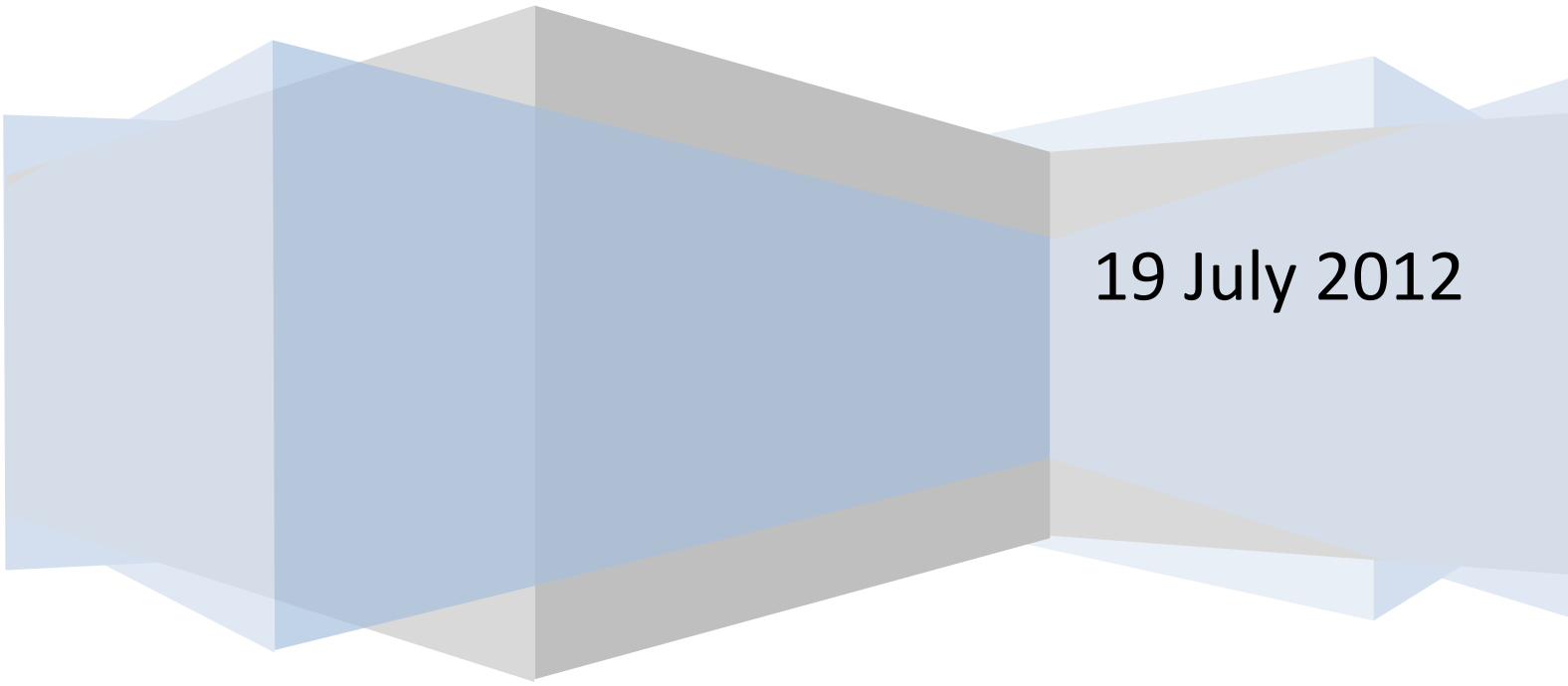
University of Canterbury

ENEL300

"A Line Following Robot"

Matt Kokshoorn, Nick Bingham, Gavin Austin & Nick Clark



19 July 2012

Abstract

This report details the design and development of the ENEL300 Second Semester group project. This project required the implementation of a line-following robot. This robot was realised through the use of a single sensor, an Atmega8 Microcontroller, a motor, a motor driver and other analogue peripherals all combined on to a PCB (printed circuit-board). The microcontroller was used to control the movement of the robot, allowing the performance to be enhanced through the use of an efficient algorithm. The other components were all aspects of the basic hardware required for either driving the motor or supplying power or information to the microprocessor. The final design was a success as the robot was able to accurately follow a printed line and stop at the desired end-point.

Contents

1. *Introduction* 4
2. *Project Management* 6
3. *Design Description* 8
   - 3.1. *PCB Schematic Design* 8
   - 3.2. *PCB Layout* 9
   - 3.3. *PCB Construction* 11
   - 3.4. *Gearbox Assembly* 11
   - 3.5. *Mounting External Peripherals and Lines* 11
   - 3.6. *Configuring the ISP* 14
   - 3.7. *Software Design and Driver Libraries* 14
     - 3.7.1. *LEDs* 16
     - 3.7.2. *ADC* 16
     - 3.7.3. *PWM* 16
     - 3.7.4. *Motor* 17
   - 3.8. *Maze Solving Algorithms* 17
     - 3.8.1. *Counter-Turn Stability* 17
     - 3.8.2. *Turn Curvature and Speed Variations* 18
     - 3.8.3. *Oscillation Compensation* 18
     - 3.8.4. *Cornering for Turns* 19
     - 3.8.5. *Breaking Out of Loops* 19
4. *Design Problems and Solutions* 21
5. *Design Results* 24
6. *Discussion* 27
7. *Conclusion* 30
8. Bibliography 31
9. *Appendix* 32

User Manual 37

1. Introduction

As the field of robotics advances, the technology is moving out of laboratories and high-tech manufacturing companies into mainstream industries and warehouses. Robotic arms, also known as industrial robots, are the most common and are generally found anchored to one spot in the factory. They are used for welding, painting, packaging and many other applications [1]. These are not the only robots in use, however, as a second field of mobile robots are now being used for moving goods within warehouses, such as Fonterra's automated forklifts [2]. This technology allows the warehouses to operate more efficiently; the robots can work around the clock and can be linked into the warehouse's stock-tracking software to keep the company's stock records constantly up to date. One way for the robots to navigate the warehouses is by following a network of lines painted on the floor, a simple and cost effective solution that can keep the robot on track to many locations within each warehouse.

Figure 1 - Fonterra's Automated Line Following Forklifts

The aim of this project was to create a robot able to find its way through a route of black lines on white paper by following the lines in a manner similar to the industrial line-following robots. The robot was to be completely designed and fabricated from scratch, requiring development of a range of aspects from basic hardware through to abstracted programming to solve the maze. The goal for the robot was, first and foremost, to finish the maze but also to do this in as little time as possible and with as few sensors as possible. The robot also had to be implemented within a restricted budget and limited time frame. These considerations were designed to mimic real world product development constraints where performance and cost are crucial in determining a product's success or failure.

This assignment was completed in teams of four and required the full use of the wide range of individual skills and knowledge available for a successful product to eventuate. Project management and teamwork were therefore imperative to the success of the project. All team

members were allocated tasks which utilised their personal strengths and skillsets and all tasks were completed to schedule to ensure the success of the project within the time constraints.

2. Project Management

Immediately after receiving the assignment a group meeting was held in which the team strategies, methodologies, and allocation of work were decided upon. This was the main planning session for the group, where realistic deadlines for the major milestones within the project were set and appropriate work was allocated to the team members to match their experience, knowledge base and interests. This allowed for the most productive use of each team member's time. The basis of the Gantt chart outlining the timeline of the group's activities, was discussed and formed to keep the group on track and to maintain a constant workload across the duration of the assignment. This initial planning session proved invaluable as it set the standards and the direction of the group as well as confirming that all members were comfortable with the development process and allocation of tasks.

At this initial meeting it was also decided that regular team meetings would be held to maintain high levels of communication within the team. These meetings were held in an informal social setting. Such meetings were possible due to the similar time restrictions on members and so were employed in preference to formal meetings to reduce pressure. This also meant that these meetings could be relaxed with undefined time restrictions, increasing group discussion and creativity. Group cohesiveness was also increased by working in a stress free environment, allowing team members to work more effectively with one another and increasing the group's overall productivity. The group meetings ranged from online conversations within a closed forum to meetings in person. The use of an online group forum allowed all members to actively communicate at any time, allowing all members to be kept up to date on progress as it eventuated. This eroded the need for formal meetings, helping to promote informal discussions; only two other formal meetings were necessary across the duration of the assignment. High levels of communication within the team meant that as soon as a task ended the following dependent stage of the development could begin. This was of high importance when dealing with tasks on the critical path of the assignment, as a delay in switchover of tasks had potential to delay the end date of the project, which could in turn jeopardise the entire assignment.

When allocating group members to tasks and planning the project schedule the focus was on designing an efficient critical path. It was decided that the project could not be reasonably expected to have worked carried out on it by any team members across study week or the exam period; it was in the interest of every team member to have the opportunity to focus on the end of semester exams. When initially scheduling the project this was taken into consideration, so that an adequate quantity of work would be completed before the examinations and the break. Other issues arose with the availability of team members over the holiday break. The critical path therefore had to be carefully analysed to ensure that each member's contribution to the project would not be impaired by their holiday plans. The sections of the project which needed to be performed over the holidays were allocated to team members who remained in

Christchurch. This resulted in minimum disruption to the development of the robot. Also considered when determining the critical path was that time should be set aside in case tasks required manoeuvring. This meant that the planned finish date of the project was before the actual due date so unplanned delays could be accommodated, which were considered likely and did, in fact, occur. Since no one in the team had previous experience with this type of project, it was difficult to gauge the expected times each task would take. Worst case times were used in the Gantt chart to ensure that problems at each stage of the project would not disrupt the critical path. On top of this, if anything went wrong in such a way that the worst case time was exceeded it was still unlikely that the development time would go past the final assignment due date as there was sufficient time set aside for such delays.

Since the all members of the team had successfully worked together and were well acquainted the team had high cohesion from square one. This proved beneficial as all of the team members were able to honestly critique each other's work, allowing for a higher quality end product.

3. Design Description

The goal of the project was to create a robot capable of following a line through a maze using up to three sensors. Our team decided to implement the robot using only one sensor as one of the goals of the project was to minimise the number of sensors used and the group believed that realisation of this was possible. Initially this was planned to be accomplished by following one edge of the maze throughout the course, with an exception to this rule occurring when the robot is in a loop whereupon the other edge would be briefly followed. This method is known as the Pledge algorithm [3].To deal with the exception of starting in a loop the robot counts the number of turns it has done in a row in the same direction. If this passes a certain number the robot knows it is in a loop so it crosses the line and follows the other edge. This method is simple and guaranteed to solve the maze, making it a good method considering the limited capabilities of the robot, both in processing power and in sensor capabilities.

The design process was split into two separate components: the physical hardware design and the software design. The hardware was then split into the electronic schematic design and the PCB layout, and hardware construction and testing. Software was similarly split into low-level drivers and high level algorithm implementation.

3.1. PCB Schematic Design

The PCB schematic was designed using the PCB artwork creator, Altium Summer Release 09. To begin with, design was carried out to cater for the fundamental functionality of the robot. This included the header for the voltage input, the Atmega8 (and surrounding required circuitry), the parallel port configuration, the headers to the sensors, the motor driver, and the output headers to the motors. Each functional group's circuitry was determined from their corresponding data sheets.

The sensors (QRD113 reflective object sensors) required both an input and an output. The inputs, the regulated voltage, needed to be supplied to the infrared emitter side of the optocouplers with a 1kΩ current limiting resistor and the outputs, the signals from the sensors, were required to be fed back into the ADC inputs of Atmega8. The output voltage was produced from feeding the regulated supply into a voltage divider consisting of a 1kΩ resistor and the optocoupler's phototransistor.

The circuitry surrounding the L293 Quadruple Half-H motor drivers required one header per driver for the output to the motor. Control over these drivers was exerted through inputs from selected logical outputs on the Atmega8.

The next step was aimed at improving the circuit's functionality by adding additional features. The first of these was the addition of the MC7805 Voltage Regulator. This voltage regulator

takes an input (ideally of 7-20V) and outputs 5V at maximum current of 1A [4]. The voltage regulator should increase reliability by providing an accurate and constant voltage supply to the microcontroller and the sensors. This effect was further enhanced by the placement of a 1uF capacitor to ground before the voltage regulator to remove noise and a 100uF capacitor after to ensure constant supply. It was decided that the regulated voltage should be used to supply only the 'voltage critical' components such as the microcontroller and the sensors whereas the motor drivers could be supplied directly from the batteries in order to reduce the power levels flowing through and being dissipated in part by the regulator.

The final addition was the debugging LEDs. The ability to turn LEDs on/off would provide simple but effective means of debugging the software. This was to be implemented using three basic MOSFET configurations with each gate connected to the logical outputs of PB0, PB6 and PB7 on the Atmega8.

The final design can be seen below in Figure 2.

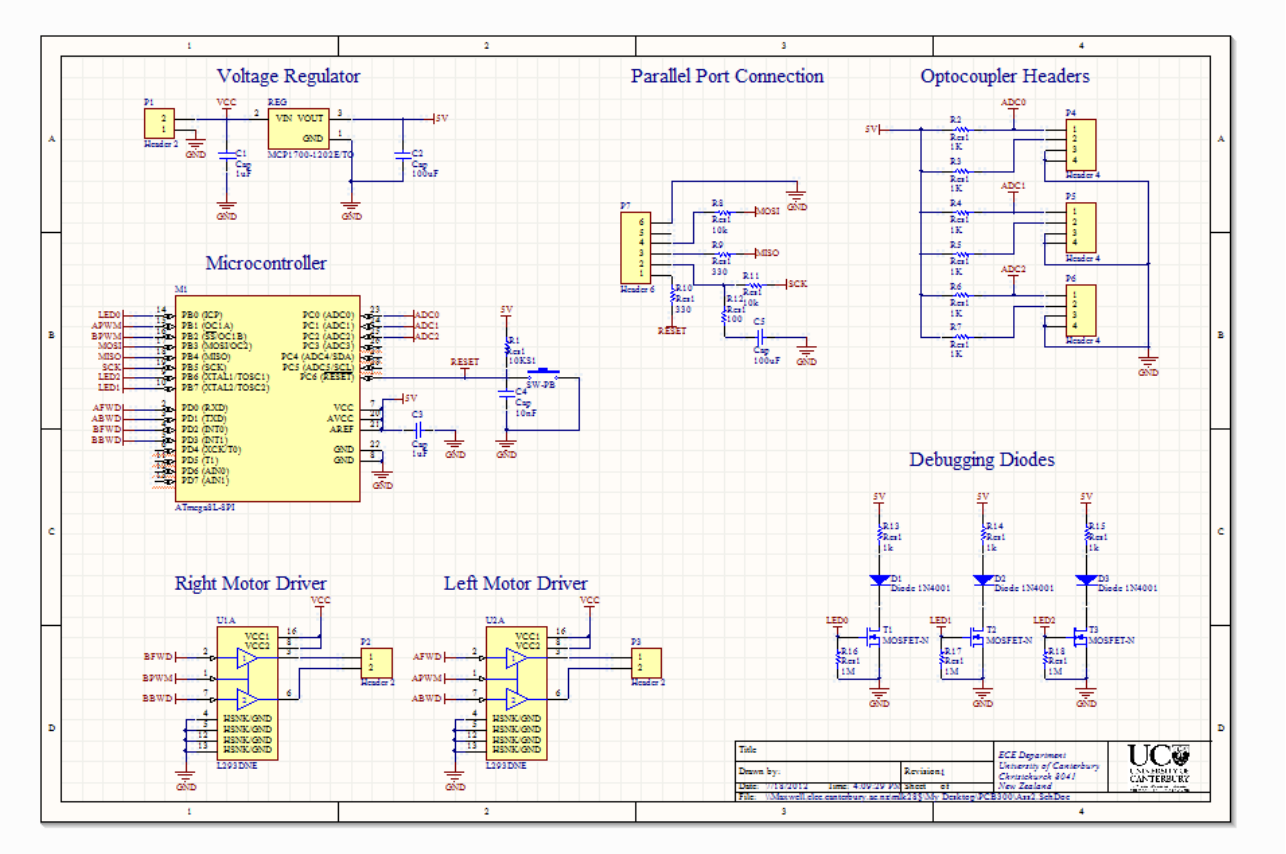


Figure 2 - Electrical schematic diagram for robot electronics.

3.2. PCB Layout

When laying the components out on the PCB board the first consideration was on the ground

and power tracks. This was because all components will in some way be connected to these two tracks, whether directly or through other components. The length of these two power tracks was minimised as much as possible while avoiding sharp corners and large loops formed by the tracks. An emphasis was placed on grouping components with similar functionality or which acted as a group together. This not only made it far easier to visually inspect the PCB and determine the function of each component, but also allowed for easier debugging. It also meant that the analogue and digital circuitry was physically separated on the board. This was an important consideration in the minimisation of noise on the board. A digital and analogue signal interfering with each other is one of the worst forms of noise generation possible, meaning that their physical separation is highly advantageous in the accuracy of readings on the board. The tracks were laid out using standard PCB design rules, adhering to standard sizes and rules, such as eliminating all 90 degree corners within the tracks. While it was not possible to create the PCB without the use of any jumpers, their use was minimised as much as possible, keeping them small and out of the way of major components wherever possible.

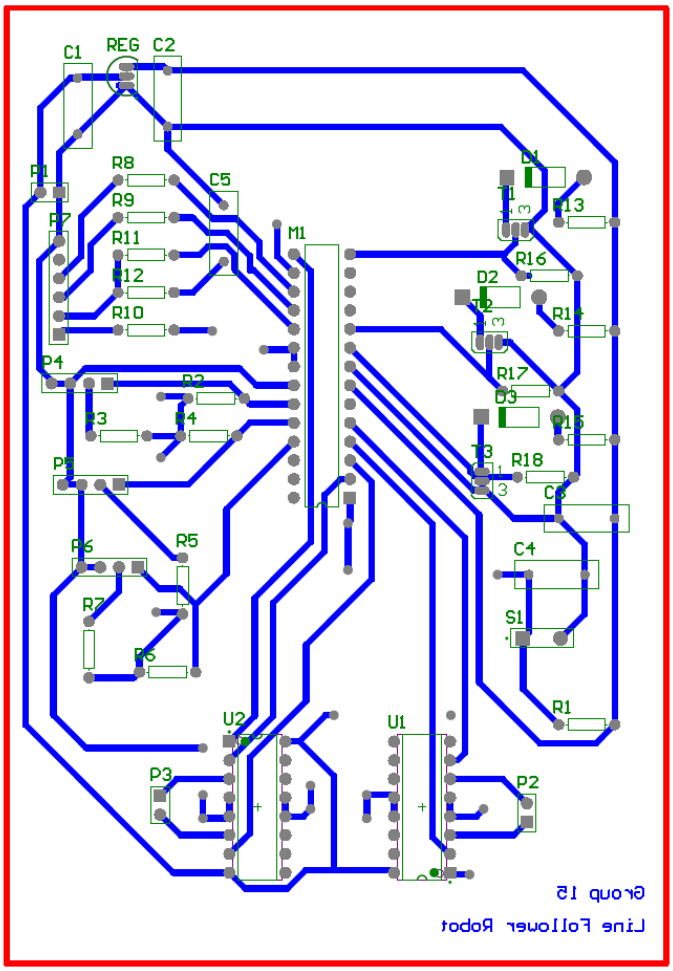


Figure 3 - Completed PCB Layout

3.3. PCB Construction

The physical PCB construction and component placement was carried out in sets of functional groups. These key groups of construction were the headers, power supplies, LED circuits, sensor circuits, motor drivers, ISP programming configuration and button circuits. Each group was tested for functionality as best as it could without the use of the Atmega8 upon completion. This method allowed any incorrect solders, short circuits or other errors to be narrowed down to the group most recently appended to the PCB. This allowed a relatively error free construction of the board. It did not, however, allow for any means to fix any flaws inherent in the board's design. These problems were identified throughout the board's construction and the most effective solution was then determined and implemented.

3.4. Gearbox Assembly

The gearbox was assembled in the B configuration giving a rated torque of 27.3×10^3 Nm and a rotation speed of 245 rpm. There were no major limitations on maximum torque for the robot and if the speed was to be too high this could simply be reduced by lowering the duty cycle value.

3.5. Mounting External Peripherals and Lines

The next step in the construction process was mounting the external peripherals. This involved the secure fastening of the Tamiya double gearbox and motors along with the placement of the sensors.

For the motor and gearbox to be mounted on the underside of the board parts of the PCB had to be cut away at the sides in order for the wheels to not be touching the PCB. This was done carefully in the student mechanical workshop followed by the drilling of two small holes for the motors screws.

The location for mounting the optocoupling sensor was important for the project and had to be carefully considered. It was determined, due to the nature of our proposed maze algorithms, that a single sensor should be mounted as close to the wheel base as possible. As it did not seem practical to mount it directly under the motor the sensor was attached to a piece of Vero board and mounted at the front of the gearbox (the closest point to the central wheel base). As

discussed below, this was not the best placement of the sensor as there was actually sufficient room between the gearbox and the surface of the paper to mount the sensor directly under the axis. However, by the time this was considered as an option it was too late to change the location as the software relied on this placement and therefore time constraints would not allow it.

The sensor was also distanced very close to the ground (~2mm) so that higher precision readings could be obtained. Being this close to the ground mean all the light being received was from directly under the sensor, so the reading was precise. If the sensor had been higher off the ground, the reading would have been an average of the light coming from the area at the front of the robot. This would have made navigation hard as it would have effectively blurred the edge of the maze. It would also have made the identification of the grey even worse, as even at 2mm there is substantial readings of grey by the sensor and at greater mount distances this effect is amplified.

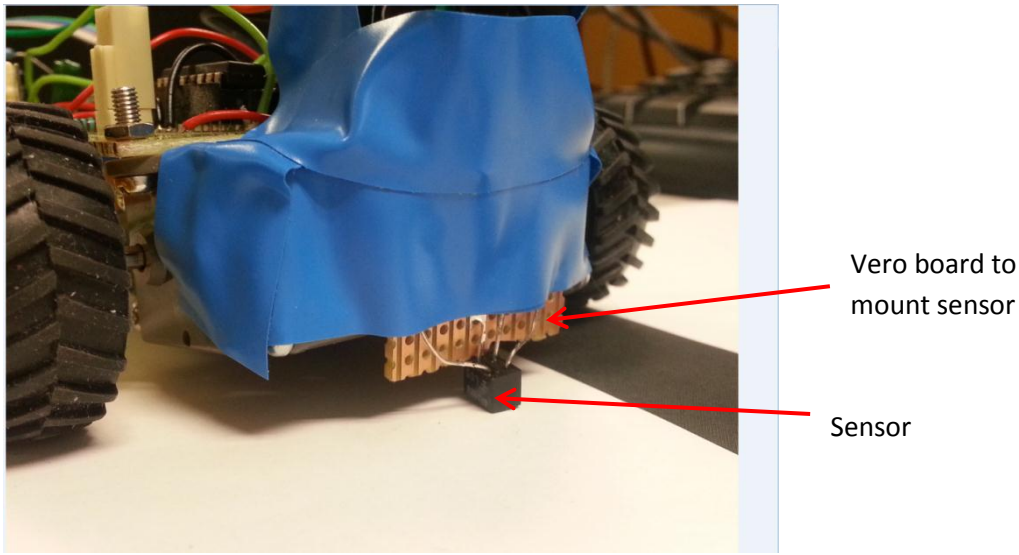


Figure 4 - Placement of the sensor between wheels.

The final piece of hardware to configure was the parallel port required for interfacing the Atmega8 and a PC. This was done by carefully interpreting the diagrams provided in the Atmega8 documentation [5]and soldering each pin to the corresponding wires to be attached on our PCB. The configuration for this can be seen below in Figure 5 and Figure 6.

Headers were connected to leads for both motors, programming port and the sensors and were then able to be easily attached/detached from the PCB. With the physical construction of the hardware complete the path was set for the software portion of the project to commence.

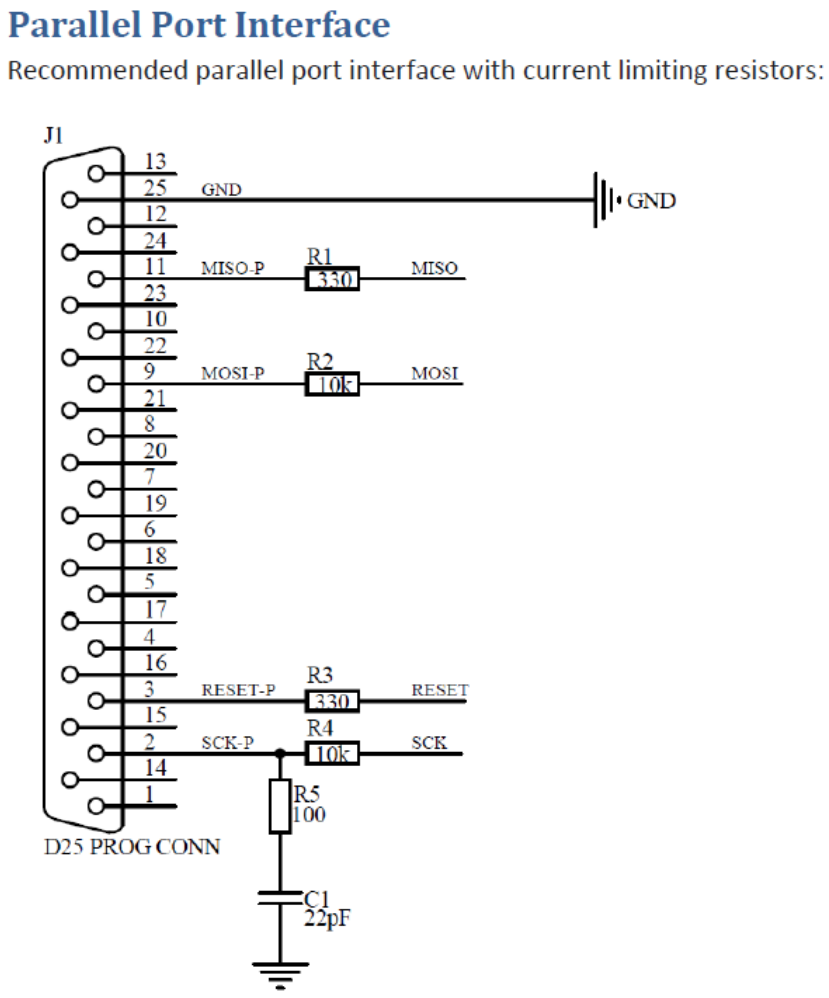


Figure 5 -Schematic diagram for parallel port plug to computer.

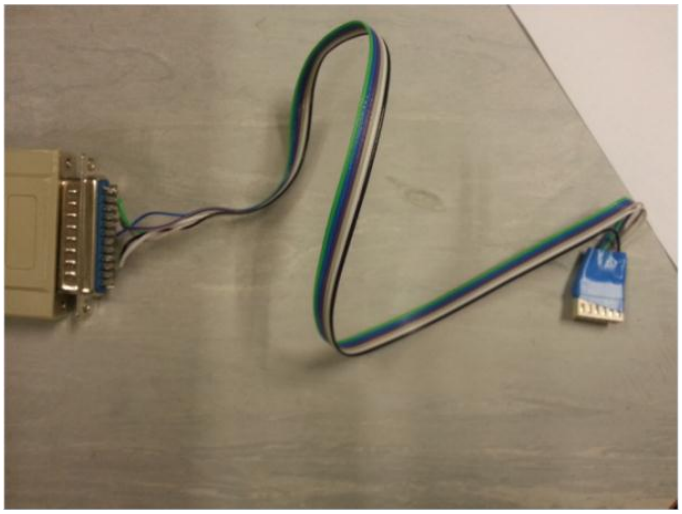


Figure 6 - Parallel port wire to connect robot to computer.

3.6. Configuring the ISP

The first step in the programming of the Atmega8 was testing the connection of the parallel port. This was done by using the provided instructions from the provided Atmega8 documentation [5]. The command show in Figure 7 could then be used within winAVR to detect an active connection.

```
avrdude -p m8 -P lpt1 -c spl2
```

Figure 7 - Command to Check Connectivity to the Atmega8.

This did not yield positive results on the first attempt. However upon re-soldering of one of the leads the command indicated the controller was connected and operational.

The commands shown in Figure 8 were then combined into a single file 'start.cmd' so that the program could easily be compiled, linked and loaded simply by typing start.

```
avr-gcc -g -Wall -mmcu=atmega8 -std=c99 -Os -I. main.c  -o output.elf
avr-objcopy -O ihex -R .eeprom output.elf output.hex
avrdude -p m8 -P lpt1 -c spl2
load-pp output
```

Figure 8 - Commands to compile, link and load the program.

This allowed additional '.c' files to easily be linked by adding them after 'main.c'

3.7. Software Design and Driver Libraries

The software was designed using a bottom up approach. Low level code was written first, beginning with the code to initialise the microcontroller and set the registers. This was then built on with the code for controlling the motors and sensors, allowing the high level maze solving algorithm to be abstracted from the low level code. This method was highly beneficial as it allowed each part of the code to be written and tested before it was used in more complicated functions. Using this method made debugging easier as once each section had been written and thoroughly tested, it could be assumed to be fault proof, limiting the search for any bugs in the code to areas of new code. The method also allowed for the different levels of code

to be written by different team members, as long as there was communication about the function names and variables being used. For example, the code for the motors used register values in order to create functions for basic motor controls, such as forward and backward movement. However, the person using those functions could do so without knowledge of the specific registers used by them as long as they knew what response to expect when they were used. Hence another person without the knowledge of how the low-level functions were constructed could still use these functions to create more complicated functions for variable turning speeds.

In order to maintain this essential ability to use other people's code, it became paramount to keep documentation of each new function up to date – especially when the lower level drivers were still being developed. As such, specific standards were used across the group to ensure that others could easily understand the inputs and outputs to the function, without needing to understand the process in the function itself. The standard used was the official C Standards Coding Specification [6] which, as well as keeping the code easily readable, provides an easily understandable means of understanding parameters and return values of functions based on the descriptions at the start of each function.

By developing the software in this layered manner, high level functions became dependent on low level functions in order to operate properly, as can be seen in the dependency tree in Figure 9 below. Abstraction is clearly demonstrated here, as the basic function modules are used by the higher level modules, but not the other way around. This is good one-directional development practice. It can be seen, for example, that 'avr/io.h' is a low level module – which it is, being the vital link between the software and the hardware.

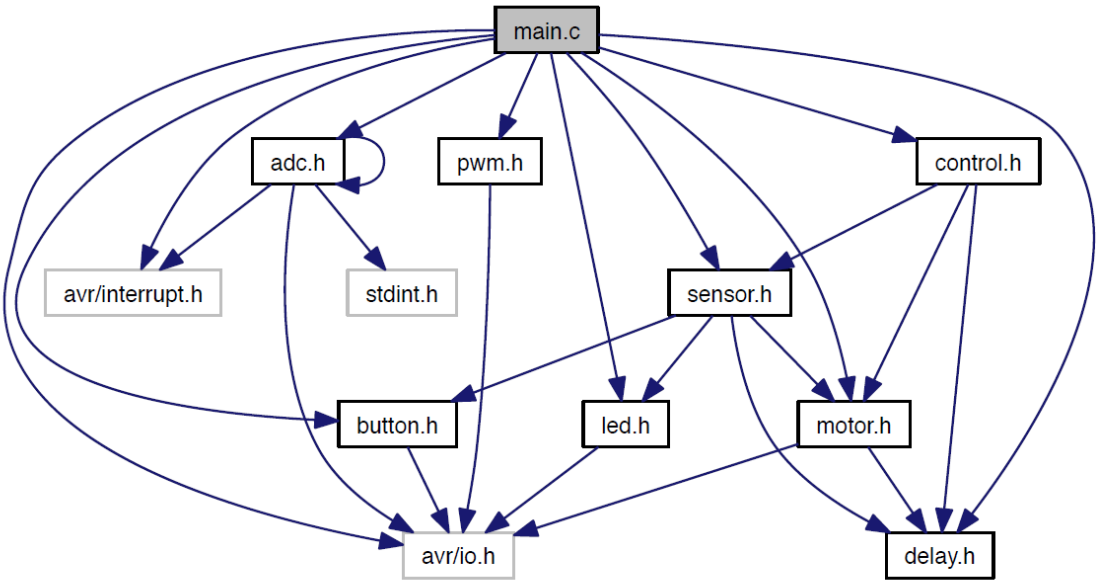


Figure 9 - The Dependency Tree for the Robot Software

The writing of the driver libraries was then carried out in logical manner, beginning with those necessary for debugging.

3.7.1. *LEDs*

The first module that was required (and could easily be observed to be operational) was that for the LEDs. The LEDs were driven simply by initializing PB0, PB6 and PB7 as logical outputs and then setting the corresponding bits on PORTB to high. This was soon abstracted into four main functions for led initialization, switching on, switching off, and toggling. These functions can be seen in 'led.h' in the appendix. Additionally an LED test function was created to toggle all LEDs periodically. The module worked well and the proposed concept of driving the LEDs with a logic low output was operational. This also made the implementation of the remaining drivers easier as the LEDs served as an invaluable debugging tool.

3.7.2. *ADC*

The analogue to digital converter implementation proved to be exceedingly difficult. The first attempt was a compilation of c example code pieces from the data sheet. This yielded no results so research was carried via online resources to examine operational ADC modules for an Atmega8. This unearthed a working combination of pieces of code found online [5] and examples from the Atmega8 datasheet. The code was then abstracted into two core structures: the ADC initialization function and the interrupt service routine (ISR), as seen in 'adc.h'.

The ADC was initialized to have the following features:

- 125 kHz sample rate.
- Using AVCC as reference voltage.
- Set as 8 bit ADC.
- Operates in Free Running mode.

3.7.3. *PWM*

The PWM module was implemented in a similar manner to the ADC. It required careful examination of the Atmega8 datasheet in combination with online research [6]. The end result can be seen in pwm.h in the appendix. The PWM was configured to have the following features;

- Output to pin PB3.
- Operates at 488 Hz.
- Operates in fast PWM mode.

The duty cycle was then altered by modifying the OCR2 register (0-255).

Attempts were given to produce an additional PWM waveform for the second motor but it was decided that the modulation could simply be done by feeding the appropriate logic into a single motor driver. Therefore the same PWM waveform could be fed into both the left and the right inputs of the motor drivers making a second PWM waveform unnecessary.

3.7.4. Motor

The motor logic module remained very simple using four GPIO pins from the Atmega8 set as logic outputs to provide appropriate values for motor control to the motor driver. This was done by initializing PD0, PD1, PD2 and PD3 as digital outputs and then writing to the corresponding value on PORTD. A function was then made to control each logical port in terms of its side, direction and new state (0 or 1). From here a set of simple functions could be abstracted to describe simple robot movements. This set included seven fundamental control functions: stop, move backwards, move forwards, turn right, turn left, pivot right and pivot left as can be seen in 'motor.h' in the appendix.

3.8. Maze Solving Algorithms

The focus on high levels of abstraction allowed for the code to be very easy to use and manipulate at high levels. This was highly advantageous when considering our team's goals; we wanted to use only a single sensor to solve the maze. However, we were unsure if this was viable and hence we wanted as much of the code to be capable or reuse in the scenario where extra sensors are required. These high levels of abstraction and low levels of coupling within the code were therefore essential if the single sensor version did not work.

The sensors were read based on an interrupt service routine, meaning that periodically each sensor would be checked to determine whether it was over a black track line, a grey finish line, or white empty space. This was particularly important when developing code which would be able to be implemented in a three sensor scenario. This was run as a foreground task, meaning that regardless of where the background code was running the sensors would always be up to date allowing for a quick response by the robot if the need should arise.

The fundamental idea of the algorithm was easy to implement using basic abstraction. The idea: every time the robot crosses from white to the black, turn to the left to try and reach the black again. Similarly, every time the robot crosses from the black to the white, turn to the right to try and reach the white again. However, at such a basic level this was unstable even when travelling along a white line. Several important development steps were therefore necessary to carry out.

3.8.1. Counter-Turn Stability

In order to stop the oscillations of the robot along a straight line becoming out of control, the first development that had to be made to the basic code was recording the amount of time that the robot had been travelling on the turn and to try and counteract that with a proportionally large pivot following the transition onto the new colour. If this pivot was similar in magnitude to the size of the turn that had just been made on the other colour then much of the swinging motion could be counteracted and the robot would be constantly correcting itself to travel closer to the line.

However, this did not solve the problem at corners, as the accumulation of the error when moving around a corner would cause an overcorrection when that error was realised in a pivot. In order to help with this a maximum limit was imposed in the amount of error that could be accumulated.

3.8.2. Turn Curvature and Speed Variations

Even with the correction for turns added to the code, the robot could still become unstable at high speeds. In order to increase the speed while retaining stability the code was made to make different turns for the turn on the black and on the white. The turn was set to a pivot on the black, as at high speeds there was a need to move off the black as quickly as possible – however, if the incident angle and speed onto the black was too high, the momentum would carry the robot across to the other edge, causing it to skip the black and end up on the other side of the line. This situation can result in the robot never finding the end of the maze or getting stuck in a loop, and was avoided at all costs.

In order to deal with this problem, the speed at which the turn on the white was taken was set to a minimum. Also, as the white did not provide anywhere near as significant movement constraints as the black, the motion was set to be a turn rather than a pivot, creating more of a forward motion in the turn than in that of the black.

3.8.3. Oscillation Compensation

The problem that was introduced with this advancement was the reduced speed when the robot was travelling very close to the line, as the pivot component of the turning sequence when the robot was on the black would cause it to move very slowly and, in some cases, stall. In order to deal with this, the motion on the black for the first few values was set to be a turn. However, this could cause problems when the black edge was hit with high incident and speed as the robot was far more likely to shoot straight through to the white on the other side.

The solution to this was relatively easy – if the robot was on the white for a substantial amount of time then the turn would be determined as likely to have too high an incident angle and speed and therefore, in these situations, the response on the black would be to immediately

pivot. This provided a good means of dealing with this 'shoot through' problem and allowed the speed to be increased significantly.

3.8.4. Cornering for Turns

Turns on the white were now causing the robot to move slowly due to the curvature of the white motion forcing it regularly onto the black, where it would oscillate. Although this effect was reduced by compensating with a turn rather than a pivot where possible, it was still hard to determine whether it was safe to turn rather than pivot, especially when approaching a left turn in the track. In order to increase the amount of time that the robot spent on the white a new method of turning was created; the robot would travel nearly straight (very slightly curved towards the black) for a fixed amount of time then would sharply pivot roughly 50 degrees to the right. If the robot was travelling along the edge of the white and black then it would quickly move back onto the white and repeat this straight motion.

This greatly improved the speed of the robot along straights. It also worked well for corners, as this 50 degree turn at regular intervals would bring the robot back around to the wall after several of these turns.

By this stage, the robot could easily solve mazes at a significant speed. The problem that was identified at this stage to further increase the speed was a hardware problem that was extremely hard to solve with software alone; namely, the placement of the sensor on the front of the robot meant that the speed was limited by the 'shoot through' problem as described above, where the momentum of the robot would carry it through the black and onto the white on the other side. The easiest solution to this was determined to be the relocation of the sensor to a point midway between the wheels, just below the axis. However, there was not enough time left to perform this relocation of the sensor and the subsequent adjustments to the code. This problem is further analysed below in the Discussion section.

3.8.5. Breaking Out of Loops

One problem identified earlier in this report was the problem of a robot stuck inside a loop. In such a loop, the robot will be continuously turning left. As such, the aim in such a situation is to move across to the right side of the loop and continue briefly on the right there before moving back to the left – or, alternatively, never moving back to the left unless the right places the robot in another loop.

In order to implement this solution, the robot first had to be capable of identifying left and right turns. This was fairly simple to implement, as a left or right turn that continued beyond a certain (empirically measured) number of turn cycles would count as a corner in that direction. If a corner was found for a left turn, a count would increment; if a left, it would decrement. If the number reached four then the robot would move straight ahead across the black until the white on the other side was reached, whereupon it would continue on its way. This solved the

problem as long as the thresholds for the left and right accumulated values were correctly chosen.

3.8.6. Grey Detection

Detecting the grey finishing line proved to be one of the most difficult components of the project. Since the grey color was highly variable between printed out sheets, it was determined that the best approach to identifying the line would be to calibrate the robot to the grey each time it is to be run. This meant that the robot could be used in a far wider range of locations, and removed the environmental effects from ambient lighting and variety of print qualities.

The ADC read a range of values between 0 and 255. These then had to be interpreted as black, white or grey. When the calibration was run when the robot was started a single value of grey was identified. Since variability had to be allowed for in the program, as the color would not be exactly identical across the line ranges either side of this value were used.

Two different criteria were used to identify whether a grey line was detected based on two different scenarios. The first condition was that a very precise value of grey had to be read for 6 consecutive readings. The allowable range for these values was ±1. This helped to eliminate false readings by ensuring an exact color match had been detected. The second condition was to ensure that the sensor remained grey for a reasonable period of time. False grey readings were often obtained by the first method as there were small patches of grey at the boundary between the white and black lines and in patches where the track had been worn. To ensure that these were not interpreted as being the finish line a large number of samples were taken, where a majority of them had to fall within an acceptable range. A count was taken, where one was added each time a grey was detected, and 5 subtracted each time a non-grey was detected. If the count exceeded 15 it was determined that a line was detected. Because of the large number of samples taken, and the strong effect of non-grey readings, the range was increased. The final range used was ±10. If both the above conditions were met, it was determined that the robot had reached the end of the maze, so both the motors were stopped.

When developing this method there were many problems with false grey readings in the middle of the maze. It was very important to ensure that these were eliminated as if they triggered in the middle of a run, the robot would stop and be unable to complete the maze. This meant that very harsh completion criteria needed to be used. However as these restrictions were introduced the likelihood of the robot missing the end of the maze also increased. The main reason for the robot missing the end of the maze was that it would be driving along a long straight line. The robot moves quickly across these sections, meaning only a limited number of readings are able to be taken across the grey area. This meant that it was possible that the robot could have driven across the line before a sufficient number of readings had been taken. A trade off therefore had to be made between the

probability of missing the actual line and the chance of stopping in the middle of the maze. The final values chosen optimized this, so that both possibilities were minimized

4. Design Problems and Solutions

Many problems were encountered within the design process, some so significant that a "start over" approach would be required to fix completely. Since this was not an option at the later stages of the project, alternate options needed to be considered. These had varying levels of success, with some yielding results better than anticipated, whilst others introduced areas for potential faults at a later stage.

The first of these issues was the attempt to run a track between two IC pins as seen below in Figure 10 below. This was easily fixed by cutting the tracks with a scribe where needed and then running a jumper to replace the original track.

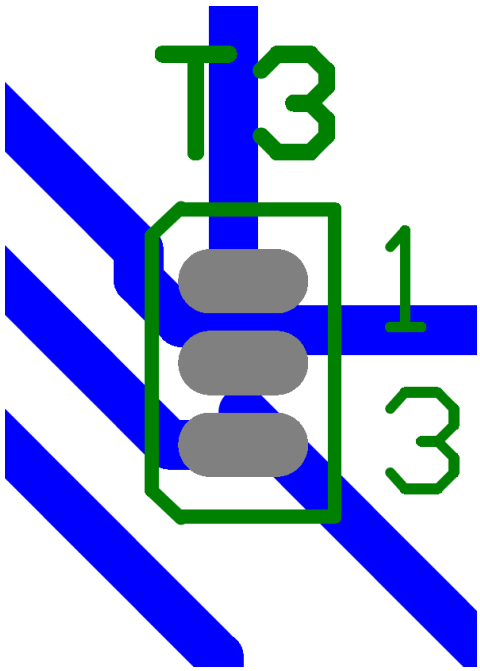


Figure 10 - Track layout between IC pins causing a fault.

The voltage regulator footprint used in the PCB artwork design was incorrect. The consequence of this was that the pinholes were stationed far too closely together and the V_{in} and ground tracks were in the wrong positions. This problem was troubling as our design was set up specifically to use the MCP7805 voltage regulator. The final solution to this was to bend the IC pins around one another so that it agreed with the design and fit near the PCB pads. This solution, while undesirable, did not have any adverse effects on the final product, and did not

cause any further issues.

When the FETs were used to drive the LEDs on breadboard lighting could not be achieved. Resultantly it was decided to use BJTs to drive the LEDs. This worked well as they were smaller and more cost efficient. Furthermore the choice of PNP type BJTs allowed the LEDs to emit light with a logic low applied to the base, indicating some circuit functionality before microcontroller programming commenced. The proposed transistor configuration and simulation can be seen in Figure 11and Figure 12 respectively.

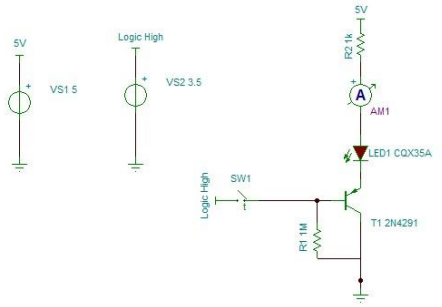


Figure 11 - Schematic for BJT driven LEDs.

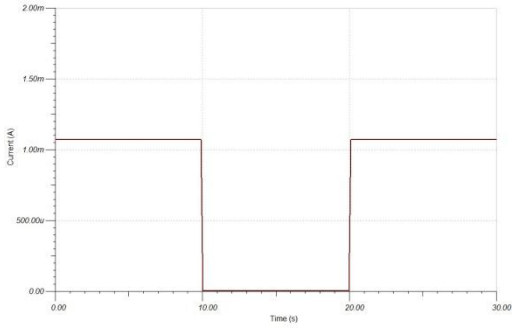


Figure 12 - TINA Simulation for BJT configuration with switch on for ten seconds.

A major design flaw arose from the misinterpretation of the motor driver data sheet. The group interpreted the IC as being able to drive only one motor each and thus requiring two in total, however only one was required and supplied. Because of this the PCB design was far more complicated than necessary, and many of the jumpers and tracks could have been removed if this problem had been discovered at an earlier stage of the design process, however the PCB had already been fabricated when this was discovered. This meant it was not practical to redesign the PCB, therefore the final product contains a large number of redundant and unused tracks. This also meant that additional holes had to be drilled along the existing driver, and once placed on the board jumpers were installed from these to the original motor logic holes.

It was realised upon implementation of the analogue to digital convertor (ADC) drivers that a capacitor had not been placed on the AREF pin to ground on the Atmega8 this was suspected to be causing problems so the capacitor was added.

It was realised that the sensor's voltage output did not have an ideal range with the 1kΩ resistor, but this was resolved by replacing it with a 75Ω resistor. Access to the original resistor was restricted so the new resistor had to be hard wired to a flying lead from the sensor as seen below in Figure 13.

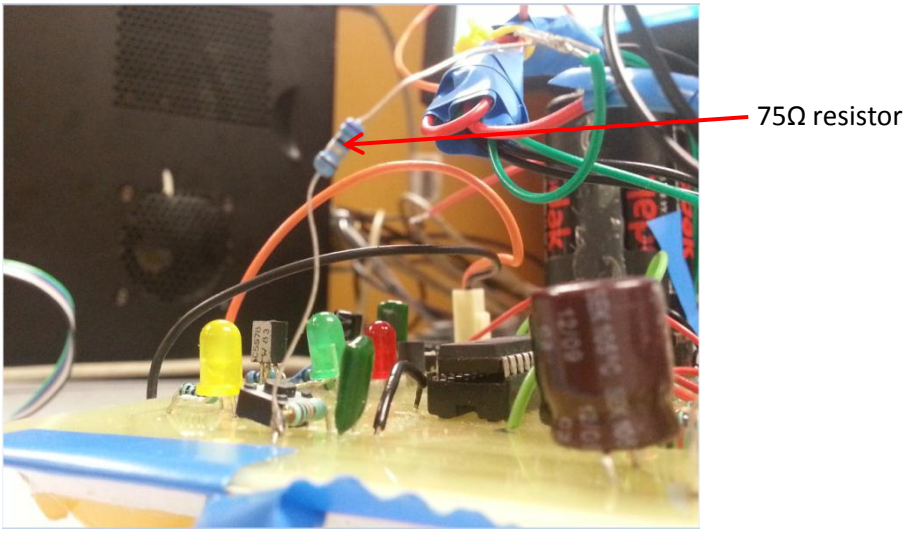


Figure 13 - 75Ω resistor added after testing to replace 1kΩ resistor.

The grey 'finishing line' proved to be difficult to detect using the required sensors. This was due to the highly variable readings that the sensor could potentially provide while on the grey line, while was emphasised by the ambient lighting in the room. Care needed to be taken when defining threshold limits for the grey reading so that there was a very low likelihood of false grey readings, while ensuring that the robot could consistently respond to the finish line. Taking the importance of this reading into consideration it was decided to calibrate the outputs of the sensor to grey of the finishing line before the robot commenced solving the maze. This was performed by placing the robot directly on the grey finish line to determine an accurate reading. To implement this additional button needed to be added to the design to ensure correct calibration. The circuit addition is shown below in Figure 14. The button was multiplexed to PB0 pin as button inputs were only required prior to operation.

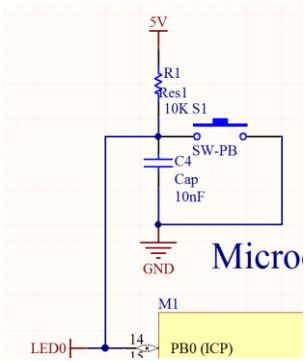


Figure 14 -Schematic Diagram for added button.

5. Design Results

An important part of the project was the testing the final stages of which involved timing the robot through a maze similar to last year. The maze used can be seen below in Figure 15. The robot took around 50 seconds to solve and stopped accurately on the grey finishing line. A video of the maze completion was taken and the approximate path taken by the robot can be seen in Figure 15.

As can be seen the robot follows a 'bouncing' type of path around the left hand side of the maze. This works well for majority of the maze however at sharp corners and dead end paths it can turn too widely so that it follows the white tape between the sheets of paper. This can potentially cause the robot to get stuck inside a loop, or lose the maze completely. This problem does not appear when fully charged batteries are used, so should not prove to be a problem in actual testing situations.

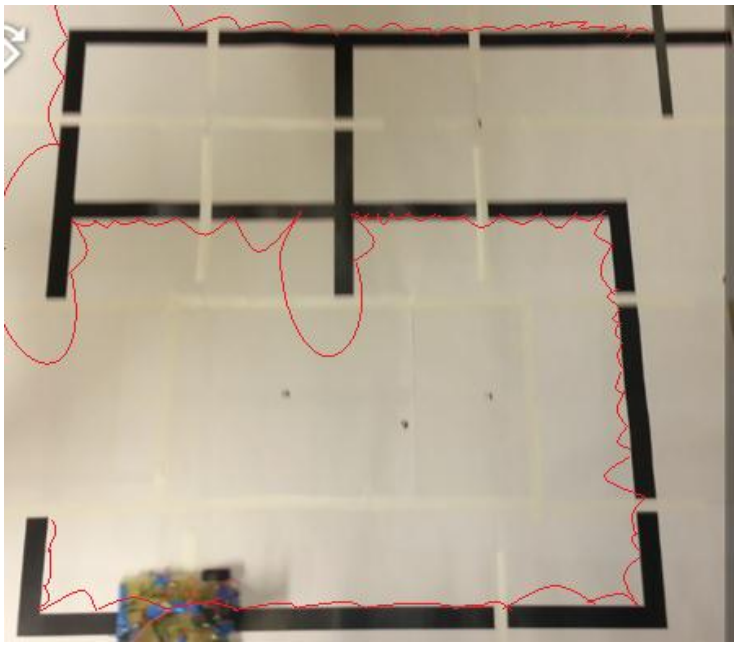


Figure 15 - Photo of example maze with approximate path taken

To effectively assess the performance of the line following robot, several electrical elements were assessed using the oscilloscope and compared with that to be expected.

It is important to consider the overall power consumption of the robot as it is, in essence, a portable device running off 4 AA batteries equating to approximately 6V. The device draws an idle current of approximately 80 mA so the power dissipated is approximately 0.5Watts. This can be compared with the ~1.2A drawn while the motors are operating, giving a power dissipation of ~7.2Watts. It is easy to see that there are issues surrounding the high power consumption during the robots operational state. It explains why voltage drops are to be expected due to over drawing current and why battery lifetime is greatly reduced.

One of the key issues in the development was known to be the differencing of the black and grey shades. This can be seen in Figure 16, which shows the different sensor signals for the shades of white black and grey from left to right respectively. From observation it is clear that the white and black signals hold significantly differently values. The oscilloscope is set to 2 Volts per division meaning that the voltage difference between the two is approximately 3.5V, confirming that the difference when digitally converted should be easily distinguishable. It is also clear that grey and black signals are more closely related, having an approximate 0.25V difference. This also confirms that the difference is significantly less, explaining the need for certain procedures when digitally comparing.

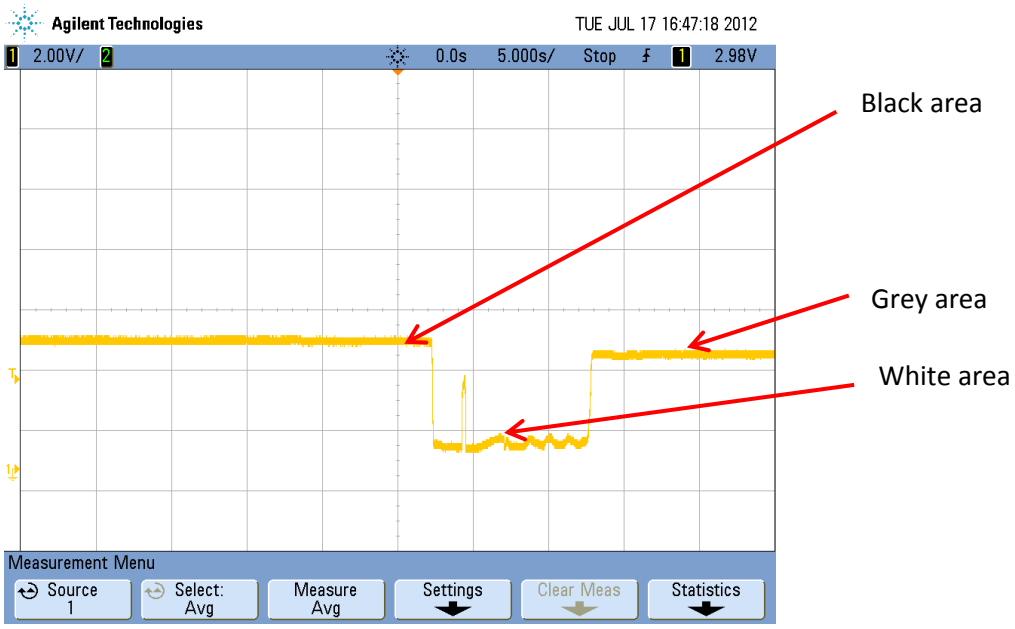


Figure 16 - Feedback voltage from sensor when over the black, white and grey areas of the maze.

Another observable electrical property of the robot is the out PWM waveform which can be seen below in Figure 17. The waveform capture was taken with a duty cycle register value of 150 (150/255 = approximately 58.8%) with an assigned value of 488 Hz. As can be seen the waveform duty cycle and frequency is exceptionally close to that predicted, indicating the PWM waveform is operating as expected.

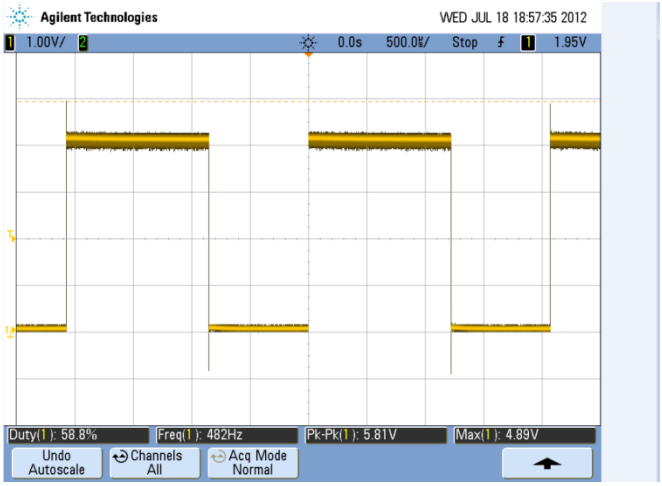


Figure 17 - Atmega8 PWM waveform

Finally the regulated voltage was compared with the voltage on the input during motor operation with mildly discharged batteries. This can be seen below in Figure 18 which shows that while the two signals are subject to a lot of noise, the regulated voltage remains significantly more noise free than the batteries levels.

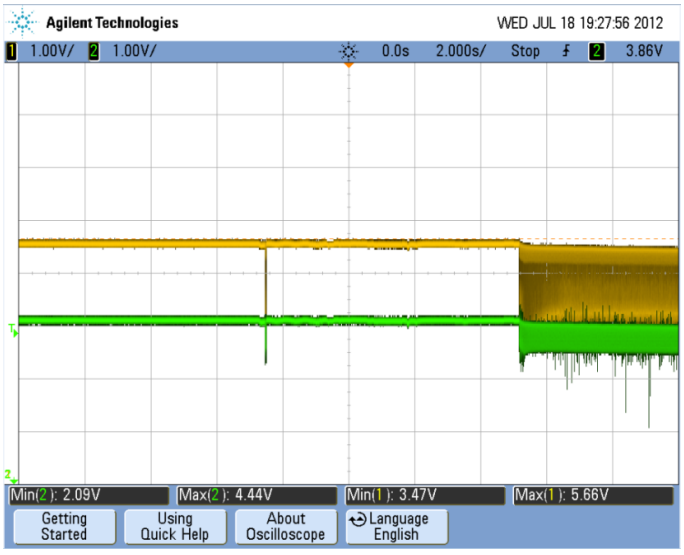


Figure 18 - Voltage Supplies Before and During Motor Operation.

6. Discussion

Our team's initial design goal was to be able to implement the robot using only a single sensor. However, during the planning stage of the design process we were unsure whether this would provide the best outcome. Even though there were two bonus marks available based on the use of only one sensor, there were five marks based on the speed in which the robot was able to complete the maze. Since it was unknown how the removal of sensors would compromise the speed of the robot, provision was allowed to utilise all three sensors. This resulted in a more complicated design process, however minimised risk at the end of the project after the construction and programming of the robot. Three sensors were allowed for in the entire design process, and the PCB included three separate headers, one for each of the sensors, to allow for easy removal.

The single sensor version of the code was created first, as this was our preferred configuration. It was discovered that the robot still performed quickly using one sensor, so this configuration was kept. Although the use of three sensors in such a robot could potentially result in less side to side movement on the track, and therefore travel more quickly, the lack of processing power in the microcontroller used rendered the three sensor configuration too slow for this case.

The placement of the sensor could be changed to improve performance. The sensor was located forward of the axle, so when the robot pivoted, the sensor swung around, as shown in Figure 18. This caused the sensor to overshoot the line during turning, resulting in oscillations after each corner. It also had the potential to cause more serious problems in certain scenarios, as shown in Figures 19-20. If the sensor had been located at the centre of rotation, these oscillations and other problems would have been greatly reduced. Unfortunately this was not possible due to difficulties in mounting a sensor under the gearbox and time constraints. By the time the problem with the sensors location had been found, a large amount of code would have had to be altered if the sensor had been moved.

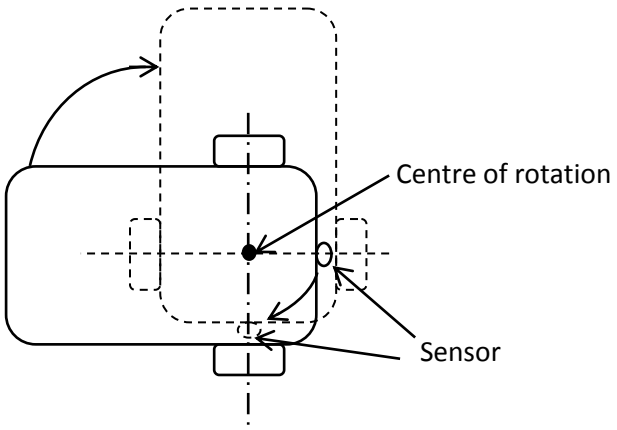


Figure 19 - Diagram of robot showing movement of sensor during pivoting action.

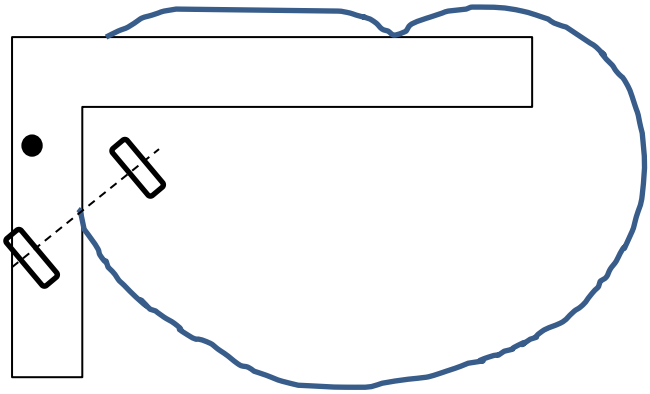


Figure 20 - The speed of the robot causes it too swing out and hit the track after the corner.

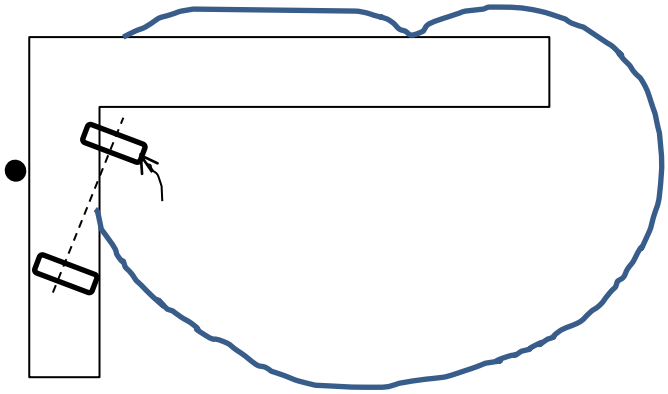


Figure 21 - The algorithm causes the robot to swing to the left to relocate the track, but in doing so the sensor is carried across the track and enters the white space on the other side. If the sensor had been between the wheels the robot would have had more time to complete the turn while keeping the sensor within the track. This would therefore allow the robot to travel at a higher speed around this corner.

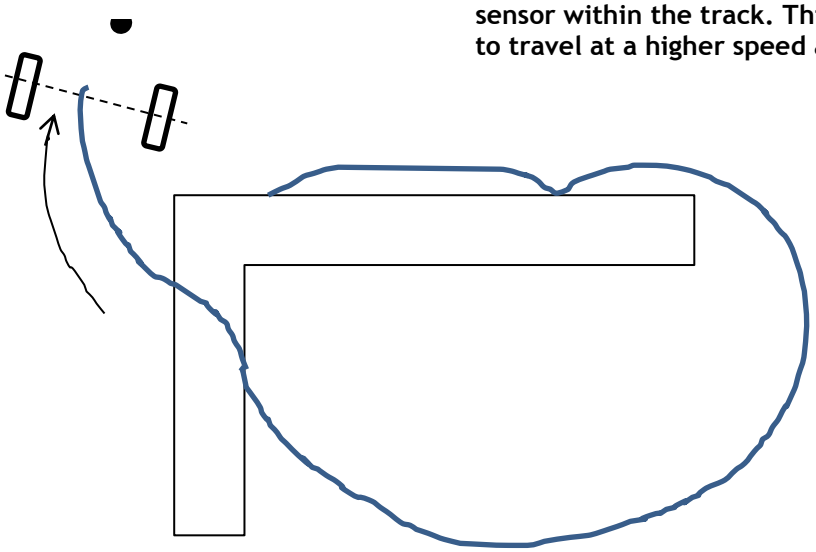


Figure 22 - The speed of the robot causes the robot to overshoot the line and travel off the track.

To improve the robot's performance in the maze there are several changes that mostly with improved hardware. Research into other line maze solving robots performance increased with more sensors being used. Extra sensors on each side need for bouncing along one side of the maze, as the side sensors are used for intersections while the middle sensor is used for keeping the robot straight, speed as the robot can now drive straight. Even better than this is using sensors along with multiple sensors at the front. With multiple sensors, the pattern of lines would match up to different intersections or be able to determine if the straight, as clearly shown in

Figure 23.

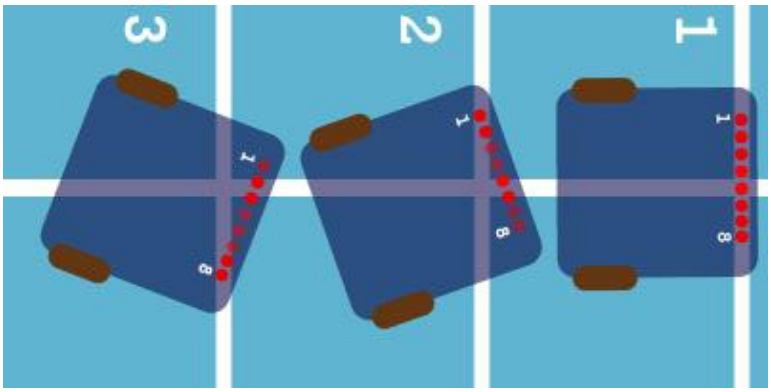


Figure 23 - Patterns of activated and inactivated sensors make intersections easier to recognise [7].

More sensors would require a faster processor, although the increase in memory that would come with a microcontroller upgrade would be beneficial. If the memory was large enough the robot could record each intersection it passed through. By recording every turn, and rewriting dead ends as the correct turn, the robot could plot the direct path from start to finish as a series of nodes. This would allow the robot to follow this path on its second run, significantly reducing the time taken to navigate the maze.

With this knowledge of the maze the robot could also travel faster on the straights, as it would know when to speed up or slow down for an intersection. This advantage could be enhanced with more powerful motors for the wheels.

These improvements involve hardware not available to this project. Considering the hardware available, and the fact that less sensors resulted in more marks, the design chosen can be seen to be the best option.

The nature of our design left a lot of the hardware open to modifications at all times. This worked well as it allowed us to deal with unforeseen problems that arose. If we were to fix everything into strict positions this would not have been possible. An example of where this was beneficial was the calibration of the grey sensor value. Although in some ways this made the robot appear somewhat ‘piecemeal’ the benefits that came with this approach outweighed any drawbacks.

7. Conclusion

The robot created in the process of this assignment successfully meets the criteria of being able to follow a line and solve a maze. The outcome only requires the use of a single sensor, both saving on cost of the robot and resulting in a better performing product. The speed in which the robot is able to complete the maze is acceptable to our group, and should result in a competitive ranking when directly raced against other robots. Test trials showed that the robot was able to complete an example course in a time of 55 seconds with 100% accuracy. This is an acceptable result, and shows a high level of reliability of the robot.

8. Bibliography

[1] RobotWorx, "www.robots.com," [Online]. Available: http://www.robots.com/industry. [Accessed 7 2 2012].

[2] Unlimited, "Inro and Fonterra's automatic tactics," 29 6 2001. [Online]. Available: http://unlimited.co.nz/unlimited.nsf/innovation/inro-and-fonterras-automatic-tactics. [Accessed 2 7 2012].

[3] H. &. d. A. Abelson, Turtle Geography: The Computer as a Medium for Exploring Mathematics, Cambridge, MA: The MIT Press, 1986.

[4] Shanghai Chipswinner Electronics Co. Ltd, "MC7805 3-Terminal 1A Positive Voltage Regulator," 4 2007. [Online]. Available: http://www.chipswinner.com/DS/MC7805.pdf. [Accessed 28 6 2012].

[5] Matias, "Atmega8 ADC setup," 19 6 2008. [Online]. Available: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=65424&start=0. [Accessed 14 6 2012].

[6] "PWM On The ATmega8," [Online]. Available: https://sites.google.com/site/qeewiki/books/avr-guide/pwm-atmega8. [Accessed 16 6 2012].

[7] Kamal, I, "Line tracking sensors and algorithms," 28 2 2008. [Online]. Available: http://www.ikalogic.com/line-tracking-sensors-and-algorithms. [Accessed 20 6 2012].

9. Appendix

	Cost ($) /Unit	Quantity	Cost ($)
Resistors			
100Ω	0.05	1	0.05
330Ω	0.05	2	0.10
1kΩ	0.05	9	0.45
10kΩ	0.05	3	0.15
1MΩ	0.05	3	0.15
Capacitors			
10nF	0.20	1	0.20
1µF	0.30	2	0.60
100µF	0.20	2	0.40
LEDs			
Green	0.25	1	0.25
Blue	0.25	1	0.25
Red	0.20	1	0.20
Regulators			
MCP7805	1.00	1	1.00
Transistors			
CSS7BW83 BJT	0.25	3	0.75
Diodes			
1N4001	0.20	1	0.20
Optocouplers			
QRD1114	1.50	3	4.50
Headers			
2 way Socket	0.15	3	0.45
2 way Plug	0.10	3	0.30
4 way Socket	0.30	3	0.90
4 way Plug	0.20	3	0.60
6 way Plug	0.40	1	0.40
6 way Socket	0.28	1	0.28
Switches			
Buttons	0.30	1	0.30
Microcontrollers			
Atmel ATmega8	6.00	1	6.00
Motors	0.00	2	0.00
L293D Motor Driver	4.00	1	4.00
Gearboxes	0.00	1	0.00
Wheels	0.00	2	0.00
Battery Holders	0.00	1	0.00
D-connector	0.00	1	0.00
Total ($)			**22.48**

Figure 24 - Bill of materials after planning stage.

Project Name: Line Following Robot

	Cost ($) /Unit	Quantity	Cost ($)
Resistors			
75Ω	0.05	1	0.05
100Ω	0.05	1	0.05
330Ω	0.05	2	0.10
1kΩ	0.05	8	0.40
10kΩ	0.05	4	0.20
1MΩ	0.05	3	0.15
Capacitors			
10nF	0.20	3	0.60
1µF	0.30	2	0.60
100µF	0.20	2	0.40
LEDs			
Green	0.25	1	0.25
Blue	0.25	1	0.25
Red	0.20	1	0.20
Regulators			
MCP7805	1.00	1	1.00
Transistors			
2N5485G MOSFET	0.35	3	1.05
Diodes			
1N4001	0.20	1	0.20
Optocouplers			
QRD1114	1.50	1	1.50
Headers			
2 way Socket	0.15	3	0.45
2 way Plug	0.10	1	0.10
4 way Socket	0.30	3	0.90
4 way Plug	0.20	3	0.60
6 way Plug	0.40	1	0.40
6 way Socket	0.28	1	0.28
Switches			
Buttons	0.30	2	0.60
Microcontrollers			
Atmel ATmega8	6.00	1	6.00
Vero Board	0.60	1	0.60
Motors	0.00	2	0.00
L293D Motor Driver	4.00	1	4.00
Gearboxes	0.00	1	0.00
Wheels	0.00	2	0.00
Battery Holders	0.00	1	0.00
D-connector	0.00	1	0.00
Total ($)			20.93

Figure 25 - Bill of materials at end of project after corrections.

User Manual

To operate the line following robot it needs to have 4 AA batteries inserted correctly into the battery holder on the top of the board. Once the batteries are in place, press the calibration button (Figure 26). Following this the robot will turn on, and await calibration. Once the board is ready for use, place it so that the sensor is above the grey finishing line. Press the calibrate button (Figure 26). The red led will light up when the calibrate button is pushed, and remain on for the duration of the calibration process. Upon completion of this process the light will turn off and the robot will be ready to solve the maze. It must then be placed at the start of the course, and the calibrate button pushed again. As soon as the button has been pressed it will begin to follow the line and attempt to solve the maze. Upon completion of the maze the robot will come to a stop over the grey line.

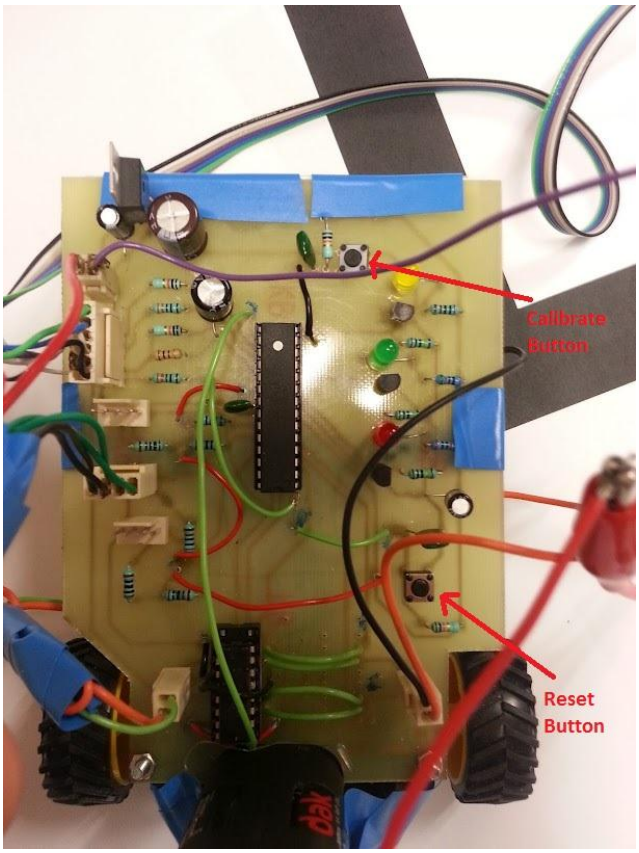
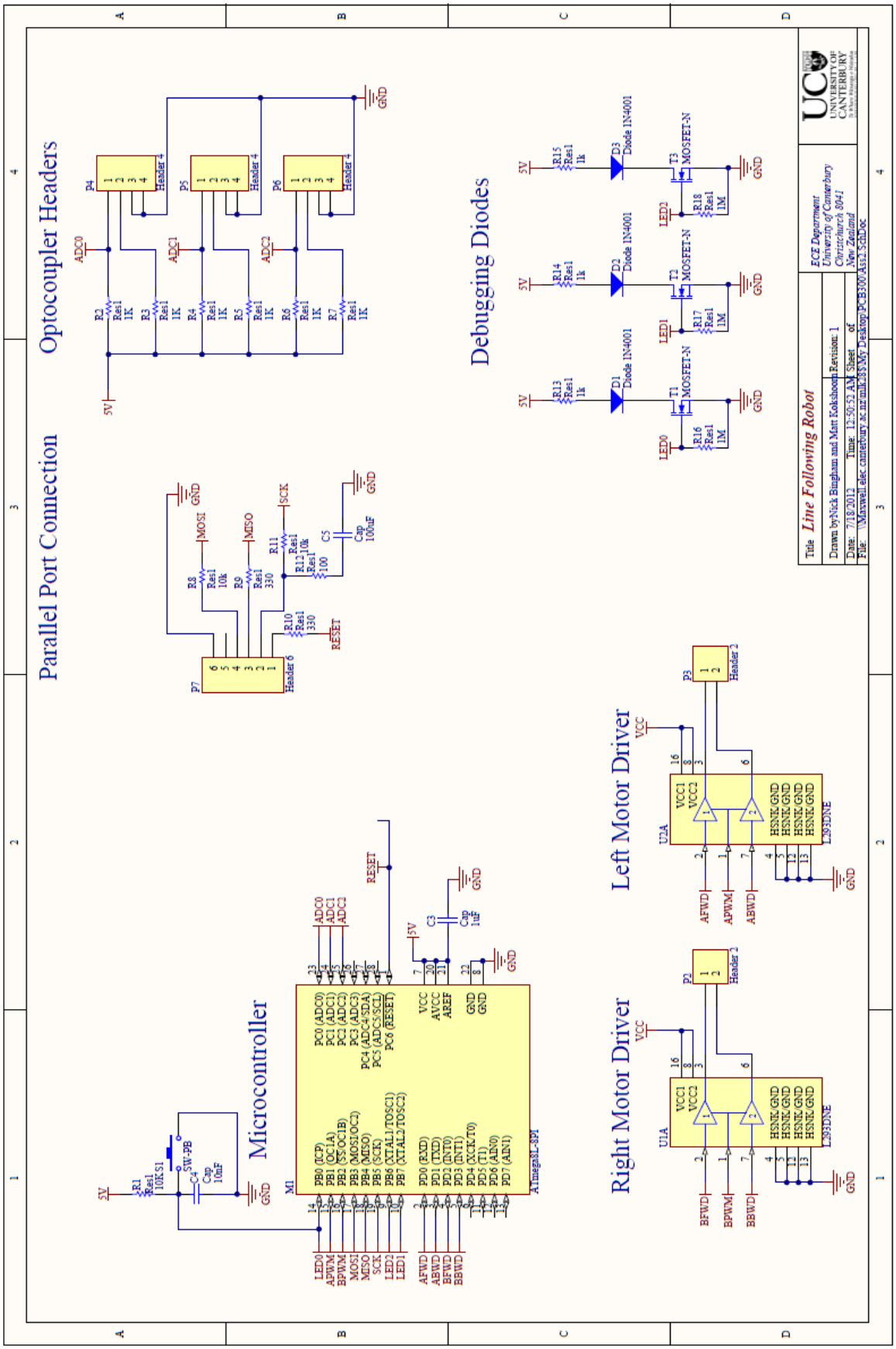
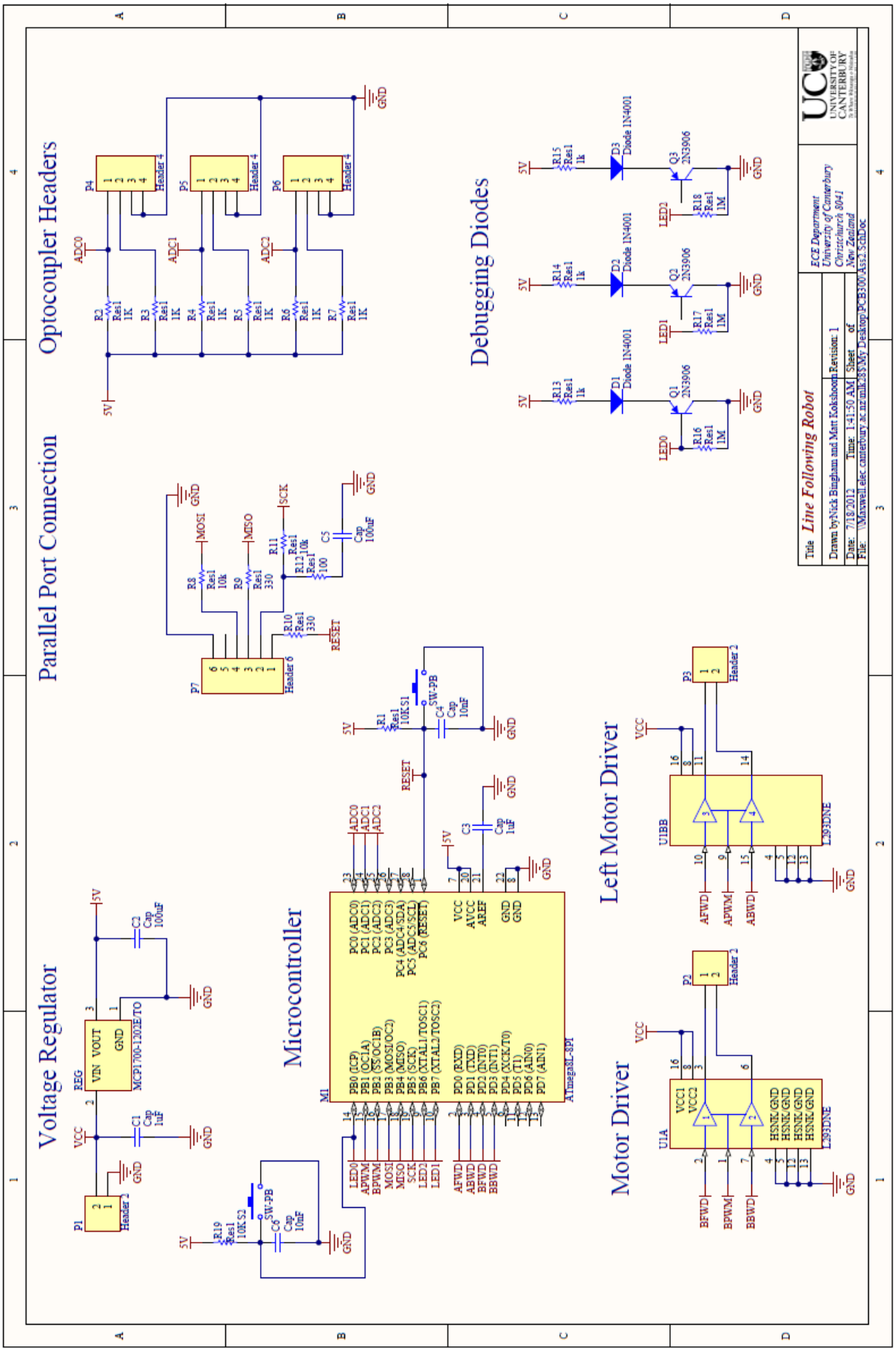


Figure 26 - Patterns of activated and inactivated sensors make intersections easier to recognise [7].

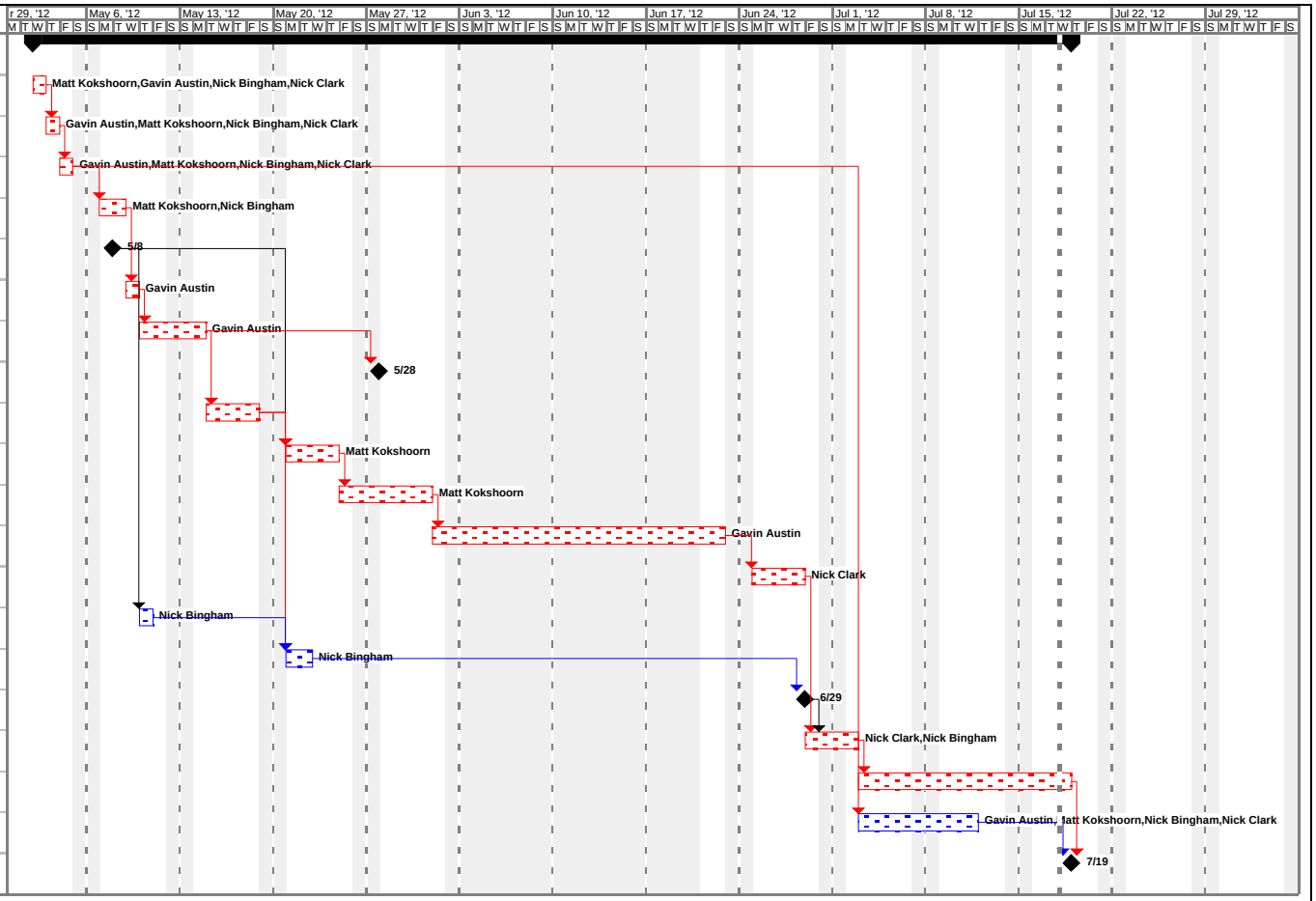
Original Schematic



Revised Schematic



ID	Task Name	Duration	Start	Finish	Predecessors
1	**Line Following Robot**	**43 days**	**May 2**	**July 19**	
2	Receive project specifications	1 day	May 2	May 2	
3	Choose and Submit Team	1 day	May 3	May 3	2
4	Initial Team meeting	1 day	May 4	May 4	3
5	Schematic Design	2 days	May 7	May 8	4
6	Receive Components	0 days	May 8	May 8	
7	Schematic Check	1 day	May 9	May 9	5
8	PCB Layout	3 days	May 10	May 14	7
9	PCB Submission deadline	0 days	May 28	May 28	8
10	PCB Construction	4 days	May 15	May 18	8
11	Soldering PCB	4 days	May 21	May 24	10,6
12	Initial Programming of micro	5 days	May 25	May 31	11
13	Develop API	3 days	June 1	June 22	12
14	Program Maze solving algorithm	4 days	June 25	June 28	13
15	Construct motor gear box	1 day	May 10	May 10	6
16	Construct Robot	2 days	May 21	May 22	10,15
17	Finish Programming Robot	0 days	June 29	June 29	16
18	Test Robot	2 days	June 29	July 2	14,17
19	Time Allowed for problems	12 days	July 3	July 18	18
20	Write report	7 days	July 3	July 11	4
21	Due Date	0 days	July 19	July 19	20,19



Project: Group Allocation
Date: July 18

Task | Split | Milestone | Project Summary | External Milestone
Project Guide: Critical Task | Progress | Summary | External Tasks | Deadline

```
1  /** led.h
2
3    @file Header providing abstraction functions for the LEDs.
4    @author Matt Kokshoorn
5
6  **/
7
8  #ifndef _LED_H_
9  #define _LED_H_
10
11 #include <avr/io.h>
12
13 #define YELLOW 0
14 #define RED 6
15 #define GREEN 7
16
17 /**
18   Initilises the pins to drive the LEDs as outputs.
19 **/
20 void led_init(void);
21
22 /**
23   Toggles the LED.
24   @param (int) Logic output position on PORTB.
25 **/
26 void led_toggle(
27   int led
28   );
29
30 /**
31   Turns the LED on.
32   @param (int) Logic output position on PORTB.
33 **/
34 void led_on(
35   int led
36   );
37
38 /**
39   Turns the LED off.
40   @param (int) Logic output position on PORTB.
41 **/
42 void led_off(
43   int led
44   );
45
46 /**
47   Tests all leds by entering an infinite loop toggling all LEDs sequentially.
48 **/
49 void led_test(void);
50
51 #endif
```

```
1  /** led.c
2
3    @file Abstracted functions to initilise, toggle and turn on/off LED's.
4    Also has a test led function to cycle through LED's to ensure functionality.
5    @author Matt Kokshoorn
6
7  **/
8
9  #include "led.h"
10
11 /**
12   Initilises the pins to drive the LEDs as outputs.
13 **/
14 void led_init(void)
15 {
16   DDRB|=(1<<YELLOW)|(1<<RED)|(1<<GREEN);
17   led_off(RED); led_off(GREEN); led_off(YELLOW);
18 }
19
20 /**
21   Toggles the LED.
22   @param (int) Logic output position on PORTB.
23 **/
24 void led_toggle(
25   int led
26   )
27 {
28   PORTB=PORTB^(1<<led);
29 }
30
31 /**
32   Turns the LED on.
33   @param (int) Logic output position on PORTB.
34 **/
35 void led_on(
36   int led
37   )
38 {
39   PORTB=PORTB&(~(1<<led));
40
41 }
42
43 /**
44   Turns the LED off.
45   @param (int) Logic output position on PORTB.
46 **/
47 void led_off(
48   int led
49   )
50 {
51   PORTB=PORTB|(1<<led);
52 }
53
54 /**
55   Tests all leds by entering an infinite loop toggling all LEDs sequentially.
56 **/
57 void led_test(void)
```

```
58 {
59   volatile long i;
60   volatile long j=0;
61   while(1) {
62
63     if(j==1) led_on(YELLOW);
64     if(j==2) led_on(GREEN);
65     if(j==3) led_on(RED);
66     if(j==4) led_off(YELLOW);
67     if(j==5) led_off(GREEN);
68     if(j==6) led_off(RED);
69     if(j==7) j=0;
70     j++;
71     for (i = 0; i < 1000; i++) continue; //delay loop
72   }
73
74 }
```

```
1  /** button.h
2
3    @file Header providing abstraction functions for the button.
4    @author Matt Kokshoorn
5
6  **/
7
8  #ifndef _BUTTON_H_
9  #define _BUTTON_H_
10
11 #include <avr/io.h>
12
13 #define BUTTON_BIT 0
14
15 /**
16   Function initilises pin for with button as an input.
17 **/
18 void button_init(void);
19
20 /**
21   Abstracted fucntion to check the state of the button.
22   @return  (char)  The button state 1 or 0.
23 **/
24 char button_pressed(void);
25
26 /**
27   Provides button debounce support.
28 **/
29 void button_debounce(void);
30
31 #endif
32
```

```
1   /** button.c
2
3     @file Abstrated fucntions to initilise, check the state of the button.
4     @author Matt Kokshoorn and Nick Bingham
5
6   **/
7
8   #include "button.h"
9
10  /**
11    Function initilises pin for with button as an input.
12  **/
13  void button_init(void)
14  {
15    DDRB=DDRB&(~(1<<BUTTON_BIT));
16  }
17
18
19  /**
20    Abstracted fucntion to check the state of the button.
21    @return  (char)  The button state 1 or 0.
22  **/
23  char button_pressed(void)
24  {
25    return !((PINB&(1<<BUTTON_BIT))>0);
26  }
27
28  /**
29    Provides button debounce support.
30  **/
31  void button_debounce(void)
32  {
33    uint8_t button_on_count = 255;
34    while (button_on_count) {
35      if (button_pressed()) {
36        button_on_count = 255;
37      }
38      else {
39        button_on_count--;
40      }
41    }
42  }
43
```

```
1  /** adc.h
2
3    @file Header providing abstraction functions for the ADC.
4    @author Matt Kokshoorn
5
6  **/
7
8  #ifndef _ADC_H_
9  #define _ADC_H_
10
11 #include <avr/io.h>
12 #include <stdint.h>
13 #include <avr/interrupt.h>
14 #include "adc.h"
15
16 #define BIT(x) (1 << (x))
17
18 #define ADC0 0
19 #define ADC1 1
20 #define ADC2 2
21 #define MUX_ADC0 (ADMUX & ~(BIT(MUX3) | BIT(MUX2) | BIT(MUX1) | BIT(MUX0) ) )
22 #define MUX_ADC1 ((ADMUX & ~(BIT(MUX3) | BIT(MUX2) | BIT(MUX1) | BIT(MUX0) ) ) |
   BIT(MUX0))
23 #define MUX_ADC2 ((ADMUX & ~(BIT(MUX3) | BIT(MUX2) | BIT(MUX1) | BIT(MUX0) ) ) |
   BIT(MUX1))
24
25 /**
26   Initilises the ADC for use and begins the corresponding ISR.
27 **/
28 void adc_init(void);
29
30 #endif
```

```
1  /** adc.c
2
3    @file Initilises the Analog to Digital Convertor.
4    @author Matt Kokshoorn
5
6  **/
7
8  #include "adc.h"
9
10 /**
11   Initilises the ADC for use and begins the corresponding ISR.
12 **/
13 void adc_init(void)
14 {
15   /* prescaler 8 so sample rate is 125 kHz */
16   ADCSRA |= BIT(ADPS1) | BIT(ADPS0);
17   /* Set reference as AVcc*/
18   ADMUX |= BIT(REFS0);
19   /* Turn ADC into 8 bit*/
20   ADMUX |= BIT(ADLAR);
21   /* ADC0 to be ADC input */
22   ADMUX = MUX_ADC0;
23   /* Set ADC to Free-Running mode */
24   ADCSRA |= BIT(ADFR);
25   /* Enable the ADC */
26   ADCSRA |= BIT(ADEN);
27   // Enable ADC Interrupt
28   ADCSRA |= BIT(ADIE);
29   // Enable Global Interrupts
30   sei();
31   /* Start the ADC measurements */
32   ADCSRA |= BIT(ADSC);
33 }
34
35
36
37
```

```
1  /** pwm.h
2
3    @file Header providing basic drivers for the PWM.
4    @author Matt Kokshoorn
5
6  **/
7
8
9  #ifndef _PWM_H_
10 #define _PWM_H_
11
12 #include <avr/io.h>
13
14 #define BIT(x) (1 << (x))
15 #define DUTY_CYCLE OCR2
16
17 /**
18   Initilise the PWM output to PB3.
19 **/
20 void pwm_init (void);
21
22 #endif
```

```
1  /** pwm.c
2
3    @file Provides basic drivers for the PWM.
4    @author Matt Kokshoorn
5
6  **/
7
8  #include "pwm.h"
9
10 /**
11   Initilise the PWM output to PB3.
12 **/
13 void pwm_init (void)
14 {
15   /* PB3 as output */
16   DDRB = BIT(DDB3);
17
18   /* Assign PWM to PB3 using Timer2 */
19   TCCR2 = BIT(WGM20) | BIT(WGM21)  /* Fast PWM mode */
20           | BIT(COM21)             /* Clear OC2 on Compare Match */
21           | BIT(CS21);             /* 1/8 prescale = 488 Hz */
22
23   OCR2 = 0x80;
24 }
```

```
1  /** motor.h
2
3    @file Header providing drivers and abstraction functions for the motor.
4    @author Matt Kokshoorn and Nick Bingham
5
6  **/
7
8  #include <avr/io.h>
9  #include "delay.h"
10
11 #define LEFT 0
12 #define RIGHT 1
13
14 #define FWD 0
15 #define BK 1
16
17 #define ON 1
18 #define OFF 0
19
20 #define RIGHT_FWD 0x04
21 #define LEFT_BK 0x01
22 #define RIGHT_BK 0x08
23 #define LEFT_FWD 0x02
24
25 /**
26   Initilises motor logic outputs.
27 **/
28 void motor_init(void);
29
30 /**
31   Function that changes the state of one of the motor outputs.
32   @param  (int)  The side of the motor to drive, either LEFT or RIGHT.
33   @param  (int)  The direction of the motor to drive, either FWD or BK.
34   @param  (int)  The value of the motor state, either OFF or ON.
35 **/
36 void motor_logic(
37   int side,
38   int dir,
39   int val
40   );
41
42 /**
43   Abstracted function to stop the motors.
44 **/
45 void motor_stop(void);
46
47 /**
48   Abstracted function to drive the motors in reverse.
49 **/
50 void motor_backward(void);
51
52 /**
53   Abstracted function to drive the motors forward.
54 **/
55 void motor_forward(void);
56
57 /**
```

```
58   Abstracted function to drive the motors in an arcing right turn.
59 **/
60 void motor_right(void);
61
62 /**
63   Abstracted function to drive the motors in an arcing left turn.
64 **/
65 void motor_left(void);
66
67 /**
68   Abstracted function to drive the motors in a pivoting right turn.
69 **/
70 void motor_pivot_right(void);
71
72 /**
73   Abstracted function to drive the motors in a pivoting left turn.
74 **/
75 void motor_pivot_left(void);
76
77 /**
78   Abstracted function to provide multiple turning components in one turn.
79   This function leaves active the last one of the modes used, in the order:
80     Stop > Forward > Turn > Pivot.
81   @param  (unsigned int)  Minimum time desired to turn.
82   @param  (unsigned int)  Time desired to pivot.
83   @param  (unsigned int)  Time desired to move forward.
84   @param  (unsigned int)  Time desired to stop.
85   @param  (unsigned int)  Direction of movement, either LEFT or RIGHT.
86 **/
87 void motor_turn(unsigned int turn_delay,
88   unsigned int pivot_delay,
89   unsigned int forward_delay,
90   unsigned int stop_delay,
91   unsigned int direction
92   );
93
94
```

```
1  /** motor.c
2
3    @file Provides basic drivers and abstracted functions for the motor.
4    @author Matt Kokshoorn and Nick Bingham
5
6  **/
7
8  #include "motor.h"
9
10 /**
11   Initilises motor logic outputs.
12 **/
13 void motor_init(void)
14 {
15       DDRD|=LEFT_FWD|RIGHT_FWD|LEFT_BK|RIGHT_BK;
16 }
17
18 /**
19   Function that changes the state of one of the motor outputs.
20   @param  (int)  The side of the motor to drive, either LEFT or RIGHT.
21   @param  (int)  The direction of the motor to drive, either FWD or BK.
22   @param  (int)  The value of the motor state, either OFF or ON.
23 **/
24 void motor_logic(
25   int side,
26   int dir,
27   int val
28   )
29 {
30   if(dir==FWD){
31     if (val==ON){
32       if (side==LEFT){
33         PORTD|=(1<<1);
34       }
35       else if(side==RIGHT){
36         PORTD=PORTD|RIGHT_FWD;
37       }
38     }
39     else if(val==OFF){
40       if (side==LEFT){
41         PORTD&=~(LEFT_FWD);
42       }
43       else if(side==RIGHT){
44         PORTD&=~(RIGHT_FWD);
45       }
46     }
47   }
48   else if(dir==BK){
49     if (val==ON){
50       if (side==LEFT){
51         PORTD|=LEFT_BK;
52       }
53       else if(side==RIGHT){
54         PORTD|=RIGHT_BK;
55       }
56     }
57     else if(val==OFF){
```

```
58        if (side==LEFT){
59          PORTD&=~(LEFT_BK);
60        }
61        else if(side==RIGHT){
62          PORTD&=~(RIGHT_BK);
63        }
64      }
65    }
66  }
67
68  /**
69    Abstracted function to stop the motors.
70  **/
71  void motor_stop(void)
72  {
73    motor_logic(LEFT,FWD,OFF);
74    motor_logic(RIGHT,FWD,OFF);
75    motor_logic(LEFT,BK,OFF);
76    motor_logic(RIGHT,BK,OFF);
77  }
78
79  /**
80    Abstracted function to drive the motors in reverse.
81  **/
82  void motor_backward(void)
83  {
84    motor_logic(LEFT,FWD,ON);
85    motor_logic(RIGHT,FWD,ON);
86    motor_logic(LEFT,BK,OFF);
87    motor_logic(RIGHT,BK,OFF);
88  }
89
90  /**
91    Abstracted function to drive the motors forward.
92  **/
93  void motor_forward(void)
94  {
95    motor_logic(LEFT,FWD,OFF);
96    motor_logic(RIGHT,FWD,OFF);
97    motor_logic(LEFT,BK,ON);
98    motor_logic(RIGHT,BK,ON);
99  }
100
101 /**
102   Abstracted function to drive the motors in an arcing right turn.
103 **/
104 void motor_right(void)
105 {
106   motor_logic(LEFT,FWD,OFF);
107   motor_logic(RIGHT,FWD,OFF);
108   motor_logic(LEFT,BK,ON);
109   motor_logic(RIGHT,BK,OFF);
110 }
111
112 /**
113   Abstracted function to drive the motors in an arcing left turn.
114 **/
```

```
115 void motor_left(void)
116 {
117   motor_logic(LEFT,FWD,OFF);
118   motor_logic(RIGHT,FWD,OFF);
119   motor_logic(LEFT,BK,OFF);
120   motor_logic(RIGHT,BK,ON);
121 }
122
123 /**
124   Abstracted function to drive the motors in a pivoting right turn.
125 **/
126 void motor_pivot_right(void)
127 {
128   motor_logic(LEFT,FWD,OFF);
129   motor_logic(RIGHT,FWD,ON);
130   motor_logic(LEFT,BK,ON);
131   motor_logic(RIGHT,BK,OFF);
132 }
133
134 /**
135   Abstracted function to drive the motors in a pivoting left turn.
136 **/
137 void motor_pivot_left(void)
138 {
139   motor_logic(LEFT,FWD,ON);
140   motor_logic(RIGHT,FWD,OFF);
141   motor_logic(LEFT,BK,OFF);
142   motor_logic(RIGHT,BK,ON);
143 }
144
145 /**
146   Abstracted function to provide multiple turning components in one turn.
147   This function leaves active the last one of the modes used, in the order:
148     Stop > Forward > Turn > Pivot.
149   @param  (unsigned int)  Minimum time desired to turn.
150   @param  (unsigned int)  Time desired to pivot.
151   @param  (unsigned int)  Time desired to move forward.
152   @param  (unsigned int)  Time desired to stop.
153   @param  (unsigned int)  Direction of movement, either LEFT or RIGHT.
154 **/
155 void motor_turn(unsigned int turn_delay,
156   unsigned int pivot_delay,
157   unsigned int forward_delay,
158   unsigned int stop_delay,
159   unsigned int direction
160   )
161 {
162   if (direction == LEFT){
163     motor_pivot_left();
164     delay_flat(pivot_delay);
165     if (turn_delay > 0) {
166       motor_left();
167       delay_flat(turn_delay);
168     }
169   }
170   else {
171     motor_pivot_right();
```

```
172     delay_flat(pivot_delay);
173     if (turn_delay > 0) {
174       motor_right();
175       delay_flat(turn_delay);
176     }
177   }
178   if (forward_delay > 0) {
179     motor_forward();
180     delay_flat(forward_delay);
181   }
182   if (stop_delay > 0) {
183     motor_stop();
184     delay_flat(stop_delay);
185   }
186 }
187
188
189
```

```
1  /** delay.h
2
3    @file Header providing abstracted delay functions.
4    @author Nick Bingham
5
6  **/
7
8  #ifndef _DELAY_H_
9  #define _DELAY_H_
10
11 /**
12   Delays the processor for a short period from 0 to 255 cycles.
13   @param  (unsigned long int)  Delay period in the range 0-255.
14 **/
15 void delay_flat(
16   volatile unsigned long int value
17   );
18
19 #endif
20
21
```

```
1  /** delay.c
2
3    @file Provides abstracted delay functions.
4    @author Nick Bingham
5
6  **/
7
8  #include "delay.h"
9
10 /**
11   Delays the processor for a short period from 0 to 255 cycles.
12   @param  (unsigned long int)  Delay period in the range 0-255.
13 **/
14 void delay_flat(
15   volatile unsigned long int value
16   )
17 {
18   while (value > 0) {
19     value--;
20   }
21 }
22
23
```

```
1  /** sensor.h
2
3    @file Header providing abstraction functions for the sensor.
4    @author Nick Bingham
5
6  **/
7
8  #ifndef _SENSOR_H_
9  #define _SENSOR_H_
10
11 #include "delay.h"
12 #include "button.h"
13 #include "led.h"
14 #include "motor.h"
15
16 #define CALIBRATE_COUNT 255
17 #define CALIBRATE_DELAY 100
18 #define DEBOUNCE_DELAY 120
19
20 #define GREY_LOWER grey_val-8
21 #define GREY_UPPER grey_val+8
22 #define GREY_MIN 12
23 #define ANTIGREY_MAX 1
24 #define ANTIGREY_PENALTY 3
25
26 #define BLACK 1
27 #define WHITE 0
28
29 /**
30   Determines the average value of the sensor over a short time interval.
31   @param  (unsigned int *)  Address of the sensor ISR value.
32   @return  (unsigned int)  Average sensor reading, in the range 0-255.
33 **/
34 unsigned int sensor_calibrate(
35   unsigned int *sensor_reading
36   );
37
38 /**
39   Determines the average value of the sensor over a short time interval.
40   For use in a continuous loop waiting for the button conditions to be
41   satisfied during the various stages of sensor calibration.
42   @param  (unsigned int *)  Address of the sensor ISR value.
43   @param  (unsigned int *)  Address at which the calibrated value is stored.
44   @param  (unsigned int *)  State of calibration, either 1 (ready) or 0.
45   @return  (unsigned int)  Average sensor reading, in the range 0-255.
46 **/
47 unsigned int sensor_init(
48   unsigned int *sensor_middle,
49   unsigned int *value,
50   unsigned int *ready
51   );
52
53 /**
54   Checks whether the sensor is currently over grey. If it is, this function
55   will increment counters and stop after several cycles, displaying the looped
56   led flashing function.
57   @param  (unsigned int)  The calibrated grey value.
```

```
58   @param  (unsigned int *)  Address holding the sensor ISR value.
59   @param  (unsigned int *)  Address storing the number of times that a grey
60   value has been reached recently.
61   @param  (unsigned int *)  Address storing the number of times that a non-grey
62   value has been reached recently.
63 **/
64 void sensor_grey_check(
65   unsigned int grey_val,
66   unsigned int *sensor_middle,
67   unsigned int *grey_count,
68   unsigned int *grey_anticount
69   );
70
71 /**
72   Checks whether the sensor has changed since it was last checked. If it is, the
73   value of the global 'middle' will be adjusted to accommodate this.
74   @param  (unsigned int)  The calibrated grey value.
75   @param  (unsigned int *)  Address holding the sensor ISR value.
76   @param  (unsigned int *)  Address holding the current state of the sensor,
77   either BLACK (1) or WHITE (0).
78 **/
79 void sensor_change_detect(
80   unsigned int grey_val,
81   unsigned int *sensor_middle,
82   unsigned int *middle
83   );
84
85 #endif
86
87
```

```
1  /** sensor.c
2
3    @file Provides abstracted functions for the sensor.
4    @author Nick Bingham
5
6  **/
7
8  #include "sensor.h"
9
10 unsigned int prev_mid = 1;
11 unsigned int curr_mid = 1;
12
13 /**
14   Determines the average value of the sensor over a short time interval.
15   @param  (unsigned int *)  Address of the sensor ISR value.
16   @return  (unsigned int)  Average sensor reading, in the range 0-255.
17 **/
18 unsigned int sensor_calibrate(
19   unsigned int *sensor_reading
20   )
21 {
22   unsigned long int calibrate_total = 0;
23   unsigned int count = 0;
24
25
26   while (count < CALIBRATE_COUNT) {
27     calibrate_total += *sensor_reading;
28     delay_flat(CALIBRATE_DELAY);
29     delay_flat(CALIBRATE_DELAY);
30     count++;
31   }
32   return (unsigned int)calibrate_total/CALIBRATE_COUNT;
33 }
34
35 /**
36   Determines the average value of the sensor over a short time interval.
37   For use in a continuous loop waiting for the button conditions to be
38   satisfied during the various stages of sensor calibration.
39   @param  (unsigned int *)  Address of the sensor ISR value.
40   @param  (unsigned int *)  Address at which the calibrated value is stored.
41   @param  (unsigned int *)  Address holding the state of calibration, either 1
42   (ready) or 0.
43   @return  (unsigned int)  Average sensor reading, in the range 0-255.
44 **/
45 unsigned int sensor_init(
46   unsigned int *sensor_middle,
47   unsigned int *value,
48   unsigned int *ready
49   )
50 {
51   unsigned int complete = 0;
52   if(button_pressed() && !*ready){
53     led_on(RED);
54     *ready = 1;
55     *value = sensor_calibrate(sensor_middle);
56     button_debounce();
57   }
```

```
58    else if (button_pressed()){
59      led_on(RED);
60      complete = 1;
61      button_debounce();
62      led_init();
63    }
64    delay_flat(DEBOUNCE_DELAY);
65    led_off(RED);
66    return complete;
67  }
68
69  /**
70    Checks whether the sensor is currently over grey. If it is, this function
71    will increment counters and stop after several cycles, displaying the looped
72    led flashing function.
73    @param  (unsigned int)  The calibrated grey value.
74    @param  (unsigned int *)  Address holding the sensor ISR value.
75    @param  (unsigned int *)  Address storing the number of times that a grey
76    value has been reached recently.
77    @param  (unsigned int *)  Address storing the number of times that a non-grey
78    value has been reached recently.
79  **/
80  void sensor_grey_check(
81    unsigned int grey_val,
82    unsigned int *sensor_middle,
83    unsigned int *grey_count,
84    unsigned int *grey_anticount
85    )
86  {
87    if (*sensor_middle > GREY_LOWER && *sensor_middle < GREY_UPPER) {
88      *grey_count += 1;
89      if (*grey_count > GREY_MIN) {
90        motor_stop();
91        led_test();
92      }
93    }
94    else {
95      *grey_anticount += 1;
96      if (*grey_anticount > ANTIGREY_MAX) {
97        if (*grey_count > (ANTIGREY_PENALTY+1)) {
98          *grey_count -= ANTIGREY_PENALTY;
99        }
100       else {
101         *grey_count = 0;
102       }
103       *grey_anticount = 0;
104     }
105   }
106 }
107
108 /**
109   Checks whether the sensor has changed since it was last checked. If it is,
      the
110   value of the global 'middle' will be adjusted to accommodate this.
111   @param  (unsigned int)  The calibrated grey value.
112   @param  (unsigned int *)  Address holding the sensor ISR value.
113   @param  (unsigned int *)  Address holding the current state of the sensor,
```

```
114   either BLACK (1) or WHITE (0).
115 **/
116 void sensor_change_detect(
117   unsigned int grey_val,
118   unsigned int *sensor_middle,
119   unsigned int *middle
120   )
121 {
122   prev_mid = curr_mid;
123   if (*sensor_middle > grey_val) {
124     curr_mid = BLACK;
125   }
126   else {
127     curr_mid = WHITE;
128   }
129   if (curr_mid == prev_mid){
130     *middle = curr_mid;
131   }
132 }
133
134
135
```

```
1  /** control.h
2
3    @file Header providing control functions for the robot.
4    @author Nick Bingham
5
6  **/
7
8  #ifndef _CONTROL_H_
9  #define _CONTROL_H_
10
11 #include "motor.h"
12 #include "sensor.h"
13
14 #define OUTSIDE_MAX 250
15 #define INSIDE_MAX 145
16 #define WOBBLE_LIMIT 5
17
18 #define OUTSIDE_TURN 2
19 #define OUTSIDE_PIVOT 0
20 #define OUTSIDE_FWD 15
21 #define OUTSIDE_STOP 1
22
23 #define INSIDE_TURN 0
24 #define INSIDE_PIVOT 30
25 #define INSIDE_FWD 0
26 #define INSIDE_STOP 0
27
28 /**
29   Function to move in an arcing forward direction at a predetermined rate. The
30   radius of the arc will depend on the current value (BLACK or WHITE) of the
31   sensor and also which side of the line is being followed.
32   @param  (unsigned int)  The side currently being followed, either LEFT or
33   RIGHT.
34   @param  (unsigned int)  The number of turns made on the current side of the
35   line up to that point.
36   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
37   @param  (unsigned int *)  The number of successive left turns.
38   @param  (unsigned int *)  The number of successive right turns.
39   @param  (unsigned int *)  The number of wobbles since a left or right turn.
40 **/
41 void control_forward(
42   unsigned int current_side,
43   unsigned int turn_count,
44   unsigned int middle,
45   unsigned int *turn_left_count,
46   unsigned int *turn_right_count,
47   unsigned int *wobble_count
48   );
49
50 /**
51   Function to stabilise the robot after crossing over a black/white
52   intersection. This compensates for the length of the previous turn.
53   @param  (unsigned int)  The side currently being followed, either LEFT or
54   RIGHT.
55   @param  (unsigned int)  Address holding the number of turns made on the
56   current side of the line up to that point.
57   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
```

```
58   @param  (unsigned int *)  Address holding the number of wobbles since a left
59   or right turn.
60 **/
61 void control_stabilise(
62   unsigned int current_side,
63   unsigned int *turn_count,
64   unsigned int middle,
65   unsigned int *wobble_count
66   );
67
68 /**
69   This function comes into play when the robot is stuck inside a loop. It causes
70   the robot to rotate and move to the other side of the line before continuing.
71   @param  (unsigned int)  The side currently being followed, either LEFT or
72   RIGHT.
73   @param  (unsigned int *)  Address holding the number of turns made on the
74   current side of the line up to that point.
75   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
76   @param  (unsigned int *)  Address recording which side to next travel to.
77 **/
78 void control_switch(
79   unsigned int current_side,
80   unsigned int *turn_count,
81   unsigned int middle,
82   unsigned int *move_direction
83   );
84
85 #endif
86
87
```

```
1  /** control.c
2
3    @file Provides control functions for the robot.
4    @author Nick Bingham
5
6  **/
7
8  #include "control.h"
9
10 unsigned int move_first = 1;
11
12 /**
13   Function to move in an arcing forward direction at a predetermined rate. The
14   radius of the arc will depend on the current value (BLACK or WHITE) of the
15   sensor and also which side of the line is being followed.
16   @param  (unsigned int)  The side currently being followed, either LEFT or
17   RIGHT.
18   @param  (unsigned int)  The number of turns made on the current side of the
19   line up to that point.
20   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
21   @param  (unsigned int *)  Address holding the number of consecutive left
     turns.
22   @param  (unsigned int *)  Address holding the number of consecutive right
     turns.
23   @param  (unsigned int *)  Address holding the number of wobbles since a left
24   or right turn.
25 **/
26 void control_forward(
27   unsigned int current_side,
28   unsigned int turn_count,
29   unsigned int middle,
30   unsigned int *turn_left_count,
31   unsigned int *turn_right_count,
32   unsigned int *wobble_count
33   )
34 {
35   unsigned int outside = current_side;
36   unsigned int inside = !current_side;
37   if (middle == WHITE){
38     motor_turn(OUTSIDE_TURN, OUTSIDE_PIVOT, OUTSIDE_FWD, OUTSIDE_STOP,
       outside);
39     if (turn_count == OUTSIDE_MAX && *wobble_count > WOBBLE_LIMIT) {
40       if (current_side == LEFT) {
41         *turn_right_count++;
42         *turn_left_count=0;
43         *wobble_count = 0;
44       }
45       else {
46         *turn_left_count++;
47         *turn_right_count = 0;
48         *wobble_count = 0;
49       }
50     }
51   }
52   else{
53     motor_turn(INSIDE_TURN, INSIDE_PIVOT, INSIDE_FWD, INSIDE_STOP, inside);
54     if (turn_count == INSIDE_MAX && *wobble_count > WOBBLE_LIMIT) {
```

```
55        if (current_side == RIGHT) {
56          *turn_right_count++;
57          *turn_left_count=0;
58          *wobble_count = 0;
59        }
60        else {
61          *turn_left_count++;
62          *turn_right_count = 0;
63          *wobble_count = 0;
64        }
65      }
66    }
67  }
68
69  /**
70    Function to stabilise the robot after crossing over a black/white
71    intersection. This compensates for the length of the previous turn.
72    @param  (unsigned int)  The side currently being followed, either LEFT or
73    RIGHT.
74    @param  (unsigned int *)  Address holding the number of turns made on the
75    current side of the line up to that point.
76    @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
77    @param  (unsigned int *)  Address holding the number of wobbles since a left
78    or right turn.
79  **/
80  void control_stabilise(
81    unsigned int current_side,
82    unsigned int *turn_count,
83    unsigned int middle,
84    unsigned int *wobble_count
85    )
86  {
87    unsigned int outside = current_side;
88    unsigned int inside = !current_side;
89    if (*turn_count > 100) {
90      *turn_count = 100;
91    }
92    *turn_count = (*turn_count/2);
93    while (*turn_count > 0) {
94      if (middle == WHITE){
95        motor_turn(0, (OUTSIDE_TURN+OUTSIDE_PIVOT), 0, 0, outside);
96      }
97      else{
98        motor_turn(0, (INSIDE_TURN+INSIDE_PIVOT), 0, 0, inside);
99      }
100     *turn_count -= 1;
101   }
102   *wobble_count++;
103   *turn_count = 0;
104 }
105
106 /**
107   This function comes into play when the robot is stuck inside a loop. It
      causes
108   the robot to rotate and move to the other side of the line before continuing.
109   @param  (unsigned int)  The side currently being followed, either LEFT or
110   RIGHT.
```

```
111   @param  (unsigned int *)  Address holding the number of turns made on the
112   current side of the line up to that point.
113   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
114   @param  (unsigned int *)  Address recording which side to next travel to.
115 **/
116 void control_switch(
117   unsigned int current_side,
118   unsigned int *turn_count,
119   unsigned int middle,
120   unsigned int *move_direction
121   )
122 {
123   unsigned int move_continue = 0;
124   unsigned int outside = current_side;
125   if (*turn_count > 140) {
126     *turn_count = 140;
127   }
128   *turn_count = (*turn_count/2);
129   while (*turn_count > 0 && move_first) {
130     motor_turn(0, (INSIDE_TURN+INSIDE_PIVOT), 0, 0, outside);
131     *turn_count -= 1;
132   }
133   move_first = 0;
134   if (middle == BLACK) {
135     motor_turn(0, 0, 1, 0, outside);
136   }
137   else {
138     move_continue = 1;
139   }
140   if (move_continue) {
141     move_first = 1;
142     *move_direction = !*move_direction;
143   }
144 }
145
146
```

```
1  /** main.c
2
3    @File Main function for the ENEL300 Assignment 2 2012.
4    @Author Matt Kokshoorn and Nick Bingham
5
6  **/
7
8  #include <avr/io.h>
9  #include <avr/interrupt.h>
10 #include "led.h"
11 #include "adc.h"
12 #include "pwm.h"
13 #include "motor.h"
14 #include "button.h"
15 #include "sensor.h"
16 #include "delay.h"
17 #include "control.h"
18
19 unsigned int choice = ADC0;
20 unsigned int sensor_left = 0;
21 unsigned int sensor_right = 0;
22 unsigned int sensor_middle = 0;
23
24 unsigned int grey_val = 0;
25 unsigned int grey_ready = 0;
26 unsigned int executing = 0;
27
28 unsigned int grey_count = 0;
29 unsigned int grey_anticount = 0;
30
31 unsigned int middle = 1;
32 unsigned int old_mid = 1;
33
34 unsigned int turn_count = 0;
35 unsigned int turn_left_count = 0;
36 unsigned int turn_right_count = 0;
37 unsigned int wobble_count = 0;
38 unsigned int move_direction = RIGHT;
39
40 int main (void){
41
42   adc_init();
43   pwm_init();
44   motor_init();
45   led_init();
46   button_init();
47   DUTY_CYCLE=250;  // 0-255
48
49   while(!executing){
50     executing = sensor_init(&sensor_middle, &grey_val, &grey_ready);
51   }
52
53   while(executing){
54     // Update sensor information.
55     sensor_grey_check(grey_val, &sensor_middle, &grey_count, &grey_anticount);
56     sensor_change_detect(grey_val, &sensor_middle, &middle);
57
```

```
58              // Execute one cycle of the algorithm.
59              if (turn_left_count > 4) {
60                // Respond to sensor information following right edge.
61                if (move_direction == RIGHT) {
62                  control_switch(LEFT, &turn_count, middle, &move_direction);
63        }
64                else if (middle != old_mid){
65          old_mid = middle;
66          control_stabilise(RIGHT, &turn_count, middle, &wobble_count);
67        }
68        else {
69          control_forward(RIGHT, turn_count, middle, &turn_left_count,
            &turn_right_count, &wobble_count);
70        }
71      }
72      else {
73        // Respond to sensor information following left edge.
74        if (move_direction == LEFT) {
75          control_switch(RIGHT, &turn_count, middle, &move_direction);
76        }
77        else if (middle != old_mid){
78          old_mid = middle;
79          control_stabilise(LEFT, &turn_count, middle, &wobble_count);
80        }
81        else {
82          control_forward(LEFT, turn_count, middle, &turn_left_count,
            &turn_right_count, &wobble_count);
83        }
84      }
85      turn_count++;
86
87      // Limit values.
88      if (turn_count > 250) {
89        turn_count = 250;
90      }
91      if (wobble_count > 10) {
92        wobble_count = 10;
93      }
94
95      // Binary counter for LEDs.
96      if (turn_left_count & (1<<0)) {
97        led_on(RED);
98      }
99      else {
100       led_off(RED);
101     }
102     if (turn_left_count & (1<<1)) {
103       led_on(GREEN);
104     }
105     else {
106       led_off(GREEN);
107     }
108     if (turn_left_count & (1<<2)) {
109       led_on(YELLOW);
110     }
111     else {
112       led_off(YELLOW);
```

```
113     }
114   }
115 }
116
117 ISR(ADC_vect){
118       // 0 ->2; 2->1; 1->0
119     if (choice == ADC0){
120     sensor_left = ADCH;
121     /* start measuring adc1 */
122     ADMUX = MUX_ADC1;
123     choice = ADC1;
124     }
125     else if (choice == ADC1){
126       sensor_right = ADCH;
127       /* start measuring adc2 */
128     ADMUX = MUX_ADC2;
129     choice = ADC2;
130     }
131     else if (choice == ADC2){
132        sensor_middle = ADCH;
133        /* start measuring adc0 */
134        ADMUX = MUX_ADC0;
135        choice = ADC0;
136     }
137 }
138
139
140
```