

University of Strathclyde
Department of Electronic and Electrical Engineering

Submitted for the Degree of MEng
in Electronic and Electrical Engineering
with European Studies

2009/2010

Paul MONSINJON

*Motion control, positioning system
and software design of a robot.*

Supervisor – Dr. David Harle

April 2010

Except where indicated to the contrary, all the work reported in this project is my own.

Signature _____ Date _____

Final Year Project

Eurobot 2010



Motion control, positioning system
and software design of a robot

23rd of April, 2010

Student: Paul MONSINJON

Advisor: Dr. David HARLE

Abstract

Robotics has become an important domain of research and engineering during the last decades. Combined with the constant growth of computation power, a wider domain of engineering topics have been enabled. It's now likely to see different kind of robotics contests all around the world.

In this context, Eurobot is a major actor of robotics events since 1998. This is an international amateur competition which occurs every year around mid-May. This report states a part of the accomplished work regarding the development of a robot for this competition. It's indeed a shared project where tasks have been explicitly divided between involved students.

The detailed work concerns mainly the robot's motion (the "legs"). This involves an accurate measure of moves and the ability to locate the robot at any time. Moreover, given this feedback, the robot must be able to go from a point to another. From the hardware to the software, the different parts have to be designed, tested and validated.

The results of this project are likely to be linked with the robot's performance during the contest. However, there are other parameters to take into account and a good way to evaluate the robot is still to run unitary tests on different sub-systems.

Finally, the entire work of this project can leads to further development (students project, research). One of the aim is also to enable different works on the same basis, that's why modularity has also be kept in mind during the development.

Acknowledgements

First of all, I would like to thank Dr. David Harle for his advice and important support throughout this year. It's also important for me to thank the university of Strathclyde for its financial support.

I also would like to thank the entire workshop staff for the numerous parts manufacturing and the great job accomplished.

I would also like to thank Schwarzer and ACP System for their sponsorship.

Table of Contents

1 Introduction.....	5
1.1 Report outline.....	5
1.2 Eurobot overview.....	5
1.3 Project objectives.....	7
1.4 Project scheduling.....	8
2 System design.....	10
2.1 Tools and means used.....	10
2.1.1 Linux operating system.....	10
2.1.2 Cadence Orcad and Cadsoft Eagle.....	10
2.1.3 gHDL and Xilinx ISE.....	10
2.1.4 Code::Blocks IDE.....	11
2.1.5 SVN repository.....	11
2.1.6 Website (wiki).....	11
2.1.7 Lab facilities.....	12
2.1.8 Workshop facilities.....	12
2.2 Armadeus board.....	12
2.2.1 APF27.....	13
2.2.2 APF27Dev.....	14
2.3 Motion control.....	15
2.3.1 Motion control theory.....	15
2.3.2 Hardware.....	21
2.3.3 VHDL design.....	26
2.3.4 Low-level software (C code).....	32
2.4 Beacon positioning system.....	35
2.4.1 System overview.....	35
2.4.2 Beacons theory.....	36
2.4.3 Hardware design.....	38
2.4.4 Software design.....	42

2.5 High-level software.....	44
2.5.1 Overview of main software.....	44
2.5.2 Strategy manager.....	45
2.5.3 CAN manager.....	45
2.5.4 Log manager.....	45
3 Testing and validation.....	47
3.1 PCB debugging.....	47
3.1.1 Test the tracks.....	47
3.1.2 Solder the vias.....	47
3.1.3 Solder the power supply components.....	48
3.1.4 Solder the rest bloc by bloc.....	48
3.2 VHDL simulation.....	48
3.3 Software debug and test.....	49
4 Conclusion and future work.....	50
4.1 General conclusion.....	50
4.2 Project planning.....	51
4.3 Eurobot event.....	51
4.4 Further work.....	51
5 Appendices.....	52
5.1 Initial Gantt Chart.....	52
5.2 Software makefile.....	53
5.3 Motors board schematic.....	56
5.4 Motors board masks.....	57
5.5 Motors board implementation layout.....	58
5.6 VHDL code of AD_to_XYT converter.....	59
5.7 TX Beacon schematic.....	66
5.8 TX Beacon masks.....	67

5.9 TX Beacon implementations.....	68
5.10 RX Beacon schematic.....	69
5.11 RX Beacon masks.....	70
5.12 RX Beacon implementations.....	71
5.13 Gtkviewer sample output.....	72
5.14 Gnuplot sample output.....	73

Illustration Index

Illustration 1.2.1: 3D View of the playground.....	6
Illustration 1.3.1: Projects objectives.....	7
Illustration 2.2.1: Armadeus connections.....	13
Illustration 2.2.2: APF27 board.....	14
Illustration 2.2.3: APF27Dev board.....	15
Illustration 2.3.1: Motion control localization principle.....	16
Illustration 2.3.2: DC-motor Laplace model.....	18
Illustration 2.3.3: PID Laplace model.....	19
Illustration 2.3.4: Position control loop.....	20
Illustration 2.3.5: Position and speed orders.....	21
Illustration 2.3.6: Motors board inputs/outputs.....	22
Illustration 2.3.7: Optocouplers polarization.....	23
Illustration 2.3.8: LMD18200 connection.....	24
Illustration 2.3.9: Incremental encoders signals.....	25
Illustration 2.3.10: AM26LV32 connection.....	25
Illustration 2.3.11: VHDL Architecture.....	26
Illustration 2.3.12: Encoder state machine diagram.....	28
Illustration 2.3.13: Round corners trajectory generation.....	34
Illustration 2.4.1: Beacons principle.....	36
Illustration 2.4.2: Beacon mathematical purpose.....	37
Illustration 2.4.3: Beacon IR Modulation.....	38
Illustration 2.4.4: TX Beacon emission stage.....	39
Illustration 2.4.5: TSOP connection.....	41
Illustration 2.4.6: RX Beacon receivers multiplexing.....	42
Illustration 2.5.1: Software diagram.....	44

1 Introduction

1.1 Report outline

This report is divided in different parts. This first one aims to describe the project's background and all the necessary knowledge. It is important to describe this environment in order to state clearly the objectives and what actually need to be done. With these objectives in mind, the different tasks have been identified and eventually split again in smaller issues. All this enables *a priori* scheduling, which has actually been revised a couple of times during the project.

The second part states a detailed account of the accomplished work. First, explanations about tools and their role in the project are given. The next section describes how the different tasks have been thought and designed. The report goes on to examine the features of the robot's motion control system, the beacon positioning system and the main software development.

The next part is a logical following regarding the designed systems. It explains the methodology to test and validate the different parts. Results are also shown.

Finally the latest sections summarize the project and emphasize the future work that could follow.

1.2 Eurobot overview

Eurobot is an amateur robotics contest. Started in 1998 by the association "Planete Sciences" (from France), the main objective over the years remains the cultural and technical promotion of science (especially robotics in this case). Last year, more than 200 teams participated to this contest which last 5 days during May. Teams are either students or independent clubs.

Being a team of Eurobot not only implies some technical knowledge. In fact, it's often that students/members have to find sponsorship and thus produce communication booklets. This is also about sharing knowledge and friendship between teams. It has been clearly seen that the robots are more and more sophisticated, this exchange and open-source mind contribute to a better global result and after all, to a better show.

Every year the contest's theme is different. However some common specifications remain, like dimensions which are roughly the same, the fact that matches last 90 seconds and two robots play at the same time.

This year, robots have to gather three different kind of fruits: tomatoes, ear of corn and oranges. Each one worth a different value (150, 250 and 300 points respectively). The robot that score the most win the match and get some additional points. As shown on the *Illustration 1.2.1*, each team owes a specific colour and has to put items in their dedicated basket. Another point concern the corns: some of them are actually fake (the ones in black) and screwed in the table.

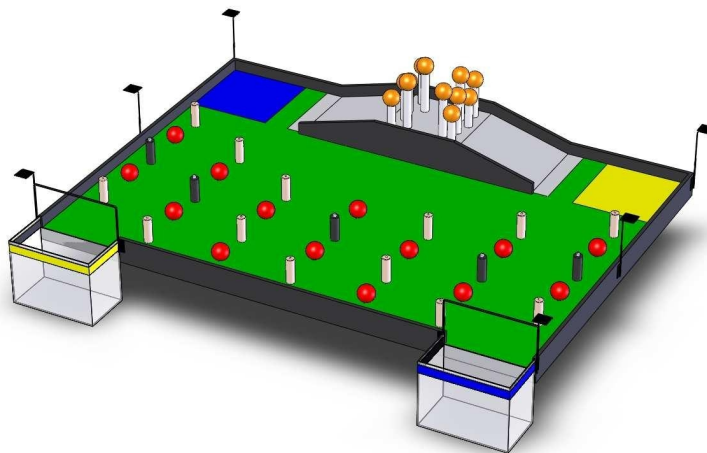


Illustration 1.2.1: 3D View of the playground

The contest is organized in three specific phases:

- During the first stage, the robot has to pass an approval test. It consists of a physical examination of the robot and a practical trial whereby at least one point must be scored.

- The qualification rounds: 5 matches, where each team try to score as much as possible.
- The final round, where the 16 best teams from the previous round compete in a classical knock-out basis. Exception is done for the final where it's played in two winning sets.

As a team of two students, we will represent the Strathclyde university. Sébastien Brulais and I have worked all this year on the robot.

1.3 Project objectives

Regarding to the contest rules and specifications, the different tasks can be chosen with a certain degree of freedom. However, my involvement in this project implies to work only on specific tasks (the ones highlighted). The *Illustration 1.3.1* shown below is an overall view of the different project's tasks. Mainly, the work concerns the robot's motion and positioning.

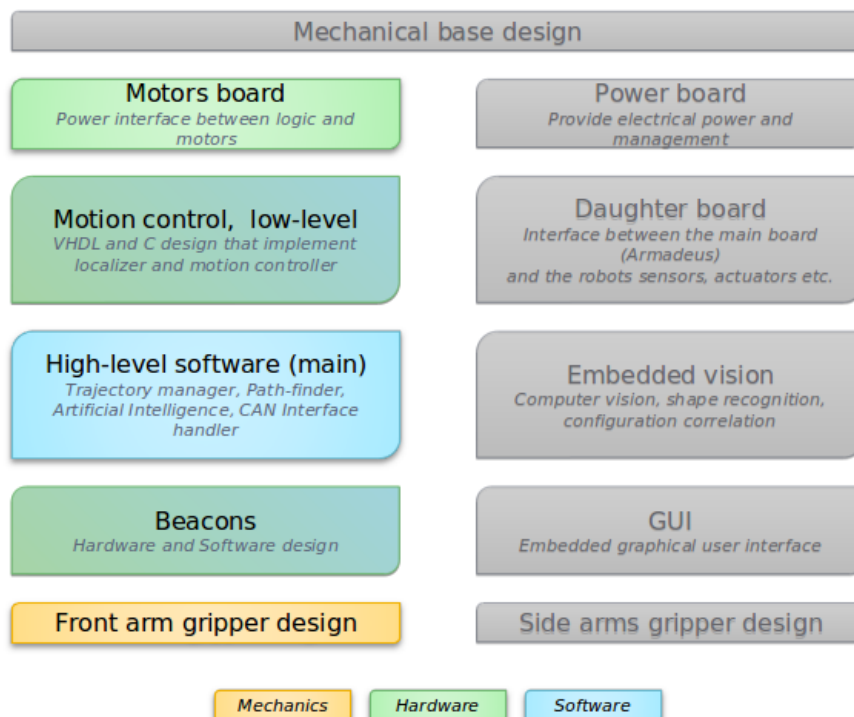


Illustration 1.3.1: Projects objectives

- The first stage in the conception of the robot concerns the **mechanical parts**. The robot uses two DC motors for its propulsion and two incremental encoders which are rotation sensors. These two components are mounted on 4 wheels: two for the propulsion (large width and strong grip) and the last two for the positioning (thin wheel to improve accuracy).
- The **motors board** allows any logical driver (microcontroller, processor, fpga...) to drive safely the motors independently.
- The **motion control** is insured by the Armadeus board (the main board, details given chapter 2.3). There are two main devices which work together, critical tasks are done in hardware in an FPGA while the high level operations are executed on an embedded linux OS.
- It's also very useful to know roughly the opponent's position. The **beacon positioning system** provide this capability and can also be used to have an absolute positioning system.

1.4 Project scheduling

When working on a long project it's often useful to plan the different objectives and milestones. Tasks length estimation is important and if this is correctly done, it allows to know at a specific moment if the project or a task is delayed, according to the Gantt chart.

The project's work has been originally split in three global tasks:

- The base design, were the most important objectives of the project belong to, it includes:
 - The motors board design
 - Hardware and low-level software of motion control
- Then, an "advanced" design section comes with either mandatory and optional tasks as:
 - High level software
 - Beacon positioning system (optional)
 - Robot's arm and actuator design (optional)

- The last section is called tests and simply indicates that unitary tests need to be run during the entire project's length. At the end, when all the robot should work, some test matches can be played.

The initial Gantt chart can be found in *Appendix 5.1*.

2 System design

2.1 Tools and means used

2.1.1 Linux operating system



Linux is a powerful operating system for any developer who wishes to use open source libraries, tools and other software. The cross-compilation tool (using arm-gcc) needed to compile the motherboard's kernel is not available for windows anyway.

2.1.2 Cadence Orcad and Cadsoft Eagle



When designing a board, there are basic steps to respect in order to obtain a PCB mask at the end. Orcad and Eagle are layout and schematics editors. Both of them can be used to:

- Draw the schematics
- Define components footprints and dimensions
- Place and route
- Produce Gerber files (optional but required for professional board manufacturing)

2.1.3 gHDL and Xilinx ISE



Designing a VHDL system often requires multiple steps. The first one consists to "compile" the code in order to achieve what is called the behavioural simulation. This doesn't take into account physical constraints, delays etc. but allows the designer to validate the functions. This can be done with gHDL, a free tool based on gcc. GtkViewer can then be used to navigate into timegraphs.

The second step requires to synthesize the design, given physical parameters (target, time constraints, pinout placements...). This is done by ISE, the IDE provided by Xilinx.

2.1.4 Code::Blocks IDE



Code::Blocks is a free and light IDE that can be used for different kind of projects. Here it's mainly used for C development, associated with makefiles. Different targets are defined:

- **x86**: Compilation for x86 platform (laptop), useful to test and debug some code without having the board
- **arm**: Cross-compilation for the arm processor, it calls another compiler and linker with different libraries
- **install**: Calls the arm rule and then upload the produced binary to the board (using ethernet link and scp).
- **update_fw**: Uploads the firmware binary (for the FPGA) to the board.

The *Appendix 5.2* contains the main makefile, used to set up the different rules.

2.1.5 SVN repository



SVN is a revision control system that enables development from different computers while keep data synchronized. It is actually an invaluable tool which also allows backups and restoration from previous versions easily.

In order to use this revision tool, the users can either do it in command line (can become irksome for big projects) or graphical tools like **Tortoise** (for windows) or **RabbitCVS** for linux.

2.1.6 Website (wiki)

Because this project is also open-source, it's often appreciated to have a website to explain system designs, tests and other details. A wiki is useful and helps to promote the work done over the Internet. There is also an SVN plugin which allows to browse the latest version and see associated comments from commits.

2.1.7 Lab facilities

The University of Strathclyde provides a lab with equipments for the project:

- A playground to run different test with the robot
- A computer connected to the network
- A power supply and an oscilloscope
- A soldering station

The lab is located in R3.53 (Cidcom lab) and is shared with other researchers and students.

2.1.8 Workshop facilities

The Electronic and Electrical Engineering department has two dedicated workshops: the mechanical workshop where different parts have been manufactured and the electronic workshop where PCB can be made. This is a great service and helps a lot regarding to the very practical aim of the project.

2.2 Armadeus board

One of the main interest of this project concerns the main board. Armadeus systems is a company that promote open source electronic designs and software. Their main product at the moment is the **APF27** and can be either used in a standalone version (it's then up to the designer to build a board to plug it in) or with its dedicated development board: the **APF27Dev**.

The combination of these two boards is the motherboard of the robot. In order to interconnect dedicated components (sensors, actuators etc.), the **APF27Dev** is connected to an interface board (called daughter board). This board has the same size and is stacked on the **APF27Dev**. The general overview diagram is given by the *Illustration 2.2.1* (the power board does not appear on the diagram).

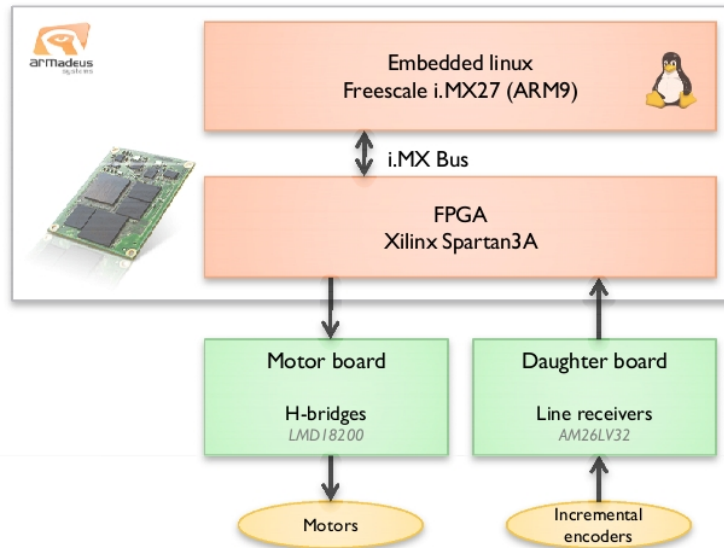


Illustration 2.2.1: Armadeus connections

More informations and details can be found on the Armadeus systems website:

<http://www.armadeus.com>

The company is also an association, and a well documented wiki is available at

<http://www.armadeus.com/wiki/>

2.2.1 APF27

For the project's purpose, an APF27 and the development board (in full version) are used. Bellow are the APF27 specifications and the *Illustration 2.2.2* is a picture of the board.

- Processor: Freescale 400MHz i.MX27 (ARM929)
- RAM: 128MB of 32bits Mobile DDR
- Flash: 256MB of 32bits Mobile NAND
- FPGA coprocessor: Xilinx Spartan3A, 200k gates and up to 62 GPIOs
- Operating system: Linux 2.6.27



Illustration 2.2.2: APF27 board

2.2.2 APF27Dev

The development board is bigger (160x100mm) and embedded a lot of interfaces/features such as:

- On-board power supply, input from 5 to 16V
- RS232 port (serial console)
- Ethernet port
- USB 2.0 controller, two ports
- CAN interface controller
- ADC (7 channels, 10 bits)
- DAC (2 channels, 10 bits)
- MicroSD card slot
- RTC with backup battery
- Stereo audio controller
- Video output (HDMI port, embedded LCD connector)

A picture of the APF27Dev is shown on *Illustration 2.2.3*.



Illustration 2.2.3: APF27Dev board

2.3 Motion control

The motion control design is the major development of the entire project. It has multiple aspects and requires both hardware and software engineering. There are two main functions achieved by the motion controller:

- The localization, which means given the incremental encoders signals, be able to compute the position, speeds and acceleration of the robot.
- The control itself, which drive the motors in a smooth and accurate way.

2.3.1 Motion control theory

The first thing to know in order to design the main functions is the robot's structure and mechanics. As stated previously there are two motors and two encoders with their associated wheels. They are all in the same axis but completely independent. The robot is symmetric (left/right axis).

2.3.1.1 Localization

The incremental encoders give a position value. The idea of the localization is to compute it (i.e., calculate the position and other motion parameters) at a specific rate (not too slow but not too fast, for example, 100Hz). Thus the position will be sampled and it'll correspond to a small move from each wheel, as shown on *Illustration 2.3.1*. In fact it also corresponds to a slight motion in the X/Y plane, called dX and dY .

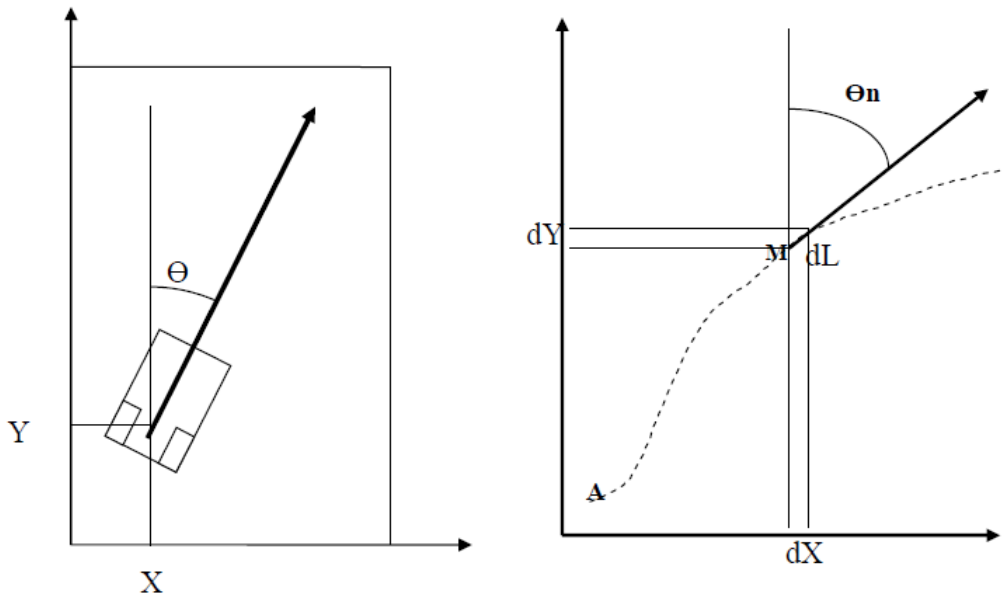


Illustration 2.3.1: Motion control localization principle

Computing these variations can be done with the dL and dR values, which are respectively the small variations from the left and right wheels. The linear motion is called dd and the angle formed by the robot is θ_n (given previous angle θ_0).

The formulas are then as follow:

$$dd = \frac{dR + dL}{2}$$

$$\theta_n = \theta_0 + (dR - dL)$$

The trajectory between two consecutive points is approximated by a straight line.

This is not enough to know the X and Y motion. The linear position $L_n = dd$ can be derived, so that it gives the speed of the robot:

$$V_n = L_n - L_{n-1}$$

then,

$$\begin{aligned} dX &= V_n \cdot \cos(\theta_n) \\ dY &= -V_n \cdot \sin(\theta_n) \end{aligned}$$

If now X and Y are integrated, we get back to a position:

$$\begin{aligned} X_n &= X_{n-1} + dX \\ Y_n &= Y_{n-1} + dY \end{aligned}$$

This is basically the function insured by the localizer. However, computing a sine and a cosine at a high rate in an embedded system can lead to CPU load issue.

2.3.1.2 Control

The final aim is to make the robot move according to a specific order. This function is insured by the controller, given the localization feedback. There are two ways of doing that:

- Controlling each wheel (left or right) independently. So for example, to go straight the order should be the same on each controller.
- Using a polar controller: there is still two controllers but one represent the translation and the other one the orientation.

The first thing to do is considering the DC-motor model. Physics law give the bellow equations:

$$\begin{aligned} u(t) &= e(t) + R \cdot i(t) + L \frac{di(t)}{dt} \\ e(t) &= K_e \cdot \omega(t) \\ T_m(t) &= K_t \cdot i(t) = J_t \cdot \frac{d\omega(t)}{dt} + Tr(t) \end{aligned}$$

where:

- t is the time variable
- u is the applied voltage
- e is the electromotive force

- i is the intensity through the motor
- ω is the rotation speed
- T_m is the motor's generated torque
- T_r is the resistive torque
- K_e is the speed constant
- K_t is the torque constant

The Laplace model is then given by the *Illustration 2.3.2*:

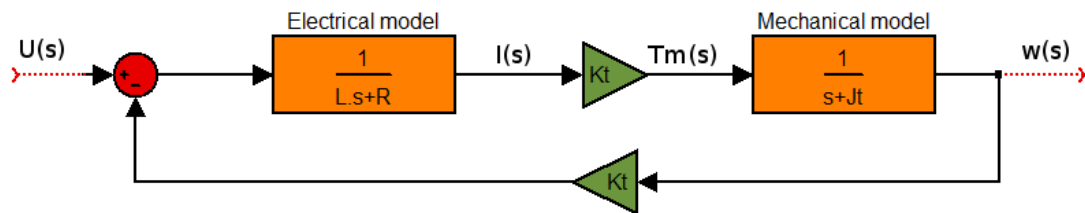


Illustration 2.3.2: DC-motor Laplace model

note: the resistive Torque doesn't appear here, but it acts on $T_m(s)$ (simple subtraction).

The output $\Omega(s)$ is the speed of the motor.

The overall equation is then:

$$\Omega(s) = \frac{K_t U(s) - (R + L \cdot s) \cdot T_r(s)}{K_e \cdot K_t + R \cdot J_t \cdot s + L \cdot J_t \cdot s^2}$$

If $L \ll R$, this can be simplified to a 1st order equation (its actually the case, normal values for a DC-motor are $R = 2\Omega$ and $L = 0.0002H$).

Now this system can be inserted in a control loop. As explained before, there are conversions between Left and Right to Delta and Alpha domains. The corrector is a parallel PID, which stands for Proportional Integral Derivative. The filter itself contains three multiplications (P, I and D coefficients) and a sum. The *Illustration 2.3.3* shows the Laplace model of this filter.

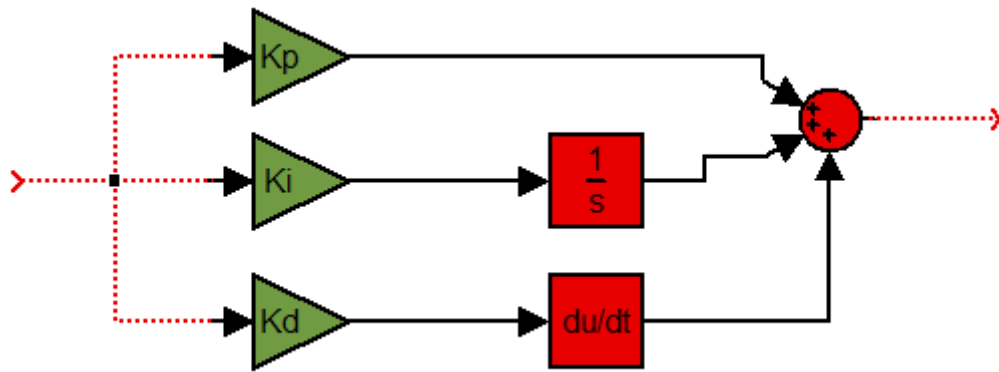


Illustration 2.3.3: PID Laplace model

Finally, these correctors are inserted in the global loop (*Illustration 2.3.4*)

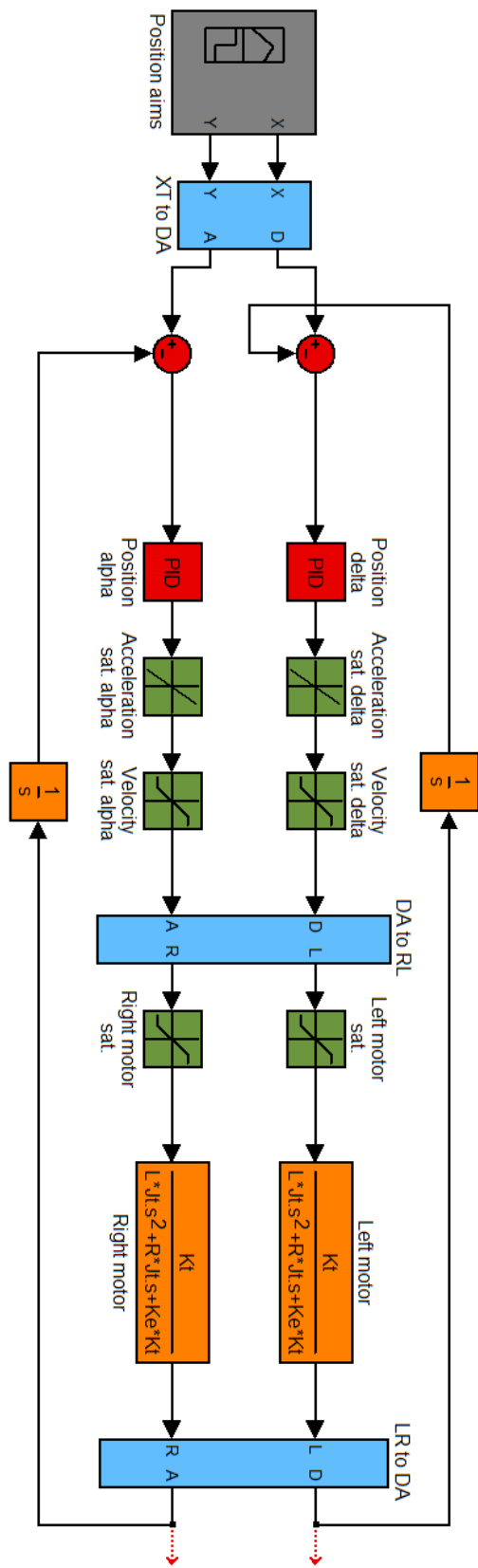


Illustration 2.3.4: Position control loop

The last stage consists to give a proper input to the controller. For instance, the robot cannot physically move between two points in a null time. It means steps as position orders aren't good. If ramps are used it's better, but it means the robot has to change it's speed in a null time... which is equivalent to an infinite acceleration.

This finally leads to a well know velocity profile:

- Constant acceleration phase
- Maximum speed is reached, move at a constant speed
- Constant deceleration

The resulting position order is then the integration of this trapezoid, i.e. a square polynomial function. This is shown in *Illustration 2.3.5* (speed order in purple, position order in orange)

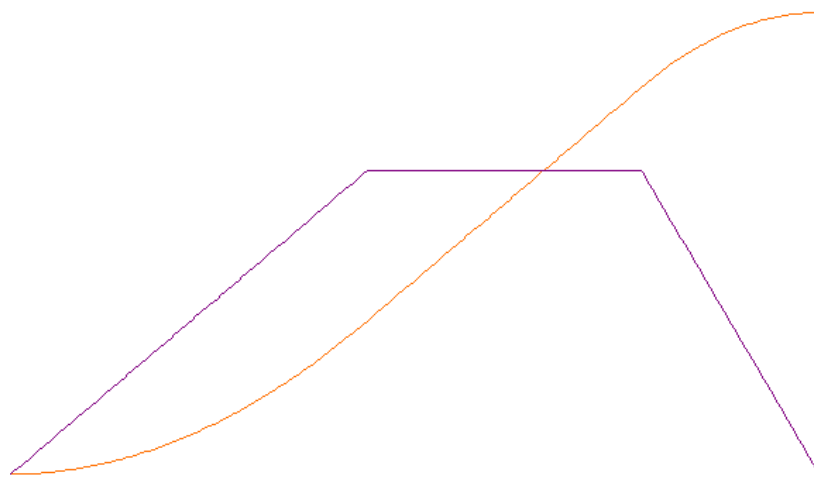




Illustration 2.3.5: Position and speed orders

This could also be directly implemented by using two consecutive PID: one for the position which give orders to the speed PID. Maximum accelerations and velocities are implemented by using saturation blocs between these two PIDs.

2.3.2 Hardware

After designing and simulating the control process, a real robot needs a practical implementation. This part describes the hardware used for it. The table shown bellow gives details about the motors and the incremental encoders used:

<p>Motors: MFA970D161</p> 	<ul style="list-style-type: none"> • Nominal voltage: 12V • No load speed: 15800 rpm • No load current: 0.52A • Maximum torque: 154.4 g.cm • Output power: 21.2W • Efficiency: 62% • Reduction ratio: 1:16
<p>Incremental encoders: Kübler type 2400</p> 	<ul style="list-style-type: none"> • Power supply: 5-24V • Maximum speed: 12000 rpm • #Pulse per revolution: 1024 • Current consumption: 50mA

The motors are connected to the **motor board** and the incremental encoders to the **daughter board**.

2.3.2.1 Motors board

The role of this board is to provide a safe and reliable interface between driving logic device and motors. It has to meet the motors requirement (see above). The **Illustration 2.3.6** shows motors board inputs/outputs.

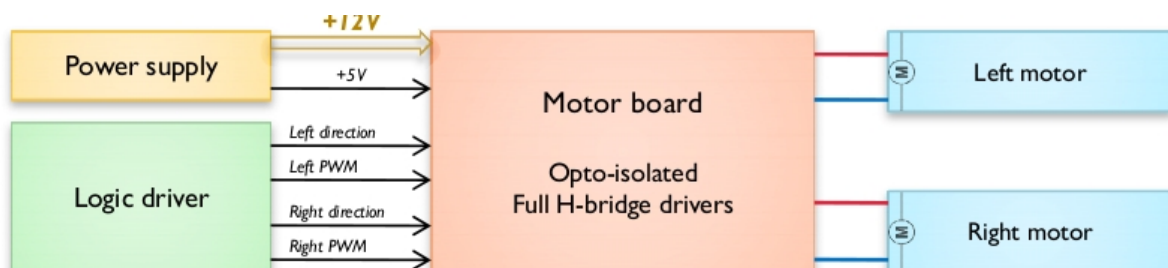


Illustration 2.3.6: Motors board inputs/outputs

The protection and voltage translation stage is done by using **optocouplers**. There are small devices that contain a diode and a photo-transistor. When the diode emits light, the photo-transistor becomes saturated and the logic level at its output is pulled high. Otherwise, the transistor is blocked and the logic level is low. The main advantage is to not have any electrical link between the input and the output. So if a short-circuit happens, the optocouplers might be destroyed but not the logic behind it.

The *Illustration 2.3.7* presents how optocouplers are polarized on the motor board. The reference used is FOD617 from Fairchild semiconductor.

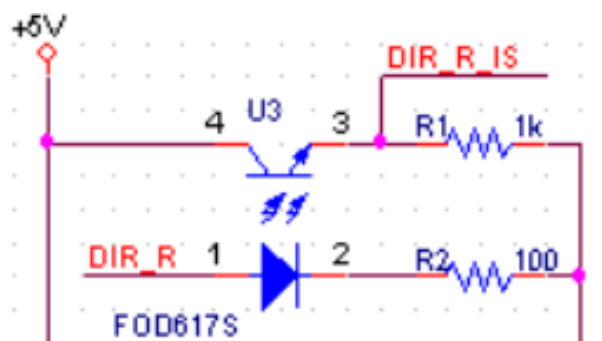


Illustration 2.3.7: Optocouplers polarization

The main devices that drive the motors are two full H-bridge drivers. This devices allow the motor to turn backward or forward at different speeds (controlled by using a Pulse Width Modulation – PWM). The chosen devices are the **LMD18200** from National Semiconductor. They can drive 3A and have a current sensing output. Two capacitors are simply connected between outputs and bootstraps pins, as shown on the *Illustration 2.3.8*.

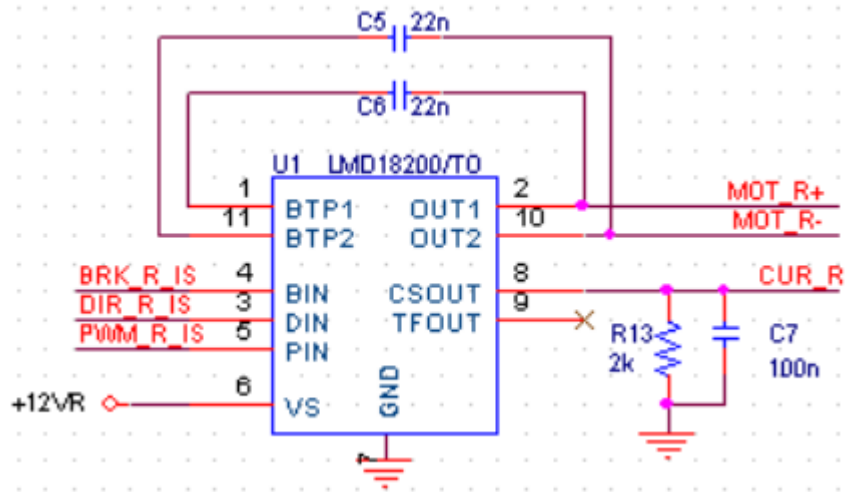


Illustration 2.3.8: LMD18200 connection

The entire schematic is available in **Appendix 5.3**. The PCB masks in **Appendix 5.4** and the implementation layout in **Appendix 5.5**.

2.3.2.2 Incremental encoder interface

The incremental encoders sensors provide three main signals:

- A line
- B line
- 0 line

The A and B lines are the main signals. Their phases are different of +/- 90 degrees. Depending on which signal raise the first, it indicates whether the encoder turns clockwise or counter-clockwise. The rate at which each signal trigger depends on the rotation speed, each raise of a line corresponds to 1 tick. The robot's encoders have 1024 ticks per rotation. The line 0 triggers on each full rotation. This can be useful to correct a drift or simply have a rough idea of the rotation speed.

The **Illustration 2.3.9** shows the principle of the A and B lines:

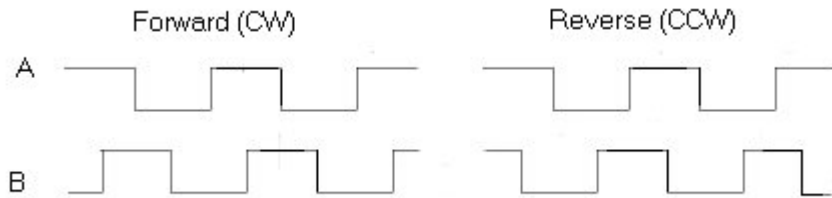


Illustration 2.3.9: Incremental encoders signals

In order to remove potential jitter, each line (A, B and 0) is a differential pairs. This means there are 6 logic signals per encoder. The encoders also work with 5V levels, so there is a need to translate this voltage. The line-receiving decoding and voltage translation can be done with the same component from Texas Instrument: the **AM26LV32**. The 0 line is not used. This device is used in the daughter board, the pinout is shown in *Illustration 2.3.10*.

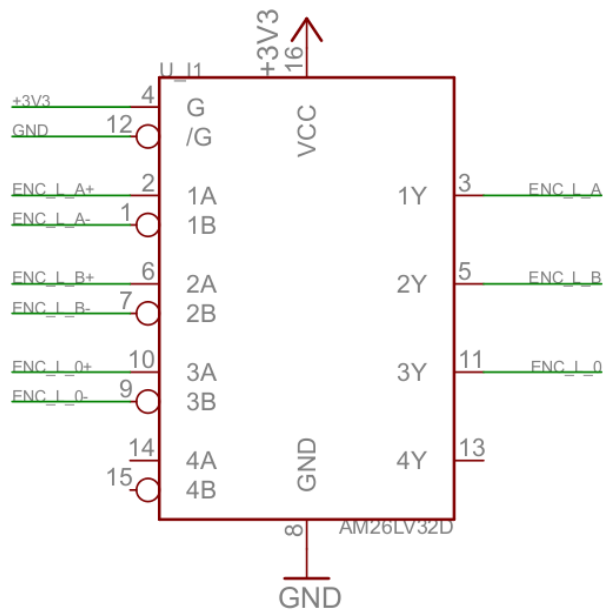


Illustration 2.3.10: AM26LV32 connection

2.3.3 VHDL design

The very first stage of the motion control is to decode the signals provided by the incremental encoders and output Pulse Width Modulated signals to control the motors speeds. This is done in VHDL and implemented in the APF27 FPGA (programmable logic device). Further features are then build upon this like speed computation, X/Y localizer, PID controllers etc.

In order to communicate properly with the CPU, the IMX bus has to be used. Signals are first of all wrapped into wishbone signals. The hierarchical structure contains components who owe a specific address range. As the address bus is 10bits wide, it has been chosen to use a maximum of 8 components with 128 registers each. The motion controller is a component (certainly the biggest!) but other components can be added (mezzanine control for instance). This principle of "virtual components" is fully detailed with a led and button example at

http://www.armadeus.com/wiki/index.php?title=A_simple_design_with_Wishbone_bus

The *Illustration 2.3.11* is a hierarchical view of FPGA entities. The motion controller component is detailed in the next section.

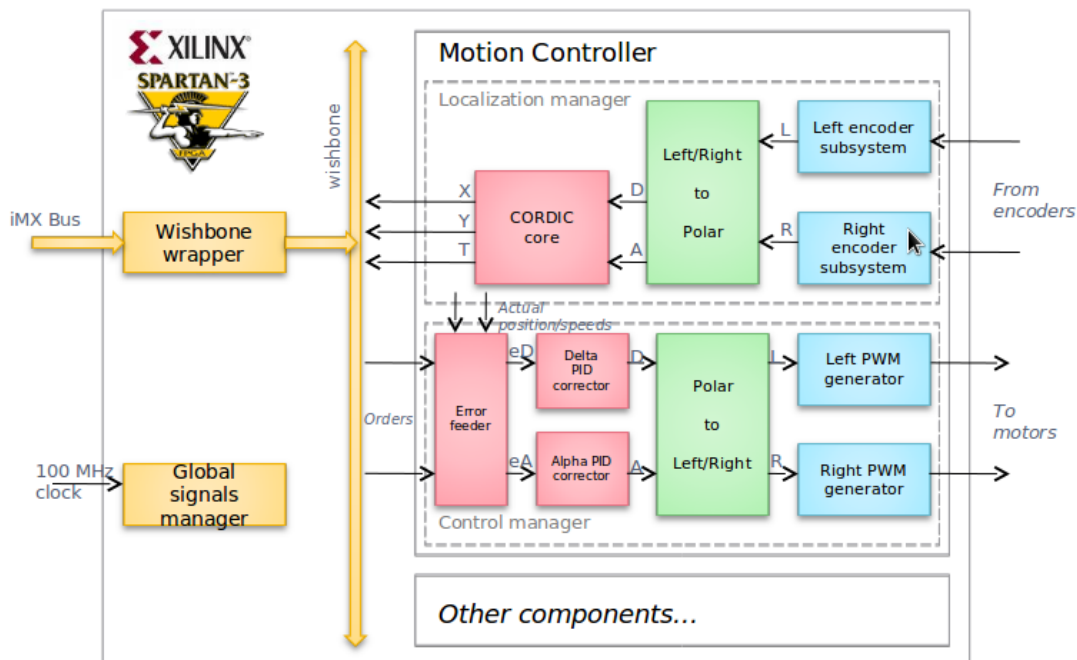


Illustration 2.3.11: VHDL Architecture

2.3.3.1 Encoder subsystem

The aim of this subsystem is to give the encoder speed (a signed value on 16 bits), given the A and B signals. The system again, can be split into different functions.

a Digital filter

The first stage consists to filter the incoming signals. The filter should be able to remove noise peaks. Due to the physical aspects of the robot, the A and B signals cannot trigger faster than a specific rate. For instance, if the encoder wheel's size is $d = 60\text{mm}$, the maximum speed is $V = 2\text{m/s}$ and the number of pulse per revolution is $R = 1024$, 1 revolution of the wheel corresponds to :

$$\pi \cdot d \Leftrightarrow R \text{ ticks}$$

So there are

$$\alpha = \frac{R}{\pi \cdot d} \sim 5433 \text{ ticks/meter}$$

Now the robot moves at V m/s, there are Vt ticks per seconds:

$$Vt = V \cdot \alpha \sim 10865 \text{ ticks/sec}$$

Finally, it corresponds to a minimum width of $46\mu\text{s}$. The filter has to remove any spike smaller than that.

The method used remains quite simple to implement and requires few logic elements. The filter uses a shift register, its input is fed by the actual signals (A, B). The filtered output is stable (high or low) if all the register values are the same. Otherwise, the values remain unchanged. The sampling rate determines the filter latency and its capability to reject noise. For instance, in order to filter less than $46\mu\text{s}$ (approximated to $40\mu\text{s}$) wide peaks, the sampling rate can be 250kHz (period = $5\mu\text{s}$) and the filter 4bits wide.

b Quadrature decoder

The second stage uses both input signals (A and B) to produce two outputs: **COUNT** and **UP**. **COUNT** is a counter output which triggers at each change (whether it is forward or backward). **UP** simply indicates if the rotation is clockwise or counter clockwise.

This stage is achieved with a 4 states machine describe in *Illustration 2.3.12* .

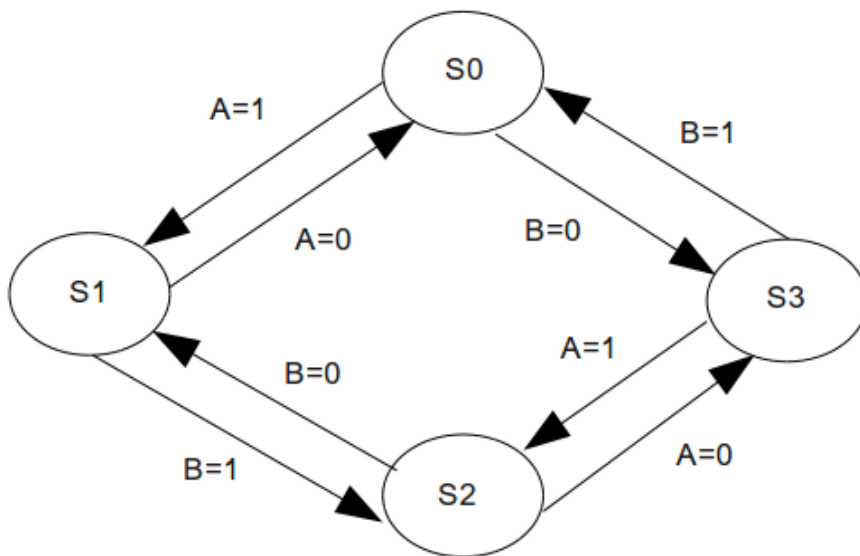


Illustration 2.3.12: Encoder state machine diagram

The combinatorial stage of outputs is given by the following truth table:

A	B	State	COUNT	UP
1	X	S0	1	1
X	1	S0	1	0
X	1	S1	1	1
0	X	S1	1	0
0	X	S2	1	1
X	0	S2	1	0
X	0	S3	1	1
1	X	S3	1	0
X	X	X	0	X

c Counter

The output provided by the quadrature decoder can be directly connected to a counter. The increment/decrement input is UP and the clock rate is COUNT. In order to compute the speed, this counter is reset at a specific period. Thus, the value contained by the counter is directly the number of ticks between two resets, it corresponds to a basic digital derivation.

However, this rate cannot be chosen randomly, it has to be in a certain range.

- If it's too slow, the speed value will be fed to the controller slowly and the physical motion between two computation might be important.
- If it's too quick, small variations may not be detected.

A common value of reset rate is around 100Hz. Thus, at the maximum speed, the speed counter contains around 109 ticks (maximum speed is around 10,865 ticks).

2.3.3.2 Left/Right to Alpha/Delta converter

The next stage is to convert these speeds into delta (linear translation) and alpha (orientation) values. The logical bloc uses signed values and compute the following formula:

$$D = \frac{L+R}{2}$$
$$A = R-L$$

2.3.3.3 Alpha/Delta to X/Y/T and Cordic core

The last stage of the localizer is the bloc that compute actual positions in X/Y domain. In order to do that, it's needed to compute trigonometric operations, this is enabled with the Cordic core (which is provided by Xilinx as a free IP).

However there are few things to compute before, it's mainly about unit conversions. The Cordic core needs radian value but the Left/Right to Alpha/Delta converter gives only a relative angle without any units.

The first thing to do is to compute the angle in radian, according to the following formula:

$$A[\text{rad}] = \frac{A}{I}$$

Where **A** is the angle and **I** is the interaxial value (distance between the two wheels). **A** and **I** have to be in the same unit (ticks), so that the result is in radian. This division is done by an hardware divider and the result is fed to the cordic core.

The Cordic core gives the values of **cos(A[rad])** and **sin(A[rad])**, the last stage is to multiply it by the linear speed **Vn** which is done by embedded multipliers.

The entity also has to manage the different latency and insure signals integrity. A specific process fetches **dX** and **dY** only when the outputs are valid. The last thing to do is insure that angle values are bounded between **-Pi** and **+Pi** (chosen representation). This is basically done by an hardware modulo.

The whole VHDL code of this entity is given in *Appendix 5.6*.

2.3.3.4 PWM generator

This entity is used to produce a Pulse Width Modulation signal which control the motors speeds. This is simply done by a counter which compares the its value to the desired speed and set the output to 0/1, whether the counter's value is above or below the threshold.

Some specific features have also been added, the PWM is actually signed and also delivers a direction output depending on the speed's sign. This output can be inverted (using a XOR gate) with the **inverted** input. This input is useful because the motors are mounted "head to head" and opposite voltage need to be applied to make the robot goes straight.

2.3.3.5 Alpha/Delta to Left/Right converter

This entity is basically the opposite of the Left/Right to Alpha/Delta entity. It computes the following formula:

$$\begin{aligned}R &= D - A \\L &= D + A\end{aligned}$$

The outputs are fed to the PWM blocs.

2.3.3.6 Error feeder

The error feeder is the link between the localization manager and the control manager. It uses orders (from the CPU) as reference, and computes the errors between those and the feedbacks. Outputs are positions and speeds in alpha/delta domain. These outputs are then used by the PID controllers.

2.3.3.7 PID Controllers

This is the heart of the controllers. As explained in the theory section, a PID bloc is basically:

- 3 multipliers, between the error and associated gains (K_p , K_i , K_d)
- An accumulator (integral effect)
- A unitary delay with a subtracter (derivative effect)
- A three inputs adder

The resources needed can start to be heavy depending on the resolution. Internally, registers are 24 bits wide but only the 16 MSB can be retrieved by the CPU.

2.3.4 Low-level software (C code)

The low-level software of the motion controller provides the link between higher software (A.I., obstacle avoidance etc.) and the FPGA registers themselves. The motion control is composed of 3 modules:

- FPGA module which is the proper driver to read/write values
- Motion Control driver, which computes the values that need to be written in the registers
- Trajectory Manager module, used to build paths that are then given to the motion control module. These paths need waypoints points (logical point where the robot has to go) which are fed by the higher levels (obstacle avoidance and pathfinder).

2.3.4.1 FPGA module

The FPGA is mapped at a specific address and can be acceded like a memory. It's achieved by using pointers and the C code is relatively compact. Here is an extract from the `fpga.c` module:

```
FILE* ffpga = NULL;
int* ptr_fpga;

int openFpga(void)
```

```

{
    if((ffpga = (FILE*) open(FPGA_DEVICE, O_RDWR|O_SYNC)) == NULL)
    {
        printf("[INT] Error: Couldn't open FPGA device\n");
        return 0;
    }

    ptr_fpga = mmap (0, 8192, PROT_READ|PROT_WRITE, MAP_SHARED,
                    (int) ffpga, FPGA_BASE_ADDR);

    return -1;
}

```

Function used to open the FPGA device

Write or read to a specific register is then straightforward and dedicated functions (*fpgaReadComponent()* and *fpgaWriteComponent()*) have been written.

```

void fpgaWrite16(int address, short value)
*(short*)(ptr_fpga+(address)) = (short)value;

```

Write a 16 bits value

```

short fpgaRead16(int address)
return *(short*)(ptr_fpga+(address));

```

Read a 16 bits value

2.3.4.2 *Motion control module*

This module is linked to the trajectory manager. The main function of the motion control module is *moveFollowPath*, which takes a path in parameter. This function write appropriate values in the FPGA registers in order to follow the computed path. The written values take into account the possible drift of the robot and then correct it accordingly.

2.3.4.3 *Trajectory manager module*

This is certainly the heavier module in terms of computation. This module provide different functions to compute a path, given a set of waypoints. The first and simpler "path rule" consist of two basic steps:

- Turn on place to align the robot on the next waypoint
- Go straight until the waypoint is reached

When the last point is reached, the robot can turn on place a last time to be on a specific direction. It's easy to see that if there are N waypoints, the total number of points/moves will be $2N-1$ (1 turn and 1 straight path per waypoint except for the 1st point). The function is called *pathStraight()* and takes two parameters: the waypoints array and its size. The returned path's points are allocated dynamically and then need to be free once the path is used.

The second function is an enhancement of the previous one, instead of turning on place at each waypoint the robot can make a complex move (rotation+translation) in order to "round" the corners. The function *pathStraightRoundCorners()* takes the same parameters as previously plus the corner size radius. The two points computed for the same waypoints are now shifted from the corner size radius along the previous and next direction respectively, as shown on *Illustration 2.3.13*.

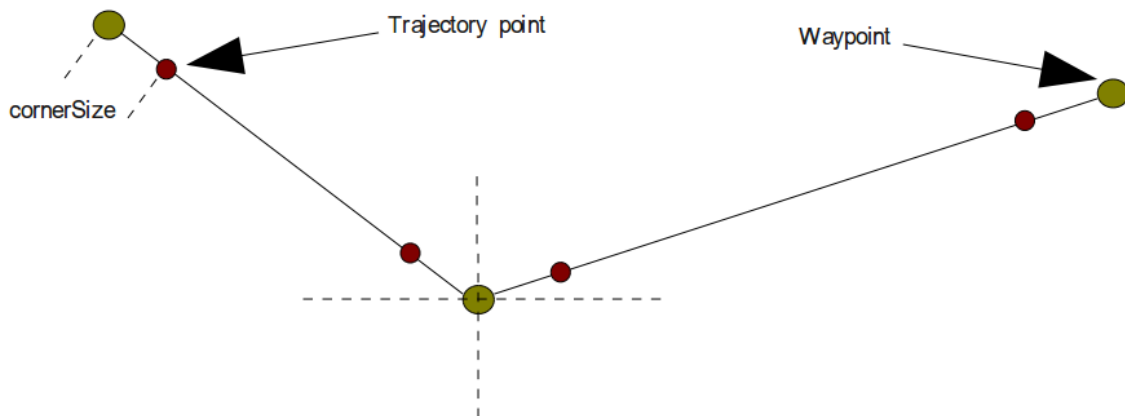


Illustration 2.3.13: Round corners trajectory generation

2.4 Beacon positioning system

2.4.1 System overview

A beacon system is always a great help for the Artificial Intelligence in the Eurobot contest. It enables two main features:

- Absolute positioning, which can be very useful when its accuracy starts to be high. Unfortunately, only few kind of beacons are enough precise to help the dead-reckoning system.
- Opponent's position detection; even if the value of the position is not very accurate (like 5 or 10 cm), it is always better than nothing! Macro decision can be taken to avoid the other robot efficiently.

There are different kind of beacons, each one of those have pros and cons. The table above gives more details about technological issues.

Beacon type	Pros	Cons
Laser	<ul style="list-style-type: none">• Very good accuracy• Robust to noise	<ul style="list-style-type: none">• Expensive• Can be dangerous• Hard to find class I lasers• Low flexibility
Infrared	<ul style="list-style-type: none">• Moderate accuracy• Flexibility• Inexpensive	<ul style="list-style-type: none">• Can be subject to noise easily
Ultrasonic	<ul style="list-style-type: none">• Moderate accuracy• Moderate robustness to noise• Flexibility	<ul style="list-style-type: none">• Good sensors are expensive

In this project, infrared beacons have been chosen. They don't especially give a very accurate position but it's enough regarding the objectives. The cost and the safety issues are also important.

Three fixed beacons can be set up around the table at known positions. They act like lighthouse and send periodically frames. These frames are decoded by the receiver, placed on the top of the opponent. Knowing the three incoming angle allows to compute the position. This position is then sent to the robot using an Xbee module. It's also possible that the robot (not the opponent) compute the angle (the board is exactly the same). The *Illustration 2.4.1* shows the beacons principle.

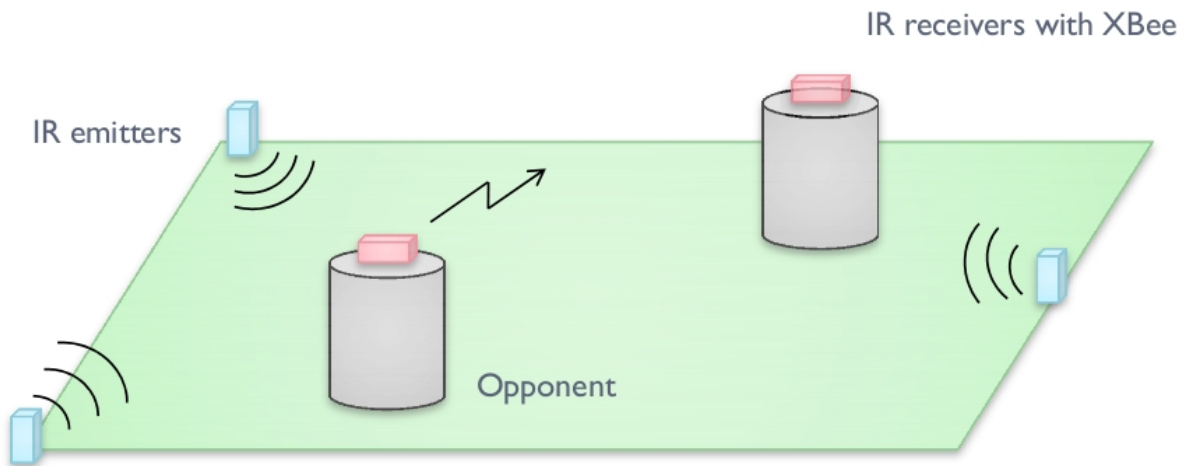


Illustration 2.4.1: Beacons principle

2.4.2 Beacons theory

In this section, it's assumed that the three incoming angle are known. The sine theorem used in the three triangles enable the knowledge of x and y .

The *Illustration 2.4.2* shows the different angles and names used.

The knowledge of i, j, k, p, m, o is used to determine the distances a, b, c . It finally leads to x, y . Here are some formulas, there are different ways to obtain x and y :

$$i = \cos^{-1} \left(\frac{\left(\left(\frac{v}{\sin(\gamma)} \right) * \cos \left(r - \gamma + \frac{\pi}{2} \right) + \left(\frac{w}{\sin(\alpha)} \right) * \cos(\alpha - \pi) \right)^2}{\left(\left(\frac{v}{\sin(\gamma)} \right) * \cos \left(r - \gamma + \frac{\pi}{2} \right) + \left(\frac{w}{\sin(\alpha)} \right) * \cos(\alpha - \pi) \right)^2 + \left(\left(\frac{v}{\sin(\gamma)} \right) * \sin \left(r - \gamma + \frac{\pi}{2} \right) + \left(\frac{w}{\sin(\alpha)} \right) * \sin(\alpha - \pi) \right)^2} \right)$$

$$j = \cos^{-1} \left(\frac{\left(\left(\frac{u}{\sin(\beta)} \right) * \cos \left(s - \beta + \frac{\pi}{2} \right) + \left(\frac{w}{\sin(\alpha)} \right) * \cos(\alpha - \pi) \right)^2}{\left(\left(\frac{u}{\sin(\beta)} \right) * \cos \left(s - \beta + \frac{\pi}{2} \right) + \left(\frac{w}{\sin(\alpha)} \right) * \cos(\alpha - \pi) \right)^2 + \left(\left(\frac{u}{\sin(\beta)} \right) * \sin \left(s - \beta + \frac{\pi}{2} \right) + \left(\frac{w}{\sin(\alpha)} \right) * \sin(\alpha - \pi) \right)^2} \right)$$

$$a = \frac{w}{\sin(\alpha)} * \sin(j)$$

$$x = a * \cos(i)$$

$$y = L - a * \sin(i)$$

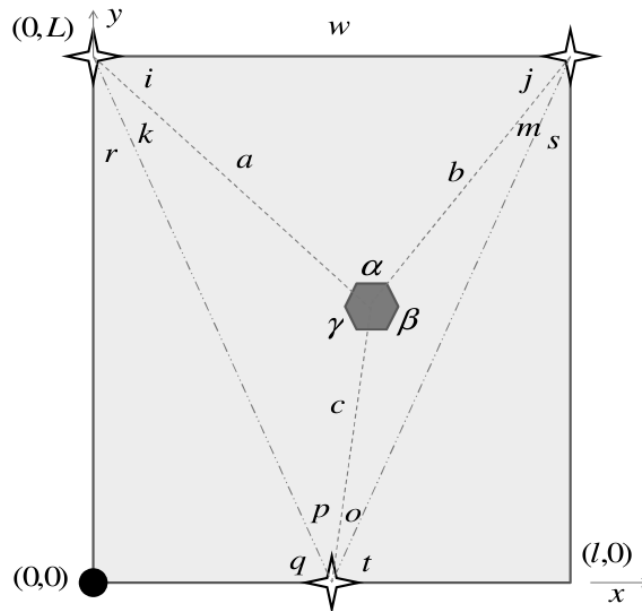


Illustration 2.4.2: Beacon mathematical purpose

It is an heavy computation. However, a lot of terms (like r,q,t,s) are constant.

2.4.3 Hardware design

All the work can now be split in two distinct parts: transmitter and receiver beacon. This section describes the hardware used and the physical phenomenon used by these infrared beacons. The main idea is to send modulated UART frames with IR diodes. This has to be in a specific frequency band in order to demodulate the signals. The receivers are TSOP modules which provide directly the proper filter.

2.4.3.1 TX Beacon

The schematic of the TX beacon can be found in **Appendix 5.7** . The masks are in **Appendix 5.8** and the implementation layout in **Appendix 5.9**. Bellow are detailed explanations about the different beacon parts.

a Modulation

The three transmitters (shorten TX) beacons behave exactly the same way. A microcontroller is responsible for sending periodically UART frames. The signal has to be modulated at 36kHz (TSOP frequency). This is achieved by using an XOR gate as shown on **Illustration 2.4.3**.

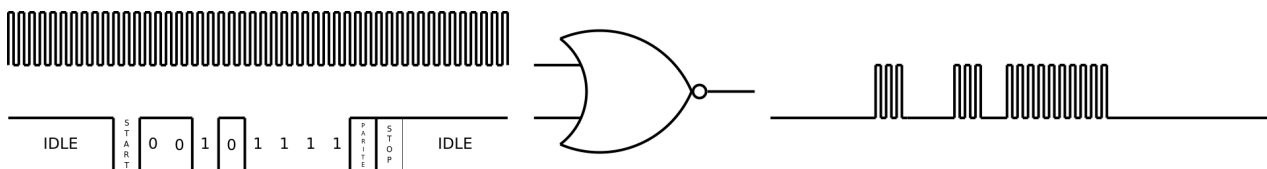


Illustration 2.4.3: Beacon IR Modulation

It's possible to find individual NOR gate (TSOP23 package). The **MC74HC1G02** is used for the project.

b Current driving

Now that a correct logic signal is generated, the infrared LEDs need to be powered. In order to have a smooth and constant field of emission, it's needed to use at least 12 LEDs for 180° range. The TX beacons use 16 LEDs. Each one of those (ref. TSUS540) can support a peak forward current of 300mA (150mA in continuous state). The LEDs are associated 4 by 4, which means a potential current drive of 1200mA!

A bipolar power transistor **TIP122** is used, it can drives 5A. The driven current on its collector is set by the base current. It's possible to adjust it easily with a potentiometer. There are some digital potentiometer, like the **CAT5114**. So it's also possible to drive the emitted power, very useful for calibrations or even more complex frames (same frame with different emission levels).

The overall stage modulation + current driving is shown on *Illustration 2.4.4*.

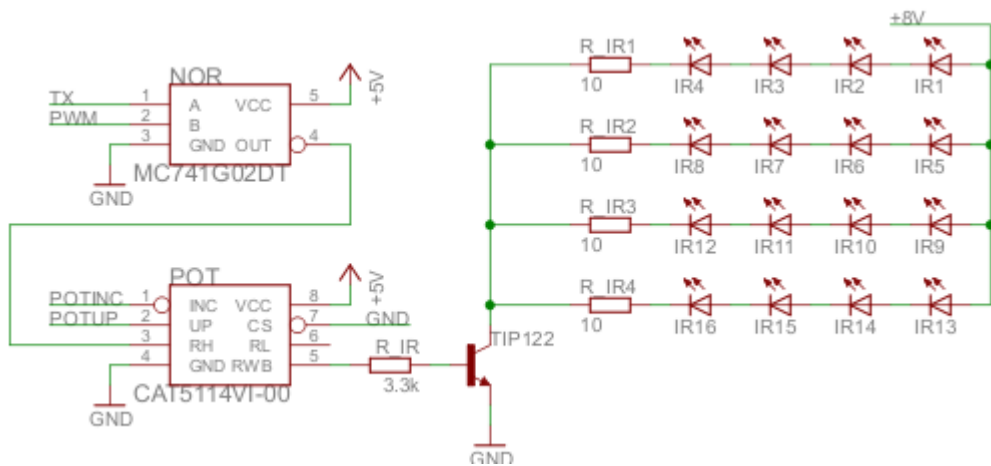


Illustration 2.4.4: TX Beacon emission stage

c CAN synchronization

Because the same frequency is used for all the TX beacons, they have to send their frames one after the other. It implies to set a synchronization protocol between them. The rules said it's possible to link them with a wire. So it has been chosen to use the CAN protocol which is fast enough and reliable to synchronize them.

The CAN transceiver used is a **SN65HVD1050** from Texas instrument, it operates under 5V power supply.

d Microcontroller

The chosen microcontroller is a **PIC18F2480**. Some peripherals are compulsory so the choice is quickly limited to few devices. Microchip has been chosen because there is already a PicKit3 programmer available for the project. The mandatory features are:

- UART, to send data
- Timer, for PWM modulation
- CAN driver, for synchronization

The choice of using an external oscillator is due to the precision needed for the UART. There are also two switches and three LEDs that can be used to indicate beacon status or anything else.

2.4.3.2 RX Beacon

The RX beacon board is both used for the opponent and project robots. It embeds:

- TSOP IR receivers
- Multiplexed UART (for multiple receivers)
- Xbee interfaced on a second UART
- CAN bus interface (not used on the opponent's board)

The full schematic is available in **Appendix 5.10**. As usual, masks and implementation layout follow in **Appendix 5.11** and **5.11**. Details about specific components are present bellow.

a TSOP IR Receivers

A total of 16 receivers are available all around the board, which means each sensor covers 22.5°. When something is detected, the TSOP output is directly the UART frame sent. The receivers are connected as shown on *Illustration 2.4.5*.

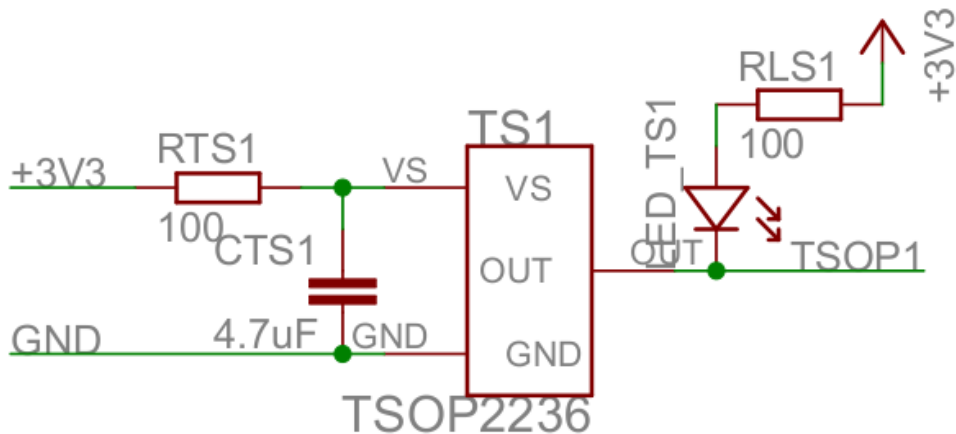


Illustration 2.4.5: TSOP connection

The LED pulled high is here in order to have a visual feedback when the sensor is actually receiving. RTS and CTS components are recommended by the manufacturer (low-pass filter, cut-off frequency around 2kHz).

b UART multiplexers

As a matter of fact, there is only one UART available for the 16 receivers. As it's impossible to find a microcontroller with 16 UART, there are two main alternatives:

- Use time multiplexing
- Use a microcontroller with reconfigurable pins

It has been chosen to use multiplexers. There are 3 CD4051B (8 to 1) but only 4 bits are used to address the whole range of sensors. The connections are routing dependent and are associated to the correct order in software. The shows the actual multiplexing bloc.

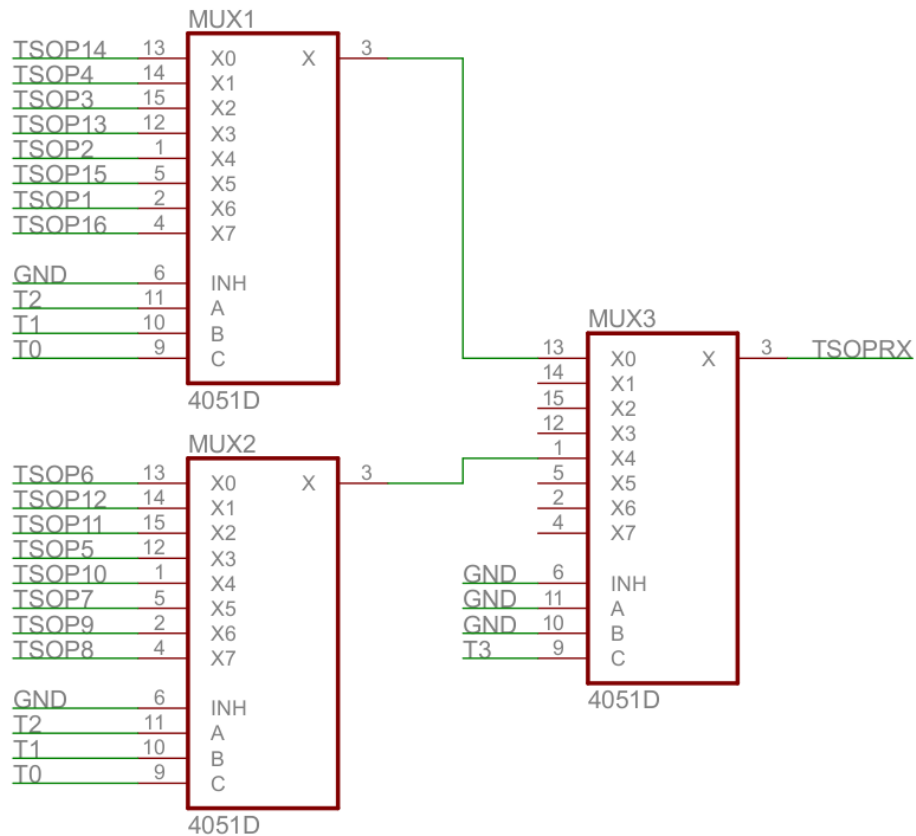


Illustration 2.4.6: RX Beacon receivers multiplexing

2.4.4 Software design

The beacons software is composed of two distinct parts, either for TX or RX beacon:

- The physical interface and management (PWM setup, power emission setup, initialisations etc.).
- The networking features (CAN or RF).

The first point is quite straightforward and is about correct initialisation of peripherals (timers, CAN, ...).

The CAN synchronization steps are explained bellow:

- On power on, initialise the beacon and don't emit anything
- Listen to the CAN, if there is nothing for 1 sec:
 - Set the beacon as master (ID #1).
 - Send periodically (for instance every 50ms) a "STATE" signal on CAN bus

- Send by IR a "STATE" frame with beacon ID 1.
- Read the CAN
- If there is nothing, go back do the 3 previous states again
- Else, register the new incoming beacon ("STATE" signal with ID 2 or 3)
- If there are less than 3 beacons, go back 5 steps behind
- else,
 - Register the beacon ID
 - Read the CAN again for 1sec
 - If there isn't any other device, set the ID to #2
 - Repeat the last two steps
 - Otherwise, set ID to #3
- All beacons are ON, entering main synchronization process:
- If ID = 1,
 - Send "SYNCHRO" signal with ID 1
 - Send IR frame(s) with ID 1
 - Read the CAN until receiving "SYNCHRO" signal with ID 3
 - Wait for 100ms (IR frames sending time)
 - Loop
- else,
 - Read the CAN until receiving "SYNCHRO" signal with ID n-1
 - Wait for 100ms (IR frames sending time)
 - Send "SYNCHRO" signal with ID
 - Send IR frames with ID
 - Loop

This pseudo code doesn't take into account faults, other timeouts or shut down procedures. For example it could be useful to send error frames to the receiver if a TX beacon doesn't respond.

2.5 High-level software

2.5.1 Overview of main software

The main program's goal is to manage all the different and individual modules. There are many ways to do that, but the program runs on an Operating System so it's possible to use the POSIX multithreading features.

Each "manager" is considered as an independent thread and is launched by the main process. This project intended to work on three threads (but they are more than that):

- The strategy manager, which calls all the pathfinding and motion control algorithms, manage to send message when it's needed etc.
- The CAN manager, which is responsible for messages reception and take proper actions.
- The Log manager; a passive thread that simply store the robot's status in a logfile.

The *Illustration 2.5.1* shows the different interconnections between software entities.

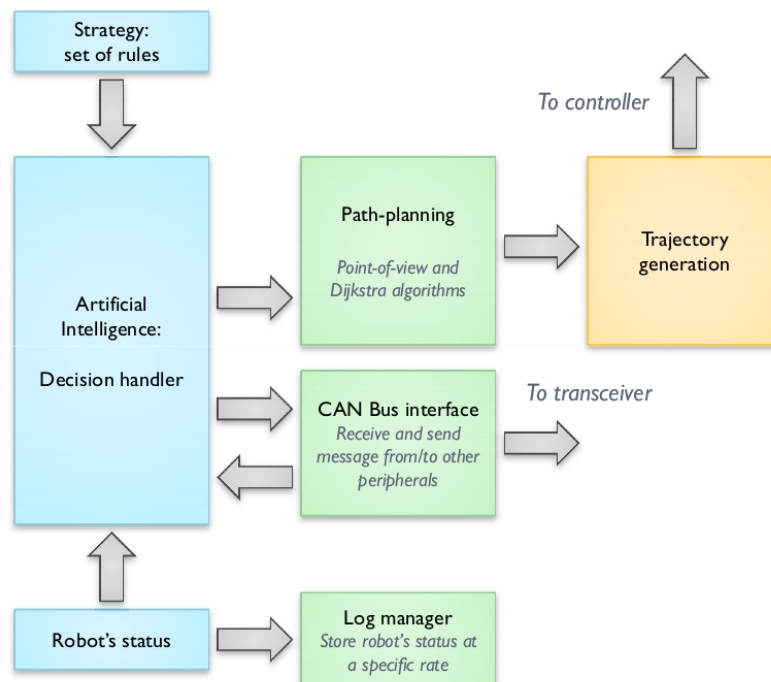


Illustration 2.5.1: Software diagram

2.5.2 Strategy manager

The strategy manager is the main thread of the program. It uses all the available functions provided by the artificial intelligence, path finder, trajectory generator, motion controller and can manager modules. It also uses the robot's status which is stored and manipulated with the context module.

The artificial intelligence uses a set of rules, determined at the beginning. These rules say, for instance, which action between two possibilities has the highest priority. It provides primitives that actually call related function(s). For example "MOVE THE ARM" corresponds to send a message on the CAN bus where "MOVE THE ROBOT" corresponds to a complex cascading call of functions.

2.5.3 CAN manager

The CAN manager module provides the high level interface with the other robot's peripherals. The initialisation function opens the device and creates a socket, using the *socketCAN* module for linux.

The main thread keeps waiting for incoming messages. When something arrives, it's decoded and fetched. Then, the robot's status is updated if needed (for example, the position of a servomotor has changed).

2.5.4 Log manager

The log manager is surely the smallest thread but remains nonetheless important. Its utility is explained further in the report.

The first thing done by the initialisation function is to create a new file with a timestamp in its name. The format is "*elder_yyyy_mm_dd-hh_mm.log*". So that files are sorted by name from the oldest to the newest.

As long as the main program runs, the context is stored in this file periodically (the rate can be adjusted, 50ms is used which corresponds to 20 samples per seconds).

When the program ends, the close function is called. The logfile is then closed and copied with a different name: "*last.log*". Then, in order to retrieve the log there is no need to know the exact name (extremely useful for automated tasks that can be scripted).

3 Testing and validation

As a part of every development, it's as important to develop an run relevant test as the design itself. This section presents different kind of tests used in this project.

3.1 PCB debugging

After an hardware design, it's likely to manufacture the PCB. A lot of things can go wrong if attention is not taken, components can be destroyed or even explode. This is even worth when the design implies batteries, some of them (the Lion polymer type) have to be treated carefully and a short circuit can leads to an important safety issue.

This part is written as a set of rules, that can be applied for any PCB.

3.1.1 *Test the tracks*

First of all, it's important before soldering anything to test the continuity of the different tracks. If a track is a broken, it can often be repaired by using some tin on it. More important is the short circuits, when testing the tracks it's also mandatory to insure that they aren't connected together if they shouldn't.

3.1.2 *Solder the vias*

The second stage is to solder the different vias (if the boards is not from a professional manufacturer!). They insure the link between top and bottom layer. After that, the first operation needs to be run again. A last test and very important is to verify the discontinuity between the Ground and different power supplies signals.

3.1.3 Solder the power supply components

Some variations can occur depending on the type of board. However, it's important to insure that there isn't any trouble with the supplies so that other blocs can be solder safely. Of course, after soldering the power supply, test it!

3.1.4 Solder the rest bloc by bloc

The other parts can be soldered. As previously, it's sometimes important for large design to not solder everything in a row.

3.2 VHDL simulation

In this chapter, tools and mean about VHDL simulation are introduced. Because the chip is a programmable device, it's not easy to probe it directly when the system is running. Some tools exists (provided by Xilinx) and need to be added during the synthesize phase. Data will be gather and can be then watched in live using JTAG probe.

However there isn't any available probe for the project, and actually this kind of test can be avoid if a proper set of simulations has been run before. The tool used is gHDL associated with gtkViewer to watch the time graphs.

The first step is to design a *testbench* file for the entity of interest. This file doesn't have to follow the synthesis rules and has no inputs/outputs. It only instantiates one or multiple components and provide stimuli for their inputs. The simulation process then compute the output states of the components which can be seen in an appropriate viewer.

As an example, the following code is given to show how a clock can be generated without synthesis considerations:

```

-- Clock generation
clk_generator: process
begin

    sig_clk <= '0';
    wait for clk_Period/2;
    sig_clk <= '1';
    wait for clk_Period/2;

end process;

```

Where *clk_period* is a constant value directly given in seconds.

The *Appendix 5.13* show an example of output test for the entity *accel_limiter* (limit the incoming speed's change to an upper and lower bound of acceleration).

3.3 Software debug and test

Debugging a program running on a remote machine with a different architecture is more complicated than a "normal" (i.e., compiled for x86) program that runs on a laptop. In this project, three different ways have been used to test the programs:

- Compile it for x86 architecture when it's possible. It's often the case with high level modules (like pathplanning, trajectory generation etc.).
- Cross-compile it for the target and debug it using **gdbdebugger**. This software uses the ethernet link to send different debugging command (run, halt, next etc.) to the target and retrieving values.
- The last possibility is to use logfiles. These ones are not only present for debugging when something goes wrong but also to store specific values that can be used/post-processed later. The *log_manager* thread stores at a specific rate informations about the robot. These ones are retrieved later using ssh. A gnuplot script is then used to see the different values of position, speeds etc. as graphs. This is then a powerful tool which helps for any control development. An example of Gnuplot output is given in *Appendix 5.14* .

4 Conclusion and future work

This section presents the conclusion made at the end of the project. It also summarizes achievements and remaining work. Due to its nature, there are always possible improvements and this is shown every year at Eurobot, teams never stop to work on better solutions. As a result, possible upgrades and further developments will be presented.

4.1 General conclusion

The aim of this project was to design, validate and build a motion control system for a robot. This had to deal with a lot of different matters, most of them coming directly from the competition's rules. The different designs are either hardware or software. The general development for each of them follow roughly the same process: specifications (identify the need), design and technological choices (thoughts about the "how"), build the system or develop the code and finally, test it.

As a global result, the aim has been achieved: the robot is able to retrieve its position according to the incremental encoders which was the first steps of the motion control system. The different parameters are then used as a feedback to close the control loop and the numerous tests ran prove that the controller works. The software design that stands behind also works properly.

However, there is still some development to work on. It concerns mainly some software parts as the Artificial Intelligence. Priority has been given to hardware developments.

4.2 Project planning

A conclusion also allows to state whether the planning has been respected or not. In general terms, the work that was planned in October have been done but considerably delayed for some developments. The fear was at the beginning, long delays concerning workshop manufacture. It wasn't the case at all and the jobs achieved by the staff were done quickly. The delays come from developments and designs that took more time than expected.

4.3 Eurobot event

The competition starts on the 13th of May. The end of the academic project doesn't mean the end of the development of work because both students want to do the best at Eurobot!

4.4 Further work

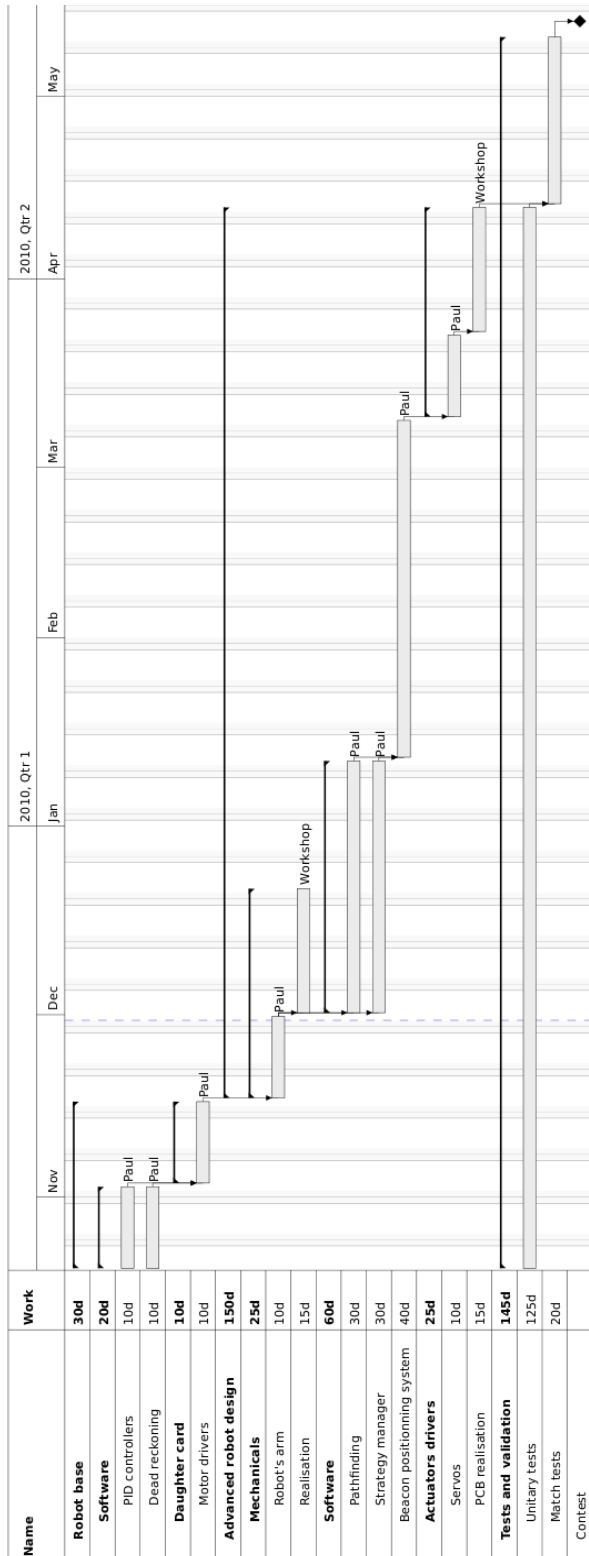
As stated previously, there are still things to develop and improve. However, most of the work has been done and further work involving projects or research topics for the next year would probably have their own specifications.

Because the Armadeus board doesn't belong to the University, the daughter board has been developed with the possibility of implementing the motion control on it. The design, is then different because it involves only a microcontroller.

Further work around the beacon system can also be done, maybe improving the communication protocol between fixed beacon is possible. Another thing could be to add an Xbee module on these lighthouses and define an auto-calibration setup at the beginning, involving the power emission modulation. Then the beacons are able to work with different kind of surrounding lights and noise.

5 Appendices

5.1 Initial Gantt Chart



5.2 Software makefile

```
# _____
# | ___| | | | _ \| ___| _ \|
# | | | | | | | | | | | | | |
# | | | | | | | | | | | | | /
# | | | | | | | | | | | | | \|
# | | | | | | | | | | | | | \|
#
#
# Global definitions
# -----

# Projects constants
EXEC_NAME := elder_arm
OBJ_DIR   := ./obj
BIN_DIR   := ./bin
SRC_DIR   := ./src
INC_DIR   := ./inc
FW_PATH   := /home/paul/Work/ise/elder_fw/elder_top.bit

# Armadeus board
ARM_ADDR  := 192.168.0.10
HOST_ADDR := 192.168.0.1
ARM_USER  := root
GDB_PORT  := 2345
INSTALL_DIR := /root/bin
FW_DIR    := /root/fpga_fw

# Host tftpboot directory
TFTPBOOT_DIR := /tftpboot
```

2010

```

# Target switch
# -----

# arm
ifeq ($(TARGET),target)

    # Armadeus base directory
    ARMADEUS_BASE_DIR = ~/Work/armadeus/git-armadeus
    include $(ARMADEUS_BASE_DIR)/Makefile.in
    ROOT_DIR := $(ARMADEUS_ROOTFS_DIR)

    # Compiler, flags and libs
    CC := $(ARMADEUS_TOOLCHAIN_PATH)/arm-linux-gcc
    STRIP := $(ARMADEUS_TOOLCHAIN_PATH)/arm-linux-strip
    CFLAGS := $(shell STAGING_DIR=$(ARMADEUS_STAGING_DIR) sh $(ARMADEUS_SDL_DIR)/sdl-
config --cflags) -I $(INC_DIR) -g
    LDFLAGS := -I $(INC_DIR)
    LIBS := $(shell STAGING_DIR=$(ARMADEUS_STAGING_DIR) sh $(ARMADEUS_SDL_DIR)/sdl-config
--libs) -lpthread -lSDL_image -lSDL_ttf -lSDL_mixer -lSDL_net

    # Defines for preprocessor
    DEFINES = -DTARGET

#x86
else ifeq ($(TARGET),host)
    # Host base directory
    ROOT_DIR:= "/"

    # Compiler, flags and libs
    CC := gcc
    STRIP := ""
    CFLAGS := `usr/bin/sdl-config --cflags` -I $(INC_DIR) -g
    LDFLAGS := -I $(INC_DIR)
    LIBS := `usr/bin/sdl-config --libs` -lSDL -lSDL_ttf -lSDL_image -I $(INC_DIR)

    # Defines for preprocessor
    DEFINES = -DHOST

endif

# Source files
SRC := $(shell find . -name "*.c" -print)

# Object files
OBJ := $(SRC:.c=.o)

# Building rules

```

```
# -----

default: $(EXEC_NAME)

arm: $(EXEC_NAME)

x86: $(EXEC_NAME)_x86

$(EXEC_NAME): $(OBJ)
    $(CC) $(DEFINES) $(LIBS) -lm -o $@ $^ $(LDFLAGS)
    mv $(OBJ) $(OBJ_DIR)
    mv $(EXEC_NAME) $(BIN_DIR)

$(EXEC_NAME)_x86: $(OBJ)
    $(CC) $(DEFINES) $(LIBS) -lm -o $@ $^ $(LDFLAGS)
    mv $(OBJ) $(OBJ_DIR)
    mv $(EXEC_NAME)_x86 $(BIN_DIR)

%.o: %.c
    $(CC) $(DEFINES) $(CFLAGS) -lm -c -o $@ $^

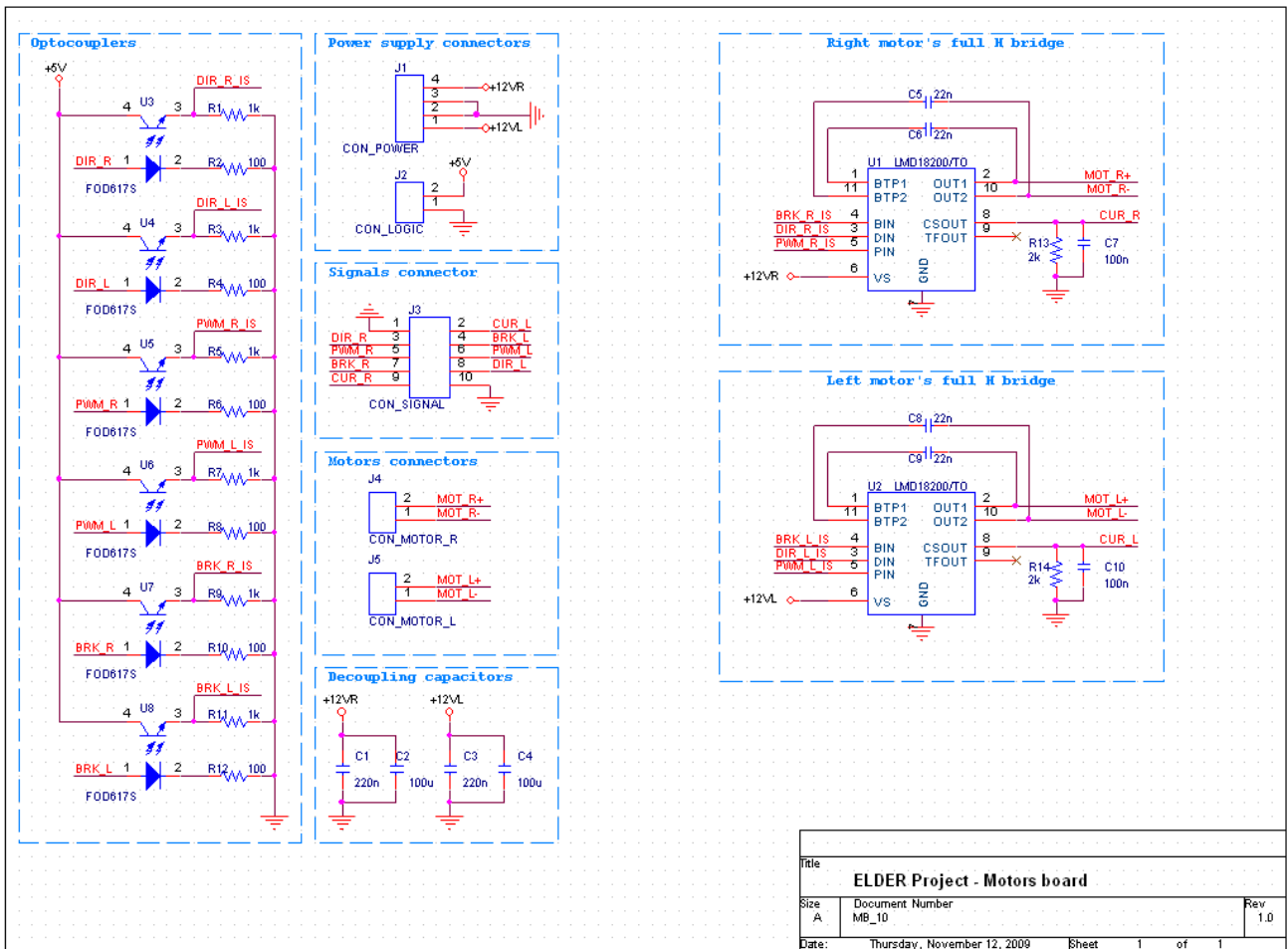
install: arm
    cp $(BIN_DIR)/$(EXEC_NAME) $(TFTPBOOT_DIR)
    scp $(BIN_DIR)/$(EXEC_NAME) $(ARM_USER)@$(ARM_ADDR):$(INSTALL_DIR)

cleanall: clean

update_fw:
    scp $(FW_PATH) $(ARM_USER)@$(ARM_ADDR):$(FW_DIR)

clean:
    rm -f $(OBJ)
    rm -rf $(OBJ_DIR)/*.o
    rm -f $(BIN_DIR)/$(EXEC_NAME)
```

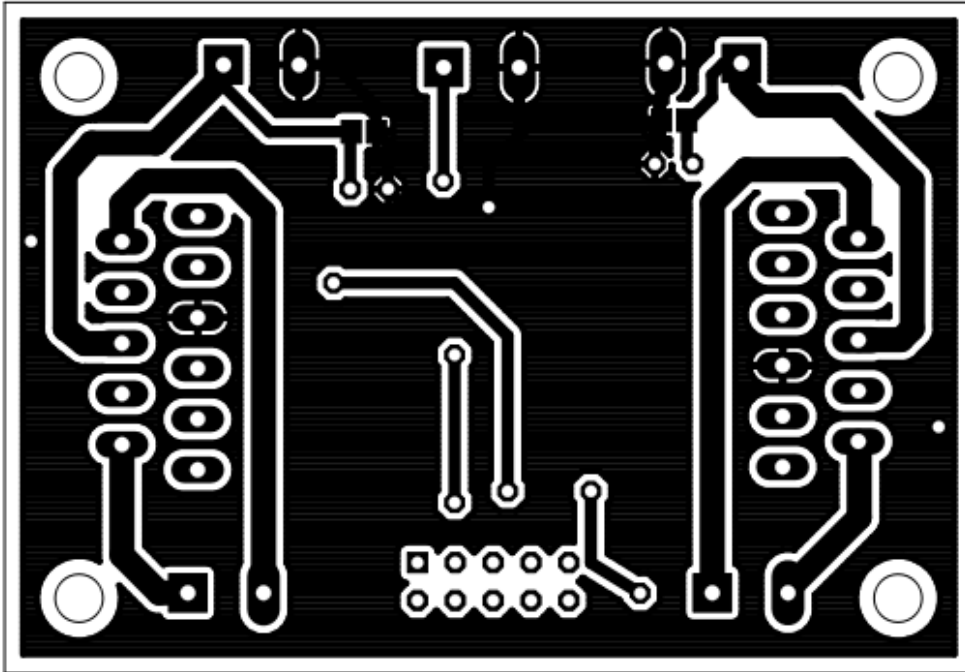
5.3 Motors board schematic



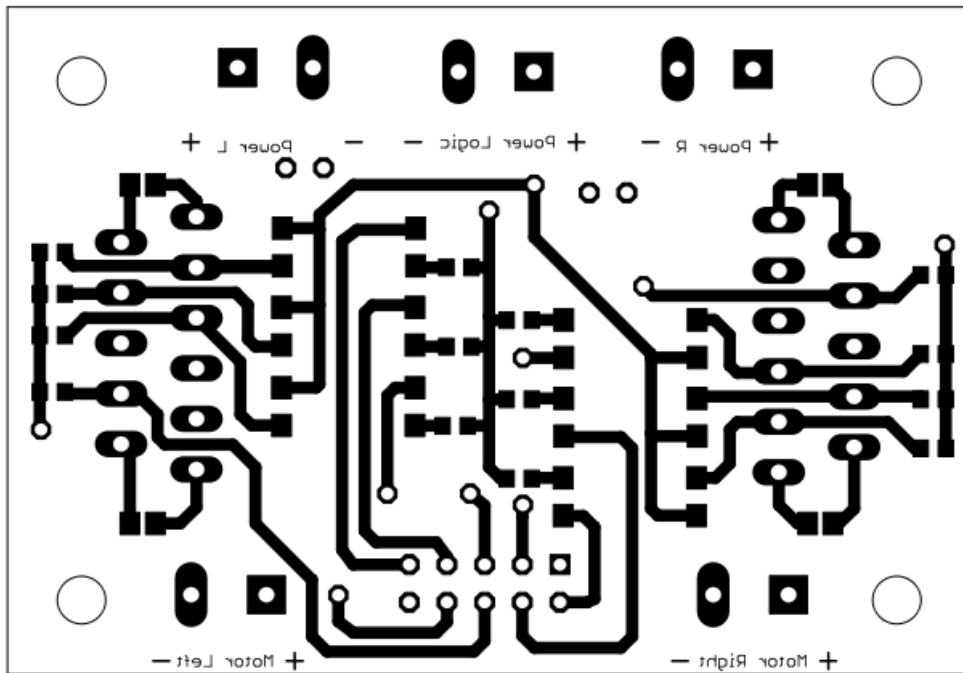
Title		
ELDER Project - Motors board		
Size	Document Number	Rev
A	MB_10	1.0
Date:	Thursday, November 12, 2009	Sheet 1 of 1

5.4 Motors board masks

(note: this isn't scaled, the actual board is smaller).

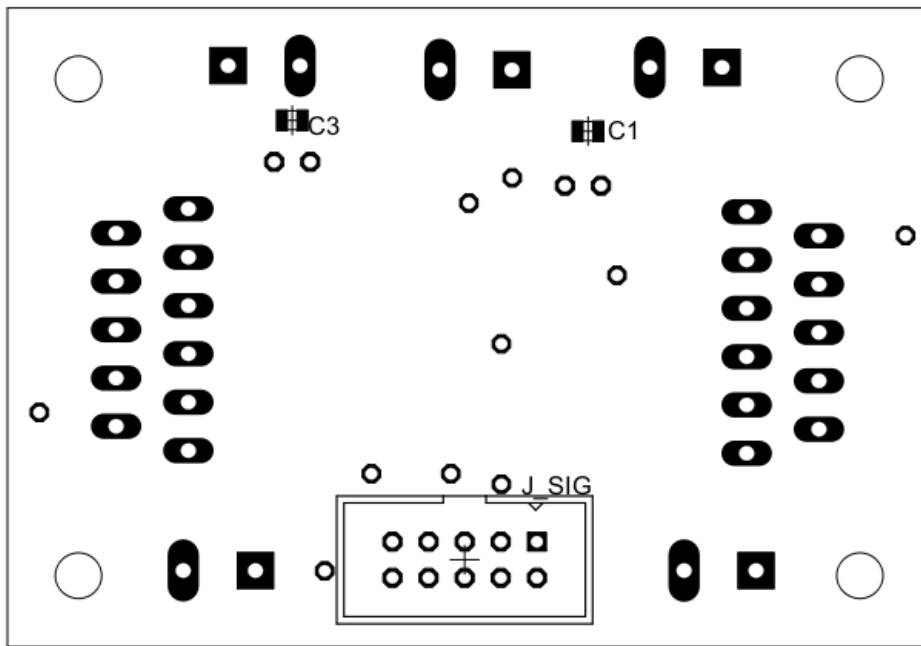


Bottom

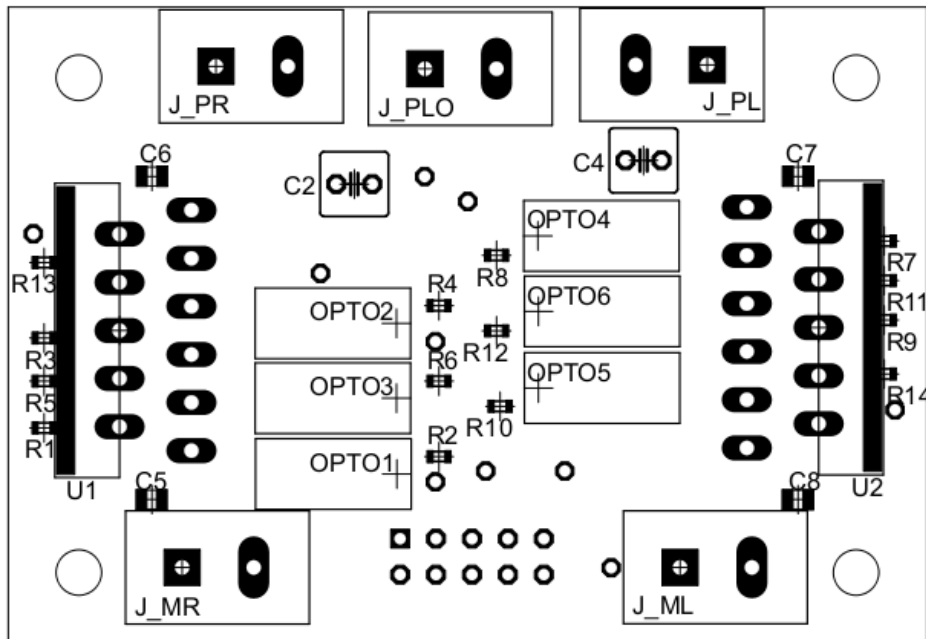


Top

5.5 Motors board implementation layout



Bottom



Top

5.6 VHDL code of AD_to_XYT converter

```
-----
-- ELDER Project
-- 2009 / 2010
--
-- Motion Control
-----
--
-- AD_to_XY.vhd
--
-- Description:
--
-- This is the main unit of the localization
-- manager. Given Angle and Advance, it computes
-- the new position every clkControl rising edge.
--
-----

-- Libraries
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-----
entity AD_to_XYT is
-----
port
    (
        -- Clock and reset
        clk          : in std_logic;

        reset        : in std_logic;

        -- Control clock
        clkControl   : in std_logic;

        -- Interaxial constant
        interaxial   : in std_logic_vector(15 downto 0);

        -- Position in tick
        posA         : in std_logic_vector(15 downto 0);

        -- Speed, in tick * fcontrol
    )
end entity AD_to_XYT;
```



```

speedD          : in std_logic_vector(15 downto 0);

-- Initial positions, assuming speeds are equal to 0
posXi           : in std_logic_vector(15 downto 0);
posYi           : in std_logic_vector(15 downto 0);
posTi           : in std_logic_vector(15 downto 0);

-- Output positions in tick
posX            : out std_logic_vector(15 downto 0);
posY            : out std_logic_vector(15 downto 0);
posT            : out std_logic_vector(15 downto 0)
);
end AD_to_XYT;

-----
architecture behav of AD_to_XYT is
-----
-- Constants
-----

-- Pi = 0011.00100100001111 in 3Q13
-- -Pi = 1100.1101101111001 in 3Q13
-- 2Pi = 0110.0100100001111 in 3Q13
constant PLUS_PI      : signed(15 downto 0) := "0110010010000111";
constant MINUS_PI     : signed(15 downto 0) := "1001101101111001";
constant TWO_PI       : signed(15 downto 0) := "1100100100001111";

constant DIVIDER_LATENCY : natural := 300; -- "M+5"*8 = (32+5)*8 rounded to 300

-- Components
-----

component cordic is
    port (
        phase_in      : in std_logic_vector(15 downto 0);
        nd             : in std_logic;
        x_out          : out std_logic_vector(15 downto 0);
        y_out          : out std_logic_vector(15 downto 0);
        rdy            : out std_logic;
        rfd            : out std_logic;
        clk             : in std_logic;
        ce              : in std_logic;
        aclr            : in std_logic);
end component;

component divider is

```

```

port (
    sclr : in STD_LOGIC;
    rfd : out STD_LOGIC;
    clk : in STD_LOGIC := 'X';
    dividend : in STD_LOGIC_VECTOR ( 31 downto 0 );
    quotient : out STD_LOGIC_VECTOR ( 31 downto 0 );
    divisor : in STD_LOGIC_VECTOR ( 15 downto 0 );
    fractional : out STD_LOGIC_VECTOR ( 15 downto 0 )
);
end component;

component multiplier is
    port
    (
        clk: in std_logic;
        a: in std_logic_vector(15 downto 0);
        b: in std_logic_vector(15 downto 0);
        sclr: in std_logic;
        p: out std_logic_vector(31 downto 0)
    );
end component;

-- Signals
-----

signal posArad                : std_logic_vector(15 downto 0); -- 2Q13

-- Actual values of X, Y, T in tick
-- and the velocities in tick * fcontrol
-- Internal precision is 32 for better accuracy
-- (save the fractionnal parts of dX and dY)
-- 1mm ~ 4096/(Pi*45mm) ~ 29 tick,
-- so we need at least 17bits for 3000mm
-- 19 bits for integer part (signed) : 28 downto 13
-- 13 bits for the fractionnal part : 12 downto 0
signal X                       : signed(31 downto 0); -- 18Q13
signal Y                       : signed(31 downto 0); -- 18Q13
signal Trad                    : signed(15 downto 0); -- 2Q13

-- Delta X and Y values in tick
signal dX                      : signed(31 downto 0);
signal dY                      : signed(31 downto 0);

-- Divider control signals

```

```

signal div_rfd          : std_logic;
signal div_ready       : std_logic;
signal quotient        : std_logic_vector(31 downto 0);
signal quotients       : signed(31 downto 0);
signal dividend        : signed(31 downto 0);

```

-- CORDIC control signals

```

signal cor_ready       : std_logic;
signal cor_rfd        : std_logic;
signal cosPosT        : std_logic_vector(15 downto 0);
signal sinPosT        : std_logic_vector(15 downto 0);
signal cosPosTs       : signed(15 downto 0);
signal sinPosTs       : signed(15 downto 0);

```

```

-----
begin
-----

```

-- Type conversion

```

cosPosTs      <= signed(cosPosT);
sinPosTs      <= signed(sinPosT);
quotients     <= signed(quotient);

```

-- Hardware Divider to convert the
-- angle in radian

```

radDivdier: divider
  port map(
    sclr          => reset,
    rfd           => div_rfd,
    clk           => clk,
    dividend      => std_logic_vector(dividend),
    quotient      => quotient,
    divisor       => std_logic_vector(interaxial),
    fractional    => open
  );

```

-- Sine & Cosine computing

```

sinCos: cordic
  port map
  (
    phase_in      => std_logic_vector(Trad),
    nd            => div_ready,
    x_out         => cosPosT,
    y_out         => sinPosT,
    rdy           => cor_ready,
    rfd           => cor_rfd,

```

```

        clk          => clk,
        ce           => '1',
        aclr         => reset
    );

-- Signals

-- pre multiplication by 2^16
dividend    <= signed(posA) & X"0000";

-- quotient(31) is the sign bit
-- quotient(30 downto 16) is the integer part
-- quotient(15 downto 0) is the fractionnal part
-- it's wrapped to a 2Q13 format, ie:
--   bit: 31          16 15          0
-- quotient: 0000 0000 0000 0000. 0000 0000 0000 0000
-- posTrad:          000. 0000 0000 0000 0
posArad(15)    <= quotient(31);          -- sign bit
posArad(14 downto 13) <= quotient(17 downto 16); -- i. part
posArad(12 downto 0) <= quotient(15 downto 3);  -- f. part

posX    <= std_logic_vector(X(30 downto 15)); -- i. part only (with factor 4)
posY    <= std_logic_vector(Y(30 downto 15));
posT    <= std_logic_vector(Trad);

-- Data validation process
process(clk, clkControl)
    variable latency : natural;
begin

    -- clkControl acts as a reset
    -- put the divider in "not ready" state first
    if clkControl = '1' then
        latency := 0;
        div_ready    <= '0';
        dX           <= (others => '0');
        dY           <= (others => '0');

    elsif rising_edge(clk) then

        -- Count each clock when div_rfd is high
        if div_rfd = '1' then
            latency := latency + 1;
        end if;
    end if;
end process;

```

```

-- we have wait enough, the ouptut data from
-- the divider is ready and the cordic input
-- data is then ready for data
if latency = DIVIDER_LATENCY then
    div_ready <= '1';
end if;

-- Cordic has just finished, output data is valid
-- we can compute the dX and dY multiplications
if cor_ready = '1' then
    dX          <= cosPosTs * signed(speedD);
    dY          <= sinPosTs * signed(speedD);
end if;

    end if;
end process;

-- Update coords process
process(clkControl, reset)

    variable newTrad : signed(15 downto 0);

begin

    -- Reset to initial positions
    -- Velocities are assumed to be null
    if(reset = '1') then
        X          <= signed(posXi) & "00000000000000"; -- fractional part is null
        Y          <= signed(posYi) & "00000000000000";
        Trad       <= signed(posTi);

        newTrad := (others => '0');

    elsif rising_edge(clkControl) then

        -- New X and Y values
        X      <= X + dX;
        Y      <= Y + dY;

        -- New angle value, modulo 2Pi
        -- -Pi <= T <= Pi
        newTrad := Trad + signed(posArad);

        -- Modulo

```

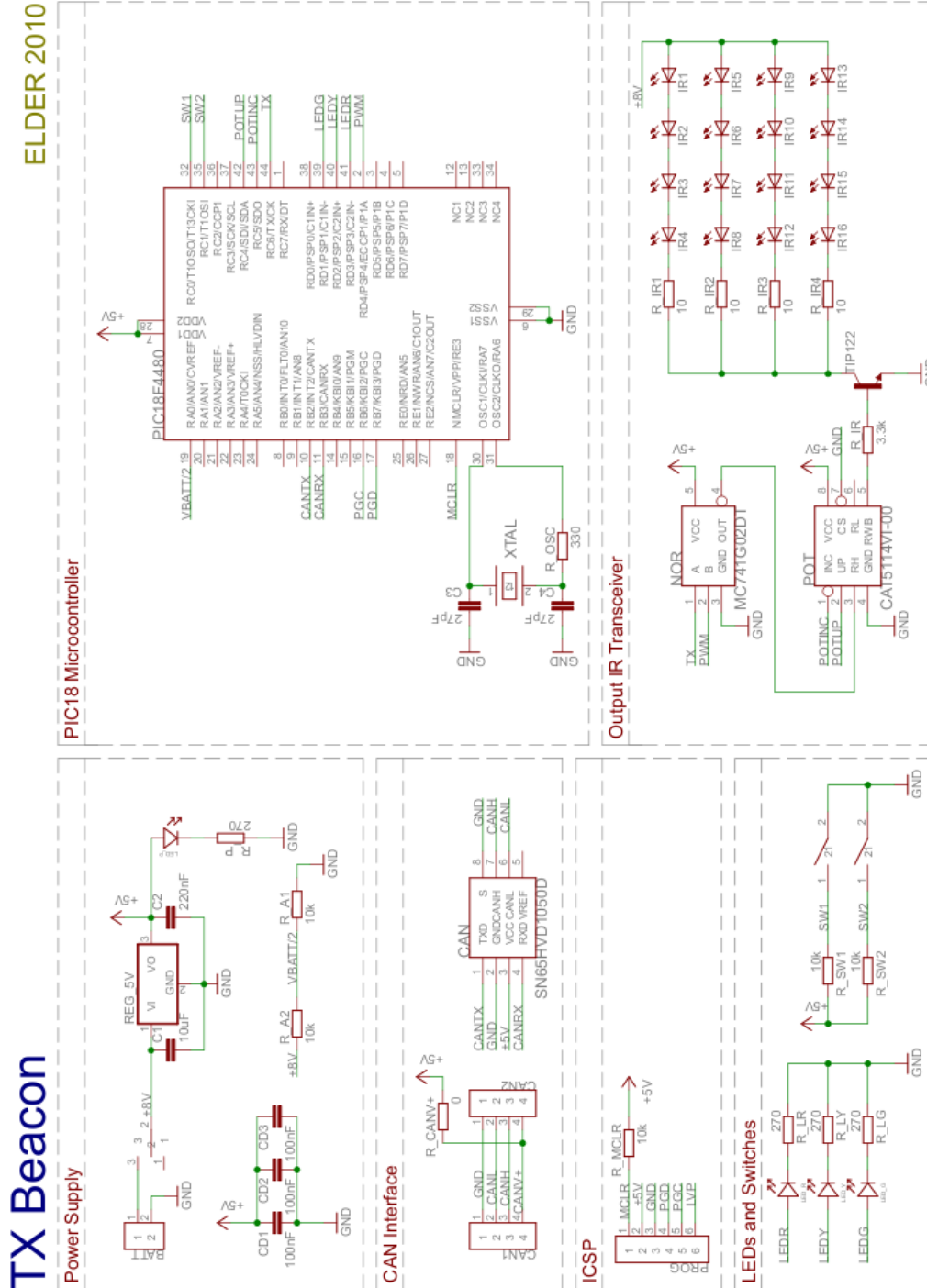
```
if newTrad > PLUS_PI then
    Trad <= newTrad - TWO_PI;
elsif newTrad < MINUS_PI then
    Trad <= newTrad + TWO_PI;
else
    Trad <= newTrad;
end if;
```

```
end if;
```

```
end process;
```

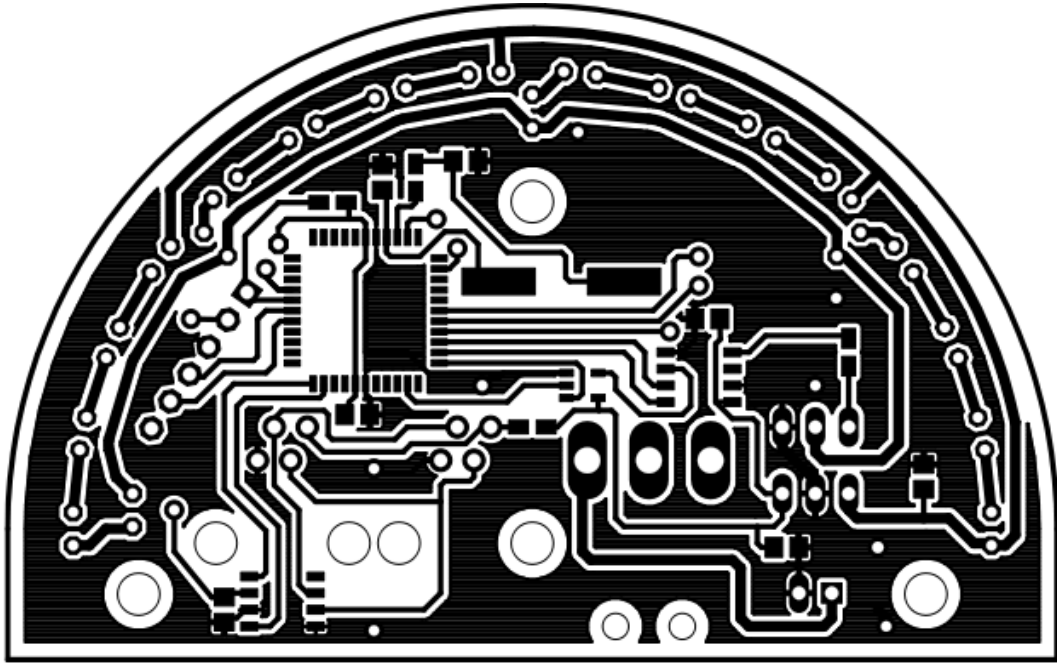
```
end behav;
```

5.7 TX Beacon schematic

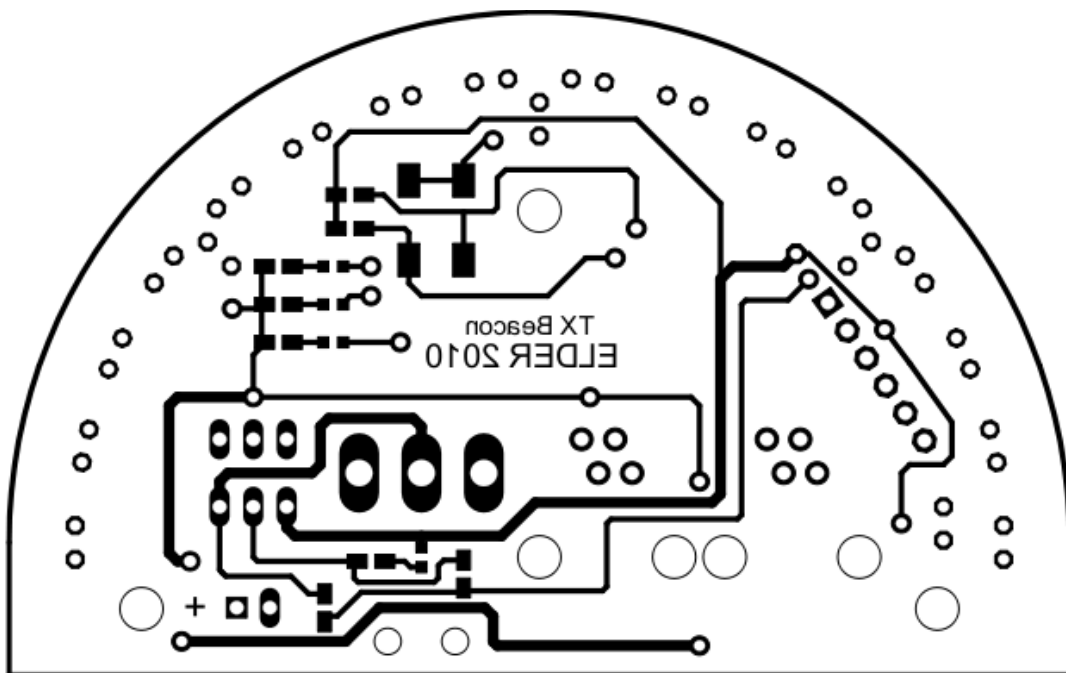


5.8 TX Beacon masks

Note: again, masks aren't scaled. The actual width is 80mm.

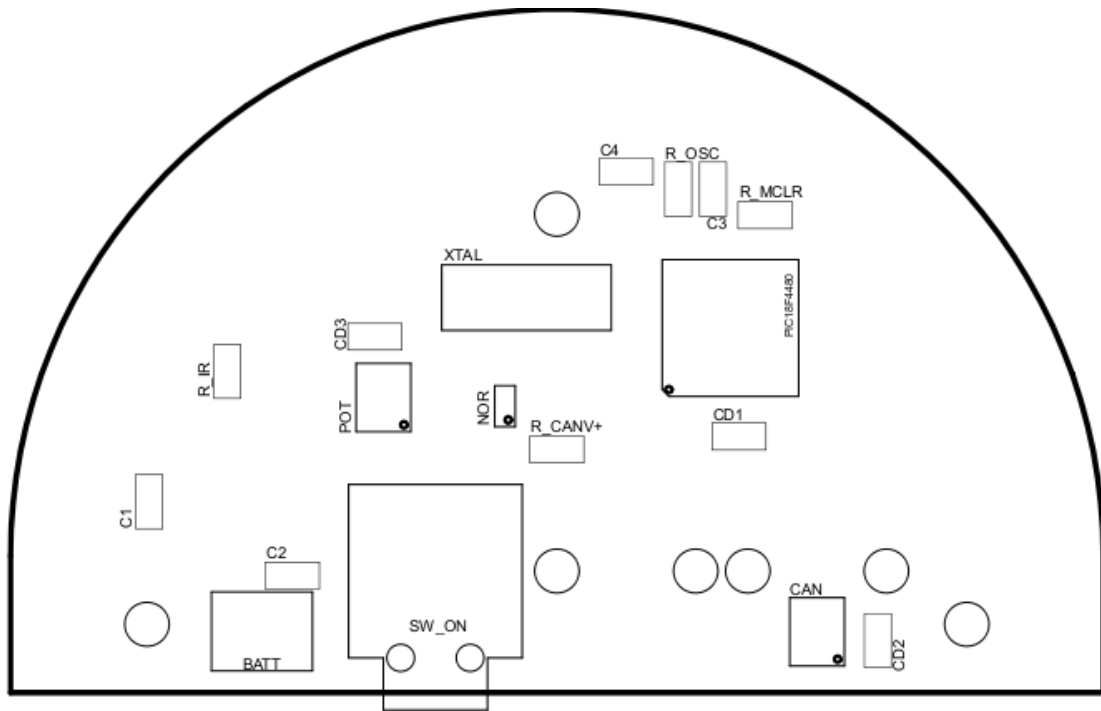


Bottom

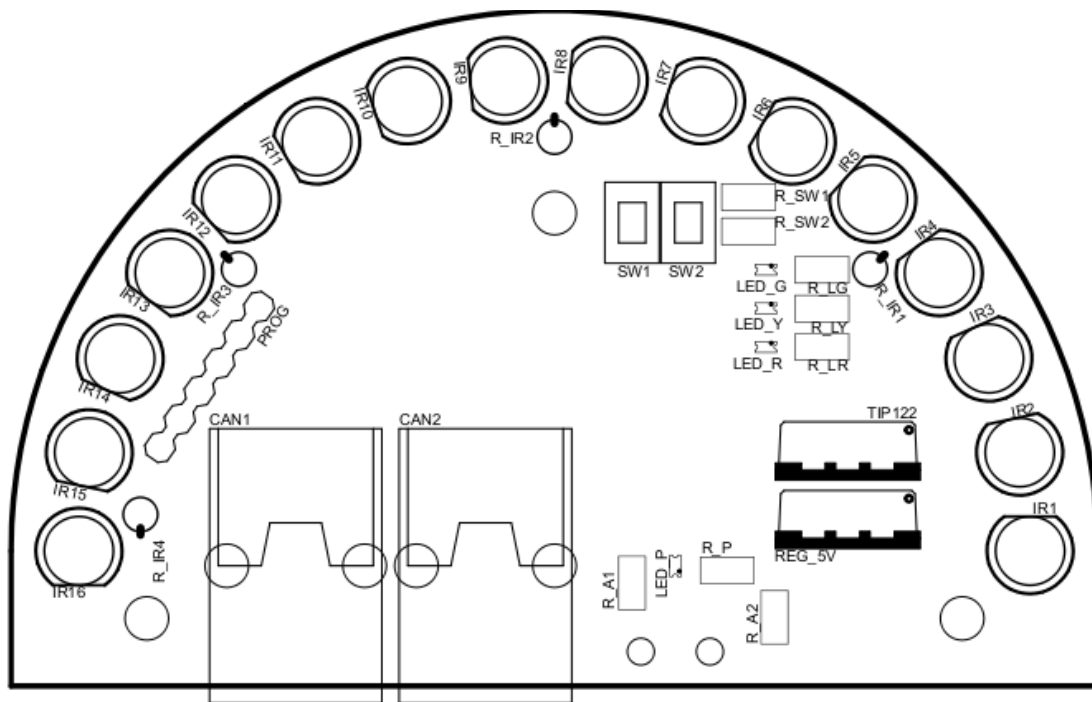


Top

5.9 TX Beacon implementations



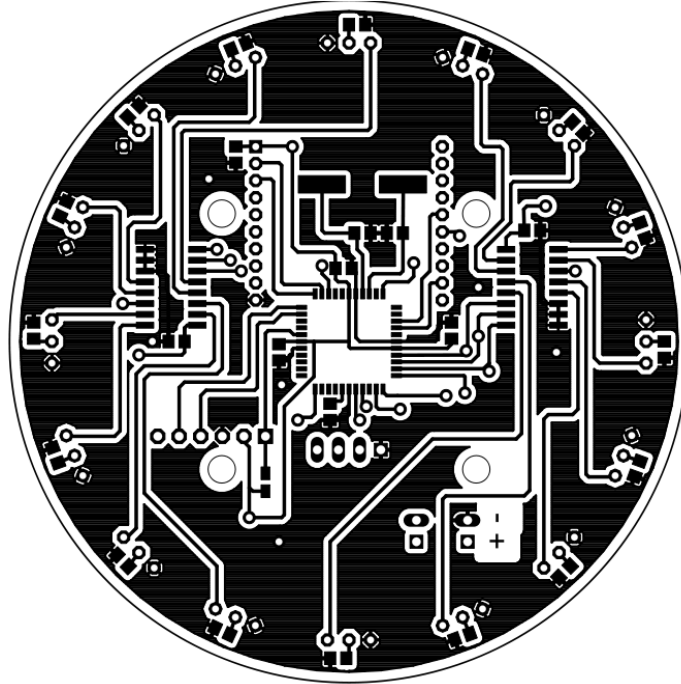
Bottom



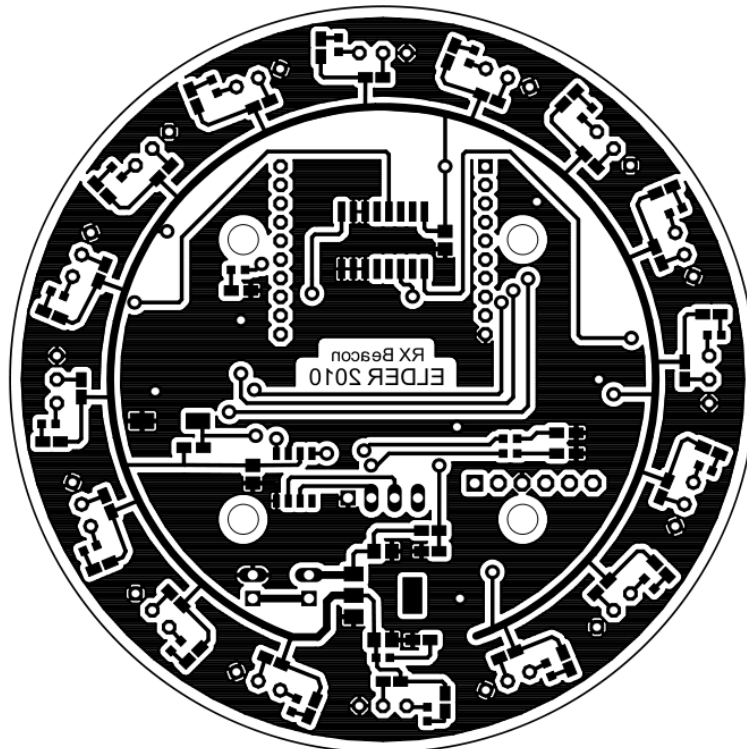
Top

5.11 RX Beacon masks

Actual size is 80x80mm.

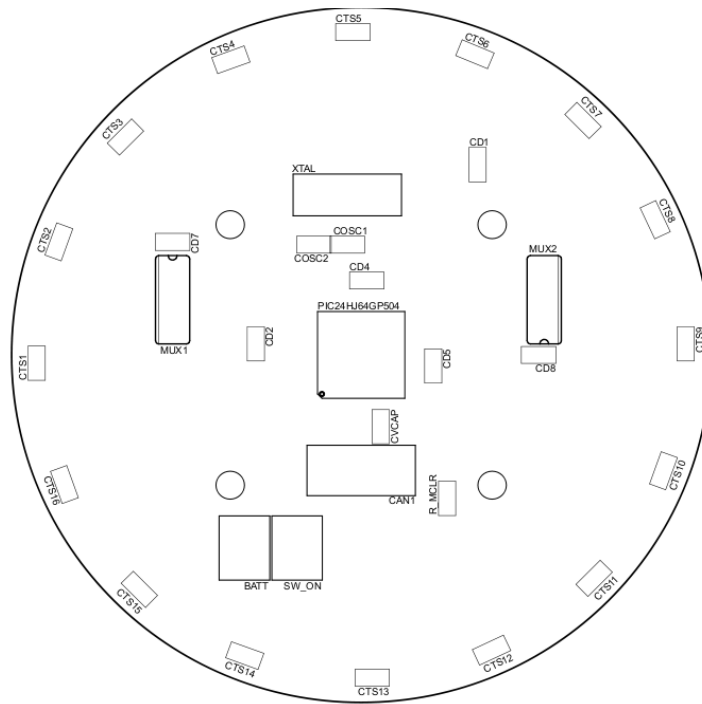


Bottom

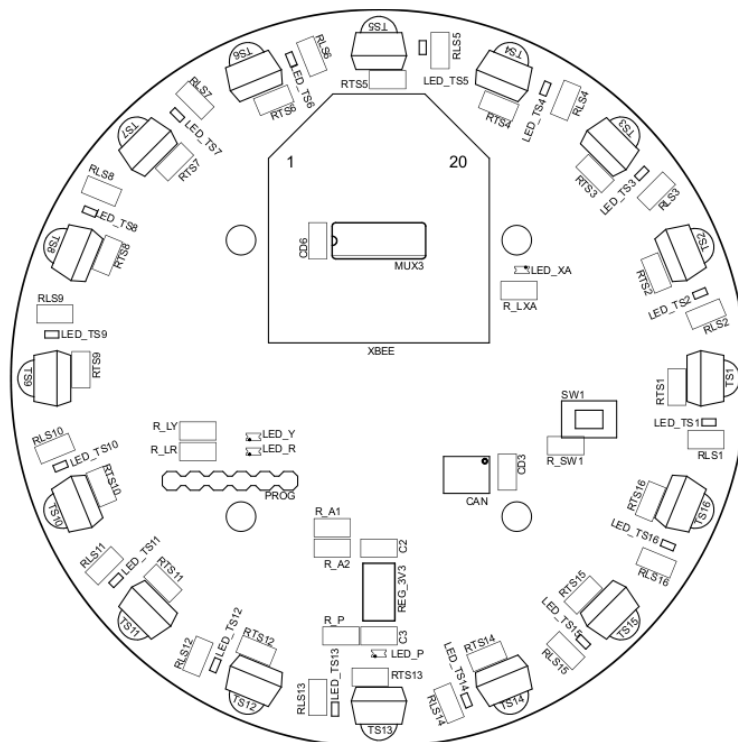


Top

5.12 RX Beacon implementations

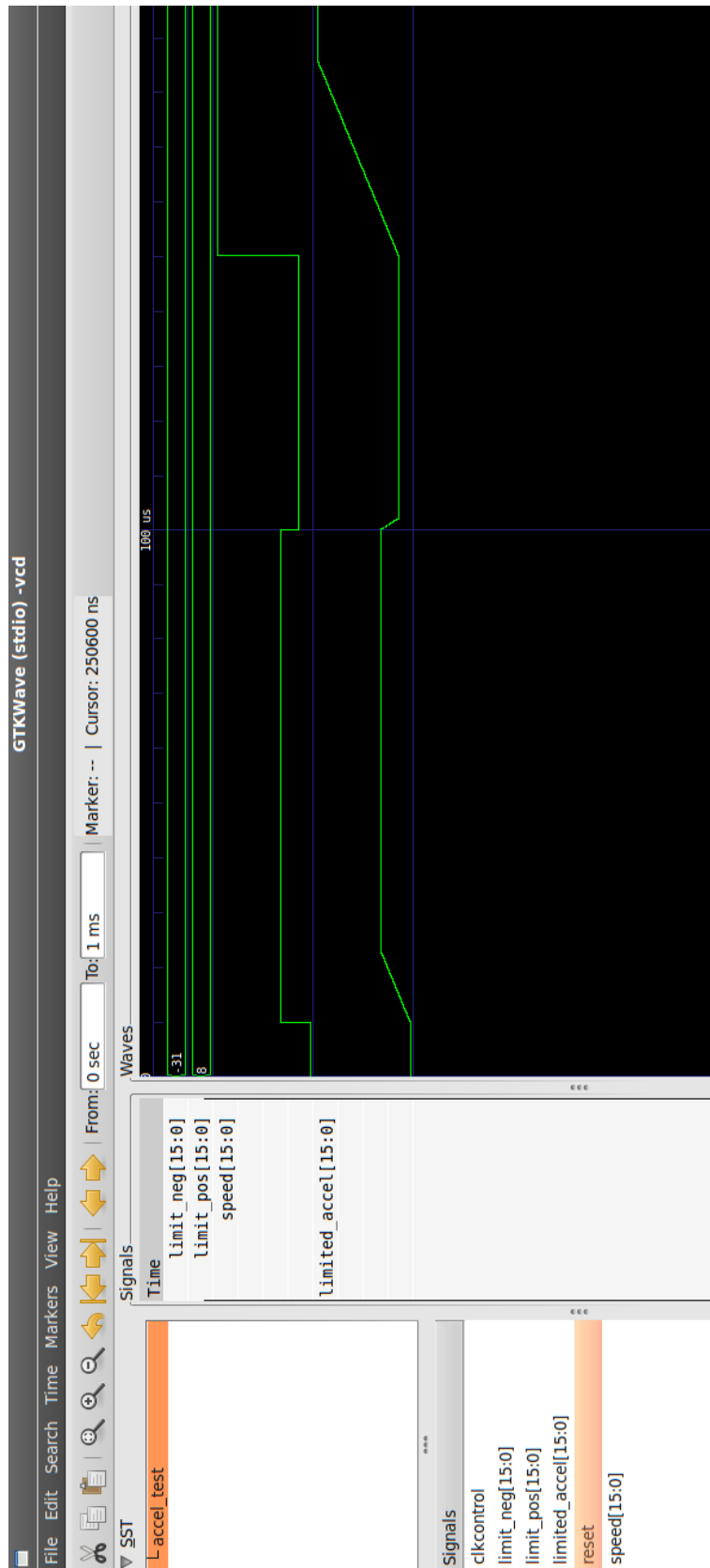


Bottom



Top

5.13 Gtkviewer sample output



5.14 Gnuplot sample output

