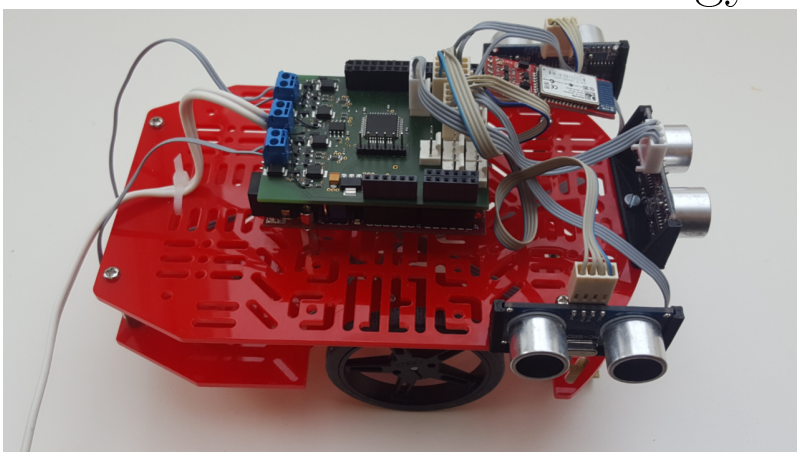
Uddannelse i virkeligheden

Fall Semester 2016

Autonomous Object Avoidance Robot

Group 2

3. Semester IT-Technology



Group members: Benjamin Nielsen - Henrik Jensen - Martin Nonboe - Nikolaj Bilgrau

Supervisors: Ib Helmer - Jesper Kristensen - Steffen Vutborg

IT-technology
Sofiendalsvej 60
9200 Aalborg SW
http://www.ucn.dk/

Title:

Autonomous Object Avoidance Robot

Project Period:

3. Semester | Fall semester 2016

Projectgroup:

Group 2

Group participants:

Benjamin Nielsen
Henrik Jensen
Martin Nonboe
Nikolaj Bilgrau

Supervisors:

Jesper Kristensen
Steffen Vutborg
Ib Helmer Nielsen

Pages: [1]

Appendices:

Completed:

Preamble

This project was written by group 2, for the 3rd semester on the IT-electronics education at university college Nordjylland, Sofiendalsvej 60. The project goal is to make an autonomous robot that can navigate a course utilizing object avoidance and localization.

Benjamin Nielsen	Henrik Jensen
Martin Nonboe	Nikolaj Bilgrau

Table of Contents

Glossary

PWM Pulse-width Modulation

IDE Integrated Development Environment

MCU Microcontroller Unit

UART Universal Asynchronous Receiver/Transmitter

IDE Integrated Development Environment

DC Direct Current

NPN Negative-Positive-Negative

CPLD Complex Programmable Logic Device

RTOS Real-Time Operating System

MOSFET Metal-Oxide-Semiconductor-Field-Effect Transistor

TACH Tachometer - an instrument measuring the rotation speed of a shaft or disk, as in a motor or other machine

Introduction 1

In countries with high wages and where manual labour is expensive, the industrial production is often organized as an automated process. To make the whole industry smarter and more customizable, new automated robots and reliabilities are needed. The many new forms of robots rely on sensors to face the many different challenges. Automation of movement and avoidance enables even robotic space exploration, as seen in the many rovers visiting the different nearby planets.

There are different sensors in play when needing to avoid obstacles or collision. To mention a few, ultrasound, infra-red and laser sensors comes to mind, all of which are viable picks when building a robot with object avoidance.

In the project at hand, we will be focusing mainly on the ultrasound sensor for building an object avoiding robot. The objective of this project is to design and implement an automotive robot capable of autonomous object manoeuvring, specially a collision avoiding robot employing light detecting sensing and ultrasound sensing.

The project was handed to the group at the start of third semester and is to be handed in at the 9th of January, 2017.

Analysis 2

This section will be focused on analysing any problems the group may face, how to approach them, and how they should be handled.

2.1 Problem statement

The problem presented to the group is how to make a robot move from point A to point B, with the help of different sensors, including ultrasound and infrared, and to make use of autonomous algorithms to avoid obstacles.

Problem statement:

- The robot should be able to move from A to B
- It should be able to stop at a predetermined point
- It needs to manoeuvre around obstacles

2.2 Problem analysis

2.2.1 Mobility from A to B

The robot receives a coordinate to reach, and will use its own starting point to determine a direction to drive towards the given coordinate. The robot will need a way to control its movement and direct current to function optimal.
The robot needs a way to effectively regulate speed and also steer itself autonomously. To dictate how quickly the robot moves, the robot will need some system that allows it to move around on a flat surface, the robot needs to be able to move around from point A to point B. .

2.2.2 Predetermined end point

After starting, the robot needs to know when to stop. The pre-determined end-point could consist of a series of circles which the robot needs to detect, or just be coordinates in a theoretical coordinate system.

2.2.3 Obstacle avoidance

As part of its functionality, the robot needs to be able to see objects that are in front of it and avoid them. After avoiding an obstacle, the robot needs to determine where it is compared to the goal, and go towards that again.

2.3 Behaviour

The robot is expected to fulfil the previously mentioned tasks on it's own. To do this, a few algorithms are to be utilized. This section will describe some of the ideas and considerations made by the group.

To solve the problem of going from A to B, 2 methods have been considered. Both of these assume that both the start and end point are known. The first and simplest method is simply to manually turn the robot towards the goal, and letting the robot drive until it detects the goal. The other, and chosen, method is to know the coordinates for the end compared to where the robot starts.

This method also solves the second problem by reading the tach count from the motors, and calculating the distance driven from these. The other method of finding the end point is by using a few sensors in a line sensing fashion, giving the robot some space to work with if the calculations are not perfect.

The last problem, obstacle avoidance, only one viable solution was discussed in the group: using 3 ultrasound sensors to detect obstacles around the robot. The placement of these, however, was considered. Due to previous experience with object avoidance, it was chosen to mount 1 sensor in the middle looking forward, and 1 to either side of this, turned 90 degrees(See figure **??**). This was chosen over having the side sensors turned 45 degrees because tests in the previous project showed ultrasound sensors to give inconsistent results when reading data from obtuse angles.

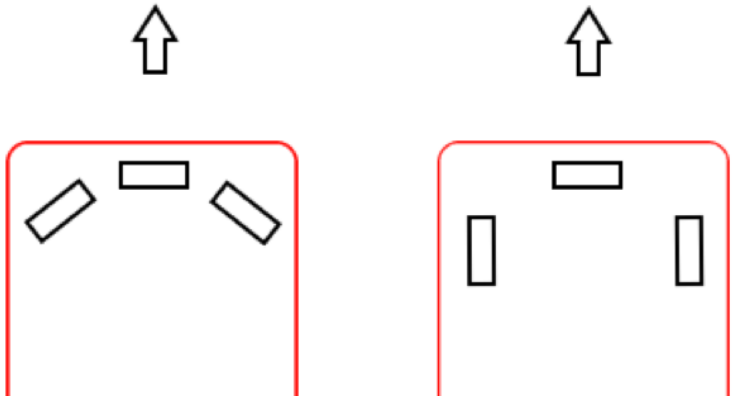


Figure 2.1: Proposed placement of sensors, right shows the chosen placement

Requirements specification 3

This section specifies the requirements. The requirements have been found through the analysis.

- The robot needs a way of detecting a goal
- The robot needs object avoidance
- The robot needs localization
- The robot needs to find a new route after avoiding an object
- The robot should make use of an H-bridge
- The robot should make use of motors
- The robot needs a way to implement motor control

Hardware section 4

4.1 Description of the hardware structure and functionality

In the following section the different parts, components and devices of the hardware will be listed, described and explained. This includes topics such as the hardware diagram, considerations involving what sensors to use, the MCU, the self designed h-bridge and the motors with the encoders.

4.2 Hardware diagram

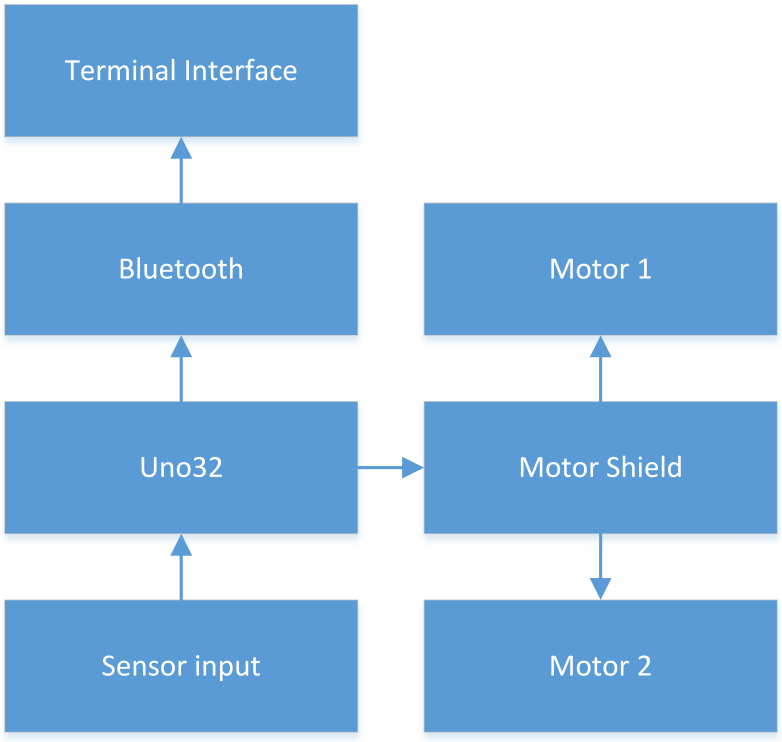


Figure 4.1: Hardware diagram with arrows

The micro controller is connected to the motor-shield. The motor-shield is then connected to the two motors, called motor 1 and motor 2 and is powering both motors. The micro controller is receiving data from the sensor set; the ultrasound sensors and the tachometer sensor. The micro controller then receives the data from the sensors and sends it further to the Bluetooth transceiver and then the Bluetooth transceiver will send it to the interface.

4.3 Sensors and sensor concept

The robot will utilize 3 ultrasound sensors. These will be working together to make the robot able to navigate open spaces more precisely. The 3 sensors are able to cover blind spots for each other and in this way provide for a more solid solution to object avoidance. They are mounted in a way such that there is one facing forwards, and one pointing out sideways from these, in a roughly 90 degree angle. This works for the robot in multiple ways; it covers blind spots and also allows the robot to "see" to its sides while moving straight forwards, this allows for more fluid object avoidance, since the robot can cut down on the amount of turns it has to make to follow an object to its side.

4.3.1 Choice of sensors

To find out what ultrasound sensors to use in navigating the robot, two sensors were compared briefly on their specifications; The HC-SR04 and the PING))) Ultrasonic Distance Sensor.
The specifications have been put into a table for a quick perspective into the sensing distance, volt, current and price for each sensor.

As seen on the table; the two sensors with slight differences, the HC-SR04 can measure an extra meter compared the PING)). Additionally, there's a staggering price difference of approximately 200,- DKK between the HR-SR04 and the PING))), which is a concerning factor when producing multiple robots.

In the case of this project, the device available is the HC-SR04. Even if the PING))) was available, the HC-SR04 has 4 pins compared to the PING))) with only 3 pins, this feature can make it easier to manage when coding the sensor.

Name	HC-SR04	PING)))
Max/min sensor distance	4m-2cm	3m-2cm
Working current	15mA	30mA
Voltage	5DC	5DC
Price	17.63,- DKK	211.53,- DKK

Table 4.1: Sensor table [**HC-SR04**] [**PING**]

Even if availability wasn't a limitation, the clear choice would be the HC-SR04, due to the more impressive range and the better pricing of the HC-SR04.

4.3.2 Ultrasound sensor - HC-SR04

When a robot should be able avoid obstacles it will need a device to inform the robot where it's position is compared to the obstacle. This is where an ultrasound sensor plays an important role. For this task the HC-SR04 has been picked.

Figure 4.2: The HC-SR04 ultrasound sensor

The way the ultrasound sensor works is by emitting acoustic waves and then waits for the waves to reflect back to the sensor. The waves are often at about 40 kHz and humans are unable to detect the sounds because of the frequencies being above the human audible range.
What is causing the device to make ultrasonic sound is a piezoelectric crystal. The crystal is receiving a rapid oscillating electrical signal, this causes the crystal to expand and contract and thereby creating a sound wave. The sound waves will then after being reflected return to a piezoelectric receiver which can then convert the waves into voltage by using the same method as explained above.

In the scope of the project, the robot will be using "Time of flight" for sensing the distance between itself and the obstacle.
The ultrasound sensor only generates pulses of sound instead of an continuous streak of sound waves. to avoid confusion. In high speed situations this will mean there is waiting time limits.

The calculation for using the ultrasound sensor is:

t = time
r = distance to object
c = speed of sound

r = c*t/2

With this the robot can calculate the distance to the object.

Considerations:

When using the ultrasound as a sensing tool, there are some factors that must be taken into consideration.
Temperature and humidity can affect the speed of sound, just as air currents have been known to be able to create invisible boundaries that can reflect ultrasonic waves.

Ultrasound sensors have something called a dead zone, this occurs when an object is in front of the sensors and the receiver can't keep up.
Some materials are very absorbent, which will result in less reflected ultrasound to be detected by the receiver.

Mounting

For putting the sensors in place facing the right directions, a 3D-printed mount was made and then bolted to the chassis of the bot.

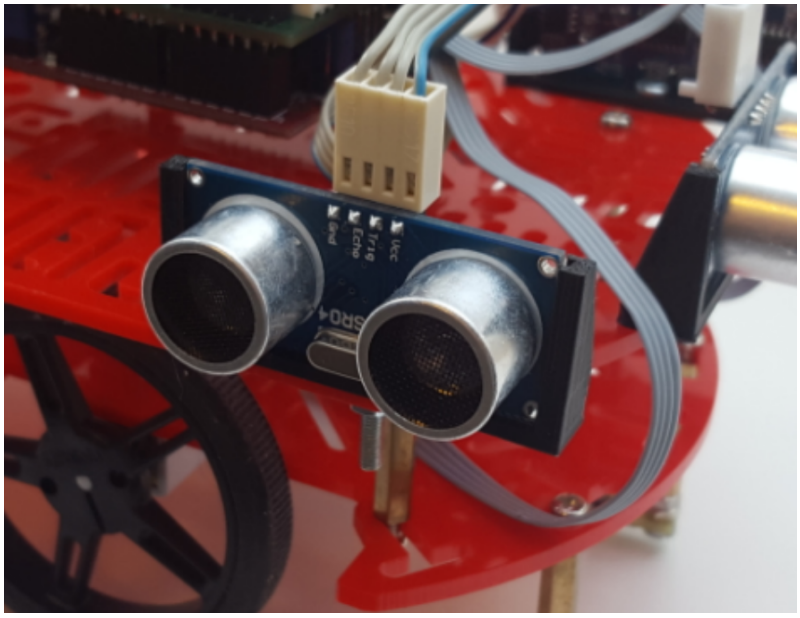


Figure 4.3: [Sensor mount]Mounts made by 3D printing, for ultrasound sensors

4.4 The chipKIT Uno32 board

The robot will utilize the chipKIT Uno32 board to execute code and act as a controller for the sensors, motors, and the Bluetooth module. The board was chosen both due to past experiences. It's proven to be simple to work with and provide multiple ports to make it possible to utilize more than enough modules for the robot's functionality.
The board is also compatible with Arduino shields, and as such designing the H-bridge for it becomes more straightforward. It's fast enough to execute the code, and works well within the input power range that the robot will operate within.

4.5 The motor shield

The motor shield is the single add on board used in the project, and contains all the features needed for making the robot work. The features are:

- Dual H-bridges.

- Low side current sensor for each H-bridge.
- CPLD for reconfigurable H-bridge logic control.
- All connectors needed for sensors and other units needed:
  - Screw terminal for motor connection.
  - Screw terminal for input power.
  - Molex connector for ultrasound distance Sensor.
  - Molex connector for distance sensor.
  - Molex connector for motor encoder.
  - Molex connector for infrared light sensor.
  - Header for Bluetooth module.

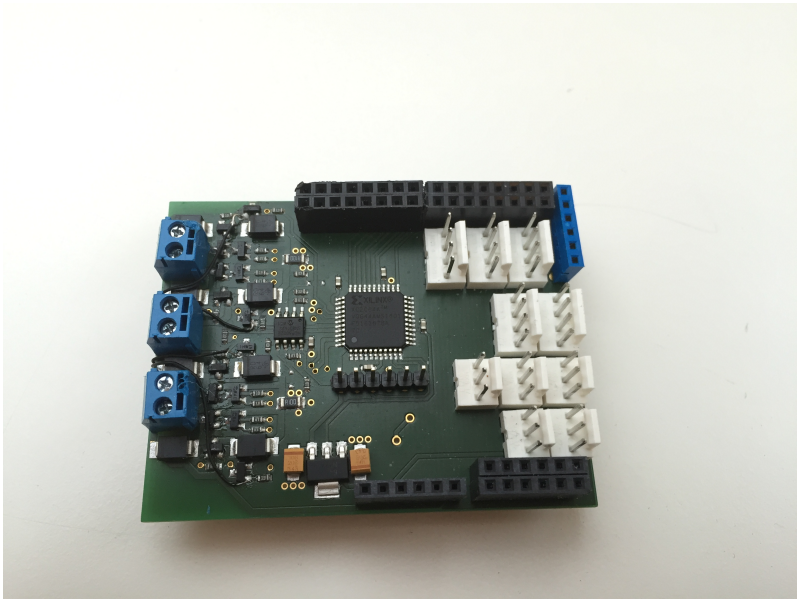


Figure 4.4: The self-designed motor-shield

4.5.1 The H bridge

The robot will make use of an H-bridge. An H-bridge is a circuit made for controlling the motor of the robot, by making sure the motor will never try to do forward and backward motion and cause errors or short circuits. The point of using an H-bridge is to ensure motor safety and functionality.

4.5.2 Pololu 100:1 Micro Metal Gearmotor 6V High Power

This is a small motor, drawing 120mA when there is no load, and 1600mA when stalling. They run up to 320 RPM; because of this, the robot will be able to move very rapidly. The ones used for the robot have an extended motor shaft, which makes it possible to utilize the Pololu motor encoders.

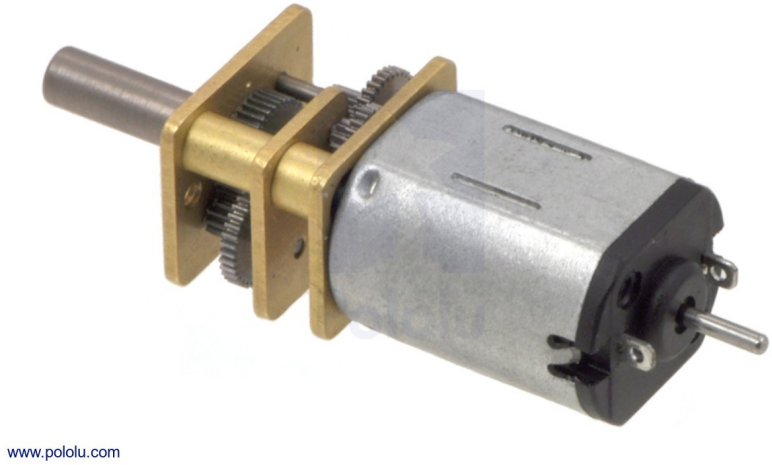


Figure 4.5: Pololu micro metal gear motors

Pololu Magnetic Encoder Pair Kit

These encoders allow the transmission of data based on motor movement. It works by having the encoder board count the revolutions of the magnetic disc mounted on the board. It does this twelve times per revolution. The transmission of this data allows the robot to monitor how long it has traveled. This data, along with directional data coming from the code, will allow the robot to track its own movement in a coordinate system. This way, the robot will always start on (0, 0), and if given a destination coordinate to navigate towards, it will be able to both find the most efficient way there, but also track its movement throughout the runtime.

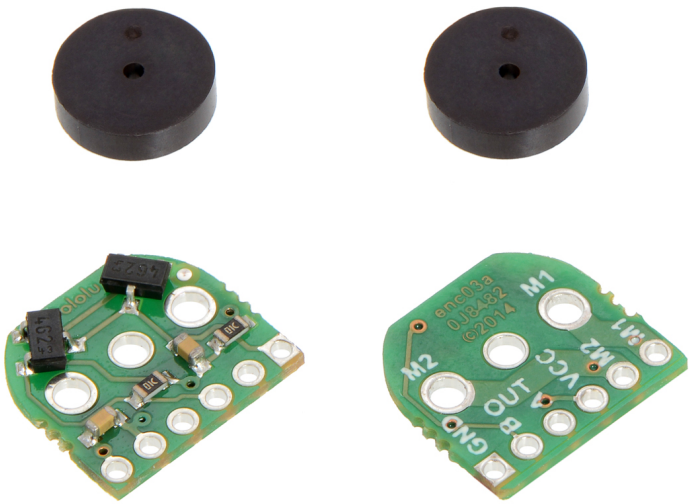


Figure 4.6: Pololu Magnetic Encoder Pair Kit

4.6 The Bluetooth tranceiver

The robot will utilize the BlueSmiRF Silver bluetooth tranceiver. The transceiver is made by Sparkfun, it is utilized to make use of an GUI, by sending data from the MCU to the computer (GUI) by the use of bluetooth.
The bluetooth tranceiver makes it possible to monitor both the inputs and the logic behind the steering. The baudrate is between 2400-115200 bps and the tranceiver can be powered from 3.3v up to 6v.

4.7 Part conclusion

After initial H-bridge problems, the rest of the process of building the robot went according to plan, and there were no future issues. The robot utillize a range of components which have been used for previous projects, which made the project much more simple to work with. This eliminated some of the learning curve that the previous robot presented, and made it possible to plan out and assemble the robot very rapidly, even though a lot of time was spent waiting for components for the motor shield, and the faulty components. This way, a lot more time could be used on programming and other software solutions.

Software section 5

The following section will introduce the most important parts of the software for the project. The design of the software will be shown as a flow chart and described. The section will focus on the feedback loop used, but will also go through some of the smaller parts, such as PWM and the code used for the CPLD.
Pulse Width Modulation is a very effective and straightforward way to control the speed of the robot rapidly. It works by limiting how long the of a given period the power is 'on' compared to 'off'. PWM is used by utilizing Output Compare on the chip, which lets the chip generate pulses based on a timer. This is initialized in the following way:

5.1 Software flowchart

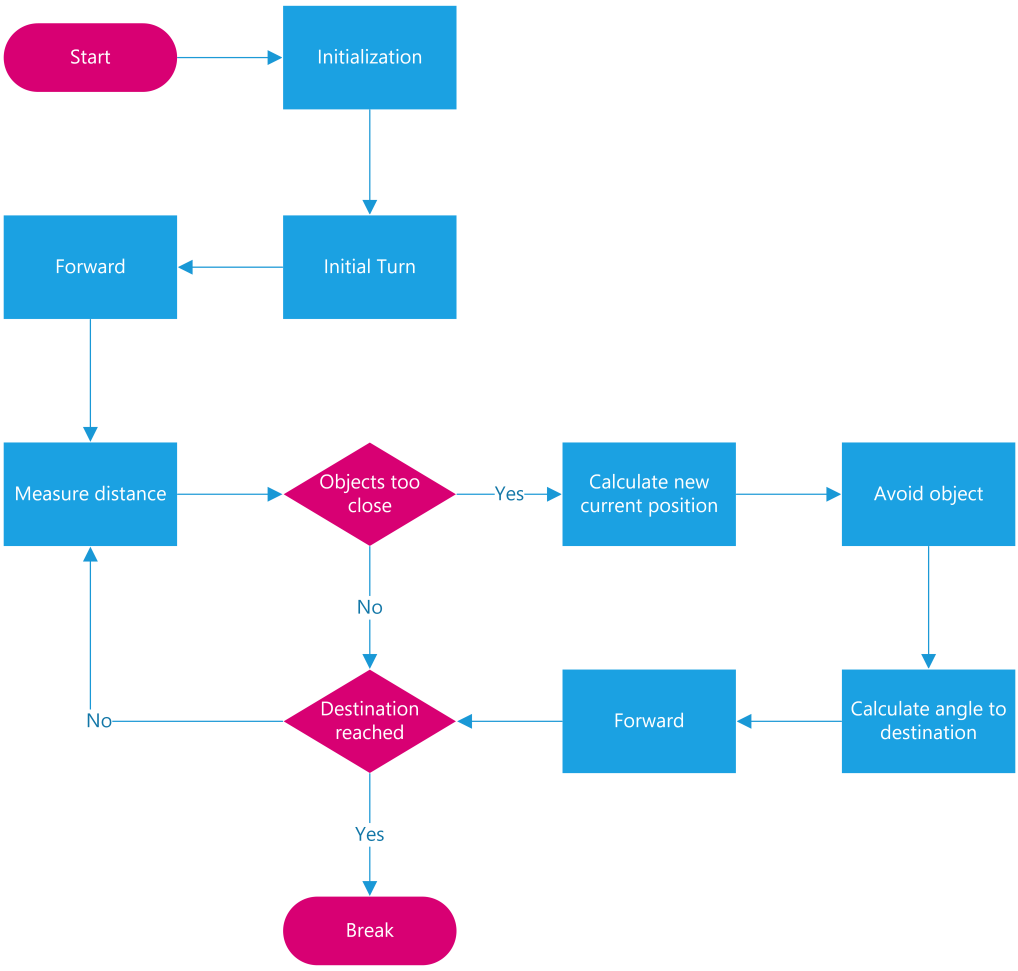


Figure 5.1: Software flowchart

5.2 Pulse-width modulation

The robot utilizes pulse-width modulation by initiating a desired frequency, setting up the output compare, starting a timer, and then finally calculating the trigger timing of the timer.

```
1  void initPWM() {
2      int sysClk = 80000000; //FPB
3      int pwmFreq = 1000; //Desired frequence
4      int prescaleV = 1;
5      int dutyCycle = 0;
6
7      PMCONbits.ON = 0;
8      PMAEN = 0;
9
10     OC4CON = 0x0000; //Turn off the Output Compare while setting
           up
11     OC4R = 0x00638000; //Config compare Register, rising edge
12     OC4RS = 0x00638000; //Secondary compare Register, falling
           edge
13     OC4CON = 0x0006; //Turn on Output Compare in PWM mode
14     OC4RS = (PR4 + 1)*((float) dutyCycle / 100); //Sets the duty
           cycle, RS = time until falling edge starts
15     OC4CONSET = 0x8020; //Enable peripheral,  bit 5: 0=16 bit
           compare mode, 1=32 bit
16
17     OC5CON = 0x0000; //Same as above
18     OC5R = 0x00638000;
19     OC5RS = 0x00638000;
20     OC5CON = 0x0006;
21     OC5RS = (PR2 + 1)*((float) dutyCycle / 100);
22     OC5CONSET = 0x8020;
23
24     T2CONSET = 0x0008; //Starts a 32-bit timer
25     T2CONSET = 0x8000; //Enables the timer
26
27     PR4 = (sysClk / (pwmFreq * 2) * prescaleV) - 1; //Calculate
           how often the timer should trigger
28     PR2 = (sysClk / (pwmFreq * 2) * prescaleV) - 1;
29 }
```

5.2.1 Duty cycle

The duty cycle is used to describe how long the power is 'on' compared to 'off'. A higher duty cycle will yield more energy than a low one. The software uses a frequency of 1000Hz, which makes it straightforward to calculate to real time, if this is needed - it also provides enough precision to make the motors responsive quickly. The duty cycle can be changed by changing how long until the Output Compare sends a falling edge:

```
void adjustDuty(int channel, int duty) {//The function takes an
    argument based on which motors PWM should change, and the
    desired duty cycle
     switch (channel) {
         case 1:
             OC5RS = (PR2 + 1)*((float) duty / 100); //Sets the
                 secondary register, to tell it how long until it
                 should send a falling edge
             break;
         case 2:
             OC4RS = (PR4 + 1)*((float) duty / 100);
             break;
     }
}
```

5.3 Feedback loop

A feedback loop is a way of controlling how something, in this case a robot, behaves by receiving an output and adjusting the performance to match a desired output. The robot made in this project utilizes a feedback loop by firstly turning towards a set goal, and then avoiding obstacles along the way through sensors measuring the distance from the robot to the obstacle. If an obstacle gets too close, the robot will turn away from the obstacle and head back towards the goal.

5.3.1 Reading sensors

To know how far away an object is, some form of sensor feedback is needed. In this case, 3 ultrasound sensors have been implemented. The microprocessor sends a turn-on signal to the sensors, starts a timer and then waits until the sensors return with their own signal. The time between the start and finish signal can then be used to calculate the distance between the robot and the object.

```
long readUltrasonic(int channel){
    long timerFinish = 0;
    long timerOld = 0;
    int timeout = 3000; //Sets a limit for how long the MCU waits
        for data
    switch(channel){ //Switches between the 3 sensors
        case 1:
            Trigger1 = 1; //Pin RG6
            DelayUs(10); //Sends a short pulse
            Trigger1 = 0;
            while(Echo1 == 0){} //Waits until pin RF6 is set to
                high
            timerOld = micros(); //Sets the start timer to time
                since the program started, in microseconds
            while(Echo1 == 1){ //While the sensors signal is high
```

```
13                  timerFinish = micros(); //Sets the end timer to
                        check for timeout
14                  if(timerFinish - timerOld > timeout ||
                        timerFinish - timerOld < 0) //If the time to
                        return is too high or below 0, return a
                        default value
15                      return 500;
16              }
17              break;
18 ...
19      }
20      timerFinish = micros();
21      long result = ((timerFinish-timerOld)*0.34)/2; //Calculate
           distance from speed of sound, divided by 2 since the sound
            wave also needs to return to the sensor.
22      return result;
23  }
```

The short pulse sent in the beginning of the function has been set by the manufacturer: [**HCSR04Datasheet**]

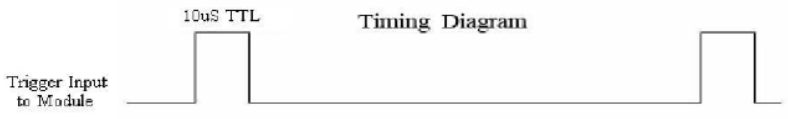


Figure 5.2: HC-SR04 timing diagram

To reduce false sensor data, which there seems to be a lot of with these sensors, a rolling average has been implemented. This calculates the average of the previous 5 readings instead of a single reading, allowing the robot to ignore spikes in data.

5.3.2 Feedback

The controller function takes the data from the 3 sensors and uses these to control how the robot should behave. The function is by no means fully optimized, but allows the robot to avoid obstacles in it's way.

```
1  void controller(int midSensor, int rightSensor, int leftSensor){
2      int midDistance = 100; //Sets the limit for the sensors, in
           millimetres
3      int sideDistance = 50;
4      if(midSensor < midDistance) //First check the middle sensor
5      {
6          brake(); //Stops the motors
7          calculatePosition(); //Defunct, but should calculate
               where the robot is now in a coordinate system
8          backwards(); //Starts backing away from the object
9          int tachTarget = tach1-300; //Sets a tach target
10         while(tach1 > tachTarget){} //Drives backwards until
               hitting the target
11         brake(); //Brake again
12         presetTurnLeft(); //Turn 30 degrees left
13         brake();
14         forwards(); //Start driving forwards again
15     }
16     else if(leftSensor < sideDistance)
17     {
18         int tachTarget = tach1-20; //Sets the target tach
19         turnRight(); //Turn right
20         while(tachTarget > tach1){} //Until target tach is hit
21         brake(); //Brakes
22     }
23     else if(rightSensor < sideDistance) //Same as left, but
           reverse
24     {
25         int tachTarget = tach1+20;
26         turnLeft();
27         while(tachTarget < tach1){}
28         brake();
29     }
30     else
31         forwards(); //If no sensor is within the limit, just
               drive forwards
32 }
```

5.4 CPLD

Compared to a regular microprocessor like the PIC32MX320F128H that is programmed in C converted to machine code in a procedural manner:

```
1 if(this) {
2   then this
3 } else {
4   this
5 }
```

a CPLD is programmed with what is called Hardware description language or HDL for short. instead of writing condition statements like you do in C, you define logic blocks and describe what that logic block do with the inputs and outputs given. [**HDL**] This allows for programming small logic circuits that act like several 74-series logic chips put together, but a CPLD can also be configured to contain a small microcontroller what internally can interpret machine code.
This make a CPLD a very powerful device since it can be configured to anything that can be made with pure logic circuitry, given that the selected CPLD have the needed amount of internal configurable logic blocks for the implementation. [**CPLD**] To simplify the programming, the IDE used to program the CPLD used on the motor shield, allows for programming by drawing schematics, which the IDE then synthensize and translate into the format the can be uploaded to the CPLD.
The CPLD on the motor shield is configured to allow full individual control of each h-bridge by using 4 control lines from the PIC32 to the CPLD for each h-bridge:

- Enable
- Direction A
- Direction B
- PWM

Using these inputs without any logic control, allows for unwanted configurations of the h-bridge that will cause the power supply to be shortet to ground, this issues have been solved by programming the CPLD with a logic block that does not allow the configuration to occur, no matter the configuration of the input to the CPLD. **??** shows the logic configurations of one of the two logic blocks that have been programmed into the CPLD.

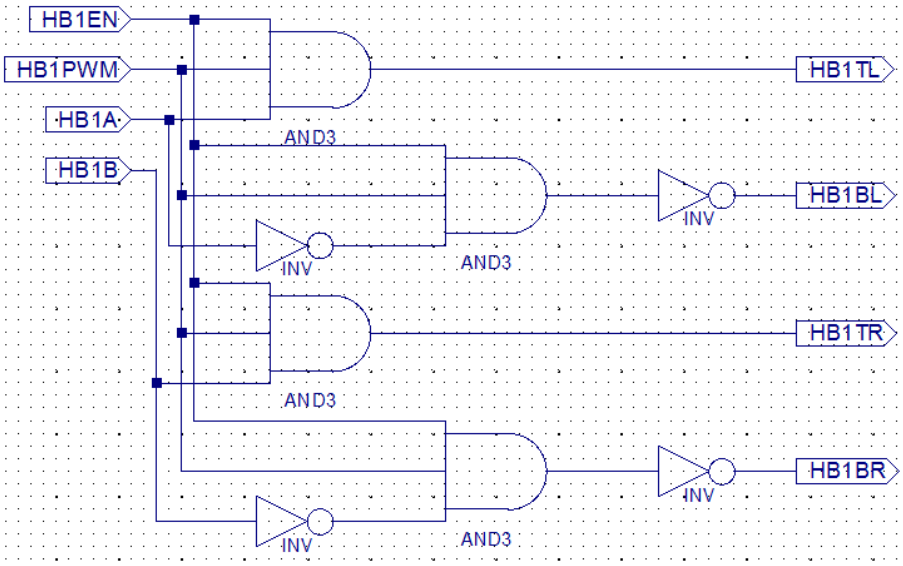


Figure 5.3: Single h-bridge logic block

5.5 Part conclusion

A few issues occurred during the development of the software, most importantly being the localization code. This part was omitted from the final project due to time constraints since it is only half way done. The code should have functioned by calculating the distance the robot has travelled by using cosine and sine functions, since it was always possible to generate a theoretical triangle from where the bot starts and where it starts turning.
This also meant that the calculations for turning towards the goal after avoiding an object did not work, since this was planned to be made from the robots position relative to the start and end point.

A consideration made in the project was to implement a real time system, such as RTOS, for the program. RTOS would make it possible for the sensors to be read simultaneously instead of one after the other. This could make a huge difference in time constricted systems, however, the system used in this project does fine without, since reading all 3 sensors take less than 10 ms.

Due to time constraints, a few requirements for the project have not been met. These include the need for line following capabilities, localization and the ability to return to a path that leads to goal.

Test 6

In this section the different parts and signals will be tested, this is to make sure everything is up to par with our requirements specifications. The tests consists of measuring different factors, such as current, voltage and response time of each fundamental part. The testing will be done with the available measuring devices such as an oscilloscope, power supply or multimeter.
[1]

FiXme Fatal: ALT TEST SKAL TJEKKES IGENNEM

6.1 Unit Testing

6.1.1 Ultrasound HC-SR04

The purpose of this test is to measure is the HRC-S04 Ultrasonic distance sensor is reliable at reading the distance to an object.

Equipment

- Agilent MSO-X 3024A Oscilloscope
- Arduino UNO Microcontroller

Setup

A small program for the Arduino has been written which allows the MCU to trigger the sensor and waits a pre-specified amount of time before triggering again, to allow for the return of the ultrasound wave.

Results

In order to to validate to the results we can calculate the actual measured length using the speed of sound, which is $340.29\frac{M}{s}$ or $2.9\frac{\mu s}{mm}$. To calculate the distance we can use: $(\frac{time}{speed of sound})/2$ [**SOF**]

Test distance	Time Pulse measured	calculated distance	Difference
30mm	152	26.2mm	3.8mm
50mm	285	49.1mm	0.9mm
100mm	600	103.4mm	3.4mm
200mm	1.32	227.6mm	27.6mm
500mm	2.32	400mm	100mm

Table 6.1: Ultrasound test results

The results show that when measuring objects closer that one meter the sensor is accurate within spec of $\pm 1cm$

[1] FiXme Fatal: ALT TEST SKAL TJEKKES IGENNEM

6.1.2 DC Motors

The purpose of this test is to measure the power requirements of the DC motor used on the robot under different scenarios.

Equipment

- Elcanic Power Supply
- Fluke 45 Multimeter

Setup

The setup consist of a single DC motor connected to the Elcanic Power Supply with on of the power leads passing through the Fluke 45 Multimeter for current measurements.
The motor will be tested in the following scenarios:

- no load.
- Under load.
- Stalled.

Results

Scenario	Current measured
No load	150mA
Under load	750mA
Stalled	2A +

Table 6.2: DC Motor test results

During the test it was observed that when testing the motor while stalled it drew more than two amps. This was apperant as the Elcanic power supply kept going into over current protection mode when doing the test and it is not able to supply more current. The test will not be made with a more powerful power supply to the risk of damaging the motor.

6.1.3 CPLD

The purpose of this test is to verify that CPLD performs the correct logic task to control the dual h-bridge solutions.

Equipment

- Agilent MSO-X 3024A Oscilloscope
- Agilent 54620-61601 Logic Analyzer Probe Cable
- ChipKit Uno32 Microcontroller
- Elcanic Power Supply

Setup

For the test logic probes was used to analyze the logic signals comming from the Uno32 into the CPLD aswell as the outputs from the CPLD going to the h-bridges.

Results

To compress the results the output will be in the following format: The results are

Logic Input				**Expected output**				**Recieved output**			
En	PWM	A	B	TL	BL	TR	BR	TL	BL	TR	BR
Bit 3	2	1	0	Bit 3	2	1	0	Bit 3	2	1	0

Table 6.3: Results example

from testing according to the truth table used to create the logic circuit programmed into the CPLD.

Logic Input	**Expected output**	**Recieved output**	**Pass**
0000	0101	0101	Yes
0001	0101	0101	Yes
0010	0101	0101	Yes
0011	0101	0101	Yes
0100	0101	0101	Yes
0101	0101	0101	Yes
0110	0101	0101	Yes
0111	0101	0101	Yes
1000	0101	0101	Yes
1001	0101	0101	Yes
1010	0101	0101	Yes
1011	0101	0101	Yes
1100	0000	0000	Yes
1101	0011	0011	Yes
1110	1100	1100	Yes
1111	1111	1111	Yes

Table 6.4: CPLD logic results

6.1.4 H-Bridge

The purpose of this test is to verify that the two h-bridges functions as intended.

Equipment

- Elcanic Power Supply
- Fluke 45 Multimeter
- ChipKit Uno32 Microcontroller

Setup

The h-bridge i controlled with the Uno32 through the logic circuit programmed into the CPLD. The Current draw and voltage of the entire system is measured with two Fluke 45 multimeters. Input power is supplied by the Elcanic power supply.
The Uno32 is programmed to test the h-bridge in the following four configurations:

- Coast
- Power from top left to bottom right
- Power from top right to bottom left
- Break

Results

H-bridge 1	H-bridge 2	Over Current
Coast	Off	No
Power Dir A	Off	Yes
Power Dir A	Off	No
Break	Off	Yes
Off	Coast	No
Off	Power Dir A	No
Off	Power Dir A	No
Off	Break	No

Table 6.5: Results from first test of h-bridge

The results from the first test shows that there is a fault in h-bridge 1.
The first procedure in order to fix the fault was to add pulldown resistors to all base connections of the high side MOSFET's. This did not fix the problem.
Due to the the scenarios causing the fault the problem was believed to be either MOSFET Q9 or Q11.
Both of these MOSFETS where replaced for new ones, but this did also not fix the fault.
After replacing the MOSFET's didn't work, all MOSFET's were de-soldered from the PCB in order to test if the fault was something other than the MOSFET's.

The CPLD was configured to allow for direct signal pass through from the Uno32 in order for easier test and debugging of both h-bridges
The Uno32 was programmed to generate a square wave on all eight pins, since this will make it easier to detect if any driver transistors have failed.

Transistor	**Signal OK**
Q5	Yes
Q6	Yes
Q7	Yes
Q8	Yes
Q13	Yes
Q14	No
Q15	Yes
Q16	Yes

Table 6.6: Test of driver transistors

The test shows that transistor Q14 is having issues with switching on and off.
Additional measurements was made to confirm the fault in Q14. These results were:

	Base	**Collector**
Working trainsistor	0.854V	0.073V
Faulty transitor	0V	0V

Table 6.7: Voltage measurements of transistor Q14

With the results of the tests it was concluded that transistor Q14 was faulty and it was replaced.
This resultet in both h-bridges working as intended under all four scenarios tested previously.

6.2 Integration Testing

6.2.1 PWM motor control

The purpose of this test is to verify if the DC motor can be controlled by the PWM signal comming from the Uno32 going to the H-bridge.

Equipment

- Agilent MSO-X 3024A Oscilloscope
- ChipKit Uno32 Microcontroller
- Elcanic Power Supply

Setup

The Uno32 is used to generate the control signals used for the H-bridge with varying PWM duty cycles. the DC RMS Voltage is measured with the oscilloscope on both terminals. the H-bridge input power is provided by the Elcanic power supply.

Results

All test was done with a supply voltage at 7.19V

Duty Cycle	DC RMS Voltage
0%	0V
10%	4.10V
20%	4.72V
30%	5.27V
40%	5.65V
50%	5.99V
60%	6.22V
70%	6.49V
80%	6.73V
90%	6.93V
100%	7.07V

Table 6.8: Dc motor control with PWM results

The test shows that the motor shield is able to control the DC motor with PWM allowing for the Uno32 to control the amount of power delivered to the motors.

6.3 System Testing

Equipment

- Elcanic Power Supply

Setup

A small course was made for the robot to run, consisting of a 400x200cm area with the start at 0x0 and end at 400x200. A few objects were placed on the course for the robot to avoid. The robot was powered through the power supply for easy shut off of power if necessary.
This test was done a few times to get a consistent result.

Results

The initial turn of the robot towards the end point worked fine, although the motors on the robot being quite aggressive, making the robot jump a bit when breaking, resulting in slight misalignments.
Whenever the robot detected an object, decent adjustments were made to the route driven, though some were a bit too hard, making the robot turn very far away from the object.
Whenever the robot encountered an object directly in front of it, it was clear to see that the distance it backed was too large, and was reduced significantly. Furthermore, the distance for the front sensor was seen to be too small, making the robot react too late for certain objects and was increased to more than double the initial value.
During testing, one of the motors started having problems, ultimately leading to the motor cutting off completely.
Because of the localization code not functioning, an acceptable test was never found, not helped by the malfunctioning motor.

Conclusion 7

The goal of this project has been to modify the robot used in the previous project, to be able to go from A to B while avoid different obstacles on the way. This was done by using feedback from a few ultrasound sensors and using this data to calculate how far away the robot is from any obstacle ahead or to the sides of the robot, and react accordingly.
Furthermore, the project was to include some form of localization, so the robot can return to its course after avoiding an obstacle.

We had a considerable amount of trouble with the hardware, especially a motor breaking down during system testing, and a driver transistor malfunctioning on the H-bridge. This meant that a lot of time was spent on fixing these, ultimately leading to problems further on.

The software part went quite smooth, except for the localization part of the code, which was never completed due to time spent fixing faulty hardware. This also resulted in the project not fulfilling all of the requirements in the form of not being able to return to a path towards the goal.

During the project, a deeper understanding of programming the CPLD through Xilinx ISE WebPACK was achieved. This included setting up logic gates and declaring pin layout of the CPLD.
During the project, we have designed and assembled a PCB consisting of an H-bridge, CPLD with power supply and connectors for the sensors. This has given us an insight into using EAGLE, and problems that lie in designing a PCB.
In addition to this, a lot of time spent fixing problems has given us a further knowledge of using the different measurement tools, including the logic analyzer for an oscilloscope and a current probe.

The project started out looking positive, with the initial tests showing that the individual parts worked fine on their own. When put together, however, problems started occurring with different parts.
Due to above mentioned problems, the project ended up being unfulfilling when put in the scope of the requirements specified in the beginning of this report.

Appendices 8

List of Figures

List of Tables

Hardware appendix 9

9.1 Hardware Schematics

9.1.1 H-bridge 1

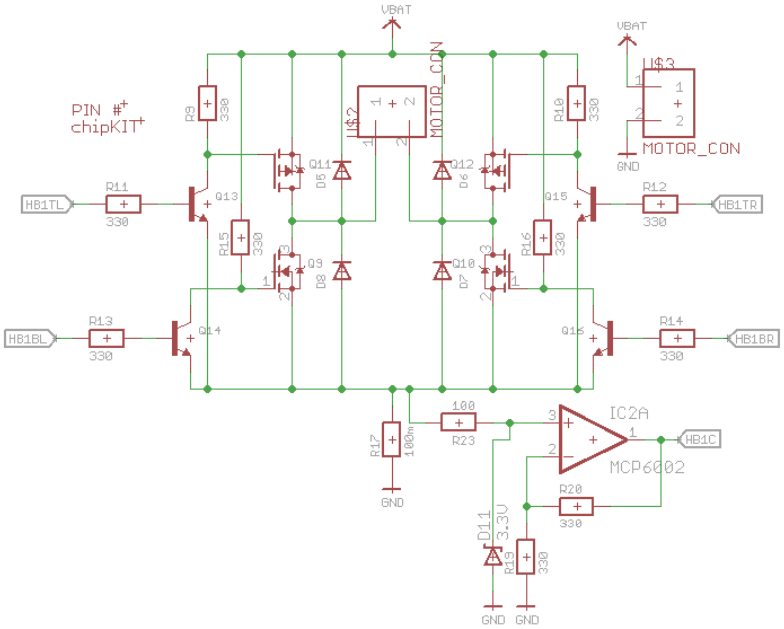


Figure 9.1: H-bridge 1 Schematics

9.1.2 H-bridge 2

9.1.3 CPLD

9.1.4 Power

9.1.5 Board schematics

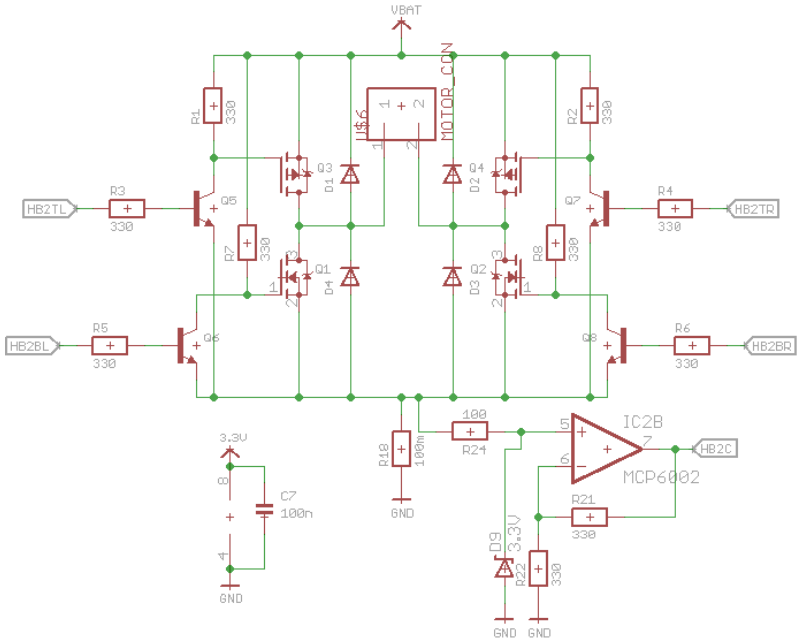


Figure 9.2: H-bridge 2 Schematics

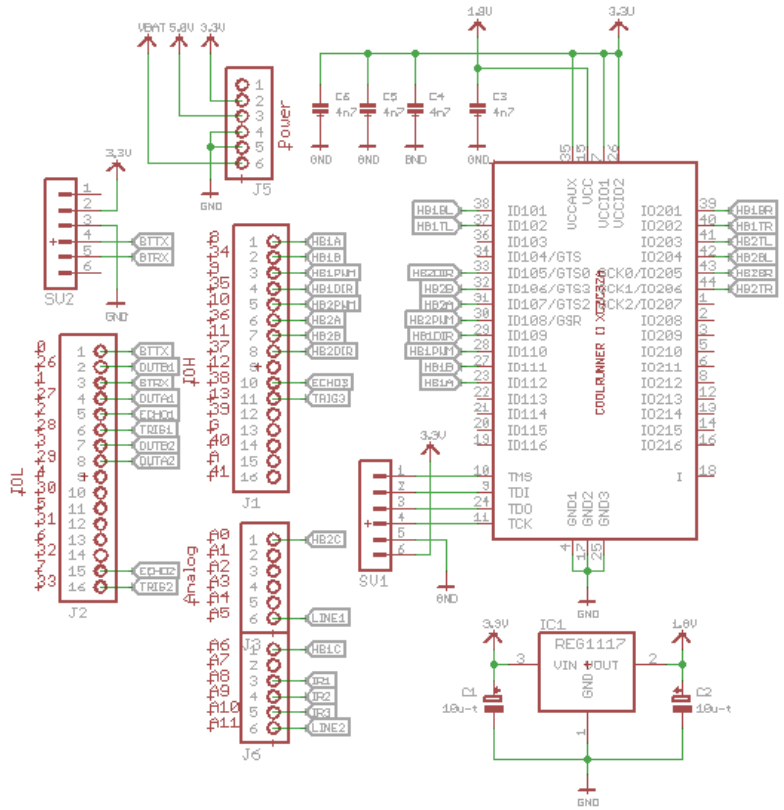


Figure 9.3: Schematics of the CPLD

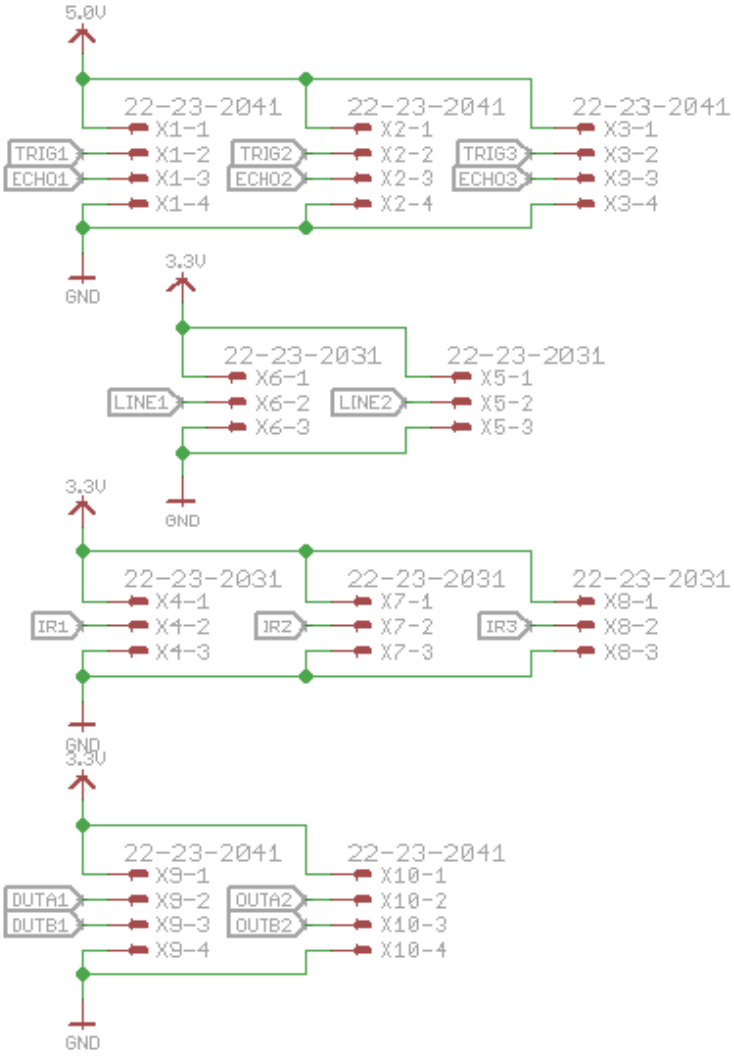


Figure 9.4: Power supply

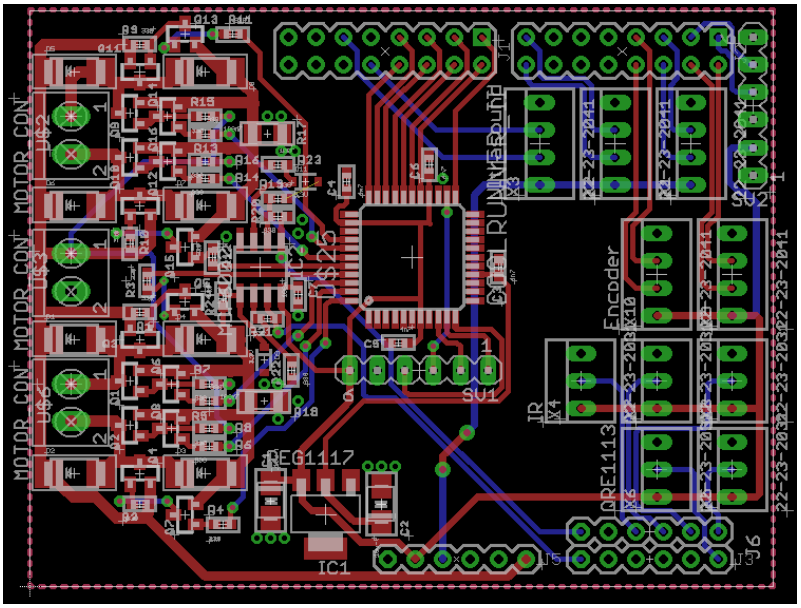


Figure 9.5: Full board Schematics

Software appendix 10

10.1 C code

main.c:

```
1  //**** GLOBAL INCLUDES ****//
2  #include <xc.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <stdint.h>
6  #define _SUPPRESS_PLIB_WARNING 1
7  #define _DISABLE_OPENADC10_CONFIGPORT_WARNING 1
8  #include <plib.h>
9  #include <math.h>
10
11 //**** PROJECT INLCUDES ****//
12 #include "setup.h"
13 #include "Functions.h"
14 #include "vehicle.h"
15 #include "UART.h"
16 #include "Delay.h"
17 #include "ADC.h"
18
19 //**** VARIABLE DEFINITIONS ****//
20 long tach1 = 0, tach2 = 0;
21 int goalX = 50;
22 int goalY = 200;
23 int printTimer = 0;
24 double angle = 0;
25 int currentX;
26 int currentY;
27 int currentTach = 0;
28 long Ultrasound[3];
29 int PWM[] = {25,25};
30 char str[10+1];
31 double rollingData[3][5] = {{500,500,500,500,500},
32                             {500,500,500,500,500},
33                             {500,500,500,500,500}  };
34 int rollingAverage[3];
35
36 //**** INTERRUPT SERVICE ROUTINE ****//
37 void __ISR(_EXTERNAL_1_VECTOR, IPL4SOFT) INT1Interrupt(){
38     asm volatile("di");
39     if(PORTDbits.RD10 == 0 && PORTDbits.RD5 == 1)//Right wheel
       backwards
40         tach1--;
```

```
41      else if(PORTDbits.RD10 == 1 && PORTDbits.RD5 == 0)//Right
            wheel forwards
42          tach1++;
43      IFS0bits.INT1IF = 0;
44      asm volatile("ei");
45  }
46
47  int main(int argc, char** argv) {
48      int i = 0;
49      int j = 0;
50
51      //**** INITIALIZATIONS ****//
52      UART1Init(115200, 1);
53      sensorIOInit();
54      startEngines();
55      initInterrupt();
56      initTimer();
57      initPWM();
58
59      UART1WriteLn("Starting");
60
61      adjustDuty(1,PWM[0]);
62      adjustDuty(2,PWM[1]);
63      initialTurn(goalX,goalY);
64      forwards();
65
66      for (;;){
67          rollingAverage();
68          controller(rollingAverage[0],rollingAverage[1],
                rollingAverage[2]);
69      }
70  }
```

functions.c

```
#include <xc.h>
#include "Delay.h"
#include <stdint.h>
#include <math.h>

#define Trigger1S TRISGbits.TRISG6
#define Echo1S TRISFbits.TRISF6
#define Trigger2S TRISEbits.TRISE7
#define Echo2S TRISEbits.TRISE2
#define Trigger3S TRISEbits.TRISE3
#define Echo3S TRISDbits.TRISD0

#define Trigger1 PORTGbits.RG6
#define Echo1 PORTFbits.RF6
#define Trigger2 PORTEbits.RE7
#define Echo2 PORTEbits.RE2
#define Trigger3 PORTEbits.RE3
#define Echo3 PORTDbits.RD0


double tachPerCm = 16.667;
int wheelbase = 9;
extern long tach1;
extern long tach2;
extern double angle;
extern int currentX;
extern int currentY;
extern int currentTach;
extern int rollingData;
extern int rollingAverage;

void initInterrupt(){
    INTEnableSystemMultiVectoredInt();
    IEC0bits.INT1IE = 0;
    INTCONbits.INT1EP = 0;
    IPC1bits.INT1IP = 4;
    IEC0bits.INT1IE = 1;
}

void initTimer(){
    T4CON = 0x0;              // Stop any 16/32-bit Timer4
       operation
    T5CON = 0x0;              // Stop any 16-bit Timer5 operation
    T4CONSET = 0x0038;        // Enable 32-bit mode, prescaler
       1:8, internal peripheral clock source
    TMR4 = 0x0;               // Clear contents of the TMR4 and
       TMR5
    PR4 = 0xFFFFFFFF;          // Load PR4 and PR5 registers with
        32-bit value
```

```
46      T4CONSET = 0x8000;
47  }
48
49  void dec_to_str(char* str, long val, size_t digits) {
50      size_t i = 1u;
51      for (; i <= digits; i++) {
52          str[digits - i] = (char) ((val % 10u) + '0');
53          val /= 10u;
54      }
55      str[i - 1u] = '\0'; // assuming you want null terminated
            strings?
56  }
57
58  long map(long x, long in_min, long in_max, long out_min, long
        out_max)
59  {
60    return (x - in_min) * (out_max - out_min) / (in_max - in_min) +
          out_min;
61  }
62
63  void initPWM() {
64      int sysClk = 80000000;
65      int pwmFreq = 1000;
66      int prescaleV = 1;
67      int dutyCycle = 0;
68
69      PMCONbits.ON = 0;
70      PMAEN = 0;
71
72      OC4CON = 0x0000;
73      OC4R = 0x00638000;
74      OC4RS = 0x00638000;
75      OC4CON = 0x0006;
76      OC4RS = (PR4 + 1)*((float) dutyCycle / 100);
77      OC4CONSET = 0x8020; //Enable peripheral, bit 5: 0=16 bit
            compare mode 1=32 bit
78
79      OC5CON = 0x0000;
80      OC5R = 0x00638000;
81      OC5RS = 0x00638000;
82      OC5CON = 0x0006;
83      OC5RS = (PR2 + 1)*((float) dutyCycle / 100);
84
85      T2CONSET = 0x0008;
86      T2CONSET = 0x8000;
87
88      OC5CONSET = 0x8020;
89
90      PR4 = (sysClk / (pwmFreq * 2) * prescaleV) - 1;
91      PR2 = (sysClk / (pwmFreq * 2) * prescaleV) - 1;
92  }
```

```
93
94 void sensorIOInit(){
95     Trigger1S = 0;
96     Echo1S = 1;
97     Trigger2S = 0;
98     Echo2S = 1;
99     Trigger3S = 0;
100    Echo3S = 1;
101 }
102
103 void adjustDuty(int channel, int duty) {
104     switch (channel) {
105         case 1:
106             OC5RS = (PR2 + 1)*((float) duty / 100);
107             break;
108         case 2:
109             OC4RS = (PR4 + 1)*((float) duty / 100);
110             break;
111     }
112 }
113
114 long micros(){
115     long result = ((TMR5<<16)+(TMR4<<0))/5; //Timer5: last 16
        bits in timer, bitshift to get a proper result
116     return result;
117 }
118
119 long millis(){
120     long result = ((TMR5<<16)+(TMR4<<0))/5;
121     return result/1000;
122 }
123
124 long readUltrasonic(int channel){
125     long timerFinish = 0;
126     long timerOld = 0;
127     int timeout = 3000;
128     switch(channel){
129         case 1:
130             Trigger1 = 1;
131             DelayUs(10);
132             Trigger1 = 0;
133             while(Echo1 == 0){}
134             timerOld = micros();
135             while(Echo1 == 1){
136                 timerFinish = micros();
137                 if(timerFinish - timerOld > timeout ||
                       timerFinish - timerOld < 0)
138                     return 500;
139             }
140             break;
141         case 2:
```

```
142             Trigger2 = 1;
143             DelayUs(10);
144             Trigger2 = 0;
145             while(Echo2 == 0){}
146             timerOld = micros();
147             while(Echo2 == 1){
148                 timerFinish = micros();
149                 if(timerFinish - timerOld > timeout ||
                       timerFinish - timerOld < 0)
150                     return 500;
151             }
152             break;
153         case 3:
154             Trigger3 = 1;
155             DelayUs(10);
156             Trigger3 = 0;
157             while(Echo3 == 0){}
158             timerOld = micros();
159             while(Echo3 == 1){
160                 timerFinish = micros();
161                 if(timerFinish - timerOld > timeout ||
                       timerFinish - timerOld < 0)
162                     return 500;
163             }
164             break;
165         default:
166             UART1Write("Default state");
167             break;
168     }
169     timerFinish = micros();
170     long result = ((timerFinish-timerOld)*0.34)/2;
171     return result;
172 }
173
174 double angleToTach(int angle)
175 {
176     return angle*3.14159/180*wheelbase/2*tachPerCm;
177 }
178
179 void initialTurn(int x, int y){
180     tach1 = 0;
181     char str[10+1];
182     double angleRadians = atan(y/x);
183     int angleD = angleRadians*180/3.14159; //Convert to degrees
184     double tachToTurn = angleToTach(angleD);
185     turnLeft();
186     while(tach1 < tachToTurn){}
187     brake();
188     angle = angleRadians;
189 }
190
```

```
191 void presetTurnLeft()
192 {
193     int tachTarget = tach1+angleToTach(30); //tachCount right //
          Calculate how far the wheel must turn to get 30 degrees
          using the arc length of a circular sector
194     DelayMs(50);
195     turnLeft();
196     while(tachTarget > tach1){}
197     forwards();
198     DelayMs(50);
199 }
200
201 void presetTurnRight()
202 {
203     int tachTarget = tach1-angleToTach(30); //tachCount right //
          Calculate how far the wheel must turn to get 30 degrees
          using the arc length of a circular sector
204     while(tachTarget > tach1)
205         turnRight();
206 }
207
208 void calculatePosition()
209 {
210     int distanceRun = tach1-currentTach; //How far has it run
          since last calculation
211     currentX = currentX+cos(angle)*distanceRun; //Calculate
          position on coordinate
212     currentY = currentY+sin(angle)*distanceRun;
213     currentTach = tach1; //Reset 0-position
214 }
215
216 void rollingAverage(){
217     int i;
218     int j;
219     for(i=0;i<3;i++){ //Run through once per sensor
220         double tempData = readUltrasonic(i+1); //Save the data on
              current sensor
221         if(tempData < 10000 && tempData > 0){ //To avoid huge and
              negative numbers
222             rollingData[i][4] = tempData; //Save data on the last
                  spot in the array
223             rollingAverage[i]=(rollingData[i][0]+rollingData[i
                  ][1]+rollingData[i][2]+rollingData[i][3]+
                  rollingData[i][4])/5; //Calculate average from
                  last 5 readings
224         }
225         for(j=0;j<4;j++){
226             rollingData[i][j]=rollingData[i][j+1]; //Move all
                  data one spot left in the arrays
227         }
228     }
```

```
}

void controller(int midSensor, int rightSensor, int leftSensor){
    int midDistance = 100;
    int sideDistance = 50;
    if(midSensor < midDistance)
    {
        brake();
        calculatePosition();
        backwards();
        int tachTarget = tach1-300;
        while(tach1 > tachTarget){}
        brake();
        presetTurnLeft();
        brake();
        forwards();
    }
    else if(leftSensor < sideDistance)
    {
        int tachTarget = tach1-20;
        turnRight();
        while(tachTarget > tach1){}
        brake();

    }
    else if(rightSensor < sideDistance)
    {
        int tachTarget = tach1+20;
        turnLeft();
        while(tachTarget < tach1){}
        brake();
    }
    else
        forwards();
}
```

vehicle.c

```
#include <xc.h>
#include "Delay.h"
#include <stdint.h>

//HB1A
#define HB1ASetup TRISDbits.TRISD10
#define HB1A PORTDbits.RD10
//HB1B
#define HB1BSetup TRISDbits.TRISD5
#define HB1B PORTDbits.RD5
//H1PWM
#define HB1PWMSetup TRISDbits.TRISD3
#define HB1PWM PORTDbits.RD3
//HB1DIR
```

```
15 #define HB1EnableSetup TRISDbits.TRISD11
16 #define HB1Enable PORTDbits.RD11
17
18 //HB2A
19 #define HB2ASetup TRISDbits.TRISD6
20 #define HB2A PORTDbits.RD6
21 //HB2B
22 #define HB2BSetup TRISGbits.TRISG8
23 #define HB2B PORTGbits.RG8
24 //HB2PWM
25 #define HB2PWMSetup TRISDbits.TRISD4
26 #define HB2PWM PORTDbits.RD4
27 //HB2DIR
28 #define HB2EnableSetup TRISDbits.TRISD7
29 #define HB2Enable PORTDbits.RD7
30
31 void startEngines()
32 {
33     HB1ASetup = 0;
34     HB1BSetup = 0;
35     HB1EnableSetup = 0;
36
37     HB2ASetup = 0;
38     HB2BSetup = 0;
39     HB2EnableSetup = 0;
40
41     HB1Enable= 1;
42     HB1A = 0;
43     HB1B = 0;
44
45     HB2Enable= 1;
46     HB2A = 0;
47     HB2B = 0;
48 }
49 void forwards()
50 {
51     HB1Enable = 0;
52     HB2Enable = 0;
53
54     HB1A = 1;
55     HB1B = 0;
56     HB2A = 1;
57     HB2B = 0;
58
59     HB1Enable = 1;
60     HB2Enable = 1;
61 }
62 void backwards()
63 {
64     HB1Enable = 0;
65     HB2Enable = 0;
```

```
66
67      HB1A = 0;
68      HB1B = 1;
69      HB2A = 0;
70      HB2B = 1;
71
72      HB1Enable = 1;
73      HB2Enable = 1;
74  }
75  void brake()
76  {
77      HB1Enable = 0;
78      HB2Enable = 0;
79
80      HB1A = 1;
81      HB1B = 1;
82      HB2A = 1;
83      HB2B = 1;
84
85      HB1Enable = 1;
86      HB2Enable = 1;
87  }
88  void turnLeft()
89  {
90      HB1Enable = 0;
91      HB2Enable = 0;
92
93      HB1A = 1;
94      HB1B = 0;
95      HB2A = 0;
96      HB2B = 1;
97
98      HB1Enable = 1;
99      HB2Enable = 1;
100 }
101 void turnRight()
102 {
103     HB1Enable = 0;
104     HB2Enable = 0;
105
106     HB1A = 0;
107     HB1B = 1;
108     HB2A = 1;
109     HB2B = 0;
110
111     HB1Enable = 1;
112     HB2Enable = 1;
113 }
```