Technical University of Denmark

André D. B. Christensen (110033)

Jacob A. F. Pedersen (120374)

Eyebot - A Line Following Robot Based On Camera Tracking

4th Semester Project in Control Theory

Eyebot - A Line Following Robot Based On Camera Tracking

This report was prepared by:
André D. B. Christensen (110033)
Jacob A. F. Pedersen (120374)

Advisors:
Henrik Møller, DTU Ballerup Campus
Anna Friesel, DTU Ballerup Campus

DTU Diplom
Lautrupvang 15
2750 Ballerup
Denmark
Tel: +45 3588 5088

diplom@diplom.dtu.dk

Project period: September 2013 - December 2013

ECTS: 10

Education: BEng

Field: Electrical Engineering

Contents

CHAPTER 1

Introduction

Autonomous robots are robotic devices capable of performing tasks based on inputs from various types of sensors, without human interaction. Much research in the recent years has focused on the development of autonomous cars (also known as Unmanned Automobiles) which can drive without interaction from the driver, but just by sensing the environment. This is an interesting field, as it may lower the number of traffic accidents and especially reduce the traffic congestion in bigger cities by providing a better traffic flow.

In this present project we designed and constructed an autonomous robot, capable of following a line by using a mixture of cameras and IR sensors for gathering input about the surroundings. Close attention was paid to the use of camera sensing and image processing for detecting and extracting the line. The traditional method has been to use an array of photo-sensors, but due to the low resolution, an alternative method is necessary for archiving a better stability.

This paper outlines the work done, and shows how a camera can be used instead of a traditional array of photo sensors. Further more, as an additional feature, it is explained how the live image data is streamed to a computer, for a live preview of what the 'robot sees'.

Our robot may in the following chapters be referred to as 'the robot', 'the system' or simply 'Eyebot'.

1.1 Subsystems and Block Diagram

The robot consist of several subsystems, which have different responsibilities. The block diagram in figure 1.1, outlines the basic components of the system with connections betweens them.

As seen on the block diagram, the main processor of the system is an ARM-based Raspberry Pi computer. The Raspberry Pi first of all acts as the overall state

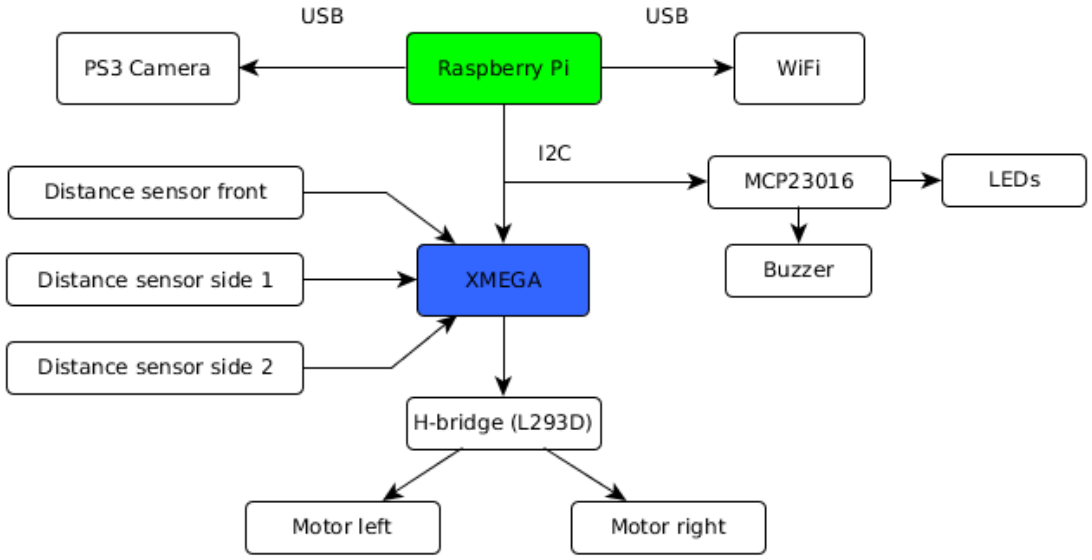


Figure 1.1: Block diagram of the subsystems

machine when driving the track. It knows when to follow the line, go to the wall, rotate and so on (more on that in chapter 7). Moreover the Raspberry Pi takes care of capturing images from the camera, processing the images, and calculating the error signals when following the line. As an additional feature, the live image data when following the line, are transmitted over WiFi to a laptop, for live preview of the robot's processed and annotated image. The image capture and processing is discussed in detail in chapter 2.

Below the Raspberry Pi, an Atmel XMEGA192C micro-controller is responsible for powering and controlling the motors. The XMEGA is instructed directly by the Raspberry Pi, by commands received over the I2C (Two Wire Interface) bus. Equally important the XMEGA reads analog signals from the three distance sensors, and makes them available for the Raspberry Pi through the I2C bus. The motor-controller is discussed in chapter **??** and the distance sensors are discussed in chapter 4.

As an additional feature, the robot has an array of LEDs mounted on its bumper, and a buzzer for indicating state changes. Those are powered by an MCP23016 I/O expander chip, that communicates with the Raspberry Pi using I2C. This subject will not receive any more attention in the report than this.

1.2 Requirements Specification

1.2.1 Assumptions

We assume an average speed for the robot of 0.5 m/s. By looking at the line, we found that it should be reasonable to make corrections every 0.02 m. This gives an maximum cycle time of 40 ms. Cycle time includes capturing and processing of the image, calculation of the error and correction value, transmitting the corrective actions to the motors, and waiting for the motors to respond and react. By tilting the camera forward, we might be able to detect future obstacles, and take required actions such as slowing down.

1.2.2 Formal requirements

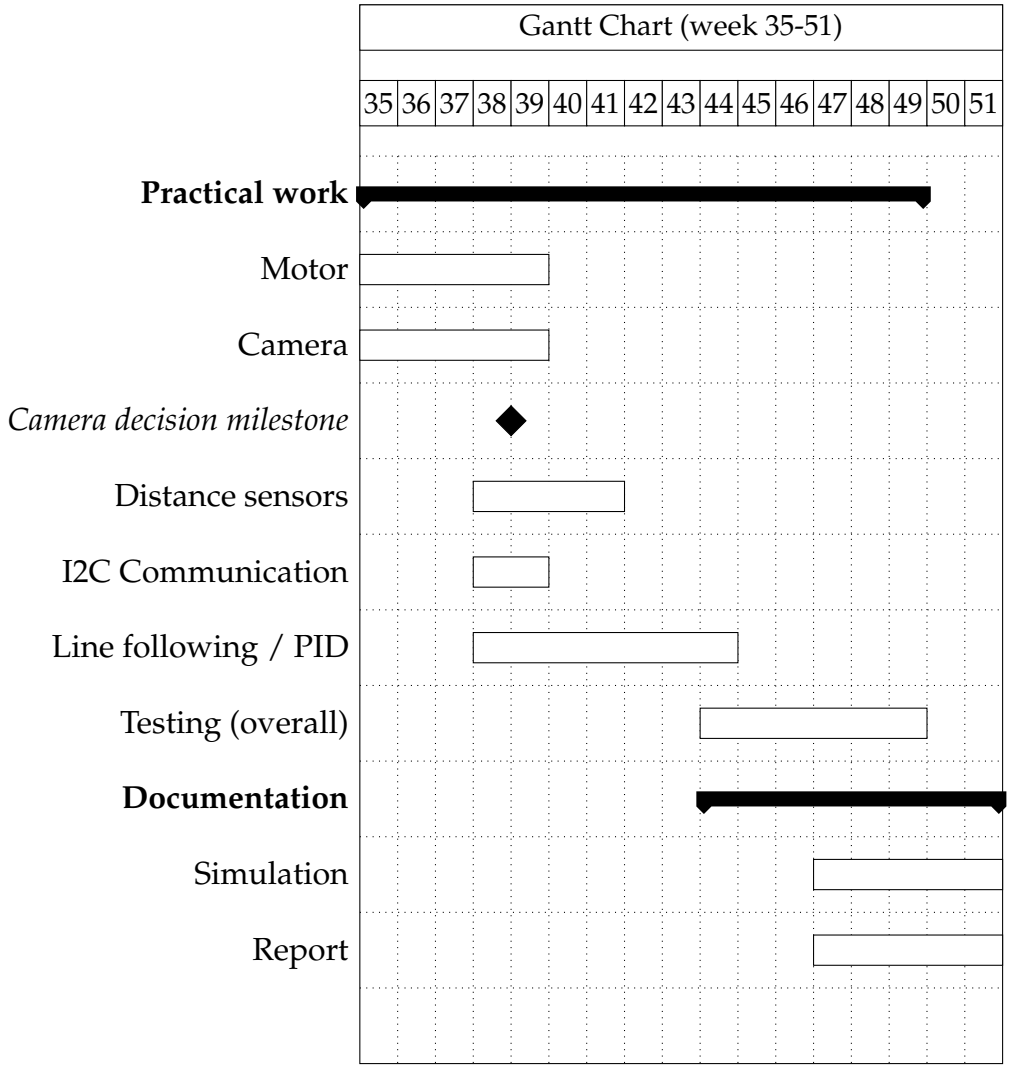
1. The robot should be able to follow the track on first floor. This includes
   - Make a decision on which way to turn when seeing the line first time
   - When a short peace of tape intersects the line, the robot should turn 90 degrees to the left and drive towards the wall. It should stop 20 centimeters from the wall, turn 135 degrees and go forward until the line is visible again, and then follow the line as usual.
   - When a short line is sensed on the left side of the line, the robot should speed up, and try to go as fast as possible for the rest of the track.
   - At the end of the track, the robot should drive to the wall, keeping a distance of 20 centimeters to the wall, and then follow the wall to the end of the track.
2. The robot should be able to drive at a maximum speed of 0.5 m/s.
3. The robot should make corrections every 0.02 m, which is every 40 ms at 0.5 m/s.
4. The line should be sensed using a camera and image processing.
5. A Raspberry Pi should be the main controller in the system.
6. The Raspberry PI should capture images of the floor, analyze the images, and calculate the error.
7. The motors should be controlled by an AVR micro-controller, that interfaces the Raspberry Pi using I^2C (TWI).

8. The speed of the motors should be controlled using PWM signals through a H-bridge.

9. The distance to the wall should be measured using a distance sensor in the front, and one or more on the left side of the robot.

1.3 Time Schedule

Below is the time schedule of the project plotted in a Gantt Chart. The Gantt chart is from week 35 (project start-up in September) to week 51 (hand-in and presentation in December). The camera has been the most experimental feature of the system, why we decided to set a milestone, at which a decision whether to use the camera or not, was to be made.



CHAPTER 2

Tracking And Processing The Line Using Image Processing

In this section the methods for capturing images and performing the necessary processing are explained in detail, with emphasis on how the image is processed to produce an error signal for the control system.

2.1 Capturing The Image

As explained in the introduction, the camera used for this project is a Sony PS3 Eye camera, which is based on OmniVision's OV543 camera sensor. All image capture and processing is carried out in the Raspberry Pi running the Raspbian Linux distribution.

The PS3 Eye camera is supported directly by the Linux kernel, and can be controlled by user software through the Video4Linux driver interface. The PS3 Eye camera was chosen for this project, because of its specification and low price. The camera is capable of shooting 125 frame per second, why it is perfect for computer vision applications (what is actually also was made for).

When the robot starts, the camera is set up to deliver 30 frames per second, in gray-scale colors. As part of the initialization, a callback is given to the camera module, and is called every time a new frame is ready for processing. We will not go further into the specific code, but just notice that it may be found in the 'camera.h' and 'camera.c' source files (see appendix).

Figure 2.1 shows two raw images of the line, captured by the camera. Later in this chapter, we will shows how the image extracts the line, and calculates the error signal.

(a) Straight line (b) Intersection

Figure 2.1: Raw images captured by the camera

2.2 Identifying and Extracting the Line

Whenever an image has been captured and is available in the memory, it is time to do the image processing and extract the line from the background. This is done using the steps listen below.

- Contrast and brightness is adjusted
- The image is 'sliced' into six horizontal slices
- For each slice, a histogram is calculated, and from that a optimum thresholding value
- Finally each pixel in each slice is marked as LINE if below the threshold value, or marked as FLOOR if above the threshold value

2.2.1 Contrast and brightness

In the following sections when referring to the image data, a simple notation will be used. The input image is given as $f(x, y)$, and the output image is given as $g(x, y)$, where x and y are the coordinates of a given pixel.

The contrast and brightness adjustments are simple point-processing operations, which can be describes with the following equation.

$$g(x, y) = a \cdot f(x, y) + b \tag{2.1}$$

The brightness b changes the overall brightness of the image, thus making all individual pixel more brighter. The contrast adjusted by a simply makes the contrast between the individual pixels bigger.

In the system the possibility of changing the brightness and contrast are available, but at time of writing, the configuration is set to $a = 1$ and $b = 0$, i.e. no changes in brightness or contrast.

2.2.2 Slicing

Due to huge changes in the illumination of the image, it is split into five horizontal slices. For each of these slices a histogram is calculated, and a thresholding value is found. By doing this instead of one thresholding value for the whole image, we obtain a much better thresholding.

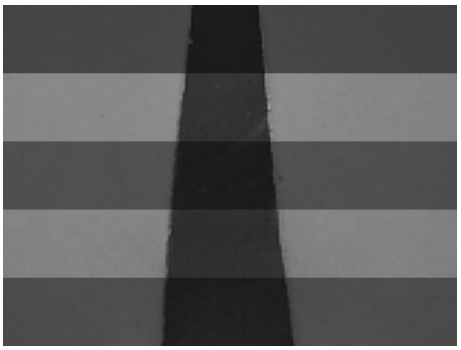


Figure 2.2: Raw image with slices marked

2.2.3 Histogram

Whenever the images are sliced into slices, a histogram for each slice is calculated. A histogram shows the distribution of pixels of different color values. The typical histogram of a gray-scale image shows the color values on the x-axis $[0-255]$, and the percentage of the given color on the y-axis $[0-1]$. The value 0 on the x-axis is the black colors, whereas the value 255 is the white colors. With this in mind, we can conclude that the first hill on the histogram is the line, centered around 50, and the second hill is the floor, centered around 135.

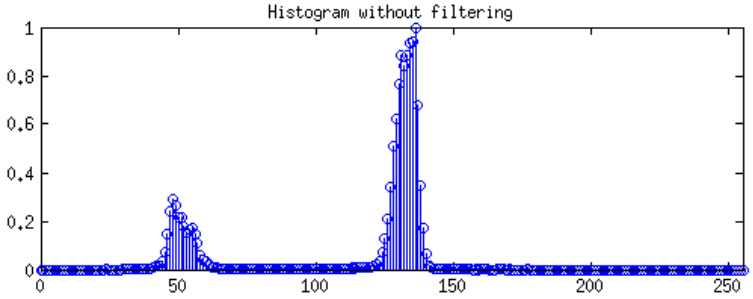


Figure 2.3: Histogram without filtered

Figure 2.3 shows the histogram of a slice from the image in figure 2.2. A lot of noise is visible on the histogram, which makes calculating the correct thresholding

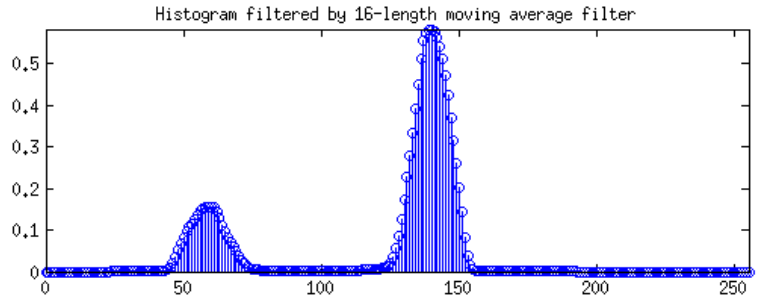


Figure 2.4: Histogram filtered with 16-point average filter

point less accurate. Hence, the histogram is filtered by a 16-point/length moving average filter, which gives the histogram shown in figure 2.4.

The filtered histogram has less spikes, and is much better when applying the Optimum Thresholding algorithm in the next section.

2.2.4 Optimum thresholding

Next, when the pretty averaged histogram is calculated, it is time to calculate the perfect threshold value for segmenting the line from the floor - in other words, finding the value on the x-axis of the histogram where a clear split between the colors of line and colors of the floor is to be found.

The Optimum Thresholding algorithm is described in [**?**], which also provides a reference implementation in C. This implementation is fitted into our existing code base, and used without major modifications.

The algorithms work by iterating through the histogram, finding the first valley. That is, by looking at a single color on the x-axis, where the color to the left has a larger or equal percentage, than the current color, and color to the right has a small percentage. If $h[x]$ is the histogram, then equation 2.2 describes the valley.

$$h[x-1] \geq h[x] < h[x+1] \tag{2.2}$$

Figure 2.5 shows the histogram zoomed in around the first valley after the top of the first hill (the line). Equation 2.2 is satisfied first time at about $x = 115$, which in this case will be the optimum thresholding value.

Listing 2.1 shows part of the implementation of optimum thresholding from image.c. In the while-loop from line 1-9, the histogram is iterated through, and a flag is set when equation 2.2 is fulfilled.

Finally, when the threshold value is calculated a slice, the image data is segmented using that value. The mathematical description of thresholding is given en equation 2.3.

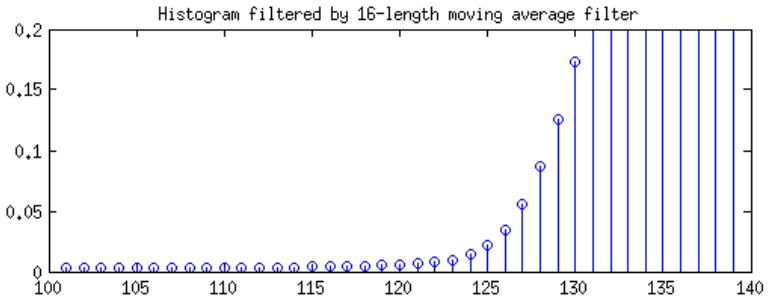


Figure 2.5: Histogram zoomed in before the 'floor' hill

Listing 2.1: Part of the thresholding code

```
1   while (flag == 0 && y < 254)
2   {
3      if (hist[y-1] >= hist[y] && hist[y] < hist[y+1])
4      {
5         flag = 1;
6         thr = y;
7      }
8      y++;
9   }
10  for (j = start; j < end; j++)
11  {
12     buffer[j] = buffer[j] < thr ? LINE : FLOOR;
13  }
```

$$\begin{aligned} &\text{if } f(x,y) \leq T \text{ then } g(x,y) = 0 \text{ (line)} \\ &\text{if } f(x,y) > T \text{ then } g(x,y) = 255 \text{ (floor)} \end{aligned} \tag{2.3}$$

Line 10-13 of listing 2.1 loops through all pixels, and decides if they are line or floor. The images presented first in this chapter, is shown below in figure 2.6 in their processed and segmented versions (all images are actual extract from the robots image processing system).

As we can see on figure 2.6b, it has not been possible to find a optimal thresholding value, probably due to a too low contrast caused by sun light. In the right side of the image, only the edge of the line is marked as line.

2.3 Calculating the Error Signal

Now that the line is segmented from the floor, next step is BLOB (Binary Large OBject) analysis. [?] presents a bunch of techniques for classification of BLOBs, i.e. distinguishing different BLOBs from each other. Each of the different characteristics a BLOB may be represented by, is denoted a feature. Common features are area, bounding box, compactness, circularity. Another one is called

(a) Straight line (b) Intersection

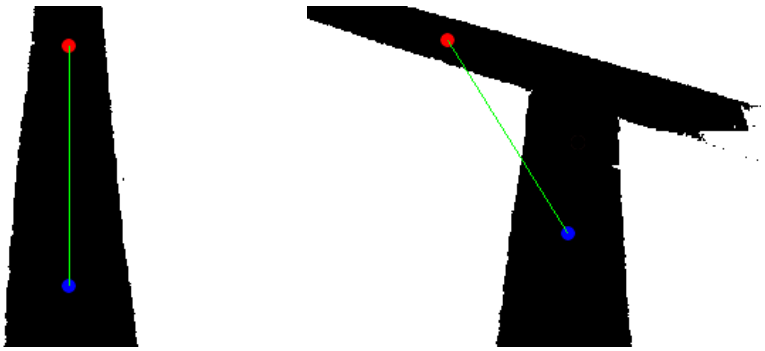
Figure 2.6: The images after being processed

'Center of mass', which is the physical location on an object, where you should place a finger in order to balance the object. In terms of image processing, it is the average x- and y-positions of the binary object (the line). It is defined as a point whose x-point is calculated by summing the x-coordinates of all pixels in the BLOB and dividing by the total number of pixels. Likewise for the y-value. The 'center of mass' feature is used to calculate the error signal the system.

In mathematical terms, the center of mass (x_c, y_c) is defined as given in equation 2.4.

$$x_c = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad y_c = \frac{1}{N}\sum_{i=1}^{N} y_i \tag{2.4}$$

Using the equation for center of mass, two error points are calculated for each image. One for the upper part of the image, which gives an indication of the error in the 'future', and an error point in the lower part of the image, which gives the actual present error. Figure 2.7 shows the two images annotated with error points. The green line indicates the optimal direction of motion.



(a) Straight line (b) Intersection

Figure 2.7: The images with two center of masses calculated

Once the points has been calculated, the horizontal error signal (sideways error from the line) is calculated by subtracting the x-position of each point, from the x-coordinate at the center of the image. Equation 2.5 gives a mathematical expression of the error signal. Hence, when the robot is to the left of the line (line is in the right side of the image), the error negative, thus when the robot is to the right of the line (line is in the left side of the image), the error is positive.

$$e = x_{center} - x_{point} \tag{2.5}$$

CHAPTER 3

Controlling The Motors

3.1 The DC-motor

The DC-motor is an essential part of the robot. The motors used in this project are the Faulhaber 2233012S which is a 12 volt armature motor. To succeed the tasks given in the assignment is it very important to gain control over the motor by understanding the datasheet that describes the motor. The motors dose not only supply the robot with driving force but is also the steering part of the robot.

3.1.1 Gears

To give the robot more torque a gear system has been implemented in the motors. The cost of high torque through a gear-system is a reduction in rotational speed. The gearing factor that is used is $\frac{1}{17.2}$ which means that the output shaft only turns one revolution every times the motor itself have turned 17.2 revolutions. The gear system do have some disadvantages by giving the motor nonlinearity like backlash and deadlocks.

3.1.2 Transfer function for DC-motor

In assignment 1 from the control theory course we calculated a transfer function for at DC-motor. By using this knowledge and theory we are able to calculate a transfer function for the DC-motor used in this project. The transfer function for the DC-motor is needed for making simulations of the hole system later on.
All motor data is fund in the datasheet and the weight of the robot is fund by weighting the robot on a digital precision scale and is fund to be 1400 g or 1.4 kg.

The general transfer function:

$$G(s) = \frac{\theta_m(s)}{E_a(s)} \Rightarrow \frac{\frac{k_t}{R_a J_a}}{s + \frac{1}{J_a}(D_a + \frac{K_b K_t}{R_a})} \tag{3.1}$$

The transfer function for the robot:

$$G(s) = \frac{\omega(s)}{E_a(s)} = \frac{\frac{1633}{23.43}}{s\frac{1}{23.43} + 1} = \frac{69.69}{0.04268 + 1} \tag{3.2}$$

Where 0.04268 is the time constant.
The step respons on the DC-motor looks like this:

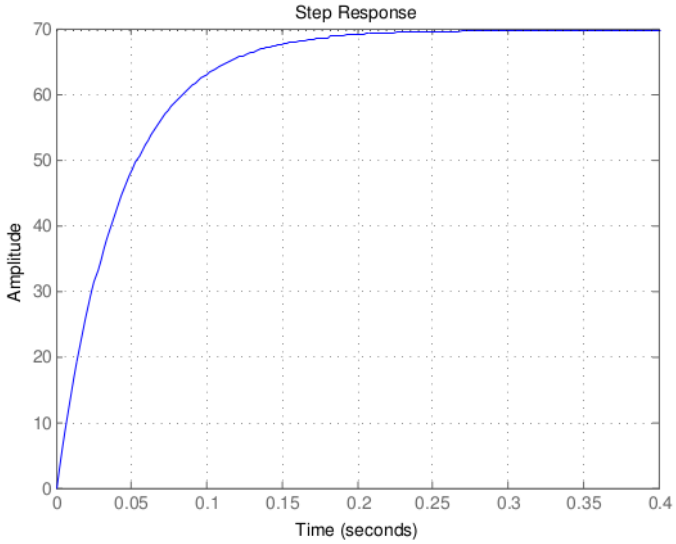


Figure 3.1: Step respons for DC-motor

3.2 Microcontroller

The MCU (Microcontroller) is the first part of the link between the sensor input and the DC-motor. In the MCU a translation is made so that the DC-motor can be controlled using simple I2C (Two wire communication) commands from the Raspberry pi. A controller is also implemented in the MCU so that the robot is able to run straight without any further instructions from the Raspberry pi.

The MCU decodes the command form the Raspberry pi and then extracts informations about direction and speed. These informations is then send down to a H-bridge and ends as Voltage in the DC-motor.

3.2.1 Atmel's XMEGA series

The MUC used in the robot is an Atmel ATxmega192c3 a 8bit low-power microcontroller with 192kbyte flash memory running a clock frequency at 32 MHz. The choice of this MCU is based on the its speed performance and its high level of functionality features. The first MCU used in this project was a Atmel Atmega 32 but after a brief discussion in the group and with the project supervisor we decided to change the MCU to a more modern model. We previously worked with the Atmega32 and wanted to explore more recent technologies and the opportunity to work with the SMD (surface mounted device) method.

3.2.2 Programming and tools

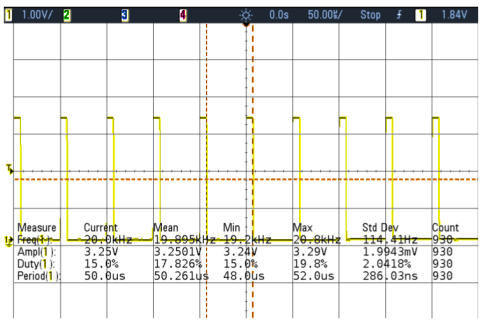
The MUC is programed using Atmel studio 6.1 and the program language used is C. The ASF (Atmel Software Framework, **[?]**) libraries has be used to a great extent.
The source code for the MCU can be found in the appendix.

3.3 H-bridge as driver

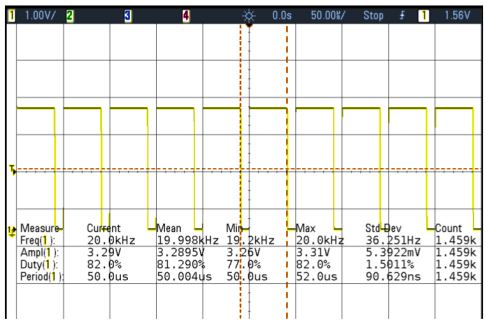
The H-bridge (TI293DNE) is a well known easy to use device. The devise have been used for robotics for many years and is well documented.
The H-bridge receive three lines per motor (six in total) from the MCU the first line controls the speed with a PWM signal and the to other lines control the rotational direction of the motor. The output from the H-bridge is connected directly to the two DC-motors.

3.3.1 PWM - Pulse Width Modulation

PWM (Pules-width modulation) is a frequently used method in electronics. The signals intensity it controlled by changing the duty cycle for a square wave signal with a predefined frequency. The controllable range is from 0% to 100% The frequency of the square has been chosen to be 20kHz so that is it over the human hearing range.
The figures below shows how a 25% and a 50% PWM looks between the MCU and the H-bridge. The duty cycle output of the MCU is about 15 % and 82% The reason for that is that the controller in the MCU manipulates the PWM, more about this in the Control System chapter.



(a) PWM with input at 25%



(b) PWM with input at 50%

3.3.2 Direction control

To control the rotational direction of the DC-motor the H-bridge receives two lines A1 and A2 from the MCU.

When line A1 is high and line A2 is low the corresponding switches S1 and S4 is closed so that the current is running in one direction. when A1 is low and A2 is high the current is running the other way and the motor it turning the other way around.

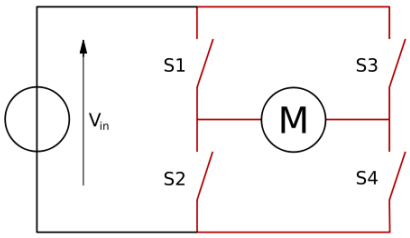


Figure 3.2: H-bridge

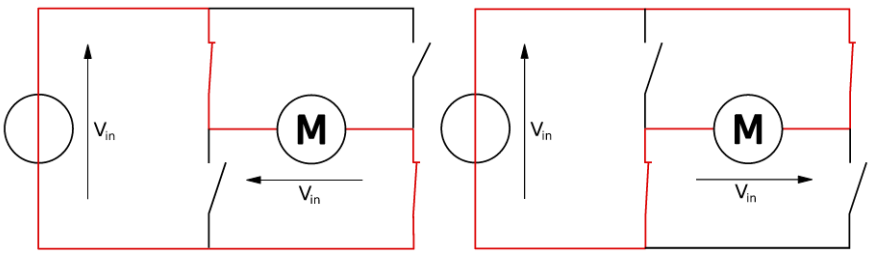


Figure 3.3: H-bridge in the two states

In figure 3.6 is a truth table that describes the functionality of the H-bridge. The EN pin is the PWM input.

EN	1A	2A	FUNCTION
H	L	H	Turn right
H	H	L	Turn left
H	L	L	Fast motor stop
H	H	H	Fast motor stop
L	X	X	Fast motor stop

L = low, H = high, X = don't care

Figure 3.4: H-bridge truth table

3.4 Encoders

The DC-motor includes two encoder outputs which is used to determine position, velocity and direction of the DC-motor. Each encoder outputs 15 pulses per round and after the gears is the resolution $15 \cdot 17.2 = 258$ pulses per round. The two encoder signals is phase shifted at about 90° and that gives us the opportunity to determine the rotational direction of the motor by looking at one encoder signal and then check if the other one it high or low. This method is illustrated below in figure 3.7.

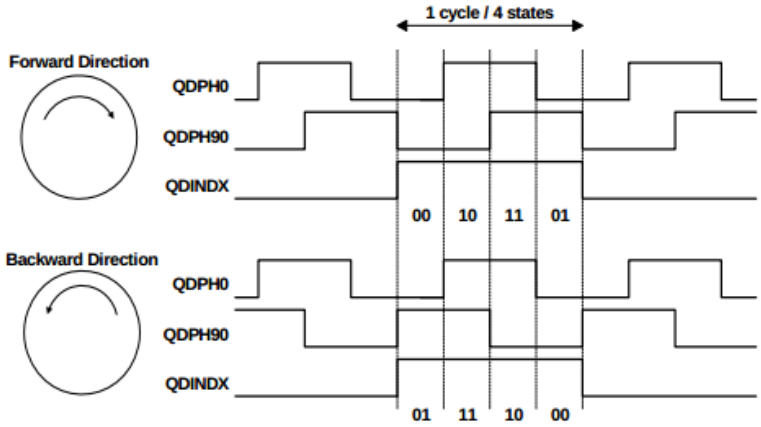


Figure 3.5: Encoder illustration (from Atmel's Application Note on Quadrature Decoders, [**?**])

Figure 3.8 shows the actual encoder output from the robot. The phase is about 100° between the two encoders. The velocity is found by sampling the encoder pulses over a time period of 10ms, by counting the pulses during this time and multiply it with a constant the velocity of the motor is given. Also the position between the two motors can be find by using two encoders on each motor. First we look at one motor encoder count and then match this up the other motor so that the encoder count from the to motors fits, then we know the exact position of the robot. We use these method in the position controller.

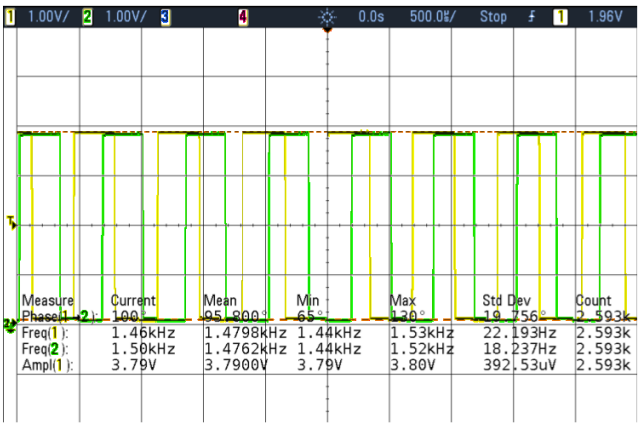


Figure 3.6: Encoder output from Motor to MCU

3.5 Feedback and Control Loop

The control loop containing the motor,MCU and encoder can be described as a unity feedback loop.

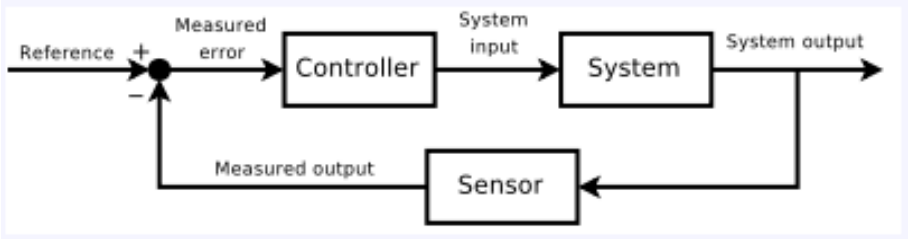


Figure 3.7: Unity feedback loop

The MCU have three different stats with three different controllers so that it can adapt to the given situation during the track. The three states are:

- **Straight**
  This state is designed to make the robot go straight without any corrections from the Raspberry Pi. This it done by given both motors a reference speed and then aligning them after the outputs from the encoders by adjusting the speed to each motor.
- **Position**
  This state is used when the robot has to do precision maneuvers.
- **Speed**
  This state is used when the robot follows the line by adjusting after instruction from the Raspberry Pi.

The selection of which current stat the MCU should be in is determinant by the raspberry pi and is set through the I^2C bus. The different controllers from the three states can be found in the control system chapter.

CHAPTER 4

Measuring distance to wall

4.1 IR sensor

The IR sensors that is used is the SHARP 2d120x which is a nonlinear devise. The IR sensor works by sending a infrared beam out, the beam then returns back to the device after hitting a surface or a obstacle, the angle that the beam returns with determines the distance.

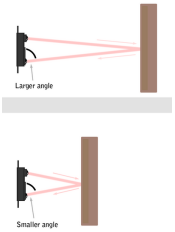


Figure 4.1: Distance IR sensor

4.2 Converting using an ADC

The IR sensor returns a voltage value that is converted in to a unsigned binary one byte number in the MCU's ADC (Analog to Digital Converter) but because of the nonlinearity in the device a equation is made by plotting measurements from 1cm to 50cm. A best fitting trend line it made and from that a equation is used with the output from the ADC to give the distance in cm.

The equations is then fund to be:

$$f(x) = 1543.68x^{-1.05} \tag{4.1}$$

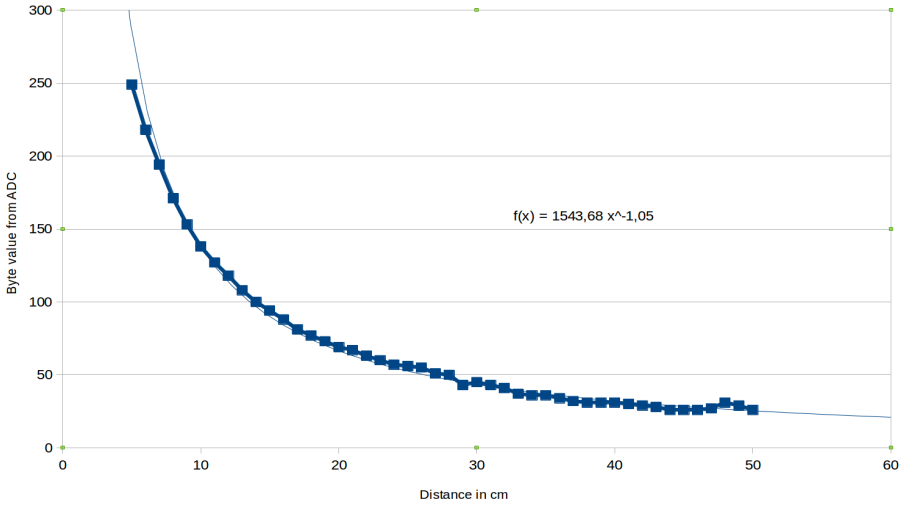


Figure 4.2: Distance IR sensor

4.3 Practical use of IR sensors

On the track is a variety of different obstacles and some of them require informations about distance. To solve these tasks the robot needs a devise to measuring distance.

The two obstacle where the IR sensors is used:

- **Obstacle 1**
  Go to the wall stop 20cm from the wall turn and return to the line.

- **Obstacle 2**
  Go around the corner with a distance of 20cm to the wall and stop at the cross marked on the floor.

There is a total of three IR sensor on board the robot, one at the front just underneath the PS3 camera and two on the left side of the robot. The front sensor it used to solve both obstacle 1 and 2 and outputs a distance to any object in front of the robot. The two side mounted sensor it used to solve obstacle 2 where the robot shout ”stick” to the wall by a given distance.

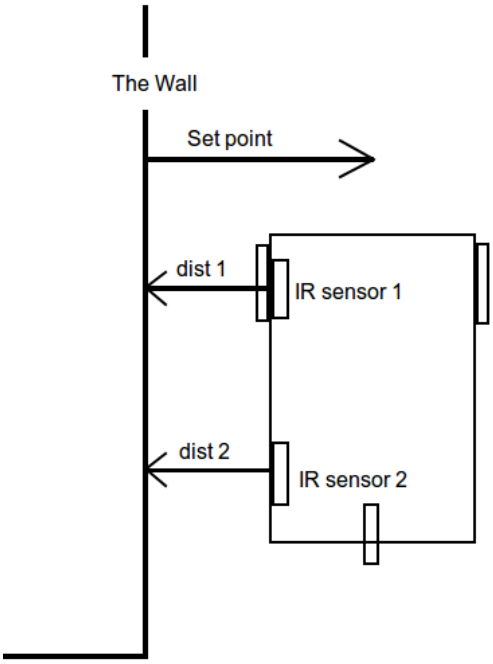


Figure 4.3: Stick to the wall

CHAPTER 5

Communication between devices

Communication between the various device in the robot is a vital issue. Without proper communications the robot will not be able to function after the intention. In this project the communication protocol has been chosen to be the I^2C protocol because is it more flexible then for example the SPI protocol.

5.1 I^2C

The I^2C (Inter-Integrated Circuit) protocol is a very used communication method in electronic devices. It is a two way multi-master communications protocol where there is one interchangeable master and up to 112 slave devises. The master device sets the clock (SCL) and sends out a data stream on the data line (SDA) starting with a byte containing a 7 bit address for the decided slave device, and a read/write bit the determines what action the master device wants do execute. After the address byte follows data transfered one byte at the time until there is no more data to transfer. A pull up resistor is added to each of the to lines to ensure that the lines a pulled high when the bus is not i use and between pulse during transmitting.

Figure 5.1: Address byte used by the I^2C protocol

5.2 Implementation

In this project the Raspberry Pi is the I^2C master because it is within this devise all the processing in produced. The Raspberry Pi is running the I^2C bus at 100

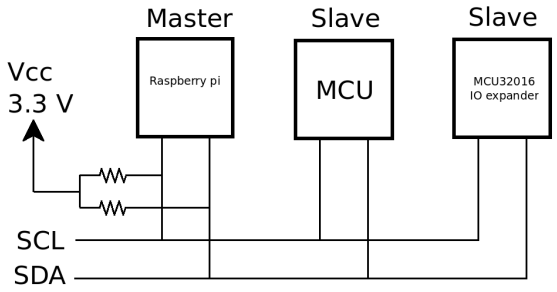


Figure 5.2: Block diagram of the I^2C bus

kHz and is equipped with two 1.8 kΩ pull up resistors.
To make the communication procedure easier, a command system has been implemented in all the devices attached to the I^2C bus (not the IO expander because this device it a preprogrammed I^2C device). The system makes the first data byte in the data stream to a command(CMD) and this command then calls different functions that deals with the rest of the data in the stream.

5.2.1 I^2C Commands

The I^2C commands are:

- **CMD 0x05**: Set Led
  The Raspberry Pi writes one byte that controls the three LEDs on the motor controller board.

- **CMD 0x10**: Get Speed
  read two bytes of data that contains the speed for each motor calculated in the MCU.

- **CMD 0x20**: Set Speed
  Raspberry Pi Writes two bytes to set the motor speed by changing the PWM value.

- **CMD 0x30**: Set Dir
  Raspberry Pi writes two byte that sets the detection on the motors by changing the four direction bits in the MCU.

- **CMD 0x50**: Set State
  Raspberry Pi writes one bite that changes the states in the MCU between Speed, Position and Straight.

- **CMD 0x60**: Set Target Position
  Raspberry Pi writes four bytes that contains the target position.
- **CMD 0x70**: Precision Stop
  Sets the position and target variables to zero.
- **CMD 0x80**: Get Dist
  Raspberry Pi reads three bytes that contains the distance from the three IR distance sensors.
- **CMD 0xA0**: Is Stable.
  Returns if the position is the same as the target position.

CHAPTER 6

Control Systems

In this section, the control systems of the robot will be discussed in more detail.

6.1 Control Systems in Eyebot

The system consist of several controllers, with some of them implemented in the Raspberry Pi (for following the line and the wall), and the other ones implemented in the MCU for specific tasks (driving straight forward, position, speed).

6.1.1 Controllers in the MCU

Three special-purpose controller are implemented in the MCU, controlled by an internal state machine (explained in chapter 3).

Speed (velocity) controller

The initial state of the MCU is, to act as a speed controller given some reference. This controller is simple P-controller, where the error signal is calculated using the sampled speed from the encoders. Figure 6.1 shows a simplistic model of the speed control feedback loop.

Position controller

When a specific position is needed (counted as pulses from the encoders), another controller is used. This time the error signal is calculated as the difference between the actual position, and a given reference. Figure 6.2 shows a simplistic model of the position control feedback loop.

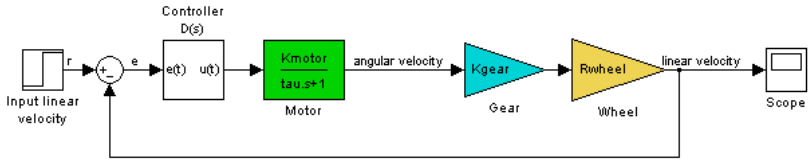


Figure 6.1: Block diagram of velocity control

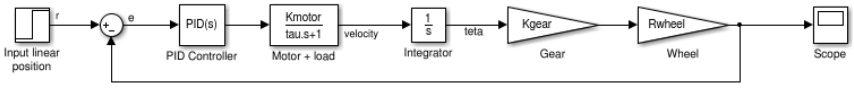


Figure 6.2: Block diagram of position control

Speed-position controller (straight forward)

The last case for the motor controller, is to use a combination of the two above. In this controller, the error is a speed for each motor, but the error signal is calculated as both the difference between the reference speed and the actual speed, but also the difference in position between the two wheels. This ensured that both motors are driving at the same speed, hence a motor slows down or speeds up, to keep up with the other motor.

This controller is implemented as a more advanced PI-controller, to ensure that the steady-state error approaches zero.

6.1.2 Controllers in the Raspberry Pi

Following the line

When following the line, a PID controller implemented in the Raspberry Pi, uses the camera as error signal, and calculates the necessary compensation. This compensation is a speed for the left and right motor, which are sent to the MCU motor controller over I2C. In the MCU a simple P-controller uses the speed received from the Raspberry Pi as a reference, and together with its encoder measurements, a speed for each motor is calculated.

Following the wall

Whenever the wall is to be followed, a special controller implemented in the Raspberry Pi is used. The error signal is calculated using the two distance sensors mounted on the left side of the robot.

$$e = (d_s - d_1) + (d_2 - d_1) \tag{6.1}$$

Equation 6.1 states that, the error is the difference between the front sensor and the desired reference (set point), plus the difference between the two sensors (d_1 and d_2). The first terms gives the an error if the robot is not the correct distance from the wall, where as the second term gives an error if the robot is not perfectly parallel with the wall.

6.2 Design of controllers

6.2.1 PID

PID (proportional-integral-derivative) controller, also known as classical three-term controller, is a closed-loop feedback controller/compensator, widely used in the industry. An error signal is calculated as the difference between a desired 'setpoint', and an actual 'process value'. Figure **??** illustrates a unity-feedback system with a PID-controller.

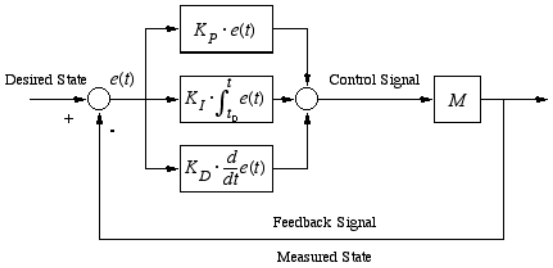


Figure 6.3: PID Controller in feedback a feedback loop (from cs.brown.edu)

The figure shows each term of the PID-controller in time-domain. More often, it is more convenient to describe controllers and systems in frequency domain. The usual forms of the transfer function for a PID controller is given in equation 6.2.

$$D(s) = K_P + \frac{K_I}{s} + K_D s = \frac{K_D s^2 + K_P s + K_I}{s} = K_P(1 + \frac{1}{T_I s} + T_D s) \tag{6.2}$$

In chapter **??**, we will look more into the actual parameters chosen for the controllers, and look at simulations and compare them against reality.

Listing 6.1: Implementation of discrete PID controller

```
1  void pid_ctrl(pid_data_t * pid, int process_value)
2  {
3     float p, i, d;
4     float temp, cv;
5     int error;
6
7     error = pid->set_point - process_value;
8
9     // Calculate P-term
10    p = pid->P * error;
11
12    // Calculate I-term
13    pid->sum_error = pid->sum_error + error;
14    if (pid->sum_error > pid->max_sum_error || pid->sum_error < -pid->max_sum_error)
15    {
16       pid->sum_error = 0;
17    }
18    i = pid->I * pid->sum_error;
19
20    // Calculate D-term
21    d = pid->D * (pid->last_process_value - process_value);
22    pid->last_process_value = process_value;
23
24    // Total correction (control variable)
25    cv = p + i + d;
26    if (pid->set_cv != NULL)
27    {
28       pid->set_cv(pid, cv);
29    }
30 }
```

6.3 Implementation of discrete controller

All controllers in the system are implemented in C using the reference implementation provided by Atmel, in their Application Note on Discrete PID controllers, [?], which again is based on theory by [?]. Another implementation is suggested in [?], which is sligthy more complicated due to an additional time constant.

The argument 'pid_data_t' is an abstract data type, with the fields needed by each calculation. It contains the P, I, D factors, the sum of error (for I-term), the last process value (for D-term), plus some other fields.

6.4 Tuning of PID controllers

6.4.1 Manual tuning

The simplest of the known tuning methods, is called manual tuning [?]. By starting with $K_P = 1$, $K_D = 0$, and $K_I = 0$, and then using the table below to tune the system. This method has been heavily used during this project.

From table 6.1 we can conclude the following:

Table 6.1: Tuning parameters

Response	Rise Time	Overshoot	Settling Time	S-S Error
K_P	Decrease	Increase	No change	Decrease
K_I	Decrease	Increase	Increase	Eliminate
K_D	No change	Decrease	Decrease	No change

Table 6.2: Ziegler-Nichols - Second method

PID Type	K_P	T_i	T_d
P	$0.5K_{cr}$	∞	0
PI	$0.45K_{cr}$	$P_{cr}/1.2$	0
PID	$0.6K_{cr}$	$P_{cr}/2$	$P_{cr}/8$

- Use K_P to increase rise time.
- Use K_D to reduce the overshoot and settling time.
- Use K_I to eliminate the steady-state error.

6.4.2 Ziegler-Nichols

Another important method for tuning PID-controllers is the Ziegler-Nichols method, which in the industry is accepted as standard [**?**]. Ziegler and Nichols actually came up with two methods, but only their second method is given here.

Tuning using Ziegler-Nichols second method requires the following steps (from [**?**]):

- Reduce the integrator and derivative gains to 0
- Increase K_P to some critical value K_{cr} at which sustained oscillations occur.
- Note the value of K_{cr}, and the corresponding period of sustained oscillations, P_{cr}

The gains are now to be found using table 6.2.

CHAPTER 7

Following The Track

This chapter goes into detail about the steps taken to complete the track. Most of this chapter may be seen as pseudo code, of what is really implemented in the robot.

7.1 Requirements for the track

Here follows a list of movements, the robots must be able to do, to complete the track.

- Drive straight forward - solved by an aggressive velocity PI-controller.
- Rotate, by driving a precise distance at one of the wheels - solved by using a position PI-controller.
- Detect the distance in front, to stop 20 cm from the wall.
- Detect the distance to the wall on the left side of the robot.
- Follow/stick to the wall using the two distance sensors mounted on the left side of the robot.

7.2 State machine

An mentioned earlier in the report, a central state machine implemented in the Raspberry Pi, is responsible for keeping track of the overall state when completing the track. Below is a detailed explanation of each state in the state machine. Chapter 3 discussed the internal states used by the motor controller, hence it is not discussed here.

- **START**
  Drive straight forward, next state set to GOTO LINE
- **GOTO LINE**
  Continue forward. If line is visible, rotate 45 degrees to left, and begin following the line by going into FOLLOW LINE state.
- **FOLLOW LINE**
  Follow the line until a crossing is captured. Then brake, and go into GOTO WALL mode.
- **GOTO WALL**
  Rotate 90 degrees to the left. Drive forward, and stop 20 cm from the wall. Rotate 135 degrees. Drive forward, and wait a while for the camera to become stable after turning against the sun light. When the line is visible, go to FOLLOW LINE AFTER WALL state

- **FOLLOW LINE AFTER WALL**
  Follow the line, this time a bit faster. When an intersection is visible, go to FOLLOW LINE SPEEDY state.
- **FOLLOW LINE SPEEDY**
  Follow the line at fastest possible speed. When the end of line is seen, brake down and go to END OF LINE state.
- **END OF LINE**
  Wait for three seconds, and then go to STICK TO WALL state.
- **STICK TO WALL**
  Drive straight forward, and stop 20 cm from the wall. Rotate 90 degrees to the right, and begin driving forward. Go to STRAIGHT UNTIL WALL DISAPPEARS state.
- **STRAIGHT UNTIL WALL DISAPPEARS**
  Continue forward until the wall on the left side, is farther away than 20 cm. Brake down and rotate 90 degrees to the left. Continue forward for a while and then go to STRAIGHT UNTIL WALL DISAPPEARS 2 state.
- **STRAIGHT UNTIL WALL DISAPPEARS 2**
  Continue forward until the wall on the left side, is farther away than 20 cm. Brake down and rotate 90 degrees to the left. Continue forward for a while and then go to FOLLOW WALL 1 state.

- **FOLLOW WALL 1**
  Follow the wall on the left side. Brake down when the wall in front is less than 20 cm away. Brake down and rotate 90 degrees to the right. Go to FOLLOW WALL 2 state.

- **FOLLOW WALL 2**
  Follow the wall on the left side. When the wall in front is less than 20 cm away, stop and go to TRACK COMPLETE state.

- **TRACK COMPLETE**
  Track is completed! Flash the LEDs and beep the buzzer!

CHAPTER 8

Simulation and Testing

8.1 SIMULINK

Appendices