Rooter – A MicroMouse Maze Solving Robot

2006 - 2007

CS39440 Major Project

Bob Clough (rlc3@aber.ac.uk)
Supervisor: Dr. Dave Barnes (dpb@aber.ac.uk)
Cyfarwyddiadau

1. Darllenwch lawiwydr y myfyrywtr a y we i sicrhau eich bod yna ymwbyddol beth yw’i thafod. Darllenwch y Datganiad Gwaith Gwriddiol sod at arwyddo yng nghomel asech dych dde y dafael i odweud mai ei gwaith chi yw hwn.
2. Cysylltirch y dafael yma i’eh guwth ar ei Nghlyfiau trwy ddelynndio’r myfyrywtr o ddelynydd.
3. Cyflwynwch eich asemaid yn ôl i cyfarwyddiadau y daithwydd.

Datganiad Gwaith Gwriddiol

Trwy arwyddo’i uchod, yr wyf yna cadrinalchau ni:

Fy ngwraith gwriddiol yw hwn, en i haf ei fod wedi soddi yneglur fel arall.

Deallaf fod y gyswllt am lle-lacram ac arhyw ymddygiad anheg arall ni llun, ac y gallu arwain at gofion mawr a ni hyd yn wedded à dynnu garedig.

Yr wyf wedi darllen yr adranau am ymddygiad anheg yn Llywyrthu Arthreladau? Myfyrwyr ar adran hwn pertisodwyd ymhlwyf syfrofiadau Llywyrthu Myfymwyd yr Adran Cyfriifiadur.

Yr wyf yna efall rhoi’i lluniau dde a chyntaf i ddweurt.

Instructions

1. Please read the student handbook on the web to ensure that you are aware of the plagiarism issue. Read the Declaration of Originality below and sign the space provided at the top right of this page to say that the work is your own.
2. Fasten this page to your work using the stapler available from reception.
3. Please submit your assignment as directed by your lecturer.

Declaration of Originality

In signing above, I confirm that:

This submission is my own work, except where clearly indicated.

I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withdrawal of a degree.

I have read the sections on unfair practices in the Student’s Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.

I understand and agree to abide by the University’s regulations governing these issues.
Table of Contents

1  Introduction & Background Information ................................................. 6
   1.1  Project Introduction ......................................................................... 6
   1.1.1  What is MicroMouse? .................................................................. 6
   1.1.2  The MicroMouse maze ................................................................. 6
   1.1.3  The different types of MicroMouse robots .................................... 6
      1.1.3.1  Wall Follower ..................................................................... 6
      1.1.3.2  Maze Solvers ..................................................................... 6
   1.1.4  My project .................................................................................. 7
1.2  Background ....................................................................................... 7
   1.2.1  History of the competition ............................................................ 7
   1.2.2  Existing Solutions to the problem ............................................... 8
      1.2.2.1  Airat2 ............................................................................... 8
      1.2.2.2  Robo-IFer ................................................................. 8
      1.2.2.3  Mkit 900 ................................................................. 8
   1.2.3  Requirements ............................................................................ 8
2  Process model ....................................................................................... 11
   2.1  The Waterfall Model ....................................................................... 11
   2.2  Extreme Programming .................................................................... 12
   2.3  Incremental Programming ................................................................ 14
   2.4  My Methodology ........................................................................... 14
3  Design Decisions - Hardware ................................................................. 16
   3.1  The Chassis .................................................................................. 16
   3.2  Motors ......................................................................................... 16
   3.3  The Electronics ............................................................................ 17
   3.4  CPU ............................................................................................. 17
      3.4.1  Microcontroller or MicroComputer? ....................................... 17
      3.4.2  Which microcontroller? ......................................................... 18
   3.5  Board Design ............................................................................... 19
   3.6  Sensors ....................................................................................... 19
   3.7  PCB Design Software ................................................................... 20
4  Design Decisions – Software ................................................................. 23
   4.1  The Simulator ............................................................................... 23
   4.2  The Microcontroller code ............................................................... 23
5  Implementation ..................................................................................... 26
   5.1  Hardware ..................................................................................... 26
      5.1.1  Chassis ................................................................................. 26
      5.1.2  Electronics ........................................................................... 26
         5.1.2.1  Power regulation and distribution .................................... 26
         5.1.2.1.1  PDB Version 1 ......................................................... 26
         5.1.2.1.2  ATX PSU Power Adaptor .......................................... 27
         5.1.2.2  Power Distribution Board, Version 2 .......................... 28
      5.1.3  Stepper driver board ............................................................... 29
      5.1.4  Sensor input .......................................................................... 30
      5.1.5  Main Board .......................................................................... 30
      5.1.6  Integration ............................................................................. 32
      5.1.7  Development Tools ................................................................ 32
   5.2  The Simulator Software .................................................................. 33
      5.2.1  Data Structures ..................................................................... 33
         5.2.1.1  The Maze ................................................................. 33
Chapter 1:

Introduction & Project Background
1 Introduction & Background Information

1.1 Project Introduction

1.1.1 What is MicroMouse?
MicroMouse is a robotics competition where small, self-contained robots attempt to map, then solve and navigate a previously unseen maze.

The robots, or ‘mice’, have to find their way from a certain pre-chosen corner cell to one of the center four cells of the maze, in the fastest possible time. To achieve this, they are allowed to make as many 'runs' from the start cell to the finish cell as they can fit within a 10-minute timeslot.

1.1.2 The MicroMouse maze
A competition standard MicroMouse maze contains 16 cells north-south, and 16 cells east-west. Each maze cell is 180mm x 180mm including the walls, with a maximum space of 168mm gap in between walls. The walls are 12mm thick, and are painted white, with a red top. The floor of the maze is painted with a non-reflective black paint.

The walls are attached to the baseboard using pegs at the junctions between walls. The rules specify that each peg must have at least one wall adjoining it.

1.1.3 The different types of MicroMouse robots
There are two main ‘species’ of MicroMouse in the UK MicroMouse competition, both of which run in their own separate class. They are the Wall Follower and the Maze Solver.

1.1.3.1 Wall Follower
A Wall Follower is the simplest type of MicroMouse, and can often work effectively without any kind of control circuitry. In fact, some of the most effective Wall Followers in recent years have consisted of a single motor attached to a battery, set up in such a way that the mouse 'hugs' the wall.

Wall followers, as the name suggests, solve the maze by following the left hand wall until the center is reached. While this is a very simple way to find the way out of a maze, it is useless in most competition mazes, as they are likely to contain an 'island' in the middle of the maze, meaning a wall following mouse will follow the left hand wall, and end up back where it started without even touching the center cell!

Wall followers will usually only make a single ‘run’ in competition, as they mostly have no intelligence, so will not know to go any faster on successive runs.

1.1.3.2 Maze Solvers
Maze Solvers are usually vastly more complicated than wall following mice, containing one or more microcontrollers and multiple perceptive and
proprioceptive sensors. In competition, a maze solver will normally make multiple runs, the first few to create a map in memory of the maze’s layout, and then runs at increasing speeds to attempt to attain the fastest speeds possible.

Normally, a mouse will only continue mapping until it has discovered enough of the maze’s layout to find the shortest or fastest route, and then switch over to speed mode.

When in the second mode, the mouse will normally make a series of runs from the start to the finish cells, possibly down different routes, and at increasing speeds, until it either runs out of time, or crashes.

1.1.4 My project
For my project, I will be building a robot to run in the MicroMouse competition, including the chassis, control electronics, and the software for solving the maze.

During the project I should hopefully be able to attempt a few different maze solving and mapping algorithms, starting with a basic flood fill algorithm, then adding in additional features, such as allowing the robot to weight the path to decrease the number of corners that need to be navigated, and only searching the parts of the maze needed for the fastest / shortest run.

1.2 Background
1.2.1 History of the competition
When the MicroMouse competition was first run, in 1979 in New York, The goal was to get to the opposite corner of the maze from the one where you started. A Wall following mouse easily won this competition, and so in subsequent years the rules were changed to move the target to the center cells. This meant that an 'Island' could be created in the center of the maze, which would stop Wall followers from solving the maze.

In 1980 the first MicroMouse competition in Europe was held, There were 18 competitors in total, none of whom a managed to solve the maze.

Various competitions then started springing up around the world, including Japanese, Singaporean, American and European competitions.

The UK competitions are held each year at the MINOS MicroMouse conference at Easter, the UK MicroMouse championships (UKMM) at TIC in Birmingham during June, and RoboTIC, Also at TIC in late November.

During the project I will be attending RoboTIC ’06, MINOS ‘07 and UKMM ’07, as these competitions are a valuable resource in finding MicroMouse information, speaking to other mouse builders, and getting technical information and advice to improve my mouse.
1.2.2 Existing Solutions to the problem.
As MicroMouse is a popular competition in the robotics world, there are a few pre-built robots available to purchase from outside sources

1.2.2.1 Airat2
The Airat2 is a complete MicroMouse chassis + control circuitry + SDK kit, available from MicroRobot. It is based around an 8051 microprocessor, and uses a pair of stepper motors. It uses Infra red sensors for detecting walls, and is programmed over an RS232 connection, using a custom SDK.

The Airat2 is a good platform, but is very expensive for what it does, costing over £400 at ActiveRobots, MicroBot's UK distributor.

1.2.2.2 Robo-lefter
The Robo-lefter is another MicroMouse designed by MicroRobot. It is based around an Atmel Atmega103 microcontroller, and uses a dual DC gearbox and Infra red sensors. It is programmed using a parallel port dongle, from the Atmel AVR studio software.

Robo-lefter is designed to be a cheaper alternative to the Airat2 above and therefore does not have as much CPU power, and is useless for maze solving in its base form due to the lack of any kind of motor odometry, which means it cannot be used accurately enough to be used as a solver. On the plus side, it costs around 1/4 of the price, at £100.

1.2.2.3 Mkit 900
The mkit 900 is a MicroMouse sold by Alpha Innovations, a Korean company. This mouse uses an NEC 78310A microcontroller. Its motors are stepper motors, and like the AiRat2, it uses infrared sensors.

It is largely comparable to the AiRat2 in features, size, and price, costing $700 before shipping and import taxes.

1.2.3 Requirements
The Requirements for the robot are as follows:
   1. Must cost less than £150 in total.
2. Must be able to map and solve a MicroMouse maze.
3. Must fit within, and be able to turn within, a single MicroMouse maze cell.
4. Must know where in the maze it is at any time.
5. Must be simple to reprogram for use in other tasks.
6. Must be able to run the maze multiple times on one battery charge.
7. Must be accurate when moving.
8. Must be able to sense the walls around it without touching them.
Chapter 2:

Process Model
2 Process model

For the project, I have evaluated three different programming methodologies for possible usage. As the project is hardware and software based, I will probably not be able to use a software development methodology wholesale, and therefore will probably take parts from different methodologies.

For my process model I first evaluated a formal methodology, The Waterfall Model, and an Agile methodology, Extreme Programming.

2.1 The Waterfall Model

The process behind the waterfall model is a simple one. The project is split up into a series of different sections, usually including:

- A requirements specification, in which all the requirements of the project are worked out between the customer and the software developers.
- A design document, in which the software to be built has its entire design sorted out and specified.
- An implementation, where the software is written as laid down in the design document.
- Integration, where pieces of software written in the implementation stage are brought together into a single large software package.
- Testing, where the software package produced is tested for basic functionality, and checked against the requirements specification and design to make sure it meets all the specifications.
- Lastly comes installation and maintenance, where the program is brought on-site, and set up. Afterwards it will be maintained and modified for as long as it is still in usage.

The name waterfall model comes from the fact that when using the model, one section is fully completed before the next section is started, one part drops into the next, as in a waterfall. Development continues in this manner until the software is completed, with no backtracking.
The main arguments for the waterfall model are that if each stage is done right, and the requirements are all set out perfectly, costly changes in the software later on in the development can be avoided, because all possible problems have been anticipated and taken care of.

Of course, on the flipside of this argument is the fact that if something is missed in the requirements specification, or not set out clearly enough in the design, it can be misinterpreted, or completely missed by the development team, and lead to large costs when noticed later in the development process, or even after installation!

### 2.2 Extreme Programming

Extreme programming is a relatively new discipline in the programming world. It emphasises constant change and high levels of communication between the programming team and the customer. It is based upon 12 principals:

1. **System Metaphor** - XP is based around a 'system metaphor'. This metaphor describes the system, as an object in the real world. An example of this would be a desktop, which can contain files, writing implements (Word processors / pens), pictures etc.
2. **The Planning Game** - During the planning session, requirements from the customer are broken down into smaller requirements, and a time required estimate is given to each.
3. **A Simple Design** - XP is designed around simplicity, and favours less code that does the job over more code that does the job, plus 20 other things that weren’t asked for. The design is the bare minimum needed for the coding, as anything that isn’t covered in the design can always be asked of the onsite customer.
4. **Pair Programming** - pair programming is one of the methods XP uses to
spread knowledge around the programming team. If programmers are in pairs, they will cross-train each other.

5. Test Driven Development - TDD is a development method where code is written to automate the testing process. Then just enough program code is written to pass the tests.

6. Small Releases - As the development continues, small releases of working programs are made, which can then be used by the customer to check they have been specified correctly.

7. Continuous Integration - Code is integrated as the project goes along. This helps find any places where the design is not right early on in the process, as different code modules will not work together as expected.

8. Coding Standards - Coding standards are rigidly adhered to, to make sure all the code is maintainable by any member of the team.

9. Collective Code Ownership - No one person owns a certain part of the code. If a bug is being caused by a piece of code a developer has not written, he can go in and fix the bug without

10. Design Malleability - At any time during the programming stages, the design can be modified, to fit in new features, or modify existing features' specifications.

11. On-site customer - XP specifies that there should be a customer on site at all times throughout the development process, as it makes communication between the customer and the development team much easier.

12. Sustainable development - Not going for 'burnout' on a single part of a project. For example, if developer overtime is needed one week for a release, they will not be forced to do overtime the week after.

The positives of extreme programming mainly center around the fact that it is very flexible when it comes to changes in the project, and that it encourages communication between all parties in the development process.

The negatives mainly center around the fact most of the knowledge about the project is stored in the developer’s brains, so development cannot easily be shelved for a later time, as it could if there were reams of paper about the project in existence.

Figure 5: The Extreme programming development lifecycle. Taken from the XP website.
After looking at these two extremes (no pun intended) of the process model spectrum, I decided neither was quite right for my project, and so decided to look at something more in the middle, Incremental Programming.

### 2.3 Incremental Programming

Incremental programming is a development method somewhere in between the Waterfall Method and Extreme Programming. Its main focus is on making many revisions of the software, each one getting closer and closer to the customer’s specifications.

When the project is started, the customer will outline the basic specifications of the project, and the development team will work on getting a first increment running. After this is done, both parties will have a meeting to discuss the next increment, and the process is repeated until the project is finished.

![Figure 6: The Iterative Development Model](image)

After each iteration, the customer takes the release and uses it to test the functionality. After he has done evaluating it, he will go back and modify the design to fix any problems he has found during the evaluation, and then the programming team will continue development with the new design.

### 2.4 My Methodology

For my project I have chosen to take ideas from multiple methodologies, and use an incremental, evolutionary approach.

Electronic subsystems will be designed, prototyped, and tested on a breadboard, and once completed will be rebuilt and refactored within a piece of PC software. This software will then be used to design a Printed Circuit Board, which will then be made, soldered and re-tested.

The software will be written in a similar manner, getting the first version of the software designed alongside the hardware, and incrementally improving upon it as the project progresses.
Chapter 3:

Design Decisions – Hardware
3 Design Decisions - Hardware

None of the robots I have documented above fit the requirements I have set out. The closest is the Robo-lefter, but this is difficult to repurpose, and has a low accuracy on the drivetrain. Because of this, I will be building my own robot.

I will need to take care of three different parts to build a complete robot:
- The Chassis
- The Electronics, Including sensors, motors and processor.
- The Software

3.1 The Chassis

There are two obvious methods to build the chassis. The first is to go with a PCB-based design, where the electronics of the robot support the motors and wheels. The second is to build a chassis around the motors, using a metal or plastic construction material. As the robot needs to be durable, I personally feel that the second method would be better.

3.2 Motors

To move the robot, I’ll need a motor of some type. There are three choices for motors, DC motors, Servomotors, and Stepper motors.

A DC Motor’s rotation direction depends on the direction of the voltage applied to them. Likewise, their speed depends on the level of voltage applied. DC motors often run very fast, and need a gearbox when used in robotics situations.

A DC motor would be driven either through an arrangement of transistors known as a H-bridge, or using a dedicated motor driver IC.

A servomotor is a small, self-contained system comprising of a motor, gearbox, and control electronics. They are very simple to drive, requiring only a 4-8V dc input, a ground connection, and a PWM (Pulse width modulation) pulse every 20uS. The width of the pulse tells the servo what position to rotate to, from 0˚ to 180˚ degrees.

Unfortunately, servomotors are limited in that they are only designed to turn 180˚, so need modifying to turn the full 360˚ needed. This modification usually involves opening up the servo, removing a small plastic lug from one of the gears that stops it turning,
and replacing the potentiometer used to detect where the motor currently is with a pair of resistors.

A stepper motor is driven by activating its coils in a certain sequence. It does not spin continuously like a DC motor, but is locked in place one ‘step’ further forwards every time the sequence is changed. Stepper motors are very strong, and do not need additional gearing to get them to sensible speeds.

Electronically, a stepper motor needs a lot more circuitry. A high-power driver chip, or a dual h-bridge is needed, along with a method of sequencing the output.

I have decided to use Stepper motors for the robot, as their strength and accuracy fits perfectly with Requirement 7.

3.3 The Electronics

3.4 CPU

3.4.1 MicroController or MicroComputer?
In the world of embedded CPUs, there are 2 main groups of chips, the 32 bit microcomputer chips, and 8 bit microcontroller chips.

Microcomputers normally require external RAM and storage, alongside the main processor chip. Some examples of commonly used 32bit chips are the 486 processor, as commonly used in early-90’s computers, now repurposed as everything from washing-machine controllers to POS terminals in shops, and ARM and AVR32 architecture chips, as commonly used in portable computing applications, such as PDAs and mobile phones. Their speeds normally vary between 50 and 400 MHz, giving an abundance of power for MicroMouse. Microcomputers use a comparatively large amount of power, and as they need external support circuitry normally need larger PCBs. They can be difficult to prototype with, as most are in tiny surface mount packages, needing expensive SMD soldering apparatus.

Microcontrollers are normally smaller, self-contained chips, containing a small amount of on-die memory and storage space. They normally run in the range of 1 – 50 MHz, and contain a plethora of onboard peripherals, such as Analog to Digital Converters and onboard communication hardware. Microcontrollers are used in practically every small electronic device nowadays, due to their small size, repurposability, and ease of upgrade compared to traditional, hardwired circuitry. They are available in a range of different packages, from breadboardable PDIP, to tiny surface mount BGA and QTFN.

For my MicroMouse I have chosen to use microcontrollers throughout, as one of my specifications calls for low power usage, and the algorithms I will be
using do not require a huge amount of CPU power.

3.4.2 Which microcontroller?
The three different microcontroller architectures I have evaluated for this project are: Atmel’s AVR, Microchip’s PIC, and the 8051 architecture, available from most microcontroller manufacturers. The things I am looking for in a microcontroller are re-programmability, in-circuit programming ability, multiple ADCs, availability of free / low-cost C compilers, and price of development tools.

The 8051 Architecture is a well-supported architecture, with many different compilers and programmers available. It is a very widespread chip type, with most microcontroller manufacturers having their own version. Herein lies the main problem with the architecture though, as each manufacturer adds parts to their lines with additional hardware such as ADCs and Serial Transceivers, their parts have moved further away from a ‘basic’ 8051, and hence code written for one manufacturer’s 8051 needs a lot of work when moving to a different manufacturer’s parts. As there has been over 30 years of this type of development, some 8051s have little to no resemblance to each other!

A company called General Instruments developed early PIC processors in the late 1970s as an I/O adaptor for one of their new microchips. Since then they have been built and expanded upon, and the entire range has moved nearly completely over to reprogrammable flash-based chips. Additional onboard peripherals have been added over time, leaving microchip with a large catalogue of PICs, with a range of options available for any purpose.

There is a range of development tools available for PICs, including C compilers such as CC5X and PICC, in circuit debuggers and simulators. The main problem with the PIC architecture comes here though, as the C compilers for the PIC are mainly designed for companies, and therefore can be quite expensive. The only free PIC compiler I have found is SDCC, which is a relatively new compiler, and is not yet stable enough to use.

The AVR architecture was originally developed by two students at the Norwegian Institute of Technology. It was later bought and further developed by Atmel Norway.

The entire range of AVR microcontrollers support in circuit programming, and are powerful, with parts from 1-20 MIPS (Million Instructions Per Second) available. A majority of the microcontrollers available in the AVR range have onboard peripherals, such as ADCs, USART Serial Transceivers, and PWM channels, which would allow me to use.

In terms of programming tools available, there are a wide range of commercial compilers such as the Bascom AVR BASIC compiler and the Codevision AVR C compiler, and also a large number of free compilers, such as the Atmel-supported AVR-GCC C compiler, PyMite Python Interpreter and NanoVM Java interpreter.

After weighing up the pros and cons of each architecture, I have decided to go...
with the AVR solution.

3.5 Board Design

I have chosen to split the modules used into separate components, each of which will have its own separate purpose. This will help ease the problems of parts not working, as anything that does not work in the first revision can easily be switched out with an improved design. I have decided to split the boards into 4 separate subcomponents:

The Power Board – The power board will take its power from a battery, and provide a regulated 3.3v and 5v output to the rest of the circuitry.

The Sensor Board – This board will take in a 5v and ground input, and an on / off signal for each sensor. When activated, each sensor will output an analog signal representing the current distance from a surface.

The Stepper Driver Board – This board will take in step & direction signals for each motor and output the signals needed to control the stepper motors.

The Main Board – This will bring all the other boards together, and do all the controlling of the mouse.

The specific microcontroller parts I have chosen are the ATtiny2313, a 20-pin 8MIPS part for the stepper driver board, and an ATmega16, a 40 pin 16 MIPS for the main board.

The ATtiny2313 was chosen because it is a small part, it has two external interrupt pins, and has just enough IO pins to drive two motors (4 + 4) and read the motor inputs (2 + 2), and allow serial communications for debugging purposes if needed.

The ATmega16 was chosen due to its high speed, and number of onboard analogue inputs. These analogue to digital converters (ADCs) have a 10-bit accuracy, so are perfect for reading the inputs from the sensor board. It has space for a serial port for ease of debugging, as well as loads of spare IO pins for future expansion. It is also pin compatible with the faster Atmega32 part, which provides me with a future upgrade path if I find myself hitting the limits of the microcontroller.

3.6 Sensors

Requirement 8 states that the robot must be able to sense the walls around it without touching them. This gives me two main options, Infrared and Ultrasonic sensing. There are also other, higher cost (both money-wise and in CPU time) options, an example of which would be CCD cameras.

IR sensors are commonly used in small robotics projects because of their durability and ease of use. The distance to the wall is often measured using a phototransistor, which will give a greater or lower resistance depending on the
amount of light being reflected from the surface. This can then be either read using an analogue input on the microcontroller, or run through an Operational Amplifier to provide a logic output. Also available are small devices that output a logic level (TTL) output when the IR level passes a certain threshold, meaning the OpAmp circuit can be done without.

Infrared has problems with interference from strong light sources, such as overhead lighting or sunlight. This can be overcome quite easily by shielding the sensors, or modulating the output, and filtering out any frequencies outside the modulated frequency picked up by the sensors.

Ultrasonic sensors are not used as often as infrared sensors, because the have some problems. To measure a distance using ultrasonic sensors, you send out a pulse of sound, and then measure the amount of time taken for that sounds wave to echo back to your sensor. Using this time, you can work out a distance pretty accurately.

The problems with ultrasonic sensors include crosstalk, where one sensor will pick up an echo from another, and the fact the sensor has to be at an angle of 90° from the surface to get a good reading.

I have decided to use Infra Red sensing on my Robot, as its ease of use and overall simplicity put it above Ultrasonics.

3.7 PCB Design Software

After deciding what hardware I would use for my project, I had to decide on the software I would use to design and build the Printed Circuit Boards. I will be using a piece of software from cadsoft, a German company, called Eagle.

Eagle is a cross-platform schematic and board design program. It allows the user to enter their schematic, and then convert it to a PCB design, using its built-in autorouter tools.
Figure 10: A screenshot of Eagle, showing one of the example projects.
Chapter 4:

Design Decisions – Software
4 Design Decisions – Software

4.1 The Simulator

The MicroMouse simulator is the first piece of software I will write. It has a few specifications that will help me when writing the software for the MicroMouse:

1. Must be able to take in a maze in the standard .ma format
2. Must be able to perform a flood fill on a maze, and display the flooded maze
3. Must be able to take the flooded maze, and work out the shortest route and display it.
4. Must work in a cross-platform manner

For this task I first had to choose a language. I evaluated three languages, Java, C#, and Ruby.

Java is an object-oriented language created by Sun Microsystems. It is strongly typed, and distributed under a GPL license. Its main advantage is its large collection of standard libraries, and its cross platform ability.

C# is a C-style language designed by Microsoft. It is interpreted, by the .NET common language runtime (CLR). In its original form it was windows only, but due to Microsoft releasing the language specifications, there is a project running to implement a CLR for Linux, named Mono, unfortunately, it does not implement any of the UI libraries, so new GUIs would need implementing for each platform.

Ruby is a heavily object-oriented programming language. It is completely open source, and cross platform. It is often described as a cross between Python and Perl. It uses the Tk user interface, and so can be displayed on any machine with the Tk libraries installed.

I have chosen to use Ruby for my simulator, in the end the choice came down to Ruby or Java, and as I prefer to program in Ruby, it was chosen over Java.

4.2 The Microcontroller code

First of all, I need to decide whether I will program my microcontrollers in a high level, compiled language, or using assembly code. To help Decide, I made a list of Pros and Cons for each:

High Level Language
Pros: Much easier to visualise what is going on, easier to debug.
If the compiler is good, there can be little noticeable difference in speed.
A high level language offers improved portability, if it needs to run on a different uC, you can often just change the target and recompile.

Assembly Code
Pros: Well written Assembly code can be lightning fast.
Timing is more predictable, as you can see exactly how many clock cycles are used per function.
If space is an issue, assembler code is normally smaller and more efficient than a compiled language.
Cons:
A programmer can never be sure what his compiler is doing behind his back in the way of optimisation, this can lead to weird bugs in correct-looking code.

Cons:
ASM can be difficult to learn for a programmer who is used to high-level languages, poking registers and moving memory contents before modifying them make it seem like an alien language! ASM is harder to read than most compiled languages.

After weighing up the pros and cons of each, I have chosen to use a high level language in my project, mainly because timing isn't 100% important, and I believe the ease of debugging the code will outweigh any speed advantages given by assembler, and as the atmega16, where the majority of the software will reside, has 16kb of flash memory, space is not an issue.
Chapter 5:

Implementation
5 Implementation

5.1 Hardware

5.1.1 Chassis
The Chassis design is a simple enough concept. It is all based around the stepper motors, which are standard NEMA 17 motors, chosen for their small size and low cost.

The chassis is then built around the two motors placed back-to-back, with the mounting holes on the motors used to hold the whole thing together.

The wheels are made of foam rubber, and are designed for Radio controlled planes. They were chosen due to their low weight and high friction in between the wheel and the maze floor.

As stepper motors can provide their full torque at low speeds, I was able to attach the wheels directly to the motor shafts. To do this, I simply drilled out the central hole of the wheel to 4mm, and pushed them on to the shaft.

Attaching boards to the chassis was the next problem I came across, they had to be attached well, but able to be easily removed for modifications and replacements. Eventually, I hit upon the idea of Velcro, and covered the top and insides of the chassis in a layer of 'soft' Velcro. After that, any board that needed mounting simply needed the 'hard' side of the Velcro attaching to fix them to the robot.

5.1.2 Electronics

5.1.2.1 Power regulation and distribution

5.1.2.1.1 PDB Version 1
The power distribution board was the first board to be designed and built, because it was needed to run all the other boards!

This board takes the 6v input from the battery, and provides a regulated 5V supply to run the electronics, and a regulated 3.3v supply to run the motors.

For the board I chose to use an LM1084 3.3V 5A regulator to provide the 3.3v supply, and an LM2940 5V low dropout regulator.
regulator to provide the 5v supply.

A low dropout regulator was chosen for the 5V supply because the battery’s nominal voltage is only 6V, at which a normal regulator such as the commonly used 7805 5V Regulator wouldn’t be able to provide the full 5V needed by my circuitry.

![Figure 13 – The PDB Schematic](image)

5.1.2.1.2 ATX PSU Power Adaptor

Due to the annoyance of constantly recharging the battery while writing software for rooter, I decided that a mains power adaptor was needed.

I originally looked into using a ‘wall-wart’ type power block for testing, but it was impossible to get one that could source the ~4A needed to run both the electronics and the motors in a small size for a sensible price.

![Figure 14 – The ATX Adaptor](image)

Then I realised that there is an easy to use, cheap and readily available power supply I already had, an ATX power supply.

Most modern ATX power supplies use switching regulators to function, and therefore need a small load attached to the output at all times while active. As I will have the robot’s electronics and motors attached to the output, this will pull enough power to keep the output stable.

After there is sufficient load placed on the line, to activate the power supply, we simply have to connect the 0V line to the Green (POWER-ON) line.
To build my adaptor, I used an ATX power connector removed from an old PC motherboard, and wired up the correct power lines to a power plug as used on rooter.

The Cables connected were Orange to 3.3V, Red to 5V, black to 0V (ground). Then the green and black wires were connected through a large toggle switch, and an LED was attached to both the 5v line (through a 330R resistor), and to the 3.3V line. This is simply to let me see that the power is turned on, and providing the correct voltages.

5.1.2.2 **Power Distribution Board, Version 2**

Unfortunately, I started having some major problems with my original power distribution board, mainly due to the large size of the heatsink needed to run the 3.3v regulator at 3 amps, and the incredible amount of heat being generated.

After some deliberation, and internet research, I decided on a regulator module from TI, the PTH08T231W. This is a switching regulator, which can provide up to 6A current, while only dissipating 1W of heat, which is an amazingly low amount. According to the TI datasheet (see appendices), it is between 80% and 95% efficient, a far cry from the 50% to 60% efficiency of the original LM1084 regulator.

![Figure 15 – The PDB, version 2.0](image)

![Figure 16 – Schematic for PDB version 2.0](image)
5.1.3 Stepper driver board

The stepper motor driver board was built and programmed on a breadboard.

During development, I ran the microcontroller alone on the breadboard, and used LEDs to provide visual feedback of the current output state of the motors.

This allowed me to work out the necessary maths to get the output sequence perfect. Once this was done, I linked the motors up through a UDN2559 High current driver chip, and the motors worked!

After getting the software built and hardware designed, I designed a PCB in Eagle and soldered it up. I plugged it into the power distribution board, and tested it using a push-button switch, and it worked first time!

As the stepper driver board worked as expected, no further versions were deemed necessary.
5.1.4 Sensor input

After doing some research on OpAmp circuitry to increase the voltage range given out by my sensors, I decided that they were unnecessary, as I was able to get a smooth enough input simply using a potential divider with a resistor.

This means my sensor boards were relatively simple, consisting of an IR LED / Sensor pair and two resistors for each sensor.

After building evolution 1 of the sensor board, I had a problem. None of the IR LEDs were working, evidenced by my looking at them through my digital camera’s viewfinder. An additional problem with this evolution was the centred connection header, which was interfering with the stepper driver board’s input above it. Thus, a second Evolution was deemed necessary.

For the second evolution, I took the schematic file from v1, and switched around the LED’s input legs. Then I built a new board design based upon this schematic, making sure the header was now in the correct place, and adding a green LED to show that the board was powered.

5.1.5 Main Board

The main board of my robot is designed around an atmega16 microcontroller.
As this microcontroller has 8 analog input channels, and over 30 digital I/O channels, I was able to give this board a lot of expandability.

As I had the analog inputs free, I decided to add the header for a second sensor board, to allow one to be added to the back of the robot. This would allow the entire robot to be reversible, and save valuable seconds in not having to turn around before backing out of dead ends etc.

This board also contains a 4 pin connector for the stepper drive board, a 5 pin connector for the rs232 connector (including ground, 5v, and an additional I/O pin), and a 10 pin connector for additional debugging boards to be connected, comprising of 8 I/O lines, +5v, and 0v.

Also added to this board was an in-circuit programming socket, to allow the atmega to be programmed without removing it from its socket, thus saving time during the software development phase, and making the chip less likely to get damaged during constant removal and reinsertion from the socket.

![Figure 21 - The Mainboard Schematic](image)

As the Mainboard worked as expected, no more versions were deemed necessary.
5.1.6 Integration
After all the boards were completed, the hardware integration took place. In its most basic form, all this took was attaching the various boards to the chassis using Velcro, and connecting wires in between them. After that, software was required to bring all the parts together.

5.1.7 Development Tools
Unfortunately, these boards would be useless without some way of accessing them. To program them, I decided to buy a pre-made USB in circuit programmer from tuxgraphics.org. I chose this over a cheaper serial or parallel port based option because none of the computers I have for development have any legacy ports, and USB devices can be plugged into all modern computers with no problem.

After buying the programmer, I realised I also needed a method of connecting the Mainboard to a PC to read debugging data. To do this, I needed to change the rs232 levels from the 5V / 0V supply used by the microcontroller up to the 10V / -10V voltages required by the rs232 specifications.

To do this, I chose to use a MAX232 chip. The MAX232 is a level-shifting chip that provides its own +10V and -10V supply, using clever internal circuitry. All I needed to add was 5 external capacitors, and an LED to show the unit was powered.
To test the unit, I shorted the Receive and Transmit leads on the microcontroller side of the MAX232, applied power, and then typed some text into a serial terminal emulator. As the text was echoed back, I concluded that the circuit was working correctly.

5.2 The Simulator Software

5.2.1 Data Structures

5.2.1.1 The Maze

The first task, when given a maze to solve, is to work out a method of representing the maze in the software. For this, I chose to use the most simplistic approach possible: a 2 dimensional array.

A 2d array is basically a static length list of variables, each of which contains another array of the same size. As the MicroMouse maze is a 16 x 16 grid, I chose to simulate it using a 16-element array, each containing its own 16-element array.

Once the array is set up, we reference the elements by giving them coordinates. For example, the array in the main array’s third element would be at location 2. The element in that array’s third element would be location 2, 2.

5.2.1.2 The Cell

As I was implementing the Simulator software as I hoped to implement the software on the MicroMouse itself, I chose to use a single 8-bit byte to represent a maze cell.

This was chosen over a simpler object-oriented approach, as it would give me a taster of the bit-maths I would have to do during the actual implementation of the robot’s software, and hence give a simulator that is closer to how the final robot should work.

I decided to use the lower nibble of the byte to represent the four walls, and the lowest bit of the upper nibble to log whether or not the cell had been visited.

```plaintext
byte = 0 0 0 v t r b l

0 = left
1 = bottom
2 = right
3 = top
4 = has cell been visited
```
5.2.2 Algorithms

One of the best maze solving algorithms I could find during my research was the flood fill algorithm[^FFILL]. I decided I would implement this algorithm for the simulator software.

Basically, a flood fill is performed by stepping out a cell at a time from the center of a maze, and giving each cell a number one greater than the cell before. Once this has been completed for the entire maze, the fastest path can be followed from the start to the finish by simply going to any surrounding cell that has a value lower than the value of your current cell.

So for the flood fill, I implemented this algorithm:

1. Set all cell values to an impossible value (e.g. -1).
2. Start at the Target cells.
3. Set the target cells to 0, and put 0 into a currentvalue variable.
4. Add any accessible cells to a stack
5. Increment the currentvalue variable
6. Step through the stack one cell at a time, for each:
   1. Set cell’s value to currentvalue.
   2. Add any accessible cells with the value -1 to a second stack
7. Once stack is empty, replace with second stack
8. Repeat steps 5 – 8 until the second stack is empty. This means there are no more accessible cells with no value in the maze.

Once the maze has its floodfill values set, we can continue with working out the shortest path through the maze:

1. Start at Starting cell
2. Check all accessible cells’ values
3. Move to accessible cell with lowest value
4. Repeat 2 – 4 until the target cell is reached

The only additional thing I needed during the implementation of this algorithm was a method of choosing which cell to move to if two adjacent cells have identical values. In this algorithm, it makes no difference to the path length, and either direction will be the same length, so I chose an arbitrary order, preferring to move left then right then up and lastly down.

5.2.3 The .maz file

The information in this section was originally based on the mousetech wiki[^MAZFILE]

The .maz file format originated in Japan, but has since become the standard file format for storing and interchanging maze data. It consists of 256 bytes of data, each byte describing one cell in the maze.

Of each byte, only four bits are used, the lowest nibble. Each bit represents a
wall, in the order North, East, South, and West.

The origin cell is in the 240th byte, 0xF0.

0x0f .. .. .. 0xff
.. .. .. .. ..
0x00 .. .. .. 0xf0

Bit 7 6 5 4 3 2 1 0
use - - - - West South East North

Diagram From [MAZFILE]

5.2.4 Implementation

5.2.4.1 Version 1

Once I had designed the data structures needed for the simulator, implementation went largely smoothly. I first built a simple CLI program that would take in a .maz file, process it into a maze, and echo it back into the console:

$> ./simulator.rb MM03HEAT.MAZ

Printing Maze:
1b 18 1a 1c 1d 1b 1c 19 1a 1a 1a 1a 18 1a 1a 1c
19 12 1c 11 12 18 12 12 1a 18 18 1a 12 1a 1c 15
11 1c 13 14 1b 12 18 1a 18 16 15 1d 19 1e 15 15
15 11 1a 14 1d 19 14 1d 15 1b 12 12 16 1d 11 16
15 11 1c 13 12 16 13 14 15 1b 1a 18 1c 15 11 1c
15 15 13 1c 19 18 1c 17 15 1b 1a 16 15 11 16 15
11 12 1c 13 16 15 15 1d 13 1a 1a 1a 12 16 19 14
15 1f 13 1c 1d 15 15 11 18 1a 1a 1c 1b 1a 16 15
11 1c 19 12 14 11 14 13 14 1d 1d 13 18 1a 1c 15
15 11 16 1d 15 15 11 1c 15 15 11 1c 13 18 12 16
15 15 19 14 15 11 14 15 15 17 17 15 1b 14 1d 1d
15 11 16 11 12 14 13 14 11 1a 18 16 1b 12 12 16
11 12 1c 15 1d 11 1c 15 17 1d 11 1a 1c 1d 1b 1c
15 1d 13 14 11 14 15 13 18 16 15 19 12 14 19 14
11 12 1c 15 15 17 11 1c 13 1c 13 12 1c 13 16 15
13 1e 17 13 16 1b 16 13 1a 12 1a 1a 12 1a 1a 16

The code to process the maze had to perform two functions. It had to first of all rotate the maze from the read in position, with the start cell in the bottom left, to my position in the top left, and then parse out and set up the walls for each cell one at a time. After a cell had been read, its visited bit would be set to 1. The simulator would then print out each the cell’s byte values in a hexadecimal format.
5.2.4.2 Version 2

For Version 2, I decided it was time for the program to go graphical. Using Tk’s Canvas object, I added the functionality into my program to draw an ‘overhead view’ of the maze, similar to the overhead camera views shown in videos of MicroMouse competitions.

While I was programming the Graphical Display, I also added the functionality to display 1 or 2 characters in each cell.

![Figure 25 – The Maze Display](image)

5.2.4.3 Version 3

In Version 3, I added the flood fill algorithm to the software. The software was now able to take in a .maz file, parse it into its internal representation, and run a flood fill on the parsed data. After adding a method to print out the flood values to the console, increment 3 was completed.

Flood filling maze:

```
Flooded Maze
21 20 1f 1e 1d 1c 1b 1a 19 18 17 16 15 16 15 14
22 21 20 1d 1c 1b 1a 19 18 17 16 15 14 13 12 13
23 22 1f 1e 1d 1c 1b 1a 19 18 17 1a 1b 1c 11 12
23 21 20 1f 20 1d 1c 1f 1a 19 18 19 1a 13 10 11
22 21 22 20 1f 1e 1d 1e 19 18 17 16 15 12 0f 0e
21 20 21 20 1d 1c 1b 1f 18 19 18 17 14 11 10 0d
20 1f 1e 1f 1e 1b 1a 01 17 16 15 14 13 12 0d 0c
21 ff 1d 1c 1b 1a 19 00 00 01 02 03 10 0f 0e 0b
20 1f 1c 1b 1a 19 18 00 00 ff 0a 04 05 06 07 0a
21 1e 1d 1a 19 18 17 16 01 ff 09 08 06 07 08 09
20 1d 1a 19 18 17 16 15 02 ff 0a 07 09 08 0b 0c
1f 1c 1b 18 17 16 15 14 03 04 05 06 0a 09 0a 0b
1e 1d 1c 19 18 15 14 13 04 13 06 07 08 0b 0c 0f
1f 22 1b 1a 17 16 13 12 11 12 07 0a 09 0a 0d 0e
20 21 22 1b 18 17 12 11 10 0f 08 09 0a 0b 0c 0f
21 22 23 1a 19 14 13 10 0f 0e 0d 0c 0b 0c 0d 0e
```
5.2.4.4 Version 4

Version 4 simply involved some small modifications to the flood fill algorithm to make it display the flood fill values on the Graphical Display.

This increment made it possible to follow the path from the start to the finish using the floodfill values.

5.2.4.5 Version 5

After completing the floodfill, all that was left to do was to display the fastest path. I chose to do this by highlighting each cell along the path with a white border around the floodfill value.

To do this, I implemented the algorithm described in 5.2.2, only after each loop I would mark the current cell I was in with a white square. After the center was reached, I would call the method that drew the flood-fill numbers on all the cells, to make sure the numbers were still displayed.

Finding fastest path: Solved
5.3 Robot Software Design

5.3.1 The Stepper Driver

A stepper motor is driven by activating four coils in a certain order.

There are 2 types of stepping modes (orders the coils can be stepped in), full stepping and half stepping mode.

<table>
<thead>
<tr>
<th>Full Stepping Mode</th>
<th>Half Stepping Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1 1 0 0</td>
<td>A 1 1 0 0</td>
</tr>
<tr>
<td>B 0 1 1 0</td>
<td>B 0 1 1 0</td>
</tr>
<tr>
<td>C 0 0 1 1</td>
<td>C 0 0 1 1</td>
</tr>
<tr>
<td>D 1 0 0 1</td>
<td>D 1 0 0 1</td>
</tr>
</tbody>
</table>

In full stepping mode, the coils are activated two at a time, in order. In half stepping mode, the coils in between are also activated on their own, hence the motor will step half as far each time, giving twice the resolution. As I did not need the higher resolution offered by half stepping, I decided to use Full stepping mode.

To do this for one motor, I had to:
1. Wait for step signal
2. Check direction line
   1. If direction line high, rotate output bits left
   2. If low, rotate output bits right
3. Check for an overflow on either end
4. If there has been an overflow, set the bit at the opposite end to 1.

Of course, some modifications had to be made to let the system work to control two separate motors. Here’s how I did it:

```c
int main(void)
{
    // Disable timer.
    wdt_disable();

    // Set PORTB as outputs;
    DDRB = 0xFF;
    // Set the PIND as inputs.
```
DDRD = 0x00;

// config variable to make sure we're only picking up each pulse once
// for each motor
// config[0] = motor 1 cleared
// config[1] = motor 2 cleared
uint8_t config = 0x03;

// Configure the outputs.
PORTB = 0x66;
while (1)
{
    uint8_t x;
    uint8_t y;
    uint8_t z;

    // Read PIND (switches)
    x = PIND;

    // Read current contents of PortB
    y = PORTB;

    // Split portb into two variables, with the upper nybble in z
    // (motor 2) and lower nybble in y (motor 1)
    z = y & 0xf0;
    y &= 0x0f;

    // check if we are receiving a pulse for motor 1
    if( bit_get( x, M1_STEP ) )
    {
        // if the last pulse received has been cleared
        if( bit_get( config, M1_CLEARED ) )
        {
            // check direction bit for motor 1
            if( bit_get( x, M1_DIR ) )
            {
                // if on, we rotate the bits in the nybble left.
                y = y<<1;
                // check if the nybble has overflowed into the upper nybble
                if( bit_get( y, M2_LOWEST ) )
                {
                    bit_set( y, M1_LOWEST );
                    bit_clear( y, M2_LOWEST );
                }
            }
            else
            {
                // if off, we rotate the bits right
                y = y>>1;
                // check for an overflow using the carry
                // bit
if( bit_get( SREG, CARRY ) )
{
    bit_set( y, M1_HIGHEST );
}

} // we have processed a pulse, and it has not
// been cleared yet.
bit_clear( config, M1_CLEARED );
}
else
{
    // the motor 1 step pin is 0, the previous pulse
    // has been cleared
    bit_set( config, M1_CLEARED );
}

// check if we are receiving a pulse for motor 2
if( bit_get( x, M2_STEP ) )
{
    // if the last pulse received has been cleared
    if( bit_get( config, M2_CLEARED ) )
    {
        // check direction bit for motor 1
        if( bit_get( x, M2_DIR ) )
        {
            // if on, we rotate the bits in the
            // nybble right.
            z = z>>1;
            // check if we have underflowed (is this
            // a word) into the lower nybble
            if( bit_get( z, M1_HIGHEST ) )
            {
                bit_set( z, M2_HIGHEST );
                bit_clear( z, M1_HIGHEST );
            }
        }
        else
        {
            // if off, we rotate left
            z = z<<1;
            // use the carry bit to check for an
            // overflow
            if( bit_get( SREG, CARRY ) )
            {
                bit_set( z, M2_LOWEST );
            }
        }
    }
    // we have processed a pulse, and it has not
    // been cleared yet.
    bit_clear( config, M2_CLEARED );
5.3.2 The Main Board Microcontroller

5.3.2.1 The ADCs

My first task was to try and get the ADCs up and running. After careful reading of the AtMega16 datasheet, I eventually worked out what needed to be done.

The first thing that needs to be done to use the ADCs is to initialise them. This is done in two stages:

- Set the ADCs to use the external AREF (analog reference voltage) pin as the reference voltage.
- Enable the ADCs.

The first stage is done by setting bit 2 of the ADMUX register to 1:

```
//select external (AREF) voltage as the reference voltage (0100 0000)
ADMUX = 0x40;
```

The second, by setting bit 1 of the ADuCSR register to 1:

```
//enable ADC (1000 0000)
ADCSRA |= 0x80;
```

After the ADCs have been configured, I needed a way of accessing them. To take a reading from the ADCs, we first have to select which pin the ADC will be reading from. We do this using the lower nibble of the byte to select the ADC number. Not forgetting to keep the AREF pin selected as the input!

```
// set the adc to the chosen channel
ADMUX = n;
// set the reference voltage to the aref pin
ADMUX |= 0x40;
```

Next, we setup the ADC clock to 1/16th of the CPU clock, and start the
conversion.

```c
// start the ADC conversion, set the ADC clock to
// cpu clock / 16 (0100 0100)
ADCSRA |= 0x44;
```

Then we simply have to wait until the conversion has completed, which is shown by the conversion start bit being unset.

```c
// wait for the adc conversion to be completed
while((ADCSRA & 0x40) != 0){};
```

Once the conversion has completed, the reading is stored in the ADC register.

As the reading is 10 bits, we have two options for using it. Either we can sacrifice accuracy by dividing the result by 4, thus fitting it into an 8-bit byte, or we can use a 16-bit integer, which is a ‘virtual’ 16-bit integer, provided by the compiler.

Once I had the code written for the ADCs, I used an atmega16 on my breadboard, with the ADC input attached to a potentiometer, and 8 LEDs attached to the output. This allowed me to see that the ADCs were working as expected.

### 5.3.2.2 RS232 link

After getting the ADCs up and running, I next wanted to get some way of remotely reading their status from a PC. To do this, I had to work out how to use the UART (Universal Asynchronous Receiver / Transmitter) in rs232 mode.

Step one was to work out how to turn the serial port on. To do this, I turned to the atmega16 datasheet (see appendices).

The first thing I had to do is work out what divider was needed to run the UART at a known baud. The Equation to work this out is:

\[
\text{(CPU Speed} / (16 \times \text{serial link speed})) - 1
\]

I put the CPU frequency and link speed (baud) into a #define constant, and then used a simple equation to work out the needed speed:

```c
// work out baud. this section is modified from page 151 of
// the atmega16 datasheet
int ubbr = F_CPU / 16 / baud - 1;
```

Next, I had to put this value into the UBBR register. As the value here can be greater than 255, it has to be done in two parts, first the high byte, then the low byte.

```c
// set the high bit of the ubbr register
UBRRH = (unsigned char) (ubbr>>8);
// set the low bit of the ubbr register
UBRRL = (unsigned char) ubbr;
```
Next, it was simply a case of enabling the transmit and receive pins, by setting the correct bits in the UCSRB register:

```
// Enable transmit and receive
UCSRB = ( 1 << RXEN | 1 << TXEN );
```

After getting the initialisation in place, I had to build functions for sending and receiving data.

To send data over the serial link, we simply wait until the UART has sent all the data from its buffer, and then add the new data we want to send into the buffer.

To check if the buffer currently has any data in it, we need to check the UDRE (data register empty) bit of the UCSRA (USART Control & Status Register A) register is set to 1. If so, we can proceed with putting our new data into the buffer.

```
// transmit a character on the uart
void uart_tx ( unsigned char data )
{
    // wait until the uart data register is empty
    while ( !(UCSRA & (_BV(UDRE))) );
    // put the data into the transmit buffer
    UDR = data;
}
```

To receive data from the serial link, we check if the USART hardware has received a character, by checking the RxC (USART receive complete) bit of the UCSRA register.

Once the bit is 1, we can get the data from the UDR buffer, and use the character as we wish.

```
// receive a character from the uart
unsigned char uart_rx ( void )
{
    // wait for character to arrive
    while ( !(UCSRA & (_BV(RXC))) );
    // return received character
    return UDR;
}
```

As an addition, while flicking through the atmega16 datasheet, I found a way to set up an stdout variable to use my uart tx function to send any characters thrown at it. I set this up as described in the datasheet, and was then able to use regular printf commands to send debugging information back to the host PC.

```
static FILE mystdout = FDEV_SETUP_STREAM(uart_tx, NULL,
    _FDEV_SETUP_WRITE);
```
5.3.2.3  Moving the motors

Next, I built a small program to make the motors each take one step forwards. To do this, I had to send a single pulse down each of the ‘step’ lines.

After that, I built a function to take in a number of steps, and step the motors that far.

Basically, the function takes it number of steps, it doubles it, then toggles the ‘step’ pin on and off that many times, with a 2uS delay in between each toggle. This produces a train of steps, which have a 4uS cycle, corresponding to a step speed of 250 steps per second. This corresponds to a shaft speed of 1.25 revolutions per second, or 75 revolutions per minute.

```c
void steps ( uint16_t numsteps )
{
    numsteps += numsteps;
    while (numsteps > 0)
    {
        numsteps--;
        _delay_ms(2);
        bit_flip(PORTD, LEFT_STEP);
        bit_flip(PORTD, RIGHT_STEP);
    }
}
```

5.3.2.4  Integration – the RS232 command program

Once I had all these parts working, I decided I needed to bring them all together into a single program. This program was the rs232 controller, which had a lot of useful debugging features, including:
1. Full remote control
2. Displaying sensor readings for all sensors
3. Displaying the current map layout

The code basically consists of a main loop, which waits for a character over the rs232 link. Once a character is received, the program runs it through a collection of if statements and does an action depending on the character received.

```c
for(;;)
{
    x = 0;
    a=ReceiveByte();
    if (a == '8')
    {
        bit_clear(PORTD, LEFT_DIR);
        bit_clear(PORTD, RIGHT_DIR);
        steps(20);
    }
    if (a == '2')
    {
        bit_set(PORTD, LEFT_DIR);
```
bit_set(PORTD, RIGHT_DIR);
steps(20);
}  
if (a == '6')
{
    bit_clear(PORTD, LEFT_DIR);
    bit_set(PORTD, RIGHT_DIR);
    steps(20);
}
if (a == '4')
{
    bit_set(PORTD, LEFT_DIR);
    bit_clear(PORTD, RIGHT_DIR);
    steps(20);
}
if (a == '5')
{
    for (x = 0; x < 8; x++)
    {
        a0 = readadchan(x);
        printf("sensor %d: reading %d, 0x%x\n", x, a0, a0);
    }
    TransmitByte(\n');
}
if (a == 'm')
{
    int y = 0;
    printf("    0  1  2  3  4  5  6  7  8  9  10 11
12 13 14 15\n");
    for (x = 0; x < 16; x++)
    {
        printf(" %2d ", x);
        for (y = 0; y < 16; y++)
        {
            printf("%02x ", maze[(x * 16)+y]);
        //printf("%x\n ", x, y);
        }
    }
    TransmitByte(\n');
}
if (a == 'b')
{
    printf("\0\0\0");
    for (x = 0; x < 256; x++)
    {
        TransmitByte(maze[x]);
    }
    printf("\0\0\0");
}
At this time, my next task was to try and get the robot to move under its own power and stop when it detected something in front of it. Unfortunately, at this time, I started having problems with the robot's hardware and was unable
to continue, as described in the next section.
Chapter 6:

Problems Encountered
6 Problems Encountered

6.1 Hardware Problems
Towards the middle of my project, after getting the hardware finished, I started getting odd behaviour from the stepper driver. Occasionally, the motors would stop responding to step commands, and would freewheel. The only way to fix this was to reset the entire MicroMouse’s hardware by disconnecting, then reapplying the power.

My first thought on encountering this problem was that there was a problem with the 3.3v regulator not sourcing enough current when it heated up. As I was already having problems with the regulator’s heat output and the size of heatsink needed for it, I rebuilt the power board into the version 2 board mentioned above. This seemed to fix the problem for 20 minutes or so, but soon after the same thing was happening again.

My second attempt to fix this problem involved removing the stepper driver processor, and rebuilding the breadboarded test circuit I had used for initial software development. After rebuilding this circuit and installing the microcontroller, I found that it was working as expected, and was outputting the correct sequence to the LEDs.

My next attempt to repair the problem involved removing and replacing the driver chips for the motors. Again, this did not make a difference to the problem.

Eventually, I tracked down the problem: the power connectors! Apparently, I had a bad connection on the 5v and 3.3v lines going to the Mainboard and the Stepper driver boards! Unfortunately, I discovered this problem too late to do anything about it, and was unable to get the MicroMouse moving for a long enough period of time.

This was not the only problem with the hardware though, as half way through the project the ADCs on my atmega16 stopped returning any values!

To fix this, I took my spare atmega16 and switched it with the one I had been using. After setting the fuses, and programming it, I connected the rs232 adaptor to it and... nothing! For some strange reason, I was unable to get my second atmega16 to communicate at all via rs232, and so could not continue with the code on the robot without a replacement.

After waiting weeks for it to arrive, my replacement chip arrived, an atmega32 chip. The atmega32 is pin and code compatible with the atmega16, but comes with additional RAM and flash space. After recompiling and uploading the code and setting the fuses on this chip, I was unable to get this one connecting either! If I had not been pressed for time, I would have breadboarded the atmega32, and attempted to get my first iteration of rs232 code working on it, to find out where exactly the bug lies.
Chapter 7:

Evaluation
7 Evaluation

I believe that during this project, I achieved quite a lot of my initial objective. It is said in MicroMouse circles that 'the hardest bit is to get the mouse moving', and while I was unable to get the robot to solve the maze due to hardware failures, my implementation of the simulator shows that had the hardware been perfect, I would have achieved this.

During the project, I have learnt a lot about electronics systems, including power supply design and analog electronics, along with embedded and single-threaded system design. Poking registers to use peripherals was definitely a new experience!

During the project, I managed to get a lot of small electronic subsystems working, and I was quite impressed by the amount of boards produced:

- Power Board v1
- Power Board v2
- Stepper Board
- Main Board
- Sensor Board v1
- Sensor Board v2
- RS232 level shifter

Not only did I manage to get the board working, but also I managed to get them all working together!

Unfortunately, the project didn’t end as well as I’d hoped. The programming was effectively cut short when the hardware started failing for strange reasons. I think that most of the hardware problems probably came about due to my relative inexperience with modern electronics. I covered electronics years ago at A-level, but as I hadn’t used that knowledge in years it was outdated or misremembered.

I think that with a few more weeks work, I would have been able to get everything working as expected, and will be working towards this goal for UK MicroMouse 2007 in June!

7.1 Requirements Revisited

1. Must cost less than £150 in total.
   PASS – The total cost of the robot was under £100 if the programmer is not counted, or £140 if it is.

2. Must be able to map and solve a MicroMouse maze.
   CONDITIONAL PASS – The simulator showed that the algorithms used were able to solve the maze; unfortunately the robot was not finished on time to do this.

3. Must fit within, and be able to turn within, a single MicroMouse maze cell.
PASS – The robot’s maximum dimension is 120mm, and it is easily able to turn within a 168mm maze cell.

4 Must know where in the maze it is at any time.
   FAIL – As the robot’s software was not completed, it was not able to move around the maze or know where in the maze it was.

5 Must be simple to reprogram for use in other tasks.
   PASS – To reprogram the mouse, all that a user needs to do is import the rooter.h headers, and link their program against the rooter library. This provides access to the ADCs, motors and UART code.

6 Must be able to run the maze multiple times on one battery charge.
   ? – In my testing, the battery has lasted about half an hour of constant usage. Whether this would scale up to multiple runs in the maze, I’m not sure, as battery usage may be greater or smaller depending on outside factors.

7 Must be accurate when moving.
   PASS – When the stepper motor driver worked, it worked perfectly. If I told it to go forward 100 steps, it would do exactly that.

8 Must be able to sense the walls around it without touching them.
   PASS – When the IR sensors worked, they were able to pick up all the walls of the cells as expected.
Chapter 8:

Bibliography
8 Bibliography

UK MicroMouse rules
The UK MicroMouse rules, this page contains all the specifications for the maze, and was useful when I was specifying components, to make sure the mouse would fit in the maze!

Maze Solving algorithms at Cannock Chase Technical College
This page gives examples on how to perform a flood fill, and solve the maze from it.

[LBORO]
http://www.lboro.ac.uk/departments/el/robotics/micromouseindex.html
Maze Solving algorithms at Loughborough University
The algorithm described on this site allows for a fastest route as well as a shortest route component.

The history of MicroMouse, 1972 - 1991
An interesting look at how the MicroMouse competition evolved from the first competition in the 70s, to the early 90s.

[IRSLIDES]
CS36410 Intelligent Robotics Lecture slides Dr. Myra Wilson, University of Wales Aberystwyth
There is lots of useful information, including modules about sensors and robot locomotion within these slides.

MINNI: MicroMouse Incorporating Neural Network Intelligence
I read through this paper and found that it wasn't that useful within the scope of my project.

A Potential Maze Solving Algorithm for a MicroMouse Robot
An algorithm where a series of potentials are manipulated to solve a MicroMouse maze. May be useful for the algorithm stage of my project.

Has a lot of maths involving the kinematics of robots, would have been useful for working out acceleration profiles and other physics-based mathematics if the project had progressed far enough to require smooth acceleration.

The Extreme Programming Introduction site
Gave me lots of information on extreme programming, which was useful when comparing it to other programming methodologies
Desktop Power Supply from a PC
There was lots of useful information here, which I used to create my ATX power supply lead.

Mousetech wiki – MicroMouse tools
Contains a full description of the .maz file commonly used for storing mazes for use on a PC.