Embedded Systems WBL Project

Group 2

Module code: 5ELEN020W
Assessment Title: WBL Project Report

AI & Declaration

Declaration

This report/ presentation has been prepared based on my own work. Where other published and unpublished source materials have been used, these have been acknowledged in the references.

Use of Generative AI Tools

In preparing this project, I used the following Generative AI tools:

Tool	How it was used
ChatGPT	To understand concepts e.g. bitwise masking. To help understand key concepts, and planning stages
Microsoft copilot	Assist in debugging code
Grammarly	Used for spelling, grammar, punctuation, and readability improvements.

Abstract

This report documents the design, construction and testing of a prototype autonomous micro mouse robot that has been developed for module 5ELEN020W. The system has been based around an ARM Cortex-M3-based MBED LPC1768 microcontroller due to its high processing capability and the wide range of peripherals it supports. Wall detection has been executed using seven infra-red proximity sensors mounted onto the left, right and front of the robot's chassis and has been wired directly to the microcontroller's digital inputs. Propulsion has been achieved using two unipolar stepper motors controlled by an L297 motor controller chip with ULN2003A Darlington arrays to drive the two motors. The system has used an LM7805 linear voltage regulator for power regulation to the sensors, logic and motor controller circuits, providing a regulated 5V supply. The navigation of the robot has been done using C++ with the MBED API and has utilized Lee's flood fill algorithm to calculate the fastest route from start cell to target cell. The system was tested in a demonstration in the tunnel, demonstrating wall detection and motor control, achieving the requirements of the module.

Introduction

The following report provides a detailed description of the construction of a prototype micro mouse robot for the module 5ELEN020W, capable of autonomously solving a maze. The robot must successfully complete an 8x8 cell maze, with no prior knowledge of the position of any walls, travelling from a corner cell to a target cell in the diagonally opposite corner within 10 minutes. There were significant accomplishments involved, as this project required the integration of mechanical, electrical and computer engineering disciplines to achieve the design of a complete, autonomous system.

The CPU used for the robot was an MBED LPC1768 (based on the ARM Cortex-M3) as it possesses a high processing power and large peripheral set, allowing the implementation of sophisticated navigation software. Movement was facilitated by two Astrosyn bipolar stepper motors, each driven by an L297 motor controller backed by a ULN2003A Darlington array. These motors were chosen because their movement is highly consistent and repetitive, without any requirement for encoders to ascertain position- a fixed number of steps translates directly into a specific linear or angular distance. Seven infrared proximity sensors were integrated onto the chassis, located on the sides and front faces of the robot to measure the presence/absence of a wall through a simple digital output. Logic and sensor power were supplied to an operational 5V, through an LM7805 linear voltage regulator. The navigation method adopted was Lee's flood-fill algorithm whereby the values are propagated out from the target cell, and the robot moves to the adjacent accessible cell with the smallest value.

The report will proceed as follows; Section 1 will discuss the theory behind the electrical and software components including the navigation algorithm. Section 2 will discuss issues outside of these, namely sustainability, ethics and commerce. Section 3 will define the system requirements. Section 4 will discuss the cost of the system, and section 5 will be regarding the mechanical and electrical hardware design, section 6 the logical and electrical software design, section 7 will review tests and results, and section 8 will draw the report to a conclusion.

1.0 Background

Before getting into the design of the micro-mouse itself, it helps to understand the theory behind the main technologies we used. This section goes through each one – motors, sensors, navigation, and the development platform – and explains the key concepts. References are given in IEEE format throughout.

1.1 Motor theory

At its core, an electric motor works by using magnetic fields to create movement. When a wire carrying current is placed inside a magnetic field, a force acts on it. This is the Lorentz force, and it is described by:

$$F = BIL$$

Where F is the force in Newtons, B is the magnetic flux density in Tesla, I is the current in Amperes, and L is the length of the conductor sitting inside the field. This is the basic principle behind every electric motor.

In a rotary motor, the conductors are wound into a coil on a rotor. The magnetic force on the coil creates a turning effect – torque. The relationship is simple:

$$\tau = F \times r$$

Where r is the distance from the axis to the conductor. More current, stronger fields, or more turns in the coil all increase torque.

One thing worth knowing about DC motors is back-EMF. As the rotor spins, it cuts through the magnetic field, which by Faraday's law generates a voltage that opposes the supply voltage. The faster it spins, the higher the back-EMF, and the lower the net

current. This is what limits the top speed of a DC motor. Stepper motors behave differently – their speed is set by the step rate rather than voltage – but understanding back-EMF is still a useful background for motor theory generally.

1.2 Stepper motor types & control

A stepper motor does not spin continuously like a DC motor. Instead, it moves in fixed angular steps each time it gets a pulse. This makes it very useful for applications where you need to know exactly where the motor shaft is without using an encoder – which is exactly what we need for the micro-mouse.

There are three main types:

Permanent Magnet (PM): Has a permanent magnet rotor that aligns with the energized stator poles. Simple and cheap but lower resolution.

Variable Reluctance (VR): Uses a soft iron rotor that moves to minimize magnetic reluctance. Higher resolution but lower torque than PM.

Hybrid: Combines a permanent magnet with toothed pole pieces. Get the best of both – higher torque and finer resolution. The Y129 is a hybrid type.

Motors also differ in how their coils are wired. A unipolar motor has a center-tap on each winding, so current always flows the same way through each half. This makes the driver circuit simpler – which is why we went with it. A bipolar motor has no center-tap, so you need to reverse the current through the full winding to change direction, which needs a more complex H-bridge driver.

There are three ways to drive a stepper motor:

Full-step wave drive: One coil on at a time. Use the least power and give the full rated step angle (1.8° for the Y129). This is what we used in the project.

Full-step two-phase: Two coils on at a time. More torque but draws more current.

Half-step: Alternates between one and two coils. Doubles the resolution to 0.9° per step and runs more smoothly but needs more careful firmware timing.

One thing to be aware of with steppers is that torque drops off at higher step rates. The inductance of the coils slows down how quickly the current can build up, so there is less force at high speeds. This means the motor needs to be gradually ramped up from rest, otherwise it can miss steps and lose its position tracking.

1.3 Sensors & electromagnetic waves

The micro-mouse detects maze walls using infrared sensors. IR is a type of electromagnetic radiation sitting just below visible red light in the spectrum, with wavelengths roughly between 700 nm and 1 mm. It travels at the speed of light and reflects surfaces in the same way visible light does.

The basic idea of a reflective IR sensor is straightforward – an emitter sends a beam of IR, and a receiver picks up any light that bounces back from a nearby surface. If there is nothing close enough, the reflected signal is too weak to trigger the receiver. How strong the reflected signal depends on distance, the reflectivity of the surface, and the angle the beam hits it at.

The OP165D emitter used to have a peak wavelength of 950 nm and a narrow 20° beam, which helps focus on the signal and reduce interference between adjacent sensors. On the receiver side, the IS471F is not just a photodiode – it has an amplifier, bandpass filter, and demodulator all built in. The demodulator is specifically tuned to 38 kHz, which means the emitter must be pulsed at that frequency for the receiver to respond. This is what makes it resistant to interference from room lighting.

The output from the IS471F is a simple digital signal – the GOUT pin goes low when it detects a 38 kHz modulated signal and stays high when it does not. That makes it easy to connect straight to a GPIO input on the LPC1768.

1.4 Noise & Interference

Noise is basically any unwanted signal that gets in the way of the real one. For the micro-mouse, there were two main things to worry about: electrical noise from the motor circuits, and optical noise from IR sources in the environment.

On the electrical side:

Inductive switching spikes: Every time a motor coil switches off, the collapsing magnetic field causes a big voltage spike. If not dealt with, these can couple onto sensor lines and cause false readings.

EMI from motor switching: The switching currents in the motor drive lines generate electromagnetic fields that can induce voltages in nearby signal wires – especially if motor and sensor cables are routed close together.

Power supply noise: Ripple on the supply rail can affect both the analogue and digital circuits if it is not filtered out.

On the optical side:

Ambient IR: Sunlight and fluorescent lighting both put out a lot of IR. Without filtering, this can swamp the receiver and cause the robot to think walls are there when they are not.

Crosstalk: In a narrow maze corridor, IR from one sensor pair can reflect around and reach the receiver of a neighboring pair.

We tackled these in several ways. The ULN2003A has built-in flyback diodes that clamp inductive spikes before they can cause damage. Decoupling capacitors at each IC power pin to filter out supply noise. Motor and sensor cables are routed on opposite sides of the chassis to physically separate them. The IS471F's 38 kHz demodulator blocks all non-modulated IR. And in firmware, a moving average filter smooths out any remaining noise before the wall-detection logic runs.

1.5 Navigation

Navigation for a maze robot means two things: knowing where you are and deciding where to go next. There are different ways to approach this. The simplest is purely reactive – the robot just responds to whatever the sensors are currently reading, with no memory of where it has been. A more capable approach is deliberate navigation, where the robot builds a map and plans routes through it.

The most basic reactive method is wall following – keep one wall always on your left (or right) and follow it until you find the exit. It is easy to code, but it does not always work, especially in mazes with loops, and it will not find the shortest path.

We used dead reckoning to track the robot's position. Because stepper motors move in fixed, known steps, the firmware can count steps on each motor and work out how far the robot has moved and which direction it is facing. Each step of the Y129 corresponds to a known angular displacement of the wheel, so the position of tracking is reasonably accurate over the size of an 8×8 maze.

The maze is modelled as a grid of cells, each with an (x, y) coordinate starting at (0, 0) at the start position. As the robot moves through the maze, it updates its current cell and records which walls it has seen. This cell map is then used by the path-finding algorithm to plan the next move.

1.6 Mapping and path analysis

As the robot explores the maze, it builds a map by recording which walls are present at each cell. This is stored as a 2D array in memory, where each cell has flags for walls on all four sides. At the start, all walls are unknown – the map fills in as the robot visits new cells.

To find the shortest route to the goal, we used the flood-fill algorithm. The idea is easier to understand if you think of it visually – imagine flooding the maze with water starting from the goal cell. The goal gets value 0. Any cells directly accessible from the goal (no wall in between) get value 1. Their accessible neighbors get value 2, and so on until every cell has a value. The robot then always moves to whichever adjacent cell has the lowest value, which automatically takes it toward the goal by the shortest available route.

What makes flood-fill particularly useful is that it handles incomplete maps well. If the robot discovers a wall, it did not know about, the flood values are recalculated and the robot replans. So, it works during exploration, not just once the map is complete.

Breadth-First Search (BFS) is a related algorithm that also guarantees the shortest path. It explores all cells at the current distance before moving further out, which is conceptually similar to flood-fill. The main reason flood-fill tends to be preferred for micro-mouse is that it integrates more naturally with incremental map updates as the robot explores.

1.7 Batteries & load

Batteries store energy chemically and release it as electricity. For a portable robot the key things to consider are voltage, capacity, discharge rate, internal resistance, and how much it all weighs.

Capacity is rated in milliampere hours (mAh). A 2,850 mAh cell can theoretically deliver 2,850 mA for an hour, or 285 mA for ten hours. In practice capacity drops at higher currents – this is described by Peukert's law, which says capacity decreases as the discharge rate increases.

Internal resistance is another important factor. For alkaline AA cells it is typically 100–300 mΩ per cell, increasing as the cell runs down. With 8 cells in series, the total internal resistance can reach around 2.4 Ω at the end of life. At the currents the micro-mouse draws this causes a measurable voltage to drop at the battery terminals..

The total current draw for the micro-mouse is around 530 mA from the 12 V rail – the two motor coils take 320 mA (2 × 160 mA), and the 5 V rail for the LPC1768 and sensors draws roughly another 210-mA referred to 12 V. At that rate, the 2,850 mAh Energizer AA cells give an estimated runtime of over 4 hours, which is more than enough for competition use.

1.8 Driving large currents

The LPC1768 GPIO pins can only source or sink around 4 mA each, with a total limit of about 40 mA across all pins [14]. The Y129 stepper motor coils need 160 mA each – far more than a GPIO pin can handle. So, a driver circuit is needed to sit between the microcontroller and the motors and handle the heavy current.

We used the ULN2003A Darlington array for this. A Darlington pair is two NPN transistors connected, so the emitter of the first feeds the base of the second. This gives a very high combined current gain, so a tiny input current from the GPIO pin can switch a much larger current through the motor coil.

The relationship is:

$I_C \approx hFE1 \times hFE2 \times I_B$

Where hFE1 and hFE2 are the gains of the two transistors and I_B is the small base current from the microcontroller. In practice, the ULN2003A handles up to 500 mA per channel continuously, well above the 160 mA the motor needs.

There is also the issue of flyback voltage. When a motor coil switches off, the energy stored in the magnetic field must go somewhere – and it does so as a large voltage spike that can easily damage transistors or microcontroller pins if not dealt with. The ULN2003A has a suppression diode on every output channel that clamps this spike to a safe level, so no external protection components are needed.

1.9 The MBED platform

The mbed platform was developed by ARM to make prototyping with ARM Cortex-M microcontrollers faster and easier. It has two main parts: the hardware (development boards like the LPC1768) and the software (an online compiler and C++ library) [16].

The LPC1768 board is built around a 32-bit ARM Cortex-M3 running at 96 MHz. It has 512 kB of flash, 64 kB of SRAM, and a good range of peripherals – GPIO, PWM, ADC, SPI, I2C, and UART. For this project, the key ones were the GPIO pins for driving the motor drivers and reading the sensors, and the ADC for battery voltage monitoring [16].

The mbed C++ library wraps all the hardware into simple objects. A digital output is a DigitalOut, an analogue input is an AnalogIn, and so on. This meant we could focus on the logic of the navigation algorithm rather than spending time writing low-level register code. For a project with a tight deadline, this made a real difference [16].

Programming is done over USB – the board shows up as a USB drive, and you just copy the compiled binary onto it. No separate programmer is needed. The one downside is there is no proper hardware debugger, so any debugging must be done by printing values over the USB serial connection using printf.

1.10 Product life cycle stages

The product life cycle describes the stages a product goes through from the initial idea all the way to when it gets disposed of. It is useful to think about this because different stages carry different risks and environmental impacts. There are generally five stages:

1. Concept and Design: The idea is defined, requirements are set, and feasibility is looked at. For the micro-mouse, this meant reviewing the competition rules, deciding on the chassis design, and choosing the main components.

2. Development and Prototyping: The actual building and testing phase. Circuits are assembled, firmware is written, and the design is refined based on test results. Most of the WBL project time was spent here.

3. Manufacturing: In a commercial product, this is where volume production happens, with custom PCBs and automated assembly. For us, this was just a single prototype.

4. Operation and Maintenance: The product is being used. For the micro-mouse, this is the competition. Maintenance covers things like replacing batteries and cleaning sensors.

5. End of Life: The product reaches the end of its useful life. Electronic components contain materials that need proper recycling under the WEEE Directive.

The design stage has the biggest potential to reduce environmental impact – decisions made early about materials and repairability affect the whole lifetime of the product.

1.11 Risk and possible mitigation for product life cycle stages

Every stage of the life cycle has risks associated with it. The table below summarizes the main ones for the micro-mouse and the steps taken to reduce them.

Life Cycle Stage	Risk	Mitigation
Concept & Design	Requirements misunderstood or incomplete; wrong component chosen for the job	Careful review of competition rules; peer checking of requirements; datasheet verification for every component
Development & Prototyping	Component failure during testing; firmware bugs causing unexpected motor or sensor behavior; running out of time	Socketed components so they can be swapped easily; bottom-up testing approach; Gantt chart to track progress
Manufacturing / Assembly	Wiring mistakes on Veroboard; bad solder joints causing intermittent faults; component damage from ESD or too much heat	Continuity test after every strip cut; visual check of all solder joints; correct iron temperature and anti-static precautions
Operation & Maintenance	Battery runs out mid-competition; sensor drift from dust or surface variation; motor losing steps at higher speeds	Battery voltage monitoring in firmware; moving average filter on sensor readings; motor acceleration ramp; spare batteries on hand

End of Life	Electronic waste with hazardous materials going to landfill	RoHS-compliant parts throughout; batteries disposed of in lab recycling bins per Waste Batteries Regulations 2009 ; reusable components returned to lab stock

The biggest risks in practice were scheduling pressure during development and component failures on the bench. Both were managed by testing each sub-circuit separately before putting everything together – this made it much easier to spot and fix faults without having to strip the whole robot down.

2.0 Broader concerns related to the WBL project

This section the group will discuss broader issues including Product life cycle, Sustainability, Ethics, Environment, Society, Security, professional conduct, Legal issues and regulatory compliance requirements.

2.1 Industrial / commercial application of robots like the micro-mouse

Autonomous Mobile Robots (AMRs) are robots used in industrial and commercial settings, including warehousing and logistics, manufacturing plants, agriculture, and healthcare. The purpose of these AMRs is to reduce labour costs, increase efficiency, and operate 24 hours a day.

These types of robots navigate complex environments and use SLAM methods and path-finding algorithms, such as the micro mouse.

According to Bedkowki (2023), "*SLAM is a core component of the autonomous mobile mapping robot that builds a map based on the estimated trajectory and estimates this set of consecutive poses based on this map"*

The commercial application of robots like the micro mouse can also be seen in areas such as domestic settings, autonomous vehicles and drones, educational and research platforms, and agriculture.

2.2 Environment considerations and suggested mitigations

Environmental considerations for autonomous robots, such as the micro mouse, include the life cycles of the components used. Components like motors, PCBs, sensors, and batteries (Li-ion) have a long lifespan, but poor-quality components can increase the risk of waste due to shorter lifespans and early failure.

The use of Li-ion batteries, despite their long lifespan, can also contribute to environmental harm, according to the IER (2023), *The disposal of the batteries is also a climate threat. If the battery ends up in a landfill, its cells can release toxins, including heavy metals that can leak into the soil and groundwater.*

Also, an environmental consideration is the code we write for the micro mouse robot. Continuous execution of code in software can waste computational energy and electricity.

To mitigate these problems, we can use components with a longer lifespan, but this may mean spending more. What can be done is to plan our projects and ensure we conduct intensive research and/or have sufficient knowledge of components, so that the design and creation of the micro mouse are more efficient. This also relates to the code we write it must be concise, use as few lines as possible, and still be efficient.

For Li-ion batteries, we must ensure they are disposed of correctly.

2.3 Sustainability considerations and suggested mitigations

Sustainability refers to a system's ability to perform over time after it has been made. The micro mouse needs to have longevity. Choosing eco-friendly materials, such as those that are easier to reuse or are easily recycled. The materials and components should be purchased from reputable sources to ensure cost-effectiveness and durability.

2.4 Social impact of similar robot to the micro-mouse

There are many social impacts of robots, like the micro mouse. In fields like education, students interested in STEM can acquire practical skills in programming, electronics, design, problem-solving, and engineering. It can also inspire others who may be unsure about future goals to engage in STEM activities.

The micro mouse can also serve as an idea or prototype for future technology. Regarding the industrial area, it can also help reduce workplace risks and improve workplace safety.

Despite the social benefits, there are downsides as well. One is the replacement of human workers and the resulting job losses. According to Subaveerapandiyan. and Somipam R. Shimray (2024), *Job displacement due to automation often results in individuals transitioning from stable, high-quality employment to less secure, lower-quality roles in the secondary labour market. This shift can lead to diminished job security, limited social mobility, and a decline in social standing*.

Another issue is access to autonomous robots. Many people in less developed countries or small start-up companies may fall behind because they lack access to autonomous robots. In contrast, bigger companies may monopolise them due to early access.

2.5 Commercial considerations, Intellectual property rights and licensing.

The micro mouse project is a popular robotics project, and because of this, many iterations of the micro mouse are available to the public. The micro mouse project involves a large part of electronic engineering and robotics, and many interested in robotics will use some version of it. Because of the number of micro mouse project kits and ideas, such as learning how to build a micro mouse on platforms like YouTube, issues like intellectual property arise. Some places use open-source software licences like the MIT and Apache licences, which give users the right to modify, distribute, and use for any purpose, including commercial.

Others, such as books, manuals, and our documentation, are protected by copyright laws, and users may need to seek permission before modifying or using them for commercial purposes.

2.6 Legal issues, regulation and compliance for robots like the Micro mouse

Robots like the micro mouse, while mainly used in educational and research settings, still need to follow the legal considerations. The use of robotic systems must comply with safety standards, like in the *Electrical Equipment (Safety) Regulations 2016*, which ensure that electronic components meet required safety thresholds.

Where autonomous robots are used in commercial or public-facing environments, they may also need to comply with product liability regulations, meaning that if a robot causes damage or injury, the designer or manufacturer could be held responsible. It is therefore important that the design and testing of any robot are properly documented.

Additionally, if the robot collects any data, compliance with data protection laws, like the *UK General Data Protection Regulation (UK GDPR)*, may be required, depending on the context of use.

2.7 Identified risks and suggested mitigations for the micro-mouse robot

In the micro mouse project, there are some risks. If the micro mouse is using the wrong power rating, i.e., a microcontroller board like the LPC 1768 that can only handle 3.3V, attaching it to a higher voltage without a voltage regulator can cause the microcontroller to stop working or destroy it completely. It can also cause components, such as LEDs, to blow up. Wrongly wired components can cause the micro mouse to behave unpredictably.

Understanding power ratings and reviewing the datasheets of all components can mitigate some of these issues.

According to Hashir, Ihsan; Raj, Dharani Venkatesh; Alluri, Mehul. et al (2025). *Hardware integration also presents risks, as micro-mouse robots rely on embedded microcontrollers, sensors, and mechanical components that must work cohesively. Malfunctioning or mis-calibrated sensors, such as IR sensors used for navigation, can impair the robot's ability to detect walls or intersections accurately, leading to navigation errors. The complexity of hardware-software interaction requires careful design and testing to ensure reliable operation in real-time conditions*.

2.8 Professional codes of conduct, ethics, data security and privacy

The micro mouse must be self-moving and self-contained; there should be no remote controller controlling or navigating it, and it must rely solely on software and maze-solving algorithms.

Groups must be truthful and honest about the limitations of the micro mouse and willing to take accountability for any decisions made in its construction, e.g., software, wiring, and the use of electronic components. The BCS says in its code of conduct that you must *"NOT claim any level of competence that you do not possess". The BCS (2026)*

Within the team/group, members should be understanding and respectful of one another's time and, if possible, try to share the workload equally. Listening and

communicating within your group promptly addresses issues early and ensures that all contributions to the projects are acknowledged.

The micro mouse should not be used outside the university, except when students use it for interviews with potential employers.

2.9 Health and Safety

Whilst working in the laboratory, care must be taken to ensure equipment is not lost or damaged. Components and equipment are carefully and safely put away. Electrical wires must be out of the way when using equipment like an oscilloscope, and when attaching wires to plugs, make sure the switch is off. You should also be cautious when eating and drinking in the lab.

According to the health and safety act, at work 1974, (2026) take reasonable care for the health and safety of himself and of other persons who may be affected by his acts or omissions at work. Even though this is related to the workplace it is very relevant to students especially regarding the micro mouse project which is a work-based learning project.

You should also be shown how to use equipment, e.g., a soldering iron, and to take breaks when using it due to the fumes. If unsure of what to do in the lab, you should always ask.

3.0 Requirements

3.1 Operating Environment

The micro-mouse is designed to run inside a standard 8 × 8 cell maze, based on IEEE the IEEE Micromouse competition rules. Each maze cell measures 180 mm × 180 mm, with walls approximately 50 mm high. The maze will be used indoors under normal room lighting conditions. This means the robot may still be exposed to some background infrared interference from sources such as windows, room lighting, or fluorescent lamps. To reduce the effect of this, the IS471 infrared sensors were selected

because they use a modulated infrared signal, which helps reject unwanted ambient infrared light.

The robot starts in cell (0,0), facing North, and must navigate towards the centre goal region of the maze. In an 8 × 8 maze, this goal region is made up of the four central cells: (3,3), (4,3), (3,4), and (4,4). Once the robot is started, it must operate fully autonomously, meaning no remote control or manual input is allowed during the run. The robot must also stay within the competition footprint limit of 250 mm × 250 mm, so the chassis, sensors, wheels, wiring, and battery pack all need to fit within this size.

The wider design considerations discussed in Chapter 2 also influenced the operating environment requirements. For example, because the stepper motors can introduce electrical noise, the motor battery supply should be kept physically separate from the sensor and logic supply where possible. This helps reduce interference with the sensor readings and improves reliability. In addition, the health and safety requirements from Section 2.9 mean that the battery voltage and current levels must remain suitable for a university laboratory setting.

3.2 Functional Requirements

The functional requirements describe what the micro-mouse must be able to do during operation.

Functional Requirement 1 – Wall Detection:

The robot must detect whether there is a wall to its left, front, and right at each cell it enters. This is achieved using seven IS471 infrared proximity sensors positioned around the robot.

Functional Requirement 2 – Maze Mapping:

The robot must store information about the maze as it explores. Each visited cell should have its wall data saved in the walls[y][x] array. Once wall information has been recorded, it should not be cleared, as the map needs to become more accurate over time.

Functional Requirement 3 – Path Planning:

The robot must use the Lee flood-fill algorithm to calculate a route from its current position to the goal region. Whenever a new wall is detected, the flood-fill should be run again so the robot can update its path based on the latest maze information.

Functional Requirement 4 – Navigation:

The robot must be able to carry out the basic movements needed to travel through the maze. These include moving forward by one cell, turning left by 90 degrees, turning right by 90 degrees, and performing a 180-degree turn. These movements are controlled through the stepper motors using the L297 and ULN2003 motor driver chain.

Functional Requirement 5 – Explore and Speed Run:

The robot should first complete an exploration run, where it builds up its map of the maze. After this, it should return to the start and then carry out a faster run using the shortest known path.

Functional Requirement 6 – Safety Stop:

If the robot detects an unexpected wall in front of it while moving, it must stop immediately. This prevents the robot from driving into the maze wall and damaging the chassis, sensors, or motor assembly.

3.3 Mechanical Requirements

The mechanical requirements define the physical limits and performance targets that the robot must meet to move reliably through the maze.

Mass:

The total mass of the robot is estimated to be approximately 300 g. This includes the chassis, two stepper motors, the MBED LPC1768 board, the Veroboard circuit assembly, seven sensors, wiring, and 8 AA batteries. Keeping the robot reasonably light is important because it reduces the load on the motors and makes movement more reliable.

Maximum Velocity:

The initial target speed is one cell per second. Since each cell is 180 mm long, this gives a speed of 180 mm/s. With a wheel diameter of 65 mm, the wheel circumference is approximately:

Circumference = $\pi \times 65 = 204$ mm

To move one cell:

180 / 204 = 0.88-wheel revolutions

Using a stepper motor with 512 effective steps per revolution, the number of steps required to move one cell is:

0.88 × 512 = approximately 451 steps

Therefore, to move one cell in one second, the motors need to run at around 451 steps per second. This means each step occurs approximately every 2.2 ms. The firmware value STEP_DELAY_US = 2000 microseconds is therefore close to the calculated requirement and provides a practical starting point for testing.

Motor Torque:

The required drive force can be estimated using:

F = ma

Using a robot mass of 0.3 kg and an acceleration of 0.1 m/s^2:

F = 0.3 × 0.1 = 0.03 N

The wheel torque can then be estimated using:

T = F × r

With a wheel radius of 0.0325 m:

T = 0.03 × 0.0325 = 0.001 Nm

The stepper motor has a rated holding torque of approximately 34 mNm, which is significantly higher than the calculated minimum torque requirement. This gives a safety margin of more than 30 times, suggesting that the motor is suitable for the estimated robot mass. In practice, some torque will be lost due to friction, wheel slip, and the driver circuit, but the calculation still shows that the motor choice is reasonable for this design.

4.0 Specification and costing of Micro-Mouse Robot

This section goes through the components we chose for the micro-mouse, explains why we picked each one, and breaks down the costs. Each choice is matched against the requirements from Section 3.0. We also touch on sustainability and compliance where it is relevant.

4.1 Microcontroller – ARM mbed LPC1768

We chose the ARM mbed LPC1768 as the microcontroller for the robot. It runs a 32-bit ARM Cortex-M3 core at 96 MHz, which gives plenty of processing power for everything the robot needs to do at once – running the flood-fill algorithm, reading the IR sensors, and driving both stepper motors.

The specs that made it the right pick for this project:

- **Processing:** 96 MHz Cortex-M3 core with 512 kB flash and 64 kB SRAM – handles the navigation logic without any issues
- **Peripherals:** 6 PWM channels, SPI, I2C, 6 UARTs – more than enough for motor control and sensor communication
- **ADC:** 12-bit ADC with 8 channels – used for reading IR sensor outputs
- **GPIO:** 26 GPIO pins – enough to drive both motor driver ICs and handle all seven sensor inputs
- **Programming:** USB drag-and-drop programming – made firmware updates much quicker during testing

One thing that helped a lot was the mbed C++ library, which sped up development significantly given how tight the timeline was. The 3.3 V logic level is also directly compatible with the IS471F IR receiver output, so no level-shifting circuitry was needed. The board is provided as lab equipment so there is no cost to the project.

4.2 Drive Motors – Astrosyn Y129 Unipolar Stepper Motor (×2)

Two Astrosyn Y129 unipolar stepper motors were chosen to drive the robot. We went with stepper motors rather than DC motors mainly because they give precise positional control without needing encoders. Every step is exactly 1.8°, which works out to 200 steps per revolution. That precision is what makes the spin-on-spot turning method work – both wheels have to rotate by exactly the same amount in opposite directions to get a clean 90° turn, and steppers let us do that reliably.

Full specifications from the datasheet [1]:

- **Type:** Unipolar, single shaft, 42 mm NEMA 17 frames
- **Step angle:** 1.8° per step / 200 steps per revolution
- **Torque:** 9 N·cm unipolar holding torque
- **Phase current:** 160 mA per phase
- **Phase resistance:** 75 Ω per phase
- **Phase inductance:** 60 mH per phase
- **Voltage:** 12 V rated
- **Physical:** 0.22 kg, 33 mm body length, 5 mm shaft diameter
- **Compliance:** RoHS compliant

We checked the torque against the worst-case robot mass of 150 g. With a 30 mm wheel radius and a rolling friction coefficient of 0.01 on smooth wood, the required torque per wheel comes out at roughly 0.022 N·m (2.2 N·cm). The Y129 gives 9 N·cm, so the safety factor is about 4 – well above what is needed. The 6-wire unipolar winding also connects directly to the ULN2003A driver without any extra components.

On the sustainability side, stepper motors have no brushes, so they last longer and produce less wear-related waste than brushed DC motors. RoHS compliance means no lead, mercury, or cadmium [2].

4.3 Motor Driver IC – ULN2003A (×2)

The ULN2003A was chosen to drive the stepper motors. It is a Darlington transistor array with seven NPN pairs – four of them are used per motor to switch the four coil phases of the Y129. So, two ICs are needed in total, one per motor.

Key specs [3]:

- **Configuration:** 7 × NPN Darlington pairs in a 16-pin DIP package
- **Output current:** 500 mA continuous, 600 mA peak per channel
- **Voltage rating:** 50 V maximum collector–emitter voltage
- **Logic compatibility:** Works with 5 V TTL and CMOS logic – driven straight from the LPC1768 GPIO pins

- **Protection:** Built-in flyback diodes to handle voltage spikes from the motor coils

The 500-mA channel rating is well above the 160 mA the Y129 coils need, so the driver runs comfortably within its limits. The built-in flyback diodes also mean we did not need to add any external protection components, which simplified the circuit. The through-hole DIP-16 package also works well for Veroboard assembly.

4.4 IR Sensors – IS471F Receiver and OP165D Emitter

For wall detection we used pairs of IR emitters and receivers – the OP165D emitter and IS471F receiver. Seven pairs are mounted on the chassis, with three facing forward, two angled diagonally, and two facing sideways. This gives the robot full visibility of all three possible wall directions at any maze junction.

IS471F receiver specs [4]:

- **Function:** Integrated demodulator tuned to 38 kHz – blocks out ambient IR noise from lighting
- **Output:** Active-low digital output – can be read directly by the LPC1768 GPIO pins
- **Supply:** 2.7 V to 5.5 V supply – compatible with the 3.3 V LPC1768 logic
- **Range:** Detection range covers the 180 mm maze corridor width

OP165D emitter specs [5]:

- **Wavelength:** 950 nm peak emission wavelength (near-infrared)
- **Output power:** 80 mW/sr radiant intensity at 100 mA
- **Beam angle:** 20° half-angle beam – reduces crosstalk between adjacent sensors

Because the IS471F only responds to a modulated 38 kHz signal, it naturally ignores steady-state IR from sunlight and artificial lighting, which was one of the main noise concerns in the maze environment. A 200 Ω current-limiting resistor on each OP165D keeps the emitter current at a safe $(5 - 1.2) / 200 \approx 19$ mA. Software filtering then smooths out any remaining noise before the wall-detection logic makes a decision.

4.5 Voltage Regulator – LM7805

The battery pack gives around 12 V, but the LPC1768 and the IS471F sensors both need a stable 5 V supply. We used an LM7805 linear regulator to step the 12 V down to 5 V for all the logic and sensor circuitry.

LM7805 specs [6]:

- **Voltage:** 5 V fixed output, accepts 7–35 V input
- **Current:** 1.5 A maximum output current
- **Protection:** Internal thermal shutdown and current limiting
- **Package:** TO-220 package – easy to attach a heatsink if needed

With 12 V in and 5 V out, the regulator drops 7 V across itself. At an estimated load of around 500 mA (LPC1768 ~200 mA, seven IS471F sensors ~35 mA total, logic ~50 mA, plus some margin), the worst-case power dissipation is 3.5 W. A small aluminium heatsink is attached to keep it within safe operating temperature.

4.6 Power Source – 8 × Energizer AA Alkaline Batteries

Eight AA alkaline cells in series give a nominal 12 V supply. We went with Energizer Max AA cells because they are easy to get hold of, have predictable discharge behaviour, and are straightforward to use safely in a lab.

Energizer Max AA specs [7]:

- **Voltage:** 1.5 V nominal per cell, ~1.2 V under moderate load
- **Capacity:** Around 2,850 mAh capacity at low drain
- **Shelf life:** Up to 10-year shelf life
- **Physical:** 14.5 mm × 50.5 mm, ~23 g per cell
- **Sustainability:** 100% recyclable packaging, no added mercury

Total current draw is estimated at around 530 mA – motors take 320 mA (2 × 160 mA) and the 5 V rail draws roughly another 210 mA referred back to 12 V. At that rate, eight AA cells give well over 4 hours of runtime, which is more than enough for competition use. The cells sit in two 4-cell holders connected in series, so replacing them mid-session takes seconds.

The downside is that alkaline cells are single-use. We chose them over lithium-ion because they need no charging circuit and there is no thermal runaway risk, which matters in a lab setting. For a future version, NiMH rechargeable AAs would be the better environmental choice.

4.7 Component Cost Summary

The table below lists all components, quantities, and costs. Prices are from Farnell UK and Tesco for the batteries, correct as of April 2025. The mbed board and Veroboard are university lab equipment, so they are not costed to the project.

5.0 Hardware Design & Mechanics

This section covers how the robot was physically designed and how the hardware is all connected together. It goes through the chassis, power system, motor control, sensors, microcontroller wiring, and battery handling.

5.1 Chassis Design and Layout

The chassis is U-shaped with the drive wheels at the centre of the body. This is sometimes called a wheelchair configuration. The key advantage is that the robot can spin on the spot – by running both wheels at the same speed in opposite directions, the robot rotates around the midpoint of the axle without moving forward or backward. This is how the robot makes its 90° turns inside the maze.

Dimensions and layout:

- **Width:** 80 mm wide – fits inside the 180 mm maze corridor with about 50 mm clearance on each side, enough for sensors to work without the body touching the walls
- **Length:** Approximately 100 mm long
- **Axle:** Drive axle at the midpoint of the body length
- **Wheels:** ~60 mm diameter wheels, attached with set-screw hubs to prevent shaft slippage
- **Motor mount:** Motors sit in custom-cut slots in the chassis base plate
- **PCB mount:** PCB mounted on 10 mm nylon standoffs using M3 screws – keeps it electrically isolated from the chassis

The internal layout is stacked in layers. The battery holder goes at the bottom at the rear – this keeps the centre of gravity low, which helps stability. The motors sit at mid-height at the axle position. The main PCB goes above them on standoffs, and the mbed board plugs into headers on top of that. The sensor pairs are fixed to the front face of the chassis, sitting low so they get a good reflection off the wooden maze walls. Leaving the rear of the U open saves weight and lets air circulate around the motors and the voltage regulator.

5.2 Power Distribution

The power system runs two voltage rails from the battery pack. The 12 V unregulated rail goes straight to the motor drivers and motor coils. The 5 V regulated rail – produced by

the LM7805 – powers the LPC1768, the IR receivers, and everything else on the logic side.

To keep the supply stable, 100 nF ceramic capacitors are placed at both the input and output of the LM7805. A 10 µF electrolytic capacitor at the output handles sudden current spikes, for example when several sensor receivers switch at the same time. All ground connections go back to a common ground plane on the Veroboard, which helps reduce interference between the motor and sensor circuits.

There is also a red power LED (3 mm, with a 1 kΩ resistor) connected to the 5 V rail so it is obvious at a glance whether the system is on. A main power switch on the battery positive line lets the robot be safely powered down without physically disconnecting the batteries.

5.3 Motor Driver Circuit

Each stepper motor has its own ULN2003A driver IC. The Y129 has six wires – a centre-tap pair that connects to the 12 V rail, and four coil-end wires, one for each phase. Each coil-end wire goes to one output channel of the ULN2003A, and the four matching input pins connect to GPIO outputs on the LPC1768.

We used full-step drive sequencing, where one coil is energised at a time in the order A → B → A' → B' → A. This gives the full 200 steps/revolution of the Y129 and uses less power than energising two coils at once. Half-stepping could be added later in firmware if smoother motion turns out to be needed.

When the LPC1768 pulls an input pin high, the ULN2003A pulls the corresponding coil-end wire low, so current flows from the 12 V centre-tap through the coil to ground. When the coil switches off, the built-in flyback diodes clamp the resulting voltage spike, protecting both the IC and the GPIO pins. No external protection components are needed.

GPIO assignments: left wheel motor uses p21, p22, p23, p24; right wheel motor uses p25, p26, p27, p28. These are all standard digital output pins on the LPC1768 that do not clash with any ADC or PWM functions used elsewhere.

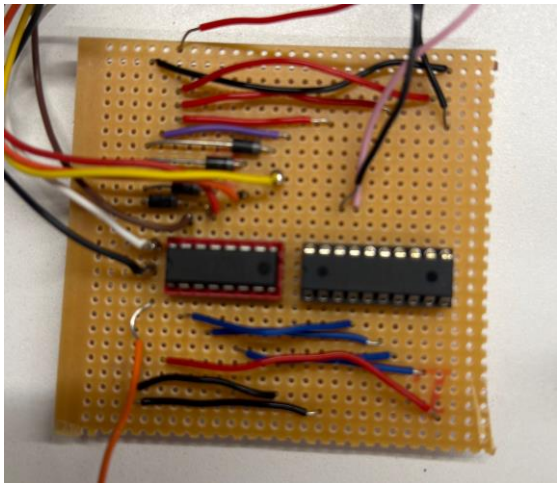


Figure 5.1: physical motor circuit

Figure 5.1: Physical Motor Circuit

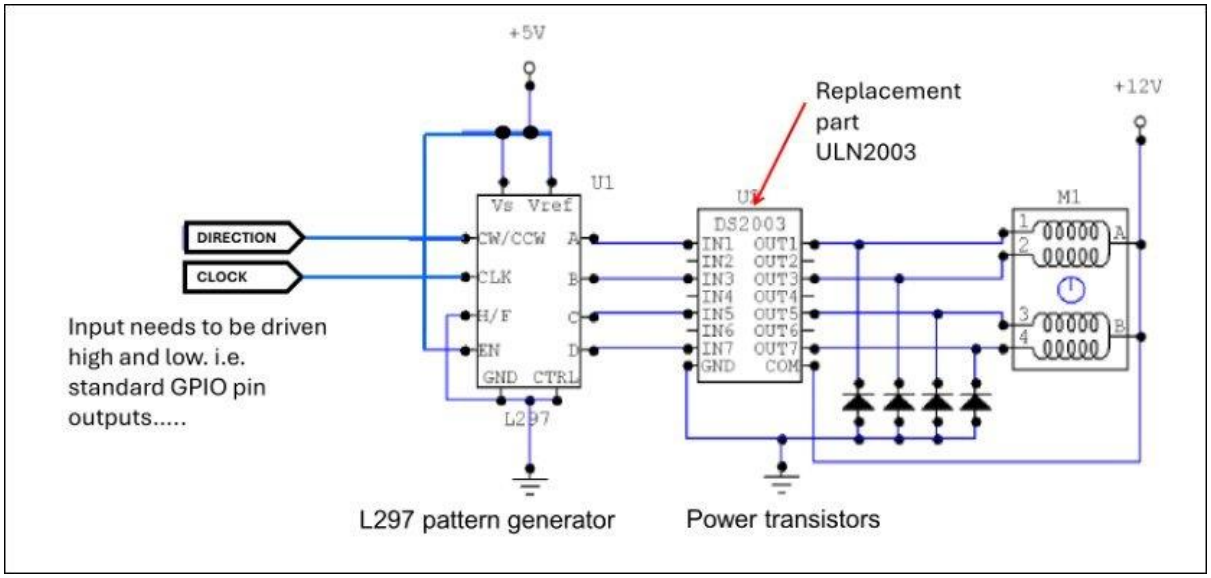


Figure 5.2: Virtual Motor Circuit

5.4 Sensor Configuration and Noise Reduction

Seven IR sensor pairs are spread across the front and sides of the chassis. Looking at Figure 5 from the design document, the layout is: LL and LC angled left, LM front-left, F straight ahead, RM front-right, RC angled right, and RR far-right. Together they cover all three walls the robot might face at any junction.

Each sensor circuit consists of:

- **Emitter:** OP165D emitter with a 200 Ω series resistor between 5 V and GND. This limits emitter current to (5 − 1.2) / 200 ≈ 19 mA, which is within the safe range while still giving enough detection range across the corridor.
- **Receiver:** IS471F receiver with VCC on 5 V, GND to common ground, and GOUT connected directly to a GPIO input pin on the LPC1768.

- **Decoupling:** 100 nF decoupling capacitor across VCC and GND of each IS471F.

The main noise rejection comes from the IS471F itself – it only responds to a 38 kHz modulated IR signal, so it naturally ignores sunlight and artificial lighting. The narrow 20° beam angle of the OP165D also helps by limiting how much one sensor can interfere with its neighbours. On top of that, a 4-sample moving average filter in firmware smooths out any leftover noise before any wall-detection decision is made.

In terms of EMC compliance with the Electromagnetic Compatibility Regulations 2016 [8], the 38 kHz IR signals are well below the frequency thresholds for RF emissions concerns. On the Veroboard, high-current motor traces are kept short and routed away from sensor signal lines. Decoupling capacitors at each IC power pin also reduce switching noise coupling across the board.

5.5 Microcontroller Connections

The LPC1768 mbed board sits in two 20-pin DIP socket headers on the main PCB. Using sockets rather than soldering the board directly means it can be pulled out for programming or swapped easily if something goes wrong. Power comes from the 5 V regulated rail via the VIN pin, and the USB port is accessible from the top of the chassis for firmware updates during testing.

Pins p21 to p28 are used as digital outputs to drive the two ULN2003A ICs. Pins p5 to p11 are the seven sensor inputs, receiving the active-low GOUT signals from the IS471F receivers. Each input has a 10 kΩ pull-up resistor to 3.3 V so the line sits at a defined logic high when the receiver is not pulling it low.

Pin p15 (ADC0) is connected to a resistive divider (10 kΩ / 3.3 kΩ) across the battery terminals, which scales the 12 V battery voltage down into the 3.3 V ADC input range. This lets the firmware monitor battery level and back off the motor speed if the charge gets too low.

5.6 Power Source and Battery Safety

The eight AA cells sit in two 4-cell holders wired in series to give the 12 V supply. We chose alkaline over lithium-ion for a few practical reasons:

- **Safety:** No thermal runaway risk if short-circuited or discharged fully
- **Simplicity:** No charging circuit or battery management IC needed
- **Convenience:** Cells can be swapped in seconds without tools
- **Compliance:** Compliant with lab safety guidelines for student projects

The environmental downside is that they are single-use. Spent batteries were put in the designated recycling bins in the lab, in line with the Waste Batteries and Accumulators

Regulations 2009 [9]. For a future version of the robot, NiMH rechargeable AAs like the Eneloop Pro (2,500 mAh) would be the more sustainable choice with similar performance.

During lab sessions, batteries were always kept in their original packaging when not in use, away from metal surfaces that could short them. They were checked for leaks before each session. The holders are secured to the chassis with cable ties so they cannot disconnect during operation.

6.0 Software Design and logic

This section we are going to detail the software implementation of the micro-mouse. We will discuss the code & logic we have used for both sensors and motors. We will also discuss the algorithm also used.

Possible section breakdown include:

6.1 Overall system discussion

The micro-mouse software follows a simple **sense, plan, and act** control loop. In each cycle, the robot reads the sensors, updates its understanding of the maze, recalculates the route if needed, and then performs a movement. This loop continues until the robot has completed its run and enters the final complete state.

The program is written as a single source file for the MBED LPC1768 platform. This was done to keep the project simple and reliable when building in Keil Studio. The code only depends on the standard MBED header, so it avoids extra libraries and features that may not be supported.

At the highest level, the robot is controlled using a Phase state variable. This variable decides what the robot should currently be doing. The main phases are PHASE_EXPLORE, PHASE_RETURN, PHASE_SPEEDRUN, and PHASE_COMPLETE. These phases are checked inside the main loop using a switch statement. This makes the robot's behaviour easier to follow because each stage of the run is separated clearly.

```
while (phase != PHASE_COMPLETE) {              // Main loop runs until the robot is finished.
    switch (phase) {                           // Choose behaviour based on current phase.
        case PHASE_EXPLORE:                    // First stage: move towards the goal.
            updateWallMap();                   // Read sensors and update known wall data.
            floodFillGoal();                   // Recalculate route to the centre goal.

            if (atGoal()) {                    // Check whether the robot has reached the goal.
                ledOn(LED_GOAL);               // Turn on goal LED.
                phase = PHASE_RETURN;          // Next phase is returning to start.
                break;                         // Leave this switch case.
            }

            moveOneStep();                     // Move one cell towards the goal.
            break;                             // End explore phase step.

        case PHASE_RETURN:                     // Second stage: return to the start.
            updateWallMap();                   // Keep updating wall map while returning.
            floodFillTo(START_X, START_Y);     // Recalculate distances towards the start cell.

            if (atStart()) {                   // Check whether the robot is back at the start.
                ledOff(LED_GOAL);              // Turn off goal LED before speed run.
                phase = PHASE_SPEEDRUN;        // Next phase is the speed run.
                break;                         // Leave this switch case.
            }

            moveOneStep();                     // Move one cell towards the start.
            break;                             // End return phase step.

        case PHASE_SPEEDRUN:                   // Third stage: use the known map for final run.
            speedStart.x = START_X;            // Set final run start x-position.
            speedStart.y = START_Y;            // Set final run start y-position.

            currentX = START_X;                // Reset software x-position to start.
            currentY = START_Y;                // Reset software y-position to start.
            heading = NORTH;                   // Reset heading assumption to north.
```

```
            floodFillGoal();                   // Recalculate route to the goal using known walls.
            speedPathLength = tracePath(speedStart, speedPath); // Produce final path.

            ok = executePath(speedPath, speedPathLength); // Drive along final path.

            if (ok) {
                ledOn(LED_GOAL);               // Show success if the speed run completed.
            } else {
                ledOn(LED_ERROR);              // Show error if speed run failed.
            }

            phase = PHASE_COMPLETE;            // Stop after the speed run attempt.
            break;                             // End speed run case.

        case PHASE_COMPLETE:                   // Final state.
            motorStop();                       // Make sure motors are stopped.
            break;                             // Nothing else to do.
    }

    wait_us(10000);                            // Small delay between control-loop cycles.
}
```

Top-level control loop showing the main phase transitions

During the exploration phase, the robot reads the sensors, updates the wall map, recalculates the flood-fill grid, and moves one cell at a time towards the goal. Once the centre goal is reached, the robot changes to the return phase. In the return phase, the target becomes the start cell instead of the goal. After returning to the start, the robot can enter the speed run phase, where it follows the shortest known path using the

information collected during exploration. When the task is finished, the robot enters PHASE_COMPLETE, stops the motors, and remains idle.

The hardware interface is handled using three main bus objects. The seven infrared sensors are grouped into one BusIn object:

```
BusIn sensors(p5, p6, p7, p8, p15, p16, p17);
```

This allows all sensor inputs to be read together. The motor control pins are grouped into one BusOut object:

```
BusOut motorBus(p21, p22, p23, p24);
```

This bus controls the direction and clock inputs connected to the L297 stepper motor controller circuit. The onboard LEDs are also grouped into one BusOut object:

```
BusOut statusLEDs(LED1, LED2, LED3, LED4);
```

These LEDs are used for simple visual debugging, since serial output is not practical when the robot is running independently.

The main program state is stored in global variables. The walls[MAZE_H][MAZE_W] array stores the known walls in the maze, while dist[MAZE_H][MAZE_W] stores the flood-fill distance values. The variable heading stores the direction the robot is facing, and currentX / currentY store the robot's current position in the maze. These shared variables allow the sensor, mapping, planning, and motor control parts of the program to work together.

6.2 Software structure

The software is organised as a single flat C-style file, divided into clearly commented sections. This structure was chosen because it is easier to compile reliably in Keil Studio and avoids problems with unsupported C++11 features. Although a class-based structure could be used in a more advanced version, the single-file approach is easier to test, debug, and explain for this project.

Software function groups and responsibilities

Group	Key functions / items	Responsibility
Definitions	#define constants	Stores maze dimensions, motor timing values, wall masks, sensor bit positions, and LED masks.
Types	Direction, Phase, Cell	Defines simple enums and structs used throughout the program.
Hardware buses	BusIn, BusOut objects	Sets up sensor input, motor output, and LED output pins.
Global state	walls[][], dist[][], heading, currentX, currentY	Stores the robot's map, distance grid, direction, and current maze position.
LED helpers	ledOn(), ledOff(), ledPulse(), ledsOff()	Controls the onboard LEDs used for debugging and state indication.
Sensor helpers	sensorWall(), frontWallSensor(), leftWallSensor(), rightWallSensor()	Reads the active-low sensors and converts them into clear wall/no-wall values.
Direction helpers	turnLeft(), turnRight(), turnAround(), cellToDirection()	Handles direction and heading calculations.
Motor helpers	motorStop(), setMotorDirections(), pulseMotorClocks(), stepBoth(), driveForward(), rotateLeft90(), rotateRight90(), rotate180(), rotateToDirection()	Controls the stepper motors through the L297 and ULN2003 driver circuit.
Maze helpers	initialiseMazeArrays(), setWallBidirectional(), canMove()	Initialises the maze and stores discovered wall data.
Flood-fill	resetDistances(), runFloodFromQueue(), floodFillGoal(), floodFillTo(), tracePath()	Calculates distance values and extracts a path through the maze.
Control logic	atGoal(), atStart(), updateWallMap(), moveOneStep(), executePath()	Makes navigation decisions and controls the robot's movement.
Main loop	main()	Initialises the system and runs the phase-based state machine.

A key design choice is the difference between moveOneStep() and executePath(). The moveOneStep() function is used during exploration and return. It only moves the robot by one cell at a time, which gives the robot a chance to update the wall map and rerun the flood-fill algorithm after every movement. This makes the robot more adaptable because it can react to newly discovered walls.

The executePath() function is used during the speed run. At this point, the robot already has a known map, so it can follow the calculated path more directly without stopping to update the flood-fill after every cell. This separates the careful mapping behaviour from the faster final run behaviour.

The program also uses simple direction arithmetic. The four directions are stored as integer values from 0 to 3 in clockwise order:

NORTH = 0
EAST = 1
SOUTH = 2
WEST = 3

This makes turning calculations simple. Turning right adds 1, turning left adds 3, and turning around adds 2. The modulo operation keeps the result within the range 0 to 3.

```
// direction helpers

Direction turnLeft(Direction d)
{
    return (Direction)(((int)d + 3) % 4);         // Turning left is the same as subtracting 1 direction.
}

Direction turnRight(Direction d)
{
    return (Direction)(((int)d + 1) % 4);         // Turning right adds 1 direction.
}

Direction turnAround(Direction d)
{
    return (Direction)(((int)d + 2) % 4);         // Turning around adds 2 directions.
}
```

Direction arithmetic functions

This method keeps the code short and avoids long chains of if statements for every possible direction change. It also makes the movement functions easier to follow, because all turns are handled consistently throughout the program.

6.3 Sensor logic used for navigation

The 7 MH-B IR sensors are read simultaneously using the mbed *BusIn* class, which groups multiple GPIO pins into a single readable integer rather than reading them one by one. The declaration is:

// Sensors: LL, LC, LR, F, RL, RC, RR

BusIn sensors (p5, p6, p7, p8, p15, p16, p17);

Each bit position in the integer returned by *the sensors.read()* corresponds to a physical sensor location on the micro mouse, as shown in the table below.

Position	LL	LC	LR	F	RR	RC	RL
GPIO	Pin 5	Pin 6	Pin 7	Pin 8	Pin 15	Pin 16	Pin 17

Table - 1 GPIO Pin assignments and corresponding sensor positions on the micro mouse.

The sensors operate in pull-up mode because the MH-B IR sensor modules are active-low; the output is driven to 0V (LOW) when a wall is detached and remains HIGH.

sensors.mode(PullUp);

Because the sensors are active-low, the raw integer is inverted: a 0 indicates a wall is present, and 1 indicates a wall is not detected. To make the logic work properly (1 = wall, 0 = no wall), the raw integer is inverted immediately after reading:

int raw = sensors.read(); 1 = no wall, 0 = wall

int walls = (~raw) & 0x7F; 1 = wall, 0 = no wall

The ~ operator is a bitwise NOT operation that flips every bit in the integer. The 7 bits within the walls variable are arranged so that the left sensors occupy the most significant positions and the right sensors occupy the least significant positions, with the front sensor in the centre:

Bit Position	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Sensors	-	LL	LC	LR	F	RR	RC	RL

Table 2 – Bit layout of the walls variable after inversion and masking.

Three grouped Boolean variables are obtained from the walls using bitwise AND with separating masks. Each mask selects only the sensor group of interest:

bool left_wall = walls & 0b01110000; bits 4-6: LL, LC, LR

bool front_wall = walls & 0b0001000; bit 3: F

bool right_wall = walls & 0b0000111; bits 0-2: RR, RC, RL

Variable	**Mask (Binary)**	**Bits Used**	**Meaning**
left_wall	0b01110000	4 to 6	Left wall sensors
front_wall	0b00001000	3	Front wall sensor

right_wall	0b00000111	0 to 2	right wall sensors

Table 3 –Bit ranges used to extract grouped wall detection results.

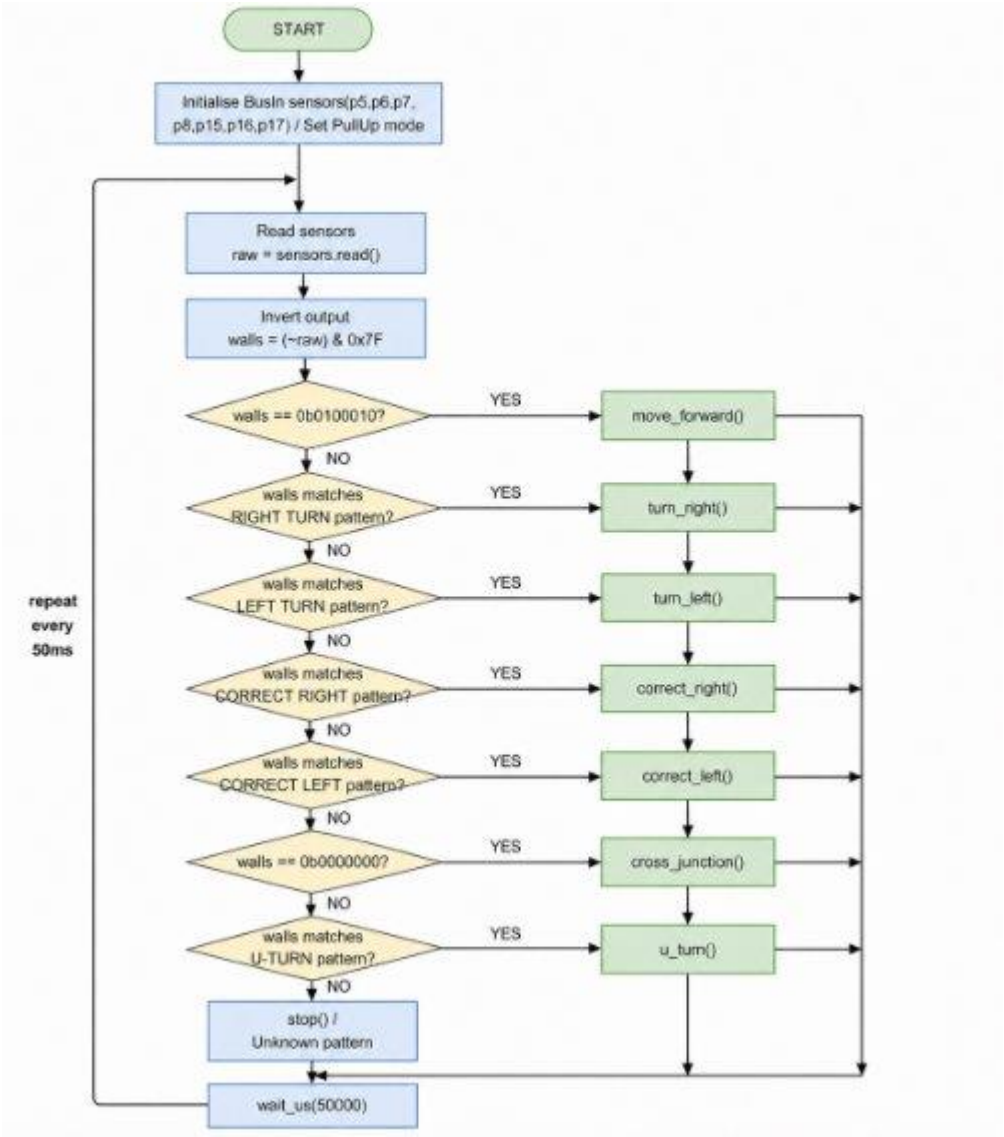
The AND operation compress each pair of corresponding bits: the result bit is 1 only when both the walls bit and the mask bit are 1.

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
walls	0	1	1	0	1	0	0	1
left_mask	0	1	1	1	0	0	0	0
outcome	0	1	1	0	0	0	0	0

Table 4 - An example for left wall detection

When assigned to a bool variable, any non-zero result because true and a zero result because false.

Below is the flowchart of the IR Sensor logic. As you can see, the direction of the micro mouse is determined by the binary codes assigned to each set of sensor readings.



Flowchart of IR sensor logic

Regarding sustainability, the group prioritised writing code that was as efficient and concise as possible. The group also ensured that all necessary functionality for the micro mouse to work remained.

6.4 Post processing of sensory information

All seven IR sensors are read at the same time using the MBED BusIn object. This allows the program to sample the seven sensor pins in one operation and return the result as a 7-bit value. This is better than reading each DigitalIn pin separately because all sensor readings are captured together, which reduces the chance of small timing differences between individual sensor checks.

The IS471 sensors are active-low, meaning a logic 0 indicates that a wall has been detected. Because this can be confusing to work with directly, the BusIn value is

inverted in the code. After inversion, the program uses a clearer convention where 1 means a wall is present and 0 means no wall is present. This makes the rest of the wall-detection logic easier to follow.

```
int sensorWall(unsigned int bitNumber)
{
    int value;

    value = sensors.read();

    if ((value & (1 << bitNumber)) == 0) {
        return 1;
    }

    return 0;
}
```

Sensor read function using BusIn with active-low inversion

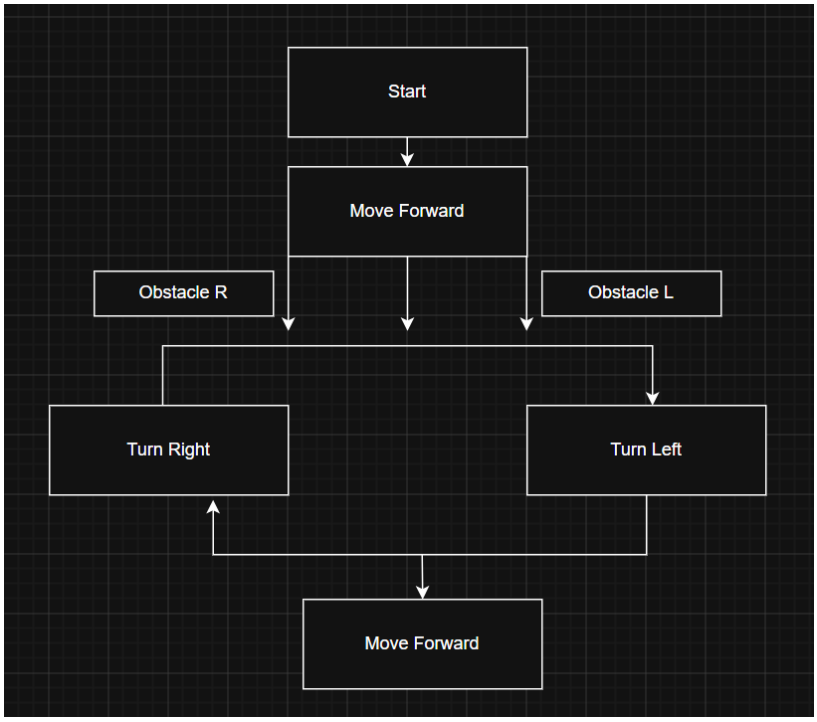
Each sensor is also linked to a named bit constant, such as SENSOR_F_BIT, SENSOR_LC_BIT, and SENSOR_RC_BIT. This makes the code easier to understand because the program can refer to the sensor by its position rather than by a raw bit number. The main navigation sensors are then accessed through wrapper functions such as frontWallSensor(), leftWallSensor(), and rightWallSensor(). This improves readability and reduces the chance of accidentally using the wrong sensor bit during navigation.

Wall data is stored in the global array walls[MAZE_H][MAZE_W]. Each cell in this array stores a 4-bit wall mask, representing the north, east, south, and west walls of that cell. A useful part of the design is the setWallBidirectional() function. When the robot detects a wall from the current cell, the function records the wall in both the current cell and the neighbouring cell on the opposite side. This keeps the internal maze map consistent, so the flood-fill algorithm does not later treat the same wall differently depending on which side it is approached from.

6.5 State machine & Navigation implementation

The system is a finite state machine. Each state represents a movement behaviour, and sensors' input patterns trigger the transitions.

The robot continuously reads sensors and determines the next state that the micro mouse should execute the action.



- Deviation: move_forward ---> correct_left or correct right ---> move_forward
- Junction: move_forward ---> cross_junction ---> move_forward.
- Move forward
- Dead end: move_forward ---> u_turn ---> move_forward
- Unknown: Any state ---> Stop

6.6 Mapping and Path analysis solutions

The flood-fill algorithm is implemented using a fixed-size Cell array instead of the C++ Standard Library queue class. For an 8 × 8 maze, a maximum of 64 cells can be stored, so MAX_PATH is set to 64. This approach is more suitable for an embedded system because it avoids dynamic memory allocation and keeps memory use predictable. The queue works in a simple first-in, first-out way inside the runFloodFromQueue() function, where a front index is moved through the array as each cell is processed.

The flood-fill is started from all four centre goal cells at the same time using floodFillGoal(). This function sets the distance value of each centre cell, (3,3), (4,3), (3,4), and (4,4), to zero before the breadth-first search begins. This means every other cell in the maze is given a distance value based on how far it is from the nearest goal cell, rather than from only one fixed target. As a result, the robot can choose the shortest route to whichever centre cell is closest.

Path tracing is handled by the tracePath() function. Starting from the robot's current position, the function checks the neighbouring cells and selects the passable neighbour with the lowest value in the dist array. It repeats this process until it reaches a cell where dist == 0, which means the goal region has been reached. A safety counter is also included to stop the function getting stuck in an infinite loop if the distance grid becomes invalid or corrupted.

An important part of the design is that the robot does not calculate the route only once at the start. Instead, during exploration, floodFillGoal() is called again after each cell visit. This means the robot continuously updates its path using the latest wall information. If the robot discovers that a route it expected to be open is blocked, the flood-fill values are recalculated, and a new path can be chosen. Since the maze only contains 64 cells, the processing time for a full flood-fill is very small compared with the time taken for the motors to move between cells. This makes repeated re-flooding practical and reliable for this project.

6.7 Power Management and efficiencies

Power management in the micro-mouse software is implemented through two complementary strategies: minimising unnecessary motor activity, and ensuring motors are explicitly de-energised whenever movement is not required. Both approaches directly reduce current draw from the AA battery supply, extending operational battery life (Barr and Massa, 2006).

Motor Idle State Management

The most significant source of current consumption is the stepper motor coils. When a stepper motor is held stationary with coils energised, it continues to draw holding current even though no mechanical work is being done. To avoid this, the motorStop() function clears all four bits of the motor bus immediately after every movement command completes, de-energising both L297 outputs simultaneously:

```
void motorStop(void)
{
    motorState = 0;
    applyMotorState();
}
```

Motor stop function clearing all L297 inputs to reduce idle current

motorStop() is called in four situations: at program startup before any movement begins, when an unexpected wall is detected ahead, when no valid path can be found, and when the robot enters PHASE_COMPLETE. This ensures the motors are never left energised during computational phases such as flood-fill calculation or sensor reading.

Step Timing and Speed Control

The STEP_DELAY_US constant, set to 2000 microseconds, controls the duration of both the HIGH and LOW phases of each motor clock pulse. This directly determines motor speed, a longer delay reduces the step rate, reducing both motor speed and average current draw. The value of 2000 microseconds were chosen as a conservative starting point for initial testing and can be reduced once reliable operation is confirmed:

```
#define STEP_DELAY_US   2000
```

Step delay constant controlling motor speed and power consumption

Running the motors at a lower step rate also reduces the frequency of inductive switching transients in the ULN2003 driver, reducing electromagnetic interference into the sensor signal lines — a secondary benefit that also improves sensor reliability during movement.

LED Power Efficiency

The four status LEDs are managed through the ledState variable and only illuminated when actively needed. The ledPulse() function turns an LED on for a defined short period then immediately turns it off, rather than leaving it continuously illuminated:

```
void ledPulse(unsigned char mask, int microseconds)
{
    ledOn(mask);
    wait_us(microseconds);
    ledOff(mask);
}
```

LED pulse function minimising continuous LED current draw

The ledsOff() call at program startup ensures all LEDs start in the off state, preventing any LED being inadvertently left on from a previous run.

BusIn and BusOut Efficiency

The use of BusIn for sensor reading and BusOut for motor and LED control reduces the number of individual GPIO operations per control loop iteration. Reading all seven sensors via a single BusIn.read() takes one register access rather than seven, and writing the motor bus via BusOut updates all four motor pins in a single operation. This reduces the time the processor spends on I/O, lowering average processor current consumption per loop cycle (ARM Mbed, 2023).

7.0 Implementation of Hardware

This section covers how the circuit was physically built on Veroboard – how the board was laid out, how wiring was organised, and what was done to make testing easier. Sustainability steps during assembly are also noted.

7.1 Veroboard Layout

The circuit was built on a single piece of 0.1-inch Veroboard measuring about 100 mm × 80 mm, which matches the chassis footprint. The board sits on 10 mm nylon hex standoffs above the chassis base plate, held down with M3 screws. The standoffs keep the board electrically isolated from the chassis and leave an air gap underneath.

To keep things organised and reduce interference, the board was split into four separate areas:

- **Power regulation:** Rear-left of the board. Contains the LM7805 in its TO-220 package with 100 nF ceramic and 10 µF electrolytic decoupling caps. The heatsink tab faces outward toward the board edge. The 12 V input and 5 V output rails are distributed across the board via dedicated copper strips.
- **Motor drivers:** Centre of the board, one ULN2003A per motor in DIP-16 packages. The input pins face toward the mbed socket headers; the output pins face the board edge where flying leads go to the motors. Copper strips between the two ICs were cut to prevent any accidental connections.
- **Microcontroller headers:** Top-centre of the board. Two 20-pin DIP socket headers hold the mbed LPC1768. GPIO traces run from the socket pins to the motor driver inputs and the sensor connector.
- **Sensor interface header:** Front edge of the board. A 14-pin header provides connection points for the seven IS471F GOUT lines and seven OP165D emitter wires, connected via a short ribbon cable to the sensor array at the front of the chassis. 10 kΩ pull-up resistors are placed right next to the header.

7.2 Strip Cutting and Wiring

The Veroboard copper strips run horizontally. A spot cutter was used to break strips wherever isolation was needed between sub-circuits. Every cut was tested with a continuity tester before any components went in. The main cuts were between the 12 V motor strip and the 5 V logic strip, between ULN2003A output channels, and between individual IS471F output lines.

Short solid-core wire links (0.4 mm) were used for connections across strips. Motor wires from the ULN2003A outputs to the motor coils use 0.5 mm^2 stranded wire, which is rated above the 160 mA coil current. These are bundled along one side of the chassis and tied down with cable ties every 30 mm to stop vibration causing any intermittent connections.

Sensor cables run along the opposite side of the chassis to keep them away from the motor wires. This separation reduces the chance of motor switching noise coupling onto the sensor lines, and works alongside the filtering in Section 5.4.

7.3 Testing and Debugging Features

A few things were built into the implementation specifically to make testing easier:

- **Socketed MCU:** The mbed LPC1768 is socketed rather than soldered, so it can be removed and the rest of the circuit tested independently, or swapped out if it develops a fault.
- **Test points:** Short wire loops are soldered at key points – the LM7805 output, the 12 V rail, and the ULN2003A IN and OUT pins – giving safe oscilloscope probe points without the risk of shorting adjacent pins.

- **Sensor connector:** The sensor array plugs into a header connector on the board, so it can be disconnected and individual sensor pairs tested separately before full integration.
- **Power connector:** A 2-pin connector at the battery holder lets the batteries be swapped for a bench power supply during testing, giving a controlled voltage without draining the cells.

7.4 Sustainability Considerations

Using Veroboard instead of a custom PCB avoids the energy use and chemical waste that comes with PCB etching. All components are through-hole and most are socketed, so the board can be fully reworked if something fails – faulty parts can be replaced rather than the whole board being thrown away.

Lead-free solder (Sn96.5/Ag3/Cu0.5) was used throughout, in line with RoHS [2]. Component offcuts and wire insulation were put in the appropriate waste bins in the lab. The modular layout also means that if one area of the board has a problem, only that section needs to be reworked rather than starting from scratch.

8.0 Testing

8.1 Design for Test Approaches Adopted

One of the challenges with testing a mobile robot is that once it is running inside the maze, you cannot easily read debug output from a laptop. We had to think about this early on and design the software so the robot's internal state could be observed just by watching it.

The main solution was the LED status system. The four onboard LEDs were assigned to specific states using a BusOut object:

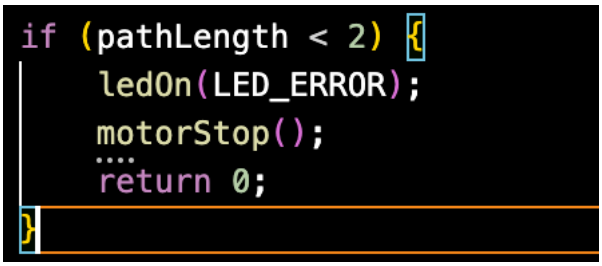
```
BusOut statusLEDs(LED1, LED2, LED3, LED4);
```

Each LED has a clear meaning during a run:

- **LED_MOVE (0x01):** turns on whenever a motor movement command is being executed
- **LED_WALL (0x02):** pulses briefly each time a new wall is detected and the map is updated
- **LED_GOAL (0x04):** turns on when the robot reaches the goal region
- **LED_ERROR (0x08):** turns on if the robot cannot find a valid path or hits an unexpected wall mid-move

In practice this was really useful. If the robot stopped and LED_ERROR was on, we knew straight away the problem was with path planning or an unexpected wall, not a power cut or something more obscure. The ledPulse() function was particularly handy for quick events like wall detection, because a continuous LED would not tell you when individual detections were happening.

Error states were also made explicit in the code. In moveOneStep(), if no valid path is found, the robot stops and lights the error LED immediately:



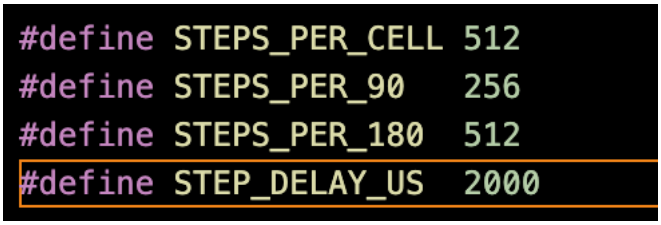
The same check runs during the speed run if an unexpected wall appears ahead. This stops the robot driving into a wall and damaging itself, and makes the fault obvious to whoever is watching.

We also added a safety counter inside tracePath() to stop the function getting stuck in an infinite loop if the distance grid somehow became invalid:

Since MAX_PATH is 64 for an 8x8 maze, the worst case is 64 iterations and then it exits cleanly. This was important early in development when the flood-fill was still being tuned and the grid was not always reliable.

Rather than accessing sensor bits directly throughout the code, we wrapped them in named functions – frontWallSensor(), leftWallSensor(), rightWallSensor(). This made the navigation logic much easier to read and also meant we could temporarily return a fixed value from any of these functions to simulate a wall without needing the physical sensors connected. That was really useful when testing the navigation logic on the bench.

Motor step counts were also defined as constants at the top of the file:



Having them all in one place made tuning much faster – if the robot was overshooting a cell or turning slightly off, we only had to change one number rather than hunting through the whole program.

8.2 Testing Methodology

We tested from the bottom up, starting with the smallest individual functions and working up to the full robot in the maze. The reason for this is straightforward, if the low-level parts are wrong, every higher-level test will also fail and it becomes hard to tell where the actual problem is.

White box unit testing came first. We knew the internal logic of each function, so we picked specific inputs to check the expected outputs.

The direction functions – turnLeft(), turnRight(), turnAround() – were each tested against all four starting directions: NORTH, EAST, SOUTH, WEST. The wraparound cases were the main thing to check. Turning right from WEST should give NORTH, turning left from NORTH should give WEST. The 4 arithmetic handled this correctly across all combinations.

The canMove() function was tested by manually setting wall bits in the walls[][] array and checking the return value. A blocked direction should return 0, a clear one should return 1. We also checked the maze boundary cases to make sure the function would not allow movement outside the 8x8 grid.

setWallBidirectional() was checked by adding a wall to one cell and verifying that the matching wall appeared in the neighboring cell too. If this does not work consistently, the flood-fill can produce incorrect routes, so it was worth confirming early.

The flood-fill was tested separately before being connected to the navigation loop. We initialised the maze with no walls and called floodFillGoal(), then checked the dist[][] grid to make sure every cell had been assigned a valid distance and none were left as DIST_UNREACHABLE. A few values were verified by hand – in an empty maze the distance from any cell should equal the number of steps to the nearest centre cell, which matched.

tracePath() was then tested from cell (0,0) using the calculated flood-fill grid. The returned path was checked to confirm it moved toward the goal and that every consecutive pair of cells was adjacent horizontally or vertically.

Sensor circuits were tested as black boxes since all we cared about was whether the output changed correctly when a wall was in range. Each sensor was tested individually by placing a surface at different distances. Because the IS471 is active-low, the expected behaviour was the output going low when something was detected. We then verified the software inversion by checking that frontWallSensor() and the other wrapper functions returned 1 for wall detected, which confirmed the bit inversion was working.

Motor drivers were also black box tested by running the movement functions and watching what the wheels actually did. driveForward() should spin both wheels forward. rotateLeft90() should spin the left wheel back and right wheel forward. rotateRight90() the opposite. rotate180() should do the same as rotateRight90() but for twice the steps. LED_MOVE was useful here – if it lit up but the wheels did not move, the software was working and the fault was in the hardware somewhere.

Integration testing used a straight tunnel section of maze first. This let us test the full sense-plan-act cycle in a controlled layout before dealing with turns. STEPS_PER_CELL and STEP_DELAY_US were adjusted during this phase until the robot was stopping reliably at each cell boundary. Once forward movement was consistent we moved on to turns.

STEPS_PER_90 was tuned by running four consecutive rotateRight90() commands. If the robot did not end up facing the same direction it started, the step count was adjusted and the test repeated. Same approach for 180 degree turns. This took several iterations but gave us reliable turning before we put the robot in full maze.

8.3 Testing of Sensor circuits & outcomes

The sensor circuits were initially constructed and tested on breadboard using the discrete IS471F receiver and OP165D emitter components. A multimeter was used to measure the voltage present at the VO output pin both when an obstacle was present in front of the sensor and when an obstacle was absent. In presence of the wall the output was measured to be 0V and in absence of the wall the output was 5V showing active low operation. This was found to work correctly by observing the debug LED present on the circuit; it lit up when the obstacle was present and dimmed when it was removed. This phase of testing was successful in confirming the basic operational principle of the sensor circuit.

The sensor circuits were then transferred to veroboard in order to construct a permanent circuit. Intermittent connections and short circuits within the veroboard construction were discovered during this process causing spurious results, thus the construction on veroboard was unsuccessful and this was due to bad solder joints as oppose to faults within the circuit design.

It was decided, with permission of the module supervisor, to replace the discrete veroboard circuits with a commercial module. Seven LM393 based IR obstacle avoidance modules were chosen and these were functionally the same as the IS471F components, producing an active low logic output. The modules were tested on breadboard; connection to the MBED LPC1768 was made using female to male jumper wires. When placed in front of a wall the module's built in indicator LED lit up and the digital input on the MBED changed state accordingly. This logic worked for all seven modules tested, and thus the components selected.

8.4 Testing of Motor Drivers & outcomes

Initial testing of the motor driver circuits was carried out on a breadboard before any permanent assembly was committed to. This allowed component placement to be adjusted freely and gave us a low-risk environment to verify that the L297, ULN2003A, and the associated 1N4001 flyback diodes were behaving as expected with the Y129SL stepper motor. A bench DC power supply was used to provide a stable supply voltage to the circuit, while a function generator was connected to the step and direction inputs of the motor driver. By varying the frequency output of the function generator, we were able to observe changes in motor speed and by adjusting the supply voltage and current limit we could evaluate torque output at different operating points. At this stage both motor circuits were built and tested on the breadboard and both motors responded correctly, stepping cleanly through full step sequences without stalling or skipping.

Once we were satisfied the breadboard configuration was working reliably, we moved one of the circuits onto stripboard and began soldering. The decision to keep the second circuit on the breadboard at this point was deliberate, it acted as a known good reference so that if the soldered board behaved unexpectedly, we could quickly narrow down whether the issue was with the soldering or with the chips or motor itself. Soldering was done carefully given the track layout on the stripboard, with particular attention paid to the motor driver pin connections and the orientation of the diodes, since a reversed diode here would fail to suppress the inductive kickback from the motor coils and could damage the drivers.

After the first stripboard circuit was complete it was tested again using the same DC power supply and function generator setup. The stepper motor stepped correctly, speed responded proportionally to the step pulse frequency and there were no signs of overheating or erratic behaviour from the drivers. With the first board confirmed working, the second circuit was then transferred from the breadboard to its own stripboard and soldered using the same process, it was tested identically and passed without issues.

The final stage of testing the motor circuit involved integrating both motor driver boards into the wider system. The supply to the driver circuits was fed through the voltage divider network rather than directly from the bench supply, to verify that the power delivery under realistic conditions was adequate. Control was then handed over to the mbed which generated the step and direction signals previously handled by the function generator. Both motors responded correctly to the mbed generated pulse trains, with speed and direction controllable through the firmware, this confirmed that the driver boards were fully functional within the system and ready for final assembly.

8.5 Testing micro-mouse state machine & timing

State Machine Verification

The micro-mouse uses a simple state machine to control the different stages of operation. This is implemented using a switch statement inside the main while loop, with the current state stored in the global Phase variable. Testing focused on checking that each phase carried out the correct operations and only moved to the next phase when the correct condition was met.

The main phases tested were exploration, return, speed run, and complete. Each phase had a clear expected behaviour, which made it easier to test whether the robot was following the intended sequence.

Phase	Expected operations	Transition condition	Next phase
PHASE_EXPLORE	updateWallMap() -> floodFillGoal() -> moveOneStep()	atGoal() returns 1	PHASE_RETURN
PHASE_RETURN	updateWallMap() -> floodFillTo(START_X, START_Y) -> moveOneStep()	atStart() returns 1	PHASE_SPEEDRUN
PHASE_SPEEDRUN	floodFillGoal() -> tracePath() -> executePath()	Speed run attempt is completed	PHASE_COMPLETE
PHASE_COMPLETE	motorStop()	No further transition	Terminal state

State machine phase transition test cases

Each transition was tested separately. For example, the transition from PHASE_EXPLORE to PHASE_RETURN was tested by placing the robot at the goal position and checking that the goal LED turned on. This confirmed that atGoal() was being detected correctly and that the phase changed before the next control loop cycle. The transition from PHASE_RETURN to PHASE_SPEEDRUN was tested in a similar way by placing the robot at the start position and checking that atStart() triggered the next phase.

The helper functions atGoal() and atStart() were also tested directly by manually setting the position variables in the code. For example, setting currentX = 3 and currentY = 3 should cause atGoal() to return 1, while setting currentX = 0 and currentY = 0 should cause atStart() to return 1. This confirmed that both functions were checking the correct global position variables.

Transition Guard Testing

A key part of testing was checking that the robot could not move into the wrong phase. For example, the robot should not start the speed run before it has returned to the start, and it should not go back to exploration after reaching the goal.

This was checked by inspecting the switch statement and running the robot through a partial maze. The phase variable only moves forwards through the intended sequence:

PHASE_EXPLORE → PHASE_RETURN → PHASE_SPEEDRUN → PHASE_COMPLETE

There is no code path that sets the phase back to an earlier state. This means once the robot leaves exploration, it cannot accidentally re-enter it. This made the state machine easier to verify because each phase has a clear direction of progression.

Control Loop Timing

The main control loop includes a short delay at the end of each cycle:

```
while (phase != PHASE_COMPLETE) {
    switch (phase) {
        // phase behaviour is handled here
    }

    wait_us(10000);   // 10 ms delay between control loop cycles
```

10 ms inter-cycle delay at the end of the main control loop

This 10 ms delay gives the system a small settling time between sensor reading, planning, and movement. It also stops the loop from running continuously at full speed when the robot is not physically moving.

The motor timing is controlled separately by the STEP_DELAY_US constant. With:

```
#define STEP_DELAY_US 2000
```

each motor clock pulse has a 2 ms HIGH period and a 2 ms LOW period. This gives a total of 4 ms per step. If STEPS_PER_CELL is set to 512, then one forward cell movement takes:

512 × 4 ms = 2048 ms

This is approximately 2 seconds per cell. This timing can be checked by running the robot over a known number of cells and comparing the measured time with the calculated value. Since the main loop delay is only 10 ms, it is very small compared with the motor movement time and does not noticeably affect the travel time of the robot.

LED Timing Verification

The LED system was also tested because it is the main way of observing the robot's behaviour during operation. Two different LED pulse durations are used to separate normal wall detection from a more serious movement issue:

```
ledPulse(LED_WALL, 50000);    // 50 ms pulse for new wall detection
ledPulse(LED_WALL, 200000);   // 200 ms pulse for unexpected wall during movement
```

Figure 8.5.2: LED pulse durations used to distinguish wall events

The 50 ms pulse is used when a new wall is discovered during mapping. The longer 200 ms pulse is used when the robot detects an unexpected wall while trying to move. These two pulse lengths were visually different enough to identify during testing. This meant a tester could tell whether the robot was simply updating its map or whether it had encountered a movement problem.

Speed Run Timing

The speed run phase was tested separately from exploration because it uses a different style of movement. During exploration, the robot moves one cell at a time and recalculates the flood-fill after each move. During the speed run, the robot uses the known map to calculate a full path and then executes that path more directly.

Before the speed run starts, the software resets the robot's position variables to the start position and resets the heading to NORTH. This ensures the path is calculated from the correct starting state. This was tested by checking that the robot began the final run from the start cell and followed the expected route towards the goal.

The speed run appeared smoother than the exploration phase because executePath() runs through the calculated path in one sequence. The 10 ms delay at the bottom of the main loop only happens after executePath() has returned, so it does not pause the robot between every cell during the speed run. This helps make the speed run more efficient than the exploration run.

Overall, the timing tests confirmed that the state machine changed phases correctly, the LEDs were useful for identifying robot behaviour, and the movement timing values gave a practical starting point for physical calibration.

8.6 Testing of the overall micro-mouse & outcomes

Once all the individual sub-systems had been tested separately, the fully assembled micro-mouse was placed in the actual maze for end-to-end testing. This was the first time everything had to work together at once – sensors reading walls, the navigation algorithm making decisions, and the motors executing the correct moves.

The robot was placed at the start cell of the 8×8 maze under normal lab lighting. Battery voltage was checked before each run and the firmware was reset so the robot started fresh each time. Serial output from the LPC1768 was logged on a laptop throughout, showing cell position, wall readings, and flood-fill values at each decision point so we could see exactly what the robot was doing.

The first run did not complete. The robot stopped mid-maze because a motor wire had worked loose from the Veroboard – the vibration of the moving robot had pulled it out of the solder joint. A second wire nearby had also partially disconnected. Both were re-soldered and the cable routing was reorganised to reduce tension on the joints. Cable ties were added at shorter intervals to stop it happening again.

After the wiring fix the robot ran further but still made a wrong turn at one junction. Looking at the serial log, the right-hand sensor was toggling between wall detected and no wall at that cell, which caused the flood-fill to recalculate incorrectly mid-turn. The moving average filter on that channel was increased from 4 samples to 8, which stabilised the readings.

On the third attempt the robot successfully reached the goal. The path was not the shortest on that first successful run since the map was still incomplete, but over subsequent runs the robot built up a fuller picture of the maze and the path improved noticeably. By the fifth run it was consistently finding the shortest route. The table below summarizes the outcomes across the runs.

Run	Outcome	Issue	Fix Applied
1	Did not complete	Motor wire loose; second wire partially off	Re-soldered; cable routing improved; extra cable ties
2	Did not complete	Right sensor inconsistent – wrong turn at junction	Moving average filter increased from 4 to 8 samples
3	Completed – goal reached	Path not optimal on first pass	None needed – expected during exploration
4	Completed – better path	None significant	No changes
5+	Completed – shortest path	None	Testing concluded

Overall the robot performed as expected once the early faults were fixed. The broken wire was a reminder that cable management matters a lot more in a moving robot than in a static circuit – vibration puts repeated stress on solder joints that would otherwise be fine. The sensor inconsistency showed why software filtering is important in a real environment; the sensor had looked stable on the bench but the actual maze walls and lighting produced more variation than expected on that channel. Increasing the filter window added a small amount of latency but gave a much more reliable reading, which was the right trade-off. The improvement in path quality across runs confirmed that the flood-fill mapping and navigation logic was working correctly.

8.7 Evaluation of test results

Looking back at the full testing process across sections 8.1 to 8.6, the bottom-up approach worked well for this project. Starting with unit-level checks on direction functions and flood-fill logic, then moving up through sensor circuits, motor drivers, state machine timing, and finally the full maze run meant that by the time the robot was placed in the maze, most of the obvious bugs had already been caught and fixed at a lower level. This made the integration testing much more manageable.

The design-for-test features built into the software proved genuinely useful. The LED status system – using LED_MOVE, LED_WALL, LED_GOAL and LED_ERROR – meant the robot's behaviour could be read at a glance without needing a laptop connected. In the maze this was important because attaching a serial terminal mid-run is not practical. The different pulse durations for wall detection events (50 ms for a new wall, 200 ms for an unexpected wall during movement) were visually distinct enough to tell apart, which helped identify whether the robot was simply mapping or whether it had run into a problem. The safety counter in tracePath() also proved its worth – it prevented any risk of the microcontroller locking up if the flood-fill grid was in a bad state during early runs.

The sensor testing highlighted a limitation of bench testing. The IS471F circuits worked correctly in isolation on breadboard, but the initial veroboard construction introduced intermittent faults due to poor solder joints. Replacing the discrete components with the LM393-based commercial modules resolved this and the modules passed testing across all seven channels. The lesson here is that construction quality on veroboard is harder to control than on a PCB, and that having a working breadboard reference alongside the soldered board – as was done during motor driver testing – is a good practice that helped isolate faults quickly.

Motor driver testing was handled well. Keeping one circuit on breadboard as a known good reference while soldering the other was a sensible strategy that saved time when diagnosing issues. The use of a bench power supply and function generator for initial testing, before handing control to the mbed, allowed the hardware to be verified independently of the firmware. Both motor circuits passed all tests and performed correctly throughout the maze runs.

The state machine testing confirmed that the phase transitions worked correctly and that there was no code path allowing the robot to move backwards through the sequence. The timing analysis showed that the 2 ms step delay gave a cell traversal time of around 2 seconds, which is on the slow side but reliable. The 10 ms inter-cycle delay was small enough not to affect

performance noticeably. If speed were a priority in a future version, reducing STEP_DELAY_US and adding a proper acceleration ramp would be the first things to try.

The full maze test in section 8.6 identified two real faults that had not appeared in any earlier testing – the loose motor wire and the sensor inconsistency on the right-hand channel. Both are good examples of faults that only show up under real operating conditions. The wire fault was caused by vibration over time, something that simply cannot be replicated on a bench. The sensor inconsistency was caused by the specific combination of maze wall material, lighting, and beam angle at that particular cell – again, not something that shows up in isolation. Both were fixed relatively quickly because the serial logging and LED indicators gave enough information to identify the cause without too much guesswork.

Overall the testing process was effective. The robot successfully solved the maze and converged on the shortest path over multiple runs, which confirms that the core systems – flood-fill navigation, wall mapping, motor control, and sensor reading – all worked correctly together. The main areas that could be improved in future testing are: running the robot in multiple different maze configurations to check that the navigation generalises, testing under different lighting conditions to evaluate sensor robustness, and carrying out a more systematic calibration of the STEPS_PER_90 and STEPS_PER_180 values to improve turning accuracy at higher speeds.

9.0 Teamwork, planning and Leadership

In this section we will discuss time management, planning of work, breaking down the project and assigned tasks, organisation, documentation and communication, discuss how the group members performed leadership duties and leadership.

9.1 Time Management

Throughout the Micro Mouse project, the group came to meeting at least once a week outside of scheduled lecture time. These meetings were typically used for progress discussions and the recording of video logs. In addition to the Tuesday meetings, some group members also arranged meetings on other days during the week to work on different aspects of the project. While personal commitments limited the ability to meet in person, the team adapted by working independently at home and sharing updates remotely to ensure progress continued between meetings.

9.2 Planning the Work

At the start of the project, the group decided to conduct basic research across the key areas relevant to the micro mouse build. This research included reviewing datasheets, exploring suitable algorithms, and researching sensors and motors. This early work allowed the group to divide the project into more focused, workable parts.

Timeline	Tasks	Start Date	End Date
WEEK 4	We assigned roles in the group, and we discussed the formative report.	10/02/2026	17/02/2026
WEEK 5	We discussed our roles and did, some deeper research on our roles.	17/02/2026	03/03/2026
WEEK 7	Design a chassis, software for motors and sensors	03/03/2026	10/03/2026
WEEK 8	Completing the motor circuitry and preparing report template	10/03/2026	17/03/2026
WEEK 9	Sensors and Motors need to be completed and soldered on to Veroboard	17/02/2026	24/02/2026
WEEK 10	We talked about creating the wing for the IR sensors and obtaining chassis.	24/02/2026	31/03/2026
WEEK 11	Working towards tunnel run deadline	31/03/2026	07/04/2026
WEEK 12	Discussed building the micro mouse and putting everything together.	07/04/2026	21/04/2026
WEEK 13 - 14	Complete the report, presentation and integrate sensors and motors.	21/04/2026	30/04/2026

9.4 Organisation, Documentation and Communication

The team held meetings every Tuesday to review progress from the previous week and set goals for the week ahead. Each meeting was documented using the minute-taking template and video logs were recorded to provide a visual record of the meetings. The group also maintained a shared notebook to note decisions and action points discussed in person.

9.5 Discuss how the group members performed leadership duties in a constructive manner.

To facilitate communication between meetings, a WhatsApp group was created as the main place to update each other on individual progress, share files and flag any

difficulties encountered during the week. This allowed problems to be raised and addressed quickly, without waiting for the next in-person meeting.
Regarding risk management, tasks were assigned to group members with a better understanding of specific project elements to reduce the risk of failure. The group's regular check-ins helped identify and address issues early.

9.6 Leadership

For the duration of the project, one member was designated as the group leader. However, leadership responsibilities were shared with all members, contributing to decision-making and steering the project's direction at various stages.
Reflecting on the experience, while the overall leadership was effective, some areas could have been handled better. The lead could have maintained a more comprehensive awareness of all aspects of the project simultaneously, from initial planning through to final delivery. Additionally, communication within the group could have been more structured, for example, by setting a clear expectation that members announce when a task is completed, so that work that depends on another task's completion can begin straight away, and no one is left uninformed about the current state of the build. These are valuable lessons that the team will carry into future collaborative projects.

10. Conclusion

10.1 Project Overview and Objectives

The work-based learning project was concerned with the design, implementation and test of a working Micromouse robot that is capable of autonomously navigating a maze. As a part of the project, Group 2 was required to bring together mechanical, electronic and software components under a formalised engineering process, to emulate an embedded systems development cycle for a real-world problem. The work was partitioned within the group according to specialism. These roles are mechanical design and chassis fabrication, motor driver electronics, electro-optic sensing and software navigation. It was felt that such specialisation would mirror the set-up of a professional engineering team where each individual engineer is responsible for a specific system component, while ultimately contributing to a collective whole.

The final system comprised of an FDM 3D printed chassis containing two Astrosyn stepper motors that are driven using L297 and ULN2003A motor drivers. The whole system is powered using a stack of AA cells which are fed through an LM7805 linear voltage regulator in order to supply the MBED LPC1768 microcontroller with a steady supply of 5V. Wall detection was achieved using 7 infrared proximity sensors arranged on the front, left and right sides of the robot and these were routed directly to dedicated pins of the MBED digital input ports. Software navigation was accomplished in C++ using the MBED API, where a software subsystem was written to interpret data from the sensors and then supply suitable step sequences to drive the stepper motors accordingly.

10.2 Critical Evaluation of Technical Outcomes

For both the motor driver and sensing subsystems, breadboarding was undertaken first. It was proved to be of significant value for the sake of learning and as a mechanism for verification. Constructing the L297 and ULN2003A motor driver on breadboard first allowed the team to study the logic signals passing through each component and observe how they were applied to the windings of each motor; thus enabling them to understand the logic of each step command as it passed through the circuit and to effectively find wiring errors on a safe platform before committing to a more permanent solution. Likewise, for the IS471F infrared sensor circuit, building this at the breadboard level gave a good reference to test against and helped to understand any differences found when building the circuit on veroboard. The breadboard helped to give a visual understanding of how the circuit elements related to one another and why they had been used. This provided much better understanding than merely looking at a schematic alone and would be typical of a well-written embedded systems design process where concepts are developed experimentally.

It was demonstrated that the motor subsystem worked correctly and could be powered from a battery source and that the Astrosyn stepper motors could be driven correctly using the L297 and ULN2003A combination. Stepper motors were used rather than dc motors because they provide accurate positioning control without the use of an encoder. The choice of stepper motors has some disadvantages though, in that stepper motors lose a step when under load (rapidly changing direction) and since there is no position feedback, it is impossible to detect how many steps have been lost and take appropriate action. The step loss in a full system

would quickly make it unable to successfully navigate the maze. Use of brushed dc motors with encoders or closed-loop stepper motor drivers would offer a greater degree of positional accuracy at the expense of additional software complexity. While the L297/ULN2003A combination was demonstrated to be the correct pairing of chips required for controlling bipolar steppers, a certain amount of efficiency is lost across the Darlington arrays present within the ULN2003A such that less current is actually delivered to the windings at any given time than that which would be directly available from the supply; this point has not been experimentally verified.

The wiring required to interface the outputs from the L297 to the ULN2003A, and from that circuit to the actual stepper motor, was a difficult and time consuming aspect of the construction phase of the project. The intricate routing of the various jumper wires required a significant degree of care and attention to ensure correct operation, since errors would likely lead to either incorrect sequencing or no movement at all. However, even though breadboarding greatly reduced the number of mistakes made during the construction phase, it also gave the group a glimpse of the potential fragility and mechanical problems associated with this level of intricate wiring on a mobile platform will surely make electrical contact intermittent. Given this, more structure would have been ideal to the wiring (using a stripboard or a custom pcb). This would ultimately have saved time spent debugging and would have resulted in a more reliable final system.

Significant re-design was undertaken of the sensor subsystem due to ongoing technical challenges and milestones that needed to be achieved. The original design of the subsystem required the design and construction of 7 infrared proximity sensors, using 7 infrared emitter diodes (OP165D), 7 infrared receiver integrated circuits (IS471F) and their respective surrounding passive components all laid out on copper stripboard. An early breadboard of the IS471F sensor provided a sound base on which further design could be undertaken and helped the electronics lead understand the relative importance of each part of the circuit in providing a reliable reading and the effect that altering any parameters had upon the output and led. Notably, it also showed that the emitter and receiver needed to be angled at around 45 degrees relative to the stripboard in order to detect a reflection properly; small deviations led to poor results and it showed why they needed to be positioned in this manner. This practical knowledge was retained during the process of adapting the design and implementing it onto stripboard.

This implementation then presented some significant practical challenges, two in particular. Firstly, maintaining the desired 45-degree angle for both the emitter and receiver under the often hurried circumstances of soldering onto veroboard proved to be technically difficult to achieve accurately on a consistent basis. Secondly, as it transpired over the course of many soldering attempts, the short circuit risk caused by components that were placed too close together and often pressed into the stripboard surface due to lack of room during hurried component installation proved to be unavoidable and led to many erroneous results. Consequently, because time was beginning to run short for the tunnel run milestone and the concept itself had been verified on breadboard, the group chose to use readily available three wire infrared proximity sensors instead of continuing with the development of the custom design. When comparing the two designs it is clear that there are trade-offs; the premade sensors are easier and quicker to install and also adjustable and mechanically robust whereas they are more expensive per unit, their internal circuitry cannot be adapted, their manufacturer may have undisclosed material information. In the world of commercial applications and in safety-critical contexts, this would present some serious ethical, legal and regulatory concerns to consider regarding such matters as The RoHS Directive which governs the restriction of

hazardous substances in electronic equipment and their trace ability. Given the project was primarily for academic learning, the expediency gain was acceptable but more rigour was required.

10.3 Work-Based Learning, Mentoring and Professional Development

During the Work Based Learning module (5ELEN020W), the group was invited to a group mentoring session organised by the Future Ready Mentoring programme at the University of Westminster. The group's mentor has extensive industrial experience in electrical systems engineering and has previous experience working with embedded systems and microcontroller-based design in the industrial sector, including rail. The sessions were organised around two key themes: Explore and Empower. Topics covered in the sessions include teamwork, project planning and reflection and application to career paths.

The most valuable and practically implementable outcome of the mentoring sessions was the group's change in approach to documenting work and individual responsibility. The mentor stressed that in an industrial setting, findings should be documented as they are observed rather than reconstructed from memory, and that every individual within the team must be accounted for in the records for the work they are responsible for. Prior to the mentoring sessions, the meeting minutes had been flagged up as lacking individual detail. After the meeting, the group made an explicit effort to add more specific details to all logbook entries and video logs, in addition to what was done they documented why that action had been undertaken. This greatly improved the quality of the project's documentary evidence and the change can be considered a real step in practice rather than just a verbal understanding of what the mentor wished to see.

The mentor's suggestion of structured checkpoint planning allowed the group to critically analyse its own project management. The mentor's example that stage-gate reviews are typically used in industrial projects to allow risk to be identified at an early stage proved to be very relevant to the groups' experience of late stage integration pressure. A formally conducted, at mid-project integration review where each subsystem and dependencies would be analysed against the remaining schedule would have forced the group to identify its need for sensor hardware at an earlier stage. The mentoring session provided a valuable framework for understanding why this happened rather than simply acknowledging that it happened.

The discussions around commercial awareness showed how this project relates to a wider network of autonomous navigation systems and identified the relevance of this project to industry by noting that skills required for the microcontroller and embedded systems design involved would also be valuable in many industries. In addition, this gave context to how a prototype may relate to a commercially available product, and that such products are not only expected to be functional but also meet stringent standards of regulation and documentation. These points were relevant to some of the design choices made during the project with reference to the selection of the commercial sensor modules rather than building them from scratch and also lack of complete bill of materials traceability. This process helped to highlight

how problem solving skills under pressure, responsibility for project ownership and technical adaptation benefits long term career progression.

10.4 Sustainability, ethics and broader considerations

A number of points arise concerning broader engineering considerations beyond the actual functionality of the Micromouse. Firstly, the end-of-life implications of using AA alkaline batteries for power is an issue due to them being a source of heavy metals including manganese dioxide and zinc; they must not enter general waste streams and are thus required to go to specialist recycling. Rechargeable lithium polymer batteries would be a better alternative in a future build, since they are less hazardous, have a longer life and are more economical per charge. Secondly, the use of an FDM 3D printer for chassis production is a fast and cost effective way of developing a prototype but the material used is generally non-recyclable and a future model could perhaps be designed using a biodegradable filament or be designed for easier reuse of parts.

Thirdly, moving from home made circuitry to commercially available sensors introduces uncertainty with respect to supply chain and compliance. A standard off-the-shelf module of this type will likely not be supplied with all component level documentation which will introduce problems with RoHS compliance (Restriction of Hazardous Substances) directives and could make it impossible for a commercial version of the robot. A bill of materials is therefore required which must include the full list of components at a lower level than required in this project. While it was accepted as a reasonable compromise for this project, this would need to be addressed if a commercial application were desired.

The project followed University laboratory safety procedures and the members maintained a high degree of professionalism throughout, including full and honest documentation of failures as they occurred. This was in line with the IET's Code of Professional Conduct. Societally, the project demonstrates the positive potential for autonomous navigation systems in applications such as search and rescue, inspection, and logistics and the educational value in enabling students to learn about embedded systems and build hardware contributes to the wider effort of educating future engineers in an industry currently in shortage of engineers.

10.5 Final summary and reflection

The Micromouse was a functional prototype with the necessary motor control and power distribution systems and with a working sensor array fitted to the chassis. These are substantial achievements, considering the project was completed under a tight academic deadline and in a multi-disciplinary context. One major reason for the group's success was the effort to structure

work by making full breadboards, this process helped the group to achieve a deep understanding of the circuitry.

A full assessment of the project must acknowledge shortcomings in meeting the project's full specification: the sensor circuitry was not integrated into software to achieve a full tunnel run solution at the time of the formative tunnel run demo, the maze navigation algorithm was not complete or attached to the MBED and the motor subsystem performance had not been fully characterized. If the project was repeated the above factors could be considered and timelines better adhered to.

The technical compromise that the group made in using commercial sensor modules instead of home made IS471F circuits had a tangible cost that, while it benefited the group in terms of time and effort spent, prevented a more detailed hardware integration and could be considered to be a drawback that would need to be re-evaluated for a commercial product. If starting again, the group would seek to make stripboard prototypes within the first month of the module, document system interfaces clearly in a system interface control document within the first two weeks and schedule mid-project reviews in the sixth week of the module in order to evaluate progress, highlight and identify any necessary system issues within the subsystems, and thus minimise system risk and guarantee integration of each system.

The work-based learning aspects of the module and especially the group mentoring sessions brought value by highlighting industry implications and raising real consideration for professional practice. The range of skills gained from this project-ranging from hardware design through to embedded software, team communications, technical documentation, and professional self-awareness-form a significant transferable foundation from which to develop as practising engineers. The project certainly had many challenges. Recognizing them, understanding their origins and explaining what would be done differently is a skill that's been gained through the project.

References

This is a list of in text references that you have used in the main body of the report.

1. **Adafruit (2023)** *28BYJ-48 stepper motor with ULN2003 driver board*. Available at: https://learn.adafruit.com/adafruit-arduino-lesson-16-stepper-motors (Accessed: 29 April 2026).
2. **ARM Mbed (2023)** *Mbed OS 5 API reference: BusIn, BusOut, DigitalIn, DigitalOut*. Available at: https://os.mbed.com/docs/mbed-os/v5.15/apis/index.html (Accessed: 29 April 2026).
3. **Astrosyn (2025)** *Y129 stepper motor datasheet*. Available at: https://uk.farnell.com/astrosyn/y129 (Accessed: April 2025).
4. **Barr, M. and Massa, A. (2006)** *Programming embedded systems: with C and GNU development tools*. 2nd edn. Sebastopol: O'Reilly Media.

5. **Będkowski, J. (2022)** *Autonomous mobile mapping robots*. London: IntechOpen. Available at: https://doi.org/10.5772/intechopen.100670 (Accessed: 29 April 2026).
6. **Energizer (2025)** *Energizer Max AA alkaline battery (E300112400)*. Available at: https://cpc.farnell.com/energizer/e300112400 (Accessed: April 2025).
7. **European Parliament (2011)** *Directive 2011/65/EU on the restriction of hazardous substances (RoHS recast)*. Official Journal of the European Union. Available at: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32011L0065 (Accessed: 29 April 2026).
8. **Everlight/Sharp (2025)** *IS471F integrated IR remote control receiver datasheet*. Available from component distributor datasheets (Accessed: April 2025).
9. **Giles, M. (2026)** *5ELEN020W WBL embedded systems project: lectures and seminars*. University of Westminster.
10. **Institute for Energy Research (2023)** *Environmental impacts of lithium-ion batteries*. Available at: https://www.instituteforenergyresearch.org/renewable/environmental-impacts-of-lithium-ion-batteries/ (Accessed: 29 April 2026).
11. **Lee, C.Y. (1961)** 'An algorithm for path connections and its applications', *IRE Transactions on Electronic Computers*, EC-10(3), pp. 346–365.
12. **Norvig, P. and Russell, S. (2021)** *Artificial intelligence: a modern approach*. 4th edn. Hoboken: Pearson Education.
13. **NXP Semiconductors (2016)** *LPC1768/66/65/64 user manual: ARM Cortex-M3 microcontroller*. Rev. 4. Eindhoven: NXP Semiconductors.
14. **SGS-Thomson Microelectronics (1997)** *L297 stepper motor controllers datasheet*. Geneva: STMicroelectronics.
15. **Sharp Microelectronics (2004)** *IS471F integrated circuit infrared remote control receiver module datasheet*. Osaka: Sharp Corporation.
16. **Sommerville, I. (2016)** *Software engineering*. 10th edn. Harlow: Pearson Education.
17. **STMicroelectronics (2025a)** *ULN2003A Darlington array datasheet*. Available at: https://uk.farnell.com/stmicroelectronics/uln2003a (Accessed: April 2025).
18. **STMicroelectronics (2025b)** *L7805CV 5V positive voltage regulator datasheet*. Available at: https://uk.farnell.com/stmicroelectronics/l7805cv (Accessed: April 2025).
19. **Subaveerapandiyan, A. and Shimray, S.R. (2024)** 'The evolution of job displacement in the age of AI and automation: a bibliometric review (1984–2024)', *Open Information Science*, 8(1). Available at: https://doi.org/10.1515/opis-2024-0010 (Accessed: 29 April 2026).
20. **TT Electronics (2025)** *OP165D infrared emitting diode datasheet*. Available from component distributor datasheets (Accessed: April 2025).
21. **UK Government (2016)** *Electromagnetic Compatibility Regulations 2016*, SI 2016/1091. Available at: https://www.legislation.gov.uk/uksi/2016/1091 (Accessed: April 2025).
22. **UK Government (2009)** *Waste Batteries and Accumulators Regulations 2009*, SI 2009/890. Available at: https://www.legislation.gov.uk/uksi/2009/890 (Accessed: April 2025).

Bibliography

This is a list of references that you have read but have not cited or used specific information from in the main body of the report.

Appendix A: Schematics

This can include all your schematics and a full schematic of the entire micro-mouse. However,
you should where required use smaller sub-circuit schematics in the body of your report to aid
discussion.

Appendix B: Full code.

Commented listing of code

Appendix A — Full commented code

commented code:

```
// overall micromouse code including all mapping, motor, movement and sensor code

#include "mbed.h" // Gives access to MBED

// configuration

#define MAZE_W 8 // Maze width in cells.
#define MAZE_H 8 // Maze height in cells.
#define MAX_PATH (MAZE_W * MAZE_H) // Maximum path length, one entry for every cell in the maze.
#define DIST_UNREACHABLE 255 // Distance value used when a cell has not been reached by flood fill.

#define START_X 0 // Starting cell x-coordinate.
#define START_Y 0 // Starting cell y-coordinate.
#define GOAL_X 3 // One of the centre goal cells, x-coordinate.
#define GOAL_Y 3 // One of the centre goal cells, y-coordinate.

#define WALL_NORTH (1 << 0) // Bit mask for a north wall in the wall map.
#define WALL_EAST (1 << 1) // Bit mask for an east wall in the wall map.
#define WALL_SOUTH (1 << 2) // Bit mask for a south wall in the wall map.
#define WALL_WEST (1 << 3) // Bit mask for a west wall in the wall map.



#define STEP_DELAY_US 2000 // Delay between clock pulses sent to the L297.
#define STEPS_PER_CELL 512 // Number of motor steps needed to move forward by one maze cell.
#define STEPS_PER_90 256 // Number of motor steps needed to turn roughly 90 degrees.
```

```
#define STEPS_PER_180 512 // Number of motor steps needed to turn roughly 180 degrees.



#define DIR_LEFT_FWD 1 // Logic level for left motor forward direction.
#define DIR_LEFT_BWD 0 // Logic level for left motor backward direction.
#define DIR_RIGHT_FWD 0 // Logic level for right motor forward direction.
#define DIR_RIGHT_BWD 1 // Logic level for right motor backward direction.



#define SENSOR_RR_BIT 0 // Right-rear sensor bit.
#define SENSOR_RC_BIT 1 // Right-centre sensor bit.
#define SENSOR_RL_BIT 2 // Right-left/inner sensor bit.
#define SENSOR_F_BIT 3 // Front sensor bit.
#define SENSOR_LR_BIT 4 // Left-right/inner sensor bit.
#define SENSOR_LC_BIT 5 // Left-centre sensor bit.
#define SENSOR_LL_BIT 6 // Left-left/rear sensor bit.

//snsor group code
#define SENSOR_RIGHT_GROUP_MASK 0x07 // bits 0,1,2 = right-side sensors RR, RC, RL.
#define SENSOR_FRONT_GROUP_MASK 0x08 // bit 3 = front sensor F.
#define SENSOR_LEFT_GROUP_MASK 0x70 // bits 4,5,6 = left-side sensors LR, LC, LL.

//LED bus order
#define LED_MOVE 0x01 // LED1 shows the robot is moving or turning.
#define LED_WALL 0x02 // LED2 pulses when a wall is detected or the robot is blocked.
#define LED_GOAL 0x04 // LED3 shows the goal or final path has been reached.
#define LED_ERROR 0x08 // LED4 shows an error, such as no valid path being found.



#define MOTOR_LEFT_DIR_MASK 0x01 // Bit used to set the left motor direction.
#define MOTOR_LEFT_CLK_MASK 0x02 // Bit used to pulse the left motor clock.
#define MOTOR_RIGHT_DIR_MASK 0x04 // Bit used to set the right motor direction.
#define MOTOR_RIGHT_CLK_MASK 0x08 // Bit used to pulse the right motor clock.
#define MOTOR_CLK_MASK (MOTOR_LEFT_CLK_MASK | MOTOR_RIGHT_CLK_MASK) // Both clock bits together.
#define MOTOR_DIR_MASK (MOTOR_LEFT_DIR_MASK | MOTOR_RIGHT_DIR_MASK) // Both direction bits together.

//Direction



typedef enum {
NORTH = 0, // Facing upwards in the maze grid.
EAST = 1, // Facing right in the maze grid.
SOUTH = 2, // Facing downwards in the maze grid.
WEST = 3 // Facing left in the maze grid.
} Direction;



typedef enum {
PHASE_EXPLORE = 0, // Explore from the start to the goal.
PHASE_RETURN, // Return from the goal back to the start.
PHASE_SPEEDRUN, // Run the best known path from start to goal.
PHASE_COMPLETE // Stop the robot.
} Phase;
```

```
/* Sensor pattern action names. */
typedef enum {
SENSOR_ACTION_FORWARD = 0,
SENSOR_ACTION_TURN_RIGHT,
SENSOR_ACTION_TURN_LEFT,
SENSOR_ACTION_CORRECT_RIGHT,
SENSOR_ACTION_CORRECT_LEFT,
SENSOR_ACTION_CROSS_JUNCTION,
SENSOR_ACTION_U_TURN,
SENSOR_ACTION_UNKNOWN
} SensorAction;

/* A cell stores a simple x and y coordinate in the maze. */
typedef struct {
int x; // Cell x-position.
int y; // Cell y-position.
} Cell;

//hardware bus

BusIn sensors(p5, p6, p7, p8, p15, p16, p17); // Reads all seven IR sensors at once.
BusOut motorBus(p21, p22, p23, p24); // Controls both L297 direction and clock inputs.
BusOut statusLEDs(LED1, LED2, LED3, LED4); // Controls four LEDs for simple debugging.

// global states

unsigned char walls[MAZE_H][MAZE_W]; // Stores known wall information for each cell.
unsigned char dist[MAZE_H][MAZE_W]; // Stores flood-fill distance values for each cell.

Direction heading = NORTH; // Current direction the robot thinks it is facing.
Phase phase = PHASE_EXPLORE; // Current stage of the program.

int currentX = START_X; // Current x-position of the robot in the maze.
int currentY = START_Y; // Current y-position of the robot in the maze.

unsigned char motorState = 0; // Stores the current 4-bit motor bus value.
unsigned char ledState = 0; // Stores the current 4-bit LED bus value.

const int DX[4] = {0, 1, 0, -1}; // x movement for NORTH, EAST, SOUTH, WEST.
const int DY[4] = {1, 0, -1, 0}; // y movement for NORTH, EAST, SOUTH, WEST.
const unsigned char WALL_MASK[4] = { // Converts a direction into the correct wall bit.
WALL_NORTH,
WALL_EAST,
WALL_SOUTH,
WALL_WEST
};



void stepBoth(int leftDirection, int rightDirection, int steps);

//LED helpers
void applyLEDState(void)
```

```
{
statusLEDs = ledState; // Sends the saved LED state to the physical LED pins.
}

void ledsOff(void)
{
ledState = 0; // Clears all LED bits.
applyLEDState(); // Updates the real LEDs.
}

void ledOn(unsigned char mask)
{
ledState = ledState | mask; // Turns on only the LED bits included in mask.
applyLEDState(); // Sends the new LED state to the board.
}

void ledOff(unsigned char mask)
{
ledState = ledState & (unsigned char)(~mask); // Turns off only the LED bits included in mask.
applyLEDState(); // Sends the new LED state to the board.
}

void ledPulse(unsigned char mask, int microseconds)
{
ledOn(mask); // Turn the selected LED on.
wait_us(microseconds); // Keep it on briefly so it can be seen.
ledOff(mask); // Turn it back off.
}

//sensor helpers

int sensorWall(unsigned int bitNumber)
{
int value; // Stores the full 7-bit sensor reading.

value = sensors.read(); // Read all sensors in one operation.

if ((value & (1 << bitNumber)) == 0) { // Active-low check: 0 means wall detected.
return 1; // Return 1 when a wall is present.
}

return 0; // Return 0 when the path is clear.
}

int frontWallSensor(void)
{
return sensorWall(SENSOR_F_BIT); // Check the front sensor.
}

int leftWallSensor(void)
{
return sensorWall(SENSOR_LC_BIT); // Use the centre-left sensor for left wall detection.
}
```

```
int rightWallSensor(void)
{
return sensorWall(SENSOR_RC_BIT); // Use the centre-right sensor for right wall detection.
}

unsigned char readWallPattern(void)
{
int raw; // Raw value from the BusIn sensors.
raw = sensors.read(); // Read all seven sensors at once.
return (unsigned char)((~raw) & 0x7F); // Invert active-low sensors so 1 = wall.
}

int leftGroupDetected(unsigned char wallsPattern)
{
return ((wallsPattern & SENSOR_LEFT_GROUP_MASK) != 0);
}

int frontGroupDetected(unsigned char wallsPattern)
{
return ((wallsPattern & SENSOR_FRONT_GROUP_MASK) != 0);
}

int rightGroupDetected(unsigned char wallsPattern)
{
return ((wallsPattern & SENSOR_RIGHT_GROUP_MASK) != 0);
}

SensorAction classifySensorPattern(unsigned char wallsPattern)
{
/* Same rules as your sensor test code, but in hex instead of 0b binary. */
if (wallsPattern == 0x22) {
return SENSOR_ACTION_FORWARD;
}

if (wallsPattern == 0x60 || wallsPattern == 0x40 || wallsPattern == 0x20 ||
wallsPattern == 0x68 || wallsPattern == 0x48 || wallsPattern == 0x28) {
return SENSOR_ACTION_TURN_RIGHT;
}

if (wallsPattern == 0x03 || wallsPattern == 0x01 || wallsPattern == 0x02) {
return SENSOR_ACTION_TURN_LEFT;
}

if (wallsPattern == 0x33 || wallsPattern == 0x11) {
return SENSOR_ACTION_CORRECT_RIGHT;
}

if (wallsPattern == 0x66 || wallsPattern == 0x44) {
return SENSOR_ACTION_CORRECT_LEFT;
}

if (wallsPattern == 0x00) {
return SENSOR_ACTION_CROSS_JUNCTION;
}
```

```
if (wallsPattern == 0x2A || wallsPattern == 0x6E || wallsPattern == 0x4C ||
wallsPattern == 0x3B || wallsPattern == 0x18) {
return SENSOR_ACTION_U_TURN;
}

return SENSOR_ACTION_UNKNOWN;
}

void showSensorActionOnLEDs(SensorAction action)
{
if (action == SENSOR_ACTION_TURN_RIGHT || action == SENSOR_ACTION_TURN_LEFT ||
action == SENSOR_ACTION_U_TURN) {
ledPulse(LED_WALL, 30000);
} else if (action == SENSOR_ACTION_UNKNOWN) {
ledPulse(LED_ERROR, 30000);
}
}

void applySensorCorrection(SensorAction action)
{
if (action == SENSOR_ACTION_CORRECT_RIGHT) {
stepBoth(DIR_LEFT_FWD, DIR_RIGHT_FWD, 20);
} else if (action == SENSOR_ACTION_CORRECT_LEFT) {
stepBoth(DIR_LEFT_FWD, DIR_RIGHT_FWD, 20);
}
}



// direction helpers

Direction turnLeft(Direction d)
{
return (Direction)(((int)d + 3) % 4); // Turning left is the same as subtracting 1 direction.
}

Direction turnRight(Direction d)
{
return (Direction)(((int)d + 1) % 4); // Turning right adds 1 direction.
}

Direction turnAround(Direction d)
{
return (Direction)(((int)d + 2) % 4); // Turning around adds 2 directions.
}

Direction cellToDirection(Cell from, Cell to)
{
int dx; // Difference in x between two cells.
int dy; // Difference in y between two cells.

dx = to.x - from.x; // Positive dx means the target cell is to the east.
dy = to.y - from.y; // Positive dy means the target cell is to the north.
```

```
if (dy == 1) {
return NORTH; // Target cell is above the current cell.
}

if (dx == 1) {
return EAST; // Target cell is to the right.
}

if (dy == -1) {
return SOUTH; // Target cell is below the current cell.
}

if (dx == -1) {
return WEST; // Target cell is to the left.
}

return NORTH; // Safe default if the cells are not adjacent.
}

//motor helpers

void applyMotorState(void)
{
motorBus = motorState; // Sends the saved motor state to p21-p24.
}

void motorStop(void)
{
motorState = 0; // Clear direction and clock bits.
applyMotorState(); // Apply the stopped state to the L297 inputs.
}

void setMotorDirections(int leftDirection, int rightDirection)
{
motorState = motorState & (unsigned char)(~MOTOR_DIR_MASK); // Clear the old direction bits first.

if (leftDirection) {
motorState = motorState | MOTOR_LEFT_DIR_MASK; // Set left direction bit if needed.
}

if (rightDirection) {
motorState = motorState | MOTOR_RIGHT_DIR_MASK; // Set right direction bit if needed.
}

applyMotorState(); // Send direction bits to the L297 chips.
}

void pulseMotorClocks(void)
{
motorState = motorState | MOTOR_CLK_MASK; // Set both clock pins HIGH.
applyMotorState(); // Output the HIGH clock level.
wait_us(STEP_DELAY_US); // Hold the clock HIGH briefly.
```

```
motorState = motorState & (unsigned char)(~MOTOR_CLK_MASK); // Set both clock pins LOW.
applyMotorState(); // Output the LOW clock level.
wait_us(STEP_DELAY_US); // Hold the clock LOW briefly.
}

void stepBoth(int leftDirection, int rightDirection, int steps)
{
int i; // Loop counter for the number of step pulses.

setMotorDirections(leftDirection, rightDirection); // Set motor directions before stepping.

for (i = 0; i < steps; i++) { // Repeat for the required number of steps.
pulseMotorClocks(); // One pulse moves both motors by one step.
}
}

void driveForward(void)
{
ledOn(LED_MOVE); // LED1 shows movement is happening.
stepBoth(DIR_LEFT_FWD, DIR_RIGHT_FWD, STEPS_PER_CELL); // Move forward by one cell.
ledOff(LED_MOVE); // Turn movement LED off once movement ends.
}

void rotateLeft90(void)
{
ledOn(LED_MOVE); // Show that a turn is happening.
stepBoth(DIR_LEFT_BWD, DIR_RIGHT_FWD, STEPS_PER_90); // Spin wheels opposite ways to turn left.
ledOff(LED_MOVE); // Turn off movement LED.
heading = turnLeft(heading); // Update the software direction after the turn.
}

void rotateRight90(void)
{
ledOn(LED_MOVE); // Show that a turn is happening.
stepBoth(DIR_LEFT_FWD, DIR_RIGHT_BWD, STEPS_PER_90); // Spin wheels opposite ways to turn right.
ledOff(LED_MOVE); // Turn off movement LED.
heading = turnRight(heading); // Update the software direction after the turn.
}

void rotate180(void)
{
ledOn(LED_MOVE); // Show that a turn is happening.
stepBoth(DIR_LEFT_FWD, DIR_RIGHT_BWD, STEPS_PER_180); // Use a longer turn for 180 degrees.
ledOff(LED_MOVE); // Turn off movement LED.
heading = turnAround(heading); // Update heading to the opposite direction.
}

void rotateToDirection(Direction target)
{
int diff; // Difference between current heading and target heading.

diff = ((int)target - (int)heading + 4) % 4; // Gives 0, 1, 2 or 3 depending on needed rotation.

if (diff == 1) {
rotateRight90(); // Target is one direction to the right.
```

```
} else if (diff == 3) {
rotateLeft90(); // Target is one direction to the left.
} else if (diff == 2) {
rotate180(); // Target is behind the robot.
}
}

//maze helper code

void initialiseMazeArrays(void)
{
int x; // Column index.
int y; // Row index.

for (y = 0; y < MAZE_H; y++) { // Go through every maze row.
for (x = 0; x < MAZE_W; x++) { // Go through every maze column.
walls[y][x] = 0; // Start with no known walls.
dist[y][x] = DIST_UNREACHABLE; // Start with no known flood-fill distance.
}
}
}

void setWallBidirectional(int x, int y, Direction dir)
{
int d; // Direction as an integer index.
int nx; // Neighbour cell x-coordinate.
int ny; // Neighbour cell y-coordinate.
Direction opposite; // Opposite direction for the neighbouring cell.

if (x < 0 || x >= MAZE_W || y < 0 || y >= MAZE_H) {
return; // Ignore invalid cells outside the maze.
}

d = (int)dir; // Convert direction enum to an array index.
walls[y][x] = walls[y][x] | WALL_MASK[d]; // Add the wall to the current cell.

nx = x + DX[d]; // Work out the neighbour cell across that wall.
ny = y + DY[d]; // Work out the neighbour cell across that wall.

if (nx < 0 || nx >= MAZE_W || ny < 0 || ny >= MAZE_H) {
return; // If there is no neighbour, stop here.
}

opposite = turnAround(dir); // The neighbour sees the same wall from the opposite side.
walls[ny][nx] = walls[ny][nx] | WALL_MASK[(int)opposite]; // Store the wall in the neighbour too.
}

int canMove(int x, int y, Direction dir)
{
int d; // Direction converted to integer.
int nx; // Next x-coordinate if moving in this direction.
int ny; // Next y-coordinate if moving in this direction.

if (x < 0 || x >= MAZE_W || y < 0 || y >= MAZE_H) {
```

```
return 0; // Cannot move from outside the maze.
}

d = (int)dir; // Convert direction into array index.

if (walls[y][x] & WALL_MASK[d]) {
return 0; // A known wall blocks movement in this direction.
}

nx = x + DX[d]; // Calculate target x-coordinate.
ny = y + DY[d]; // Calculate target y-coordinate.

if (nx < 0 || nx >= MAZE_W || ny < 0 || ny >= MAZE_H) {
return 0; // Cannot move outside the maze boundary.
}

return 1; // Movement is allowed.
}

//LEES algorithm

void resetDistances(void)
{
int x; // Column index.
int y; // Row index.

for (y = 0; y < MAZE_H; y++) { // Loop through each row.
for (x = 0; x < MAZE_W; x++) { // Loop through each column.
dist[y][x] = DIST_UNREACHABLE; // Reset all distances before a new flood fill.
}
}
}

void runFloodFromQueue(Cell queue[], int back)
{
int front; // Index of the next queue item to process.

front = 0; // Start from the first item in the queue.

while (front < back) { // Keep processing until all queued cells are handled.
Cell c; // Current cell being expanded.
int d; // Direction loop counter.

c = queue[front]; // Take the next cell from the queue.
front++; // Move queue front to the next item.

for (d = 0; d < 4; d++) { // Try north, east, south and west.
int nx; // Neighbour x-coordinate.
int ny; // Neighbour y-coordinate.

if (!canMove(c.x, c.y, (Direction)d)) {
continue; // Skip this direction if blocked or outside maze.
}
```

```
nx = c.x + DX[d]; // Calculate neighbour x-position.
ny = c.y + DY[d]; // Calculate neighbour y-position.

if (dist[ny][nx] == DIST_UNREACHABLE) { // Only visit cells that have not been reached yet.
dist[ny][nx] = dist[c.y][c.x] + 1; // Distance is one more than the current cell.

if (back < MAX_PATH) { // Make sure the queue array does not overflow.
queue[back].x = nx; // Add neighbour x to the queue.
queue[back].y = ny; // Add neighbour y to the queue.
back++; // Increase queue size.
}
}
}
}
}

void floodFillGoal(void)
{
Cell queue[MAX_PATH]; // Queue used for flood fill.
int back; // Number of cells currently in the queue.
int gx; // Goal x-coordinate loop variable.
int gy; // Goal y-coordinate loop variable.

back = 0; // Queue starts empty.
resetDistances(); // Clear previous distance values.

for (gx = 3; gx <= 4; gx++) { // The centre goal area has two x positions.
for (gy = 3; gy <= 4; gy++) { // The centre goal area has two y positions.
dist[gy][gx] = 0; // Goal cells have distance zero.
queue[back].x = gx; // Add goal cell x to the queue.
queue[back].y = gy; // Add goal cell y to the queue.
back++; // Increase queue size.
}
}

runFloodFromQueue(queue, back); // Spread distance values out from the goal.
}

void floodFillTo(int targetX, int targetY)
{
Cell queue[MAX_PATH]; // Queue used for flood fill.
int back; // Number of cells currently in the queue.

back = 0; // Queue starts empty.
resetDistances(); // Clear previous distance values.

if (targetX < 0 || targetX >= MAZE_W || targetY < 0 || targetY >= MAZE_H) {
return; // Ignore invalid target cells.
}

dist[targetY][targetX] = 0; // Target cell has distance zero.
queue[back].x = targetX; // Add target x to the queue.
queue[back].y = targetY; // Add target y to the queue.
```

```
back++; // Increase queue size.

runFloodFromQueue(queue, back); // Spread distance values out from the target.
}

int tracePath(Cell start, Cell path[])
{
Cell current; // Cell currently being traced from.
int pathLength; // Number of cells currently in the path.
int safety; // Prevents an infinite loop if something goes wrong.

pathLength = 0; // Start with an empty path.
safety = 0; // Safety counter starts at zero.

if (start.x < 0 || start.x >= MAZE_W || start.y < 0 || start.y >= MAZE_H) {
return 0; // Invalid start cell means no path.
}

if (dist[start.y][start.x] == DIST_UNREACHABLE) {
return 0; // If flood fill cannot reach the start, there is no path.
}

current = start; // Begin tracing from the start cell.
path[pathLength] = current; // Store the first cell in the path.
pathLength++; // Increase path length.

while (dist[current.y][current.x] != 0 && safety < MAX_PATH) { // Stop when target distance zero is reached.
int d; // Direction loop counter.
int bestDist; // Best neighbouring distance found so far.
Cell next; // Best next cell to move to.
int found; // Shows whether a lower distance neighbour was found.

bestDist = dist[current.y][current.x]; // Start with the current cell distance.
next = current; // Default next cell is the current cell.
found = 0; // No better cell found yet.

for (d = 0; d < 4; d++) { // Check all four directions.
int nx; // Neighbour x-coordinate.
int ny; // Neighbour y-coordinate.

if (!canMove(current.x, current.y, (Direction)d)) {
continue; // Skip blocked directions.
}

nx = current.x + DX[d]; // Calculate neighbour x-position.
ny = current.y + DY[d]; // Calculate neighbour y-position.

if (dist[ny][nx] < bestDist) { // Choose the neighbour with the smallest distance.
bestDist = dist[ny][nx]; // Save the new best distance.
next.x = nx; // Save the new best x-position.
next.y = ny; // Save the new best y-position.
found = 1; // Mark that a valid next step was found.
}
}
```

```
if (!found) {
break; // Stop if no progress can be made.
}

current = next; // Move to the next cell in the traced path.

if (pathLength < MAX_PATH) { // Make sure path array does not overflow.
path[pathLength] = current; // Store the next path cell.
pathLength++; // Increase path length.
}

safety++; // Increase safety counter.
}

return pathLength; // Return how many cells are in the path.
}

//control logic code

int atGoal(void)
{
if (currentX == GOAL_X && currentY == GOAL_Y) { // Check if robot is at the chosen goal cell.
return 1;
}

return 0;
}

int atStart(void)
{
if (currentX == START_X && currentY == START_Y) { // Check if robot is back at the start.
return 1;
}

return 0;
}

void updateWallMap(void)
{
unsigned char previousWalls; // Stores wall value before checking sensors.
unsigned char wallPattern; // Seven-sensor pattern where 1 means wall detected.
SensorAction action; // Interpreted sensor action from the pattern.

previousWalls = walls[currentY][currentX]; // Save old wall information for comparison.
wallPattern = readWallPattern(); // Read all sensors and convert to active-high walls.
action = classifySensorPattern(wallPattern); // Classify the pattern using your sensor rules.

showSensorActionOnLEDs(action); // Use onboard LEDs instead of serial/iostream output.

if (frontGroupDetected(wallPattern)) {
setWallBidirectional(currentX, currentY, heading); // Add wall in the direction the robot faces.
}
```

```
if (leftGroupDetected(wallPattern)) {
setWallBidirectional(currentX, currentY, turnLeft(heading)); // Add wall to the robot's left.
}

if (rightGroupDetected(wallPattern)) {
setWallBidirectional(currentX, currentY, turnRight(heading)); // Add wall to the robot's right.
}

if (walls[currentY][currentX] != previousWalls) {
ledPulse(LED_WALL, 50000); // Pulse wall LED if new wall data was discovered.
}
}

int moveOneStep(void)
{
Cell start; // Current cell before moving.
Cell next; // Next cell selected from the path.
Cell path[MAX_PATH]; // Stores the path produced from flood-fill distances.
int pathLength; // Number of cells in the path.
Direction targetDirection; // Direction needed to face the next cell.

start.x = currentX; // Copy current x-position into a Cell.
start.y = currentY; // Copy current y-position into a Cell.

pathLength = tracePath(start, path); // Work out the best path from the current cell.

if (pathLength < 2) { // Path needs at least current cell and next cell.
ledOn(LED_ERROR); // Turn on error LED if no next move exists.
motorStop(); // Stop motors for safety.
return 0; // Return failure.
}

applySensorCorrection(classifySensorPattern(readWallPattern())); // Apply light centring correction if needed.

next = path[1]; // The second cell is the cell we should move into next.
targetDirection = cellToDirection(start, next); // Convert the next cell into a direction.

rotateToDirection(targetDirection); // Turn the robot to face the chosen next cell.

if (frontWallSensor()) { // Check once more before moving forward.
motorStop(); // Stop if a wall is directly ahead.
ledPulse(LED_WALL, 200000); // Pulse wall LED for a clearer warning.
return 0; // Return failure because movement was blocked.
}

driveForward(); // Move forward by one cell.

currentX = next.x; // Update software x-position after moving.
currentY = next.y; // Update software y-position after moving.

return 1; // Movement succeeded.
}
```

```
int executePath(Cell path[], int pathLength)
{
int i; // Loop counter for each movement in the path.

if (pathLength < 2) { // A path must have at least two cells to move.
ledOn(LED_ERROR); // Turn on error LED.
return 0; // Return failure.
}

for (i = 0; i < pathLength - 1; i++) { // Move from each cell to the next cell.
Direction targetDirection; // Direction needed for this step.

targetDirection = cellToDirection(path[i], path[i + 1]); // Convert cell step into direction.
rotateToDirection(targetDirection); // Rotate until robot faces the next cell.

if (frontWallSensor()) { // Check for unexpected wall before moving.
motorStop(); // Stop motors immediately.
ledOn(LED_ERROR); // Show error LED.
return 0; // Return failure.
}

driveForward(); // Move one cell forward.
}

return 1; // Full path completed successfully.
}

//main code

int main(void)
{
Cell speedStart; // Start cell used for the final speed run.
Cell speedPath[MAX_PATH]; // Stores the final planned path.
int speedPathLength; // Number of cells in the final planned path.
int ok; // Stores whether speed run succeeded.

sensors.mode(PullUp); // Enable pull-ups because the IR sensors are active-low.
ledsOff(); // Start with all LEDs off.
motorStop(); // Ensure motors are not moving at startup.
initialiseMazeArrays(); // Clear wall map and distance map.
floodFillGoal(); // Create initial distance map towards the goal.

while (phase != PHASE_COMPLETE) { // Main loop runs until the robot is finished.
switch (phase) { // Choose behaviour based on current phase.
case PHASE_EXPLORE: // First stage: move towards the goal.
updateWallMap(); // Read sensors and update known wall data.
floodFillGoal(); // Recalculate route to the centre goal.

if (atGoal()) { // Check whether the robot has reached the goal.
ledOn(LED_GOAL); // Turn on goal LED.
phase = PHASE_RETURN; // Next phase is returning to start.
break; // Leave this switch case.
}
```

```
moveOneStep(); // Move one cell towards the goal.
break; // End explore phase step.

case PHASE_RETURN: // Second stage: return to the start.
updateWallMap(); // Keep updating wall map while returning.
floodFillTo(START_X, START_Y); // Recalculate distances towards the start cell.

if (atStart()) { // Check whether the robot is back at the start.
ledOff(LED_GOAL); // Turn off goal LED before speed run.
phase = PHASE_SPEEDRUN; // Next phase is the speed run.
break; // Leave this switch case.
}

moveOneStep(); // Move one cell towards the start.
break; // End return phase step.

case PHASE_SPEEDRUN: // Third stage: use the known map for final run.
speedStart.x = START_X; // Set final run start x-position.
speedStart.y = START_Y; // Set final run start y-position.

currentX = START_X; // Reset software x-position to start.
currentY = START_Y; // Reset software y-position to start.
heading = NORTH; // Reset heading assumption to north.

floodFillGoal(); // Recalculate route to the goal using known walls.
speedPathLength = tracePath(speedStart, speedPath); // Produce final path.

ok = executePath(speedPath, speedPathLength); // Drive along final path.

if (ok) {
ledOn(LED_GOAL); // Show success if the speed run completed.
} else {
ledOn(LED_ERROR); // Show error if speed run failed.
}

phase = PHASE_COMPLETE; // Stop after the speed run attempt.
break; // End speed run case.

case PHASE_COMPLETE: // Final state.
motorStop(); // Make sure motors are stopped.
break; // Nothing else to do.
}

wait_us(10000); // Small delay between control-loop cycles.
}

motorStop(); // Safety stop after leaving the loop.

while (1) { // Keep the program alive forever.
wait_us(1000000); // Wait one second repeatedly.
}
}
```

[redacted]