DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Practical Course Report (IN2106)

Micromouse - Team Maze Runner

Nicola Baur, Philipp Burgkart, Roman Langenscheidt, Noah Bucher

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Practical Course Report (IN2106)

Micromouse - Team Maze Runner

Designing an Educational Racing-Robot from Scratch

Authors:	Nicola Baur, Philipp Burgkart, Roman Langenscheidt, Noah Bucher
Supervisor:	Dr. Alexander Lenz
Submission Date:	10.10.2023

We confirm that this master practical course report (IN2106) is our own work and we have documented all sources and material used. We have all equally contributed to this project.

Munich, 10.10.2023 Nicola Baur, Philipp Burgkart, Roman Langenscheidt, Noah Bucher

Contents

1. Introduction

The Micromouse challenge was first introduced in 1977 and has gained widespread recognition as a robotics competition. It has gained popularity especially in the U.S., Japan, India, and South Korea [1]. The challenge revolves around a maze consisting of adjustable configurations of 16x16 cells. An autonomous robotic mouse is tasked with exploring this maze, starting its journey from one of the corners. The robot's primary objective is to find a way to the goal in the center of the maze. Throughout the exploration phase, the robot enters a cell and autonomously detects which sides of the cell are blocked off by walls. Using this information, a map of the maze is created and different algorithms can be employed to determine the fastest way from start to center. The actual performance of the robot is measured based on the time it takes the robot to reach the center of the maze from the starting corner in a final run, after the maze is explored. The Micromouse challenge encompasses the entire process of conceptualization, construction, and execution of both the hardware and software components of the racing robot requiring participating teams to be multidisciplinary and knowledgeable in areas such as electronics, PCB design, and embedded software.

As a central component of the hands-on course titled "Micromouse: Designing an Educational Racing Robot from Scratch" offered by Dr. Lenz, our team embarked on the construction of such a Micromouse robot. Following a series of introductory sessions focused on theory, encompassing topics such as electronics, embedded software, and PCB design, our team took charge of the entire journey, spanning from the initial conceptual sketches to building a robot able to move through the maze and find the shortest path to the goal cell. The primary aim of this report is to provide an in-depth description of the final robot and to document the outcomes and insights gained along the way.

In Chapter 2, we provide a comprehensive rationale for the conceptual design of both the hardware and software components of our Micromouse. Moving on to Chapter 3, we delve into the practical implementation of the board design using the widely-adopted software tool Autodesk Fusion 360 [2]. Within this chapter, we detail the individual electrical components and their specific arrangements on the PCB. Chapter 4 is dedicated to an examination of our codebase, discussing the functionalities of the distinct software layers. Subsequently, Chapter 5 is devoted to the presentation of all performed hardware and software tests. Chapter 6 summarizes our accomplishments, highlights key learnings and gives an outlook on design changes for a next iteration. To access our codebase, please refer to our GitHub repository[1].

[1] https://github.com/philison/micromouse.git

2. Conceptual Design

This chapter provides an overview over the main hardware and software components of the micromouse.

2.1. Conceptual Hardware Design

The micromouse is designed as a differential drive robot. The principle design of our micromouse is shown in figure 2.1. Distance sensors for wall detection are placed in front with a relative orientation of 90°. Underneath the sensor mount for the distance sensors, a LED serves as the third contact point in addition to the placed motors in the rear. All parts and their respective mounts are directly placed onto the printed circuit board (PCB). The micromouse has a width of 99.8 mm and a length of 87.5 mm. Those dimensions were a result of the space taken up by the electronics and the dimensions of the maze (consisting of 18 cm squares) where the micromouse has to operate in.

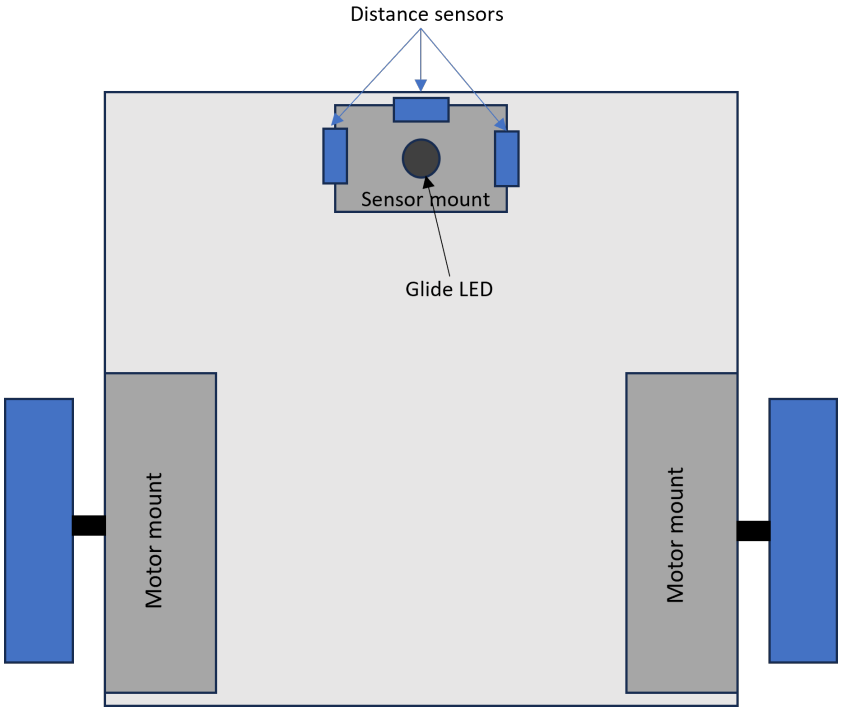


Figure 2.1.: Basic design of our Micromouse.

A high-level overview of the electronic design is shown in figure 2.2. The heart of the micromouse is a microcontroller which processes sensor information, executes the different software components and eventually controls the mouse. Infrared distance sensors are used to receive information about the maze environment. Communication is possible via UART and Bluetooth. Further visual feedback can be obtained with LEDs. Buttons allow for user input and direct access to the reset functionality of the micro controller. The DC-Motors, which were provided by Dr. Lenz, include encoders and are controlled with a dual H-Bridge. Power is provided by a 9V-Battery whereas linear voltage regulators reduce the voltage to values that are needed by the components.
The detailed design of the sensor and motor mounts as well as the electronic design is explained in chapter 3.

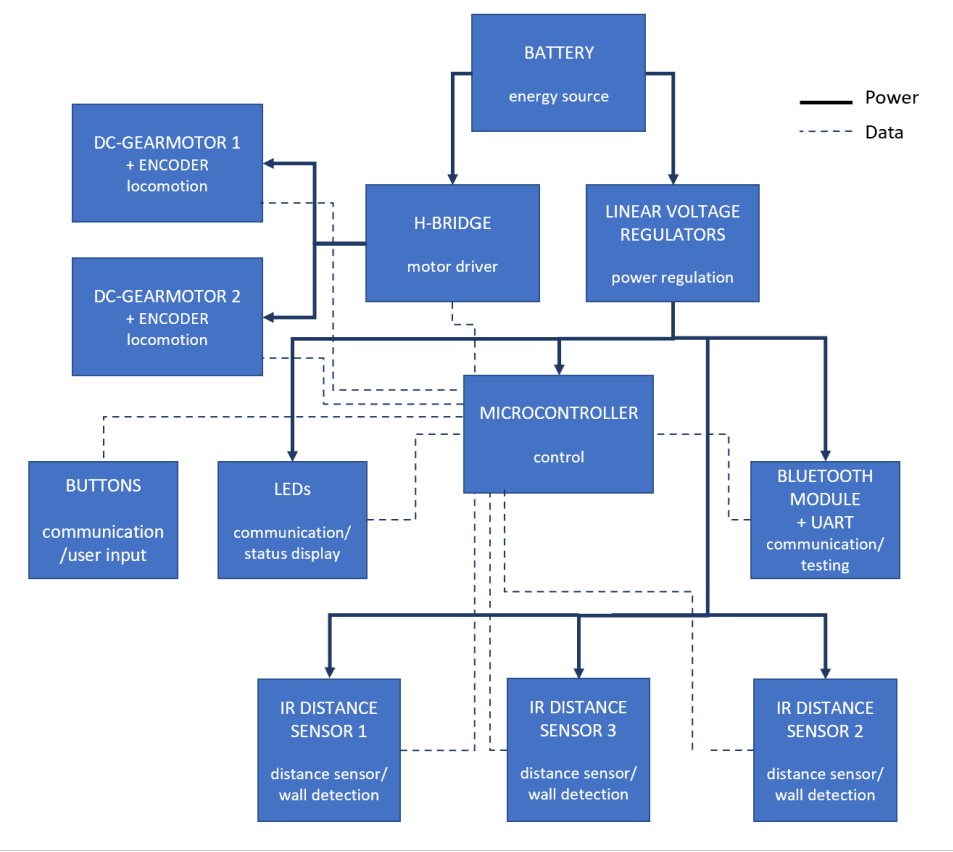


Figure 2.2.: Basic electronic design with main electronic components.

2.2. Conceptual Software Design

To enable different software modules to interact with the described hardware, a Hardware Abstraction Layer (HAL) is developed. It provides access to various data sources and enables the control of the used hardware components through simple functions. These functions, for

example, process raw sensor inputs or set appropriate register values to change the PWM duty cycle of the motors. Therefore, the HAL is naturally composed of the abstractions of the different microcontroller modules, depicted as **A**-**F** in Figure 2.3, that are designated to handle the respective hardware. A detailed list of some of the functionalities provided by the abstracted modules is provided on the right side of the figure below.

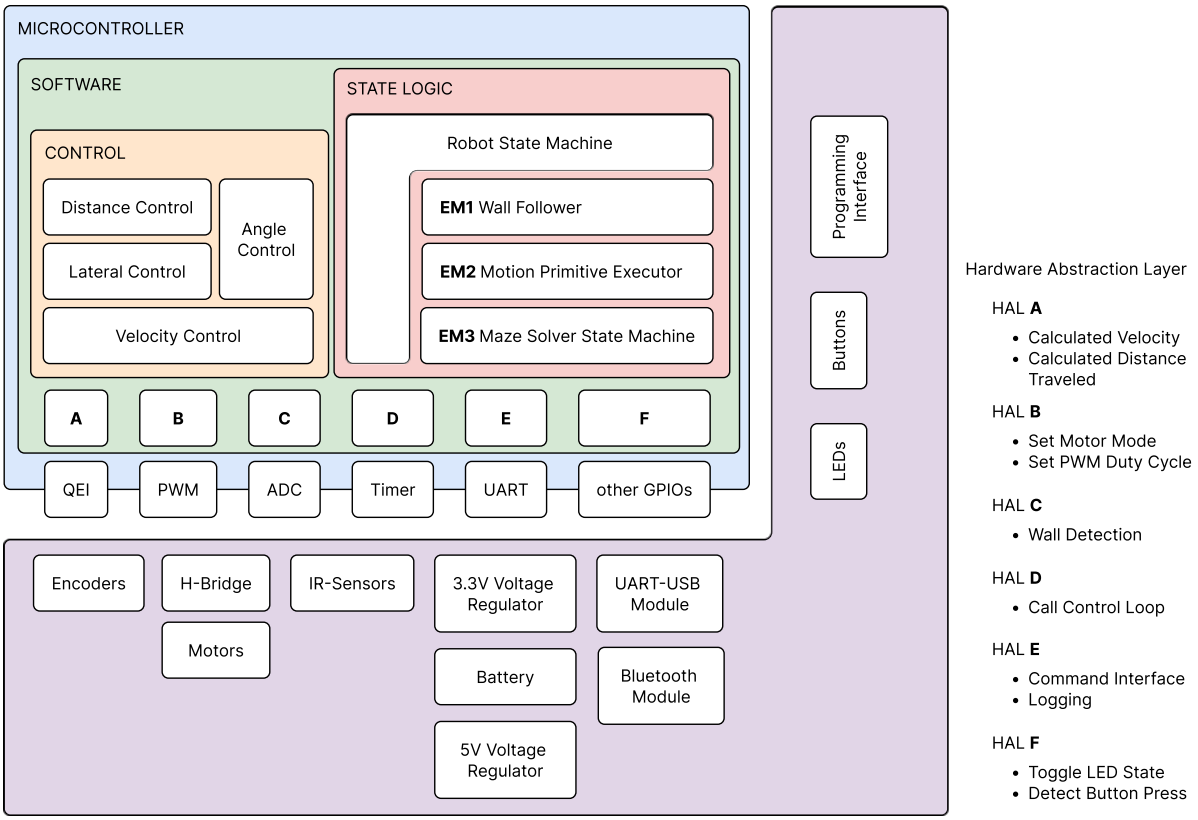


Figure 2.3.: High-level overview of the software structure and its intersection with the hardware stack. The modules named **A** to **F** constitute the Hardware Abstraction Layer, providing abstracted functionality to the software structure built on top. At most, one execution mode (**EM1** to **EM3**) can be selected to be executed by the robot state machine.

The higher-level software modules can be divided into a control stack and a state machine stack, which are marked as orange and red respectively in Figure 2.3. The control stack receives instructions from the state machine, determined by the currently selected execution mode (**EM1** to **EM3**). These instructions can be targeted directly at the velocity controller at the bottom of the control stack to drive the motors at a set velocity. Alternatively, they can, for example, make use of the implemented distance control to move the robot to a goal at a specified distance. This system enables a highly flexible software stack, allowing the robot to handle a wide variety of tasks and simplifying the implementation of new behaviors and capabilities. A new algorithm, such as achieving right wall following, can be implemented as

a new execution mode within the robot's state machine. This approach provides the flexibility to easily introduce different behaviors. Further details about the control stack and the robot state machine will be discussed in Section 4.2 and 4.3.

Section 4.4.2 focuses on the Maze Solver execution mode (**EM3**), which encapsulates the algorithm ultimately used to complete the micromouse challenge. In this mode the robot initially conducts the exploration of the maze during the exploration phase, attempting to find the shortest path to the goal located in the center of the maze. The exploration phase as well as the subsequent final run, are based around an implementation of the floodfill [3] algorithm. The implementation utilizes the robot's wall detection functionality to create a digital representation of the maze. By storing the distance from each cell to the goal, the robot can determine the shortest path leading from the start to the goal. As soon as the shortest path is found, the exploration phase concludes, and the robot attempts a final run to reach the goal as quickly as possible.

After the execution mode is completed, the encapsulating robot state machine exits the execution state, as described in Section 4.3.

3. Hardware Design

In this chapter, the design of the sensor and motor mounts as well as the design of the printed circuit board (PCB) with all corresponding components is explained.

3.1. Sensor and Motor Mounts

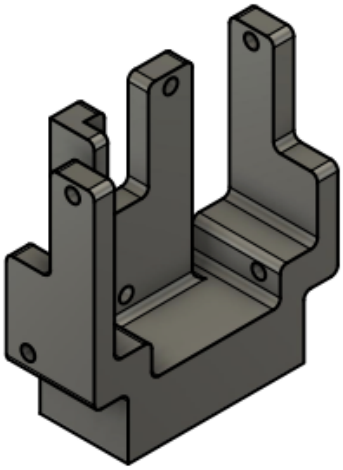


Figure 3.1.: Final design of the mount for the distance sensors.

Concerning the two motor mounts, an STL file of an existing design was provided. However, in the end, no new mounts were printed and already existing ones from the laboratory were used. Regarding the mounts for the distance sensors, we decided against the provided/existing ones from the laboratory. Those mounts were designed such that a separate mount for each distance sensor is required, taking up a lot of space on the PCB. Therefore, we designed a mount which can carry all three distance sensors, resulting in a small footprint. However, we encountered some issues with the initial design of the mount as we observed interference between the distance sensors caused by a too small distance in between them. The redesign of the mount considered greater space between the sensors and a physical separation between the closest connectors was also added. We further tried to separate the sensor cables as much as possible. The redesign of the sensor mount proofed to eliminate the interfering between the distance sensors and is depicted in figure 3.1. A technical drawing of the mount is given in the appendix A.

3.2. PCB Design

In the following, the design of our PCB is explained in detail. The wiring of the PCB is shown in figure 3.2, providing an overview of the components and the respective placement on the board. The full schematics of the PCB can be found in appendix A. While the selection and placement of the different components are explained below, the PCB design follows some general rules: To ensure easy soldering, we chose a 0805 footprint for all capacitors and resistors. Furthermore, the resistors were taken from the E12 series. Decoupling capacitors were used for all power supplies to reduce voltage fluctuations and placed close to the corresponding components. Regarding the positioning of the components on the PCB, we tried to keep the power supply wires as short as possible and separated from data wires.

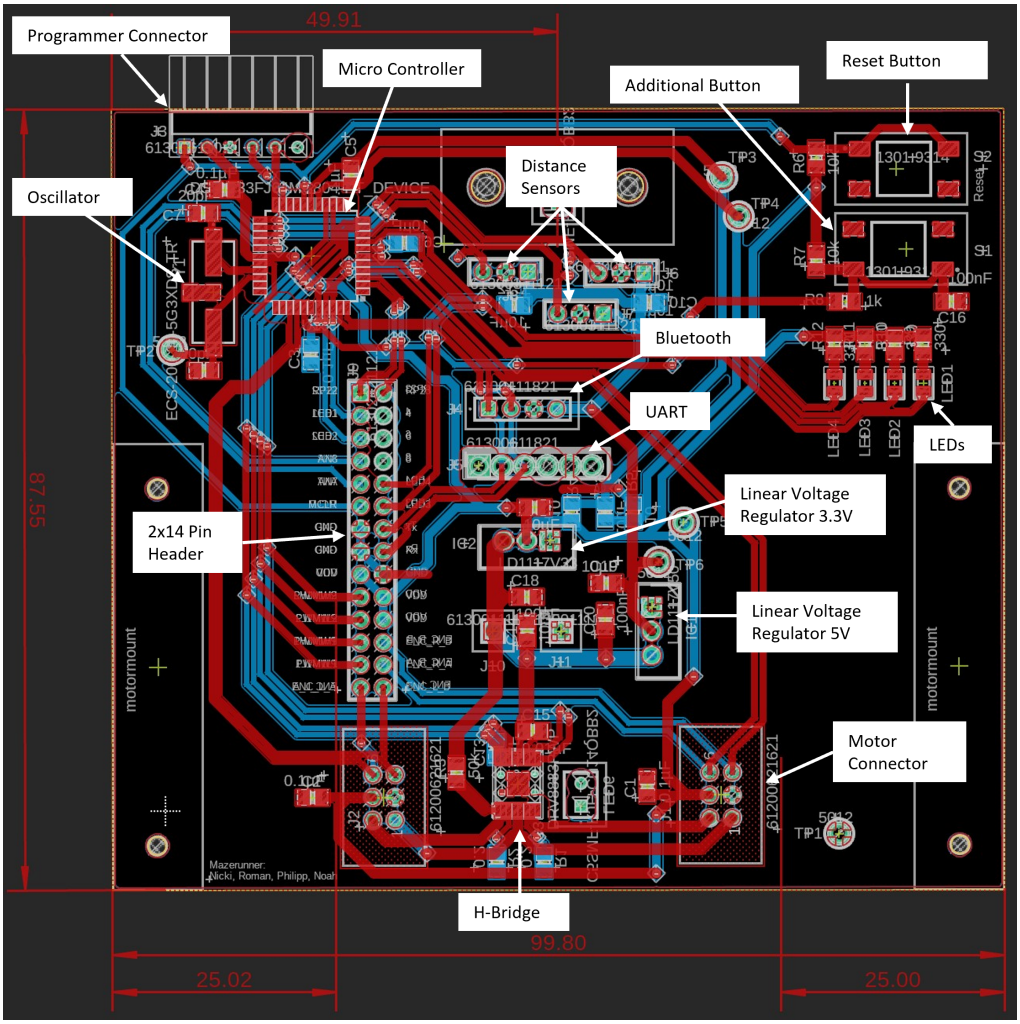


Figure 3.2.: Wiring of the PCB with the main components highlighted. The ground plane is hidden for better visibility.

3.2.1. Microcontroller, Programmer and Oscillator

The *dsPIC33FJ64MC804* microcontroller by *Microchip* is used [4]. The microcontroller is placed in the front left corner of the PCB, far away from the power supply to avoid electromagnetic interference (EMI) and relatively close to the distance sensor connectors and the 2x14 pin header to simplify the routing. The schematic overview of the microcontroller is shown in figure 3.3. All the voltage supplies are stabilized with decoupling capacitors of 0.1µF. The input for the internal voltage regulator of the microcontroller VCAP, which transforms the 3.3V supply voltage to the internal 2.5V, is connected to a 10µF capacitor as recommended in the data sheet. Close to the microcontroller, the oscillator and the programmer connector are placed. For the programmer connector, a 90° bent pin header is used (figure 3.5). The oscillator is a *520-ECS-200-20-5GXT* 20Mhz crystal as recommended in the microcontrollers data sheet with its input/output pins connected to 20pF capacitors (figure 3.4) which are placed on the same side of the PCB. Moreover, the oscillator is surrounded by a ground pour and no wires were placed under the oscillator as recommended.

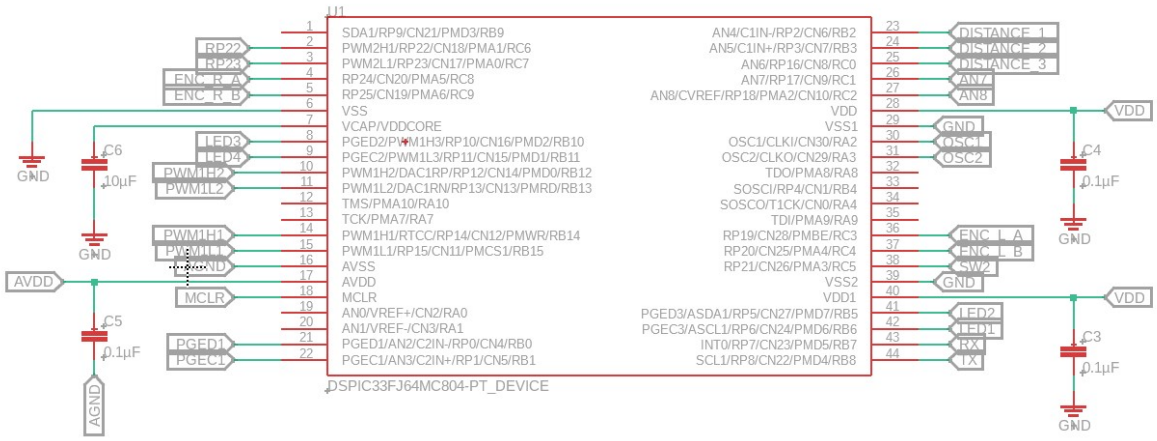


Figure 3.3.: Schematic overview of the microcontroller.

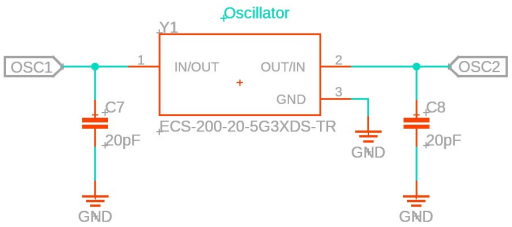


Figure 3.4.: Schematic overview of the oscillator.

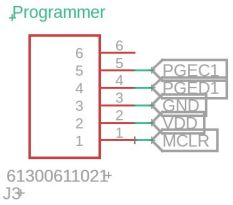


Figure 3.5.: Schematic overview of the programmer connector.

3.2.2. Power Supply

A 9V battery was selected as the power source. To supply all components, the 9V of the battery needs to be reduced to 5V (supply voltage of distance sensors) and 3.3V (supply voltage of microcontroller, encoders, Bluetooth module, LEDs), also denoted by V_{DD}. Despite the poorer efficiency, linear voltage regulators were preferred over switching voltage regulators to avoid unnecessary noise. We chose *LD111733* and *LD111750* voltage regulators from *STMicroelectronics* with a *TO-220* package [5]. For an approximate thermal calculation of the linear voltage regulators, we calculated the maximum drawn current by the components with a supply voltage of 3.3V and 5V in table 3.1 and 3.2. For each component, the maximum current was taken from the corresponding data sheets [6, 7, 8, 5, 4, 9]. Therefore, the calculated total current for each voltage regulator can be seen as an upper limit since the operating values are usually lower.

Table 3.1.: Total power drawn by components with 3.3V supply voltage

Component	max. current [mA]	amount	total current component [mA]
Controller	76	1	76
Encoder	15	2	30
LEDs	2	4	8
Regulator (quiescent current)	10	1	10
Bluetooth module	40	1	40
Total			**164**

Table 3.2.: Total power drawn by components with 5V supply voltage

Component	max. current [mA]	amount	total current component [mA]
Distance Sensor	22	3	66
Regulator (quiescent current)	10	1	10
Total			**76**

With a thermal resistance junction to ambient $R_{thJA} = 50°C/W$, an assumed ambient temperature $T_A = 50°C$, a total current $I_{\mathrm{max,3.3V}} = 164mA$, we obtain the following temperature of the junction T_J for the 3.3V regulator:

$$T_J = (9V - 3.3V) * \frac{I_{\mathrm{max,3.3V}}}{1000} * R_{thJA} + T_A = 96.74°C \tag{3.1}$$

For the 5V regulator, we obtain similarly with a total current $I_{\mathrm{max,5V}} = 76mA$:

$$T_J = (9V - 5V) * \frac{I_{\mathrm{max,5V}}}{1000} * R_{thJA} + T_A = 65.20°C \tag{3.2}$$

Given a maximum operating junction temperature of the voltage regulators of 125°C as well as the use of maximum values for calculation and the high assumed ambient temperature of 50°C, the chosen linear voltage regulators are sufficient. In the presented thermal calculation, each of the linear voltage regulators was used to directly convert the 9V battery voltage to 5V or 3.3V respectively. Another possibility would be to have the voltage regulators in a row such that the 3.3V voltage regulator only needs to convert 5V to 3.3V instead of 9V to 3.3V. This would lead to more power dissipation of the 5V regulator and less of the 3.3V regulator. However, since the calculated case above already proved to be sufficient, it was decided to use the voltage regulators this way and the thermal calculation for the other case was not carried out anymore. The schematic overview is depicted in figure 3.6. As proposed in the data sheet, decoupling capacitors of 10µF respectively 100nF were connected to the output and input. The voltage regulators were placed in the lower middle part of the PCB, close to the H-Bridge and far away from the microcontroller to avoid EMI and to allow for shorter high power connections.
To provide the microcontroller with the analog supply voltage AV_{DD}, the supply voltage V_{DD} is filtered with a low pass filter (see figure 3.7) consisting of two 10 Ω resistors and one 10µF capacitor. Later, we had to use 100 mΩ resistors instead of the intended 10 Ω resistors. The effect of using the 100 mΩ resistors is discussed in chapter 5.1.3.

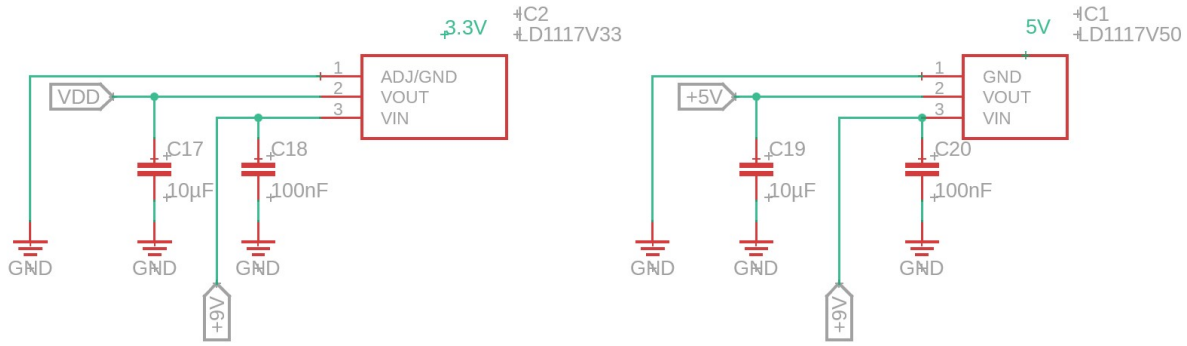


Figure 3.6.: Schematic overview of the 3.3V (left) and the 5V (right) linear voltage regulator.

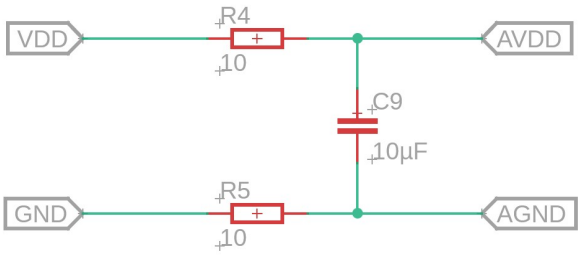


Figure 3.7.: Schematic overview of the AV_{DD} low-pass filter.

3.2.3. H-Bridge and Motor Connectors

The *DRV8833PWP* from *Texas Instruments* [10] was chosen as the H-Bridge to control the motors. It is a dual H-Bridge which simplifies the PCB Design as it can operate both motors. Furthermore, it has a maximum supply voltage of 10.8V, which is enough for the used 9V battery. To deliver the 6V needed by the motors, the duty cycle of the PWM is simply limited to a maximum of 66% in software. The *PWP* version of the *DRV8833* has a ground pad, allowing for a maximum peak current of 2A per motor. In our application, peak currents occur when the motors change direction. In this case, a motor has a maximum current of $I_{max} = 12V/8\,\Omega = 1.5A$, calculated by dividing two times the nominal voltage of the motor (6V) by the inner resistance ($8\,\Omega$) [8].
The schematics of the H-Bridge are shown in figure 3.8. While we placed the capacitors as given in the data sheet, the nSleep pin is pulled up with a $50\,k\Omega$ resistor such that the H-Bridge stays always active. Moreover, we decided not to use the nFault pin. The schematic shows two $0.2\,\Omega$ resistors connected to the AISEN and BISEN pins. These resistors were included to have the option of chopping the maximum current later, even though we decided to use zero-ohm resistors and not to chop the current in the end. The H-Bridges input pins (AIN1, AIN2 for the left motor and BIN1, BIN2 for the right motor) are connected to the PWM1 module of the microcontroller. The H-Bridge provides different driving modes for the motors (fast decay/slow decay). The modes can be switched by setting one of the input pins of a motor (e.g. AIN1 or AIN2 for the left motor) to low/high and controlling the other pin with PWM. By placing the respective pins all on PWM pins of the microcontroller, all modes of the H-Bridge can be used by setting the corresponding PWM output override bits. The output pins of the H-Bridge are connected to the corresponding pins of the motor connectors as shown in figure 3.9.
The encoders are also connected via the motor connectors. They are supplied with 3.3V (V_{DD}) stabilized with a decoupling capacitor of 0.1µF. The encoder channels are connected to remappable pins of the microcontroller.
The H-Bridge and the motor connectors are all placed in the lower/rear part of the PCB, close to the power supply.

3.2.4. UART and Bluetooth

For communication, we prepared the board with connectors for a *PmodUSBUART* module [11] and a HC-05 Bluetooth module [6]. The schematic overviews are shown in figure 3.10 and 3.11. The RX (receive) and TX (transmit) pins of the connectors were intentionally placed such that the RX pin of the microcontroller connects to the TX pin of the UART/Bluetooth module, and conversely, the TX pin of the microcontroller connects to the RX pin of the modules.

3.2.5. LEDs

We decided to have four LEDs for debugging purposes that are connected to remappable pins of the microcontroller (figure 3.12). Low power LEDs were chosen which operate at a forward

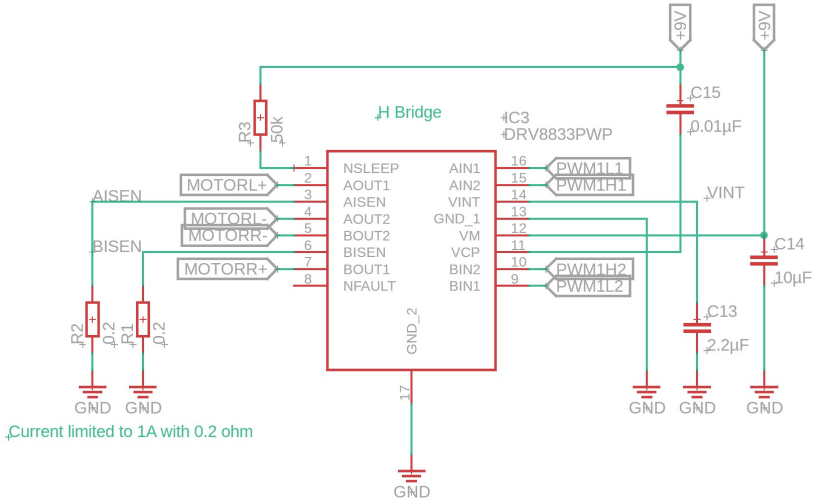


Figure 3.8.: Schematic overview of the H-Bridge

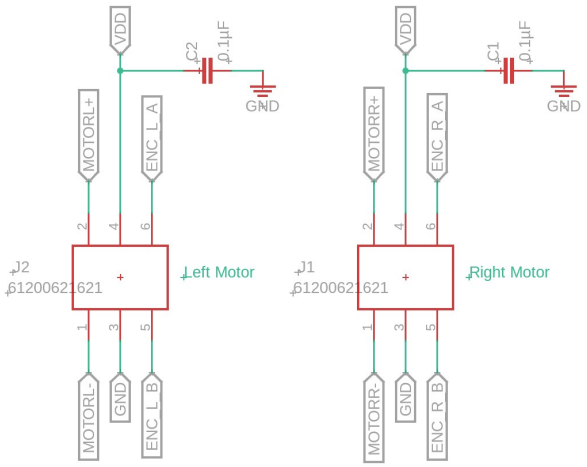


Figure 3.9.: Schematic overview of the motor connectors.

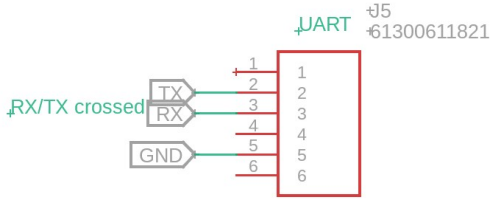


Figure 3.10.: Schematic overview of the UART connector.

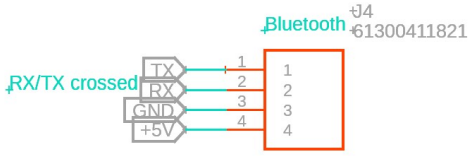


Figure 3.11.: Schematic overview of the Bluetooth connector.

current of $I_F = 2mA$ and a forward voltage $V_F = 2.65V$ [9]. Based on these specifications, the

needed resistance can be calculated:

$$R_{LED} = \frac{(V_{DD} - V_F)}{I_F} = \frac{(3.3V - 2.65V)}{0.002A} = 325\,\Omega \tag{3.3}$$

Given the calculated resistance, the next closest resistor from the E12 series was chosen which is a 330 Ω resistor.

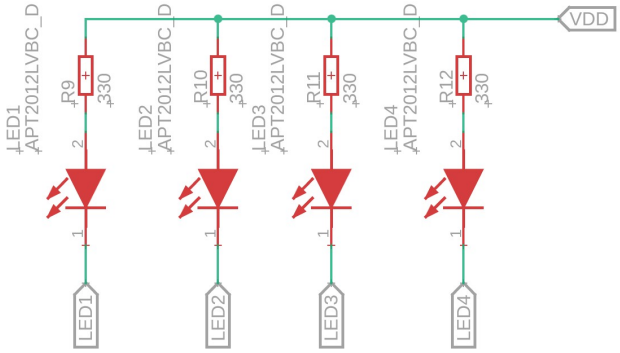


Figure 3.12.: Schematic overview of the debugging LEDs

3.2.6. Buttons

Two buttons are placed on the board, a reset button and an additional input button. For both buttons, we used SMD push buttons [12]. The reset button is connected to the MCLR pin of the microcontroller which is pulled up with a 10k resistor (figure 3.13). By pushing the button for reset, the MCLR pin is driven low. The input button is connected to a remappable pin of the microcontroller (figure 3.14). It is additionally debounced with a 100nF capacitor and a 1 kΩ resistor.

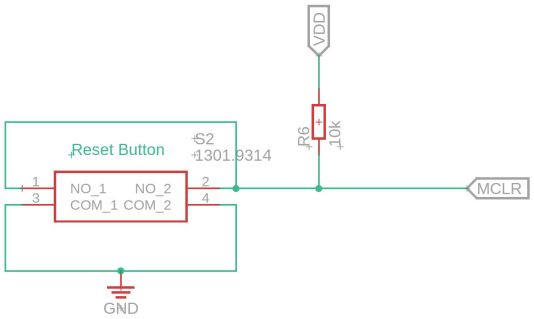


Figure 3.13.: Schematic overview of the reset button.

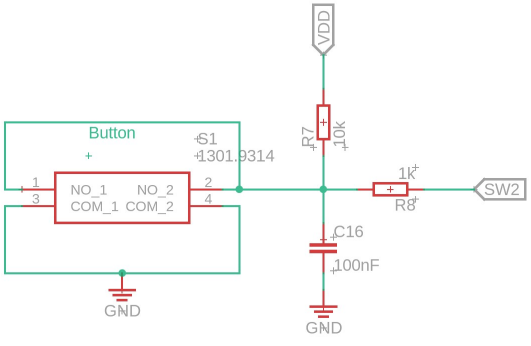


Figure 3.14.: Schematic overview of the input button.

3.2.7. Distance Sensor Connectors

For the used *Sharp GP2Y0A51SK0F* distance sensors [7], the board is prepared with pin holes which are soldered to wires connecting the sensors. The distance sensors are supplied with 5V. As recommended in the data sheet, the power supply is stabilized with a 10µF capacitor. The sensor outputs are connected analog pins of the microcontroller. The sensors are placed in the middle front part of the PCB.

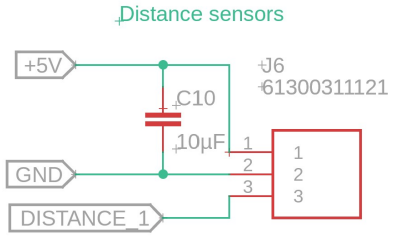


Figure 3.15.: Schematic overview of the pin holes for the distance sensors

3.2.8. Additional Pin Header and Test Pins

For debugging or repairing purposes, we additionally placed a 2x14 pin header on the board, offering the possibility to easy access important or additional pins such as PWM pins, encoder pins, unused remappable pins, etc. (figure 3.16). Moreover, we placed test pins for all voltage supplies and grounds (5V, VDD, AVDD, GND, AGND) as well as for the oscillator.

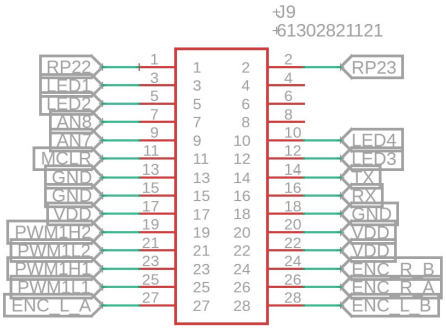


Figure 3.16.: Schematic overview of the additional pin header.

4. Software Design

In the following chapter the software deployed on the micromouse is discussed, starting from the base with the Hardware Abstraction Layer (HAL) and ending with the algorithm specific to the maze-solving challenge. This algorithm is implemented as an execution mode in the robot state machine, which determines the motion primitive executed by the control stack.

4.1. HAL

Most of the core code to configure the microcontroller was provided; therefore, the setup of the respective microcontroller modules will only be mentioned whenever it entails a notably change or impact, such as when increasing the PWM frequency to achieve silenced motor control. Accordingly, the provided table focuses mainly on a brief overview of the most important functionalities built upon the microcontroller modules and used by the discussed components.

Table 4.1.: Functionalities provided by the HAL

HAL Module	**Microcontroller Module**	**Provided Functionality**
HAL A	QEI	Calculated Velocity
		Calculated Distance Traveled
HAL B	PWM	Set Motor Mode
		Set PWM Duty Cycle
HAL C	ADC	Wall Detection
		Distance to Wall L/R
HAL D	Timer	Call Control Loop
HAL E	UART	Command Interface
		Logging
HAL F	GPIO	Toggle LED State
		Detect Button Press

4.2. Control Stack

With the hardware abstraction provided by the HAL, the robot can be driven by setting a certain duty cycle for the motor PWM and selecting a mode for each motor. However, for

precise control, which is needed when navigating an intricate maze, and robustness against outside influences, a form of velocity control becomes indispensable.

Therefore a cascaded control stack was developed that is composed of different controllers depending on the currently selected movement primitive type stored in the `movement_primitive` variable. Figure 4.1 illustrates this control stack with the three movement primitive types of `DRIVING_STRAIGHT`, `TURNING` and `PARKING` for the left wheel. The control stack is invoked and executed in fixed time intervals by the Timer1 interrupt service routine. During this invocation, the `DRIVING_STRAIGHT` and `TURNING` primitive generate a target velocity as the control signal (labled `target_velocity Left` in the figure below), whereas `PARKING` just commands a target velocity of zero. To calculate the resulting target velocity, the involved controllers access information provided by the HAL.

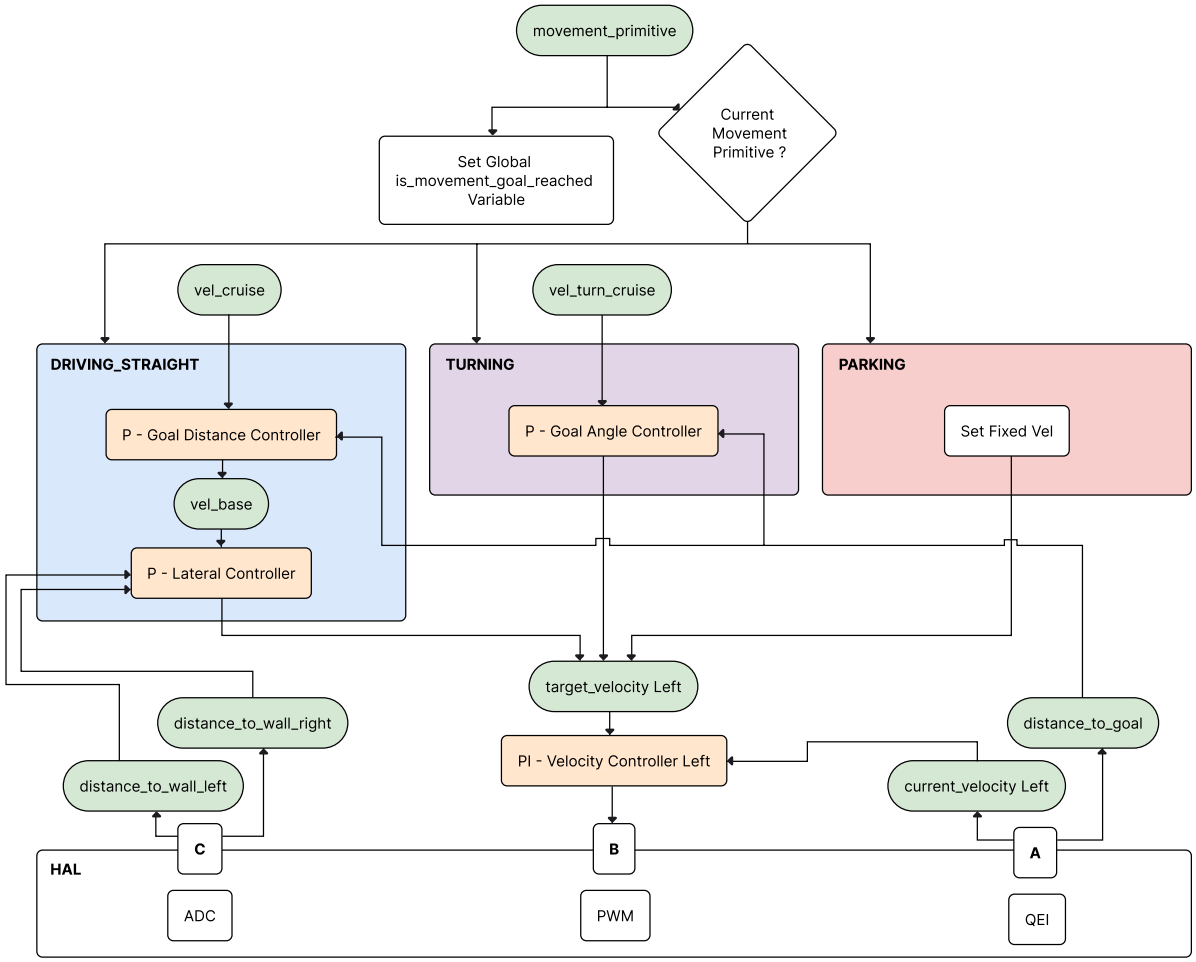


Figure 4.1.: Flow Chart of the Control Stack. Variables are marked in green, while the Controller Modules are marked in orange. In every control loop iteration, the selected motion primitive is executed and commands the target velocity for the respective wheel.

4.2.1. Angle and Distance Control

Both when executing a turn or driving straight, simple P controllers are deployed at first. They take a set of `vel_turn_cruise` and `vel_cruise` velocity, respectively, as input and calculate, based on the remaining distance to the specified goal, either in degrees or meters, the appropriate base velocity `vel_base`.

The primary function of these controllers is to approach the goal position with reduced velocity, thereby increasing positional precision. This objective is achieved through the proportional logic defined in Equation 4.1:

$$\text{ctr}(d, v, K_p) = \begin{cases} v & \text{if } \left|K_p \times d\right| \leq |v| \\ K_p \times d & \text{if } \left|K_p \times d\right| > |v|, K_p \times d > 0 \\ -K_p \times d & \text{if } \left|K_p \times d\right| > |v|, K_p \times d < 0 \end{cases} \tag{4.1}$$

In this equation, d represents the remaining distance to the goal, v is the current velocity, and K_p is the proportional constant. The logic demands the resulting velocity to fall below the input cruise velocity with a slope defined by K_p as the robot nears the goal and the remaining distance to the goal approaches zero.

The distance to the goal is provided by the encoder module of the HAL. The resulting course of the base velocity is depicted in Figure 4.2, which illustrates the relationship between velocity and distance to the goal with a K_p of 6.

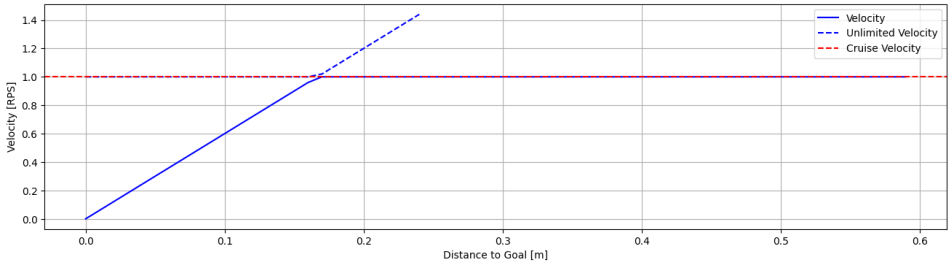


Figure 4.2.: Velocity vs Distance to Goal with $K_p = 6$ and `vel_cruise`$= 1$ *RPS* (rotations per second)

By converting the angle to the goal position into the distance a wheel has to travel on the arc of the resulting turning circle, the same distance controller can be deployed for both turning and moving straight. The wheel turning direction is managed by assigning each wheel the velocity with the appropriate sign, defining right turns as positive, and left turns as negative.

While this approach is sufficient to generate an adequate target velocity during a turn, an additional lateral controller is deployed when driving straight.

4.2.2. Lateral Control

In order to center the robot laterally within a cell, regardless of its starting position and orientation, a lateral controller is utilized, leveraging the presence of boundary walls within the maze. Initially, it needs to detect the availability of walls that could provide the necessary guidance. A wall is detected when the sensor voltage reading from the distance sensors exceeds a specific threshold (e.g. `WALL_DETECTION_THRESHOLD_SIDES`).

Since interpreting the distance sensor signals can be challenging during turning, this controller is only activated while the robot is driving straight. There are four different possible scenarios the robot can encounter while navigating through the maze, each requiring a distinct control strategy.

Walking from cell one to cell five in the maze scenario depicted in Figure 4.3, the robot passes through four different wall configurations that demand different lateral centering strategies. In each case, the controller adjusts the input base velocity, increasing or reducing it for each wheel separately, depending on whether the robot is too far from the center to the right or to the left.

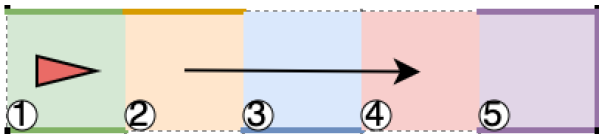


Figure 4.3.: Different Lateral Control Scenarios observed in the Maze.

In the optimal scenario where there is a wall on both sides of the robot (`TWO_WALL_CENTERING`), the base velocity is updated according to Formula 4.2:

$$\begin{aligned} \text{vel\_left} &= \text{vel\_base} - K_p \times (\text{distance\_right} - \text{distance\_left}) \\ \text{vel\_right} &= \text{vel\_base} + K_p \times (\text{distance\_right} - \text{distance\_left}) \end{aligned} \tag{4.2}$$

However, the distance mentioned refers to the raw distance sensor voltage reading as a percentage of the maximum value provided by the ADC HAL functionality. Therefore, when the robot is closer to the left wall, the voltage output by the left sensor increases as the reflected light intensity increases.

In scenarios where there is only a wall to the right or left, the robot will try to maintain a predefined distance from the respective wall. If there are no walls to stick to, the lateral controller will simply pass on the unchanged base velocity to the motor velocity controller.

4.2.3. Velocity Control

To guarantee that the robot reaches and maintains the target velocity set by the different motion primitive modules, the desired velocity is always passed to the PI velocity controller, as illustrated in Figure 4.1. A separate controller is present for each wheel, adhering to the PI control strategy. This strategy combines the swift responsiveness of a proportional term

(K_p) and a steady-state error-eliminating integral term (K_i) to adjust the appropriate motor PWM's duty cycle using the HAL.

Additionally, an anti-windup measure is implemented to prevent integration wind-up. This measure limits the maximum value the integral term can reach. Furthermore, a safety layer is in place to ensure that the duty cycle never exceeds 1.0.

$$\text{Motor PWM Duty Cycle} = K_p \times \text{vel\_error} + K_i \times \int \text{vel\_error}\ dt \tag{4.3}$$

These control properties will be examined further in the upcoming section discussing the selection of reasonable values for the K_p and K_i gains.

4.3. Robot State Machine

Moving to the highest level in the software stack, skipping the execution modes, the robot state machine is placed. This finite state machine manages general robot functionality and contains other state machines as sub-states, enabling a structured and organized representation of complex systems. Specifically, it encapsulates the execution mode that determines the movement primitive governing the control stack, as discussed in the previous section. The state machine is placed in the main execution loop of the microcontroller and is continuously iterated. Its execution is only interrupted by events such as the Timer1 interrupt to execute the control stack.

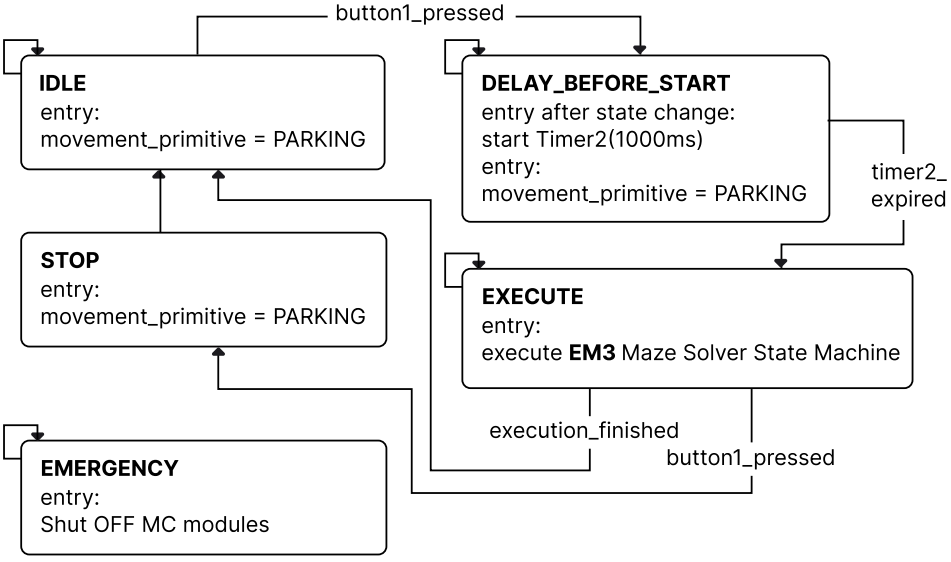


Figure 4.4.: Robot State Machine configured with the **EM3** Execution Mode to solve the maze.

When powering up the microcontroller or triggering an external reset via the Reset Button the robot is placed in the **IDLE** state of the robot state machine(as shown in Figure 4.4). This state remains active and sets the selected movement primitive to `PARKING` until Button1 is

pressed. Upon entering the **DELAY_BEFORE_START** state after the button press triggers a state change, Timer2 is started. The transition to the **EXECUTE** state occurs only after Timer2 has expired. This ensures a comfortable interaction with the robot to start the execution sequence without interfering with the distance sensor readings or blocking the robot's movement after the button press. In the **EXECUTE** state, the configured execution mode is carried out. Therefore, the motion primitive is no longer set by the robot state machine but by the selected execution mode. Besides the **EM3** maze solver state machine designed to complete the micromouse challenge, a left wall following mode (**EM1**) as well as a mode to test and execute primitive motions (**EM2**) were developed. Details on how the execution modes calculate the desired motion primitive will be shown for **EM3** in Section 4.4.2. To exit the **EXECUTE** state, one of two events must occur. Either the execution finishes, determined by the respective execution mode itself, and the robot returns to the **IDLE** state, or the execution is interrupted earlier by a button press, resulting in a transition to the intermediate **STOP** state before also ending in the **IDLE** state. An **EMERGENCY** state is planned but not yet implemented, which can be entered to ensure secure robot operation by shutting off the microcontroller's modules at risk.

4.4. Maze Solving

First a general implementation of the floodfill algorithm [3] to work with an open source micromouse maze simulator [13] is introduced, before presenting the final state machine derived from the simulator algorithm to integrate with the rest of the robots software stack in section 4.4.2.

4.4.1. Maze Solving with Floodfill Algorithm

To solve the maze and find the shortest path we implemented the floodfill algorithm. This algorithm sets the distance from every cell to the goal. The distance from one cell to another across the connecting edge is 1. In a maze without any walls this results in the distances for each cell as can be seen in 4.5 A.

Storing the maze

In order to perform the floodfill algorithm, it is necessary to know the layout of the maze (size of the maze and wall positions for each cell). Initially only the size of the maze is known, but the position of walls is unknown and the mouse has to explore the maze and build up the knowledge of the maze. To store the data we have created a data representation of the maze. It consists of an 2D array of structs. The size is *length x width* of the maze. Each struct consists of five values: four walls and the center. The walls can have the states *WALL, WAY, UNKNOWN* and the center *EXPLORED, UNEXPLORED*. In this representation each side of a cell is stored twice, but it allows much easier access to the information. The maze representation is initialized with all outer walls, as this is known per definition of the maze,

and all inner walls are set to unknown. Additionally we have a 2D *distance array* which stores the distance from each cell to the goal.

Floodfill Algorithm

We implemented the floodfill algorithm with two stacks: *current Level* and *next Level*. The goal cell is pushed on to the *current Level* stack and assigned a distance value of zero. All neighbouring cells with an open edge are pushed to the stack *next Level*. Now the stacks are switched, so *current Level* contains all points from *next Level* and *next Level* is empty again. All cells on stack *current Level* are assigned a value of one and all open neighbouring cells are pushed to *next Level*. This process is repeated and the distance increased each step, until all cells have been updated. The goal cell, from which the algorithm starts, can be set to the center and actually represent the goal of the maze or can be set anywhere and let the mouse find the shortest path from its position to the arbitrary position. The distance of each cell to the center of the maze, if there are no walls, is shown in 4.5 A. The updated distance of partially and completely explored mazes can be seen in 4.5 B and C. A pseudo-code of the floodfill algorithm is shown below (Algorithm 1).

Algorithm 1 Floodfill Algorithm

```
1: procedure FLOODFILL(currentLevel, nextLevel, distance[MAZE_SIZE][MAZE_SIZE],
   walls[MAZE_SIZE][MAZE_SIZE], goal)
2:     Initialize visited[MAZE_SIZE][MAZE_SIZE] ← 0
3:     Initialize current_distance ← 0
4:     Push goal cell onto currentLevel
5:     while currentLevel is not empty do
6:         for each cell in currentLevel do
7:             distance[cell.x][cell.y] ← current_distance
8:             visited[cell.x][cell.y] ← 1
9:             if open neighbors of [cell.x][cell.y] not visited then
10:                Push open neighboring cells onto nextLevel
11:            end if
12:        end for
13:        Swap currentLevel and nextLevel
14:        Clear nextLevel
15:        current_distance ← current_distance + 1
16:    end while
17: end procedure
```

Exploration Phase

The goal is to explore the maze just enough to find the shortest path from start to goal. The mouse is positioned at the starting cell (4.5 A: blue cell) and drives through the maze to the

goal. The next cell to move to is determined by comparing the distances of the connected neighbouring cells and choosing the one, which is one lower than the current distance. If the neighbouring cells have equal distance, the first one in clockwise direction is chosen, starting at the cell in front. After each movement to the next cell, the walls of the maze are updated and the floodfill algorithm is performed to update the distance values. Once the mouse reaches the goal, it turns around and walks back to the start (4.5 D: grey arrows). To find the fastest path from goal to start, the floodfill is now performed with the initial cell at the start position, which has then a distance of zero. The path from goal to start can be used to explore the maze even further and possibly find an even shorter path. In a 6x6 maze this is often not the case, but when testing the algorithm on larger mazes, such as a 16x16 it could be necessary to even set intermediary goals to further explore the maze.

Final Run

When the mouse has explored the maze sufficiently and returned to the start, it is time for a final run from start to goal on the shortest path. All unknown sides are defined as walls, to avoid walking there. The floodfill algorithm is performed only once in the beginning. Now the mouse simply walks from one cell to the next following the distances in decreasing order and arrives at the goal on the shortest path.

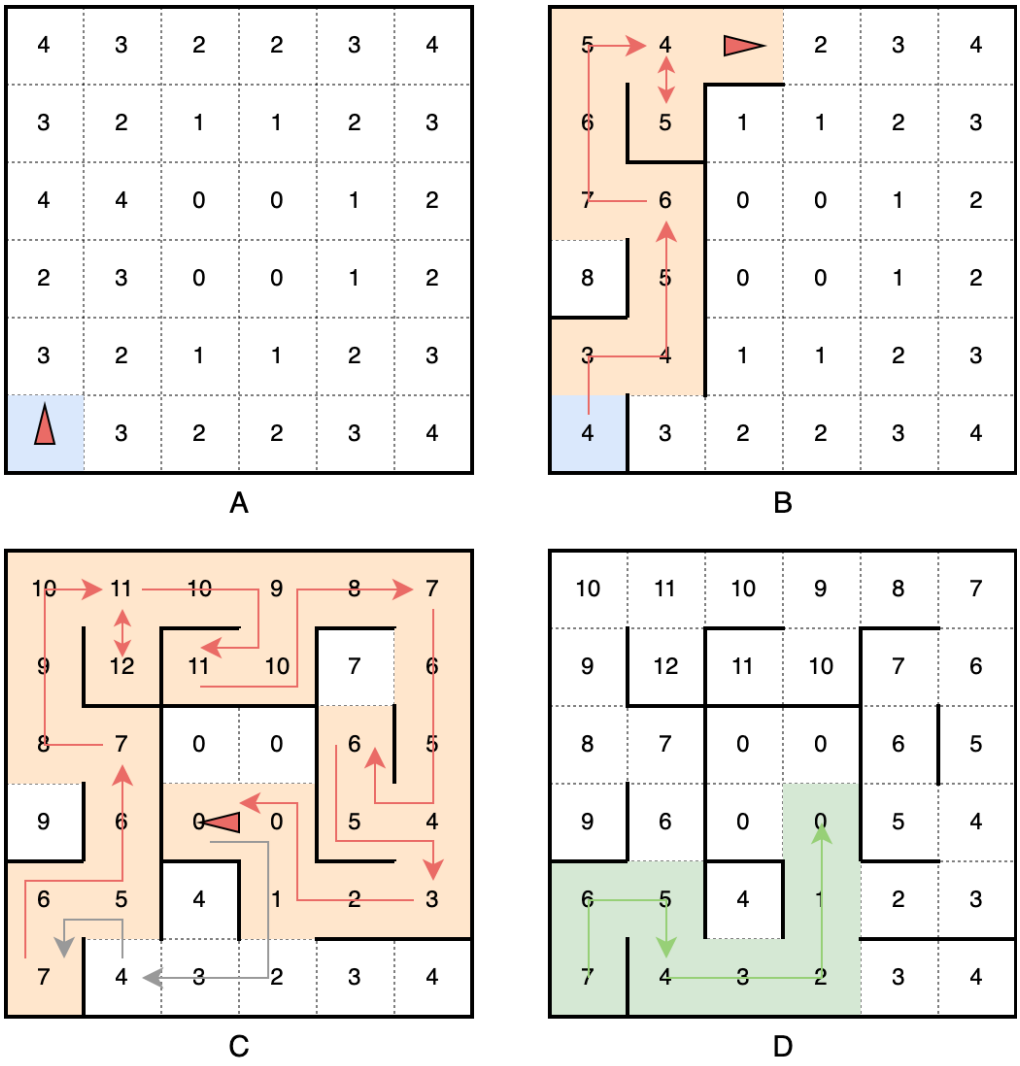


Figure 4.5.: A: Distance for every cell initialized without prior knowledge of the maze configuration. Mouse is positioned in the blue starting cell in the lower left corner B: Mouse in the middle of the exploration phase. C: Mouse has reached the goal in the center of the maze. The goal will now be set to the starting position and the mouse tries to find the shortest path from the center to the starting position by further exploring the maze (grey arrows). D: Mouse is at the start and now drives the shortest path from start to goal in the final run.

4.4.2. Maze Solver Execution Mode

In order for the described implementation of the floodfill algorithm above, developed and validated in simulation as detailed in section 5.10, to be deployed on the robot, it needs adaptation to work with the control stack and the robot state machine introduced earlier. Specifically, an execution mode that can be invoked by the **EXECUTE** state of the robot state

machine must be derived. This results in a state machine with the two introduced exploration phases and with the final run phase after the exploration is completed. Not depicted in Figure 4.6 is the maze initialization at the beginning of the execution mode. Due to the algorithm's cell-based nature, the movement logic (indicated in green) can be reused for every phase of the execution mode. In these states, the movement primitive type stored in the `movement_primitive` variable is set to the appropriate value, enabling the control stack to configure the controller accordingly. Furthermore, upon exiting the **FINAL_RUN** state after reaching the goal of the maze in the center, the `execution_finished` flag is set to true, triggering a state change in the robot state machine, as shown in Figure 4.4. This change leads to the determination of the execution mode.

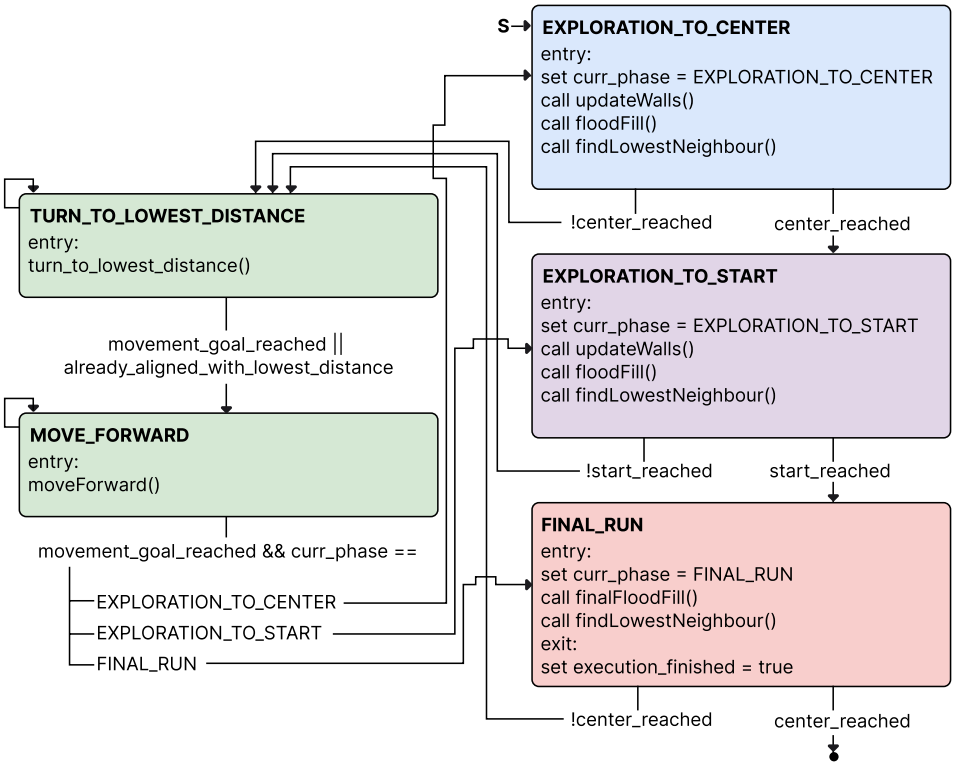


Figure 4.6.: Maze Solver Execution Mode implementing the Floodfill Algorithm

The modularity of the hierarchical state machines allows for easy modification or extension of the robot's capabilities. For instance, the execution mode can be extended to continue the exploration phase with new runs to the center or start until the shortest path lies within the explored area of the maze. This ensures finding the shortest path without necessarily having to scan the entire maze. This extension hasn't been implemented yet, as two runs have proven sufficient most of the time in the 6x6 maze provided in the laboratory.

4.5. Communication via UART and Bluetooth

The communication via the UART interface of the microcontroller is mostly used for debugging and testing purposes using the Bluetooth module (due to the cable-less convenience). Since both the UART-USB module as well as the Bluetooth module use the same UART interface, using one or the other does not entail changes to the software setup.

When serial communication is used for debugging, it is primarily employed to log data to a serial terminal for recording or interpretation purposes. An example is provided in Section 5.3 and Figure 5.9 where the internal maze representation of the robot gets printed to the UART. Expanding on the pure logging functionality, the simulation environment used to develop the maze-solving algorithm can also serve as a digital twin of the real-world robot. This is achieved by providing the simulator with the current actions and observations of the robot in the actual maze. To accomplish this, an intermediate Python application attempts to establish a serial connection with the robot's Bluetooth module. It then forwards any received commands to the simulator after ensuring that the commands match the accepted API calls of the simulator."

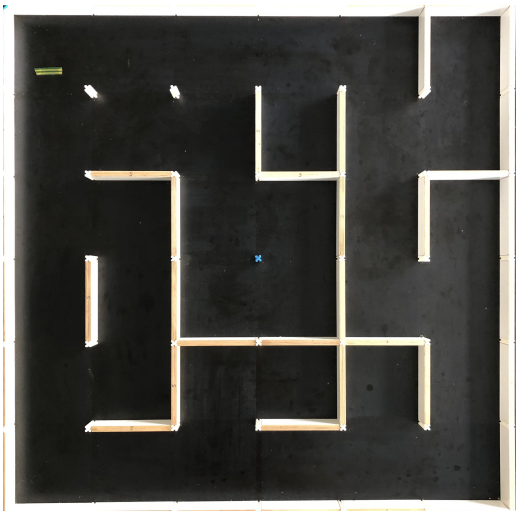


Figure 4.7.: Real World Maze (without the robot).

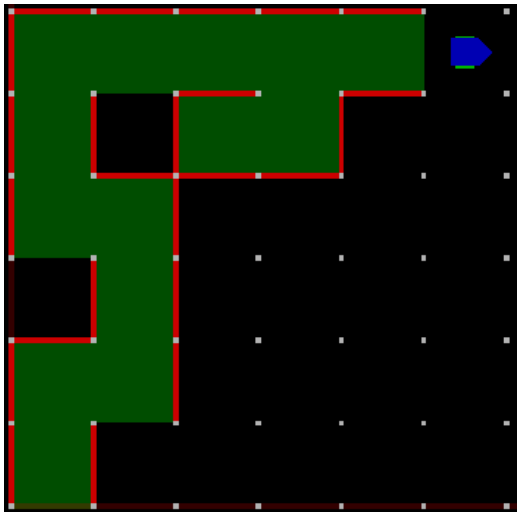


Figure 4.8.: Digital Twin in the Simulator.

Equally important is the ability to change or set certain parameters or variables from an external serial terminal while the microcontroller is running. This enables quick iteration when, for example, testing different controller configurations since no reprogramming of the robot is required; the changes can be applied over the air. The developed command parser monitors the incoming UART traffic for any of the valid commands to control the robot's motion (Table 4.3) or to tune certain parameters (4.2). For a command to be recognized, it must be encapsulated by the defined command marking characters '<' and '>'.

Table 4.2.: Parameter Tuning Commands

Command	Description
<set_kpX.Y>	Set the proportional gain parameter for the velocity controller.
<set_kiX.Y>	Set the integral gain parameter for the velocity controller.

Table 4.3.: Robot Motion Control Commands

Command	Description
<velX.Y>	Control the robot's forward or backward velocity.
<turnX.Y>	Set the angle the robot will turn.
<driveX.Y>	Set the distance the robot will drive.
<stop>	Halt the robot's motion.
<set_turn_angle_leftX.Y>	Set the default left turn angle of the robot.
<set_turn_angle_rightX.Y>	Set the default right turn angle of the robot.
<set_vel_straightX.Y>	Set the straight cruise velocity of the robot.
<set_vel_turnX.Y>	Set the turn cruise velocity of the robot.

5. Implementation and Testing

When building a micromouse, thorough testing of various components and systems is essential to ensure that the robot functions correctly, navigates the maze effectively, and performs well in competitions. This chapter will be divided into hardware tests and software tests which includes testing the control and navigation algorithms.

5.1. Hardware Testing

5.1.1. Distance Sensors

The IR distance sensors are tested to verify their accuracy and reliability. The tests are used to ensure that the sensors provide correct readings for obstacle detection, line following, and maze mapping. According to the manufacturer, the GP2Y0A51SK0F infrared distance sensors by Sharp are suitable for measuring distances between 2 and 15 cm. For the test, all three sensors were placed at the same known distance from a flat white wooden board, which is the same material that is also used for the walls of the maze. The test setup is depicted in figure 5.1. The sensors were placed at distances of 2 to 10 cm with a step size of 1 cm from the wall. Ten data points of raw sensor output were recorded for each distance with UART. Results are provided in table 5.1. The provided sensor values are the read values after the analog-to-digital conversion. For each distance and sensor, the respective mean over the ten measurements is calculated. The maximum deviation is calculated with respect to the mean values. The percentage deviation is calculated with the maximum deviation with respect to the overall mean of all sensor values per distance . We tried our best to place the sensors parallel and with the exact same distance to the wall. Assuming no errors caused by our test setup, we observe deviations up to nearly 8% percent and a bias of the right sensor compared to the others to sense a smaller distance. The bias of the right sensor would for example lead to a small offset from the center when driving between two walls. Even though there is the stated deviation of up to nearly 8%, we still decided to continue without further calibration but being aware of it. Eventually, it did not turn out to be an issue in practical tests and the sensor values were not further processed to enhance the quality of the measurements.

5.1.2. Motors and Encoders

To ensure the correct function of the motors and encoders, we did the following sanity check: We programmed a LED to time 10s and recorded a slow-motion video of the spinning wheel of one of the motors. In parallel, we read the encoder sensor data with UART. We supplied the motors with a mean PWM voltage of 3V and compared the counted rotations

Table 5.1.: Sensor test data summary. Sensor values are the read values after analog-to-digital conversion.

Distance [cm]	Sensor	Mean	Max. Deviation	Deviation from Mean of all three Sensors (%)
	Left	56.47	1.39	2.48%
2	Front	55.65		
	Right	56.36		
	Left	40.37	1.25	3.11%
3	Front	40.44		
	Right	39.70		
	Left	32.67	0.98	3.01%
4	Front	32.78		
	Right	32.09		
	Left	27.47	1.66	6.12%
5	Front	27.32		
	Right	26.64		
	Left	23.36	1.20	5.20%
6	Front	23.26		
	Right	22.61		
	Left	20.56	1.20	5.96%
7	Front	20.27		
	Right	19.55		
	Left	18.31	1.20	6.64%
8	Front	18.19		
	Right	17.70		
	Left	16.48	1.24	7.61%
9	Front	16.52		
	Right	15.86		
	Left	14.84	0.93	6.36%
10	Front	14.82		
	Right	14.20		

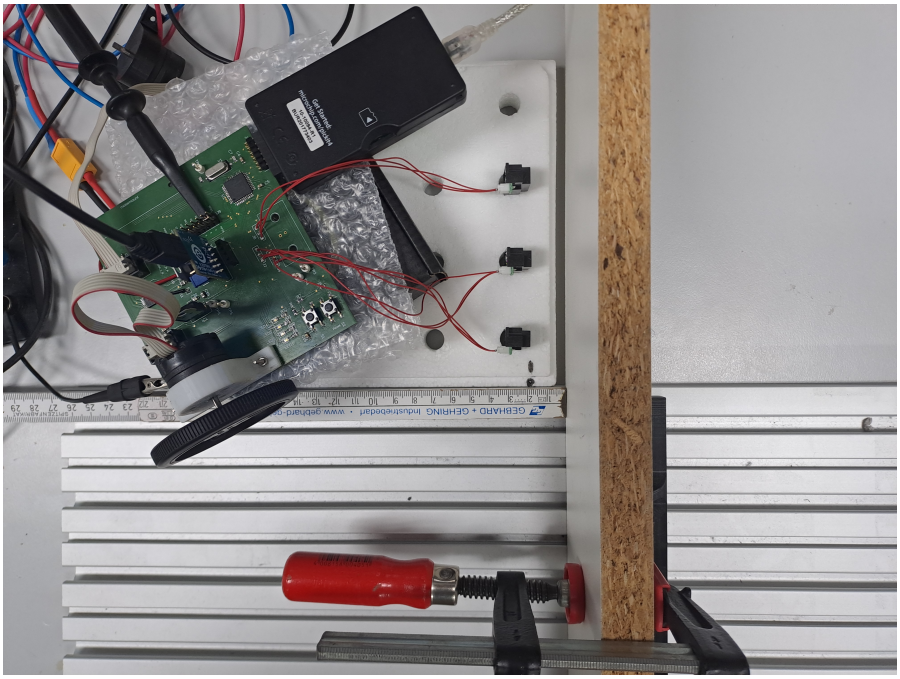


Figure 5.1.: Setup to test the distance sensors.

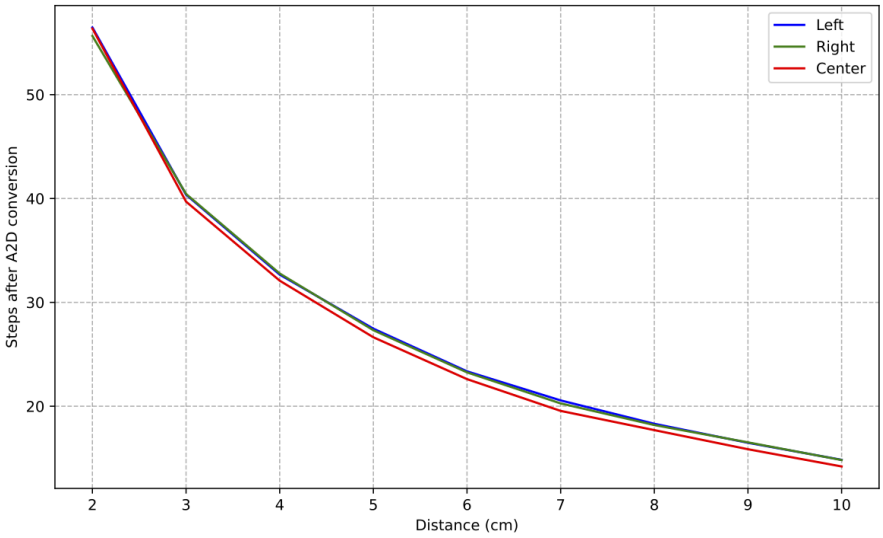


Figure 5.2.: Digital sensor measurements in percent of the 3.3V AVDD Reference Voltage, obtained from the test setup in 5.1 at different distances.

in the video with the obtained encoder readings. In the 10s, 17 rotations were counted, resulting in 1.7 rotations/s. Since the motor velocity is constant over the 10s, we took ten encoder measurement samples and calculated the mean. The ten sample measurements by the encoders (1.42, 1.42, 1.89, 1.89, 1.89, 1.89, 1.89, 1.89, 1.89, 1.89 in rotations/s) yield a mean

of 1.796 rotations/s. Finally, those values were compared with the motor velocity obtained by the speed constant of the motor's data sheet [8] which is 1130 rpm/V taking into account the gear ration of 33:

$$motor\ velocity = \frac{\frac{1130\ rpm/V}{60} * 3V}{33} = 1.71\ rotations/s \tag{5.1}$$

Comparing, the different three results (video, encoders, speed constant from data sheet), we concluded that everything is working as supposed. A reason for the deviation of the encoder data is the encoder resolution and the sample time (1ms) of our timer function with which we conducted our tests, leading to high steps within the discrete encoder signals. Later we increased the sample time to 10ms to obtain less erratic encoder signals.
Apart from the tests described above, we tested the different modes of the H-Bridge, namely:

- Slow decay, forward
- Slow decay, reverse
- Fast decay, forward
- Fast decay, reverse

The modes refer to how the H-Bridge handles the recirculation current of the motors when the drive current is interrupted by the PWM [10]. We discovered that in slow decay mode, the H-Bridge inverts the provided PWM-Signal. To account for this, we inverted the PWM signal in the code such that the produced output for the motors by the H-Bridge is as desired. In fast decay, the inversion is not necessary. Testing the different modes, a high noise level and power consumption when using slow decay was observed compared to fast decay mode. This is why we initially chose to use fast decay. We could also not observe a negative effect concerning the different behaviors of the modes (slow decay corresponds to braking, fast decay to coasting) when driving the micromouse. Later, the motor noise was further reduced by increasing the PWM frequency.

5.1.3. AVDD Filter

As stated in chapter 3.2.2, we planned to provide a filtered analog supply voltage AV_{DD} with a low-pass filter consisting of 10 Ω resistors and a 10µF capacitor. However, we accidentally ordered 100 mΩ resistors which we used instead, resulting in a much reduced filtering capability respectively higher cut-off frequency of the filter. To examine the effect of the reduced filter, we measured the analog supply voltage after filtering with our 100 mΩ filter on our board and compared it with the filtered analog supply voltage of the filter on the *Microchip* Development Board which has the same intended 10 Ω filter. The comparison of the measurements taken with the oscilloscope in the laboratory are depicted in figure 5.3. The observed voltage curves are very similar and show the same peak-to-peak voltage of 80mV. It has to be taken into account though that these measurements were taken under ideal conditions with the laboratory power supply. Nevertheless, given the good results, we decided to move on with the 100 mΩ filter since we were not expecting great negative effects.

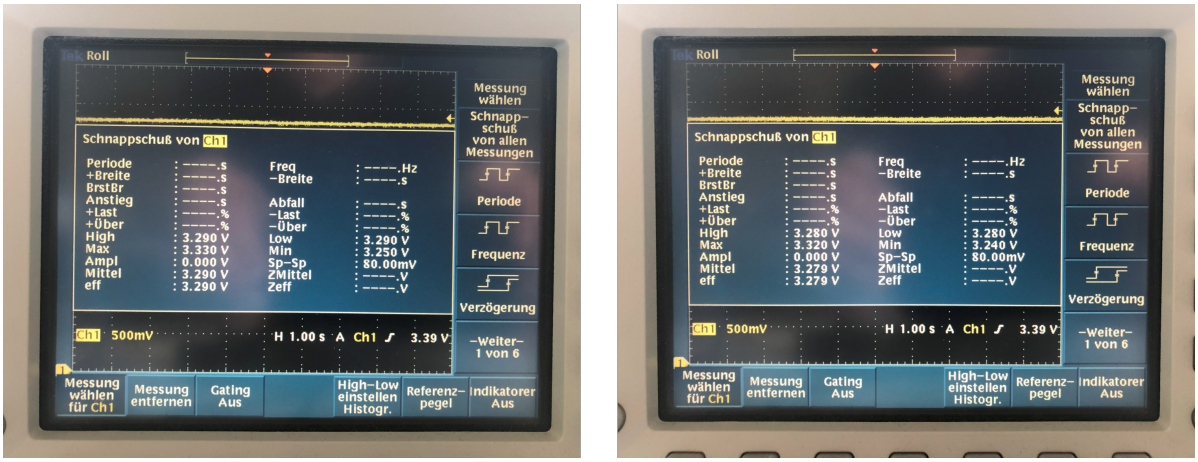


Figure 5.3.: Measured analog supply voltage with oscilloscope of 10 Ω filter on the development board (left) and of the 100 mΩ filter on our board (right).

5.1.4. Bluetooth Module

When using the Bluetooth module for debugging high-frequency applications like the controller stack, the rate at which data can be logged to a serial terminal becomes crucial. Therefore, configuring the HC-05 Bluetooth module to operate at a higher baud rate of 115200 is essential. This can be achieved by pulling the Reset Pin (PIO11) low while powering the device on, entering the command state. In this state, desired parameters such as the baud rate and a new module name (e.g., "*MazerunnerR*") can be set by issuing AT commands.

5.1.5. Power Supply

For the finalized runs we used 9V block batteries to power the motors. These proved to be okay when using a completely charged battery, but after only a few runs the power output decreased leading to errors like: slower movements, stops in the movement, Bluetooth UART communication failing. For most of the testing we used the laboratory power supply with cables, which at least provided steady power supply, but the moving mouse with cables attached had to be carefully watched to avoid failures by too short cables, cables blocking the distance sensors etc. and we repeatedly had to solder the power connectors on the PCB board, because of the mechanical stress. For a next iteration we would most likely change the linear voltage regulators to allow the usage of LiPo accumulators or similar power supplies frequently used in RC vehicles. They should provide a more stable power supply and allow recharging.

5.2. Controller Tuning

To select appropriate controller values, the entire control stack, along with all its components, must be deployed and tested on the robot. During testing, it became apparent that the goal

distance and goal angle controllers are not necessarily required to precisely turn or drive for a specified distance. This is because the robot's mass is small enough and/or the motor friction significant enough to prevent overshooting when stopping the robot after reaching the distance goal, even without reducing the velocity beforehand. This could, however, change for increased cruise velocities. An additional benefit of excluding the active velocity reduction is that the robot's trajectory becomes smoother when following a path with multiple intermediate goals, such as the cell-based navigation used to navigate through the maze. Therefore, further analysis of the angle and distance controllers will be omitted.

5.2.1. Lateral Controller - Wall Detection Threshold

When designing the lateral controller, the most crucial value to identify is the threshold that determines whether a sensor voltage reading from the distance sensors is interpreted as a detected wall or not. Otherwise, selecting the wrong control mode, which is based on the available walls as described in Section 4.2.2, can lead to a misguided robot and, in the worst case, its collision with the maze structure.

To determine this `WALL_DETECTION_THRESHOLD_SIDES`, the robot was placed within a cell, positioned as close as possible to either the left or right wall laterally. Then, the sensor reading transmitted via Bluetooth was established as the searched threshold. This approach enables the robot to detect a side wall even when maximally misplaced within a cell.

In the lateral control mode where the robot follows only one wall to its left or right, an additional distance value needs to be determined. This value guides the lateral controller to center the robot within the cell. It can be obtained by placing the robot in a maze cell and reading the respective sensor values for left and right.

Table 5.2.: Distance Sensor Constants

Constant	**Value [%]**
WALL_DETECTION_THRESHOLD_SIDES	22.0
SENSOR_VALUE_LEFT_WHEN_IN_CELL_CENTER	27.0
SENSOR_VALUE_RIGHT_WHEN_IN_CELL_CENTER	26.0

To validate the configured lateral control stack, the robot was driven along a configuration of different wall scenarios. A section of the recorded sensor values and the current lateral control mode along the driven distance can be seen in Figure 5.4. The lateral control mode is selected correctly, and the distance reading to the right wall remains relatively constant between the different control modes. The K_p value of all the lateral control modes was kept low enough to prevent the robot from twitching with every minor sensor change, yet high enough to ensure fast reactivity.

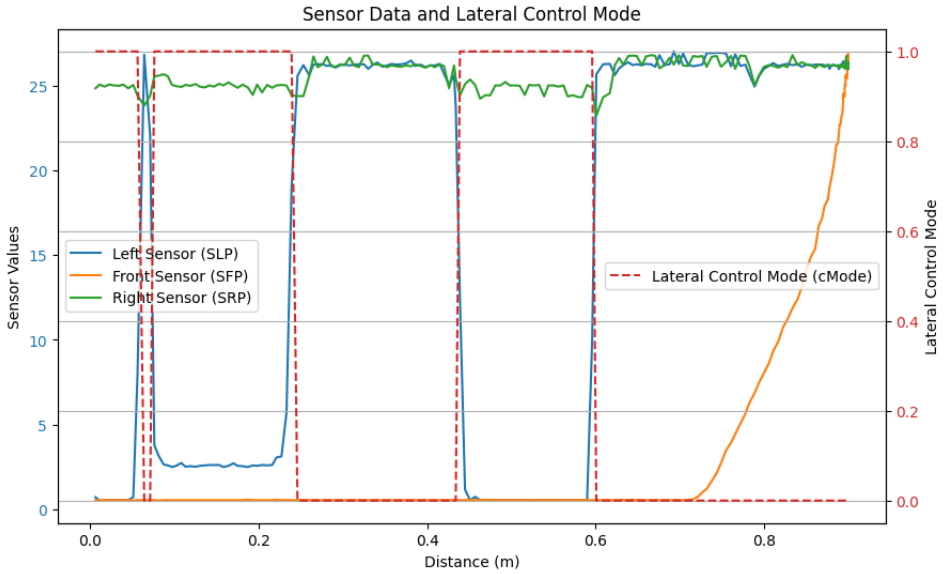


Figure 5.4.: Wall Detection Threshold Determination with the sensor values in percent. A small distance between the distance sensor and the reflecting wall results in large sensor readings of the detected intensity. (lateral control mode: 0=`TWO_WALL_CENTERING`, 1=`ONE_WALL_FOLLOWING_RIGHT`)

5.2.2. Velocity Controller Tuning

Tuning the PI velocity controller ensures that the robot responds accurately and swiftly to the provided velocity commands, ensuring stability and precision in its movements. In addition to the trial-and-error approach of testing various combinations of controller gains and selecting the most promising ones, a step response analysis is conducted. A step amplitude of 1 rotation per second (RPS) is used to observe the controller's varying ability to track the desired velocity while modifying the controller gains. The analysis was performed under different conditions: first, with the wheels suspended in the air and motors running freely, and second, under load with the robot actually driving on the floor. The tests conducted under load were deemed most relevant and will therefore be presented in the following section.

When selecting the proportional gain K_p the goal is to choose a moderate value that preserves the controller's sensitivity but does not lead to overshooting or cause the system to oscillate, unless absolutely necessary. From the tested values (Figure 5.5) the step response for the K_p value of 0.7 was determined to strike a convincing compromise between the described considerations, since the velocity only rises negligible slower compared to larger values that already cause overshooting.

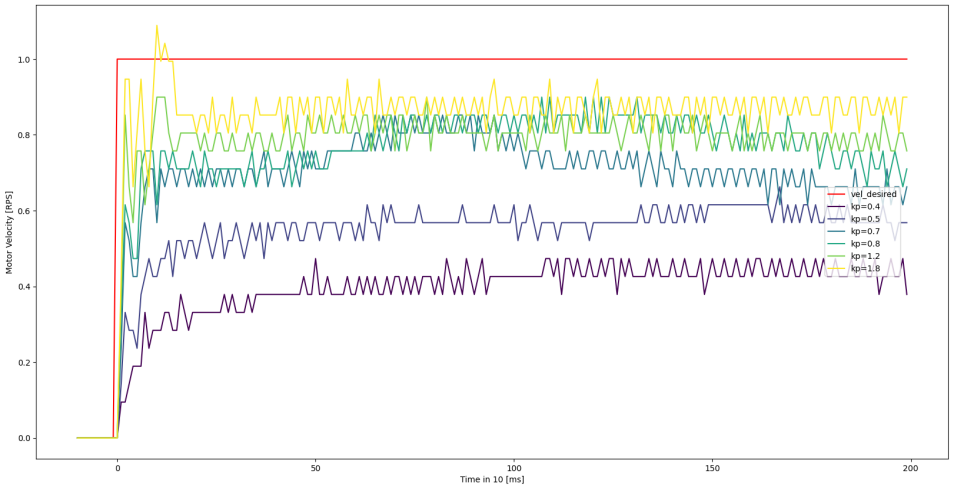


Figure 5.5.: K_p Selection with $K_i = 0$ for the left wheel.

However Figure 5.5 also demonstrates that a steady-state error remains for all of the tested values. Therefore, an appropriate value for the integral gain K_i needs to be determined to eliminate any residual error. Again, by comparing the plotted step responses (5.6) results in the selection of a K_i value of 0.08. This value is sufficient to remove the steady-state error reasonably fast without causing significant overshooting.

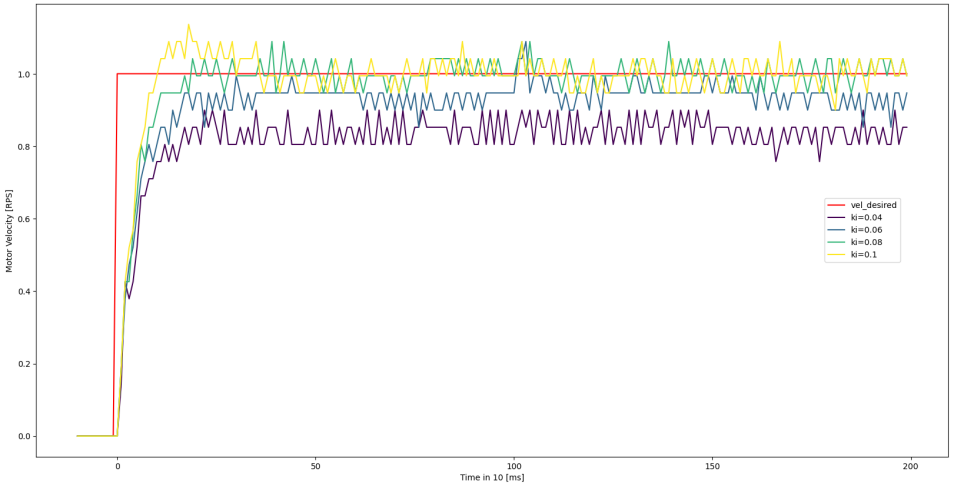


Figure 5.6.: K_i Selection with $K_p = 0.7$ for the left wheel.

To prevent any windup if a setpoint value is requested at which the limited manipulated variable goes into saturation, the maximum value of the integral is also limited. This results in the controller configuration of Table 5.3.

Table 5.3.: PI Velocity Controller Configuration

Parameter	Value
K_p (Proportional Gain)	0.7
K_i (Integral Gain)	0.08
Max Integral	+5
Min Integral	-5

In order to test the controllers performance when

5.3. Maze Solving Testing

We started with a simple wall following algorithm. This allows to first test the correct detection of walls and making the correct decisions. A data structure to store the maze is not necessary and simplifies the testing a lot. After we managed this successfully, we implemented the Floodfill Algorithm. We first developed the maze floodfill algorithm in a simulator [13]. This allowed us to easily test and verify the functionality of the algorithm before actually using it on the mouse. The simulator has an API with commands like *isWall, turn, moveForward*. We used these commands in the simulator to implement the algorithm and movements of the sim-mouse. It also provided us with the possibility to test the algorithm on larger and more complex mazes which could not have been done in the lab and allowed us to detect all edge cases and therefore make it robust. Once the mouse was able to execute and process the same API commands as the mouse in the simulation we were able to translate the simulation tested maze algorithm to a state machine representation on the real mouse. As the algorithm uses dynamically allocated memory in its stacks, we had to allocate 1000 bits as heap size in the IDE, otherwise malloc would have only result in the null pointer and every stack operation in resetting the chip. This took us some time to figure out and eventually it got solved by Dr. Lenz who was aware of this problem and gave us the hint to configure the heap size. However an even better solution would be to define the stacks as normal arrays, since we never actually free the memory. Therefore dynamically allocated memory presents no advantage. We had to adapt the previously mentioned functions for wall detection and movement as provided by the simulator with our own that actually control the real mouse. After some further debugging, which just showed that we forgot to include a function, the algorithm worked on the mouse as well and let us find the shortest path to the goal.

To overcome any complications during the implementation of the algorithm on the mouse two debugging approaches were developed. In the first we print the whole maze and its exploration state over UART to detect errors in its exploration. This can be seen in 5.7, 5.9 and 5.8. The maze consists of 6 x 6 cells, the center of each cell is either "UU" for *UNEXPLORED* or a number for the distance. Initially the sides are *UNKNOWN* and marked with "x" or "xx", but once explored they are either set to "–" or "|" for a wall or to "*" if there is a way. The maze shown in 5.9 is the same as shown in 4.5 D. For the second approach we wanted to return the position of the mouse and walls as commands which we would feed into the

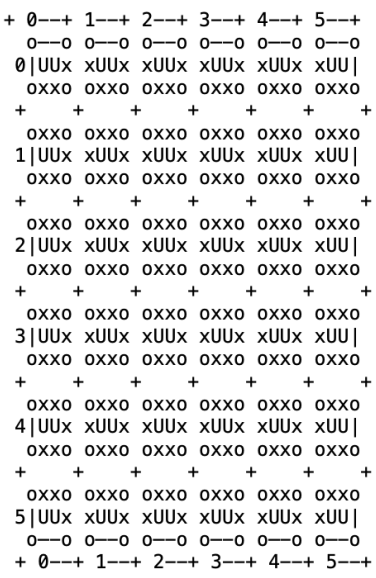


Figure 5.7.: Unexplored maze printed

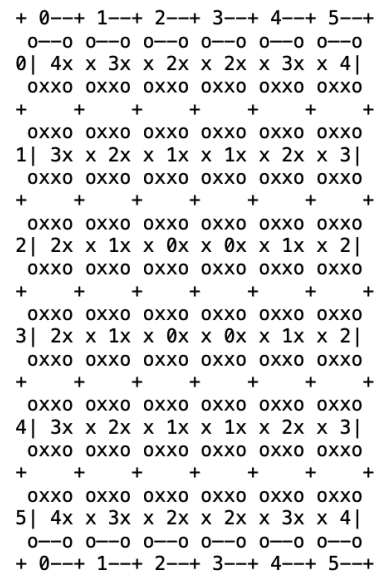


Figure 5.8.: Unexplored with distance

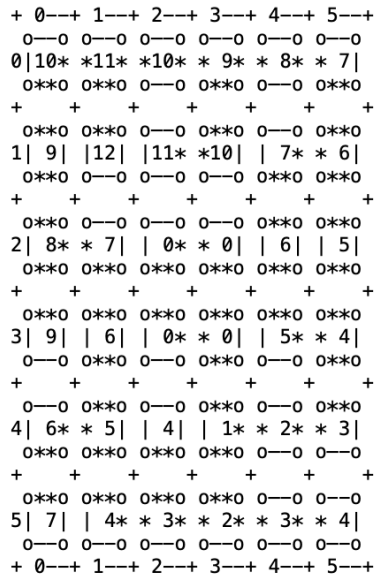


Figure 5.9.: Fully explored maze

simulator so we could follow the mouse on the laptop and visualize where it runs and which walls it sets. The commands were already provided in the simulator API. 5.10 shows the maze layout of the simulator.

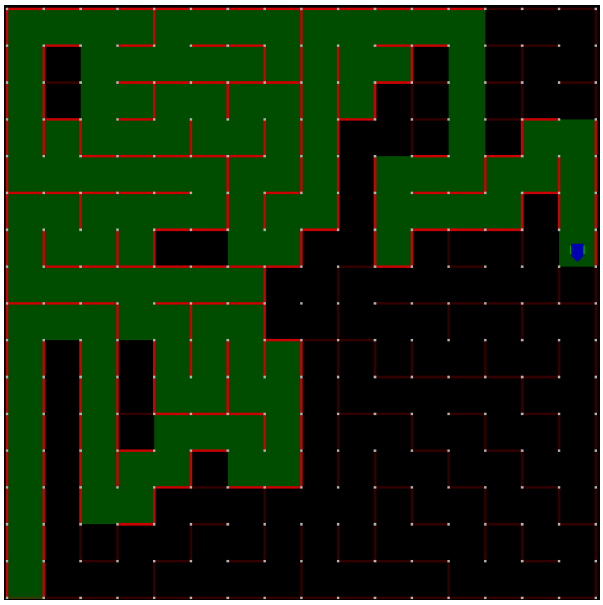


Figure 5.10.: The mouse walks through the unexplored maze and sets a wall everywhere it detects a wall. Each cell it has visited is marked with green.

6. Conclusion

Our journey of designing a micromouse racing robot from scratch has been both challenging and rewarding. Using the combined skills of all members of our team, we worked hard and learned a lot along the way to create a robot capable of navigating and solving mazes with satisfying precision and speed. This project has not only honed our technical skills but also fostered teamwork, problem-solving, and creativity.

We had initial dreams about solving the basic tasks relatively quickly and having time to focus on optimizing the path and speed to potentially have a crack on a low regional competition. This became very distant as we learned what was meant by "... designing from scratch". What seemed like an overwhelming task in the beginning was broken down into smaller sections that were accomplished step by step. Being guided by the course outline, we managed to take on every task and learn while accomplishing it. We started with the hardware design, moving over to soldering. Once finished we could actually test if its working and developed the software from a low hardware access level up to high level navigation tasks and maze solving. We started developing on both ends and defined a theoretical meeting point in the middle to connect the tasks. This division allowed us to work simultaneously on different parts, but for troubleshooting and testing we all came together. In the end everyone in our team had a good understanding of the whole codebase and was a specialist on a specific topic.

Generally, the hardware worked without facing any significant problems. The placement of the buttons next to the distance sensors should be reconsidered, as pressing the button triggered unwanted wheel movement during the earlier stages of software development, i.e. before the state machine was introduced. Also placing the distance sensors very close together caused some unwanted interactions between sensor signals which could have been avoided by placing the lateral sensors further away from the center. This problem was easily fixed by redesigning the sensor mount.

The biggest problem we encountered was the power supply by battery. The rapid depletion of the battery led to undesired resets and transmission issues with the Bluetooth module, which is problem that would need to be solved in another iteration of the robot.

Once all the parts came together and all problems were fixed, we had reached our goal of creating the fully functional micromouse *mazerunner* from scratch.

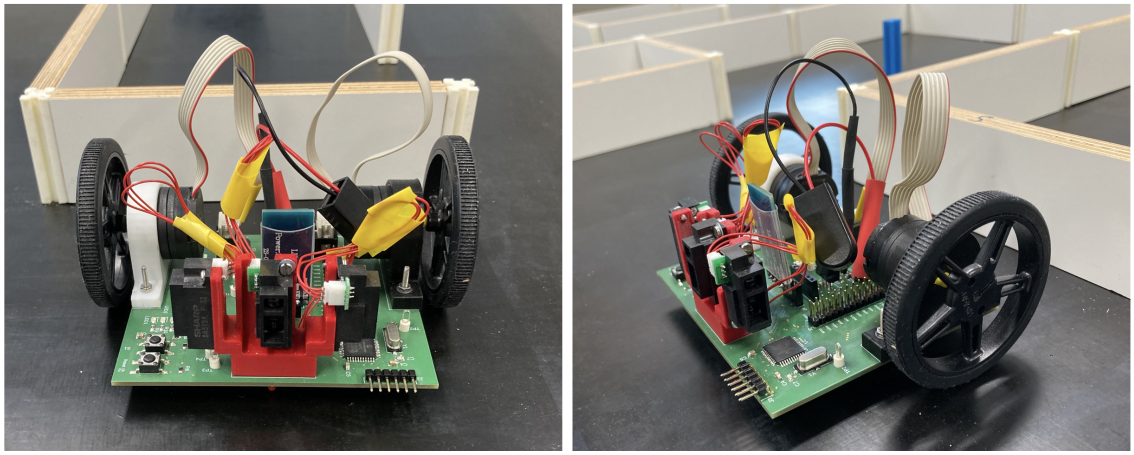


Figure 6.1.: Photographs of finished micromouse *mazerunner*.

Mazerunner is able to explore the maze thoroughly and drive the shortest path. In a next iteration a focus could be set on optimizing the *final run* by decreasing the runtime. It would involve smaller mouse dimensions to improve agility and maneuverability, enabling the mouse to drive around corners more smoothly. The path for the *final run* could be pre-calculated. Then the movement and speed to drive this path as fast as possible could be determined, e.g. with a whole forward kinematic of the robot. This would also include a lot of test drivings to determine grip, max. acceleration etc of the mouse and fine tuning the controller. The final path itself could be optimized by finding the fastest (and potentially not shortest) way through the maze by taking all "driving factors" into account.

A different direction for a new software design iteration is to implement reinforcement learning on several levels. On a lower level to learn controller parameters, but also on a higher level could be promising e.g. to let the mouse find its fastest way through the maze.

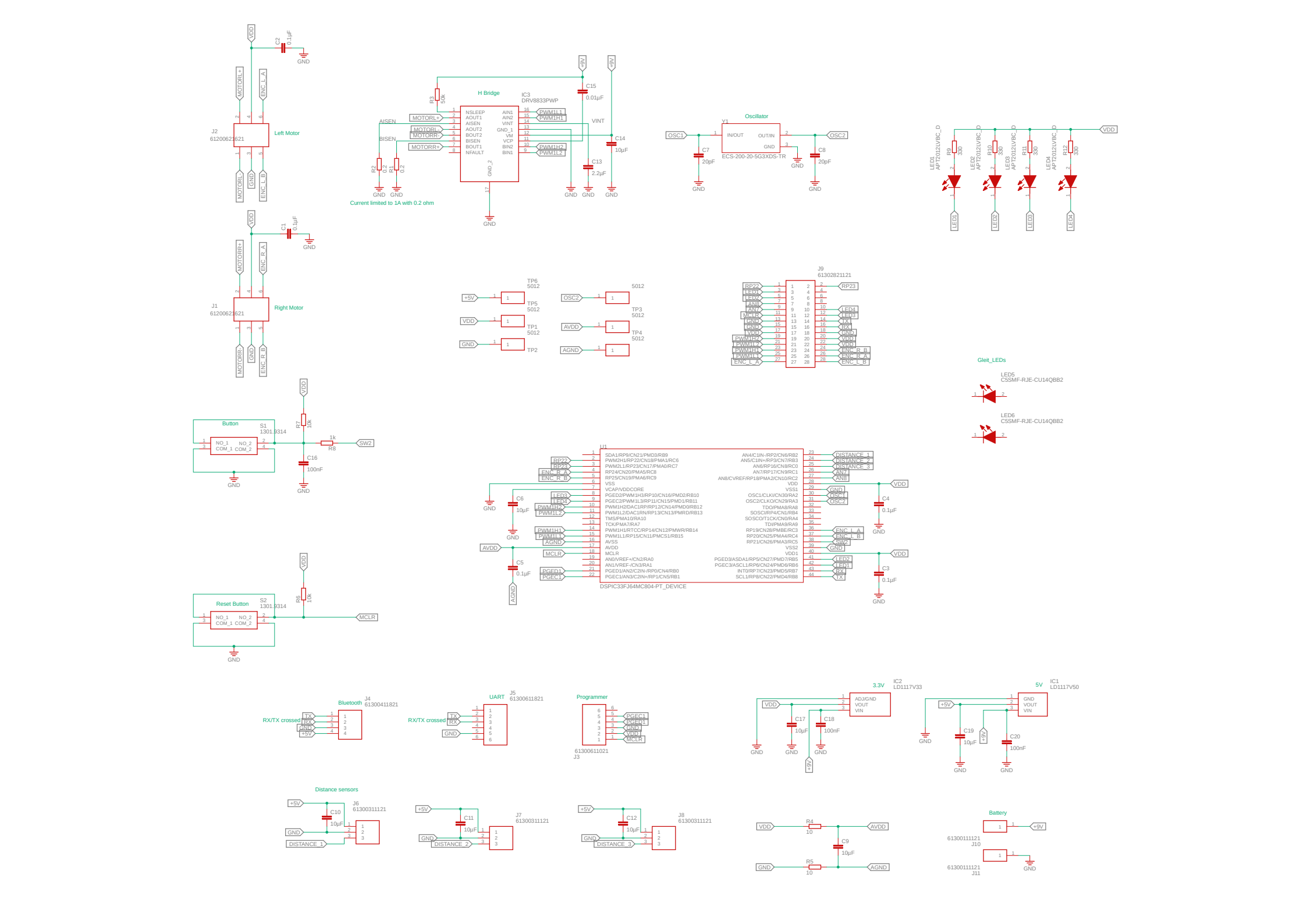
Many problems we encountered could be solved through teamwork and individual research. A valuable lesson learned early on was that most answers can be found in the manuals and data sheets of the respective hardware components. Knowing how to quickly find the info needed in a >400 page document is crucial. However the support of Dr. Lenz was indispensable, as he had overseen this course for some time now, he often directly knew the problems encountered by students and how to fix them.

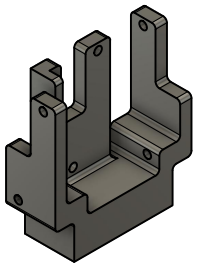
Looking ahead in our professional future, the knowledge and experience gained from this project will undoubtedly serve as a solid foundation for future endeavors in robotics and automation. The course offered a great opportunity to gain practical experience in the fields of electronics and embedded systems, making use of the theoretical knowledge that we acquired in various courses throughout our studies.

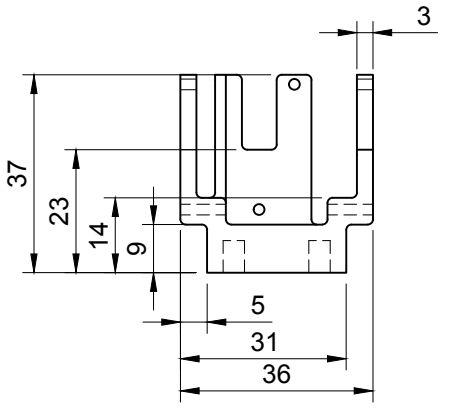
A. Appendix

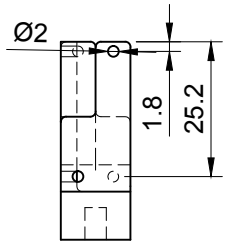
The following pages show:

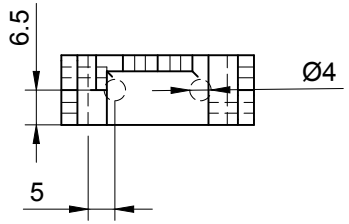
1. Schematics of the PCB
2. Technical drawing of the custom designed sensor mount for the distance sensors


J2
61200621621
Left Motor
MOTORL+
ENC_L_A
MOTORL-
GND
ENC_L_B
J1
61200621621
Right Motor
MOTORR+
ENC_R_A
MOTORR-
GND
ENC_R_B
H Bridge
IC3
DRV8833PWP
NSLEEP
AOUT1
AISEN
AOUT2
BOUT2
BISEN
BOUT1
NFAULT
AIN1
AIN2
VINT
GND_1
VM
VCP
BIN2
BIN1
GND_2
PWM1L1
PWM1H1
PWM1H2
PWM1L2
MOTORL+
MOTORL-
MOTORR-
MOTORR+
AISEN
BISEN
R2 0.2
R1 0.2
R3 50k
C15 0.01µF
C14 10µF
C13 2.2µF
VINT
+5V
Current limited to 1A with 0.2 ohm
Oscillator
Y1
ECS-200-20-5G3XDS-TR
OSC1
OSC2
C7 20pF
C8 20pF
IN/OUT
OUT/IN
GND
LED1 APT2012LVBC_D
LED2 APT2012LVBC_D
LED3 APT2012LVBC_D
LED4 APT2012LVBC_D
R9 330
R10 330
R11 330
R12 330
VDD
TP6 5012
TP5 5012
TP1 5012
TP2
TP3 5012
TP4 5012
+5V
VDD
GND
OSC2
AVDD
AGND
J9
61302821121
RP22
RP23
LED4
LED3
TX
RX
MCLR
PWM1H2
PWM1L2
PWM1H1
PWM1L1
ENC_L_A
ENC_R_B
ENC_R_A
ENC_L_B
Button
S1
1301.9314
NO_1
NO_2
COM_1
COM_2
R7 10k
R8 1k
C16 100nF
SW2
Reset Button
S2
1301.9314
R6 10k
MCLR
Gleit_LEDs
LED5
CSSMF-RJE-CU14QBB2
LED6
CSSMF-RJE-CU14QBB2
U1
DSPIC33FJ64MC804-PT_DEVICE
SDA1/RP9/CN21/PMD3/RB9
PWM2H1/RP22/CN18/PMA1/RC6
PWM2L1/RP23/CN17/PMA0/RC7
RP24/CN20/PMA5/RC8
RP25/CN19/PMA6/RC9
VSS
VCAP/VDDCORE
PGED2/PWM1H3/RP10/CN16/PMD2/RB10
PGEC2/PWM1L3/RP11/CN15/PMD1/RB11
PWM1H2/DAC1RP/RP12/CN14/PMD0/RB12
PWM1L2/DAC1RN/RP13/CN13/PMRD/RB13
TMS/PMA10/RA10
TCK/PMA7/RA7
PWM1H1/RTCC/RP14/CN12/PMWR/RB14
PWM1L1/RP15/CN11/PMCS1/RB15
AVSS
AVDD
MCLR
AN0/VREF+/CN2/RA0
AN1/VREF-/CN3/RA1
PGED1/AN2/C2IN-/RP0/CN4/RB0
PGEC1/AN3/C2IN+/RP1/CN5/RB1
AN4/C1IN-/RP2/CN6/RB2
AN5/C1IN+/RP3/CN7/RB3
AN6/RP16/CN8/RC0
AN7/RP17/CN9/RC1
AN8/CVREF/RP18/PMA2/CN10/RC2
VDD
VSS1
OSC1/CLKI/CN30/RA2
OSC2/CLKO/CN29/RA3
TDO/PMA8/RA8
SOSCI/RP4/CN1/RB4
SOSCO/T1CK/CN0/RA4
TDI/PMA9/RA9
RP19/CN28/PMBE/RC3
RP20/CN25/PMA4/RC4
RP21/CN26/PMA3/RC5
VSS2
VDD1
PGED3/ASDA1/RP5/CN27/PMD7/RB5
PGEC3/ASCL1/RP6/CN24/PMD6/RB6
INT0/RP7/CN23/PMD5/RB7
SCL1/RP8/CN22/PMD4/RB8
DISTANCE_1
DISTANCE_2
DISTANCE_3
AN7
AN8
LED2
LED1
C6 10µF
C5 0.1µF
C4 0.1µF
C3 0.1µF
Bluetooth
J4
61300411821
RX/TX crossed
UART
J5
61300611821
Programmer
J3
61300611021
PGEC1
PGED1
VDD1
3.3V
IC2
LD1117V33
ADJ/GND
VOUT
VIN
C17 10µF
C18 100nF
5V
IC1
LD1117V50
C19 10µF
C20 100nF
Distance sensors
J6
61300311121
J7
61300311121
J8
61300311121
C10 10µF
C11 10µF
C12 10µF
R4 10
R5 10
C9 10µF
AVDD
Battery
+9V
J10
61300111121
J11
61300111121









Dept.	Technical reference	Created by Noah Bucher 08.10.23	Approved by	
		Document type finished document	Document status	
		Title Sensor_Mount_V3	DWG No.	
			Rev. / Date of issue	Sheet 1/1

List of Figures

List of Tables

Bibliography

[1] "Micromouse". In: *Wikipedia* (June 2023). (Visited on 09/04/2023).

[2] *Fusion 360: Kosten | Preise & Verträge | Autodesk*. https://www.autodesk.de/products/fusion-360/overview. (Visited on 09/04/2023).

[3] S. K. Surojit Guha. *Maze Solving Algorithms for Micro Mouse*. URL: http://www.123seminarsonly.com/Seminar-Reports/038/59360985-Maze-Solving-Algorithms.pdf (visited on 10/09/2023).

[4] *Microchip dsPIC33FJ32MC302/304, dsPIC33FJ64MCX02/X04 and dsPIC33FJ128MCX02/X04 data sheet*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/70291G.pdf (visited on 09/17/2023).

[5] *LD1117 Adjustable and fixed low drop positive voltage regulator data sheet*. URL: https://www.mouser.de/datasheet/2/389/ld1117-1849389.pdf (visited on 09/17/2023).

[6] *Bluetooth module HC-05 data sheet*. URL: https://cdn-reichelt.de/documents/datenblatt/A300/DATENBLATTHC05_HC06.pdf (visited on 09/17/2023).

[7] *Sharp GP2Y0A51SK0F distance sensor data sheet*. URL: https://global.sharp/products/device/lineup/data/pdf/datasheet/gp2y0a51sk_e.pdf (visited on 09/17/2023).

[8] *Faulhaber Series 2619 ... SR ... IE2-16 data sheet*. URL: https://www.faulhaber.com/fileadmin/Import/Media/EN_2619_SR_IE2-16_DFF.pdf (visited on 09/17/2023).

[9] *Kingbright APT2012LVBC/D data sheet*. URL: https://www.mouser.de/datasheet/2/216/APT2012LVBC_D-535767.pdf (visited on 09/17/2023).

[10] *DRV8833 Dual H-Bridge Motor Driver data sheet*. URL: https://www.ti.com/lit/ds/symlink/drv8833.pdf (visited on 09/17/2023).

[11] *DIGILENT PmodUSBUART Reference Manual*. URL: https://digilent.com/reference/_media/pmod:pmod:pmodusbuart_rm.pdf (visited on 09/17/2023).

[12] *Schurter 6x6 mm tact switches data sheet*. URL: https://www.mouser.de/datasheet/2/358/typ_6x6_mm_tact_switches-1275689.pdf (visited on 09/20/2023).

[13] Mack. *mackorone micromouse simulator (mms)*. Sept. 2023. URL: https://github.com/mackorone/mms (visited on 09/28/2023).