Fall Semester 2015

Line following robot

Group 2

2. Semester IT-Technology

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

Supervisor: Jesper Kristensen - Steffen Vutborg

Uddannelse i virkeligheden

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

Title:

Line following robot

Project Period:

2. Semester | Spring semester 2016

Projectgroup:

Group 2

Medvirkende:

Benjamin Nielsen
Henrik Jensen
Martin Nonboe
Nikolaj Bilgrau

Supervisor:

Jesper Kristensen
Steffen Vutborg

Pages: TBD

Appendices: TBD

Completed TBD

Introduction

This project was written by group 2, for the second semester on the IT-electronics education at university college Nordjylland, Sofiendalsvej 60. The project goal is to make a line following robot.

Benjamin Nielsen

Henrik Jensen

Martin Nonboe

Nikolaj Bilgrau

Table of Contents

Glossary

3D print 3-Dimensional printing

ADC Analog-digital conversion

GUI Graphical User Interface

IDE Integrated Development Environment

MCU Microcontroller Unit

PID Proportional-integral-derivative

PWM Pulse-width modulation

THT Through-hole-technology

Requirements specification 1

The following section will describe the specific requirements that have been decided to fulfil to the general requirements as shown in the project description.

- Project must include light sensors
- Implement motor control
- Should make use of the Pic32 MCU
- Software will be written in MPLAPX
- Autonomous operation
- The product must make use of feedback concept e.g. a PID algorithm
- A function & performance test is to be conducted

Hardware section 2

In the following section, the hardware parts and functions will be introduced and described. The following part consists of sensor selection, ADC, pic uno32 board & motor-shield.

2.1 Hardware diagram

The micro-controller is connected to the motor-shield and the motor-shield is then powering both motor 1 and motor 2. The sensor-array is sending the collected data to the micro-controller every 25 ms, the micro-controller then sends it further to the bluetooth unit. The bluetooth unit then writes the data to the terminal.

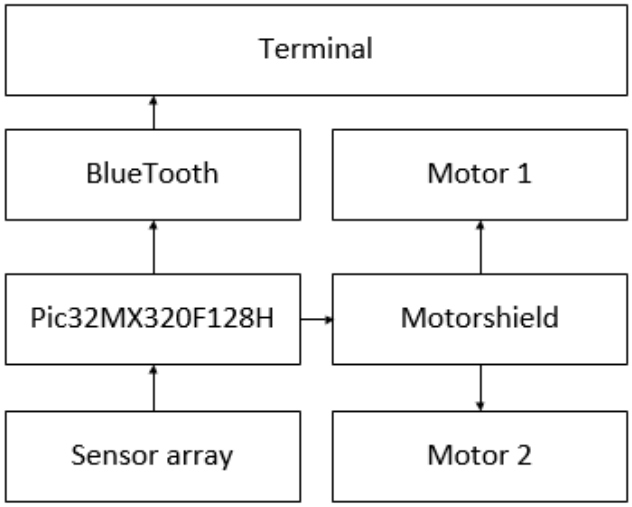


Figure 2.1: Block diagram showing the hardware.

2.2 Selection of sensor

The table shows a comparison of the two sensors that were taken into consideration for the project, that is to show the price difference and the utility from sensor to

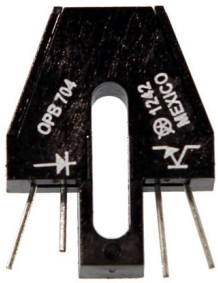
Name	QRE1113 Board	OPB704
Max sensor distance	3mm	3.8mm
Forward current	50mA	40mA
Mounting	On print	In casing
Price	19.43DKK	42.55DKK

Table 2.1: Table showing the sensors in consideration

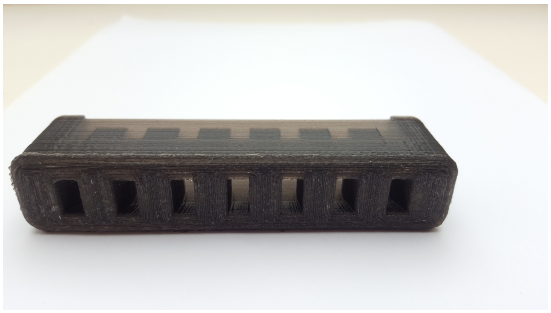
sensor. *(fig 2.1)*

2.2.1 The OPB704

The OPB704 sensor ended up being selected for the robot over the popular QRE1113 sensor board. It has a higher sensor max distance (3.8mm[1] compared to the QRE1113s 3mm) and it has the same functionality. It comes with a special casing, for which a special mounting unit was 3D-printed to accommodate an array of 7. This makes the mounting very solid and tightly fitted unto the robot, which ideally makes the sensor array more stable in case of an uneven test course.



(a) The sensor OPB 704

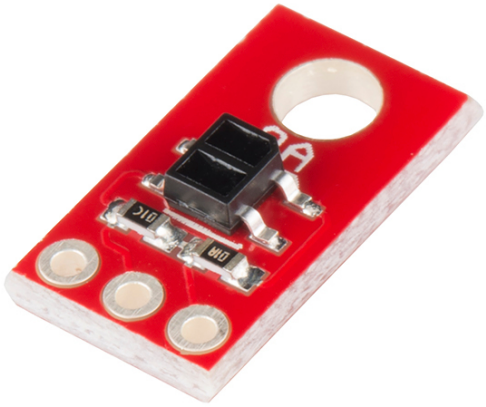


(b) 3D printed sensor array module

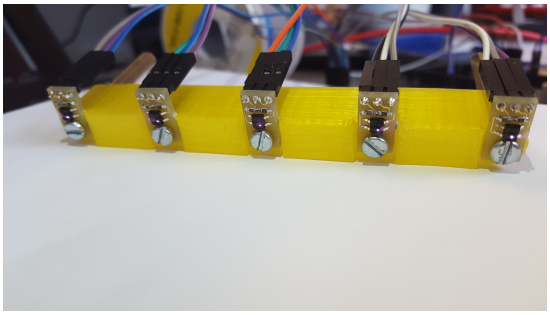
2.2.2 The QRE1113 board(Analog)

Another possible sensor selection would be the QRE1113[2] board sensor. This sensor comes complete with mounting and the necessary printing and wiring. This makes working with the sensor fairly straight forward. Although this sensor is commonly used for line following robots, when it was compared to the OPB704, it quickly became apparent that the difference in max sensor distance could become a liability, and with this in mind, the OPB704 sensor was selected.

[1] http://www.farnell.com/datasheets/1884910.pdf
[2] http://cdn.sparkfun.com/datasheets/Sensors/Proximity/QRE1113.pdf



(a) QRE1113 Sensor with own mount



(b) 3D printed sensor array module withe the QRE1113

2.3 Analog-to-digital converter (ADC)

The purpose of the ADC is to convert the analog data from the sensors to digital data that can be managed by a computer - this allows data received from the bluetooth transmitter on the product to be processed into readable data more easily, which is great for showing how the sensors are reacting. The sensors themselves cannot discern what they actually need to read, the sensors just read anything they can see and send that signal.
Analog signals can have a significant amount of noise - since any received noise is interpreted as part of the signal, a digital signal is not only more easy to work with, it will also provide more precise data. This will make for more accurate readings on the tachometer on the robot, which allows even more finely tuned monitoring of the robot and its working processes.

ADC diagram

This products usage of ADC

The usage of the ADC in the system, is to measure the returning voltage from the OPB704 sensor. Both the OPB704 and the QRE1113 sensors are reflective phototransistors. A reflective phototransistor consists of a infrared LED and a phototransistor. The phototransitors output voltage is controlled by the amount of light from the LED that is being reflected of the given surface the sensor is looking at, back to the phototransistor.

2.4 The ChipKit Uno32 board

Our main computing core is at the ChipKit Uno32 board. The board was selected both because of previous experiences, and because it met the set requirements perfectly - most importantly it has twelve analog imputs to handle our array of seven

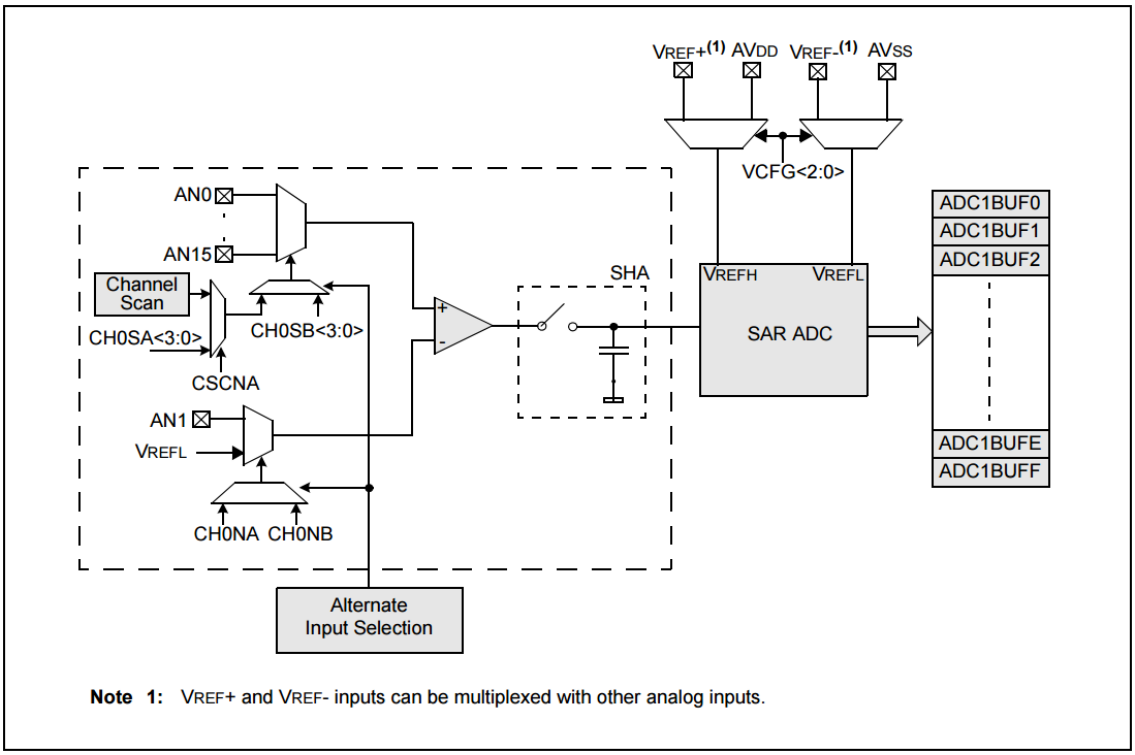


Figure 2.4: PIC32 ADC fuctional block diagram

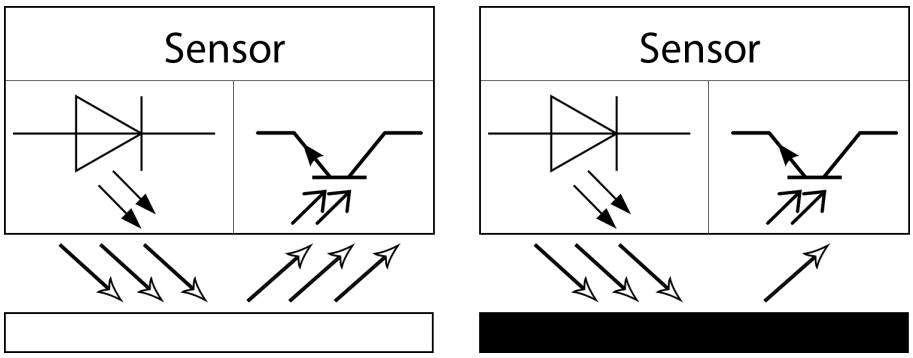


Figure 2.5: The difference in reflection on light and dark surfaces

sensors. This board utilizes the PIC32MX320F128 microcontroller, which features a 32-bit processing core running at 80Mhz with 128K of flash program memory and 16K SRAM data memory[3]. This board allowed the most important aspects of the project to become reality; namely a GUI showing both the ADC readings as well as visualizing the sensor imputs, plenty of analog inputs to allow for the sensor array, and BlueTooth transmission of data from the robot unto a computer.
The board is compatible for use with MPLAB IDE and the PICKit3 debugger.

[3]http://www.microchip.com/wwwproducts/en/PIC32MX320F128H

2.5 The motor shield - PKA03

The motor shield was chosen because it is compatible with the Uno32 board, and fits perfectly within the scope of the project. It controls both motors and receives power from the 7.2 V lithium-ion battery pack mounted on the chassis itself. The motor shield is instrumental in providing motor controls to the product. To do this, it utilizes a specific electric circuit, called an H bridge.

2.5.1 The H bridge

An H bridge is a circuit that allows a voltage to be applied across a load in either direction. The purpose of this in the case of this project is to enable controls of the two motors in a way so that they can function individually, and both drive forwards and backwards.
At first, there were experiments with custom prints and a purpose-built h bridge on the first iteration of the motor shield. However after testing, it proved faulty and was changed to accommodate time constraints.

2.6 The blue-tooth transmitter - BlueSMIRF Silver

The robot utilizes the BlueSMIRF Silver blue-tooth transmitter made by Sparkfun. Its function is to send data from the ADC and pulse width modulator (see the software section) to a C# GUI run on a computer. This way, it is possible to monitor both the inputs the robot is receiving, as well as the logic behind the steering. It allows for any stream between 2400 to 115200 bps[4].

[4]https://www.sparkfun.com/products/12577

Software section 3

3.0.1 Software diagram

TBD Softwarediagram

3.1 Description of the software structure and functionality

The following section will introduce the software segment, based on the required specifications. The design will shown in the form of a flowchart. The section consists of the PID controller & the Pulse width modulator. TBD Softwarebeskrivelse og underafsnit

3.2 Analog to digital conversion (software)

TDB når koden er pæn og der kan laves en snippet!

```
1
2 extern int analogRead(char CH) {
3     AD1PCFG = ~CH; // PORTB = Digital; RB2 = analog
4     AD1CON1 = 0x0000; // SAMP bit = 0 ends sampling
5     // and starts converting
6     //AD1CHS = 0x00020000; // Connect RB2/AN2 as CH0 input
7     // in this example RB2/AN2 is the input
8     AD1CSSL = 0;
9     AD1CON3 = 0x0001; // Manual Sample, TAD = internal 6 TPB
10    AD1CON2 = 0;
11    AD1CON1SET = 0x8000; // turn on the ADC
12    AD1CHSbits.CH0SA = CH;
13    AD1CON1SET = 0x0002; // start sampling ...
14    DelayUs(2); // for 2 us
15    AD1CON1CLR = 0x0002; // start Converting
16    while (!(AD1CON1 & 0x0001)); // conversion done?
17    return ADC1BUF0;
18 }
```

The software code is showing how the ADC is converting analog to digital for the product. TBD: Det er nok ikke nødvendigt at splitte det helt op som efterfølgende, det kan bare beskrives som en helhed.

```
1 AD1PCFG = ~CH; // PORTB = Digital; RB2 = analog
```

```
1 AD1CON1 = 0x0000; // SAMP bit = 0 ends sampling
2     // and starts converting
3     //AD1CHS = 0x00020000; // Connect RB2/AN2 as CH0 input
4     // in this example RB2/AN2 is the input
```

```
AD1CSSL = 0;
```

```
AD1CON3 = 0x0001; // Manual Sample, TAD = internal 6 TPB
```

```
AD1CON2 = 0;
```

```
AD1CON1SET = 0x8000; // turn on the ADC
```

```
DelayUs(2); // for 2 us
```

```
AD1CON1CLR = 0x0002; // start Converting
```

```
while (!(AD1CON1 & 0x0001)); // conversion done?
    return ADC1BUF0;
```

3.3 Description of the PID controller

A PID controller continuously calculates an error value as the difference to a reference point and a measured process variable.
PID is an abbreviation for a proportional-integral-derivative controller, it is a control loop feedback mechanism. The controllers job is to minimize the error value for the given devices running time. In the case of this project the reference point is the line to follow and the PID will allow the MCU to adjust the power to the engines, to steer accordingly to said reference point.

$$\mathrm{F}(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{de(t)}{dt}$$

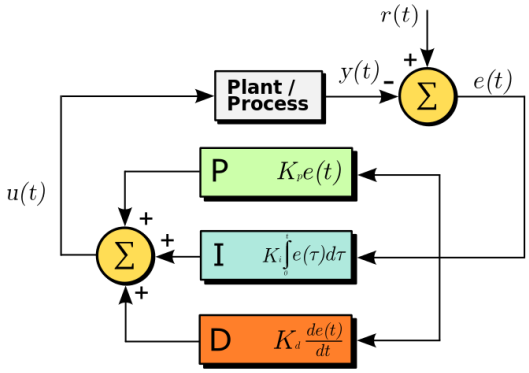


Figure 3.1: Block diagram, showing the idea of PID controller.

(a) Credits: `www.wikipedia.org/wiki/PID_controller`

3.3.1 Proportional control(P)

The proportional part creates an output value that is proportionally related to the current error value, this value can be tuned by multiplying the error by a constant K_p. A high proportional gain results in a large change in the output for a given change in the error.

$$P_{\text{out}} = K_p\, e(t)$$

If the proportional gain is too high, the system can become unstable. Contrarily, a small gain will result in the device adjusting too slowly, which decreases overall efficiency and in the case of this project, it will end up being detrimental to the steering accuracy.

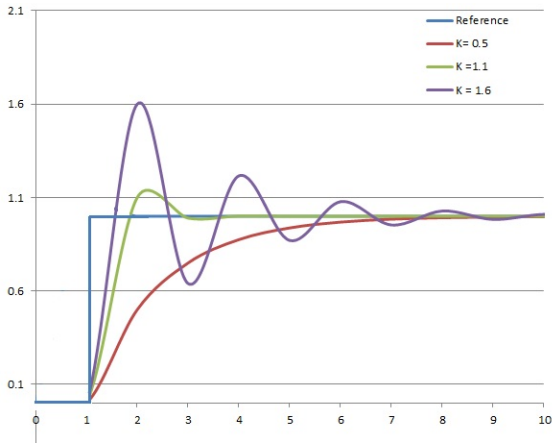


Figure 3.3: K_p with 3 values. (K_i, K_d held constant)

(a) Credits: `www.wikipedia.org/wiki/PID_controller#Proportional_term`

3.3.2 Steady-state error

When looking at figure 3.2 the reference point is the blue line and the feedback line is the green, the two lines merge up and the steady-state is achieved. When the lines don't merge, but the feedback line is slightly above or under the reference point. This steady-state error can be minimized by adjusting the proportional and integral term.

3.3.3 Integral control(I)

The integral controller is contributing proportionally to both the magnitude of the error and the duration of the error.
The integral in a PID controller is the sum of the instantaneous error over time and gives the accumulated offset that should have been corrected previously.

The controller output equals the accumulated error multiplied by the integral gain(Ki)

$$I_{\text{out}} = K_i \int_0^t e(\tau)\, d\tau$$

The integral part accelerates the movement of the process towards the reference point. Since the integral term correlates to accumulated errors from the past, it can cause the present value to overshoot the reference value.

3.3.4 Derivative control(D)

The derivative of the process error is calculated by determining the slope of the error over time and multiplying this rate of change by the derivative gain K_d. The magnitude of the contribution of the derivative term to the overall control action is termed the derivative gain, K_d.

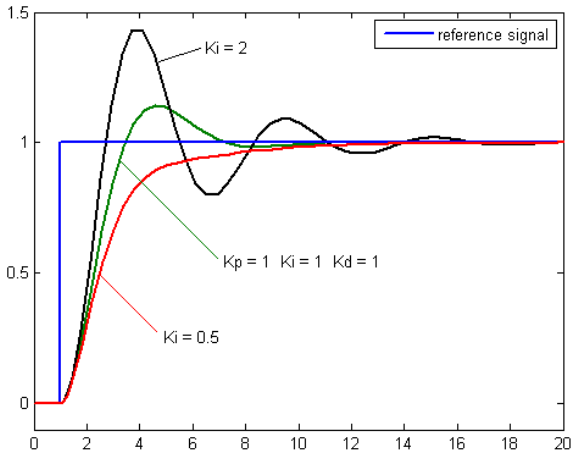


Figure 3.5: K_i shown with 3 values.

(a) Credits: `www.wikipedia.org/wiki/PID_controller#Integral_term`

The derivative term is given by:

$$D_{\text{out}} = K_d \frac{de(t)}{dt}$$

The derivative action predicts system behaviour and utilizes this to improve the settling time and stability of the system. An ideal derivative is not causal, so that implementations of PID controllers include an additional low pass filtering for the derivative term, to limit the high frequency gain and noise.

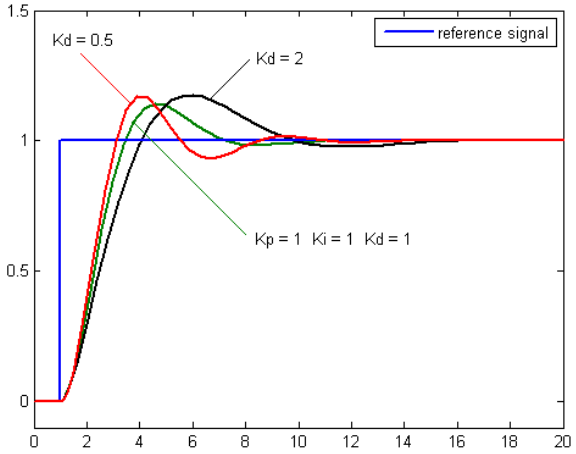


Figure 3.7: K_d shown with 3 values.

(a) Credits: `www.wikipedia.org/wiki/PID_controller#Derivative_term`

3.3.5 Loop tuning

Tuning the loop is the term used to describe the adjustments of the PID's control parameters (proportional band/gain, integral gain/reset, derivative gain/rate) to the optimal values for the given control scheme.
Stability is the first requirement; however systems can differ greatly, and different applications may have different requirements and these may even conflict with each other. For example, high speed and high accuracy often cancel each other out, because high speed may cause overshooting, while high accuracy is slow.
The ideal realistic behaviour is both as fast as possible, while also having minimum overshoot and oscillation.

Even though the process seems simple, with only three variables, it can be challenging to achieve, because it must satisfy the criteria despite being within the limitations of PID control. While adjusting the PID can seem conceptually intuitive, and while most PIDs may perform acceptably with default controls, they may very well also have an unsatisfactory performance.
This can generally be fixed through optimisation and tuning, either through computer simulations or manual testing. In this case, there was used manual tuning of the numbers.

3.3.6 Stability

If the parameters of the PID controller are set incorrectly the process input can become unstable. This means the controllers output becomes divergent, which can be limited by saturation and mechanical breaking.

3.3.7 Manual tuning

When a system must be online at all times a method for tuning is to first set K_i and K_d values to zero. Increase K_p until the loop output oscillates, setting K_p at approximately half the value for a "quarter amplitude decay" type response.
Then increase K_i until any set off is corrected in sufficient time for the process. Adding too much K_i will however cause an instability. Finally, increase K_d, if required at all, until the loop is acceptably quick to reach its reference after a load disturbance.
A fast PID loop tuning process usually overshoots slightly to reach the reference point faster.
But in the case of systems that can't accept overshoot, an over-damped closed-loop system is best suited, which requires K_p setting significantly less than half that of the K_p setting that was causing the oscillation.

Table 3.1: Manual tuning

Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improves if K_d is small

Table 3.1 explained

Table 3.1 gives an informative overview of what the different parameters does when tuned manually[1].

- To minimize the rise time, decrease K_p
- To eliminate the steady-state error, increase K_i
- To reduce the overshoot and settling time, decrease K_d

3.3.8 PID Implementation

TBD har ændret sensorAvg til sensorMean, gør dette i koden også

```
1  void PID(int sensorMean, int sensorSum)
2  {
3    if(sensorSum > 0) //At least one sensor sees black
4    {
5      sensorPos = sensorMean/sensorSum; //Position of the line on
           the sensorarray
6      sensorProp = sensorPos - 3;  //Proportional part
7      sensorInt = sensorInt + sensorProp; //Integral part
8      if(sensorInt > 100) //Reduce adjustment time by limiting Int
9        sensorInt = 100;
10     if(sensorInt < -100)
11       sensorInt = -100;
12     sensorDer = sensorProp - sensorLastProp; //Derivative part
```

The PID function has two parameters, sensorMean and sensorSum. These are used to calculate where the line is in relation to the sensors. In the beginning, a check is made to make sure the line actually is on the array somewhere, since the program would stall by trying to divide by sensorSum being 0. If at least one sensor detects the line, the position is calculated by dividing the mean value of the active sensors by the amount of sensors active. Since sensorMean is calculated using weighs on each sensor, this will give the average position. The function then calculates the different parts of the PID controller, firstly the proportional part. This is done by simply subtracting the weighted value of the middle sensor from the position calculated earlier. Next up is the integral part, which is calculated by adding the previous value of the integral to the proportional value. Right after, the integral is limited since the robot will be working fast on a small line. Derivative is the last

[1] http://saba.kntu.ac.ir/eecd/pcl/download/PIDtutorial.pdf

part calculated, which is done by subtracting the previous value of the proportional part from the current value.

```
1     sensorError = (sensorProp*kp)+(sensorInt*ki)+(sensorDer*kd);
          //PID calculation
2     sensorLastProp = sensorProp; //Saves proportional for next
          derivative
3     if(sensorError < -(initialPwm)) //Sets an upper cap for
          adjustment
4       sensorError = -(initialPwm);
5     if(sensorError > (initialPwm))
6       sensorError = (initialPwm);
```

This part of the code handles the total PID value as well as limiting this value. First of all, it multiplies the previous PID values by their constants and adding these numbers together and then saves the proportional value to be handled by the derivative part next time. At the end, sensorError is limited to the maximum speed of the motors, to make future calculations easier.

```
1     if(sensorError<0)
2     {
3       adjustedPWM[0] = initialPwm-sensorError; //Increase left
            motor, sensorError is negative here
4       adjustedPWM[1] = initialPwm+sensorError; //Decrease right
            motor
5       dir = 1; //turn right
6     }
7     else if(sensorError>0)
8     {
9       adjustedPWM[0] = initialPwm-sensorError;
10      adjustedPWM[1] = initialPwm+sensorError;
11      dir = 0; //turn left
12    }
13    else
14    {
15      adjustedPWM[0] = initialPwm;
16      adjustedPWM[1] = initialPwm;
17      dir = 2;
18    }
19  }
20 }
```

The final part of the PID function handles the duty cycle sent to the motors. If sensorError is less than 0, the robot needs to turn right, increasing engine power on the left while decreasing engine power on the right. If sensorError is positive, do the opposite. If sensorError is exactly 0, the duty cycle will remain what it was last iteration, which is a very rare case.

3.4 Description of the Pulse width modulator - PWM

A pulse width modulation technique used to encode a message into a pulsing signal. Its primary use is to control the power supply of electronic devices - the case of this project; this means the motors. The average voltage and amplitude output to the motors is altered by rapidly switching between an 'on' and 'off' state. This way, the average output is easily altered by changing the proportions between the on and off state. The longer the switch is set to on compared to off, the higher the average supply will be. Pulse-width modulation utilizes a rectangular pulse where the width of the pulse is modulated to get the variation in the average value of the waveform. Given the pulse waveform $f(t)$ over the period T, low value $y_{\min}$ and high value y_{high} and duty cycle D, the average waveform is given by:

$$\overline{y} = \frac{1}{T}\int_0^T f(t)dt$$

This expression can be simplified where $y_{\min} = 0$ as $\overline{y} = D \cdot y_{\max}$. From this it can be observed that the average value of the signal ($\overline{y}$) has a direct correlation with the duty cycle D.[2].

$$\begin{aligned}\overline{y} &= \frac{1}{T}\left(\int_0^{DT} y_{\max}dt + \int_{DT}^{T} y_{\min}dt\right) \\ &= \frac{1}{T}\left(D \cdot T \cdot y_{\max} + T(1-D)y_{\min}\right) \\ &= D \cdot y_{\max} + (1-D)\, y_{\min}\end{aligned}$$

This expression can be simplified where $y_{\min} = 0$ as $\overline{y} = D \cdot y_{\max}$. From this it can be observed that the average value of the signal ($\overline{y}$) has a direct correlation with the duty cycle D.

3.4.1 Duty cycles

The duty cycle describes the proportion of 'on' compared to any given period of runtime for the device. The duty cycle is described as a percentage, where 100% means that it's turned on the entire time, where 10% would be a tenth of the time. TBD Vores duty cycle

3.5 The GUI

The project utilizes a GUI written in C# , it shows a graphical representation of the sensor load, as well as which motor is running in accordance to the PID. The GUI has controls to set the COM-port as well as the baud rate, and a button prompt

[2] `https://en.wikipedia.org/wiki/Pulse-width_modulation#Principle`

to connect and disconnect the robots blue-tooth transmission. It also features a decision history on the right (TBD - er det der stadig?) and a terminal in the bottom, showing all the data sent from the ADC to the robot.

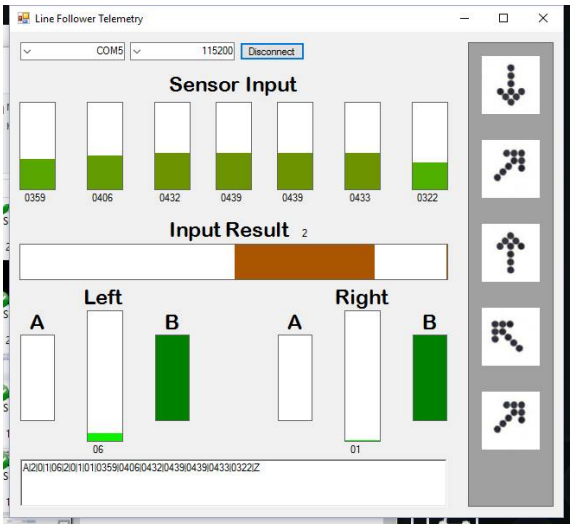


Figure 3.9: Screenshot of the C# GUI used to monitor the sensor readings.

Test 4

4.1 Testing round 1

4.1.1 Test description

To make sure the product as a whole is working according to plan, testing must be done. First the components must be tested, this is done individually for each component to make sure there are no errors or fails when the product is built. The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together. Testing is also done to watch the behavior of the product and tweak it.

4.1.2 Board Testing

TBD virker boardet?

4.1.3 Unit Testing

TBD testing af individuelle dele

4.1.4 Integration Testing

TBD testing af interfacet og evt samarbejde mellem enkelte dele

4.1.5 System Testing

TBD testing af systemet som helhed

4.1.6 Acceptance Testing

TBD testing af systemet som helhed. Acceptabel ydeevne?

Conclusion 5

TBD Konklusion

Appendices 6

6.1 Group collaboration agreement

6.1.1 Contact Information

Table 6.1: Contacts

Benjamin Nielsen	Tlf: 30427645	@: yipiyuk5@gmail.com
Henrik Jensen	Tlf: 28568934	@: henrik_kort@hotmail.com
Martin Nonboe	Tlf: 23827566	@: nonsens_4@hotmail.com
Nikolaj Bilgrau	Tlf: 29802715	@: nikolajbilgrau@gmail.com

6.1.2 Workflow

- Every friday after 12:00 is expected work consisting of three hours.
- If you aren't able of attending for scheduled study day. - Notice must be given to the project team.

6.1.3 Deadline

- Hand in June 7th.

6.1.4 Milestones and goals

In may there is listed a workshop from the 17-23 May. This was postponed and used for project days instead.

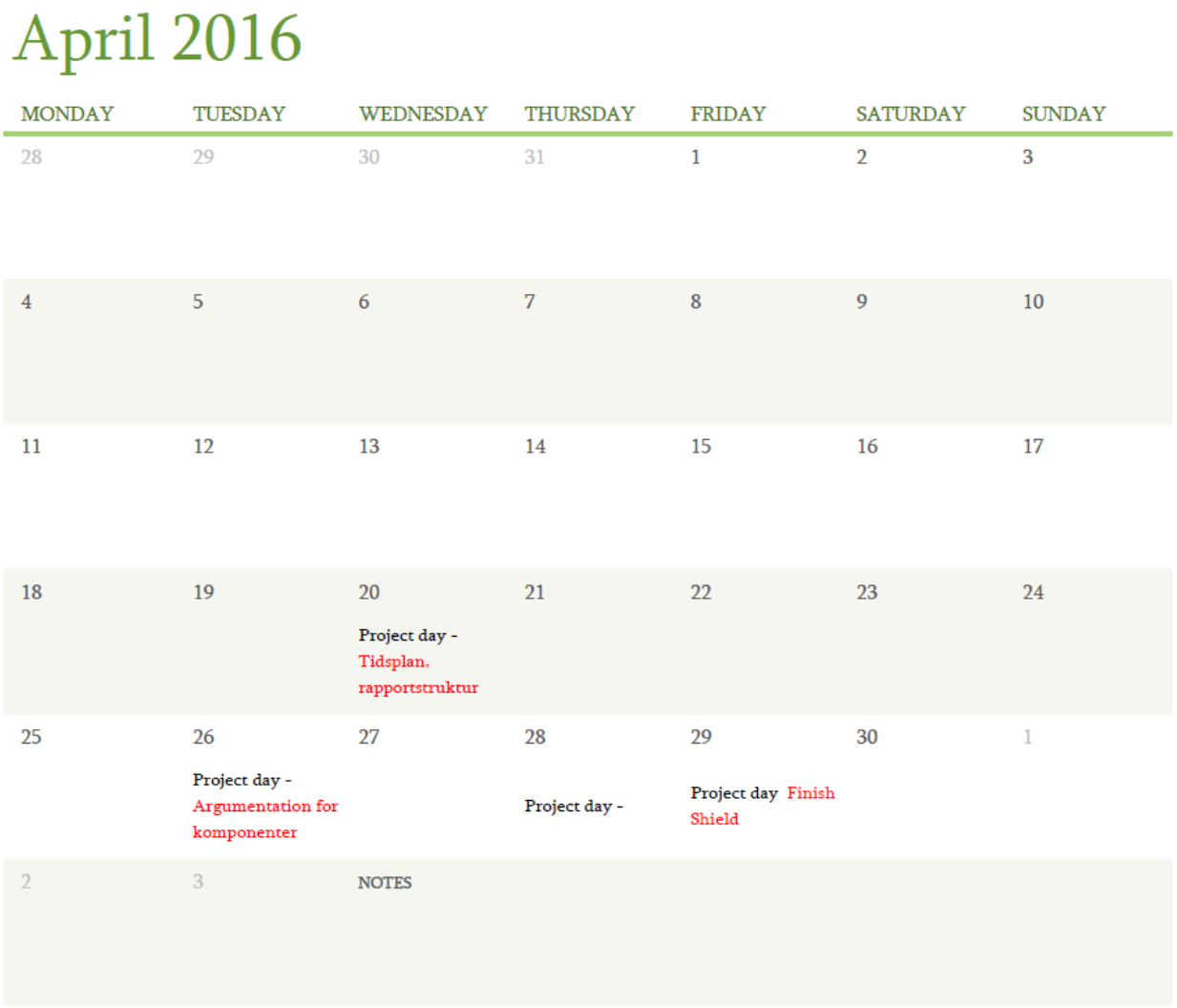
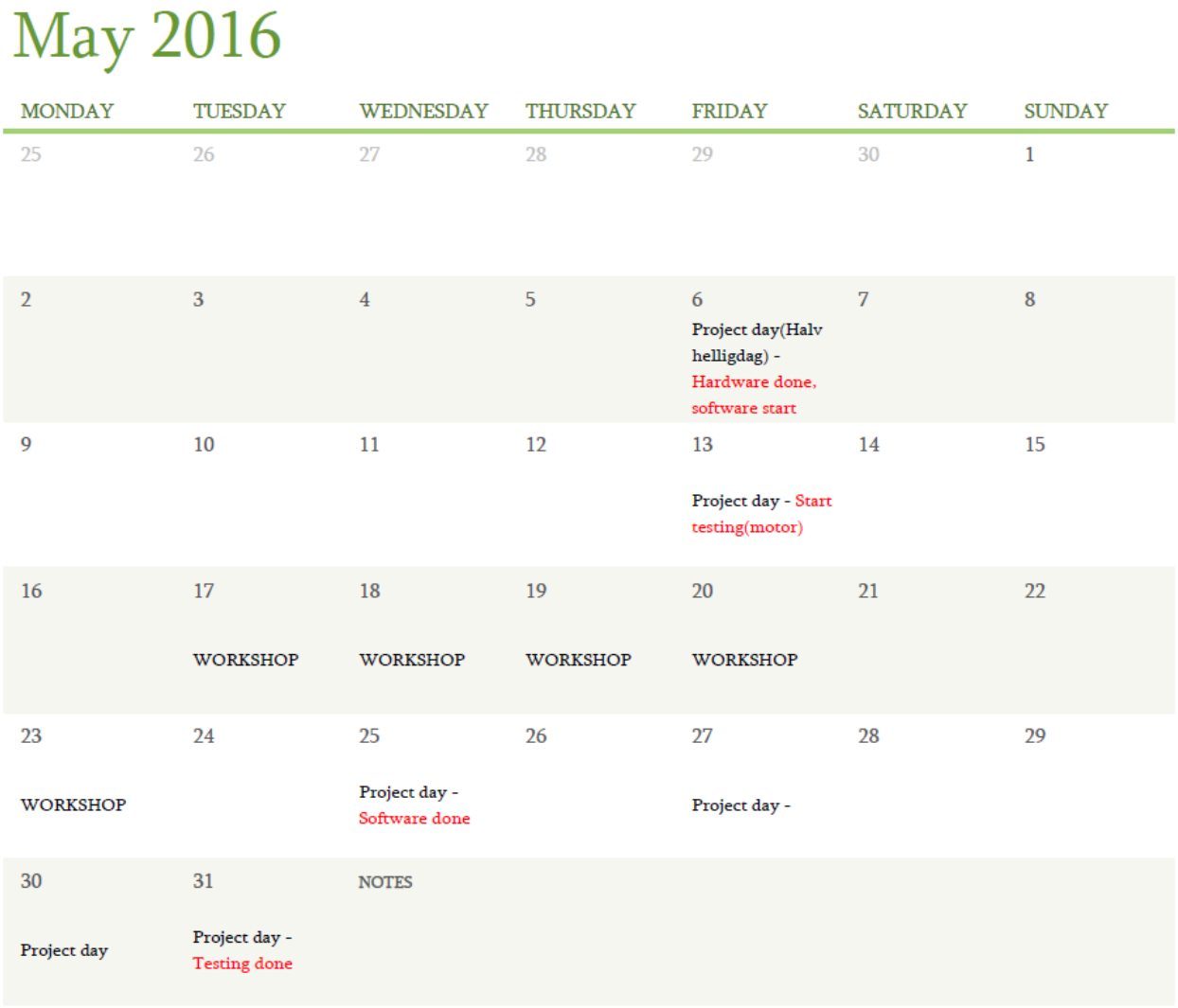


Figure 6.1: 4 Work days in April



May 2016

MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY	SUNDAY
25	26	27	28	29	30	1
2	3	4	5	6 Project day(Halv helligdag) - Hardware done, software start	7	8
9	10	11	12	13 Project day - Start testing(motor)	14	15
16	17 WORKSHOP	18 WORKSHOP	19 WORKSHOP	20 WORKSHOP	21	22
23 WORKSHOP	24	25 Project day - Software done	26	27 Project day -	28	29
30 Project day	31 Project day - Testing done	NOTES				

Figure 6.2: 6 Work days in May

June 2016

MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY	SUNDAY
30	31	1 Project day - Rapport	2 Project day - Rapport	3 Project day - Rapport	4	5
6 Project day - Rapport	7 !!!Aflevering!!! K1 12.00!	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3
4	5	NOTES				

Figure 6.3: 4 Work days in June

List of references 7

TBD list of references

PID:
https://en.wikipedia.org/wiki/PID_controller#/media0/File:PID_en_updated_feedback.svg

https://en.wikipedia.org/wiki/PID_controller#Proportional_term

http://saba.kntu.ac.ir/eecd/pcl/download/PIDtutorial.pdf

http://blog.opticontrols.com/archives/1066

https://en.wikipedia.org/wiki/PID_controller#Steady-state_error

https://en.wikipedia.org/wiki/PID_controller#Integral_term

https://en.wikipedia.org/wiki/PID_controller#Derivative_term

https://en.wikipedia.org/wiki/PID_controller#Manual_tuning

https://en.wikipedia.org/wiki/PID_controller#Control_loop_basics

http://blog.opticontrols.com/archives/1066
PWM:

https://en.wikipedia.org/wiki/Pulse-width_modulation

List of Figures

List of Tables

Page