TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY (CIT)

TECHNICAL UNIVERSITY OF MUNICH

Report Submitted to Masterpraktikum - Micromouse: Designing an Educational Racing-Robot from Scratch

Final Report

Ahmed Ghaleb, Théo Karst, Anna Schmitt, J.Arturo Sol Navarro

TECHNICAL UNIVERSITY OF MUNICH

Report Submitted to Masterpraktikum - Micromouse: Designing an Educational Racing-Robot from Scratch

Final Report

Author: Ahmed Ghaleb, Théo Karst, Anna Schmitt, J.Arturo Sol Navarro

Submission Date: October 17, 2024

We confirm that this report is our own work and we have documented all sources and material used.

Munich, October 17, 2024
Ahmed Ghaleb, Théo Karst, Anna Schmitt, J.Arturo Sol Navarro

Contents

1. Introduction

Micromouse: Designing an Educational Racing-Robot from Scratch, is a master's level practical course. The goal of this course for the students is to make a racing-robot called "mouse" which would find a path to the center of a 6x6 maze. Once found, the robot would then take the path and should reach the center in the shortest amount of time. This project is part of the broader Micromouse competition, an event that has been held since 1977, challenging participants to create intelligent robotic systems capable of efficient maze-solving.

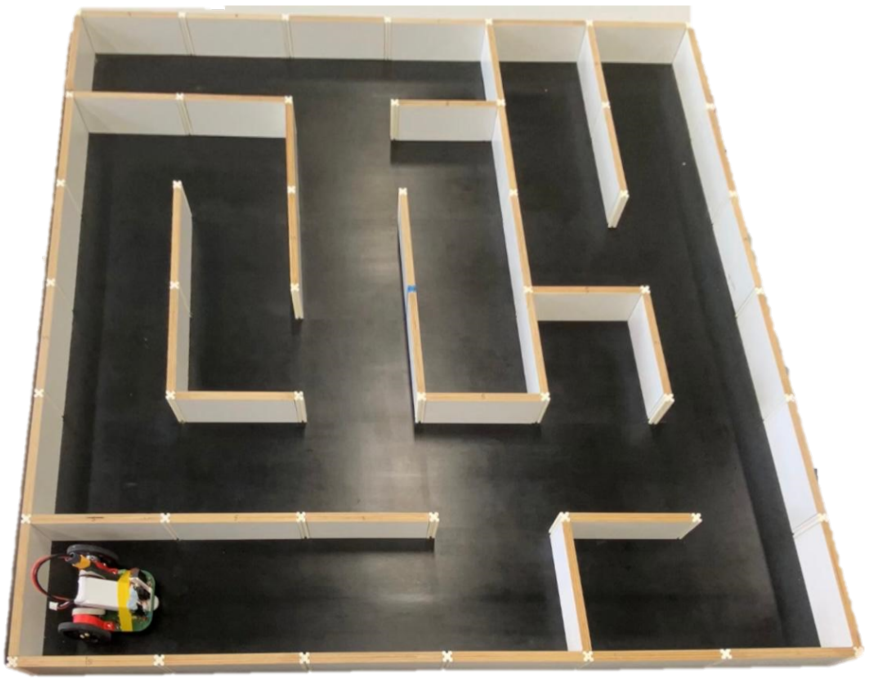


Figure 1.1.: 6x6 Maze Layout

Throughout this course students had to learn topics from different fields such as hardware design, embedded & real-time programming and control systems. Each topic was introduced with a lecture and programming exercises (in C language) to better understand concepts introduced earlier on. From there, students in groups of 4 would then combine these building blocks to make a robot capable of reliably navigating through a maze, solving it and avoiding collisions. Requirements for succeeding the course are adapted to its budget and time constraints. At the end of the course, students deliver the robot and a written report of their work.

This report details the entire process of designing and implementing the hardware and software components of the robot, from initial conceptual design to final testing and debbuging.

1.1. Conceptual design and justification of the design

To simplify the making of the robot, a few requirements were set at the start of the course:

- Two 6V DC gear motors - 2619 SR IE-16
- One microcrontroller (ref: `dsPIC33FJ128MC804`)
- A maze - We had access to a 6x6 Maze

The motors when attached to wheels, placed at the back left and back right of the robot, allow for **movement** of the device. The micro-controller contains the logic to communicate and control the components peripherals such as sensors, motors, LEDs etc.. The remaining components were chosen by the group and some with the consultation of the professor.

For **perceiving** its surroundings, the robot must be equipped with sensors. For this project, the maze is indoors which allows us to either ultra-sound or infrared sensors. Our first distance sensor were the Time of Flight VL53L0X, however setting them up was harder and more time consuming than we had imagined on a bare-metal robot. We then went for Sharp IR sensors (ref: `GP2Y0A21YK0F`), as they were much simpler to set up. To not break away from tradition three sensors were used for distances to the walls on the right, left and front of the robot.
To keep everything in place on the Printed Circuit Board (PCB), we have soldered or mounted the components on it. Mounts for the wheels were already provided to us, others such as sensor mounts were designed and printed by us.

For functional purposes we added two buttons, one for interacting with our program and another one for resetting the micro-controller. Lastly, for debugging purposes, we have added two LEDs which can be programmed to indicate the state of our program.

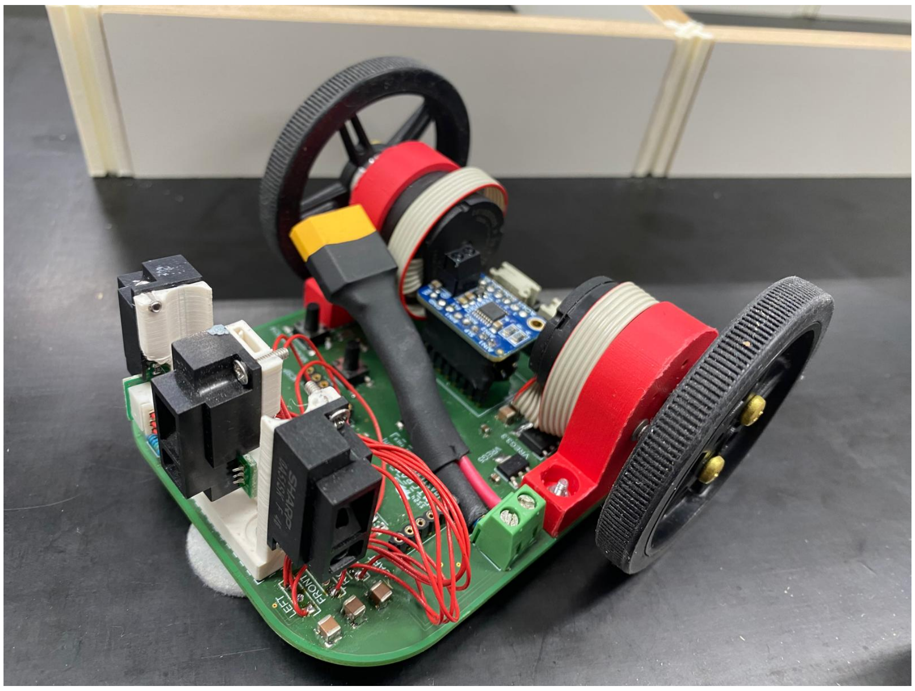


Figure 1.2.: Isometric Visualization of the final Micromouse Robot

Figure 1.3.: Watch the Micromouse in Action—Scan Here!

2. Printed Circuit Board Design

2.1. DsPIC33FJ64MC804 Basic Connections Module

According to the microcontroller's datasheet, the following minimal pin connections are required before proceeding with development. Decoupling capacitors of 0.1 µF, rated for 10-20V, with low ESR and a resonance frequency of 20 MHz or higher, should be placed on every pair of power supply pins. These capacitors must be positioned as close to the pins as possible, ideally within 6 mm, to minimize PCB track inductance.

A low-ESR capacitor, ranging from 4.7 to 10 µF (ceramic or tantalum) with an ESR less than 5 Ohms, is required on the VCAP pin to stabilize the voltage regulator's output. The trace length for this capacitor should also not exceed 6 mm.

Correct setup of the MCLR pin is crucial for device resetting, programming, and debugging. Components related to this pin should be placed within 6 mm. During programming and debugging operations, it is recommended to isolate the capacitor from the MCLR pin.

Reset capacility involves connecting the button to the MCLR (Master Clear) pin. When the button is pressed, it should pull the MCLR pin low, which will reset the microcontroller. The TS04-66-95-BK-160-SMT-TR 6x6 sensor switch was selected.

The final design of this module is shown in Figure 1.1.

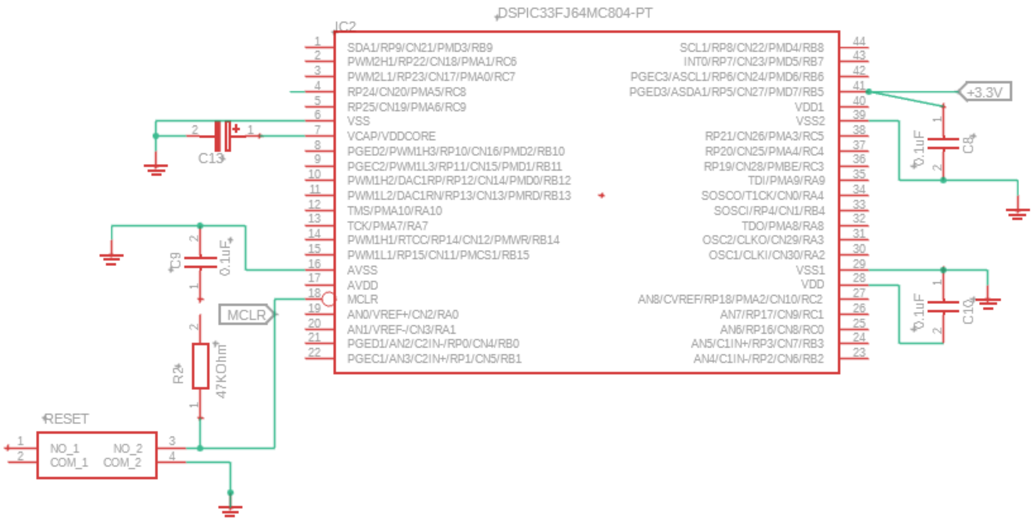


Figure 2.1.: dsPIC33FJ64MC804 Basic connections module schematic

2.2. Voltage Regulation Module

To ensure a stable and constant DC supply for powering the microcontroller and other circuit components, a linear voltage regulator is a suitable choice.

A rechargeable 9 V battery will be connected to the PCB using a 2-position fixed terminal block.

The first stage of voltage regulation uses the LM7805MP/NOPB voltage regulator, which drops the battery voltage from 9 V to a range between 4.8 V and 5.2 V. A minimum input voltage of 7.5 V is required to maintain this range. A 0.22 µF capacitor is placed between the input and ground, and a 0.1 µF capacitor is placed between the output and ground to improve transient response.

The second stage uses the LM1117DT-3.3/NOPB low-dropout regulator, which drops the voltage from 5 V to a range between 3.267V and 3.33 V. A 10 µF capacitor is placed between the input and ground, and another 10 µF capacitor is placed between the output and ground to improve transient response. The term "low-dropout" refers to the regulator's ability to operate with a small difference between the input and output voltages.

The final version of the voltage regulation module can be seen in Figure 1.2.

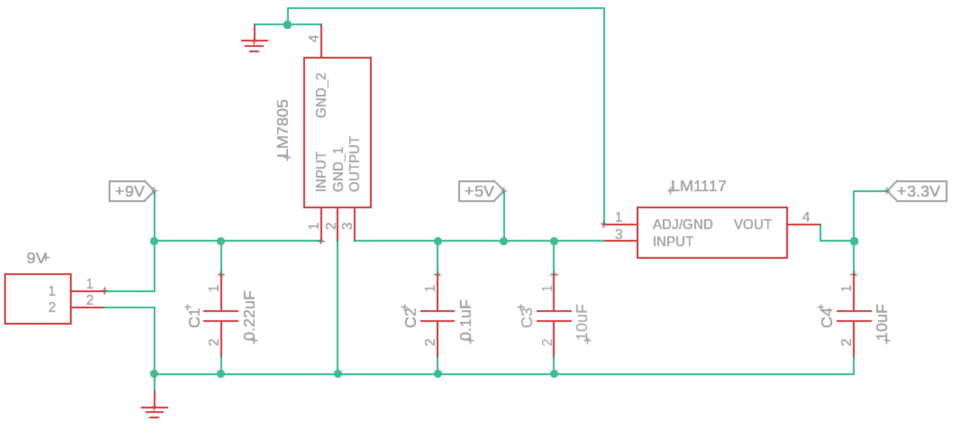


Figure 2.2.: Voltage regulation module schematic

The voltage regulator LD1117-3.3 will take the 5 V output and drop it down to a range between 3.267 and 3.333 V requiered by the microcontroller. According to the specifications a 100-nF capacitor from input to ground and a 10-µF capacitor from output to ground are required to ensure stable operation.

2.3. Motor Control Module

Two 6V DC gear motors from the series 2619 SR IE2-16, each equipped with integrated encoders featuring a 33:1 reduction ratio, have been selected. Each motor includes a 6-pin DIN-41651 connector.0.1 µF decoupling capacitors were used for the encoder's voltage supply. The functionality of each pin is detailed in the following table.

Pin Number	Description
1	Motor -
2	Motor +
3	Encoder GND
4	Encoder Voltage Supply
5	Encoder Channel B
6	Encoder Channel A

Table 2.1.: DC gear motor pin description

Pins 5 and 6 of the DC gearmotor connector correspond to Channels B and A of the quadrature encoder, respectively. These channels produce square waves that are 90 degrees out of phase with each other. By analyzing the order and timing of these signals, you can determine the motor's rotation direction and count its steps. If Channel A leads Channel B, the motor rotates in one direction; if Channel B leads Channel A, it rotates in the opposite direction. Counting the transitions (rising and falling edges) of these signals provides the motor's shaft position, while the transition rate indicates the rotation speed.

The direction and speed control of the DC gear motors will be managed using an Adafruit DRV8833 board. This board can drive two DC motors or a single bipolar or unipolar stepper motor with up to 1.2 A per channel. The key component of the board is the DRV8833 chip, which integrates two full H-bridges.The functionality of each used pin is detailed in the following table.

Pin Name	Description	Pin Name	Description
AOUT1	Motor A output	AIN1	H-Bridge A input (PWM)
AOUT2	Motor A output	AIN2	H-Bridge A input (PWM)
BOUT1	Motor B output	BIN1	H-Bridge B input (PWM)
BOUT2	Motor B output	BIN2	H-Bridge B input (PWM)
SLP	To logic high to enable	Vmotor	Motor supply voltage
GND	Shared ground		

Table 2.2.: DC Gear Motor Pin Description

The final version of this module module can be seen in Figure 1.3.

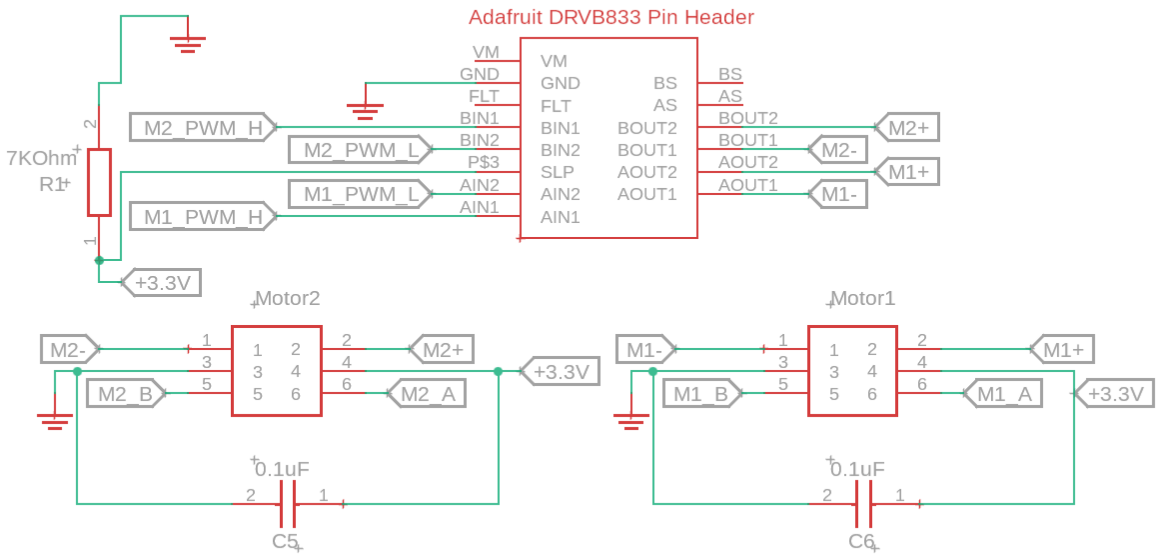


Figure 2.3.: Motor control module schematic

2.4. External Oscillator Module

An external oscillator is essential for dsPIC devices because it provides higher accuracy and stability compared to internal oscillators, which is crucial for applications requiring precise timing. External oscillators offer better stability over temperature and voltage variations, ensuring reliable performance. They can operate at higher frequencies, which is necessary for demanding processing tasks. The following 7.3728 MHz crystal oscillator was selected ABLS-7.3728MHZ-K4T.

The final version of this module module can be seen in Figure 1.4.

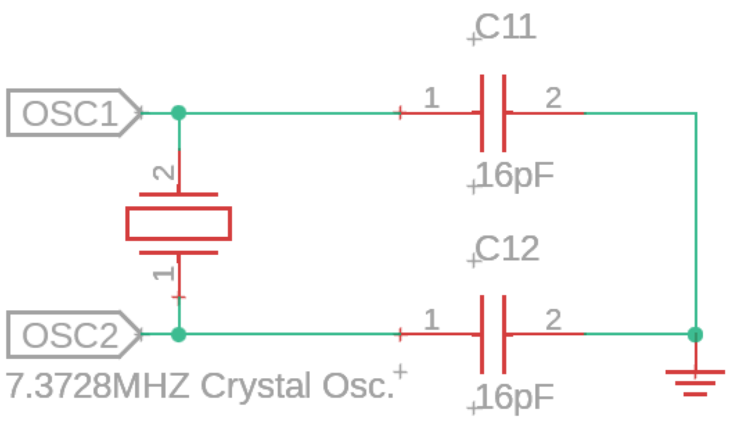


Figure 2.4.: External oscillator module schematic

2.5. In-Circuit Serial Programming Module

In-Circuit Serial Programming (ICSP) is a method for programming microcontrollers while they are still mounted on the circuit board. It uses a dedicated interface to upload firmware or data without removing the chip, facilitating easier updates and debugging.It is recommended to keep the trace length between the ICSP connector and the ICSP pins on the device as short as possible. A 45 degree 6 pin header will be used to establish connection with the PICkit™ 4 In-Circuit Debugger.The final version of this module module can be seen in Figure 1.5.

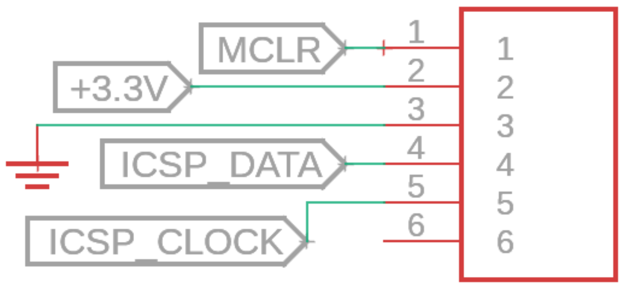


Figure 2.5.: ICSP module schematic

2.6. Universal Asynchronous Receiver-Transmitter Module

UART (Universal Asynchronous Receiver/Transmitter) is a hardware communication protocol used for serial communication between devices. It transmits data one bit at a time over a single wire, using two wires for communication: one for sending and one for receiving . UART is commonly used for communication between microcontrollers and peripherals, like sensors and computers.

The CP2102 USB UART Board (type A) is an accessory board that features the single-chip USB to UART bridge CP2102 onboard.The functionality of each pin is detailed in the following table.Pins 5 and 6 are Request to Send (RTS) and Clear To Send (CTS), respectively. RTS and CTS wll be ignored because the application doe snot requiere hardware flow control.The final version of this module module can be seen in Figure 1.6.

Pin Number	Description
1	3.3V
2	GND
3	TXD Serial Output
4	RXD Serial Input
5	Request to Send
6	Clear to Send

Table 2.3.: DC gear motor pin description

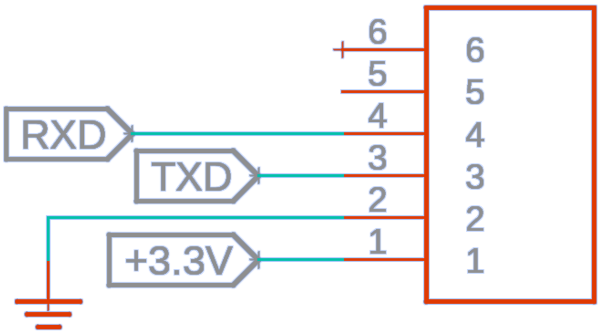


Figure 2.6.: UART module schematic

2.7. Distance Sensors Module

To perceive the distances between the mouse and it's surroundings, the distance measuring sensor unit GP2Y0A51SK0F was selected. This infrared sensor offers a measuring range between 2 and 15cm, thus fulfilling the maze width constraints. Its dimensions are approximately 27 mm x 10.8 mm x 12 mm, making it a small and lightweight option for various projects. This small size allows for easy integration into space-constrained applications.

This sensor provides an analog voltage output that corresponds to the distance measured. The operating supply voltage range is 4.5 - 5.5 V.In order to stabilize power supply line according to the data sheet, decoupling capacitors of 10 uF were included.The final version of this module module can be seen in Figure 1.7.

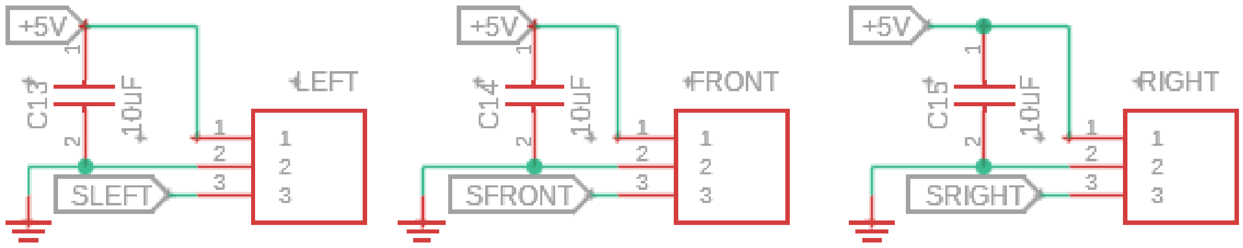


Figure 2.7.: Distance Sensors module schematic

2.8. Complementary Components Module

A TS04-66-95-BK-160-SMT-TR 6x6 sensor switch was selected to be able to switch between Exploration Mode and Speed Run Mode .

Two standard LEDs will be used in order to highlight different features in a visual way. Each LEDs will be controlled by a 2N2222 NPN transistor in order to ensure the demanded current supply.

The final version of this module module can be seen in Figure 1.8.

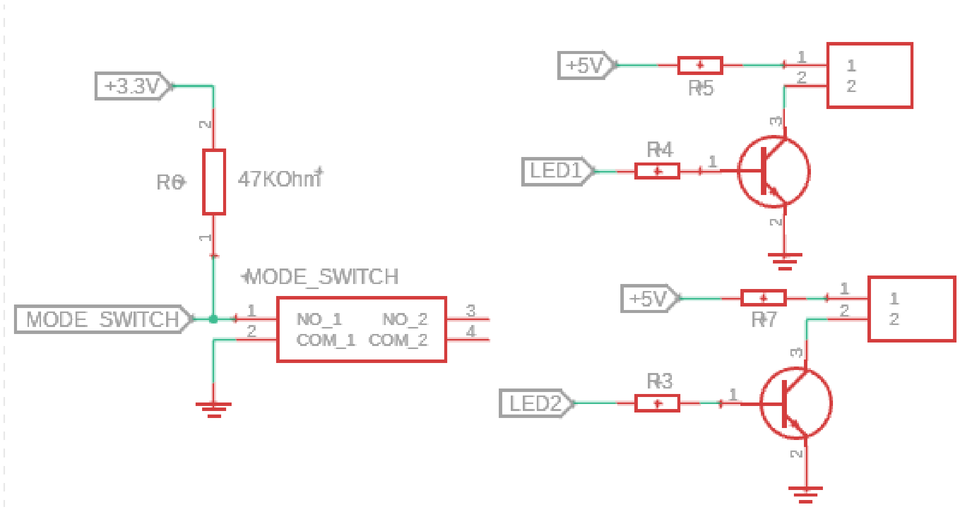


Figure 2.8.: Complementary components module schematic

2.9. Final Printed Circuit Board

The dimensions of the final version of the PCB are 8 cm x 10 cm, resulting in a total area of 0,8024 dm2. Ground top and bottom planes were used because they reduce electrical noise and improve signal integrity. Power routes were made wider to provide more area for current flow. The 5V and 3.3V routes are 0.5 mm wide, and the 9V route is 1 mm wide. In order to secure the brackets for the motors and sensors, 2 mm holes were planned.In addition, smaller holes in the middle front section were placed to solder a standard LED to ensure the stability of the robot while driving.

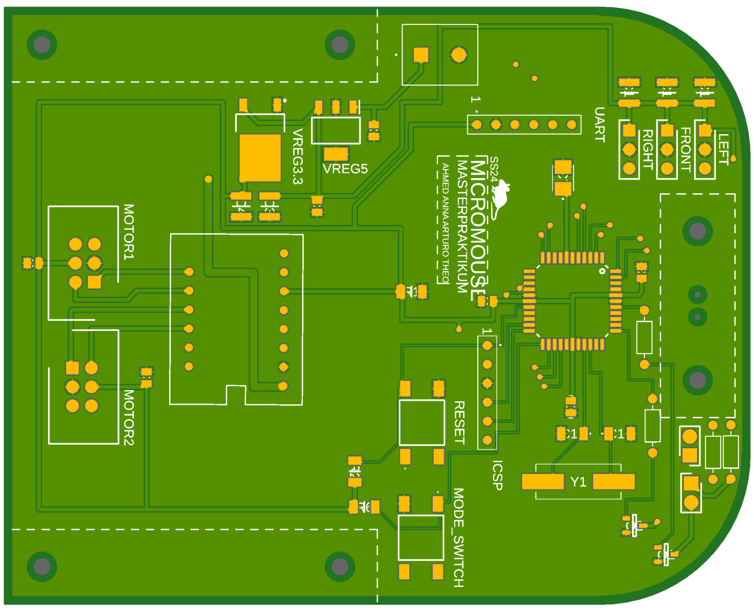


Figure 2.9.: Top layer of PCB

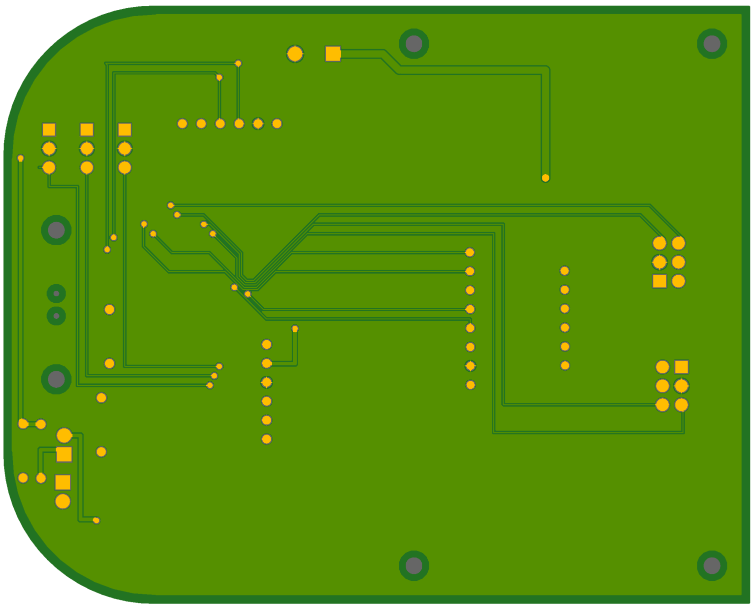


Figure 2.10.: Bottom layer of PCB

3. Software Design

3.1. Theoretical approach

The strategy for scanning the maze is a depth-first search. Starting from the initial position, the mouse travels each path it finds until it hits a dead end. Then it returns to the last crossing of paths and chooses the next path to go down until its end. This continues until each possible path has been traversed and the mouse has returned to its initial position.

We started with a Python implementation to completely figure out the logic in a quickly programmable language. This way all logical errors could be figured with the help of Python debugging tools. The code also contains a visualization in which the mouse movements can be tracked. Based on this implementation, we translated the code to C and verified the results via the Python visualization.

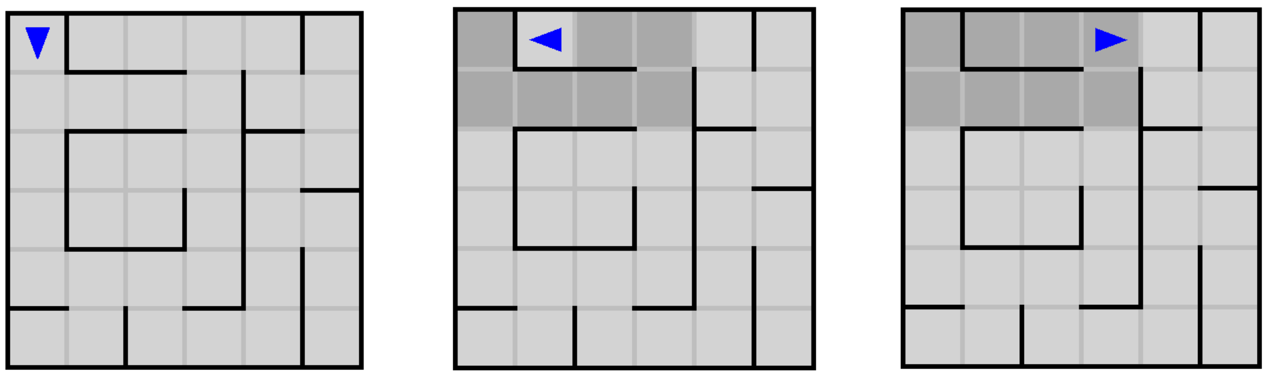
3.2. General movement

The maze is structured into a square of 6x6 tiles. The upper left corner has the coordinates [x, y] = [0, 0] where the x-coordinate specifies the column and the y-coordinate specifies the row (Figure 3.1). In the following, an example maze is used to visualize and explain the mouse behaviour.

[0, 0]	[1, 0]	[2, 0]	[3, 0]	[4, 0]	[5, 0]
[0, 1]	[1, 1]	[2, 1]	[3, 1]	[4, 1]	[5, 1]
[0, 2]	[1, 2]	[2, 2]	[3, 2]	[4, 2]	[5, 2]
[0, 3]	[1, 3]	[2, 3]	[3, 3]	[4, 3]	[5, 3]
[0, 4]	[1, 4]	[2, 4]	[3, 4]	[4, 4]	[5, 4]
[0, 5]	[1, 5]	[2, 5]	[3, 5]	[4, 5]	[5, 5]

Figure 3.1.: Coordinates of the grid tiles

The mouse starts its search from a predefined tile and continuously records the steps that were taken to reach its current position. When it takes a step forward onto an unexplored tile, this



(a) Starting position (b) Forward movement (c) Backtracking

Figure 3.2.: General movements

step is added to the list and the tile is marked as explored. In the visualization, the explored tiles are displayed in a dark grey. During backtracking the steps are removed.

In Figure 3.2a the mouse is positioned at the starting position on tile [0, 0]. When the mouse reaches the position given in Figure 3.2b the current list of steps is

$$[0, 0], [0, 1], [1, 1], [2, 1], [3, 1], [3, 0], [2, 0], [1, 0] \tag{3.1}$$

During backtracking the mouse traces back its already visited steps as shown in Figure 3.2c. The revisited tiles are removed from the step list which results in the list

$$[0, 0], [0, 1], [1, 1], [2, 1], [3, 1], [3, 0] \tag{3.2}$$

3.3. Choosing the direction

Each step the mouse takes starts with the decision, in which direction it continues moving. The directions are given as North, South, East and West. Via the sensors, the directions in which there are walls are figured out. If there is only one direction without a wall, the mouse moves one tile in this direction. If there is more than one possibility as shown in Figure 3.3a, the current tile is remembered as well as each possible direction. For Figure 3.3a this would be saved as

$$[3, 0], [WEST, EAST] \tag{3.3}$$

The mouse then takes the first direction from this list, takes the corresponding step and removes the direction from the list. Now the list contains the following:

$$[3, 0], [EAST] \tag{3.4}$$

When the mouse returns to that intersection later (see Figure 3.3b) the next direction is taken from the list without checking the possible directions again via the sensors.

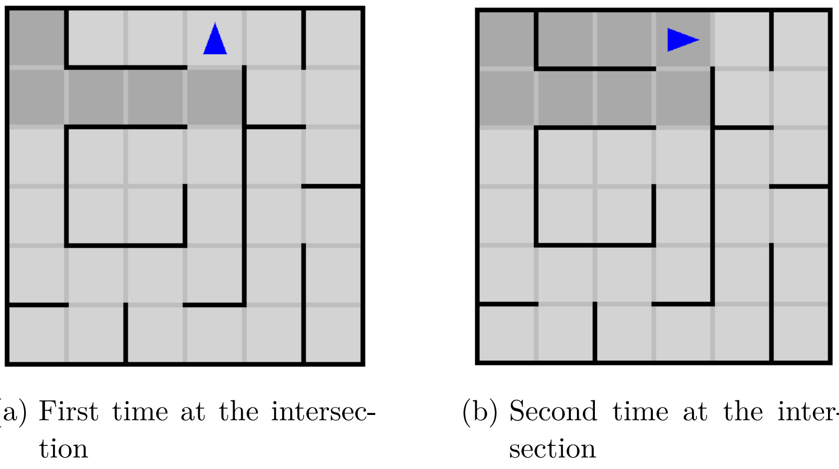

(a) First time at the intersection (b) Second time at the intersection

Figure 3.3.: Intersection behavior

3.4. Circles

Since the mouse applies a simple depth-first search to find the goal tile, it is not guaranteed that the path over which it finds the goal tiles is the shortest path. To ensure that the shortest path to the goal tile is found, the mouse remembers all circles which it encounters. A circle is found when the next step the mouse wants to take while moving forward has already been explored. This only applies when the mouse is not backtracking.

Each circle consists of the same sections as shown in Figure 3.4b: I has a linear part (green) that starts at the initial mouse position and ends at the point where the path meets itself. The loop (red) then contains the rest of the steps which start and end at the same tile.

The use of circles to compute the shortest path will be explained in section 3.6.

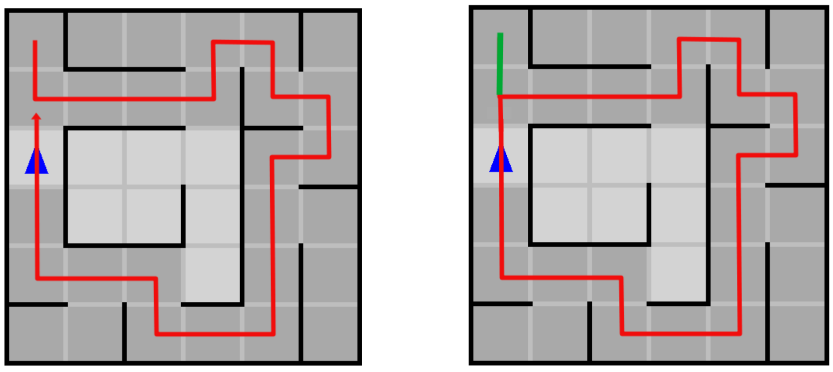

(a) Circle stored by the mouse (b) Sections of a circle

Figure 3.4.: Circle

3.5. Finding the goal

The goal is an area of 2x2 tiles. Therefore, the mouse assumes to have reached the goal area when it walked a circle where the last tile is the same as the tile 4 steps earlier as shown in Figure 3.5. A copy of the list of steps that were taken until this point is stored internally as a path to the goal area. The steps within the goal area are included in this list.

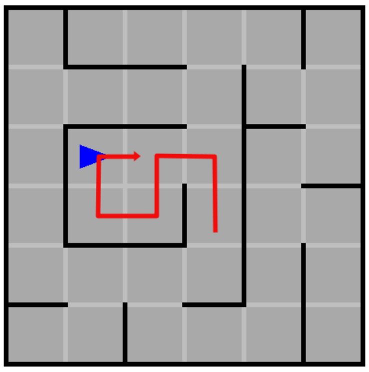


Figure 3.5.: Identifying the goal area

3.6. Completing the search and computing the final path

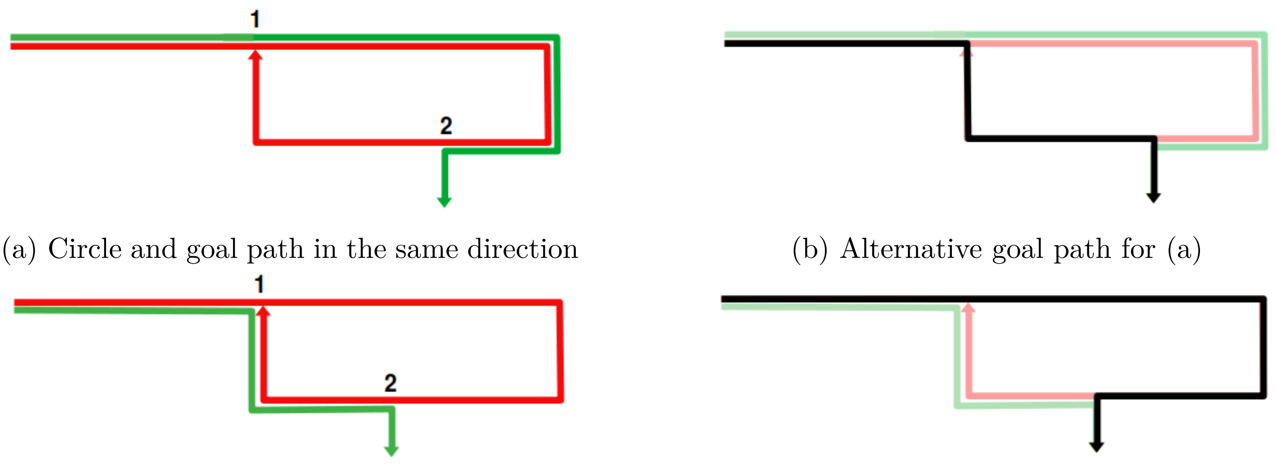
The search is assumed to be complete, when all tiles have been explored and the mouse is back in its starting position.

Now the shortest path to the goal area is computed by combining goal paths and circles. For each possible combination of goal path and circle a new path is created.

Because all goal paths and all circles start at the initial mouse position we can assume that the step lists are equal until they separate at a certain point. After that the goal path leads to the goal area and the circle back to itself. If the separation is in the straight part of the circle, the path of the circle leads its loop and the goal path to the goal area. The new path has to choose between those two paths and because the goal area is the main objective, the circle is always ignored. The resulting new path is then only a copy of the goal path, therefore, no new path is created.

If the separation lies in the loop part of the circle, a new path can be found via one of two options depending on how the goal path runs in relation to the loop.
The first option is shown in Figure 3.6a. The steps of the circle follow the goal path until position 2. From there the goal path leads to the goal area whereas the circle continues its loop until position 1 where it meets itself again. As a consequence, by walking the loop of the circle against its original direction, a new path, which leads to the goal area, can be created. This alternative path is shown in Figure 3.6b.
The second option is shown in Figure 3.6c. The goal path leads along the straight part of the

circle until position 1. It then continues against the direction of the loop until it leaves the loop at position 2. The new path therefore follows the direction of the loop from position 1 until position 2 as shown in 3.6d.



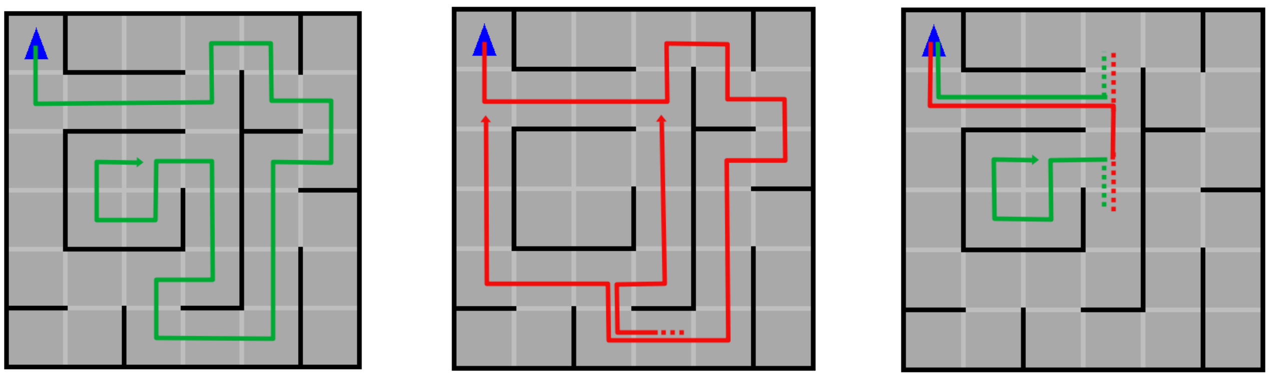
(a) Circle and goal path in the same direction (b) Alternative goal path for (a)

(c) Circle and goal path in opposite directions (d) Alternative goal path for (c)

Figure 3.6.: Combination of circles and goal paths

After computing the alternative path for each combination of circle and goal path the shortest path is picked as the result of the search.

In the maze example one goal path (Figure 3.7a) and and two circles (3.7b) are found. The combination of goal path and the inner circle results are done following the case shown in Figure 3.6a and Figure 3.6b. The resulting shortest path is shown in Figure 3.7c.



(a) Goal path (b) Circles (c) Combined goal path

Figure 3.7.: Applying the combination to the example maze

3.7. Implementation

The starting point of the mouse is hard-coded but it can be varied before flashing the code onto the microcontroller. The coordinates are stored in the macros `START_POSITION_X` and `START_POSITION_X`. The solving logic works for each starting position but the coordinates must be given as a reference point for the maze.

Exploration matrix

In the 6x6 matrix `tiles` of `int` values the explored tiles are marked. If the tile with the coordinates [x, y] (see Figure 3.1) was already explored, the matrix contains the value 1 at position [x, y]. If the tile is unexplored the value 0 is stored.

3.8. Structs

To improve the readability of the code, data is grouped into `structs` :

- Tile
  A `Tile` contains the x- and y-coordinate of a tile within the maze.

- Tile_array
  A `Tile_array` contains an array of `Tile`s. This is used to store e.g. the steps that the mouse has made so far. Because the length of the array is included in the `struct`, it doesn't have to be calculated in the code.

- Tile_matrix
  A `Tile_matrix` contains an array of `Tile_array`s. This `struct` is used to store e.g. the circles that are found during the search. Like in the `Tile_array`, the number of arrays stored in the `Tile_matrix` is stored as well.

- Revisit_tile
  When the mouse finds an interception, the position and the possible directions as described in 3.3 are stored in a `Revisit_tile`. The number of possible directions is stored in the `Revisit_tile` as well. This is important because an array of `Direction`s always has length 4.

- Tiles_to_visit_again
  `Revisit_tiles` are stored in the `struct Tiles_to_visit_again`. This array can be queried to retrieve the next possible direction for an intersection that has been visited again (see Section 3.3).

- Maze_solver
  This is the main `struct` that stores the data relevant to the search. It contains the intersections that have to be visited again, whether the mouse is currently backtracking, the current list of steps, the circles, goal paths as well as the start and goal tiles.
- Next_step
  This `struct` is returned to the calling function after each step calculation. It contains the next step that has to be taken, the direction which the mouse faces, as well as the information whether the search is done.

3.9. Method structure

In the following section select parts of the code that are either especially important or not self-explanatory are looked into in more detail.

Calling the maze solver

The maze solver is called once before every new step. The initialization takes place in the `main()` method. The calculation of the next step is triggered by the method `compute_next_direction()`. It takes the sensor data of whether there is a wall left, right or ahead of the mouse as parameters and returns an array of size 2. The first element contains the `Direction` in which the mouse moves next. The second element contains the number of steps that it takes in this direction.

Cleaning the list goal paths

When the mouse finds an intersection all unexplored directions are stored. If this intersection is also the endpoint of a circle, one of the directions is explored and marked When the mouse returns to the intersection via backtracking the next tile in this direction is already marked as explored. Therefore, the way to this tile is also interpreted and stored as a circle. These falsely stored circles are filtered out in the method `clean_goal_paths()`. It is checked whether the last saved tile occurs a second time in the list of steps. If this is not the case, the path is deleted.

As shown in Figure 3.8a the intersection is stored as

$$intersection : [0, 1], [WEST, SOUTH] \tag{3.5}$$

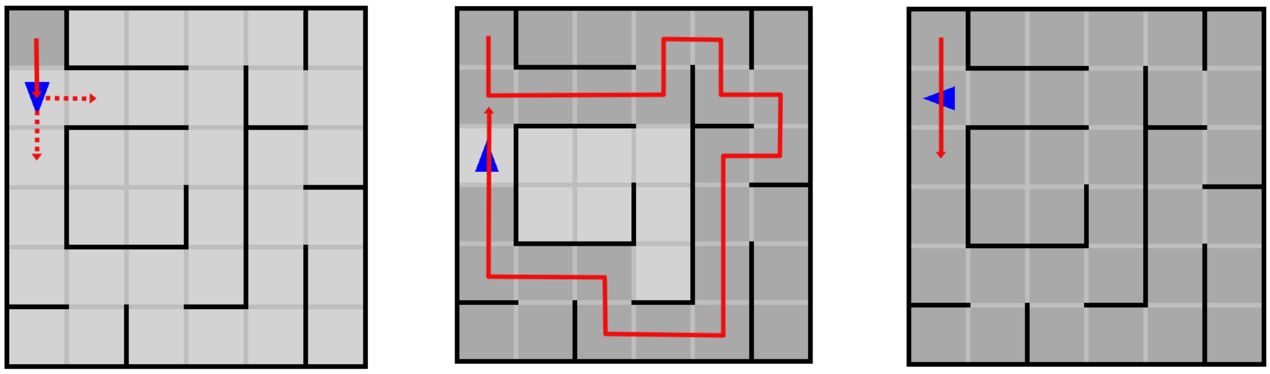
where West is removed again for the next step. However, this direction is not removed after the circle in Figure 3.8b which results in the following data being stored:

$$intersection : [0, 1], [SOUTH] \tag{3.6}$$

$$circle : [0, 0], [\mathbf{0,\ 1}], [1, 1], [2, 1], ..., [0, 3], [0, 2], [\mathbf{0,\ 1}] \tag{3.7}$$

In 3.8c this stored direction is used to explore tile [0, 2]. However, since this tile is already explored, it is assumed that another circle has been found with the following steps:

$$circle : [0, 0], [0, 1], [0, 2] \tag{3.8}$$



(a) Stored directions (b) Correctly saved circle (c) Falsely saved circle

Figure 3.8.: Stored circles at an intersection

In contrast to the circle in (3.7) where the connection tile [0, 1] is contained twice, the falsely saved circle does not have such a duplicated tile.

Combining circles and goal paths

In `compute_all_goal_paths()` for each goal path its combination with each circle is computed. Based on the strategy explained in Section 3.6 each new path consists of three parts.

The first part is the linear section of the circle (see 3.4b, green). Since only combinations where the goal path separates from the circle in its loop are considered, the linear section can be copied entirely.

The second part is the part of the circle which does not follow the goal path. To receive this section of the circle, the part of the loop that contains the same steps as the goal path are removed. If the goal path and the loop run into the same direction (see Figure 3.6a) the steps are removed from the loop in the same order in which they were saved. If they run in opposite directions (see Figure 3.6c) the steps are removed from the end of the list. The remaining steps are used as the second part of the new path. In the first case (circle and goal path run in the same direction) the list of steps has to reversed before it can be used.

The third section of the new path are the steps that the goal path makes on its own after having separated from the loop.

4. Controller Design and Approach

The goal for the controller design was to completely abstract the software from the hardware, by providing the following callback function to the "exploration and solving" part of the code:

```
typedef struct {
    int direction;
    int repeat;
} Step;

Step compute_next_step(int robot_direction,
        int left_free, int front_free, int right_free);
```

With this callback function, the "exploration and solving" part of the code receives the current direction of the robot (NORTH, EAST, WEST, SOUTH) and three booleans telling if there are currently a wall on the left, front and right of the robot. The function is then supposed to return the next step to perform, which is made of a direction (NORTH, EAST, WEST, SOUTH) and the number of steps (i.e. the number of cells) to do in that direction.

This function is called only when the previous step is complete, which makes easier the exploration and solving of the maze, that simply "sees" the robot as a discrete system, making steps in a discrete maze.

Since all the code for the control, exploration and solving runs in the same interrupt, the idea of using a callback function to make steps has the benefit that "making a step" will not lock the interrupt. During the step, the interrupts continue to be called, which allows to correct the voltage of the motors dynamically using the sensors of the robot (IR sensors and motor encoders). If during an interrupt, the previous step is observed as complete, the `compute_next_step` function is called to get the next step the controller has to perform.

4.1. Making steps

As it was said before, a step is made of a `direction` and a number of steps the robot is supposed to do in that direction. A step can thus be split in two parts (called `Actions`): the "rotating" part, to align the robot with the required direction, and the "forward" part where the robot moves forward with the required number of steps. When a step is returned from the callback function `compute_next_step`, the list of actions to perform to do that step are stacked into an array. The callback is called again only when all the actions have been executed, to get the next actions the controller has to do.

Since the direction is an absolute direction (NORTH, EAST, WEST, SOUTH), the controller always keeps in mind the current discrete orientation of the robot. This orientation is initialized with the initial orientation of the robot (that is supposed to be known) and updated at the end of each action.

4.2. Actions

An action is made of a `type` (`TURN_LEFT`: turn 90° left, `TURN_RIGHT`: turn 90° right or `FORWARD`: move one cell forward) and a `repeat` telling how many times the action should be repeated. When the current action is complete, the next one is unstacked from the list of actions to do. Using the current angle of the left and right wheels of the robot, the target left and right angles of the wheels is computed. If the `type` of the action is `FORWARD`, the target left and right angles is computed as follows:

```
float target_left = current_left + DEGREES_ONE_CELL * action.repeat;
float target_right = current_right + DEGREES_ONE_CELL * action.repeat;
```

Where `current_left` is the current angle in degrees of the left wheel, `current_right` is the current angle in degrees of the right wheel and `DEGREES_ONE_CELL` is the number of degrees the wheels should rotate to move the robot one cell forward in the maze. Similar equations are used if the `type` of the action is `TURN_LEFT` or `TURN_RIGHT`. The goal for the controller is then to reach the `target_left` and `target_right` angles for both wheels as fast as possible, by playing on the power of the motors.

4.3. Controllers

Three different functions were written to control the robot to make a given action:

- `control_left`: If the `type` of the current action is `TURN_LEFT`, this function uses the motor encoders to turn in place and makes sure that the speed of the wheels are opposite (left speed negative, right speed positive)
- `control_right`: If the `type` of the current action is `TURN_RIGHT`, this function uses the motor encoders to turn in place and makes sure that the speed of the two wheels are opposite (left speed positive, right speed negative)
- `control_forward`: If the `type` of the current action is `FORWARD`, this function uses the motor encoders and the left and right IR sensor of the robot to move forward, while keeping a safe distance from the walls

The three functions have the same structure. From the target angle of the left and right wheels, the first part is to compute the absolute power of the motors: if the error between the current angles of the wheels and the target angles is high, the power should be high, to reach the target faster. But if the error is small, this power should be decreased to avoid overshooting the target. After some experiments, it was also noticed that if the power of the wheels switches too fast between 0% and 100%, the wheels can slip on the ground, which has to be avoided since we are using the motor encoders to estimate the position of the robot. To correct this problem, a bloc (slope limit) was added to limit the variations of the power of the motors. The complete bloc, computing the absolute power of the motors is illustrated in Figure 4.1.

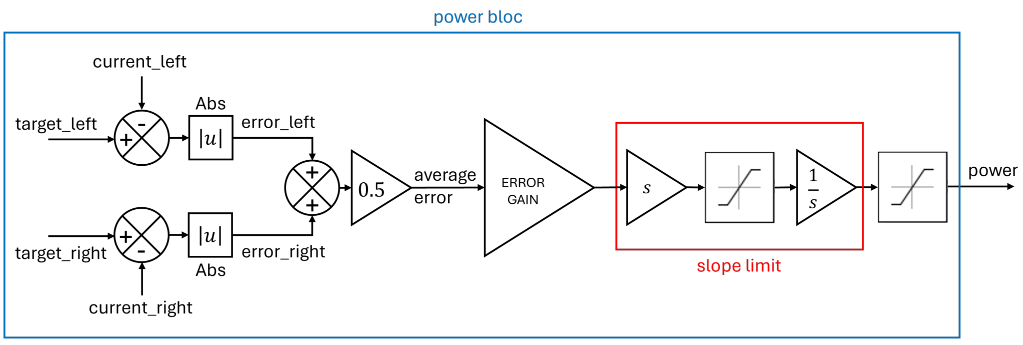


Figure 4.1.: Block diagram to compute the absolute power of the motors

From the previously computed aboslute power for the motors, the real power of the motors should be adjusted to take into account the speed error of the wheels (from the motor encoders) and the position error of the robot (from the IR sensors). Depending on these errors, the speed of the wheels is thus increased or decreased to decrease the errors.

If the goal is to turn left in place, the speed of the wheels should be opposite:

`rpm_left_motor`+`rpm_right_motor`=0, with `rpm_left_motor`<0 and `rpm_right_motor`>0.

Let `speed_error`=`rpm_left_motor`+`rpm_right_motor`.

- If `speed_error`=0: Since we want to turn left, the left wheel should be driven backward and the right wheel should be driven forward: `left_power` = −`power` and `right_power` = +`power`.
- If `speed_error` > 0: The right wheel is too fast compared to the left wheel, so the left wheel should move faster backward and the right wheel should move slower forward: `left_power` = −`power` − `speed_error` and `right_power` = +`power` − `speed_error`
- If `speed_error` < 0: The right wheel is too slow compared to the left wheel, so the left wheel should move slower backward and the right wheel should move faster forward: `left_power` = −`power` − `speed_error` and `right_power` = +`power` − `speed_error`

The reasoning is similar if the goal is to turn right in place. The final control blocs that are used to turn left in place or right in place are illustrated in Figure 4.2.

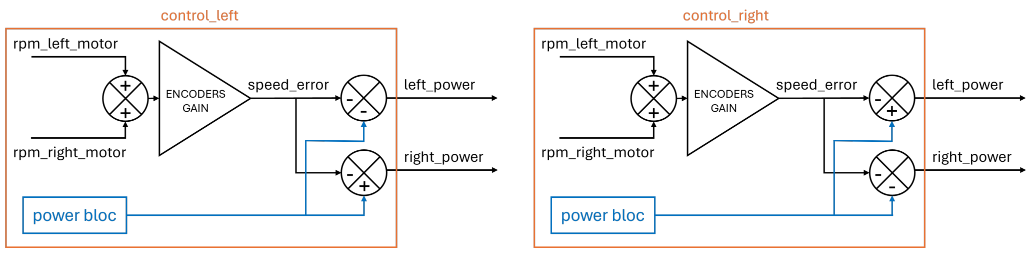


Figure 4.2.: Block diagram of the controllers to turn left or right, in place (The power bloc is illustrated in 4.1)

If the goal is to move forward, the control is similar, except that we also need to take into accound the left and right IR sensors to prevent the robot from hitting the walls. To do this, we need to use the distance measures from the left and right IR sensors of the robot, to estimate if the robot should be shifted left or right.

The first idea to do this is to compute the middle between the left and the right wall and ask the robot to align with that position. However, this will not work if the robot is crossing an intersection and only the left wall (or the right wall) is available to estimate the position of the robot. Thus, the robot is instead using the nearest wall to correct its position, by trying to keep the same distance with that wall. With this solution, the robot will be able to move straight even if only the left wall (or the right wall) is there. If both left and right walls are too far away, the position error of the robot is set to zero and the robot will have to rely only on the motor encoders to move straight. Assuming that `position_error` > 0 means the robot should move right and `position_error` < 0 means it should move left, the final control block to move the robot forward is illustrated in Figure 4.3.

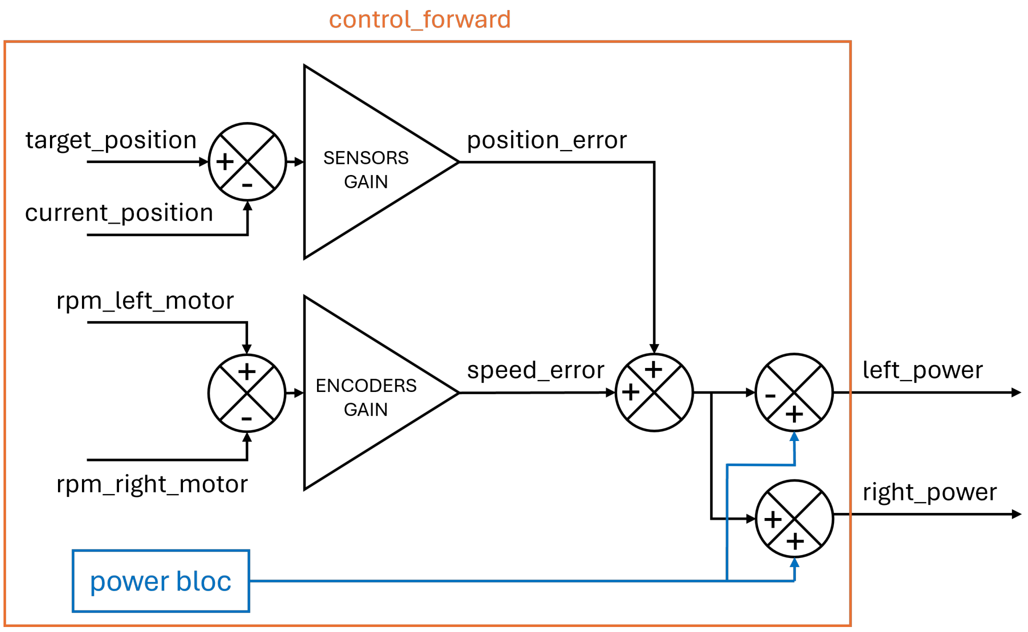


Figure 4.3.: Block diagram of the controller used to move forward (The power bloc is illustrated in 4.1)

5. Testing and Validation of Robot Performance

This section documents the results of the tests performed on the sensors, motors, and navigation system of the Micromouse robot. To access the raw values returned by the components we will use functions developed earlier on to abstract lower-level details and deal only with function calls.

5.1. Overview of navigation and control tests

Once our controller was developed, we needed to adapt the right and left error coefficients to adjust the power supplied to the motors. For testing the controller, we had a couple of trajectories to complete in order to ensure stable maze navigation.

The first trajectory was a straight line. The robot was placed in a 5-cell row and had to continuously adjust its trajectory with respect to the walls on its right and left. Through this first scenario, we determined the power levels that should be supplied to our robot and the proportional gains of our controller.

The other trajectories were 90 and 180-degree turns. These required us to adjust target angle values to avoid under- or over-steering. During testing, we encountered problems such as wall collisions when the acceleration was too high. We fixed this by limiting the voltage and adjusting the error gain in the power bloc of the controller.

Our controller takes into account distances from sensors on the sides, but front collisions on our robot were frequent. The robot would move forward based on a target angle that the wheels had to reach. With more testing, we realized that before making any more steps, we should check the front distance and stop navigation if it was smaller than a given value. This approach worked in some cases but also had shortcomings.

For instance, the robot would unexpectedly stop during 180-degree turns. We decided to remove this braking mechanism and modify the target angles instead. With enough patience and repetition, we found parameters that allowed for reliable navigation in all the scenarios our robot was tested on

5.2. Distance sensor tests

The testing and calibration of distance sensors were based on approximating the relationship between voltage readings and distance values. Initially, approximations using logarithmic, exponential, and polynomial functions were considered. However, only polynomials of order 5 or 6 provided reasonably accurate results. Despite this, polynomial functions were found to be computationally expensive.

To optimize performance, a custom software implementation was developed that approximates the curve using multiple linear functions instead of polynomials. This approach was preferred because linear functions offer faster computations, requiring only one multiplication and one addition, which significantly reduces the processing time compared to polynomials.

The software defines a function `voltage_to_distance(float voltage)`, which returns distance values based on different voltage ranges. It uses segmented linear equations, each specific to a range of voltages, to map the sensor voltage to a corresponding distance.

This method achieves sufficient accuracy while enhancing computational efficiency during real-time distance sensing.

To test our functions, we placed the robot in a cell and measured the distance between the sensors and walls using a ruler.

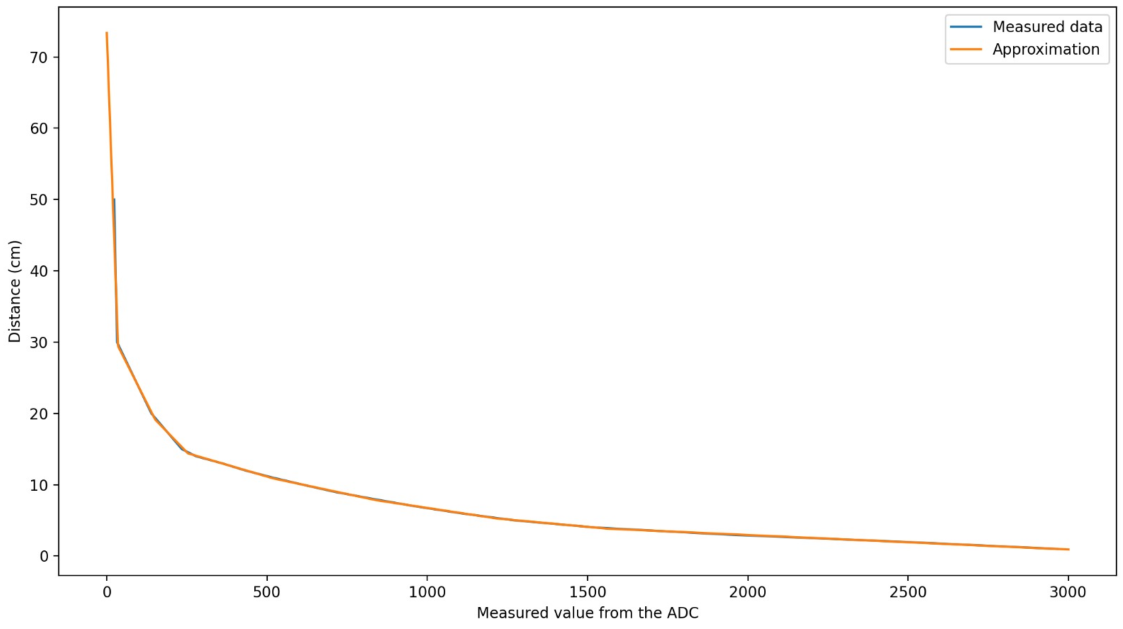


Figure 5.1.: Voltage-to-Distance Mapping Using Segmented Linear Approximations

5.3. Summary of challenges encountered

Throughout the development of the robot we have faced problems and surprises:

- **Time** management was one of our biggest challenges during this project. Working on an embedded device requires a decent amount of attention to detail as we are as close to the hardware as possible. Often times, small problems took us more than a few hours to solve in just one or two changes. Therefore our sessions dedicated to the robot, almost always went beyond the time limit we set ourselves.

- **Debugging** on a bare-metal robot without any wireless communication was not straight-forward process. Indeed, when our algorithms did not have the intended behavior, the challenge was to know in which state the robot was with respect to the task. For example in the maze exploration phase, our robot would sometimes explore only a few cells before coming back to the starting position. We needed a way to know what our sensors and algorithm returned, for that we could use the UART interface but this required keeping the cable directly above the robot and read the transmitted data. This required us to have a better understanding of the code before running tests because of the setup time before every test run.
- **Fine-tuning** values for our modules (e.g. controller, exploring, robot navigation) took a lot of time. Especially changing values for our algorithms was an iterative process. With wrong values our mouse would have collisions and it can be scary when there is a strong collision.

6. Conclusion

Valuable practical experience was gained throughout the Micromouse project in areas such as hardware design, embedded systems, and software development. Several challenges were encountered and addressed, including the design and assembly of printed circuit board, motor control, sensor integration, and algorithm development for maze navigation. Technical skills were enhanced, and problem-solving abilities were developed through working within real-world constraints.

Looking ahead, there are several opportunities to improve the Micromouse robot's design and functionality, both in the hardware and software parts. For the hardware, one critical area for improvement is the miniaturization of the printed circuit board (PCB), which could improve maneuverability, especially for 180-degree turns. Additionally, including heat dissipation mechanisms, such as heat sinks, is essential to prevent overheating of electronical components. Furthermore, the choice of UART wired communication was suboptimal, as the cable complicates data retrieval during the robot's movement. Switching to a wireless communication method would greatly facilitate the software debugging process. For the software, a better exploration and solving algorithm could've been implemented for the maze. In the current version, the robot still needs to explore every cell in the maze before solving it, while this is often not necessary if the goal is found early during the exploration. Moreover, the solving could've also been improved, since the shortest path is not necessarily the fastest one for a physical robot (straight lines can be travelled faster than quarter turns). Finally, one big improvement for the solving could've been to allow the robot to move in diagonal in the maze. This however requires a more complex and better control of the robot, as well as a smaller PCB.

Overall, this project has provided us with valuable insights into designing and building the various modules necessary for a robot to autonomously operate. As autonomy becomes an increasingly significant milestone in today's society, the knowledge and skills gained from this course will be beneficial in our future projects.

6.1. Feedback and Suggestions

- Having a practical course that teaches the needed information in the beginning is really helpful for bringing everyone to a knowledge base from where they can start the project without leaving anyone behind.
- One challenge encountered during the course was related to the timing of the PCB design and component ordering process. Our team had their PCB design and electronic components ready two weeks ahead of other teams, but due to the requirement for all teams to place orders simultaneously, we had to wait for the others to catch up. This resulted in a loss of valuable time for progressing with the project. While coordination between teams

is understood, allowing for staggered ordering in future courses could benefit students who are ready to work at their own pace.

- Perhaps it would be helpful to give the students a rough idea how long which part of the project usually takes how long or even have them create a rough timeline of their project. This way the teams have to think about this aspect in the beginning and might get a better feeling for the time they need and the time they have.

A. Appendix

A.1. Solutions to the example mazes

A.1.1. Resulting C arrays

Maze 1

[0, 0], south: [0, 0], [0, 1], [1, 1], [2, 1], [3, 1], [3, 2], [4, 2], [5, 2], [5, 3], [4, 3], [4, 4], [3, 4], [3, 3]

[5, 5], north: [5, 5], [5, 4], [5, 3], [4, 3], [4, 4], [3, 4], [3, 3]

Maze 2

[0, 0], south: [0, 0], [0, 1], [1, 1], [2, 1], [3, 1], [4, 1], [4, 2], [3, 2], [3, 3], [2, 3], [2, 2]

[5, 5], north: [5, 5], [4, 5], [3, 5], [3, 4], [4, 4], [4, 3], [4, 2], [3, 2], [3, 3], [2, 3], [2, 2]

Maze 3

[0, 0], south: [0, 0], [0, 1], [1, 1], [2, 1], [3, 1], [3, 2], [2, 2], [2, 3], [1, 3], [1, 2]

[5, 5], north: [5, 5], [5, 4], [5, 3], [4, 3], [4, 4], [4, 5], [3, 5], [2, 5], [2, 4], [3, 4], [3, 3], [3, 2], [2, 2], [2, 3], [1, 3], [1, 2]

Maze 4

[0, 0], east: [0, 0], [1, 0], [2, 0], [3, 0], [3, 1], [3, 2], [3, 3], [4, 3], [4, 2], [5, 2], [5, 1], [4, 1], [4, 0], [5, 0]

[5, 5], north: [5, 5], [5, 4], [5, 3], [5, 2], [5, 1], [4, 1], [4, 0], [5, 0]

Maze 5

[0, 0], south: [0, 0], [1, 0], [2, 0], [2, 1], [2, 2], [1, 2], [1, 1]

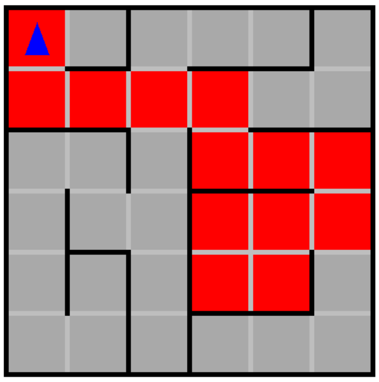
[5, 5], west: [5, 5], [4, 5], [3, 5], [3, 4], [4, 4], [5, 4], [5, 3], [5, 2], [5, 1], [4, 1], [3, 1], [3, 0], [2, 0], [2, 1], [2, 2], [1, 2], [1, 1]

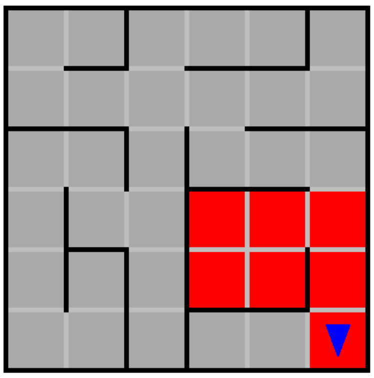
Maze 6

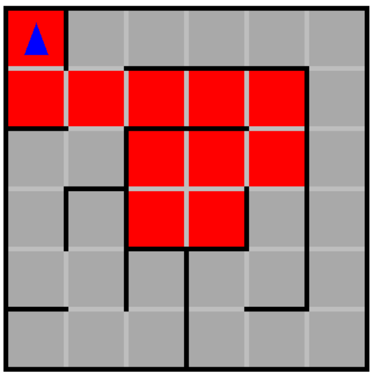
[0, 0], south: [0, 0], [0, 1], [0, 2], [0, 3], [1, 3], [1, 2], [1, 1], [2, 1], [3, 1], [3, 2], [2, 2]

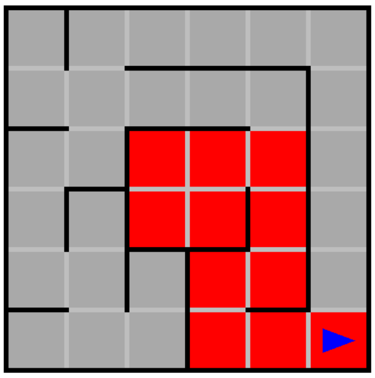
[5, 5], north: [5, 5], [5, 4], [5, 3], [4, 3], [4, 2], [4, 1], [4, 0], [3, 0], [2, 0], [1, 0], [1, 1], [2, 1], [3, 1], [3, 2], [2, 2]

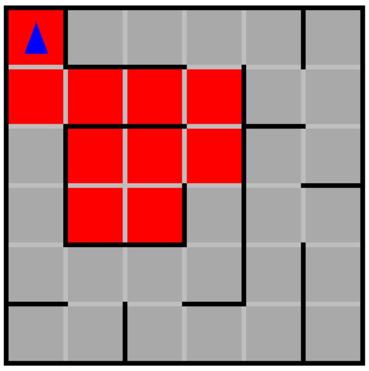
A.1.2. Python visualization

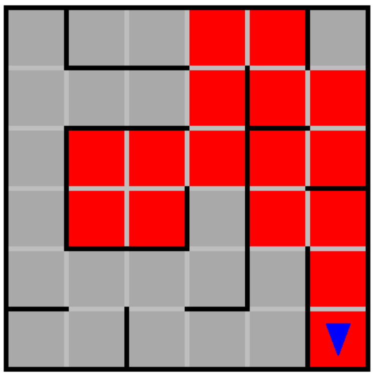

(a) Maze 1: [0, 0], south

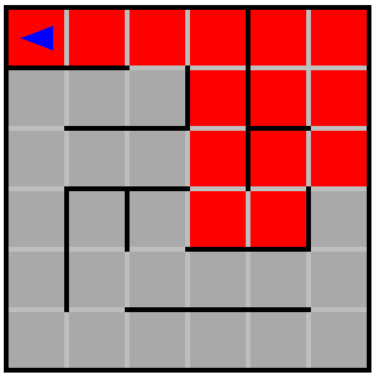

(b) Maze 1: [5, 5], north

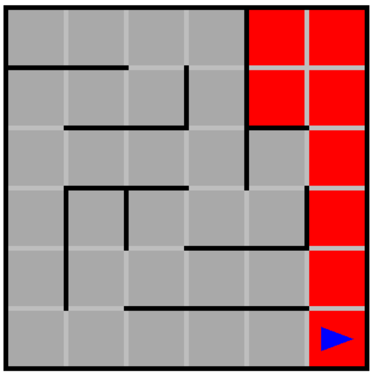

(c) Maze 2: [0, 0], south

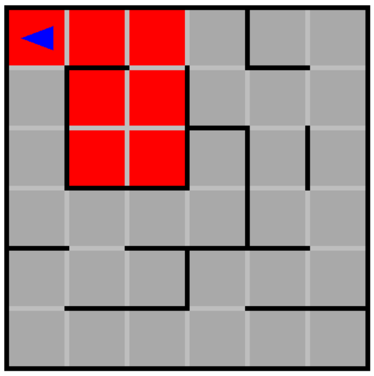

(d) Maze 2: [5, 5], north

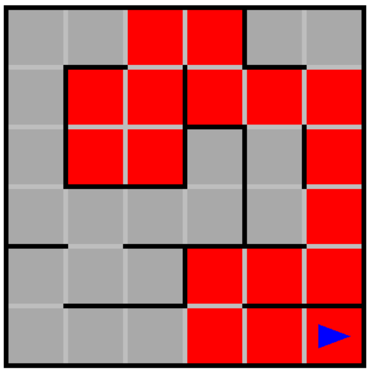

(e) Maze 3: [0, 0], south

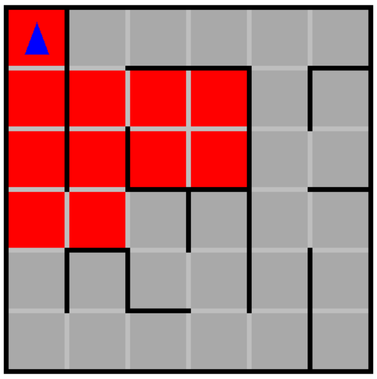

(f) Maze 3: [5, 5], north


(g) Maze 4: [0, 0], east


(h) Maze 4: [5, 5], north


(i) Maze 5: [0, 0], south


(j) Maze 5: [5, 5], west


(k) Maze 6: [0, 0], south

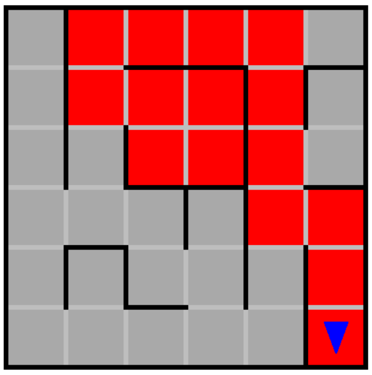

(l) Maze 6: [5, 5], north

Figure A.1.: Exemplary mazes and their solution including starting coordinates and starting direction