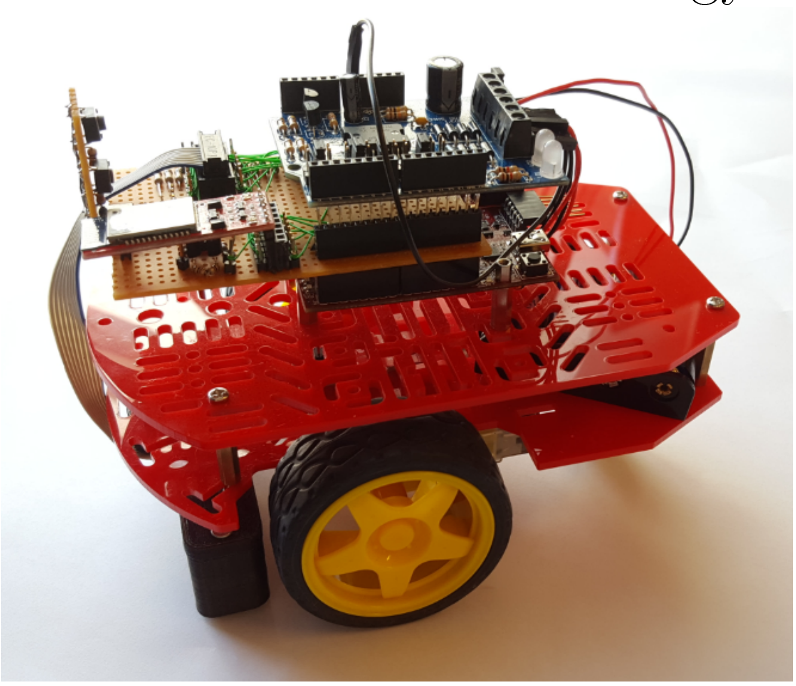
Fall Semester 2015

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

Group participants:
Benjamin Nielsen
Henrik Jensen
Martin Nonboe
Nikolaj Bilgrau

Supervisors:
Jesper Kristensen
Steffen Vutborg

Pages: 36

Appendices: 22

Completed June 6th 2016

Preamble

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

UART Universal Asynchronous Receiver/Transmitter

PCB Printed Circuit Board

1 Introduction

A line following robot is essentially a robot designed for the consumer to follow a line or path that is not predetermined. This line or path may be as simple as a strip of tape or a black line and, if developed further, can follow e.g magnetic markers, embedded lines or laser guided markers. In order to detect the various lines or paths, miscellaneous sensors or sensing methods can be used.
These methods may range from simple low cost sensors to advanced and more expensive vision systems, for example cameras. In the industry the many different types of robots are already implemented in semi to fully automatic systems.

The project was handed to the group April 12th and will be handed in at UCN Sofiendalsvej, June 7th at 12.00.
The objective of this project is to design and implement an automotive robot capable of autonomous maneuvering, specifically a line-following robot employing light detecting sensors.
The challenges at hand are to design a system for the board, to utilize the ADC capabilities of the chip and to implement a PID controller; furthermore, test the products performance on a test track to optimize the control algorithms by adjusting values and to implement a hardware solution featuring light detecting sensors.

2 Requirements specification

The following section will list 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 or the UCN board
- Software will be written in MPLAPX
- Must follow a line autonomously
- The product must make use of feedback concept e.g. a PID algorithm
- A function & performance test is to be conducted

Hardware section 3

In the following section, the hardware parts and functions will be introduced and described. The following part consists of sensor selection, ADC, chipKIT Uno32 board, motor-shield including the H bridge, and the bluetooth transmitter.

3.1 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 micro-controller reads data from the sensor array every 25 millisecond by utilizing the ADC, the micro-controller then sends it further to the bluetooth unit. The bluetooth unit then sends its data to the terminal.

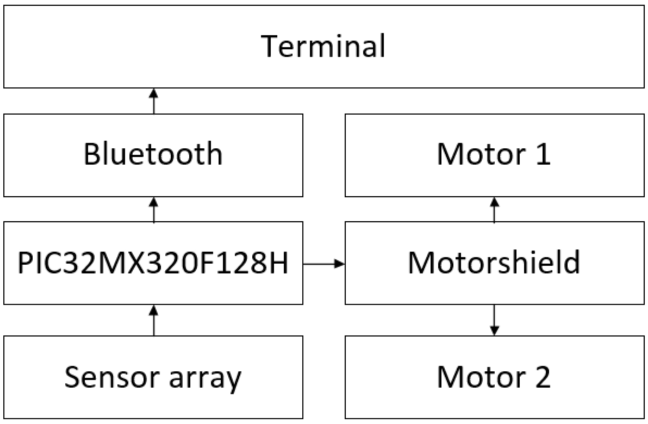


Figure 3.1: Block diagram of the hardware.

3.2 Selection of sensor

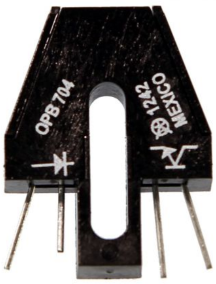
The table shows a comparison of the two sensors that were taken into consideration for the project, this is to show the price difference and the specification differences from sensor to sensor. *(fig 2.1)*

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

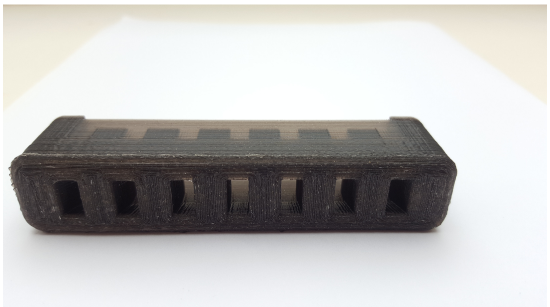
Table 3.1: Table showing the sensors in consideration

3.2.1 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.



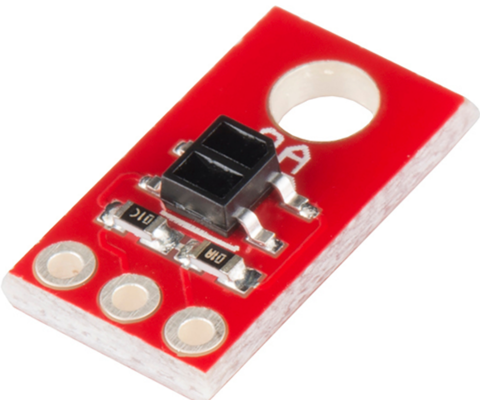
(a) The sensor OPB 704



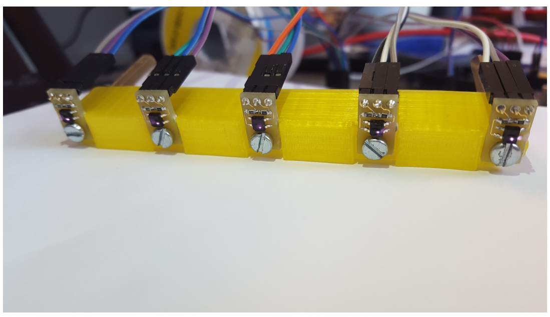
(b) 3D printed sensor array module

3.2.2 The QRE1113 board

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 straightforward. Although this sensor is commonly used for line following robots, for this project the OPB704 sensor was chosen. This was done due to the accessibility and ease of mounting of the OPB 704 sensor.



(a) QRE1113 Sensor on a breakout board



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

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

3.3 Analog-to-digital converter

The purpose of the ADC is to convert the analog signal from the sensors, to digital data that can be managed by a computer. The sensors themselves cannot discern what signals are relevant and when to send these, 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.

ADC diagram

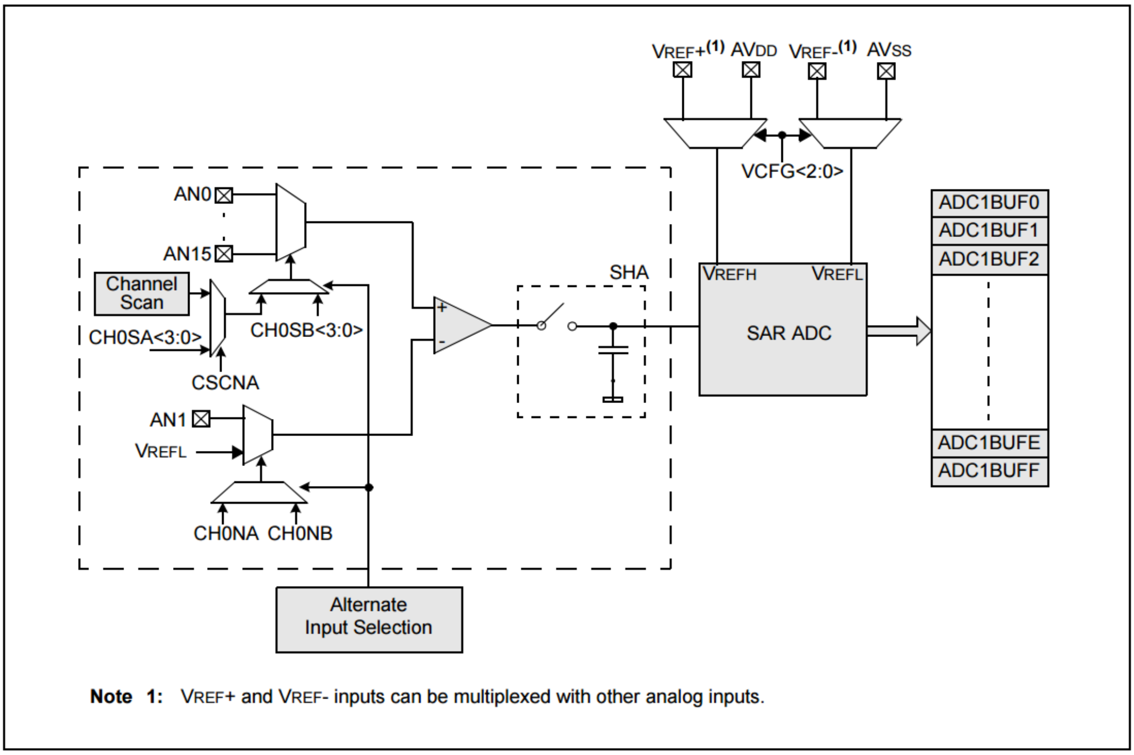


Figure 3.4: PIC32 ADC fuctional block diagram

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 off the given surface the sensor is pointed at, back to the phototransistor.

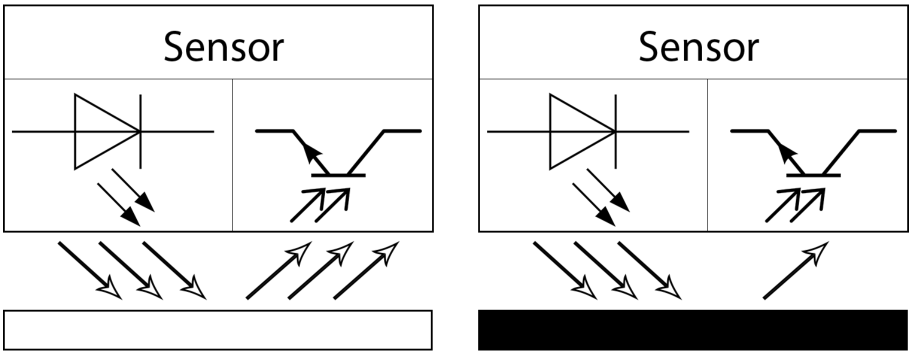


Figure 3.5: The difference in reflection on light and dark surfaces

3.4 The chipKIT Uno32 board

The main computing core is the chipKIT Uno32 board. The board was selected based on on previous experiences, and because it meets the set requirements - most importantly it has twelve analog imputs to handle our array of seven sensors; however the UCN board can only manage 6 sensors. This board utilizes the PIC32MX320F128 microcontroller, which features a 32-bit processing core running at 80MHz with 128KB of flash program memory and 16KB SRAM data memory[3]. This board has more than enough analog inputs which allows upwards of 12 sensors. Furthermore it allows for easier ADC conversions. The board is compatible for use with MPLAB X IDE and the PICKit3 debugger.

3.5 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 6.0 V 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.

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

3.5.1 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 they can both function individually, and both drive forwards and backwards.

3.6 The Bluetooth tranceiver

The robot utilizes the BlueSMiRF Silver Bluetooth transceiver made by Sparkfun. Its function is to send data from the MCU (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 baudrate between 2400 to 115200[4].

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

Software section 4

The following section will introduce the software segment, based on the required specifications. The design will be shown in the form of a flowchart. The section consists of the PID controller and the Pulse width modulator as well as showing the ADC initialization.

4.0.1 Software diagram

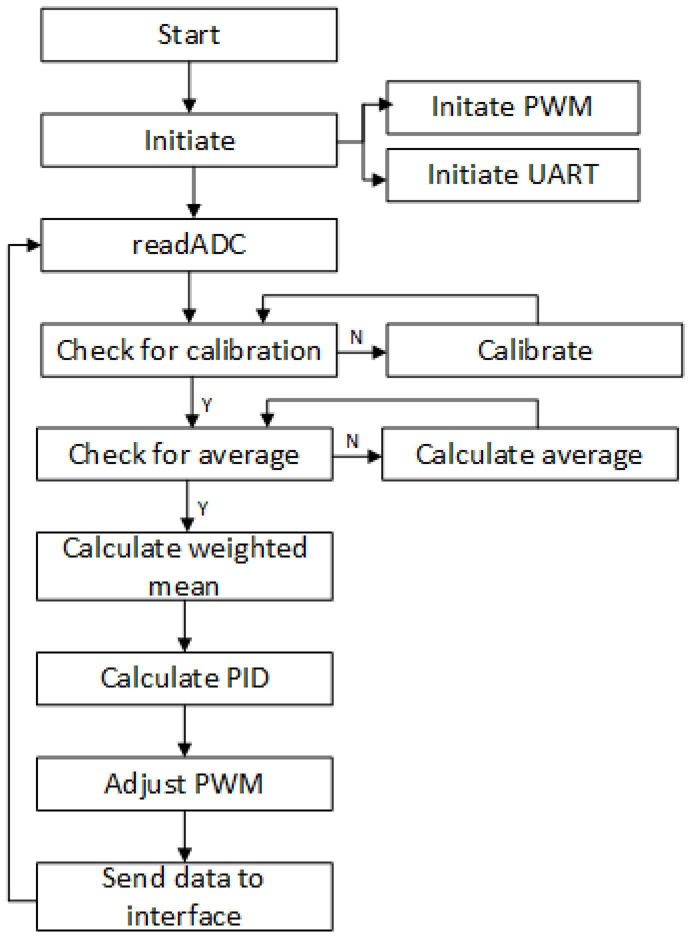


Figure 4.1: Software flowchart

4.1 Analog to digital conversion

```
extern int analogRead(char CH) {
    AD1PCFG = ~CH;
    AD1CON1 = 0x0000; //AMP bit = 0 ends sampling and starts converting
    AD1CSSL = 0; //Input scan select, not used
    AD1CON3 = 0x0001; //Manual Sample, TAD = internal 4 TPB
    AD1CON2 = 0; //No scanning, interrupt at completion of each sample, one 16-word buffer
    AD1CON1SET = 0x8000; //Turn on the ADC
    AD1CHSbits.CH0SA = CH;
    AD1CON1SET = 0x0002; //Start sampling
    DelayUs(2); //Wait 2 microseconds
    AD1CON1CLR = 0x0002; // Start converting
    while (!(AD1CON1 & 0x0001)); //Waits for conversion
    return ADC1BUF0; //Returns content of the buffer
}
```

The software code shows how the ADC is converting analog to digital signals for the product.

4.2 PID controller

A PID controller continuously calculates an error value as the difference to a reference point and measured process variables.
PID is an abbreviation for proportional-integral-derivative, which 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 motors, to steer towards said reference point.

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

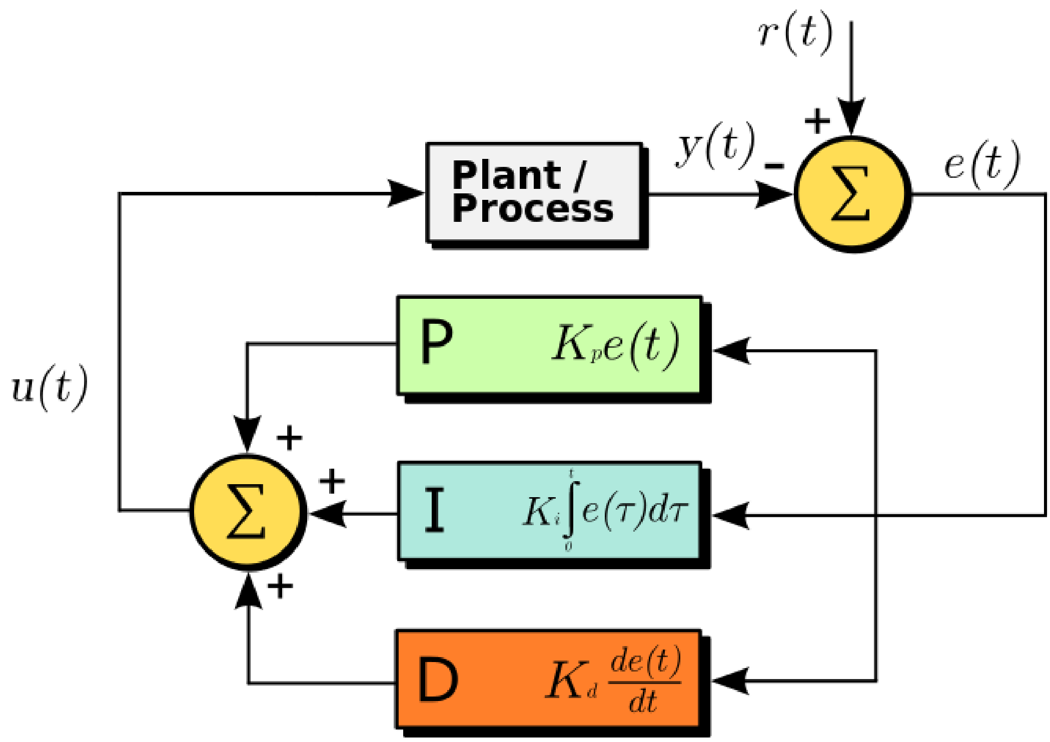


Figure 4.2: Block diagram showing the PID controller

Credits: `www.wikipedia.org/wiki/PID_controller`

4.2.1 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.

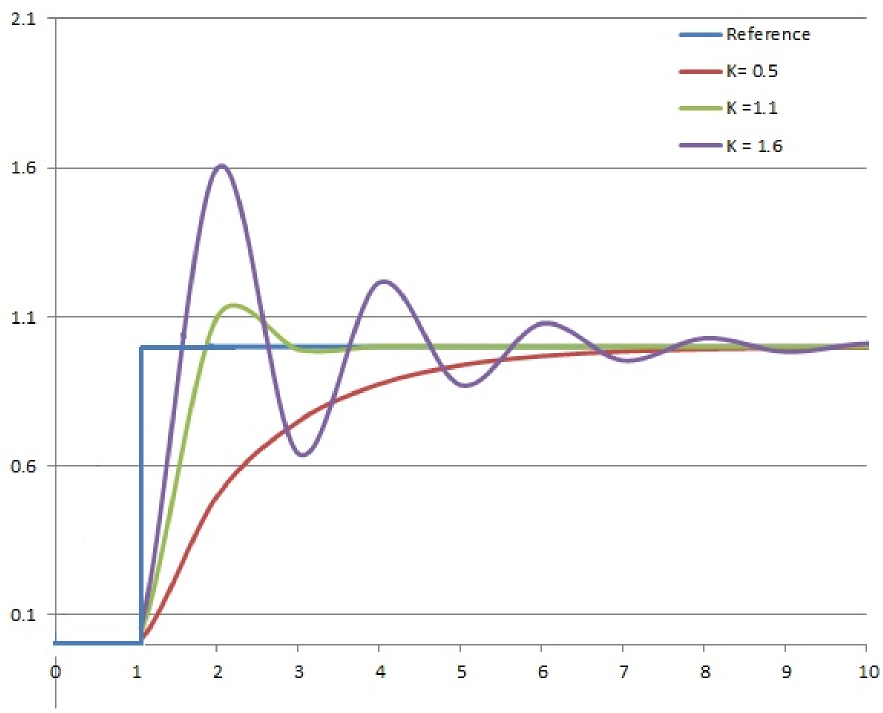


Figure 4.4: K_p with 3 values (K_i, K_d held constant)

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

4.2.2 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.

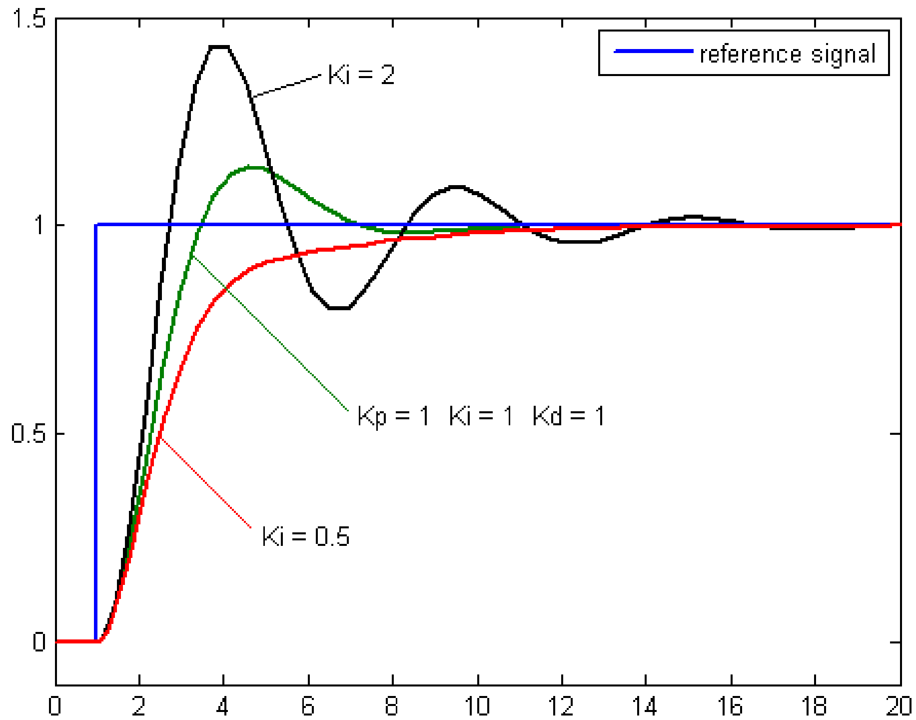


Figure 4.6: K_i shown with 3 values

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

4.2.3 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. 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.

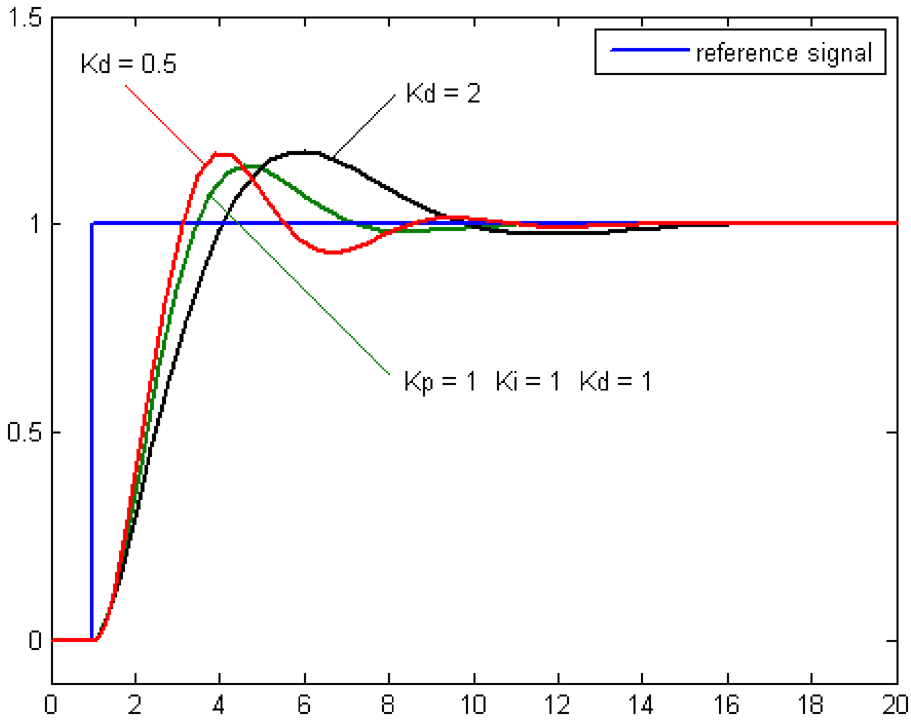


Figure 4.8: K_d shown with 3 values

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

4.2.4 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.

4.2.5 Steady-state error

In figure 3.2 the reference point is the blue line. The goal is to make the lines merge up and the steady-state is achieved. When the lines don't merge and the feedback line is slightly above or under the reference point there is a steady-state error. This steady-state error can be minimized by adjusting the proportional and integral term.

4.2.6 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.

4.2.7 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, and then reduce K_p by 10-20%.
Then set K_d to about 100 times the value of K_p and increase or decrease until the oscillation occurs, then reduce K_d by 10-20%.
If needed, K_i can be set to a very low number, in the case of this project, 0.01. This number is then increased or decreased, to give a fast response time without overshooting.
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 4.1: 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 4.1 explained

Table 4.1 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

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

4.2.8 PID Implementation

In the following example, the PID constants are global variables, and therefore not part of the described function. The values are as follows: kp = 8.5, ki = 0.01, kd = 850.

```
void PID(int sensorMean, int sensorSum)
{
  if(sensorSum > 0) //At least one sensor sees black
  {
    sensorPos = sensorMean/sensorSum; //Position of the line on the sensorarray
    sensorProp = sensorPos - 3;  //Proportional part
    sensorInt = sensorInt + sensorProp; //Integral part
    if(sensorInt > 100) //Reduce adjustment time by limiting Int
      sensorInt = 100;
    if(sensorInt < -100)
      sensorInt = -100;
    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 part calculated, which is done by subtracting the previous value of the proportional part from the current value.

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

```
    if(sensorError<0)
    {
      adjustedPWM[0] = initialPwm-sensorError; //Increase left
         motor, sensorError is negative here
      adjustedPWM[1] = initialPwm+sensorError; //Decrease right
         motor
      dir = 1; //turn right
    }
    else if(sensorError>0)
    {
      adjustedPWM[0] = initialPwm-sensorError;
      adjustedPWM[1] = initialPwm+sensorError;
      dir = 0; //turn left
    }
    else
    {
      adjustedPWM[0] = initialPwm;
      adjustedPWM[1] = initialPwm;
      dir = 2;
    }
  }
}
```

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 motor power on the left while decreasing motor 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.

4.3 Pulse-width modulation

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. Pulse-width modulation utilizes a square-wave signal, 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].

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

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

4.3.1 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.

$$D = \frac{T}{P} \cdot 100\%$$

- Where **D** is the duty cycle.
- **T** is the time the signal is active.
- **P** is the total period of the signal.

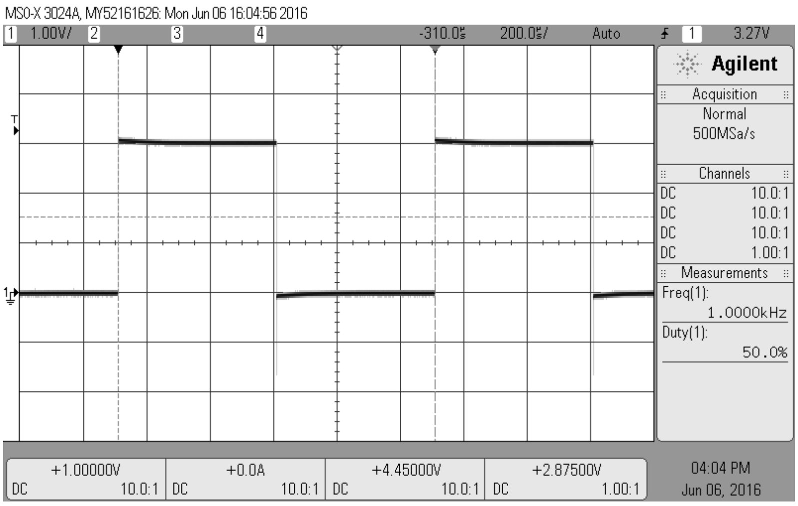


Figure 4.10: Picture of the duty cycles on an oscilloscope.

4.4 The interface

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 to connect and disconnect the robots blue-tooth transmission. It also features a terminal in the bottom, showing all the data sent from the MCU to the robot.

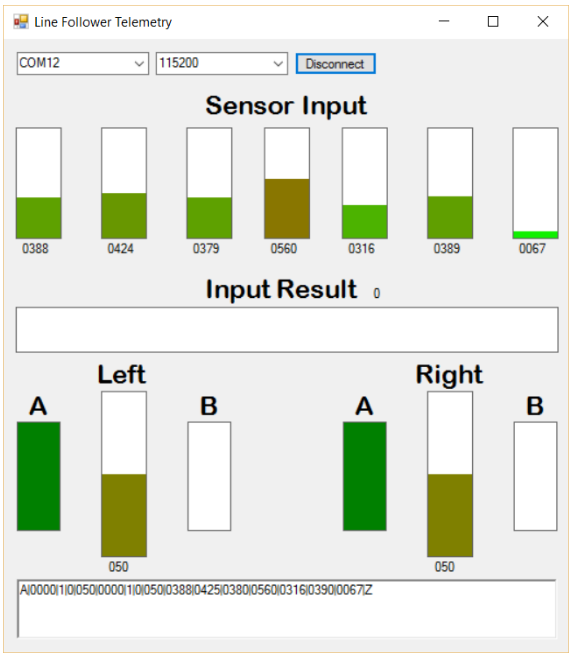


Figure 4.11: Screenshot of the C# GUI used to monitor the input readings

Test 5

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.

5.1 Unit Testing

The individual parts of the system will be tested in the following section

5.1.1 Sensor

The following test is to make sure that the selected sensors work as intended for the project.

Equipment

- Hameg HM8040-2 Triple Power Supply
- Fluke 45 Multimeter

Setup

Sensor is powered through the robot. Output from sensor is measured with multimeter.

Results

White Surface: Average voltage measured: 230 mV
Black Surface: Average voltage measured: 2,6 V
This shows that the sensor clearly measures a difference between light and dark surfaces, making it ideal for this application.

5.1.2 DC Motors

The following test is to make sure that the DC motors work as intended.

Equipment

- Hameg HM8040-2 Triple Power Supply
- Fluke 45 Multimeter

Setup

The motor is powered through the power supply. Amount of drawn current is measured with multimeter.

Results

Both motors have been measured at 6 V and show a steady current draw of 0,11 A when running freely.

5.1.3 H-Bridge

The following test is to make sure the H-bridge works as intended.

Equipment

- Hameg HM8040-2 Triple Power Supply
- Fluke 45 Multimeter
- Agilent MSO-X 3024A Oscilloscope.

Setup

H-Bridge is powered by the power supply. PWM and duty cycle is controlled by the function generator in the Oscilloscope. Output is measured with the multimeter. H-bridge motor direction is controlled manually.

Results

Direction: 0

- **Duty Cycle 20%:** 1,76 V measured.
- **Duty Cycle 50%:** 2,74 V measured.
- **Duty Cycle 80%:** 3,58 V measured.

Direction: 1

- **Duty Cycle 20%:** -1,57 V measured.
- **Duty Cycle 50%:** -2,55 V measured.
- **Duty Cycle 80%:** -3,50 V measured.

The results show that the H-Bridge unit is capaple of controlling the voltage output in both directions, with the output voltage level controlled by the PWM signal applied to the unit.

5.1.4 PWM

This test is to measure the PWM capability of the PIC32MX320F128H as well to see is the software implementation is capable of controlling the PWM module of the MCU.

Equipment

- Agilent MSO-X 3024A Oscilloscope.
- chipKIT Uno32 with PICkit 3 programmer.

Setup

The Uno32 board is programmed to run the PWM at different frequencies and duty cycles. Output is measured with the Oscilloscope.

Results

The test shows that the PWM module and developed software is working as intended.

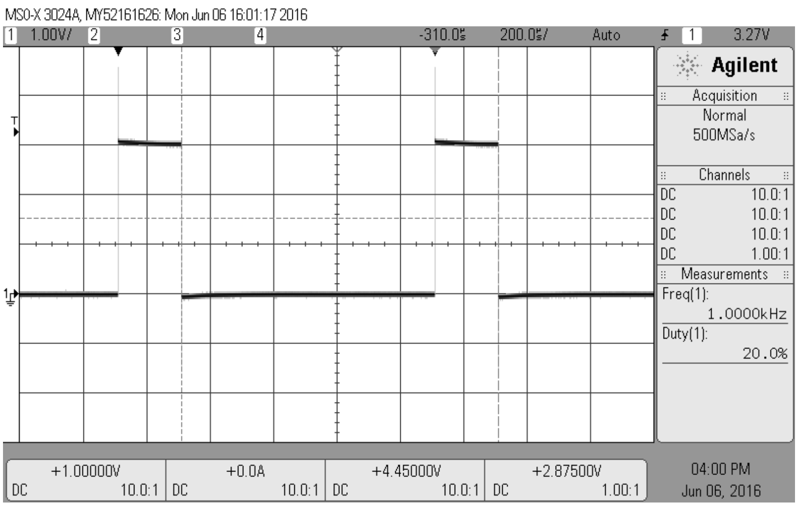


Figure 5.1: Output with 1000 Hz frequency and 20% Duty Cycle

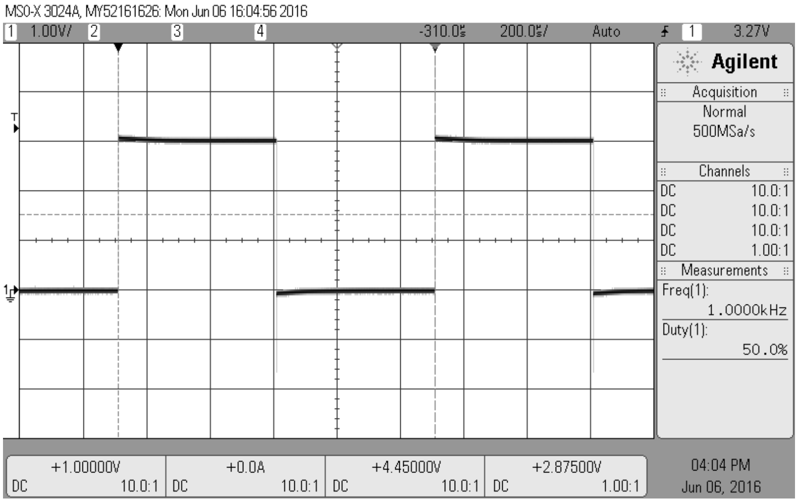


Figure 5.2: Output with 1000 Hz frequency and 50% Duty Cycle

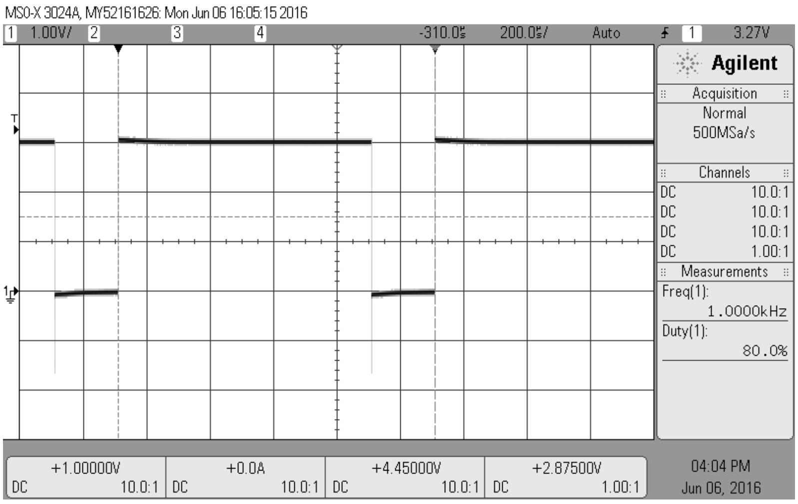


Figure 5.3: Output with 1000 Hz frequency and 80% Duty Cycle

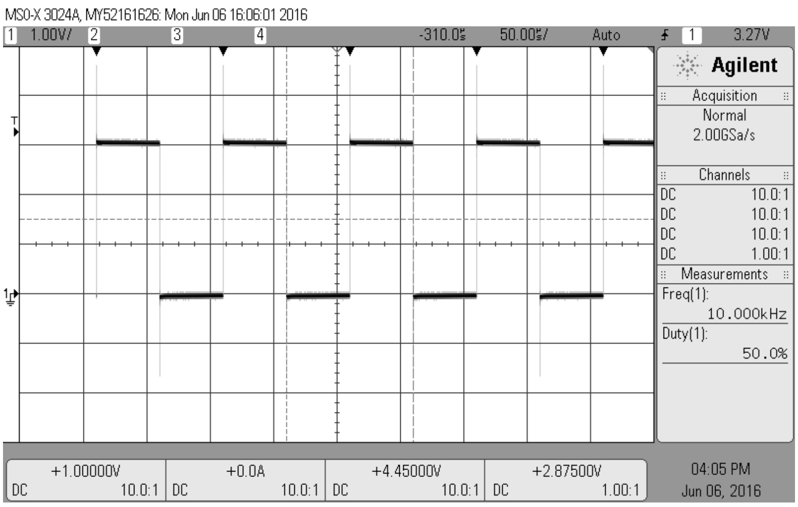


Figure 5.4: Output with 10000 Hz frequency and 50% Duty Cycle

5.1.5 ADC

This test is to show that the ADC module and the developed software for the module is working as intended.

Equipment

- Hameg HM8040-2 Triple Power Supply.
- chipKIT Uno32 with PICkit 3 programmer in debug mode.

Setup

The Uno32 is powered from the PICkit 3 programmer in debug mode to read the contents of a variabel containing the output of the ADC module. The input voltage on the ADC is supplied by the power supply.

Results

- **PS:** 0,499V - **Measured:** 163 (0,526V) - **Error:** 5,3%
- **PS:** 1,000V - **Measured:** 331 (1,068V) - **Error:** 6,7%
- **PS:** 2,006V - **Measured:** 665 (2,145V) - **Error:** 6,4%
- **PS:** 3,299V - **Measured:** 1023 (3,300V) - **Error:** 0,03%

This shows that the ADC and the developed software works within the required area for the application.

5.2 Integration Testing

5.2.1 PWM motor control

The following test is to show that the DC motors can be controlled by the H-bridge unit.

Equipment

- Hameg HM8040-2 Triple Power Supply.
- chipKIT Uno32 with PICkit 3 programmer.

Setup

The H-bridge module is powered from the power supply. The control signals is controlled by the Uno32 board.

Results

- **PWM: 20%:** Motors are not turning. Audible whine.
- **PWM: 30%:** Motors are not turning. Higher audible whine.
- **PWM: 37%:** Motors are turning slowly with assisted start.
- **PWM: 40%:** Motors are turning slowly.
- **PWM: 100%:** Motors are turning maximum speed.

This shows that we the current setup of Motors and H-bridge unit, the variable control span is between an duty cycle of 37% and 100%.

5.2.2 Robot to Interface communication

This test is to show that the system is capable of transmitting current data from the robot to the interface and accurately show it to the user.

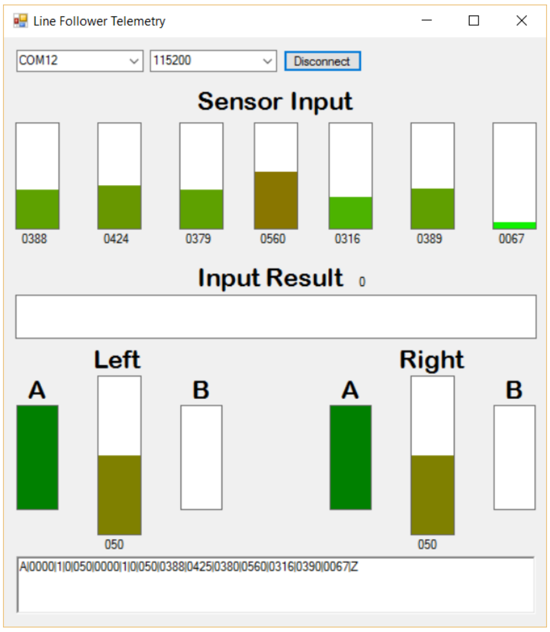
Equipment

- chipKIT Uno32 with PICkit 3 programmer.
- BlueSMIRF silver Bluetooth module.
- PC running C# interface.

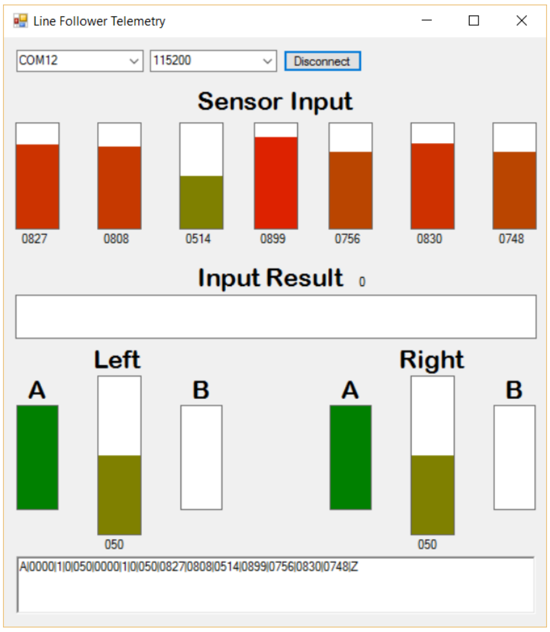
Setup

The robot is placed so that the sensors measures over a white and a black surface. data is sent over bluetooth to the interface and data recorded.

Results



(a) Interface showing data sent with sensors placed over white surface



(b) Interface showing data sent with sensors placed over black surface

5.3 System Testing

This test is to show that the system works as intended.

Equipment

- chipKIT Uno32 with PICkit 3 programmer.
- BlueSMIRF silver Bluetooth module.
- PC running C# interface.

Setup

The system is tested by placing the robot in 3 different situations.

- Centered on line.
- Line at the right end of sensor array.
- Line at the left end of sensor array.

Results

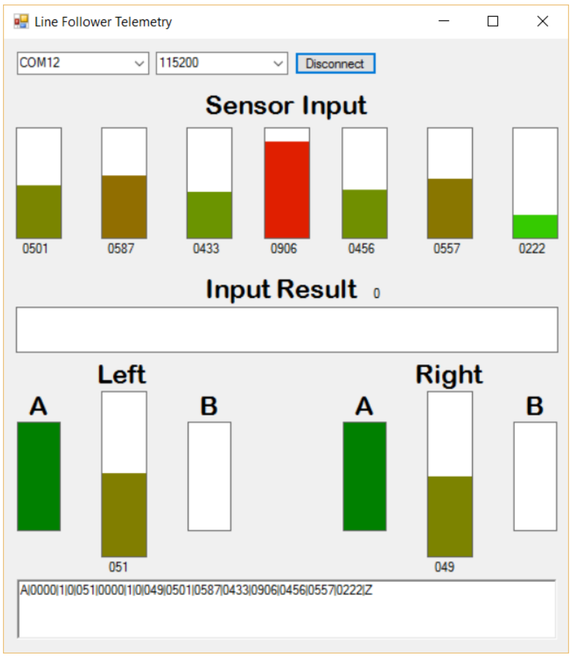
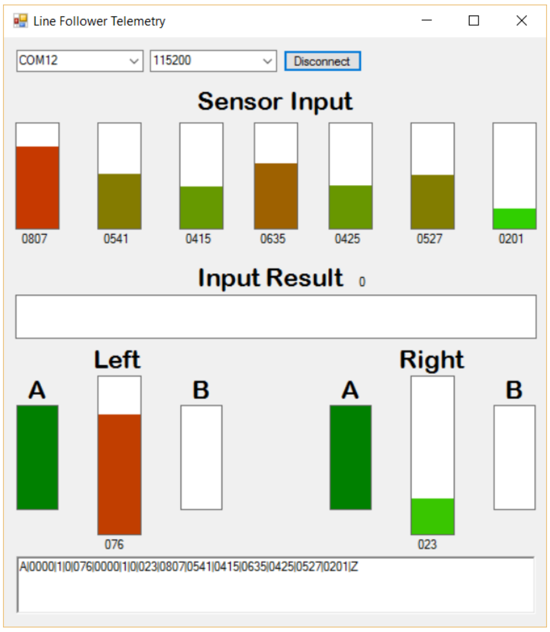
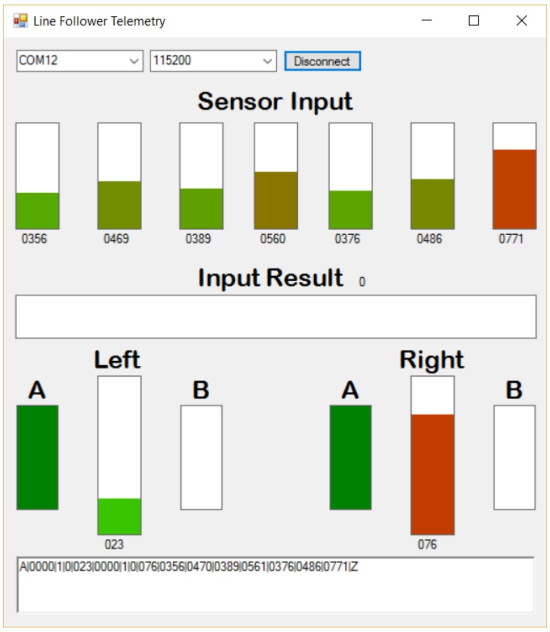


Figure 5.6: Only the middle sensor is detecting the line



(a) Sensor on the left detects line, initiate left turn



(b) Sensor on the right detects line, initiate right turn

5.4 Acceptance Testing

This test is to show that the product as a whole works on a simple test track.

Equipment

- Complete robot with working control software
- Test track

Setup

The robot is calibrated on the test track and placed on the line. The robots behaviour is recorded and deemed acceptable or not.

Results

The robot drives acceptably on the track most of the time. Other times, it seems to overshoot on the corners. See attached videos for example tests.

Conclusion 6

The goal of this project was to make robot that could follow a black line by utilizing sensors and feedback control algorithms.

During the project, a higher understanding of the workings of systems such as PID, ADC and PWM, and how they can be implemented in hardware and software was achieved. The solution was designed around the Magician Chassis and chipKIT Uno32 MCU Board. C code for the MCU has been made and implements the required systems: PID, ADC and PWM. Furthermore, an interface has been developed in C#, that receives data through Bluetooth that the robot transmits through a BlueSMiRF Silver module.

During the process, several problems occurred, most severe issues with the implementation of the PID algorithm. The algorithm was at first implemented wrong, adding in the derivative part instead of subtracting. This lead to a serious increase in time taken to tune the algorithm.
Furthermore, the first iteration of the robot included a homemade motor shield, which turned out to be faulty.
The product development ended in a success, resulting in a working robot that is able to navigate a line following track using the developed PID algorithm.

Appendices 7

7.1 Group collaboration agreement

7.1.1 Contact Information

Table 7.1: 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

7.1.2 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.

7.1.3 Deadline

- Hand in June 7th.

7.1.4 Milestones and goals

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

April 2016

MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY	SUNDAY
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20 Project day - Tidsplan, rapportstruktur	21	22	23	24
25	26 Project day - Argumentation for komponenter	27	28 Project day -	29 Project day Finish Shield	30	1
2	3	NOTES				

Figure 7.1: 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 7.2: 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!!!*** ***Kl 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 7.3: 4 Work days in June

List of references 8

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

9 Software appendix

9.1 C code

main.c:

```c
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#define _SUPPRESS_PLIB_WARNING 1
#define _DISABLE_OPENADC10_CONFIGPORT_WARNING 1
#include <plib.h>
#include "setup.h"
#include "Functions.h"
#include "UART.h"
#include <math.h>
#include "Delay.h"
#include "ADC.h"

//Left wheel
#define DIRASetup TRISFbits.TRISF1
#define DIRA PORTFbits.RF1
//Right wheel
#define DIRBSetup TRISDbits.TRISD10
#define DIRB PORTDbits.RD10
//Bottom button
#define CalLSetup TRISDbits.TRISD8
#define CalL PORTDbits.RD8
//Top button
#define CalHSetup TRISDbits.TRISD9
#define CalH PORTDbits.RD9

int ADCHighestValue[7] = {0,0,0,0,0,0,0};
int ADCLowestValue[7] = {1024,1024,1024,1024,1024,1024,1024};
int caliFlag = 0;

//PWM
float kp = 8.5;
float ki = 0.01;
float kd = 850;
float lastError = 0;
float integral = 0;
int maxPwm = 80;
int initialPwm = 50;
int dir = 2; //0 = left, 1 = right
int turn = 0;
```

```
int DIR[] = {0, 0};

//PID variables
float sensorMean = 0;
float sensorSum = 0;
float sensorPos = 0;
float sensorProp = 0;
float sensorInt = 0;
float sensorDer = 0;
float sensorLastProp = 0;
float sensorError = 0;
float sensorOn[7] = {0,0,0,0,0,0,0};
int adjustedPWM[2] = {0,0};
int correctFlag = 0;

int main(int argc, char** argv) {
    int ADCs[] = {2, 8, 14, 3, 5, 9, 15}; //Pins for each sensor, in order
    int ADCAvgValue[7];

    int ADCData[7];
    int PWM[] = {0, 0};

    int POS[] = {0, 0};
    int i = 0;
    int averageFlag = 0;

    UART1Init(115200, 1);
    initPWM();
    DIRASetup = 0;
    DIRBSetup = 0;
    CalLSetup = 1;
    CalHSetup = 1;

    for (;;) {
        for(i = 0;i<7;i++){
            ADCData[i] = analogRead(ADCs[i]); //Reads data on all sensors
        }
        checkCalState(ADCs);
        //Working PID
        if(caliFlag == 1) //Run if calibrated
        {
            if(averageFlag == 0) //If average value on each sensor has not been calculated yet
            {
                for(i = 0;i<7;i++)
                    ADCAvgValue[i] = (ADCHighestValue[i]+ADCLowestValue[i])/2; //Calculate average value for each sensor
                averageFlag = 1;
```

```
            }
            for(i=0;i<7;i++){
                if(ADCData[i]>ADCAvgValue[i]) //If sensor detects a line, set this sensor to be 1 in sensorOn
                    sensorOn[i] = 1;
                else
                    sensorOn[i] = 0;
            }
            sensorMean = 0;
            sensorSum = 0;
            for(i=0;i<7;i++){
                sensorMean += sensorOn[i]*i*1; //Makes sensors ready for weighted mean calculation
                sensorSum += sensorOn[i]; //Calculate how many sensors are on
            }
            PID(sensorMean, sensorSum);//Calculates PID
            PWM[0] = adjustedPWM[0];
            PWM[1] = adjustedPWM[1];
            adjustDuty(1, PWM[0]); //Adjusts the duty cycle for left motor
            adjustDuty(2, PWM[1]); //Right motor
        }
        DIRA = DIR[0];
        DIRB = DIR[1];

        //Telemetry to interface
        telemetryOut(ADCData, PWM, DIR, POS); //Sends all data through UART
        DelayMs(12);
    }
    return (EXIT_SUCCESS);
}
```

Functions.c:

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

#define CalL PORTDbits.RD8
#define CalH PORTDbits.RD9

extern int ADCHighestValue[];
extern int ADCLowestValue[];
extern int caliFlag;

extern float kp;
extern float ki;
extern float kd;
extern int initialPwm;
extern int maxPwm;
extern int dir;

extern float sensorPos;
extern float sensorProp;
extern float sensorInt;
extern float sensorDer;
extern float sensorLastProp;
extern float sensorError;
extern int adjustedPWM[2];

int calLFlag = 0;
int calHFlag = 0;

void dec_to_str(char* str, int val, size_t digits) {
    size_t i = 1u;
    for (; i <= digits; i++) {
        str[digits - i] = (char) ((val % 10u) + '0');
        val /= 10u;
    }
    str[i - 1u] = '\0'; // assuming you want null terminated
       strings?
}

void initPWM() {
    int sysClk = 80000000;
    int pwmFreq = 1000;
    int prescaleV = 1;
    int dutyCycle = 0;

    OC2CON = 0x0000;
    OC2R = 0x00638000;
    OC2RS = 0x00638000;
    OC2CON = 0x0006;
```

```
    T2CONSET = 0x0008;
    PR2 = (sysClk / (pwmFreq * 2) * prescaleV) - 1;
    OC2RS = (PR2 + 1)*((float) dutyCycle / 100);

    T2CONSET = 0x8000;
    OC2CONSET = 0x8020;

    OC4CON = 0x0000;
    OC4R = 0x00638000;
    OC4RS = 0x00638000;
    OC4CON = 0x0006;
    T4CONSET = 0x0008;
    PR4 = (sysClk / (pwmFreq * 2) * prescaleV) - 1;
    OC4RS = (PR4 + 1)*((float) dutyCycle / 100);

    T4CONSET = 0x8000;
    OC4CONSET = 0x8020;

}

void adjustDuty(int channel, int duty) {
    switch (channel) {
        case 1:
            OC2RS = (PR2 + 1)*((float) duty / 100);
            break;
        case 2:
            OC4RS = (PR4 + 1)*((float) duty / 100);
            break;
    }
}

void telemetryOut(int* ADCData, int* PWM, int* DIR, int* POS) {
    char str[4u + 1u];
    int i = 0;

    /* TRANSMISSION START */
    UART1Write("A|");

    dec_to_str(str, POS[0], 4u);
    UART1Write(str);
    UART1Write("|");
    if(DIR[0] == 0) UART1Write("0");
    else UART1Write("1");
    UART1Write("|");
    if(DIR[0] == 0) UART1Write("1");
    else UART1Write("0");
    UART1Write("|");
    dec_to_str(str, PWM[0], 3u);
    UART1Write(str);
    UART1Write("|");

```

```c
    dec_to_str(str, POS[1], 4u);
    UART1Write(str);
    UART1Write("|");
    if(DIR[1] == 0) UART1Write("0");
    else UART1Write("1");
    UART1Write("|");
    if(DIR[1] == 0) UART1Write("1");
    else UART1Write("0");
    UART1Write("|");
    dec_to_str(str, PWM[1], 3u);
    UART1Write(str);
    UART1Write("|");

    for (i = 0; i < 6; i++) {
        dec_to_str(str, ADCData[i], 4u);
        UART1Write(str);
        UART1Write("|");
    }
    dec_to_str(str, ADCData[6], 4u);
    UART1Write(str);

    UART1WriteLn("|Z");
    /* TRANSMISSION END */
}

//Checks for calibration, if not, calibrate
void checkCalState(int* adc)
{
    int i;
    int j;
    int tempRead;
    if(CalL < 1){
       for(i = 0;i<10;i++){
           for(j = 0;j<7;j++){
               tempRead = analogRead(adc[j]);
               if(tempRead < ADCLowestValue[j])
                   ADCLowestValue[j] = tempRead;
           }
       }
       calLFlag = 1;
    }
    if(CalH < 1){
       for(i = 0;i<10;i++){
           for(j = 0;j<7;j++){
               tempRead = analogRead(adc[j]);
               if(tempRead > ADCHighestValue[j])
                   ADCHighestValue[j] = tempRead;
           }
       }
       calHFlag = 1;
    }
```

```
    if(caliFlag != 1){
        if(calLFlag == 1 && calHFlag == 1) {
            DelayMs(1500);
            caliFlag = 1;
        }
    }
}

void PID(int sensorMean, int sensorSum)
{
    int PWM[] = {0,0};
    if(sensorSum > 0) //As long as there is at least one sensor active
    {
        sensorPos = sensorMean/sensorSum; //Position of the line on the sensorarray
        sensorProp = sensorPos - 3;  //Proportional part, position minus middle sensor position
        sensorInt = sensorInt + sensorProp; //Integral part
        if(sensorInt > 100) //Makes sure Integral is not too large, reduces time to adjust
            sensorInt = 100;
        if(sensorInt < -100)
            sensorInt = -100;
        sensorDer = sensorProp - sensorLastProp; //Derivative part
        sensorError = (sensorProp * kp)+(sensorInt*ki)+(sensorDer*kd); //PID calculation
        sensorLastProp = sensorProp; //Saves proportional for next derivative
        if(sensorError < -(initialPwm)) //Sets an upper cap for adjustment
            sensorError = -(initialPwm);
        if(sensorError > (initialPwm))
            sensorError = (initialPwm);
        if(sensorError<0){
            PWM[1] = initialPwm+sensorError; //Decrease right motor, sensorError is negative here
            PWM[0] = initialPwm-sensorError; //Increase left motor
            dir = 1; //turn right
        }
        else if(sensorError>0){
            PWM[1] = initialPwm+sensorError;
            PWM[0] = initialPwm-sensorError;
            dir = 0; //turn left
        }
        else{
            PWM[0] = initialPwm;
            PWM[1] = initialPwm;
            dir = 2;
```

```
        }
    }
    else
    {
        PWM[0] = adjustedPWM[0];
        PWM[1] = adjustedPWM[1];
    }
    adjustedPWM[0] = PWM[0];
    adjustedPWM[1] = PWM[1];
}
```

UART.c:

```c
#include <xc.h>

extern void UART1Init(int baudrate, int stopBit) {
    if (U1STAbits.OERR != 0) {                  // Check recieve buffer for overflow
        U1STAbits.OERR = 0;                     // Reset flag if set
    }

    TRISBbits.TRISB4 = 0;                       // Set B4 as output
    U1MODEbits.BRGH = 1;                        // Baudrate generator high mode
    U1MODEbits.PDSEL = 0;                       // Parity and data selection bits 8bit, no parity
    if (stopBit == 1) {
        U1MODEbits.STSEL = 0;                   // Stopbit set to 1 stopbit
    } else {
        U1MODEbits.STSEL = 1;                   // Stopbit set to 2 stopbit
    }
    U1BRG = (40000000 / 4 / baudrate) - 1;      // Set baudrate register
    U1STAbits.URXEN = 1;                        // Enable Reciever
    U1STAbits.UTXEN = 1;                        // Enable Transmitter
    U1MODEbits.ON = 1;                  // Turn on USART
}

extern char UART1Putc(char c) {
    while (U1STAbits.UTXBF);                    // Waits for trasmit buffer to bet not full
    U1TXREG = c;                                // Puts contents of C into buffer
    return c;
}

extern void UART1WriteLn(char *data) {
    while (*data) UART1Putc(*data++);
    UART1Putc('\n');
}
```

ADC.c:

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

extern int analogRead(char CH) {
    AD1PCFG = ~CH; // PORTB = Digital; RB2 = analog
    AD1CON1 = 0x0000; // SAMP bit = 0 ends sampling
    // and starts converting
    //AD1CHS = 0x00020000; // Connect RB2/AN2 as CH0 input
    // 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
    AD1CHSbits.CH0SA = CH;
    AD1CON1SET = 0x0002; // start sampling ...
    DelayUs(2); // for 2 us
    AD1CON1CLR = 0x0002; // start Converting
    while (!(AD1CON1 & 0x0001)); // conversion done?
    return ADC1BUF0;
}
```

9.2 C# code - interface

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO.Ports;

namespace Line_follower_Telemetry
{
    public partial class Main : Form
    {
        int[] baudRates = { 2400, 4800, 9600, 19200, 38400,
            57600, 115200 };
        string[] ports = SerialPort.GetPortNames();

        public delegate void SCDelegate();
        SerialPort SP = new SerialPort();
        delegate void printDelegate(string data);

        int VDA = 0, VDB = 0, HDA = 0, HDB = 0;

        int res = 0;
        int resMin = -3;
        int resMax = 3;
        int resPosOffset = 0;

        public Main()
        {
            InitializeComponent();
            for (int i = 0; i < baudRates.Length; i++)
            {
                BaudCB.Items.Add(baudRates[i]);
            }
            BaudCB.SelectedItem = 115200;
            PortsCB.Items.AddRange(ports);
            PortsCB.SelectedIndex = 0;
            Point resP = ResPositive.PointToScreen(Point.Empty);
            resPosOffset = resP.X-8;
            resLabel.Text = resPosOffset.ToString();
        }

        private void Main_Load(object sender, EventArgs e)
        {

```

```
        }

        void testPrint(string data)
        {
            RawSerial.Text = data;
            serialDeconstruct(data);
            if(data[data.Length - 1] == 'Z')
            {
                RawSerial.AppendText("\n");
            }
            RawSerial.SelectionStart = RawSerial.Text.Length;
            RawSerial.ScrollToCaret();

        }

        private void serialDeconstruct(string data)
        {
            string[] dataP = data.Split(new string[] { "|" }, StringSplitOptions.None);
            if (dataP.Length == 17)
            {
                vPWMLabel.Text = dataP[4];
                if(Convert.ToInt32(dataP[4]) > 0 && Convert.ToInt32(dataP[4]) < 150)
                {
                    vPWM.Height = map(Convert.ToInt32(dataP[4]), 0, 100, 150, 0);
                    backpanel1.BackColor = Color.FromArgb(map(vPWM.Height, 150, 0, 0, 255), map(vPWM.Height, 150, 0, 255, 0), 0);
                }
                VDA = Convert.ToInt32(dataP[2]);
                if (VDA == 1) vDirA.BackColor = Color.Green;
                else vDirA.BackColor = Color.White;
                VDB = Convert.ToInt32(dataP[3]);
                if (VDB == 1) vDirB.BackColor = Color.Green;
                else vDirB.BackColor = Color.White;
                hPWMLabel.Text = dataP[8];
                if(Convert.ToInt32(dataP[8]) > 0 && Convert.ToInt32(dataP[8]) < 150)
                {
                    hPWM.Height = map(Convert.ToInt32(dataP[8]), 0, 100, 150, 0);
                    Backpanel2.BackColor = Color.FromArgb(map(hPWM.Height, 150, 0, 0, 255), map(hPWM.Height, 150, 0, 255, 0), 0);
                }
                HDA = Convert.ToInt32(dataP[6]);
                if (HDA == 1) hDirA.BackColor = Color.Green;
                else hDirA.BackColor = Color.White;
                HDB = Convert.ToInt32(dataP[7]);
```

```
                if (HDB == 1) hDirB.BackColor = Color.Green;
                else hDirB.BackColor = Color.White;
                Sensor1Label.Text = dataP[9];
                Sensor1.Height = map(Convert.ToInt32(dataP[9]), 0, 1023, 100, 0);
                Backpanel3.BackColor = Color.FromArgb(map(Sensor1.Height, 100, 0, 0, 255), map(Sensor1.Height, 100, 0, 255, 0), 0);
                Sensor2Label.Text = dataP[10];
                Sensor2.Height = map(Convert.ToInt32(dataP[10]), 0, 1023, 100, 0);
                Backpanel4.BackColor = Color.FromArgb(map(Sensor2.Height, 100, 0, 0, 255), map(Sensor2.Height, 100, 0, 255, 0), 0);
                Sensor3Label.Text = dataP[11];
                Sensor3.Height = map(Convert.ToInt32(dataP[11]), 0, 1023, 100, 0);
                Backpanel5.BackColor = Color.FromArgb(map(Sensor3.Height, 100, 0, 0, 255), map(Sensor3.Height, 100, 0, 255, 0), 0);
                Sensor4Label.Text = dataP[12];
                Sensor4.Height = map(Convert.ToInt32(dataP[12]), 0, 1023, 100, 0);
                Backpanel6.BackColor = Color.FromArgb(map(Sensor4.Height, 100, 0, 0, 255), map(Sensor4.Height, 100, 0, 255, 0), 0);
                Sensor5Label.Text = dataP[13];
                Sensor5.Height = map(Convert.ToInt32(dataP[13]), 0, 1023, 100, 0);
                Backpanel7.BackColor = Color.FromArgb(map(Sensor5.Height, 100, 0, 0, 255), map(Sensor5.Height, 100, 0, 255, 0), 0);
                Sensor6Label.Text = dataP[14];
                Sensor6.Height = map(Convert.ToInt32(dataP[14]), 0, 1023, 100, 0);
                Backpanel8.BackColor = Color.FromArgb(map(Sensor6.Height, 100, 0, 0, 255), map(Sensor6.Height, 100, 0, 255, 0), 0);
                Sensor7Label.Text = dataP[15];
                Sensor7.Height = map(Convert.ToInt32(dataP[15]), 0, 1023, 100, 0);
                Backpanel9.BackColor = Color.FromArgb(map(Sensor7.Height, 100, 0, 0, 255), map(Sensor7.Height, 100, 0, 255, 0), 0);
                res = Convert.ToInt32(dataP[5]);
                resLabel.Text = res.ToString();
                if (res < 0 && res >= resMin)
                {
                    ResBack.BackColor = Color.FromArgb(map(res, resMin, 0, 255, 0), map(res, resMin, 0, 0, 255), 0);
```

```
                    ResNegative.Width = map(res, 0, resMin, 248, 0);
                } else if (res > 0 && res <= resMax)
                {
                    ResBack.BackColor = Color.FromArgb(map(res, resMax, 0, 255, 0), map(res, resMax, 0, 0, 255), 0);
                    ResPositive.Width = map(res, 0, resMax, 248, 0);
                    ResPositive.Left = resPosOffset + ((248 / 3) * res);
                } else if (res == 0)
                {
                    ResPositive.Width = 248;
                    ResNegative.Width = 248;
                    ResPositive.Left = resPosOffset;
                }
            }
        }

        private void SPPrint(object sender, SerialDataReceivedEventArgs e)
        {

            SerialPort SP = (SerialPort)sender;
            printDelegate PD = new printDelegate(testPrint);
            this.Invoke(PD, SP.ReadLine());
        }

        private void SPSPrint(object sender, RTSerialCom.DataStreamEventArgs e)
        {
            RTSerialCom.SerialClient SPS = (RTSerialCom.SerialClient)sender;
            printDelegate PD = new printDelegate(testPrint);
            byte[] ByteArray = new byte[18];
            SPS.Receive(ByteArray, 0, ByteArray.Length);
            string data = System.Text.Encoding.Default.GetString(ByteArray);
            this.Invoke(PD, data);
        }

        public void SerialClose()
        {
            SP.DiscardInBuffer();
            SP.DiscardOutBuffer();
            SP.Close();
            SP.DataReceived -= new SerialDataReceivedEventHandler(SPPrint);
            ConBot.Text = "Connect";
        }
```

```

        private int map(int x, int in_min, int in_max, int out_min, int out_max)
        {
            return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
        }

        private void ConBot_Click_1(object sender, EventArgs e)
        {
            if (SP.IsOpen)
            {
                BeginInvoke(new SCDelegate(SerialClose));
            }
            else
            {
                if (BaudCB.SelectedItem == null || PortsCB.SelectedItem == null)
                {
                    RawSerial.Text = "Parameters not set!";
                }
                else
                {
                    SP.BaudRate = (int)(BaudCB.SelectedItem);
                    SP.PortName = PortsCB.SelectedItem.ToString();
                    SP.Open();
                    SP.DataReceived += new SerialDataReceivedEventHandler(SPPrint);
                    ConBot.Text = "Disconnect";
                    RawSerial.Clear();
                }
            }
        }
    }
```