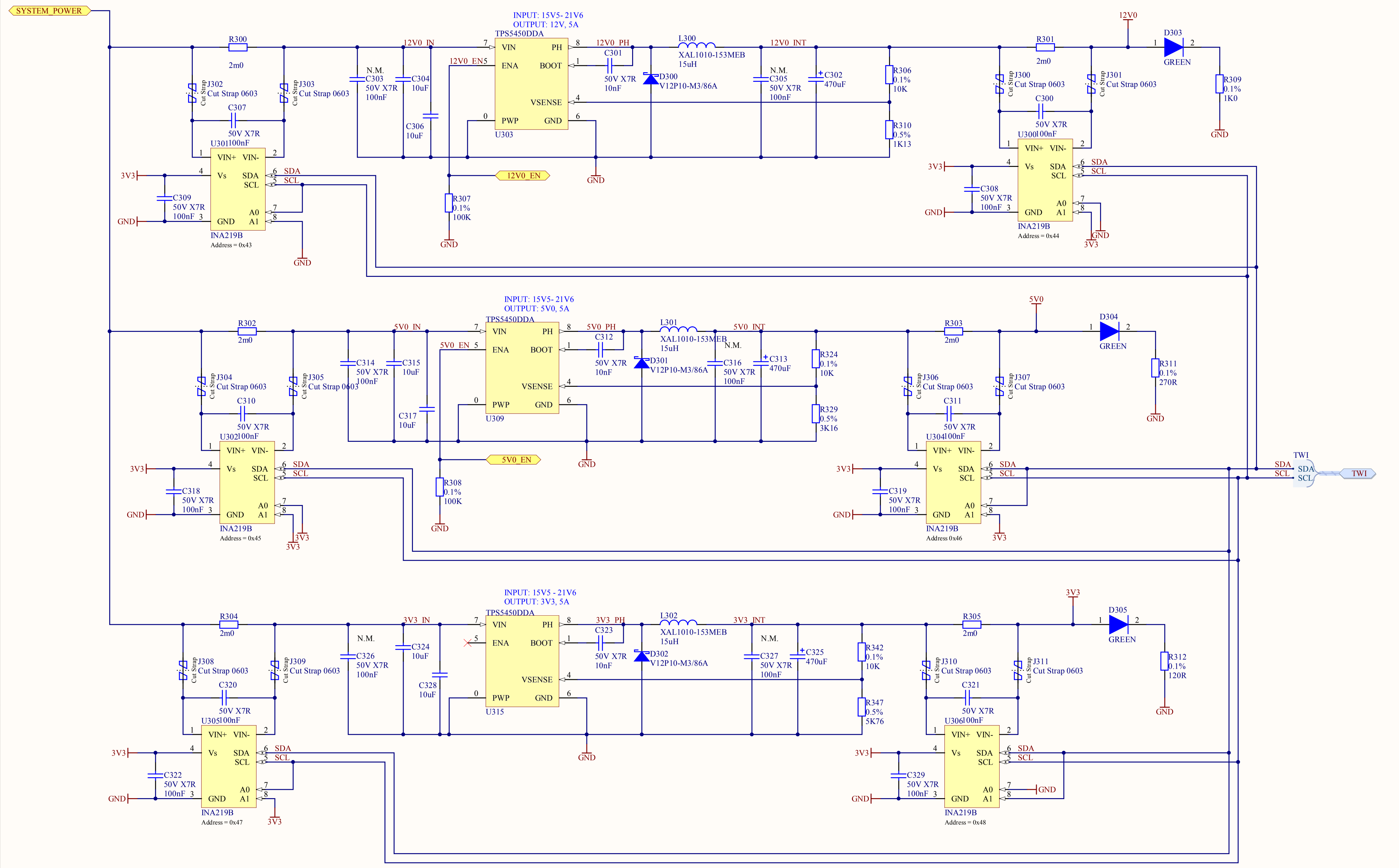


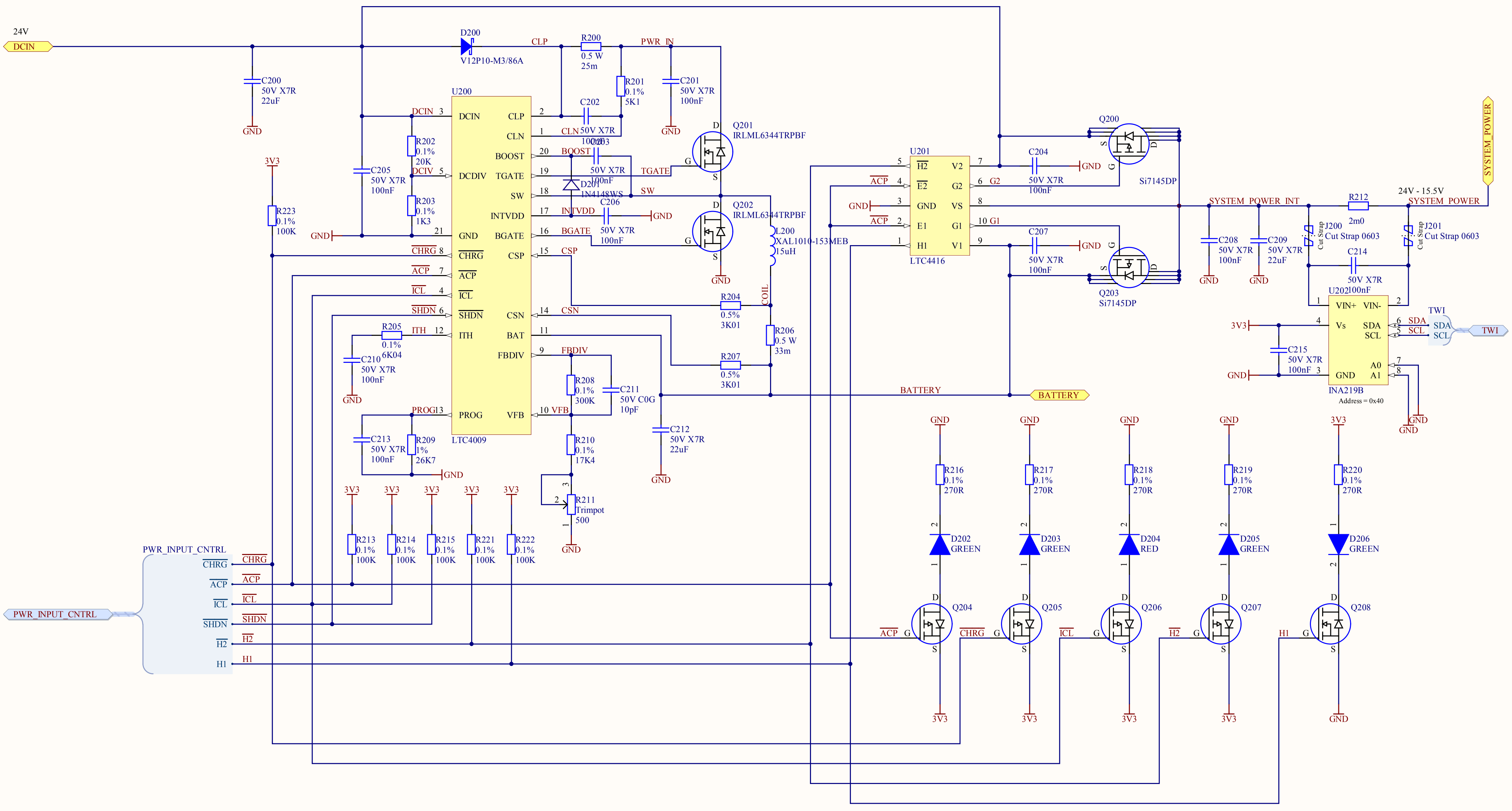
Figure C.2: 3D-model of power card



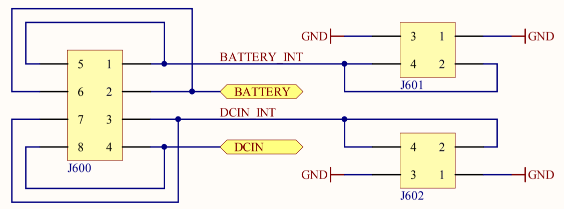
Title			
Size A3	Number		Revision
Date:	28.05.2012	Sheet of	
File:	C:\Users\..\top_level.SchDoc	Drawn By:	



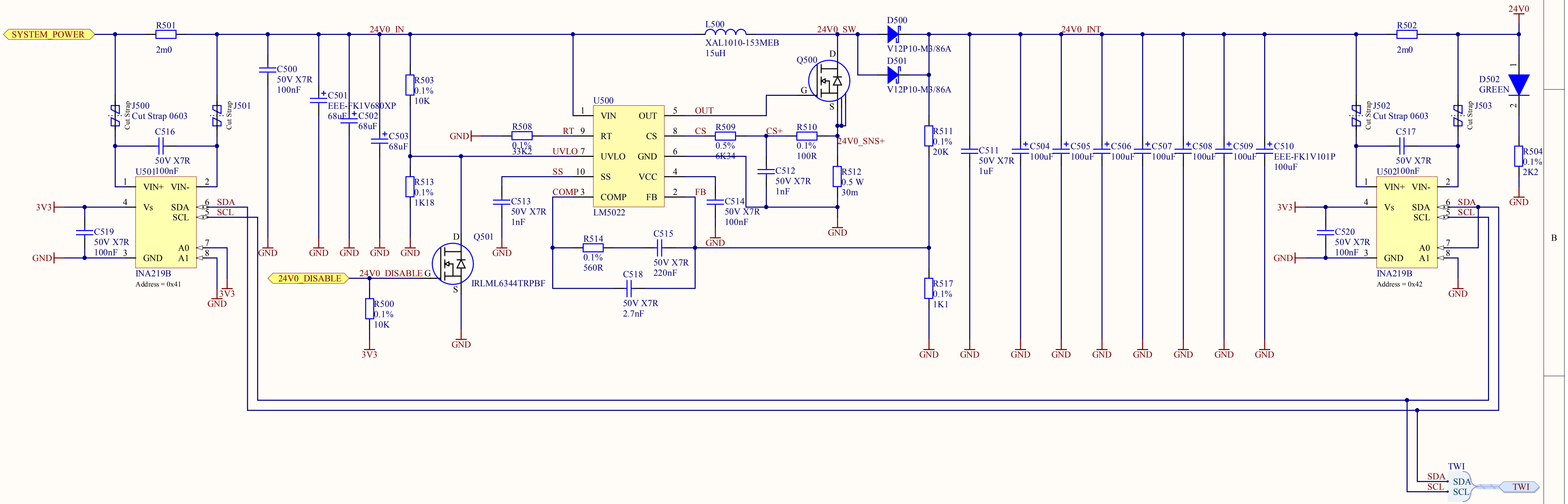
Title		
Size A3	Number	Revision
Date: 28.05.2012		Sheet of
File: C:\Users\..\pwr_3v3_5v0_12v0.SchDoc		Drawn By:



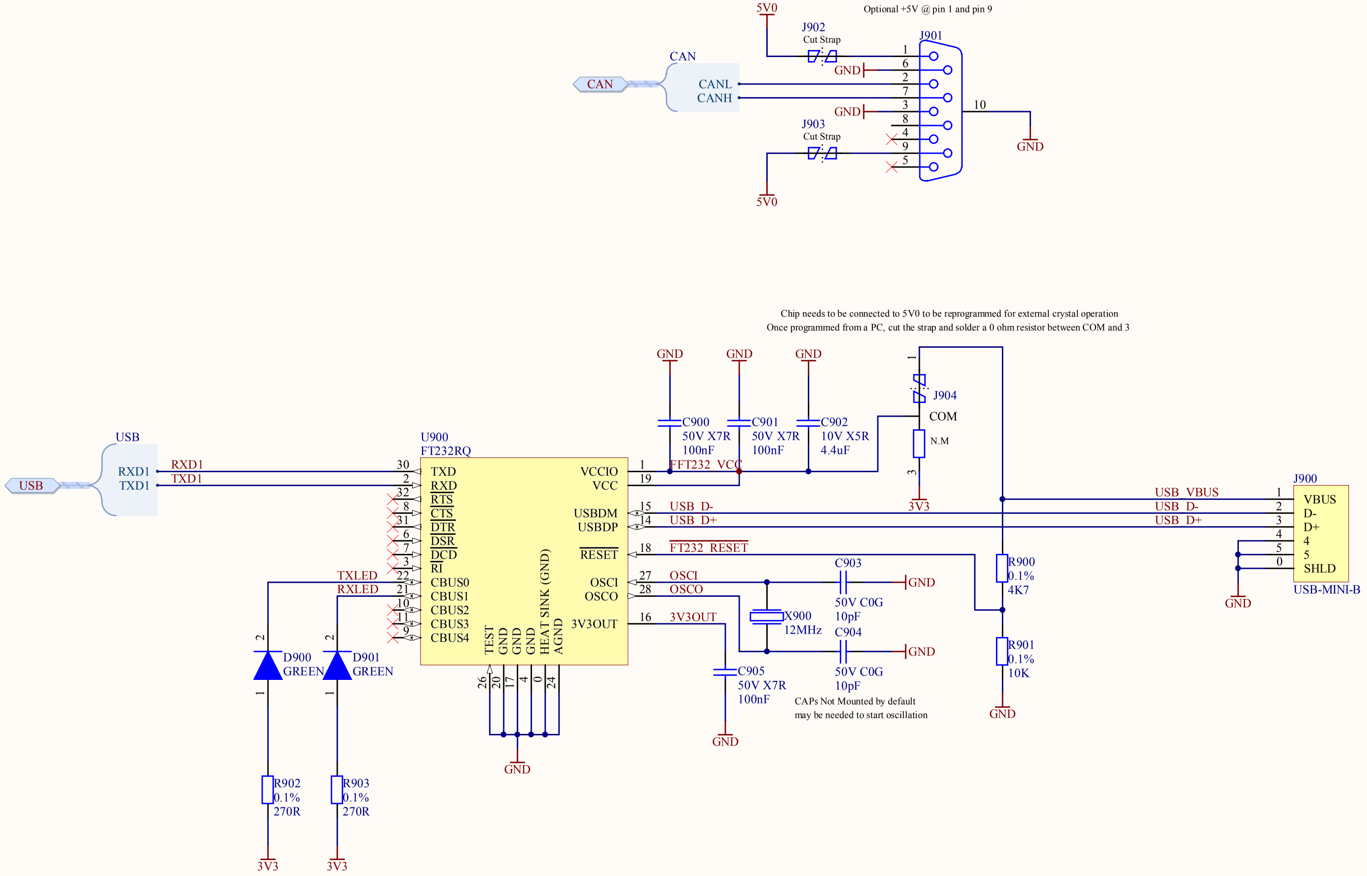
Title		
Size	Number	Revision
A3		
Date:	28.05.2012	Sheet of
File:	C:\Users\..\pwr_input.SchDoc	Drawn By:



Title		
Size A3	Number	Revision
Date:	28.05.2012	Sheet of
File:	C:\Users\..\pwr_input_connectors.SchDoc	Drawn By:



Title		
Size	Number	Revision
A3		
Date:	28.05.2012	Sheet of
File:	C:\Users\..\PWR_24V.SchDoc	Drawn By:

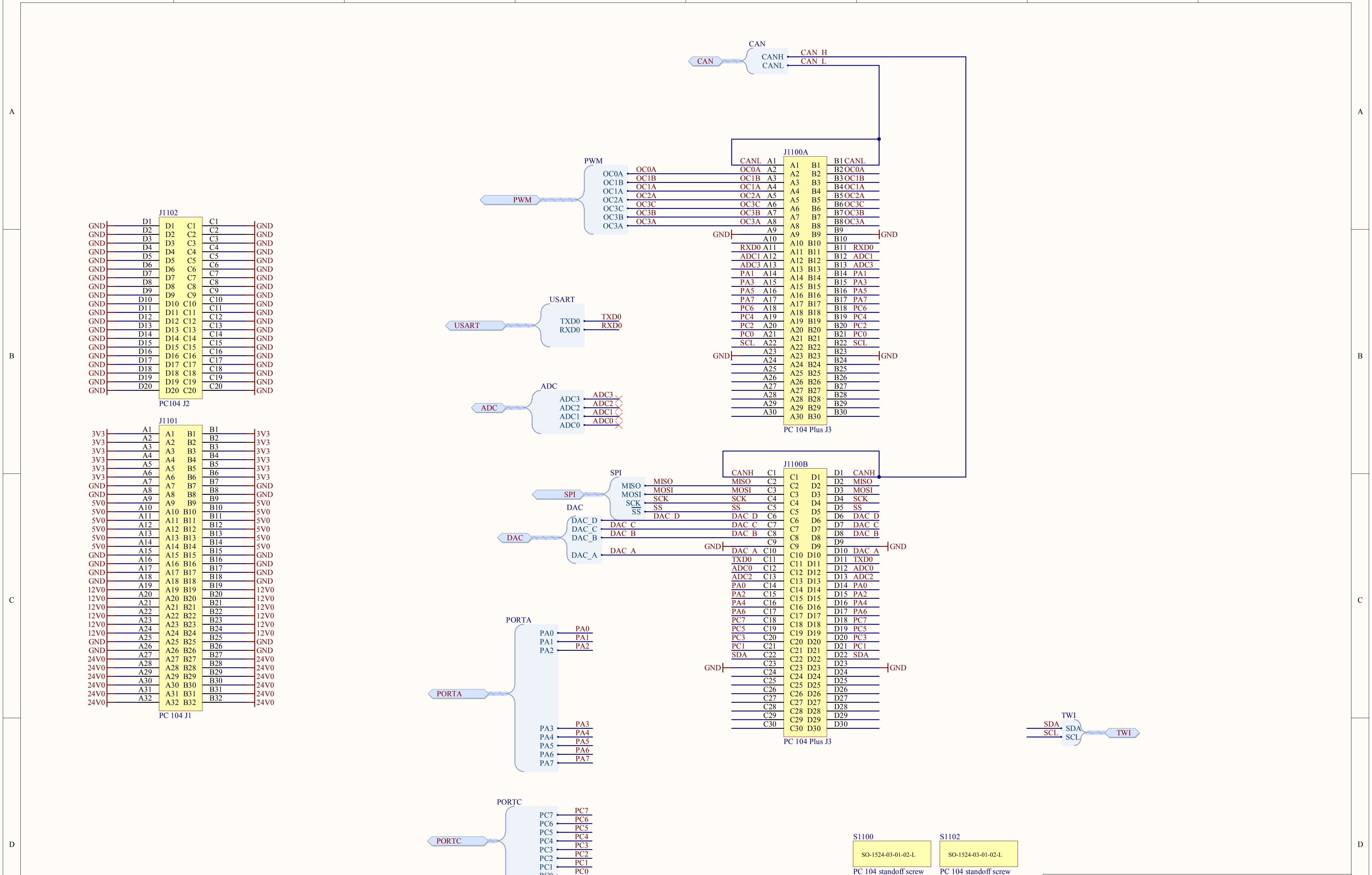


Optional +5V @ pin 1 and pin 9

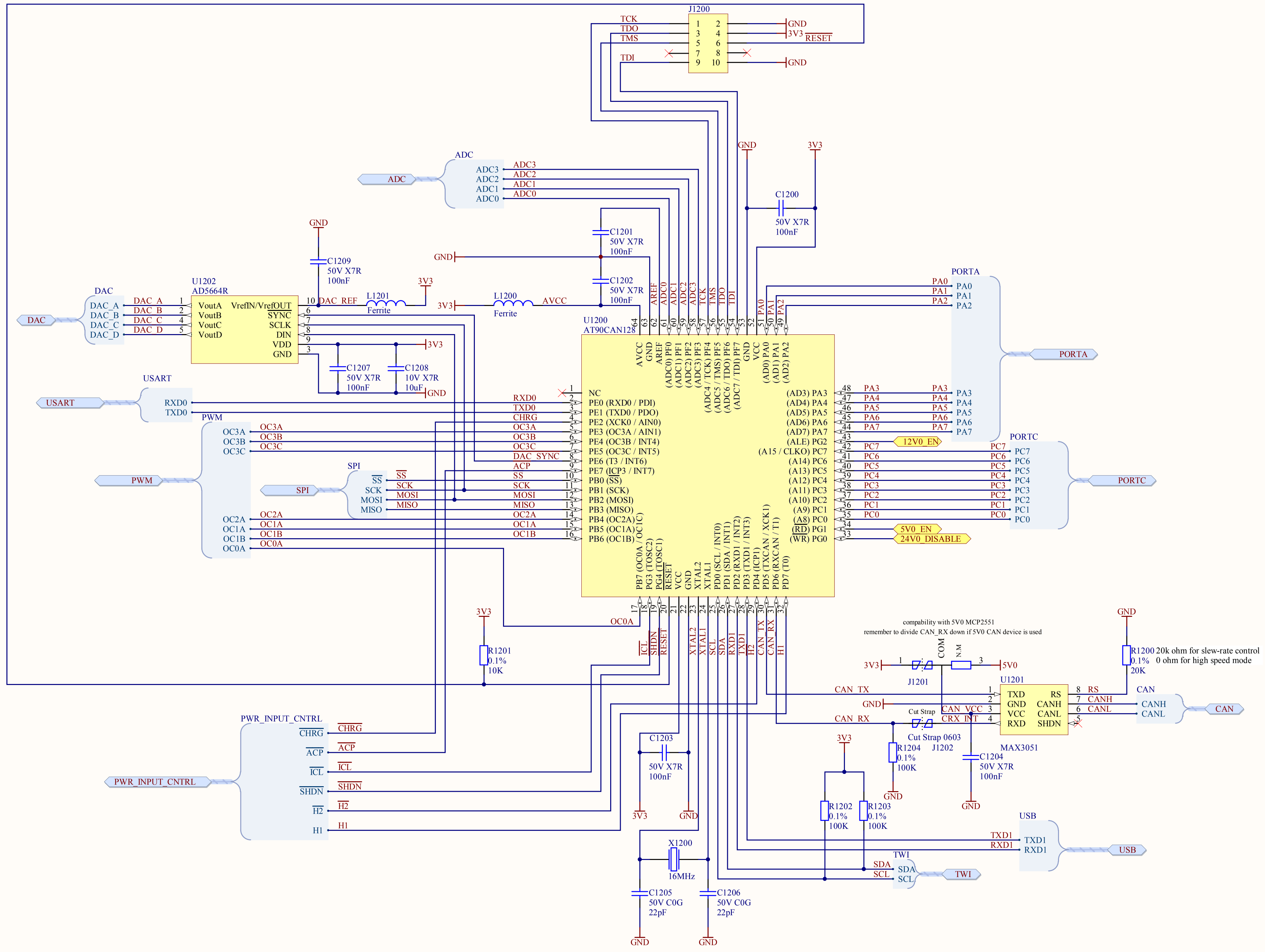
Chip needs to be connected to 5V0 to be reprogrammed for external crystal operation
Once programmed from a PC, cut the strap and solder a 0 ohm resistor between COM and 3

CAPs Not Mounted by default
may be needed to start oscillation

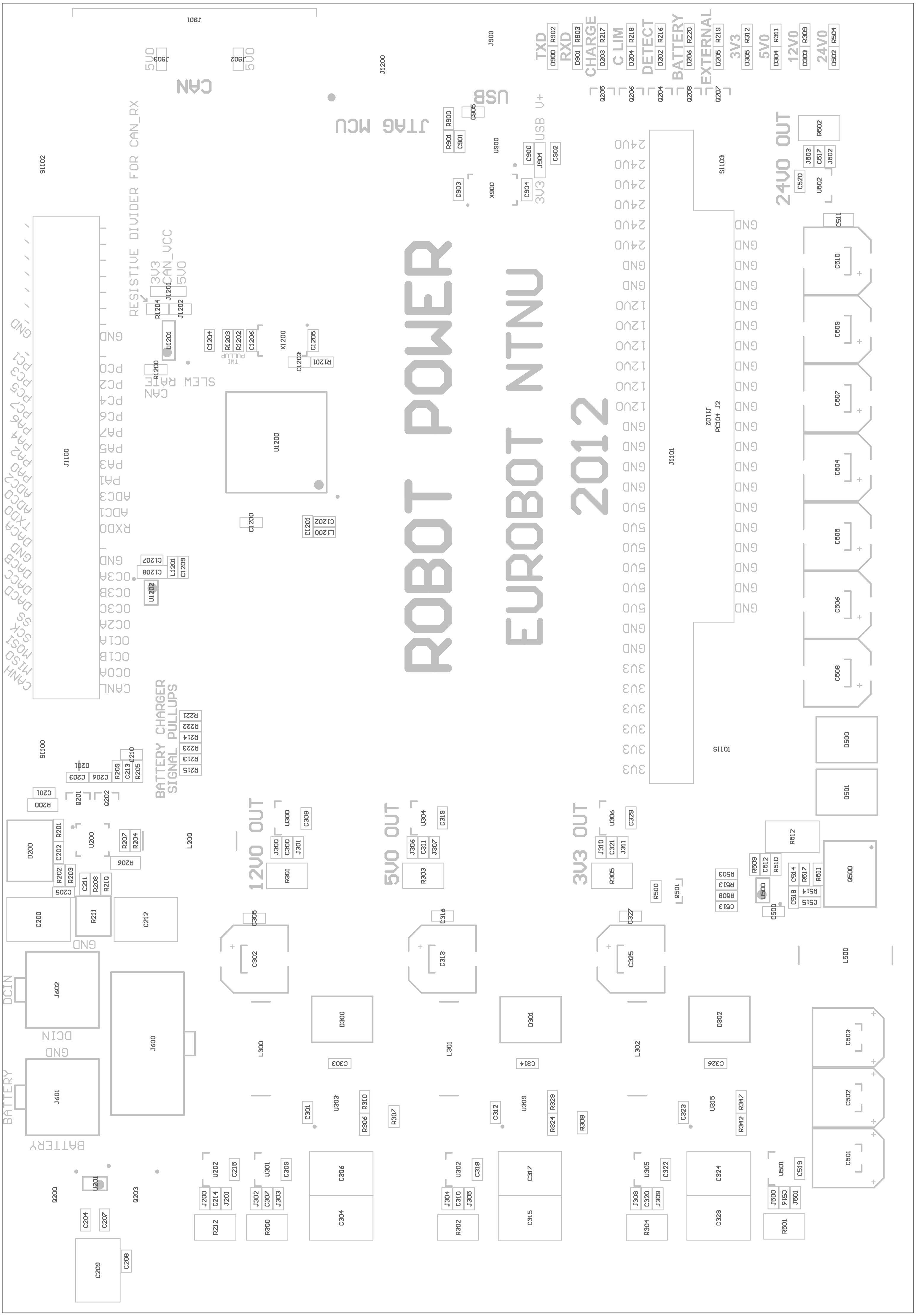
Title		
Size A3	Number	Revision
Date: 28.05.2012		Sheet of
File: C:\Users\..\output_connectors.SchDoc		Drawn By:



Title		
Size A3	Number	Revision
Date:	28.05.2012	Sheet of
File:	C:\Users\..\pc104_plus.SchDoc	Drawn By:



Title		
Size A3	Number	Revision
Date:	28.05.2012	Sheet of
File:	C:\Users\..\user_MCU.SchDoc	Drawn By:


ROBOT POWER
EUROBOT NTNU
2012
CAN
USB
JTAG MCU
USB V+
3V3
TXD
RXD
CHARGE
C LIM
DETECT
BATTERY
EXTERNAL
3V3
5V0
12V0
24V0
24V0 OUT
12V0 OUT
5V0 OUT
3V3 OUT
RESISTIVE DIVIDER FOR CAN_RX
3V3
CAN_VCC
5V0
SLEW RATE
CAN
TWI PULLUP
BATTERY CHARGER SIGNAL PULLUPS
GND
PC0
PC2
PC4
PC6
PA7
PA5
PA3
PA1
ADC3
ADC1
RXD0
GND
OC3A
OC3B
OC3C
OC2A
OC1A
OC1B
OC0A
CANL
CANH
MISO
MOSI
SCK
DAC
TXD0
ADC0
ADC2
PA0
PA2
PA4
PA6
PC7
PC5
PC3
PC1
GND
24V0
12V0
GND
5V0
3V3
PC104 J2
DCIN
BATTERY
GND
U1200
J1100
J1101
J1102
J600
J601
J602
D200
D300
D301
D302
D500
D501
Q500
L200
L300
L301
L302
L500
C200
C209
C212
C302
C313
C325
C304
C306
C315
C317
C324
C328
C501
C502
C503
C504
C505
C506
C507
C508
C509
C510
R211
R212
R300
R301
R302
R303
R304
R305
R501
R502
R512
S1100
S1101
S1102
S1103
J900
J901
J902
J903
U900
X900
X1200

Designator	Quantity	MPN	Manufacturer	Value	Description	Digikey
C200, C209, C212	3	C2220C226M5R2C	Kemet	22uF	CAP CER KPS 22UF 50V X7R 2220	399-5817-1-ND
C201, C202, C203, C204, C205, C206, C207, C208, C210, C213, C214, C215, C300, C303, C305, C307, C308, C309, C310, C311, C314, C316, C318, C319, C320, C321, C322, C326, C327, C329, C500, C514, C516, C517, C519, C520, C900, C901, C905, C1200, C1201, C1202, C1203, C1204, C1207, C1209	46	06035C104JAT2A	AVX	100nF	CAP CER 0.1UF 50V 5% X7R 0603	478-5778-1-ND
C211, C903, C904	3	C1608C0G1H100C	TDK Corporation	10pF	CAP CER 10PF 50V C0G 0603	445-5048-1-ND
C301, C312, C323	3	C0603C103J5RACTU	Kemet	10nF	CAP CER 10nF 50V 5% X7R 0603	399-1092-1-ND
C302, C313, C325	3	EEV-FK1C471P	Panasonic	470uF	CAP ALUM 470UF 16V 20% SMD	PCE3402CT-ND
C304, C306, C315, C317, C324, C328	6	C5750X5R1H106M	TDK	10uF	CAP CER 10UF 50V X5R 20% 2220	445-3498-1-ND
C501, C502, C503	3	EEE-FK1V680XP	Panasonic	68uF	CAP ALUM 68UF 35V 20% SMD	PCE3844CT-ND
C504, C505, C506, C507, C508, C509, C510	7	EEE-FK1V101P	Panasonic	100uF	CAP ALUM 100UF 35V 20% SMD	PCE3835CT-ND
C511	1	UMK212B7105KG-T	Taiyo Yuden	1uF	CAP CER 1.0UF 50V X7R 10% 0805	587-2910-1-ND
C512, C513	2	GRM188R71H102KA01D	Murata	1nF	CAP CER 1nF 50V 10% X7R 0603	490-1494-1-ND
C515	1	GCM188R71H224KA64D	Murata	220nF	CAP CER .22UF 50V X7R 0603	490-5402-1-ND
C518	1	GRM188R71H272KA01D	Murata	2.7nF	CAP CER 2.7nF 50V 10% X7R 0603	490-1502-1-ND
C902	1	C1608X5R1A475K/ 0.50	TDK Corporation	4.4uF	CAP CER 4.7UF 10V 10% X5R 0603	445-7482-1-ND
C1205, C1206	2	C1608C0G1H220F	TDK Corporation	22pF	CAP CER 22pF 50V C0G 1% 0603	445-5366-1-ND
C1208	1	C2012X7R1A106K	TDK Corporation	10uF	CAP CER 10UF 10V 10% X7R 0805	445-6857-1-ND
D200, D300, D301, D302, D500, D501	6	V12P10-M3/ 86A	Vishay/ General Semiconductor		DIODE SCHOTTKY 12A 100V SMPC	V12P10-M3/ 86AGICT-ND
D201	1	1N4148WS	Fairchild Semiconductor		DIODE 75V 150MA SOD323F	1N4148WSFSCT-ND
D202, D203, D205, D206, D303, D304, D305, D502, D900, D901	10	APT1608SGC	Kingbright		LED 1.6X0.8MM 568NM GRN CLR SMD	754-1121-1-ND
D204	1	APT1608EC	Kingbright		LED 1.6X0.8MM 625NM RED CLR SMD	754-1117-1-ND
J600	1	39-29-3086	Molex		CONN HEADER 8POS 4.2MM VERT TIN	WM3846-ND
J601, J602	2	39-28-1043	Molex		CONN HEADER 4POS 4.2MM VERT TIN	WM3801-ND
J900	1	897-43-005-00-100001	Mill-Max Manufacturing Corp.		CONN RECEPT MINI-USB TYPE B SMT	ED90341CT-ND
J901	1	190-009-163R001	Norcomp Inc		CONN DB9 MALE R/A SOLDER SMD	190-09MA-ND
J1100	1	ESQT-130-03-G-Q-368	Samtec		Non Stack Through	
J1101	1	ESQ-132-12-G-D	Samtec		Non Stack Through	
J1102	1	ESQ-120-12-G-D	Samtec		Non Stack Through	
J1200	1	TSM-105-01-L-DV-P	Samtec Inc		CONN HEADER 10POS .100" SMT GOLD	TSM-105-01-L-DV-P-ND
L200, L300, L301, L302, L500	5	XAL1010-153MEB	Coilcraft	15uH	INDUCTOR POWER 15UH 13.8A SMD	732-1224-1-ND
L1200, L1201	2	FBMH1608HM102-T	Taiyo Yuden		FERRITE BEAD 1000 OHM 0603	587-1739-1-ND
Q200, Q203	2	SI7145DP-T1-GE3	Vishay		MOSFET P-CH D-S 30V 8-SOIC	SI7145DP-T1-GE3CT-ND
Q201, Q202, Q208, Q501	4	IRLML6344TRPBF	International Rectifier		MOSFET N-CH 30V 5A SOT23	IRLML6344TRPBFCT-ND
Q204, Q205, Q206, Q207	4	BSS84-7-F	Diodes Inc		MOSFET P-CH 50V 130MA SOT23-3	BSS84-FDICT-ND
Q500	1	RJK0332DPB-01#J0	Renesas Electronics America		MOSFET N-CH 30V 35A LFPAK	RJK0332DPB-01#J0CT-ND
R200	1	WSLP0805R0250FEB	Vishay/ Dale	25m	RES .025 OHM 1/ 2W 1% 0805 SMD	WSLPB-.025CT-ND
R201	1	PAT0603E1781BST1	Vishay	5K1	RES 5.1K OHM 1/ 10W .1% 0603 SMD	RG1608P-512-B-T5
R202, R511, R1200	3	RG1608P-203-B-T5	Susumu	20K	RES 20.0K OHM 1/ 10W .1% 0603 SMD	RG16P20.0KBCT-ND
R203	1	RG1608P-132-B-T5	Susumu	1K3	RES 1.3K OHM 1/ 10W .1% 0603 SMD	RG16P1.3KBCT-ND
R204, R207	2	RR0816P-3011-D-47H	Susumu	3K01	RES 3.01K OHM 1/ 16W .5% 0603 SMD	RR08P3.01KDCT-ND
R205	1	RG1608P-6041-B-T5	Susumu	6K04	RES 6.04K OHM 1/ 10W .1% 0603 SMD	RG16P6.04KBCT-ND
R206	1	WSLP0805R0330FEB	Vishay/ Dale	33m	RES .033 OHM 1/ 2W 1% 0805 SMD	WSLPB-.033CT-ND
R208	1	RG1608P-304-B-T5	Susumu	300K	RES 300K OHM 1/ 10W .1% 0603 SMD	RG16P300KBCT-ND
R209	1	MCR03EZPFX2672	Rohm Semiconductor	26K7	RES 26.7K OHM 1/ 10W 1% 0603 SMD	RHM26.7KHCT-ND
R210	1	RG1608P-1742-B-T5	Susumu	17K4	RES 17.4K OHM 1/ 10W .1% 0603 SMD	RG16P17.4KBCT-ND
R211	1	PVG3A501C01R00	Murata Electronics North America	500	TRIMMER 500 OHM 0.25W SMD	490-2654-1-ND
R212, R300, R301, R302, R303, R304, R305, R501, R502	9	CSNL1206FT2L00	Panasonic, Stackpole Electronics Inc	2m0	RES .002OHM 1W 1% 1206 SMD	CSNL1206FT2L00CT-ND
R213, R214, R215, R221, R222, R223, R307, R308, R1202, R1203	10	RG1608P-104-B-T5	Susumu	100K	RES 100K OHM 1/ 10W .1% 0603 SMD	RG16P100KBCT-ND
R216, R217, R218, R219, R220, R311, R902, R903	8	RG1608P-271-B-T5	Susumu	270R	RES 270 OHM 1/ 10W .1% 0603 SMD	RG16P270BCT-ND
R306, R324, R342, R500, R503, R901, R1201	7	RG1608P-103-B-T5	Susumu	10K	RES 10.0K OHM 1/ 10W .1% 0603 SMD	RG16P10.0KBCT-ND
R309	1	RG1608P-102-B-T5	Susumu	1K0	RES 1.0K OHM 1/ 10W .1% 0603 SMD	RG16P1.0KBCT-ND
R310	1	RR0816P-1131-D-06H	Susumu	1K13	RES 1.13K OHM 1/ 16W .5% 0603 SMD	RR08P1.13KDCT-ND
R312	1	RG1608P-121-B-T5	Susumu	120R	RES 120 OHM 1/ 10W .1% 0603 SMD	RG16P120BCT-ND
R329	1	RR0816P-3161-D-49H	Susumu	3K16	RES 3.16K OHM 1/ 16W .5% 0603 SMD	RR08P3.16KDCT-ND
R347	1	RR0816P-5761-D-74H	Susumu	5K76	RES 5.76K OHM 1/ 16W .5% 0603 SMD	RR08P5.76KDCT-ND
R504	1	RG1608P-222-B-T5	Susumu	2K2	RES 2.2K OHM 1/ 10W .1% 0603 SMD	RG16P2.2KBCT-ND
R508	1	RG1608P-3322-B-T5	Susumu	33K2	RES 33.2K OHM 1/ 10W .1% 0603 SMD	RG16P33.2KBCT-ND
R509	1	RR0816P-6341-D-78H	Susumu	6K34	RES 6.34K OHM 1/ 16W .5% 0603 SMD	RR08P6.34KDCT-ND
R510	1	RG1608P-101-B-T5	Susumu	100R	RES 100 OHM 1/ 10W .1% 0603 SMD	RG16P100BCT-ND
R512	1	WSLP0805R0330FEB	CSRN2010FK30L0	30m	RES .03 OHM 1W 1% 2010 SMD	CSRN2010FK30L0CT-ND
R513	1	RG1608P-1181-B-T5	Vishay	1K18	RES 1.18K OHM 1/ 10W .1% 0603 SMD	RG16P1.18KBCT-ND
R514	1	RG1608P-561-B-T5	Susumu	560R	RES 560 OHM 1/ 10W .1% 0603 SMD	RG16P560BCT-ND
R517	1	RG1608P-112-B-T5	Susumu	1K1	RES 1.1K OHM 1/ 10W .1% 0603 SMD	RG16P1.1KBCT-ND
R900	1	RG1608P-472-B-T5	Susumu	4K7	RES 4.7K OHM 1/ 10W .1% 0603 SMD	RG16P4.7KBCT-ND
S1100, S1101, S1102, S1103	4	SO-1524-03-01-02-L	Samtec		15.24mm	
U200	1	LTC4009CUF#PBF	Linear Technology		IC CHARGER BATTERY MC 20-QFN	LTC4009CUF#PBF-ND
U201	1	LTC4416EMS#PBF	Linear Technology		IC CTLR POWERPATH 10-MSOP	LTC4416EMS#PBF-ND
U202, U300, U301, U302, U304, U305, U306, U501, U502	9	INA219BIDCNT	Texas Instruments		IC MONITOR PWR/CURR BID SOT-23-8	296-27898-1-ND
U303, U309, U315	3	TPS5450DDA	Texas Instruments		IC BUCK ADJ 5A 8SO	296-21715-5-ND
U500	1	LM5022MM/ NOPB	National Semiconductor		IC BOOST/ SEPIC SYNC 1A 10MSOP	LM5022MMCT-ND
U900	1	FT232RQ-REEL	FTDI		IC USB FS SERIAL UART 32-QFN	768-1008-1-ND
U1200	1	AT90CAN128-16AU	Atmel		IC MCU AVR FLASH 128K 64-TQFP	AT90CAN128-16AU-ND
U1201	1	MAX3051ESA+	Maxim Integrated Products		IC TXRX CAN 1MBPS 8-SOIC	MAX3051ESA+-ND
U1202	1	AD5664RBRMZ-3	Analoge Devices Inc		IC DAC NANO 16BIT 1.25V 10-MSOP	AD5664RBRMZ-3-ND
X900	1	7B-12.000MAAJ-T	TXC CORPORATION		CRYSTAL 12.000 MHZ 18PF SMD	887-1099-1-ND
X1200	1	7B-16.000MAAJ-T	TXC CORPORATION		CRYSTAL 16.000 MHZ 18PF	887-1104-1-ND

C.3 PC/104 mechanical specification

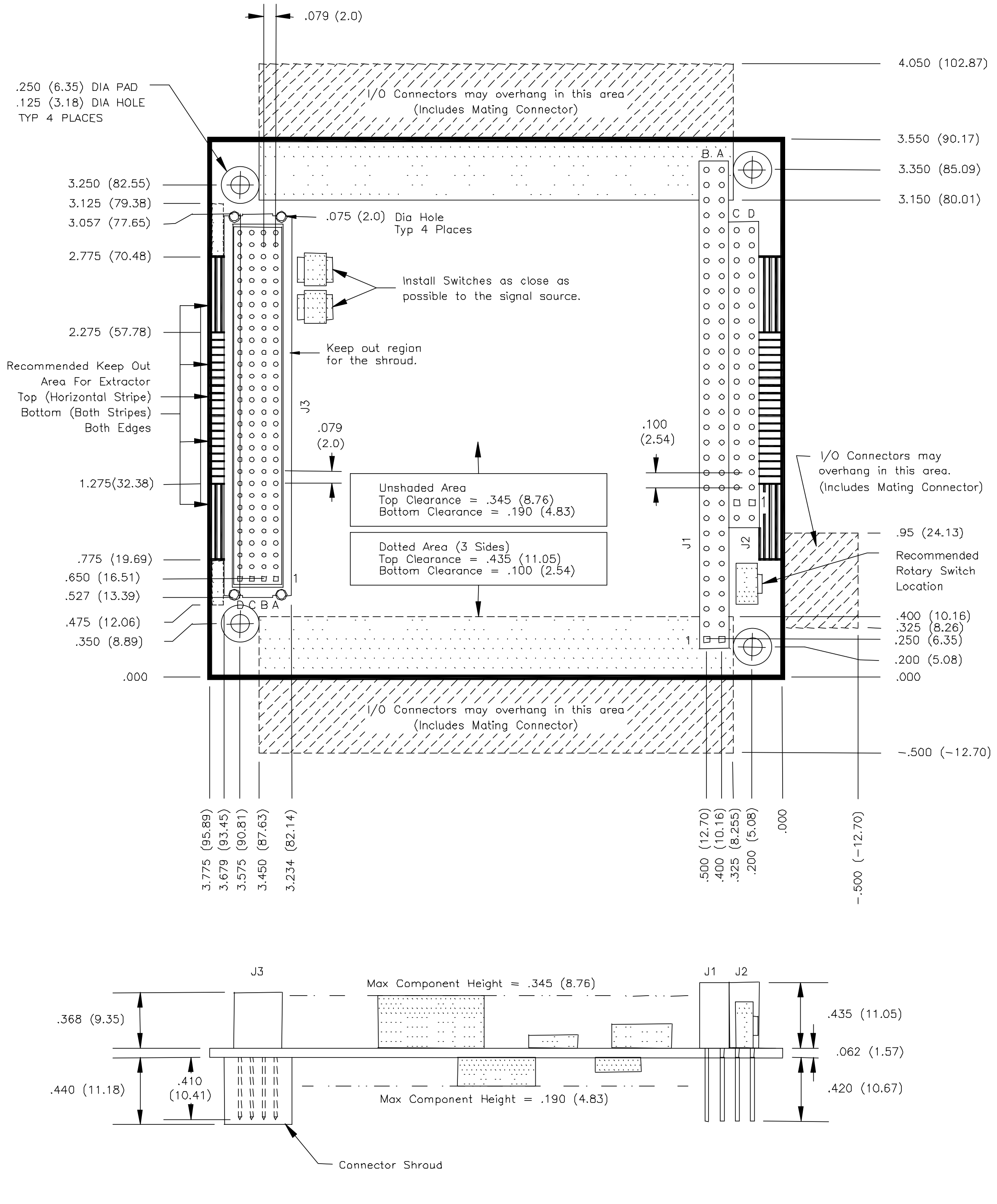


Figure C.3: PC/104-Plus

J1/P1		
Pin	**Row A**	**Row B**
1	3.3V	3.3V
2	3.3V	3.3V
3	3.3V	3.3V
4	3.3V	3.3V
5	3.3V	3.3V
6	3.3V	3.3V
7	GND	GND
8	GND	GND
9	5.0V	5.0V
10	5.0V	5.0V
11	5.0V	5.0V
12	5.0V	5.0V
13	5.0V	5.0V
14	5.0V	5.0V
15	GND	GND
16	GND	GND
17	GND	GND
18	GND	GND
19	12.0V	12.0V
20	12.0V	12.0V
21	12.0V	12.0V
22	12.0V	12.0V
23	12.0V	12.0V
24	12.0V	12.0V
25	GND	GND
26	GND	GND
27	24.0V	24.0V
28	24.0V	24.0V
29	24.0V	24.0V
30	24.0V	24.0V
31	24.0V	24.0V
32	24.0V	24.0V

J2/P2		
Pin	**Row D**	**Row C**
1	GND	GND
2	GND	GND
3	GND	GND
4	GND	GND
5	GND	GND
6	GND	GND
7	GND	GND
8	GND	GND
9	GND	GND
10	GND	GND
11	GND	GND
12	GND	GND
13	GND	GND
14	GND	GND
15	GND	GND
16	GND	GND
17	GND	GND
18	GND	GND
19	GND	GND
20	GND	GND

Table C.1: ISA Connector Pin Definitions

J3/P3				
Pin	**A**	**B**	**C**	**D**
1	CAN_L	CAN_L	CAN_H	CAN_H
2	OC0A	OC0A	MISO	MISO
3	OC1B	OC1B	MOSI	MOSI
4	OC1A	OC1A	SCK	SCK
5	OC2A	OC2A	SS	SS
6	OC3C	OC3C	DAC_D	DAC_D
7	OC3B	OC3B	DAC_C	DAC_C
8	OC3A	OC3A	DAC_B	DAC_B
9	GND	GND	GND	GND
10	NC	NC	DAC_A	DAC_A
11	RXD0	RXD0	TXD0	TXD0
12	ADC1	ADC1	ADC0	ADC0
13	ADC3	ADC3	ADC2	ADC2
14	PA1	PA1	PA0	PA0
15	PA3	PA3	PA2	PA2
16	PA5	PA5	PA4	PA4
17	PA7	PA7	PA6	PA6
18	PC6	PC6	PC7	PC7
19	PC4	PC4	PC5	PC5
20	PC2	PC2	PC3	PC3
21	PC0	PC0	PC1	PC1
22	SCL	SCL	SDA	SDA
23	GND	GND	GND	GND
24	NC	NC	NC	NC
25	NC	NC	NC	NC
26	NC	NC	NC	NC
27	NC	NC	NC	NC
28	NC	NC	NC	NC
29	NC	NC	NC	NC
30	NC	NC	NC	NC

Table C.2: PCI Connector Pin Definitions

C.4 Motor controller breakout board

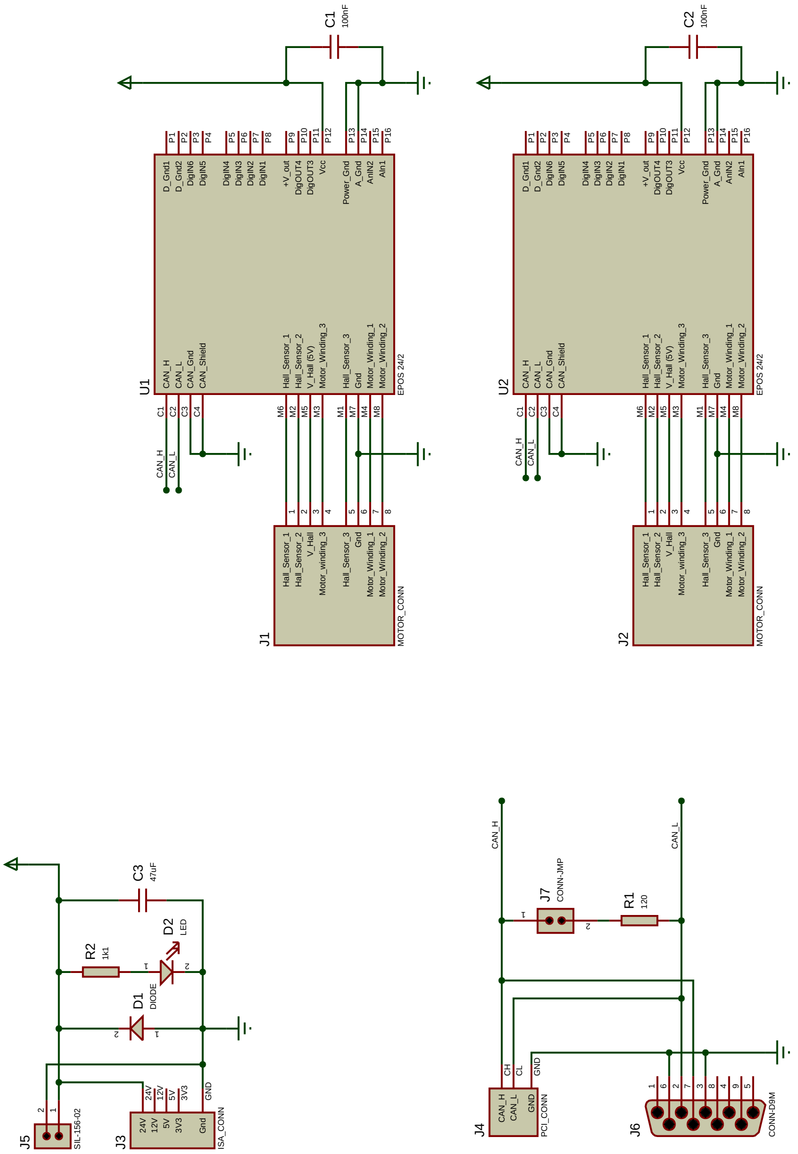


Figure C.4: Schematics

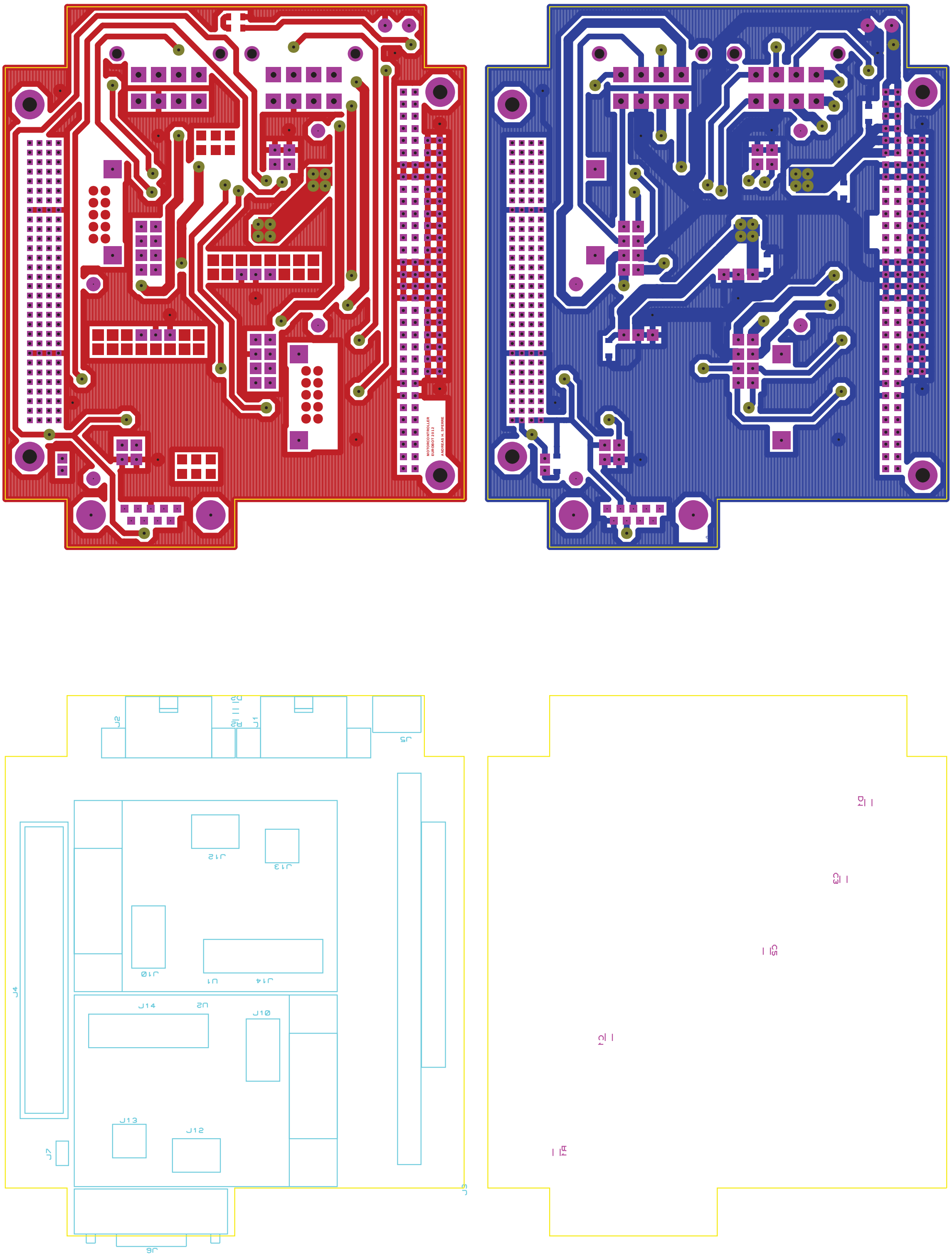


Figure C.5: Layout

C.5 Top card

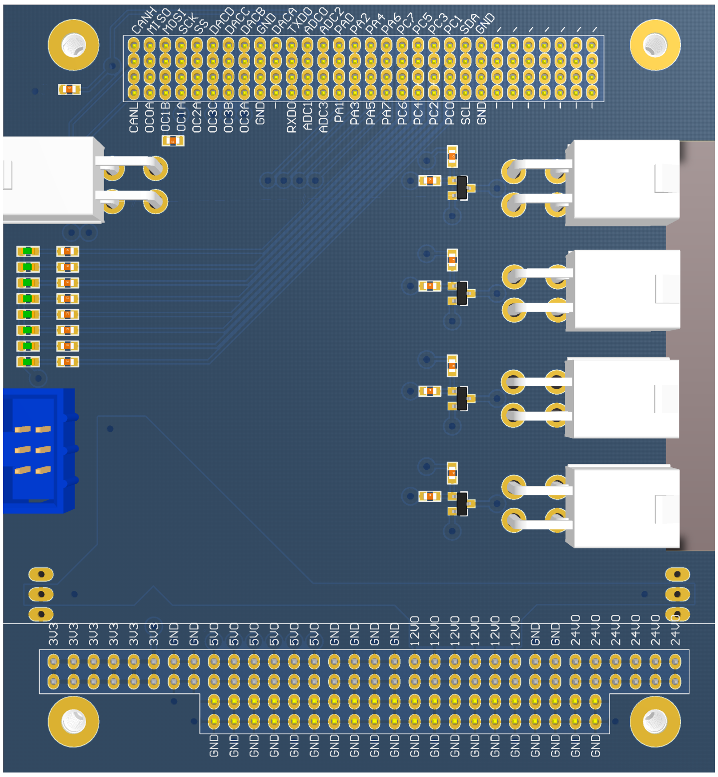
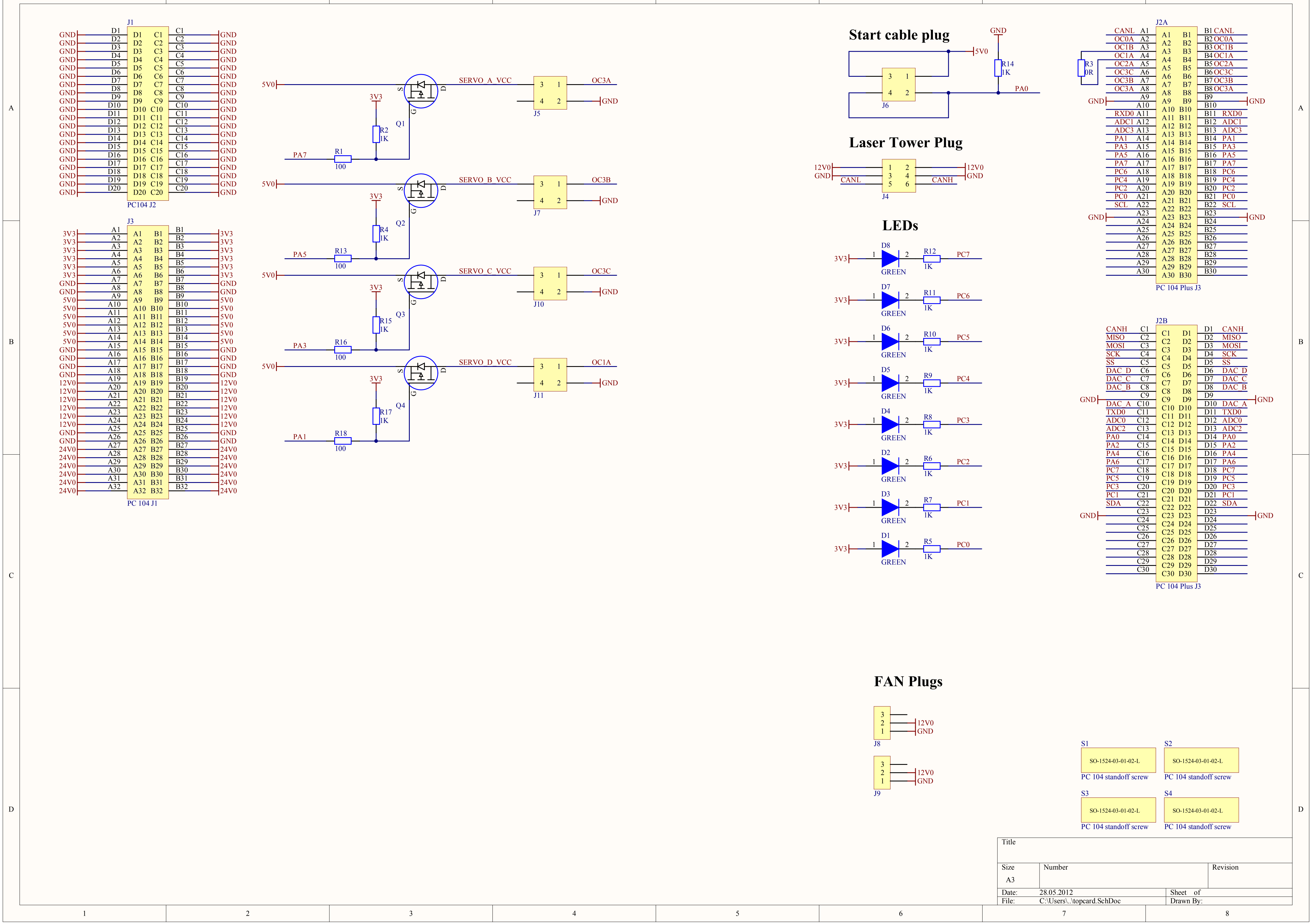
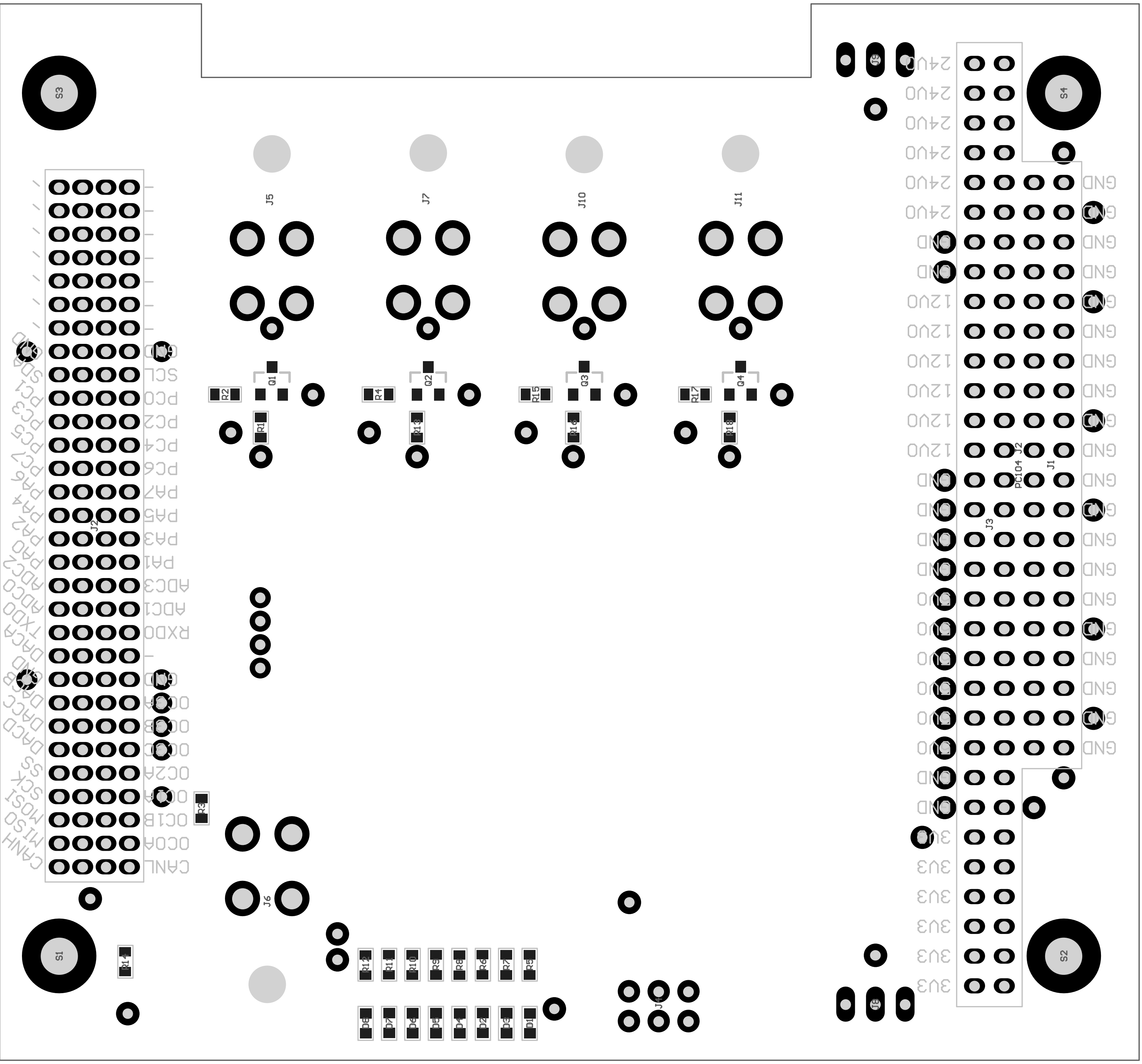


Figure C.6: 3D-model of top card

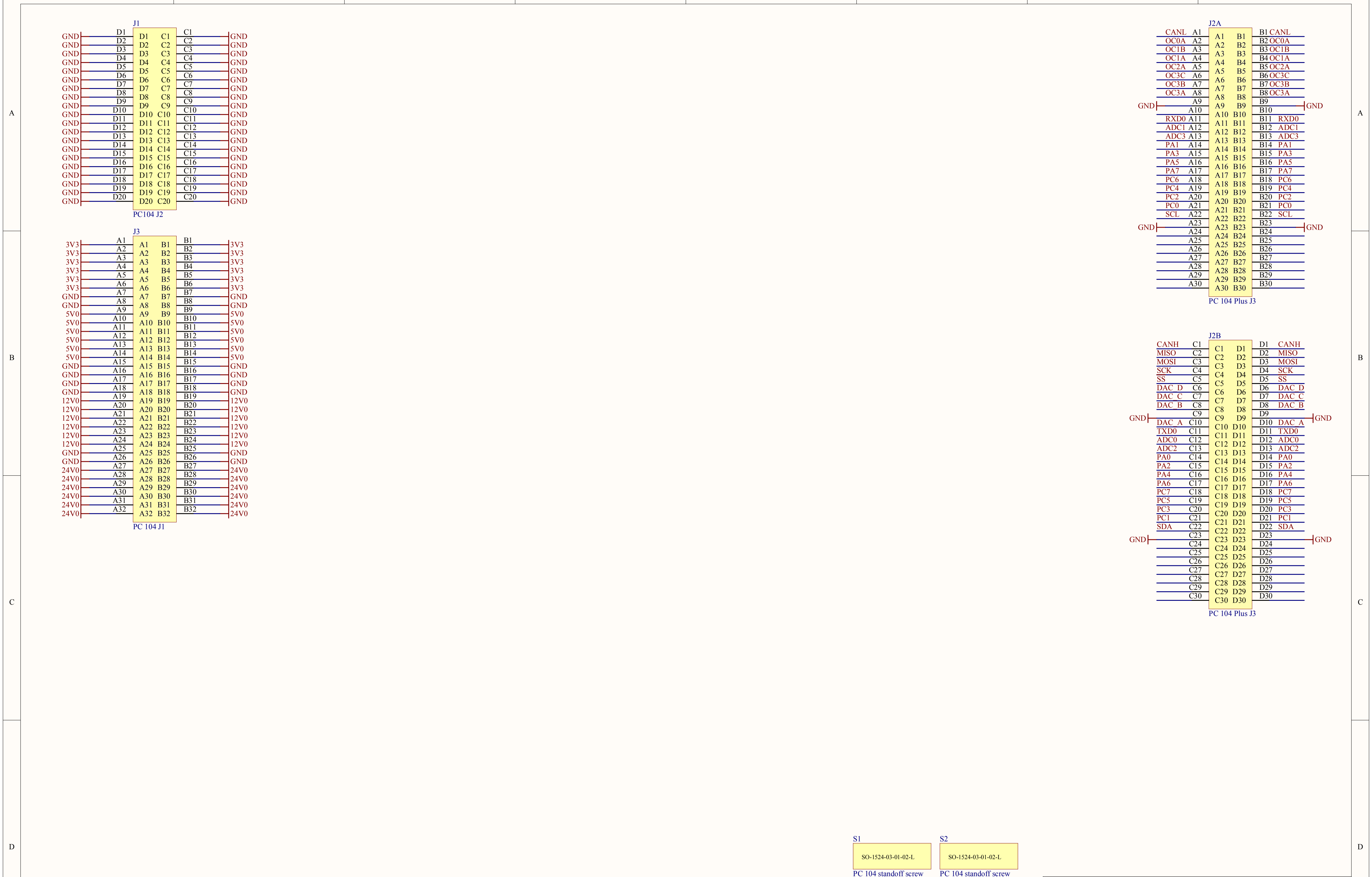


Start cable plug
J6
5V0
GND
R14
1K
PA0
R3
0R
Laser Tower Plug
J4
12V0
GND
CANL
CANH
LEDs
D1
D2
D3
D4
D5
D6
D7
D8
GREEN
3V3
R5
R6
R7
R8
R9
R10
R11
R12
1K
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7
FAN Plugs
J8
J9
12V0
GND
S1
S2
S3
S4
SO-1524-03-01-02-L
PC 104 standoff screw
J1
PC104 J2
J3
PC 104 J1
J2A
J2B
PC 104 Plus J3
Q1
Q2
Q3
Q4
SERVO_A_VCC
SERVO_B_VCC
SERVO_C_VCC
SERVO_D_VCC
OC3A
OC3B
OC3C
OC1A
J5
J7
J10
J11
5V0
3V3
R1
R2
R4
R13
R15
R16
R17
R18
100
1K
PA7
PA5
PA3
PA1
CANL
OC0A
OC1B
OC1A
OC2A
OC3C
OC3B
OC3A
RXD0
ADC1
ADC3
PA1
PA3
PA5
PA7
PC6
PC4
PC2
PC0
SCL
CANH
MISO
MOSI
SCK
SS
DAC_D
DAC_C
DAC_B
DAC_A
TXD0
ADC0
ADC2
PA0
PA2
PA4
PA6
PC7
PC5
PC3
PC1
SDA
GND
12V0
24V0
Title
Size
A3
Number
Revision
Date:
28.05.2012
Sheet of
File:
C:\Users\..\topcard.SchDoc
Drawn By:

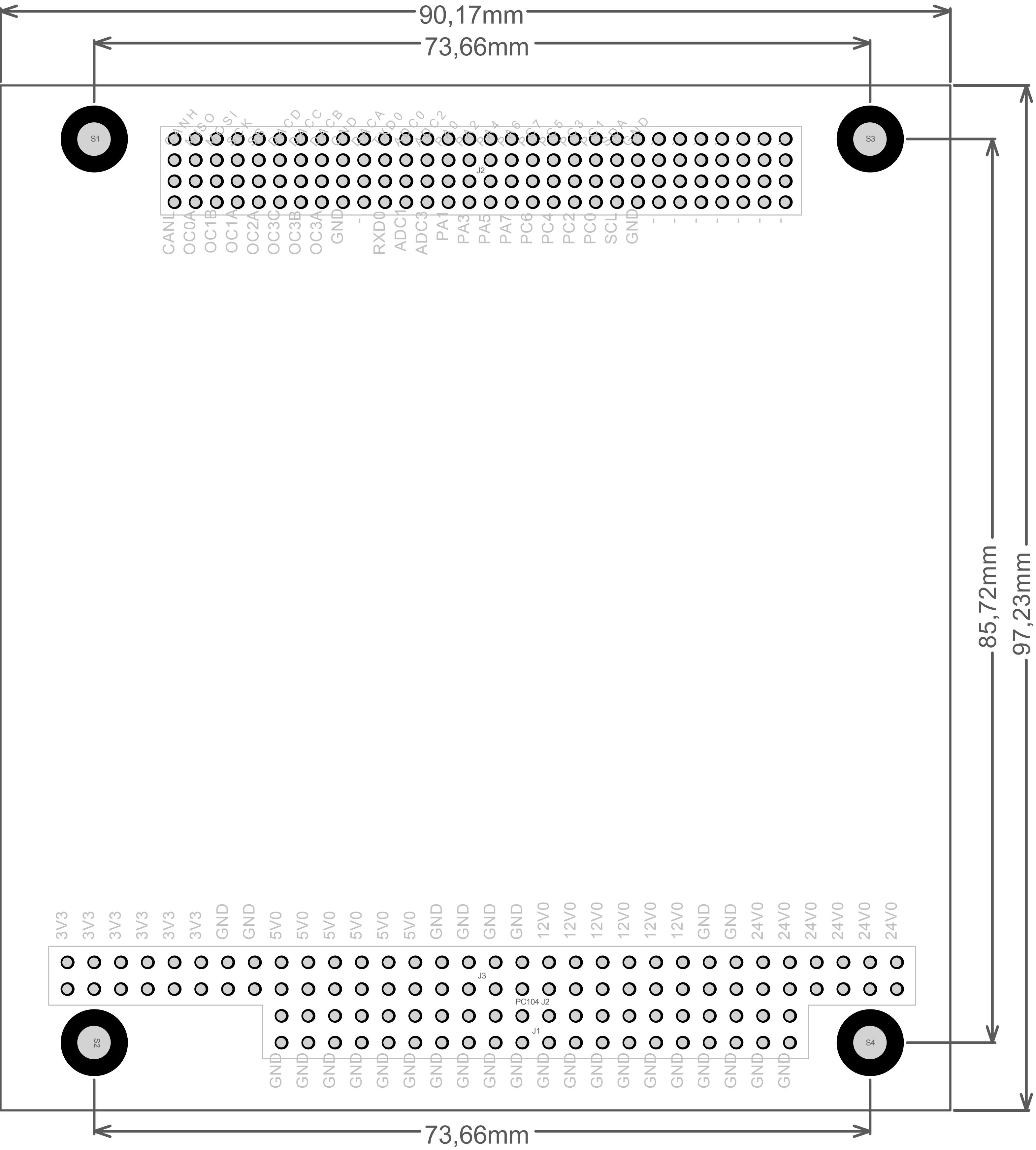

S1
S2
S3
S4
J1
J2
J3
J4
J5
J6
J7
J8
J9
J10
J11
PC104
Q1
Q2
Q3
Q4
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
D1
D2
D3
D4
D5
D6
D7
D8
GND
SCL
PC0
PC2
PC4
PC6
PA7
PA5
PA3
PA1
ADC3
ADC1
RXD0
OC0A
OC0B
OC0C
OC2A
OC1A
OC1B
OC0A
CANL
SDA
PC1
PC3
PC5
PC7
PA6
PA4
PA2
PA0
ADC2
ADC0
TXD0
DACA
DACB
DACC
DACD
SS
SCK
MOSI
MISO
CANH
24V0
12V0
3V3

Designator	Quantity	MPN	Manufacturer	Value	Description	Digikey
D1, D2, D3, D4, D5, D6, D7, D8	8	APT1608SGC	Kingbright		LED 1.6X0.8MM 568NM GRN CLR SMD	754-1121-1-ND
J1	1	ESQ-120-12-G-D	Samtec		Non Stack Through	
J2	1	ESQT-130-03-G-Q-368	Samtec		Non Stack Through	
J3	1	ESQ-132-12-G-D	Samtec		Non Stack Through	
J4	1				Generic 100mil pin-header	
J5, J6, J7, J10, J11	5	39-29-1040	Molex		CONN HEADER 4POS 4.2MM VERT TIN	
J8, J9	2				Generic RA 100mil pin-header	
Q1, Q2, Q3, Q4	4	IRLML6402TRPBF	INTERNATIONAL RECTIFIER		MOSFET P-CH -20V -3.7A SOT23-3	
R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18	18			0R, 1K, 100	Generic resistor	
S1, S2, S3, S4	4	SO-1524-03-01-02-L	Samtec		15.24mm	

C.6 Top card template



Title		
Size A3	Number	Revision
Date: 28.05.2012		Sheet of
File: C:\Users\..\topcard.SchDoc		Drawn By:


90,17mm
73,66mm
S1
S3
CANH
MISO
MOSI
SCK
SS
DACD
DACC
DACB
GND
DACA
TXD0
ADC0
ADC2
PA0
PA2
PA4
PA6
PC7
PC5
PC3
PC1
SDA
GND
J2
CANL
OC0A
OC1B
OC1A
OC2A
OC3C
OC3B
OC3A
GND
-
RXD0
ADC1
ADC3
PA1
PA3
PA5
PA7
PC6
PC4
PC2
PC0
SCL
GND
-
-
-
-
-
-
-
85,72mm
97,23mm
3V3
3V3
3V3
3V3
3V3
3V3
GND
GND
5V0
5V0
5V0
5V0
5V0
5V0
GND
GND
GND
GND
12V0
12V0
12V0
12V0
12V0
12V0
GND
GND
24V0
24V0
24V0
24V0
24V0
24V0
J3
PC104 J2
J1
S2
S4
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
GND
73,66mm

C.7 Switchmode tests

Figure C.7 shows a picture of the setup used during the power card tests, next is the schematics for the electronic load.

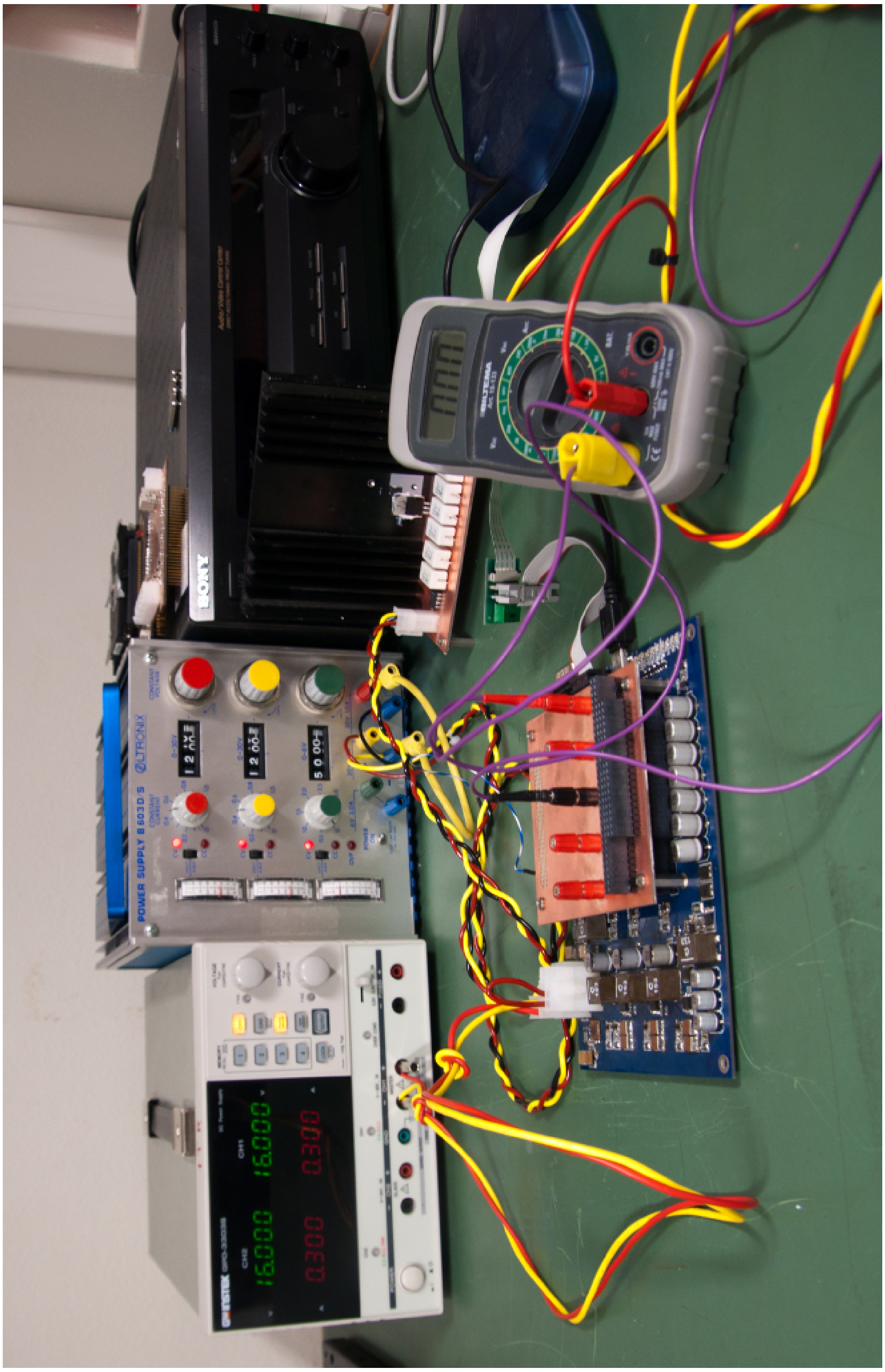
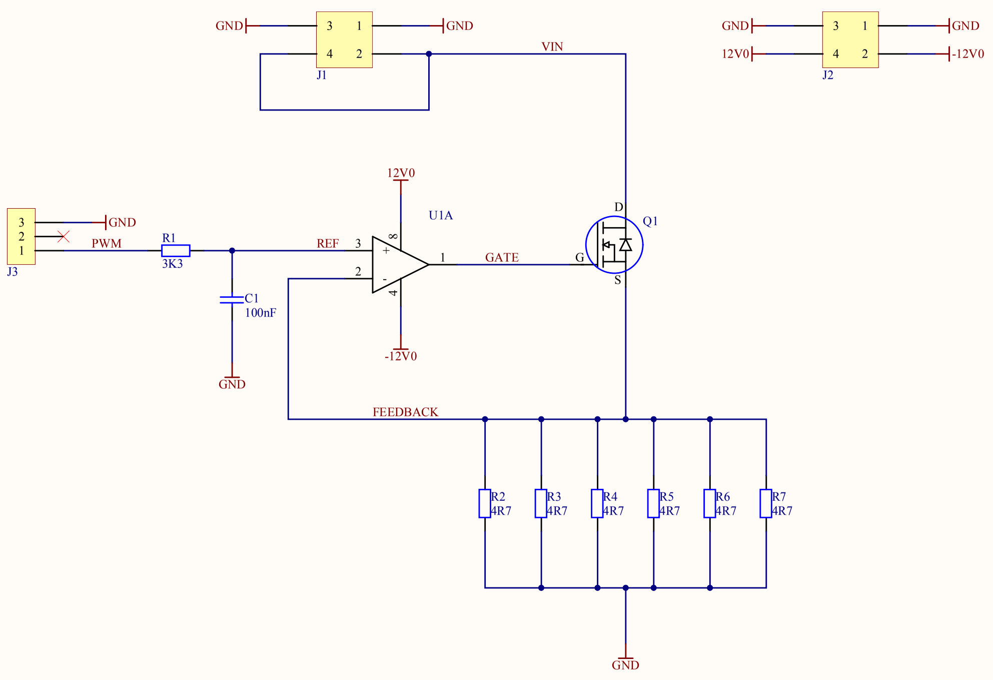


Figure C.7: Test Setup



Title		
Size A3	Number	Revision
Date:	04.06.2012	Sheet of
File:	C:\Users\..\source.SchDoc	Drawn By:

Designator	Quantity	MPN	Manufacturer	Value	Description	Digikey
D1, D2, D3, D4, D5, D6, D7, D8	8	APT1608SGC	Kingbright		LED 1.6X0.8MM 568NM GRN CLR SMD	754-1121-1-ND
J1	1	ESQ-120-12-G-D	Samtec		Non Stack Through	
J2	1	ESQT-130-03-G-Q-368	Samtec		Non Stack Through	
J3	1	ESQ-132-12-G-D	Samtec		Non Stack Through	
J4	1				Generic 100mil pin-header	
J5, J6, J7, J10, J11	5	39-29-1040	Molex		CONN HEADER 4POS 4.2MM VERT TIN	
J8, J9	2				Generic RA 100mil pin-header	
Q1, Q2, Q3, Q4	4	IRLML6402TRPBF	INTERNATIONAL RECTIFIER		MOSFET P-CH -20V -3.7A SOT23-3	
R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18	18			0R, 1K, 100	Generic resistor	
S1, S2, S3, S4	4	SO-1524-03-01-02-L	Samtec		15.24mm	

Appendix D

Software

D.1 Independent software projects

Some source where extracted from the Eurobot repository, and released as independent Open Source projects. The idea was that the development of this projects should be able to continue separate from the Eurobot project. This is most relevant for future students.

Extended Kalman Filter Library: extkalman
https://github.com/sperre/extkalman

A* Library: astar
https://github.com/sperre/astar

Library to handle metric units: units
https://github.com/smyrman/units

Goroutine dispatcher: rungo
https://github.com/smyrman/rungo

Tool to create virtual Go environments: goenv
https://github.com/smyrman/goenv

D.2 Dispatcher source code

Listing D.1: eurobot-ntnu.no/dispatcher/dispatcher.go

```go
package dispatcher
import (
	"sync"
)

// Implementations of this interface should periodically check the termc
// channel and return if there is something on the it.
type Interface interface {
	Run(termc <-chan bool)
}


type Routine struct {
	wg sync.WaitGroup
	termc chan bool
	I Interface

}

// Call I.Run() in a new goroutine, and return a dispatcher object that can be
// used to wait for it's completion, or to ask for it's termination.
func NewGoroutine(I Interface) *Routine {
	d := new(Routine)
	d.termc = make(chan bool, 1)
	d. I = I
	d.wg.Add(1)
	go d.run()
	return d
}

func (d *Routine) run() {
	defer d.wg.Done()
	d.I.Run(d.termc)
}

// If the termc channel of the Interface is not full, signal it. Block until
// the routine has ended. This call should be considered thread-safe. If the
// routine has already ended, this function returns at once.
func (d *Routine) Terminate() {
	select {
	default: // the termc channel is full
	case d.termc <- false: // termination signal sent
	}
	d.wg.Wait()
}

// Block until the routine has ended.
func (d *Routine) Wait() {
	d.wg.Wait()
}
```

D.3 Software line count

The following listings show a *cloc* count of the *robot*, *simulator* and *ntnu-eurobot.ntnu* folders:

Listing D.2: All packages

```
http://cloc.sourceforge.net v 1.56  T=3.0 s (37.3 files/s, 3398.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Go                              56            963            633           4567
C                               21            426            163           1348
Javascript                       4            102            120            570
C/C++ Header                    18            188            192            367
HTML                             4             24              0            161
CSS                              1             24              0            153
make                             6             46             28             72
Python                           2             14             11             22
-------------------------------------------------------------------------------
SUM:                           112           1787           1147           7260
-------------------------------------------------------------------------------
```

Listing D.3: Task-specific code (the robot and simulator package)

```
http://cloc.sourceforge.net v 1.56  T=1.0 s (55.0 files/s, 6197.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Go                              37            668            424           3492
Javascript                       4            102            120            570
C                                4             77             37            229
HTML                             4             24              0            161
CSS                              1             24              0            153
C/C++ Header                     4             17              0             43
make                             1             23              9             24
-------------------------------------------------------------------------------
SUM:                            55            935            590           4672
-------------------------------------------------------------------------------
```

Listing D.4: Libraries and drivers (the eurobot-ntnu.no folder)

```
http://cloc.sourceforge.net v 1.56  T=2.0 s (28.5 files/s, 1998.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                               17            349            126           1119
Go                              19            295            209           1075
C/C++ Header                    14            171            192            324
make                             5             23             19             48
Python                           2             14             11             22
-------------------------------------------------------------------------------
SUM:                            57            852            557           2588
-------------------------------------------------------------------------------
```

D.4 Benchmark script

Listing D.5: run_bench.bash

```
#!/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
source $DIR/gopath/sourceme.bash

# Test for $1:
if [[ "$1" == "" ]]; then
        echo "Usage:"
        echo " $0 RESULTNAME"
        exit
fi

# Define resultsdir
RDIR="$DIR/results.$1"
mkdir -p "$RDIR" || exit
cd "$RDIR" || exit

# Compile:
go test github.com/sperre/astar -c
go test github.com/humanfromearth/gopathfinding -c

# Create CPU profiles:
./astar.test -test.cpuprofile=astar.bench-Astar4-100x100.pprof -test.run=none -test.bench=Astar4
./gopathfinding.test -test.cpuprofile=gopathfinding.bench-Astar4-100x100.pprof -test.run=none -test.bench=Astar4

# Create profiling reports:
go tool pprof -text astar.test astar.bench-Astar4-100x100.pprof > astar.bench-Astar4-100x100.txt
go tool pprof -pdf astar.test astar.bench-Astar4-100x100.pprof > astar.bench-Astar4-100x100.pdf
go tool pprof -text gopathfinding.test gopathfinding.bench-Astar4-100x100.pprof > gopathfinding.bench-Astar4-100x100.txt
go tool pprof -pdf gopathfinding.test gopathfinding.bench-Astar4-100x100.pprof > gopathfinding.bench-Astar4-100x100.pdf

# Run benchmarks:
./astar.test -test.run=none -test.bench="Astar(4|8)" > astar.bench.out
./gopathfinding.test -test.run=none -test.bench="Astar(4|8)" > gopathfinding.bench.out

# Print
echo "####################"
echo "Human from earth:"
cat gopathfinding.bench.out
cat gopathfinding.bench-Astar4-100x100.txt

echo "####################"
echo " Eurobot−NTNU:"
cat astar.bench.out
cat astar.bench-Astar4-100x100.txt
```

Appendix E

Competition log

This appendix provides a log from each of the five matches played in La Ferté-Bernard, France 2012. Note that our robot is referred to as Loke, which is the name the NTNU-Eurobot robot was given this year. Videos of some of the matches are available on `http://insomnia.ed.ntnu.no/eurobot2012/videos`.

In addition, there is a possibility that multiple-camera productions of each match can be retrieved from the Eurobot organization `http://www.planete-sciences.org/robot/live/coupe2012/eurobot/`.

E.1 Series 1

Opponent team:	Robot Racing Team
Opponent robots:	1 robot
Game tactic:	Homologation
Video available:	No
Software version:	Git tag: N/A Git commit: unknown
Game log:	1. Loke drives out from starting area 2. Loke pushes gold bar to the ships deck 3. Loke pushes button for close bottle 4. All tasks completed, robot halted 5. Opponent robot halted due to some bug
Results:	19 points, victory
Opponent results:	4 points
Notes:	The team was called while taking a field trip into town. We had to meet up for the match on a very short notice, and ended up running the “homologation” game tactics by mistake.

E.2 Series 2

Opponent team:	Robo2BME
Opponent robots:	1 robot
Game tactic:	Passive
Video available:	No
Software version:	Git tag: eurobot12-round2 Git commit: 4dfa9ed73c831c98a0f00b1917d21b4b6a6d270d
Game log:	1. Loke drives out of the starting area 2. Loke pushes gold bar to the ships deck 3. Loke pushes the button of the close bottle 4. Loke heads for the center of the field in order to drive-by the closest totem pole 5. Loke drives by totem pole and pushes points to the ships deck 6. Loke drives against the center of the field in order to drive-by the closest totem pole from the other side 7. Loke drive-by the totem pole and manages to catch the gold bar with it's right wing 8. Before Loke gets a chance to drop of the points: Time's up!
Results:	25 points, victory
Opponent results:	12 point
Notes:	The opponent robot froze

E.3 Series 3

Opponent team:	Greenbirds
Opponent robots:	1 robot
Game tactic:	Passive
Video available:	Yes
Software version:	Git tag: N/A Git commit: unknown
Game log:	1. Loke drives out of the starting area 2. Loke pushes the button of the close bottle 3. Loke heads for the center of the field in order to drive-by the closest totem pole 4. The opponent block Loke's path 5. The opponent backs out 6. Loke drives by totem pole and pushes points to the ships deck 7. Loke drives against the center of the field in order to drive-by the closest totem pole from the other side 8. Time's up!
Results:	16 points, loss
Opponent results:	26 points
Notes:	The passive game tactics have been altered to not push the gold bar to the ships deck as the first task. The opponent robot drives much faster then us. We could probably adjust the driver to increase Loke's acceleration and max speed.

E.4 Series 4

Opponent team:	Ben Dover
Opponent robots:	1 robot
Game tactic:	Passive
Video available:	Yes
Software version:	Git tag: eurobot12-round4 Git commit: 6314313bec73ce13753bead6384c4a41f5aac38e
Game log:	1. Loke drives out of the starting area 2. Loke pushes the button of the close bottle 3. Loke heads for the center of the field in order to drive-by the closest totem pole 4. The opponent block Loke's path 5. Deadlock
Results:	15 points, victory
Opponent results:	2 points
Notes:	Loke's speed has been increased by approximately 2x compared to series 3. The deadlock could have been prevented if the "opponent avoidance" routine had been more intelligent, or if the area that was supposed to be free for Loke to chose the totem pole task had been defined larger in the passive tree.

E.5 Series 5

Opponent team:	UNICT-TEAM
Opponent robots:	2 robots
Game tactic:	Passive
Video available:	Yes
Software version:	Git tag: eurobot12-round5 Git commit: dd144914c673b838a819accb559e94e447eef339
Game log:	1. Loke drives out of the starting area 2. Loke altered course and crashed into the nearest totem pole 3. Are asks the referee to push the emergency button
Results:	0 points, forfeit
Opponent results:	51 points
Notes:	Before the match, all three beacon supports where measured to be approximately $0.5cm - 1cm$ to tall. The beacon support closest the robot was also not completely level. The screws where tightened to much to allow adjustment. From studying the video, it seems like the robot gets confused about it's heading - possible a bad measurement is received.

Appendix F

Poster

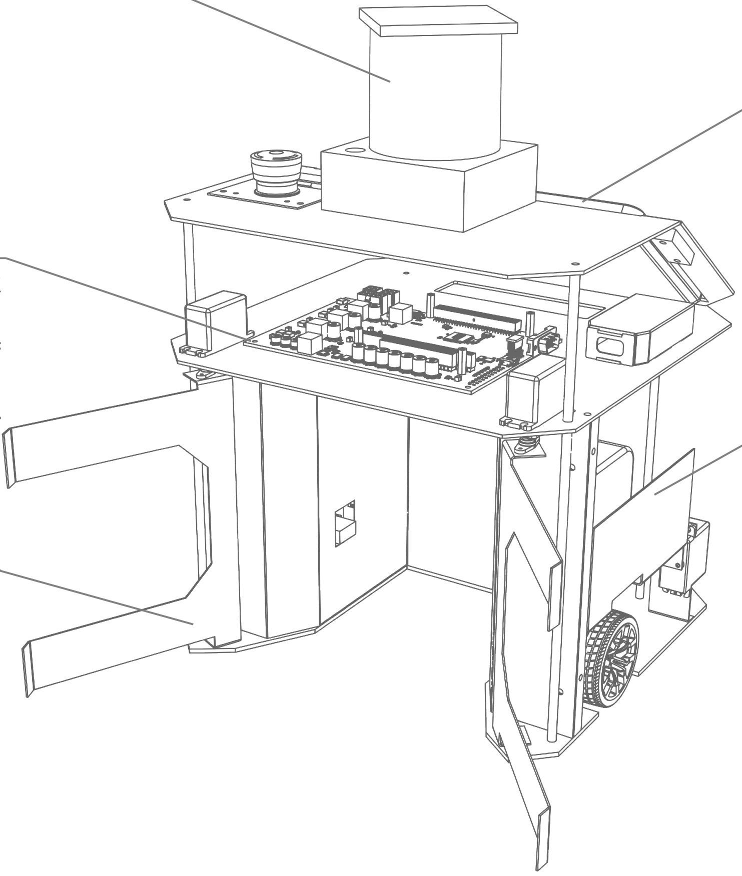
TREASURE ISLAND

Legend of Norway

EUROBOT 2012

LA FERTÉ BERNARD, FRANCE

Loke; Hardcore Trickster



Positioning

Two rotating parallel lasers and radio communication is used to discover the angle and distance of beacons. An extended kalman filter joins this with data from the motors' hallsensors.

Hardware stack

A "PC104-pluss" stack consiting of: A custom power-supply, capable of delivering 24V/12V/ 5V/3V3 @ 5A; a motor controller card; breakout-board.

Doors

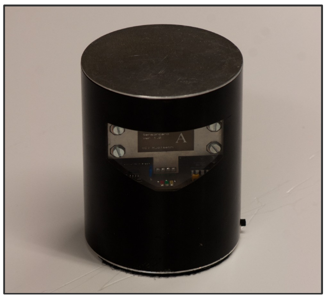
The doors are used to collect dublons. They are controlled by an Atmel ATmega-90CAN micro-controller.

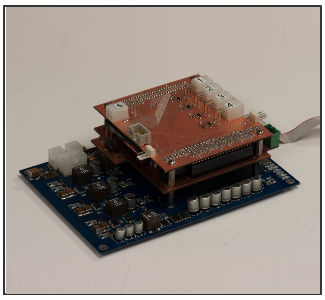
Tablet

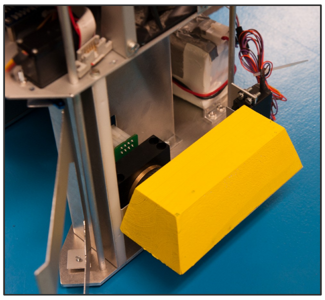
An HP Slate 2, running Arch Linux, commands the hardware over a CANopen bus. The main software is written in Go, drivers are written in C.

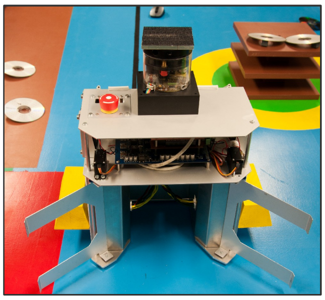
Wings

The wings are used to collect gold bars from the totem poles. They are controlled by an Atmel ATmega-90CAN micro-controller.









KONGSBERG

SINTEF
Cybernetics

NTNU
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Engineering Cybernetics

Appendix G

Digital attachments

The following files can be found ad digital attachments to this report. There is either a CD with the report, or the files can be downloaded from DAIM [1] where this report is published.

The digital attachments include all source code for the robot, all design files for produced electronics and production drawing for the mechanical parts.

- e2012_rules_en_final.pdf
- HP_slate2.pdf
- poster.pdf
- video.webm
- git repository
  eurobot-master
- hardware design
  battery
  bordkort
  current measure
  current source
  Enemy
  eurobot2012_topcard
  lasertower_power
  library
  robot_motherboard
  servo
  start
  stop_enkel
  topcard_template
- mechanical design

[1] url: http://daim.idi.ntnu.no