Micromouse

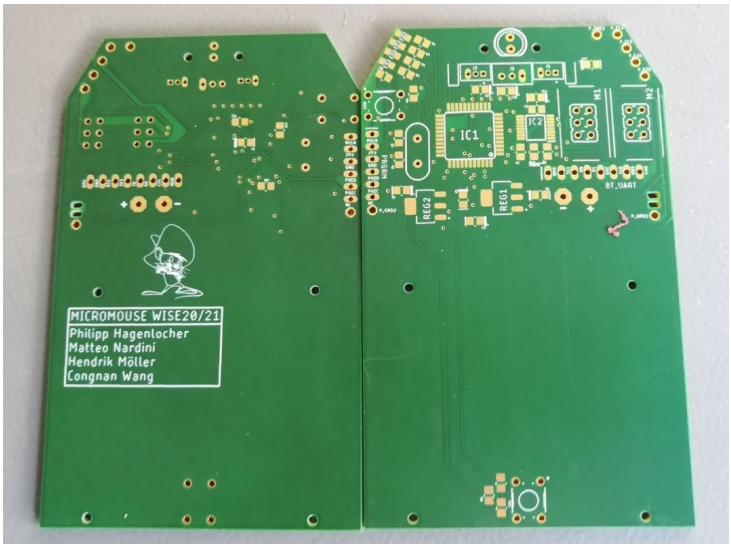
Designing an Educational Racing-Robot from Scratch

Master Practical Course at Technical University of Munich

Authors
Philipp Hagenlocher,
Hendrik Möller,
Matteo Nardini,
Congnan Wang

Supervisor:
Dr. Alexander Lenz

Date:
March 30, 2021



Contents

1 Introduction

The "Amazing Micro-mouse challenge" is a robotic competition that has been introduced in 1977 [4]. Since then, it has become a well-known introductory exercise in the field of robotics, as it is challenging enough to be non-trivial, yet simple enough to be approachable by someone without significant experience.

The Micromouse competition consists of building a robot, which can solve a previously unknown maze. First, the robot must explore the maze of 16×16 cells, each being a square with sides of 18 cm. It starts from a fixed corner position and must find the center of the labyrinth, composed of four empty unit squares. After that, the robot races from start to finish, trying to succeed in the least time possible.

Over the years, the rules of the challenge have evolved. We followed the set of rules proposed by the University of York [31].

Figure 1: Example of a real-world micromouse challenge [taken from 16].

This challenge may, at first glance, sound simple, yet entails a lot of different tasks. Starting from scratch, we had to plan our own hardware, design schematics with the help of EAGLE and construct a PCB design [11]. Then we had to order the respective parts and solder them together. On the software side awaited many problems to be solved. There needs to be a controller that can ensure, despite any deviations, that the robot is able to move in a straight line, while higher level modules must keep track of the robots movement and "solve" the maze. These are just some of the issues that were presented to us.

In this report, we will describe our solution to this challenge, developed during the "Micromouse: Designing an Educational Racing-Robot from Scratch" Advanced Practical Course offered at the Technical University of Munich in the winter semester 2020-2021.

In section 2, we detail our conceptual design and justify our high-level design choices. Section 3 presents all the hardware we chose and summarize their respective functionalities. Then, all our software plans and modules are established in section 4, explaining the sequence of our program. Section 5 discusses the problems we encountered.

2 Conceptual Design

This section offers an high-level overview of our solution. A picture of our Micromouse is presented in Figure 2, as we think it will be easier to follow the remainder of the document with a visual image of our solution. The robot is 111mm long, 81mm wide and 49mm tall.

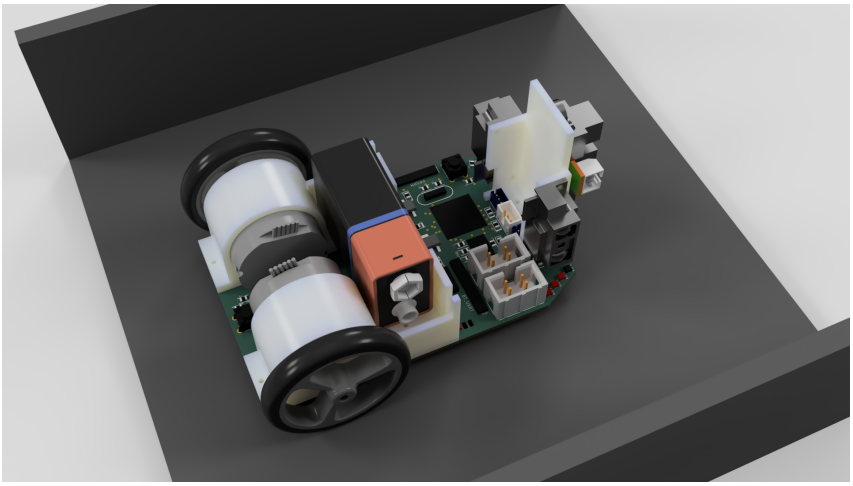


Figure 2: An overview of our solution.

2.1 Hardware overview

Besides the rules cited in section 1, we had a few additional hardware constrains:

- The motor to be used were given. The usage of two Faulhaber EN 2619 S 006 SR IE2-16 was mandatory.
- The microcontroller to be used was given. The usage of the dsPIC33FJ64MC804 was mandatory.

Given these constraints, we sketched a high-level overview of the hardware required for our mouse, which is shown in Figure 3.

The setup is fairly standard and can be divided into four main areas:

- Sensing:
  We decided to use three infrared distance sensors to detect the presence of the maze walls. They are connected directly to the microcontroller.
- Acting:
  The two motors were given, and we chose to drive them with a double H-bridge driven by the microcontroller's PWM module.
- Communication:
  We decided to support both wired UART, which is useful in the initial debugging phases, and UART over Bluetooth, which is more comfortable to use while the mouse is solving the maze.

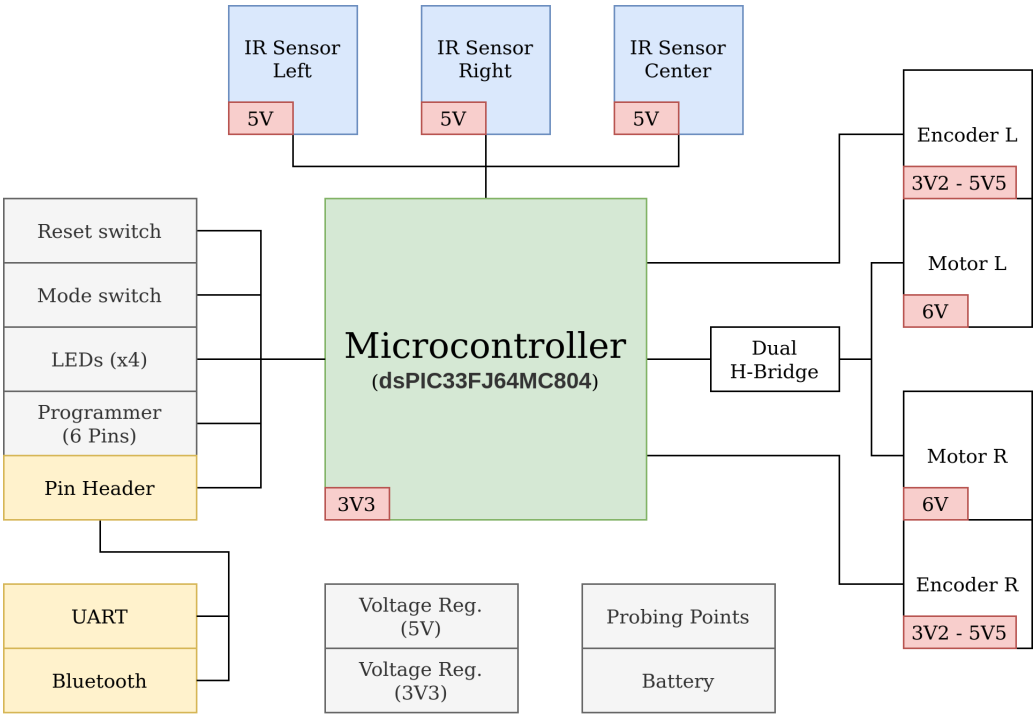


Figure 3: The initial concept for the hardware setup.

- Support:
  All the components needed to make the others work: power supply, programming headers, LEDs and switches (useful for debugging).

Given the dimensions of the mouse, we chose to use the PCB itself as the main structural element. On top of it we decided to mount a couple of custom-designed and 3D-printed motor mounts and sensor mounts to hold these components.

2.2 Software overview

On the software side we settled on fairly standard design patterns for embedded development.

We implemented an Hardware Abstraction Layer (HAL) that exposes an intuitive API to manipulate the microcontroller's state without having to always interact directly with its registers. Besides that, we implemented a chain of controllers: the lower levels of the chain are directly aware of hardware details and take care of lower level functionalities like moving in a straight line, whereas higher levels of the chain are responsible for being aware of the maze and use the functionalities exposed from the lower levels to navigate it.

On the control side, we decided to use a PID controller, but discarding the derivative component, as it makes tuning harder and our application is simple enough that we do not require the faster convergence offered by it. We have two set of controllers: one responsible for maintaining the correct distance from the walls and one responsible for maintaining each individual motor to the desired speed.

A block diagram of our software architecture is shown in Figure 4.

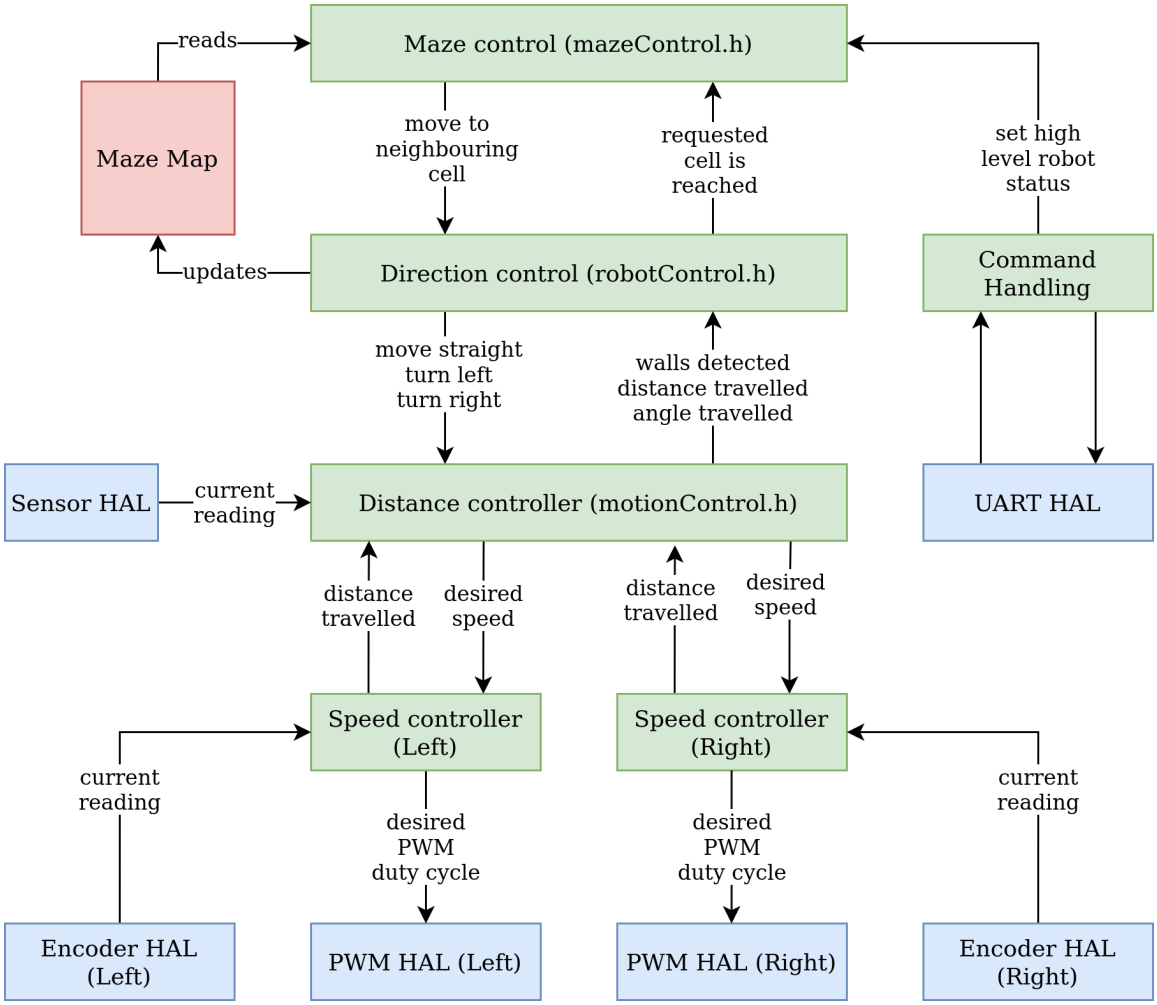


Figure 4: An high level overview of the software architecture.

3 Hardware Design

This section describes our choice of parts and their respective functions within the Micromouse.

3.1 Power Delivery

Based on our initial concept seen in Figure 3, we identified the need for a 5 Volt and 3.3 Volt regulator, because we wanted to use a 9 Volt battery for power delivery. Since the motors are directly connected to the dual H-Bridge, the 6 Volt can be supplied by it through PWM. The software has to scale the PWM signal accordingly to never exceed maximum power capabilities of the motors. This topology has the advantage that power going to the motors does not have to be regulated, leading to smaller power consumption and heat generation.

The final choice fell on the LM1117 series voltage regulators due to their maximum output current of 800 mA and relatively low cost [13]. A single regulator also only requires two additional components for it to function and their small SOT-223-4 footprint make them ideal candidates for our board in terms of electrical capabilities and space requirements. In addition, their typical dropout voltage of 1.2 Volts is low enough to use them in series.

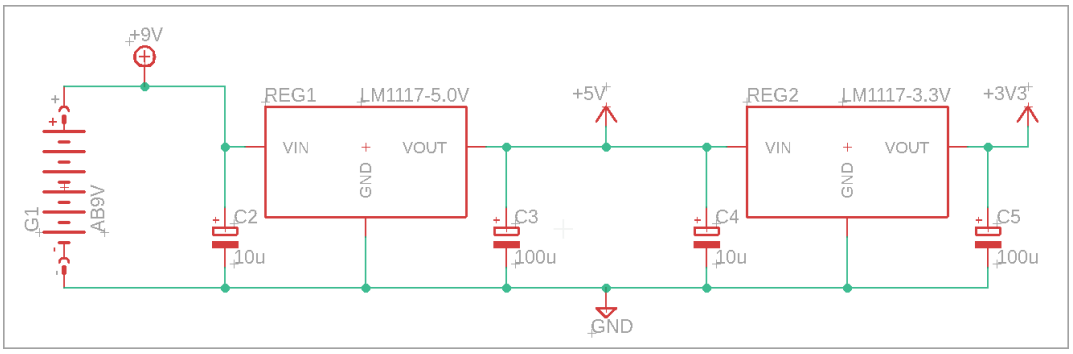


Figure 5: Voltage Regulation

Figure 5 depicts our wiring for the voltage regulators with the battery. The regulators are wired in series to minimize the power dissipated by the 3.3 Volt regulator and the capacitor values were chosen according to the recommendations of the datasheet [13, p. 15-16].

From Table 1, we see that the maximum of 800 mA the regulators can source are more than enough, even for maximum current spikes. However, we wanted to make sure, that the heat dissipated by the regulators is not too high, in order to ensure safe operation. For the calculation, we made the assumption that the thermal resistance from the component junction to the board is lower than the thermal resistance from component junction to ambient. This assumption is reasonable since the components are directly soldered to the board and copper and solder material have a lower thermal resistance than air. Thus, we will use

$$R_{\text{thermal}} = R_{\theta JA} = 61.6 \, \frac{^\circ C}{W} \tag{1}$$

Component	Typ. (mA)	Max (mA)	Amount
Controller	63	76	1
Sensor	12	22	3
LED	1.89	2	4
Encoder	8	15	2
Regulator (quiescent current)	5	10	2
Total	I_{Typ} : 126.56	I_{Max} : 200	12

Table 1: Components and their typical and maximum current draw from regulated voltage sources [22, p. 360][7, 6, 15, 14, 13].

as the thermal resistance for our calculations to get an upper bound on the temperature of our components junctions [13, p. 3]. The power dissipated by the regulators is calculated by multiplying the voltage drop across them with the current flowing through them. Multiplying this with the thermal resistance will give us the temperature at the junction of the component. We do this calculation for the expected typical and maximum current draw.

$$\textbf{5V Typ} = (9V - 5V) \cdot I_{\text{Typ}} \cdot R_{\text{thermal}} \approx 31.18\ °C \tag{2}$$

$$\textbf{5V Max} = (9V - 5V) \cdot I_{\text{Max}} \cdot R_{\text{thermal}} \approx 49.28\ °C \tag{3}$$

Using the values from Table 1, Equation 2 and Equation 3 show the typical and maximum temperatures generated by power dissipation. The temperatures for the 3.3 Volt regulator must be below these, since the voltage drop across it is smaller. Even with an ambient temperature of 24 °C added to the equation both regulators are well within the recommended operating temperatures of 0 – 125 °C [13, p. 3].

3.2 Oscillator

When choosing the external oscillator for our microcontroller, we were aiming for a simple, yet effective design. The number of external components should be rather small but still give a usable accuracy. Additionally, since 16 MHz oscillators were already available, we did not choose to go with a 20 MHz oscillator over a 16 MHz variant since we do not think the performance gain is worth ordering another component.

The oscillator is a quartz crystal, which in turn is an LC circuit consisting of an inductor in series with a capacitor. In order to use such a crystal as a stable oscillator for our purpose, special topologies have to be used. The usual setup for a project like ours is the Pierce-Oscillator topology shown in Figure 6 [21]. Only two additional capacitors are needed for the crystal to produce a usable oscillator with the `XTAL` pins on the microcontroller. The capacitance added by the capacitors is called the load capacitance C_{Load} that the crystal requires. Another reason to choose the 16 MHz oscillator is its load capacitance of 18 pF, which is more suitable for a fixed frequency oscillator [3].

When calculating the capacitance for the capacitors (called C6 and C7, see Figure 6), we aimed for a realistic model, so we tried to take the stray capacitance C_{stray} of the

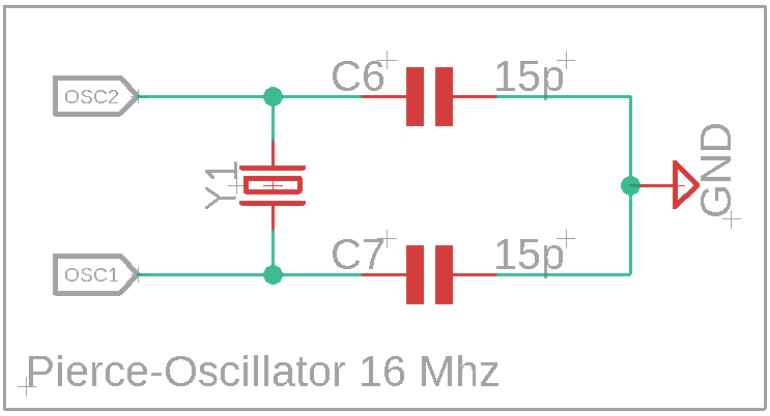


Figure 6: Oscillator schematic

PCB tracks and the load capacitance introduced by the microcontroller into account, since they contribute to C_{Load}. Therefore, in our model the nets `OSC1` and `OSC2` also have capacitances going to ground which we will refer to as C_{pin}. The capacitances of C6 and C7 are called C_{ext}. Our full model is constructed after [3]. We also assume the capacitances of the pins and capacitors to be the same.

$$\begin{aligned} C_{\text{Load}} &= \frac{(C_{\text{pin}} + C_{\text{ext}})(C_{\text{pin}} + C_{\text{ext}})}{C_{\text{pin}} + C_{\text{ext}} + C_{\text{pin}} + C_{\text{ext}}} + C_{\text{stray}} \\ &= \frac{(C_{\text{pin}} + C_{\text{ext}})^2}{2 \cdot (C_{\text{pin}} + C_{\text{ext}})} + C_{\text{stray}} \end{aligned} \tag{4}$$

Since the microcontroller datasheet does not specify the capacitance of its `XTAL` pins, we assume 5 pF for both of them [3]. Choosing $C_{\text{ext}} = 15$ pF and using $C_{\text{pin}} = 5$ pF and $C_{\text{stray}} = 5$ in Equation 4, we arrive at a load capacitance of 15 pF, which, after discussion with our advisor, seemed to be a safe choice.

3.3 Motor driver & Encoders

In order to control the motors of our mouse, we have to deliver current at a voltage that the controller cannot supply. For that reason, we are using an H-Bridge IC which, in turn, is controlled by the controller via pulse-width modulation. When choosing the IC following criteria were of importance:

- Dual H-Bridge setup to drive both motors with one component
- A short-term maximum supply current of at least 2 Ampere
- Small footprint
- Fairly good efficiency

All of these characteristics match up with our choice, the DRV8833-PWR Dual H-Bridge [12], which supports 2 Ampere short term current for each motor, has a small TSSOP-16 footprint and a low on-resistance for its internal MOSFETs of 360 mΩ. Furthermore, it

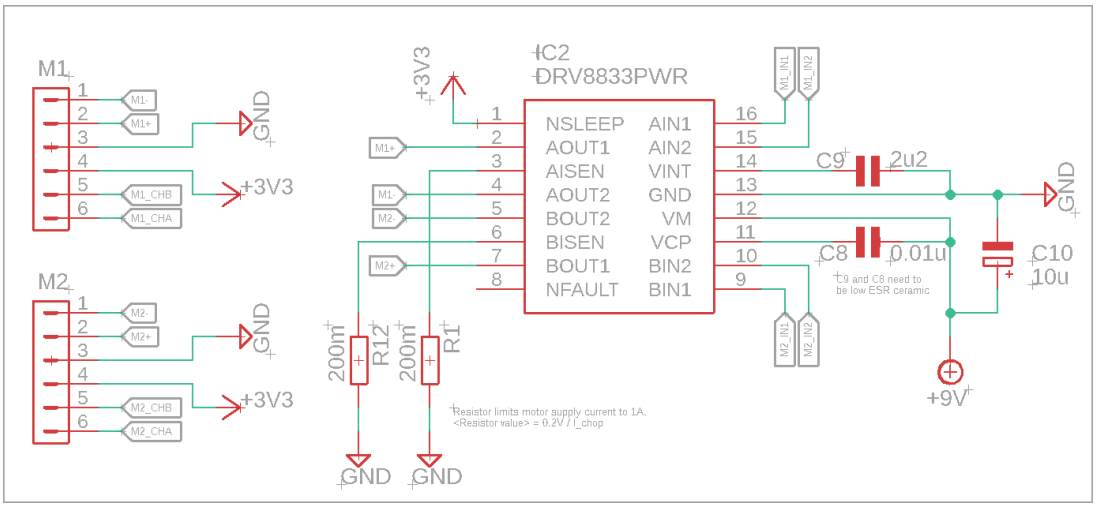


Figure 7: H-Bridge and Motor wiring

supports two different "decay modes" that can be used to either actively brake or let the motor coast. Since our motors operate at 6 Volts, we have to scale the power delivered to the motor by using PWM [14]. Additionally we are clamping the maximum current supplied by the H-Bridge to the motor down to one Ampere per motor. This is achieved by connecting the sensing pins of the IC to ground with a current sensing resistor of a low resistance value. Equation 5 shows how to calculate resistance for a certain current at which to clamp down. Since $I_{\text{chop}} = 1$ Ampere for our case, R_{isense} is 200 mΩ. The IC also provides a pin for detecting faults (overtemperature, overcurrent) but we decided against using it since we do not expect a fault to be triggered. Figure 7 shows the relevant part of the schematic for the motor control. The external passive components for the H-Bridge IC are directly taken from the recommendations of the datasheet.

$$R_{\text{isense}} = \frac{0.2V}{I_{\text{chop}}} \tag{5}$$

The encoders of the motor are driven with 3.3 Volts to keep them at a logical level that the controller can work with. Thus, the encoder outputs of the motor are directly wired to pins on the controller.

3.4 Sensors

We identified two main type of sensors that could have been used in our mouse:

- The Sharp GP2Y0A51SK0F and GP2Y0A41SK0F infrared sensors [7, 6]. They are quite bulky and their 25ms refresh rate does not make them particularly fast. Moreover, being infrared sensor, they are quite sensible to the reflectivity of the surface being tested. However, they provide a simple to use analogue output that can be directly fed to the microcontroller's ADC. They can measure distances between 2-15cm and 4-30cm, respectively.
- The STMicroelectronics VL53L0X time-of-flight sensors [19]. They are smaller, more precise, less susceptible to reflectivity of the surface being tested and (in

high speed configuration) faster than the Sharp sensor. However, due to soldering requirements, we would need to buy them in a pre-made breakout board (giving up some of the size advantage), and are significantly more complex to use, as they can operate in different modes and require I2C communication. Moreover, given that our microcontroller supports a single I2C bus, these three sensors would have to be individually powered on and configured with a non-conflicting I2C address before being usable.

We decided to use the Sharp ones. They are overall worse with respect to the VL53L0X, but we favoured their simple interface, due to the fact that many members of the group are beginners in the field of hardware design and embedded development. In addition, given the extremely predictable conditions that we can expect in the maze, many of the disadvantages of the Sharp sensor become less relevant.

3.5 LEDs

For the purposes of debugging, we wanted to add LEDs to the PCB in order to get visual feedback from the mouse if other communication fails. Since the microcontroller has limited capabilities for supplying and sinking currents, working with typical LEDs proves to be challenge. An early solution was the usage of dedicated driver transistors for each LED, which would add the benefit of almost no current flowing from or into the microcontroller. However, we decided against the additional components due to space constraints and a negligible advantage. In the end we decided to use LEDs with very low forward current of 2mA [15]. Each LED is wired in series with a 820 Ω resistor and the controller which sinks the current:

$$I_{\text{LED}} = \frac{(V\text{cc} - V_{\text{drop}})}{R} = \frac{(3.3V - 1.75V)}{820\Omega} \approx 1.89\text{mA} \tag{6}$$

For our choice of LED, the microcontroller has to sink approximately 1.89mA per pin (see Equation 6), which is well within the specification.

3.6 Switches, Probe points & Pin Headers

In order to reset the robot and start its operation, we are equipping it with two tactile push-buttons. Figure 8 shows the corresponding schematic. The `MCLR` label is directly connected to the corresponding pin on the controller, pulling the signal to a low potential if the button is pressed and thusly initiating a reset. The `SW1` label is connected to a remappable pin. `R2` and `R4` are pull-up resistors causing a voltage potential read by the controller as a logical high. `R3` and `R5` used for over-current protection in the case of wrongly configured pins pulled to ground. Additionally, `R5` protects the pin from current flowing from the capacitor `C12`, which is used for debouncing the switch. Such a capacitor is not used on the `MCLR` net in order to ensure operation with the programmer. Such a capacitor is not recommended by the user's guide for our programmer [23, p. 21]. Values for the parts where chosen to be similar to the values of the dsPic starter kit [20, p. 41] and were cross-validated with the datasheet of the controller [22, p. 19].

For debugging and testing purposes, we have added probing points, four of which are connected to the encoder outputs of the motors. The other three probing points are connected to ground and spread over the board to have reference points for testing the motor encoders, the voltage regulators (which can easily be probed on the big pad of the SOT-223-4 package) and the UART pin header. Their positions can be seen in Figure 11 marked `P_name`.

We are using angled pin headers for connections with the programmer and an additional bluetooth module handling UART communication. Since the UART signals are sent through the corresponding pin header we can opt out of bluetooth communication and use a wired UART connection if we so desire.

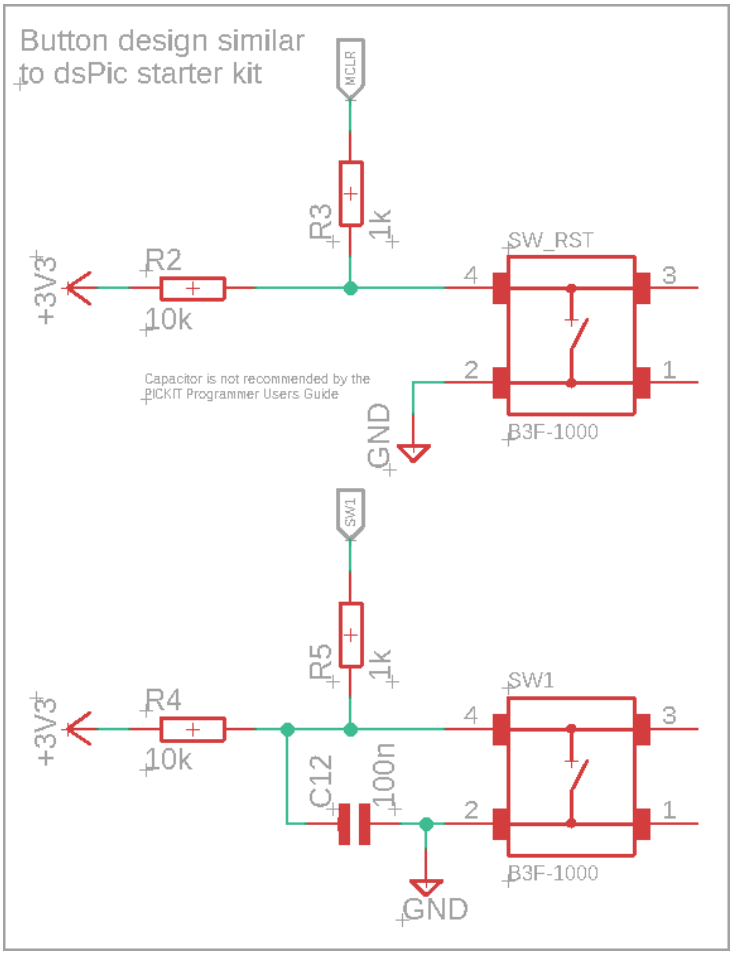


Figure 8: Schematic for the push-buttons.

3.7 PCB design

The main challenge for the PCB design was the minimization of used space, since a smaller mouse has more choice for movement. The size of the PCB is mainly governed by the general size of the mouse and the space taken up by the connecting parts for the motors and sensors. We initially started with a rough concept for the mouse which can be seen in Figure 9. In the end, the shape of the mouse did not change significantly except for a shorter length, as shown in Figure 2.

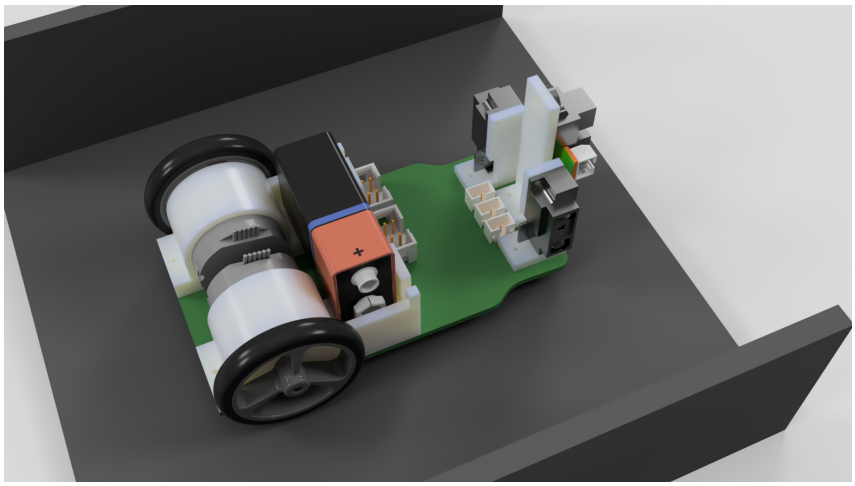


Figure 9: The rendering of the first iteration of our design.

Figure 10 shows the first draft of our PCB with the connecting parts for motors and sensors highlighted. In order to ease the back and forth synchronization between EAGLE and Fusion, we roughly marked the footprints for the connecting parts on the silkscreen of the PCB to have a visual aid for the rest of the design. It was clear that almost all components needed to live in the space between sensors and motors. While we could have put parts between the motors, this would have caused the need to route power tracks down to the board, which, in turn, would require us to reroute signals with vias (by switching sides on the two layered board) which we wanted to avoid. Another problem was the placement of the LEDs (originally we wanted to use 3mm THT LEDs, which we later changed to SMD LEDs) for which we had no space with the dedicated driver transistors (see subsection 3.5).

After a design review among our team, we recognized many opportunities for condensing the space, mainly by switching the positions for the motor connectors (called `M1` and `M2`) and the voltage regulators (called `REG1` and `REG2`). Additionally, we were able to rework the sensor mount, allowing the placement of components in the area marked in red in Figure 10. By doing so, almost all of the board is used efficiently either by containing components or supports for motors or sensors. Additionally, we designed custom footprints to further aid our design (see subsection 3.8). The final top part of our design can be seen in Figure 11.

An important design consideration was the signal integrity of the sensor signals, coming from the larger pads of the sensor connectors (marked `J1`, `J2` and `J3` in Figure 11). This meant keeping the tracks as short as possible, which led us to the decision to put the microcontroller as close to the connectors as possible. This also restricted our further choices since we had to connect the oscillator and the PWM pins to the corresponding components, which made the placement of the oscillator and H-Bridge IC mandatory. The oscillator was placed as close as possible to the microcontroller without overlapping other tracks on the top-side of the PCB (red tracks in Figure 11). We payed attention to put power tracks on the outside of the board to be able to route most signal tracks on the top of the PCB: in this way we managed to keep the ground-plane of the bottom layer with as little separations as possible. It was particularly important to have a direct path from the ground connections of the sensors to the ground connection of the battery,

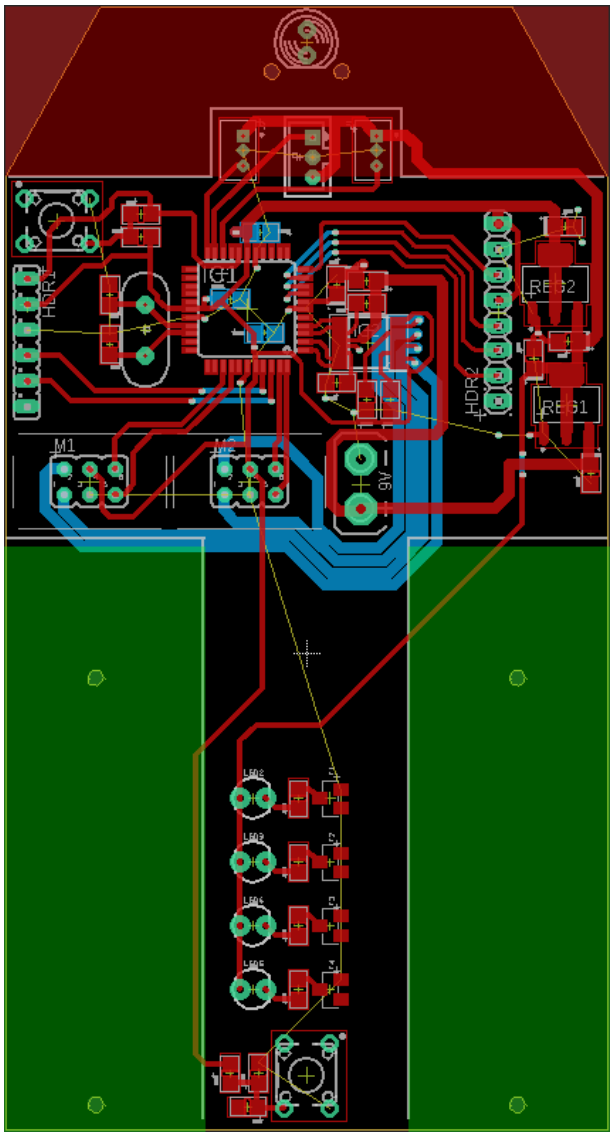


Figure 10: First draft of the PCB with motor and sensor supports marked.

in order to have a stable reference for the sensor value. The ground connection of the battery connector was placed near the middle of the board to achieve this goal.

Bypass capacitors for the voltage regulators, the microcontroller and the sensors as well as capacitors needed by the H-Bridge IC were kept as close to the corresponding components as possible. This meant putting the bypass capacitors for the microcontroller on the bottom side of the board, which is still within the 6mm maximum length requirement [22, p. 17].

Track widths were adjusted to the pin sizes for the corresponding components, keeping them thick enough to make it possible to manually reroute connections if our design should prove to be faulty.

Except for a few 90° angles, sharp angles were generally avoided in the design of tracks to minimize the risk of acid traps in the PCB production process. Connections to vias and through-hole pins were converted to a tear drop design which also aids in the etching process for PCBs. The adjustments are visible on the `P_Exx` pins and the `3V3` pin on the

left pin header, where manual adjustments to the track were done. The tracks going to the LEDs on the top left have a curved design to have a near 90° entry angle into the pads. Curved tracks should also minimize reflections, should we choose to let the LEDs blink at a high frequency.

Superfluous vias where added to the design to create short current return paths for our components and signals. We can generally assume the return path to go directly to ground since we are not generating high-frequency signals. The most important signals, the sensor signals, for which signal integrity is vital, have a horizontally spaced line of vias between them and the microcontroller, which could aid in improving signal integrity due to the vias ability to act as coupling points for electro-magnetic interference from the microcontroller.

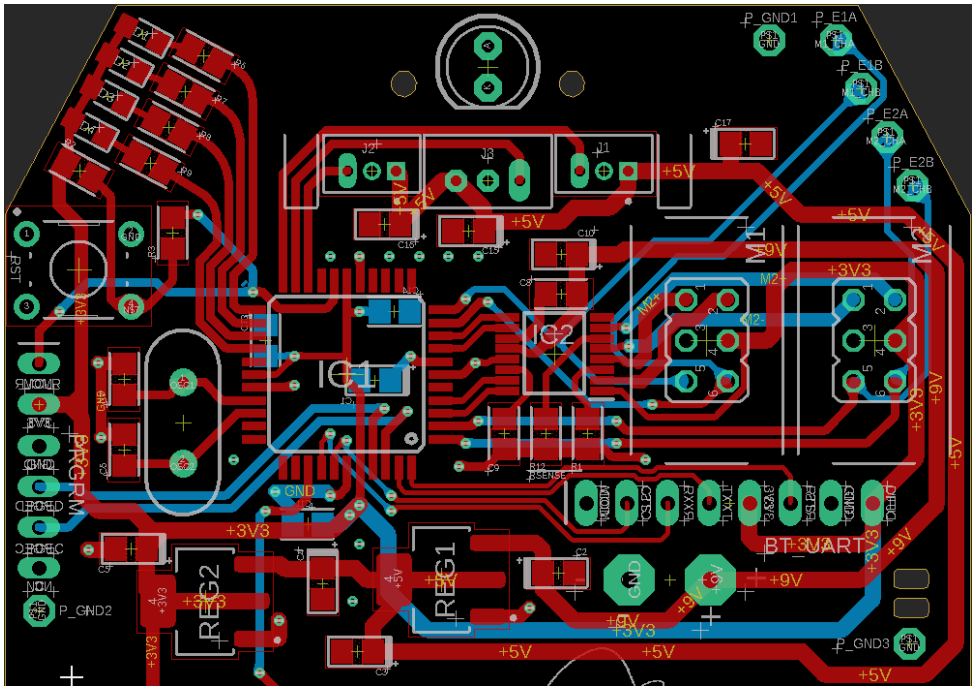


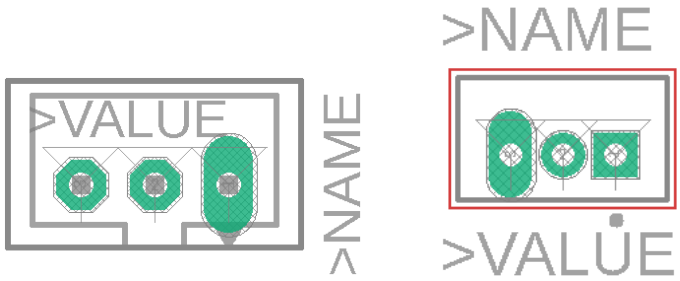
Figure 11: Top part of the final PCB

Even though the components are close together, a "soldering order" has been established to make the building process easier. The first element to be soldered is the microcontroller, followed by external components for the H-Bridge IC, the IC itself, the regulators, then the external components and after that, the rest of the SMD components, soldering the through-hole components last. The PCB features two holes on the right acting as a stress relief for the wires of the battery connector. We also designed custom footprints to create a cleaner silkscreen layer (see subsection 3.8). The design should thus aid in creating an easy to build, yet still effective and reliable robot.

In the design process both EAGLE and Fusion360 [11] were used, which proved challenging at times. The synchronization of the board outline seems to be a faulty one way connection between the two programs. However, we liked the ability to see changes of our PCB immediately in a 3D view, which made verification of our design easier. For soldering, we are using an interactive board viewer created with a KiCad plugin [18].

3.8 Footprints

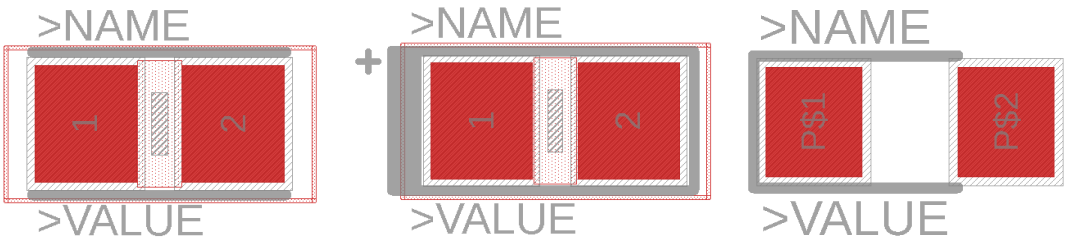
Many of the footprints used in our design were designed by us to aid the build process. To save space we restricted ourselves strictly to 0805 footprints.



(a) B3B-PH (b) B3B-ZR

Figure 12: Footprints for sensor connectors

The footprints for the sensor connectors adhere to the spacing for B3B-ZR [9] and B3B-PH [8] connectors by JST. Figure 12 depicts both of them. The elongated pad corresponds to the signal pin of the sensor, making probing the signal easier.



(a) 0805 (b) 0805 (polarized) (c) 0805 LED

Figure 13: Footprints for 0805 components

Capacitors, resistors and LEDs are in a 0805 package for which we designed footprints shown in Figure 13. Figure 13a is used for non-polarized capacitors and resistors, Figure 13b is used for polarized capacitors and Figure 13c is used for our LEDs. Footprints for resistors and capacitors have a keep-out zone between the two pads to make sure that no tracks are accidentily routed between them. The footprint for the LEDs were made to the specification from the datasheet of our LEDs [15].

The motor connectors, probe points and pin headers also use custom footprints. The motor connector footprints have the outline of the physical connectors shown on the silkscreen (see Figure 11). The probe points have a diameter of 1.32mm to use the RS Pro probe points.

3.9 Communication

Even a simple robot has to internally maintain a quite complex state. Debugging and/or influencing such state only through buttons and LEDs is limiting and frustrating. Therefore, we needed to implement some form of communication between our laptops and the robot.

We decided to use UART, as it is simple to implement and use and does not require a shared clock between the two devices. In a first phase, in order to eliminate potential sources of error, we want to test UART communication via wire. However, as soon as a basic level of functionality is achieved, we plan on using UART over Bluetooth Low Energy (BLE) to communicate with the mouse without the hindrance of wires, enabling us to receive telemetry even during the maze exploration.

In order to achieve this, we connected the UART pins of the microcontroller to a pin header on the PCB. In the first phase, wires can be connected directly to these pins. Afterwards, our BLE module can be connected to them and replace the wires. These pins act also as structural support for the BLE module, as it would have been quite difficult to find a space for a dedicate mount on our PCB.

We decided to use the Adafruit 2479 BLE module: products from Adafruit tend to target amateurs, and therefore are usually very simple and intuitive to use, although the flexibility they offer is a bit limited. The 2479 BLE module is a breakout board for a BLE module that is designed to enable the use case of "UART over BLE communication" with minimal configuration: since our focus is not on Bluetooth functionalities but rather on programming a functional robot, we find this to be the perfect match for our requirements.

This module starts by default in a "data mode" that enables "transparent UART". All the data send to the UART RX pin of the breakout board is automatically forwarded in the `RX` GATT characteristic of the UART Service that is exposed by the BLE device, and similarly all data written to the `TX` GATT characteristic is relayed to the UART TX pin of the breakout board [1].

3.10 Structural components

As stated in section 2, due to the constrained dimensions of our robot, we decided that the PCB of our mouse will also be its main structural element. This allows us to keep a compact form factor (although the size of the sensors is sub-optimal for this purpose), which improves the flexibility of the mouse inside the maze cells.

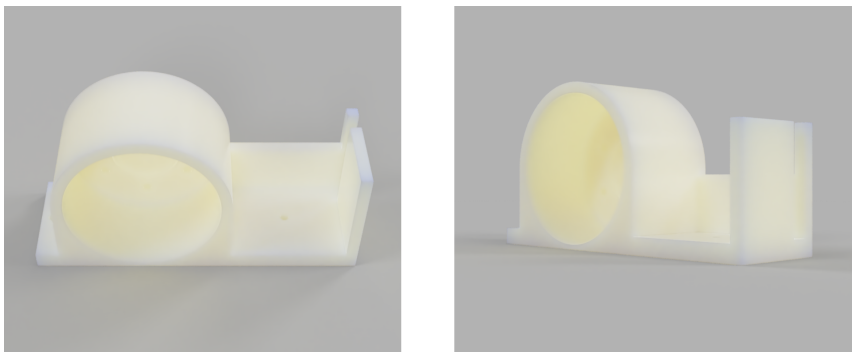


Figure 14: The combined motor and battery mount that we designed

Nevertheless, besides the main PCB we also needed motor mounts, sensor mounts and some kind of battery holder. We decided to design these parts from scratch: this allowed us to reach a fairly integrated solution, which again enables a fairly compact assembly

while reducing the amount of screws used (and therefore weight). These parts have been designed with Fusion360 [11] and then 3D printed.

The combined motor and battery mount we designed is shown in Figure 14. Initially, each sensor had its own sensor mount, but we quickly realized that this lead to wasted space on the PCB and to an unnecessary increase in the number of used screws: we decided therefore to combine them into a single one, reaching the final design shown in Figure 15.

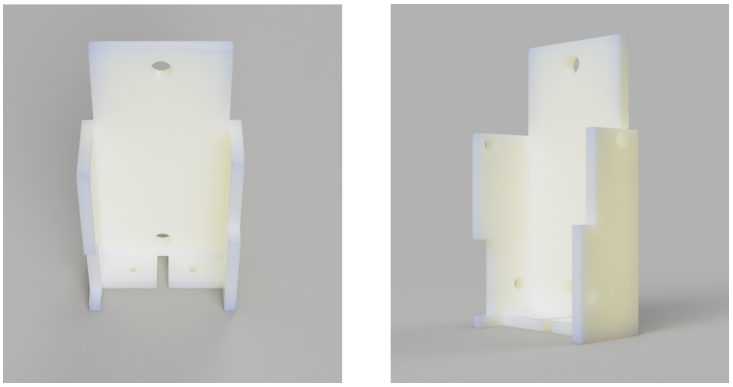


Figure 15: The sensor mount that we designed for our three sensors

The width of the robot is limited by the size of the motors, and the placement of the battery allows for an easy access to it while also contributing in moving the center of mass forward, reducing the possibility of the robot flipping. The third contact point with the ground is a THT LED positioned in the front of the PCB, on the bottom side: since the floor is very smooth, such a solution is practical and sufficient.

4 Software Design

As can be seen in Figure 4, we divided the software into different layers and modules. The next sections will give a short overview over each module and then explain in detail how every part of it works.

4.1 Workflow Overview

First of all, we would like to present an overview of how the data flow inside our software stack works, as presented in Figure 4.

The positions measured by the motor encoders are obtained from the hardware through our HAL by the two "Speed" PI controllers, one for the left motor and one for the right motor. The set point for these two controllers are in turn determined by another "Distance" PI controller that uses the sensors data to ensures that the correct distance from the walls is maintained, allowing for movement in a straight line.

This distance control layer offers a high-level API to the "Direction Control" layer that exposes functionalities such as "straight ahead" or "turn on the spot". Additionally, other parameters such as linear and angular distance travelled from the last API invocation are provided to higher levels of control.

"Direction Control" is the first layer that is aware of the notion of maze cell. It keeps track of positional information (in which cell the robot is in), of its orientation, and updates the maze map with the walls of the current cell. This maze map is shared with the highest level of control, "Maze Control", which keeps track of what should be the next action of the robot. It offers some callbacks that can be used from "Direction control" to determine the neighbouring cell where the mouse should move next.

Commands received from UART trigger the invocation of API calls exposed by "Direction Control" and "Maze Control" and result in changing the robot state from an idle situation to an exploration and maze-solving phase.

4.2 Hardware Abstraction Layer (HAL)

At the lowest level of the software stack, we decided to follow the "Hardware Abstraction Layer" design pattern, or "HAL" for short [28]. Using such a design pattern entails creating a low level API that applies a set of changes to the hardware registers to achieve a desired low level behaviour (e.g. starting a timer) without the need for the caller of the API to be aware of all the hardware details.

When compared to others [30, 10], our HAL offers a slightly higher level of control to the rest of the system: instead of exposing directly hardware concepts such as PWM units, ADC units and DMA channels, it provide more ergonomic functionalities such as obtaining the distance read from the front sensor in millimeters and setting the duty cycle of the left motor. This reduces the portability of our code, but makes it significantly easier to reason about when reading higher level control code, making the trade-off acceptable. Such API can be found in the "halapi.h" file, and an example of some of the available

functions are shown in the following extract:

```
/**
 * Returns the distance in millimeters from the front wall or
 * ERR_WALL_DISTANCE_NO_AVAIL if not available (either too close or too far).
 */
float getDistanceFront_mm();

/**
 * The left motor moves forward modulated by ‘intensity‘.
 * @param intensity Between 0 (motor does not drive) and 1 (motor drives at
    full speed)
 * @return ERR_OK if the operation was successful, an error code otherwise
 */
int setMotorLeftForward(float intensity);
```

This API is in turn implemented via a more traditional set of HAL APIs like those shown (for example) in “pwm.h”.

Additionally, our HAL layer performs the conversion from sensor readings to a distance measured in SI units. The current implementation uses a list of measured sensor outputs and the corresponding measured distance that was obtained during an initial sensor calibration phase. The HAL then performs a piece-wise linear interpolation of the curve defined by these points to convert from arbitrary sensor readings to distances in SI units. Any sensor reading that is smaller than the smallest value or larger than the largest value available in such list is discarded and reported as invalid.

4.3 Controller Design

In order to achieve precise control of our Micromouse, we needed to implement controllers. Those are agents which can automatically adjust commands of actuators to produce desired system outputs.

4.3.1 Control theory

Feedback controllers are a type of controller which use information feedback from sensors to generate control commands [25]. As shown in Figure 16, a feedback controller can be divided into three major components:

- Controller, where control commands are issued
- System, where commands are being executed
- Sensor, where control outcomes are measured and evaluated. This information will loop back to Controller on the next command generation

According to different modeling principles, feedback controllers can be separated into two types: model-based and model-free controllers. Model-based means using mathematical models to describe System and Sensor behaviors. This enables the analytical computation of the optimal parameters for the controller, which spares efforts for parameter tuning.

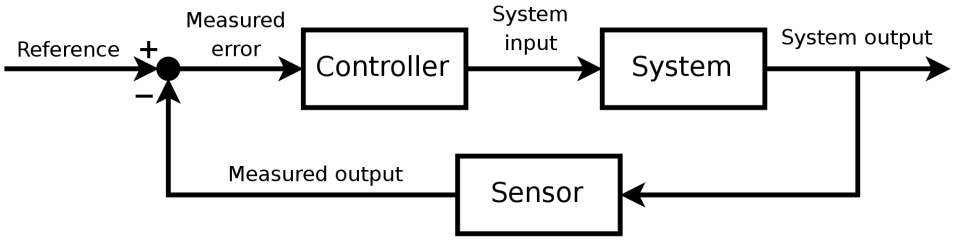


Figure 16: Major components of a feedback controller [taken from 25].

However a precise mathematical modeling is not always achievable. In some situations, we do not have any knowledge about the System we control, thus model-free controllers are useful. Model-free controllers keep track of errors between desired sensor values and real sensor values. By applying certain arithmetical operations on errors, the next control command can be derived. Based on errors, model-free controllers adjust their commands accordingly. Model-free controllers do not require prior knowledge on System or Sensor.

4.3.2 PI controller

The proportional–integral (PI) controller is known as a model-free feedback controller. It is a simplified version of the proportional–integral–derivative (PID) controller by omitting the derivative part of its original design. A PID controller has the structure shown in Figure 17. At moment t, the error between desired sensor value and real sensor value $e(t)$ are sent into proportional (P), integral (I) and derivative (D) gates for control gain calculation.

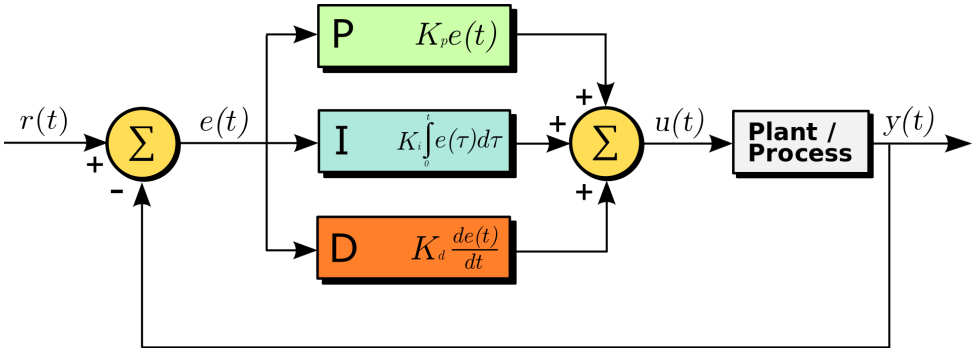


Figure 17: Structure of a PID controller [taken from 29].

In the P gate, the error $e(t)$ is multiplied with the parameter k_p to yield a proportional control gain. The error is integrated in the I gate and differentiated in the D gate before being multiplied respectively with parameters k_i and k_d. Finally, all portions of control gains are summed together. k_p, k_i and k_d are controller parameters which can be tuned to achieve the desired control performance.

The P gate is useful for shrinking the error, however using merely the P portion control gain will suffer from steady state error, which is the small constant error that cannot be compensated. The I portion helps to reduce steady state error and the D portion damps the system to enable a faster converge to the desired value.

With above considerations, we chose to implement a PI controller for our Micromouse. We realized it by creating a struct holding all necessary parameters:

```
typedef struct{
    float kp; // P gate parameter
    float ki; // I gate parameter
    float e_sum; // integration of error
    float sp; // set point (desired outcome)
    int enable; // 1: enable controller ; 0: disable controller
} PI;
```

Besides the formerly introduced parameters k_p and k_i, we hold `e_sum` as the accumulated error of the I gate and `sp` for set point. Moreover, we reserve `enable` to change the working status of the controller. Finally, we define a function to automatically calculate and accumulate the error on new data received:

```
/* Function: step PI controller and return control gain on receiving new input
    data
 * if controller disabled, pause error accumulation, return 0*/

float _stepPI(PI *controller, float pv){ // pv stands for present sensor value

    if (controller->enable == 1){
        // calculate current system error
        float error = controller->sp - pv;

        // update I portion of controller
        controller->e_sum = controller->e_sum + error;

        // calculate and return control gain
        float adjust = controller->kp * error + controller->ki *
            controller->e_sum;
        return adjust;
    } else {
        return 0; // if controller disabled return 0
    }
}
```

4.3.3 Controller system design

The ultimate control task is to enable a collision-free movement of the Micromouse inside the maze and to precisely perform rotations. To break this goal down, we need distance controllers to adjust the velocity on wheels so that the Micromouse avoids collision with walls. To achieve that desired velocity, we need velocity controllers adjusting the motor via PWM.

As shown in Figure 18, we planned a cascade structure of controllers. The velocity controllers are embedded inside distance controllers, hence the name “cascade”. First, velocity commands will be slightly tweaked by the distance controllers. Then, the velocity

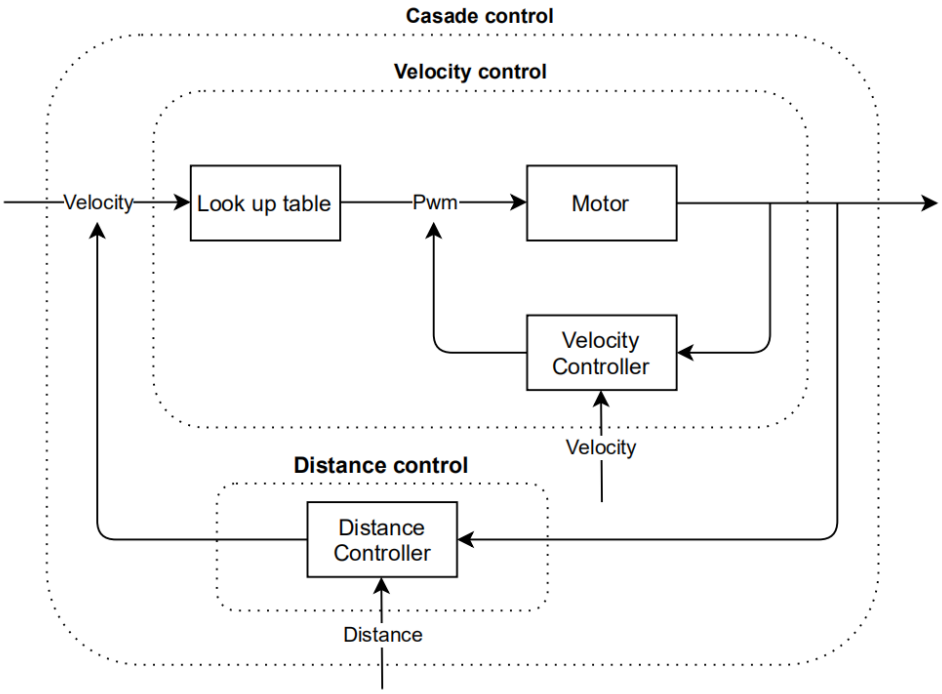


Figure 18: Work principle of cascade control.

controllers will take over and adjust the PWM module to reach that speed.

We initialize two PI controllers for velocity control, one for each motor. And three PI controllers are reserved for distance control, one for each sensor. This structure is achieved by the following code:

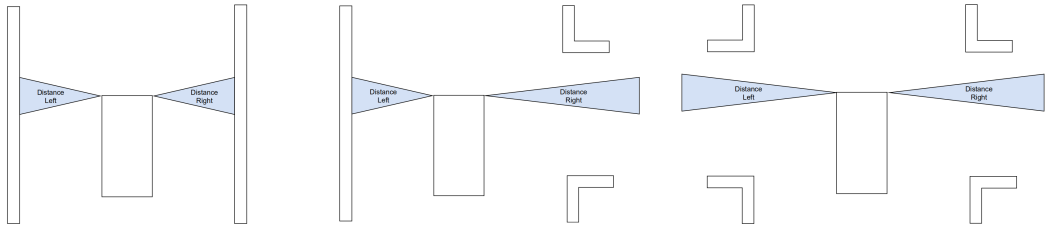
```
typedef struct{
    PI R; // Velocity controller Right motor
    PI L; // Velocity controller Left motor
} velocityControllers;

typedef struct{
    PI R; // Distance controller Right sensor
    PI L; // Distance controller Left sensor
    PI F; // Distance controller Front sensor
} distanceControllers;
```

4.3.4 Distance control

Distance controllers help the Micromouse to keep constant distances to the walls to the robot's sides while moving straight. That's why they are always disabled while rotating. To adapt to various maze scenarios, we developed different strategies.

There are three scenarios: corridor scenario, one-side opening scenario and both-sides opening scenario. The Micromouse analyzes the sensor values to determine the current scenario-type. It takes values smaller than 150mm as valid distance values for detected

walls. This makes sense as the maximum distance between the mouse and a cell wall (assuming that the current cell has a wall) is definitely less that 150mm, given that the width of the mouse is 81mm and the maximum width of a corridor with walls on both sides is 168mm as by the rules [31].



(a) corridor scenario (b) one-side opening (c) both-sides opening

Figure 19: Plots of three straight moving scenarios.

Corridor scenario: as shown in Figure 19a, the Micromouse's right and left sensors both return valid values smaller than 150mm, which indicates there exist walls on both sides. The set point of both distance controllers will be:

$$sp = \frac{\text{maze inner width}}{2} - \frac{\text{width front sensor}}{2} = \frac{168\text{mm} - 25\text{mm}}{2} = 71.5\text{mm} \quad (7)$$

One-side opening scenario: as shown in Figure 19b, if only one sensor returns valid values, it indicates that on the other side there exists an opening. In this case we set our distance controllers to keep constant distance from the remaining wall.

Both-sides opening scenario: as shown in Figure 19c, if both left and right sensors indicate no walls, the Micromouse is in a both-side opening scenario. In this scenario, we control our mouse to achieve same velocity on both wheels in order to drive a straight line.

The logic is implemented by following code:

```
float sp = 71.5;
if (validR && validL){ // corridor scenario
    _setPI(&cset->D.L,sp);
    _setPI(&cset->D.R,sp);

    _enablePI(&cset->D.L);
    _enablePI(&cset->D.R);
}
else if (validR && !validL){ // one-side opening: only right wall
    _setPI(&cset->D.R,sp);

    _enablePI(&cset->D.R);
    _disablePI(&cset->D.L);
}
else if (validL && !validR){ // one-side opening: only left wall
    _setPI(&cset->D.L,sp);

    _enablePI(&cset->D.L);
```

```
    _disablePI(&cset->D.R);
}
else if (!validL && !validR){ // both-sides opening
    _disablePI(&cset->D.R);
    _disablePI(&cset->D.L);
}
```

4.3.5 Velocity control

The velocity controllers are responsible for achieving the desired speed on the wheels. After a velocity is set by the distance controllers, the final desired velocity will be passed to the lookup table, which determines the PWM intensity that should be applied to the motors. In the following iteration, the encoder values are supplied as feedback to the velocity controllers in order to adjust again the control gain. To enable the precise control of both motors, we define two velocity controllers, one for each motor.

PWM interpolation Firstly, we build a lookup table for our motors. We consider two solutions for table construction:

- **Solution 1:** construct lookup table by sampling points and interpolating them. A possible outcome is illustrated in Figure 20 using the red characteristic line.
- **Solution 2:** do complex motor modeling of motor behaviors, try to exploit the relation between PWM and velocity analytically. A possible result is illustrated in Figure 20 as the black characteristic line.

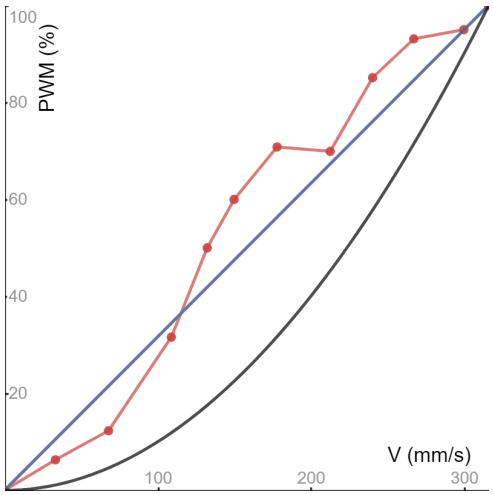


Figure 20: PWM-Velocity curve.

We decided the first option to be our solution. By sampling along the curve, we get values under real working situation of motors. In contrary, solution 2 is not based on testing. We fear that the idealistic modeling assumption could be oversimplified and may not adapt to real working situation.

However, due to the Covid-19 pandemic, we are not able to build the Micromouse before writing this report. As a result, we cannot sample points in the lab. We therefore calculate

a sample point based on the motor's datasheet [14]. Our result is illustrated in Figure 20 using the blue line.

We assume the maximal motor turning speed is achieved at $PWM = 1$. Reading from the motor data sheet, we know our motors, with a $33:1$ reduction ratio, have a maximal velocity of $n_{\max} = 151$ rpm. Our wheel is the 40mm diameter POLOLU1452 [5].

$$v_{\max} = n_{\max} \cdot \pi \cdot d = 151 \text{ rpm} \cdot \pi \cdot 40\text{mm} \approx 316 \frac{\text{mm}}{\text{s}} \tag{8}$$

To provide a general approach, we formulate sample points in form {velocity, PWM}, which is implemented as:

```
typedef struct {
    float velocity;
    float PWM;
} PWMCurvePoint;
```

In our current case, we have two points: {0,0} and {316,1}. We developed a general approach for points interpolation of arbitrary amount. We realize PWM interpolation using the code below:

```
// declare sample points
#define N_POINTS_PWMCurvePoint 2
PWMCurvePoint samplekurve[] = { { 0, 0 }, {316, 1 },};

float _interpolatePWMCurve(const PWMCurvePoint *curve, int N_points, float
    desiredVelocity) {

    // allocate the range desiredVelocity falls in
    int pointIndex, i;
    for (i = 0; i <= N_points - 2; i++) {
        if (curve[i].velocity <= desiredVelocity && curve[i + 1].velocity >
            desiredVelocity){
            pointIndex = i;
            break;}}

    PWMCurvePoint head = curve[pointIndex]; // nearest front sample point
    PWMCurvePoint tail = curve[pointIndex + 1]; // nearest tail sample point

    // return proportional location to front
    float delta = (desiredVelocity - head.velocity) / (tail.velocity -
        head.velocity);
    // return interpolation result
    return head.PWM + delta * (head.PWM - tail.PWM);
}
```

Speed and distance on wheel Using the above interpolation method, we read out the open loop PWM value. For velocity controllers to take effect we still need a measurement

of current velocity on wheels as feedback.

We retrieve the velocity of wheels by reading the motor encoder values. We declare following parameters for calculation:

- $N_{\text{pulse}} = 16$, the number of encoder pulses per full motor revolution
- $K_e = 4$, the multiplier of the QEI measurement mode that is used in our mouse
- $Gr = 33$, the gear ratio of the motor gearbox
- $D_{\text{wheel}} = 40\text{mm}$, the wheel diameter
- V_{cps}, the number of encoder ticks per second (s^{-1})
- f, the PI update timer frequency in Hz
- α, the angle per encoder tick in degrees
- ω, the angular velocity of the wheel in $\frac{^\circ}{s}$
- V_{linear}, the linear velocity of wheel in $\frac{\text{mm}}{\text{s}}$

We calculate the difference between current encoder count and last encoder count at each timer interrupt. By multiplying it with the timer frequency, we compute change the rate of encoder count. Finally, the angular velocity and wheel velocity can be derived. Following formulas are used for calculation:

$$V_{\text{cps}} = (Count_{\text{current}} - Count_{\text{last}}) \cdot f \tag{9}$$

$$\alpha = \frac{360}{N_{\text{pulse}}} \tag{10}$$

$$\omega = \frac{V_{\text{cps}} \cdot \alpha}{K_e \cdot Gr} \tag{11}$$

$$V_{\text{linear}} = \frac{\omega}{180} \cdot \pi \cdot \frac{D_{\text{wheel}}}{2} \tag{12}$$

We calculate the distance travelled on wheels between two encoder counts similarly. Such functionalities are implemented in "controlUtils.h":

```
0 // compute wheel speed (mm/s) according to QEI counts
  float _getWheelSpeed(float lastCount,float currentCount,int timerFrequency);

  // compute wheel distance (mm) according to QEI counts since command start
  float _getWheelDistance(float startCount,float currentCount);
```

Now we have all the pieces we need for velocity control. We implement the velocity controllers according to the block diagram shown in Figure 21. The following code computes the final PWM values:

```
0 //look up PWM without control
  float lookupPWMR = _lookupPWM(cset->desiredVelocityR);
  float lookupPWML = _lookupPWM(cset->desiredVelocityL);

  //calculate PWM adjustment base on current wheel speed
```

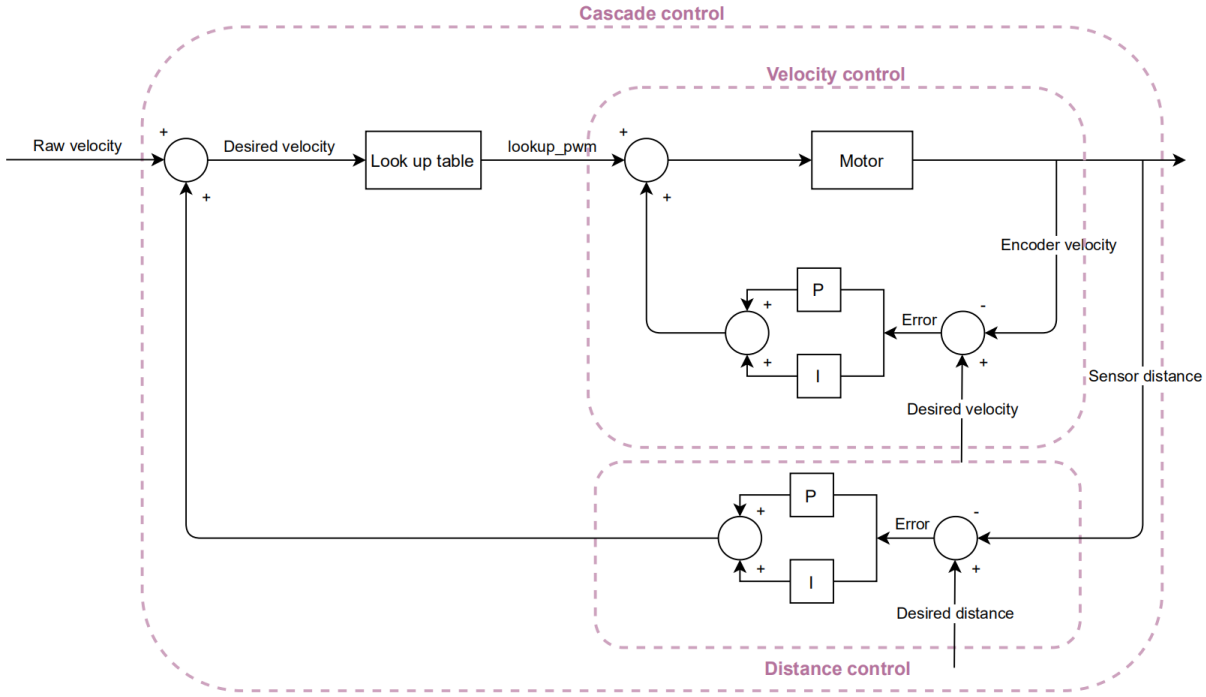


Figure 21: Block diagram of a cascade controller.

```
float adjustR = _stepPI(&cset->V.R, wheelSpeedR);
float adjustL = _stepPI(&cset->V.L, wheelSpeedL);

//calculate final PWM after control
float finalPWMR = lookupPWMR + adjustR;
float finalPWML = lookupPWML + adjustL;
```

4.3.6 Motion control

The Micromouse needs to be able to perform two motions: move and rotation. Move is running a straight line and rotation corresponds to orientation change. To enable more flexibility for rotation, we developed two motion models under rotation: turn and spin. Turn enables the Micromouse to rotate while still moving. Spin enables it to rotate on the spot.

We implemented the above ideas by providing three API calls, each corresponds to one of the three motions: move, turn and spin. The calls can be found in the "motionControl.h" file:

```
/* Go straight line
 * both motors set to rawVelocity(mm/s) */
void move(Controllerset *cset, float rawVelocity);

/* Rotate while running
 * outer wheel set to rawVelocity(mm/s), inner wheel stop
 * +: turn right
 * -: turn left */
```

```
void turn(Controllerset *cset, float rawVelocity);

/* Rotate on the spot
 * both motors set to rawVelocity(mm/s), but reversed to each other
 * +: turn right
 * -: turn left*/
void spin(Controllerset *cset, float rawVelocity);
```

MOVE By calling `move()`, we issue the same velocity commands on both wheels. Velocity commands will then pass to distance controllers for adjustment and executed on motors using PWM generated by velocity controllers. While executing move motion, `move()` the API also keeps track of distance traveled since the command has been executed. We compute distances on left and right wheels by calling `_getWheelDistance()` introduced above.

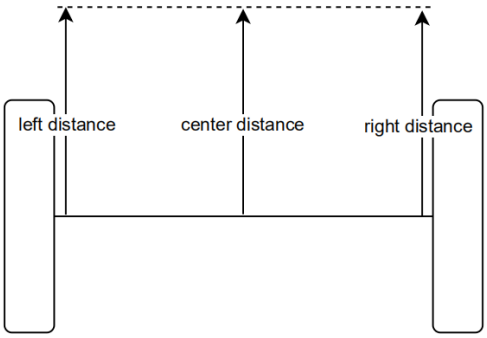


Figure 22: Motion model: move.

Due to disruptive factors (e.g. command adjustments from controllers, control error, sensor error etc.), even if the mouse is moving straight, the distance traveled by the left and right wheels can be different. To achieve a normalization effect, we calculate center distance traveled by averaging the left and right distances as shown in Figure 22:

$$D_{center} = \frac{D_{left} + D_{right}}{2}$$

TURN By calling `turn()`, we set velocity on the outer wheel and brake the inner wheel. In case of turning right, as shown in Figure 23, left wheel is the outer wheel and right wheel is the inner wheel. By keeping track of distance traveled on outer wheel, we can compute the angle α (in degree) turned by our Micromouse using below formula:

$$\alpha = \frac{D_{outer}}{\pi \cdot w_{track}} \qquad w_{track} = 74\text{mm}$$

Notice: the track width from our Micromouse design is about 74mm.

SPIN By calling `spin()`, we set same velocity on both wheels but in reversed direction. In case of spinning right, as shown in Figure 24, left motor will be set to move forward and

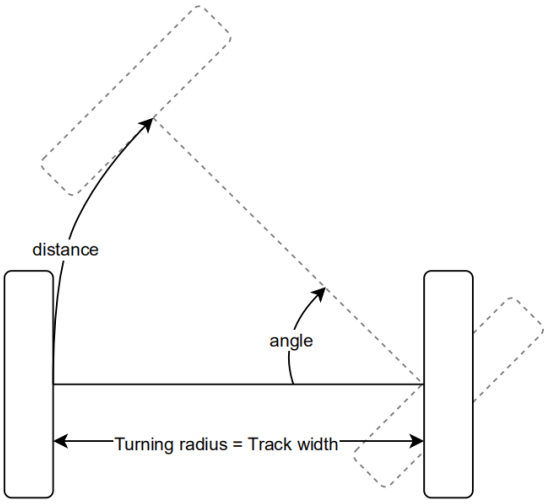


Figure 23: Motion model: turn.

right motor will be set to move at same speed backward. Similar as before, we compute the rotation angle α (in degree) using the following formula:

$$\alpha = \frac{D_{outer}}{\pi \cdot 0.5 \cdot w_{track}} \qquad w_{track} = 74\text{mm}$$

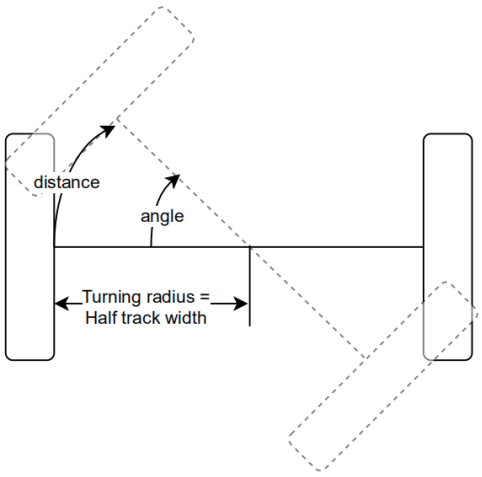


Figure 24: Motion model: spin.

Notice: the turning radius of `spin()` is half the track width.

4.4 Direction Control

This module is the transition point between the lower level modules, which do not "know" anything about maze cells, and the higher level work which is done grid-based. Every module from this point forward forms an event-based system, and does not have any direct tie to the notion of timer interrupts.

The direction control keeps an internal state of the robot, tracking all necessary data for the higher-level movement control unit. This state is implemented via the following struct:

```
typedef struct
{
    // An enum containing the movement states IDLE, MOVE or TURN
    movementState movState;
    // wall placement sensed from current position
    uint8_t walls;
    // latest 2D cell position in maze
    position curPos;
    // the cardinal direction given by the mazeSolver to move next
    dir nextDirection;
    // last known cardinal direction of the robot
    dir curDirection;
    // distance from last cell to the point of latest wall change detection
    float distanceSinceLastWallChange;
} robotState;
```

To showcase the direction control's functionality, the program sequence is demonstrated. On each timer interrupt, the distance controller sends an update signal with the following data:

```
// boolean, whether the sensors (R=right, L=left, F=front) detect a wall
BOOL sensorR;
BOOL sensorL;
BOOL sensorF;
// the average of both motor encoder deltas since latest movement call (in mm)
float distanceDelta;
// the angle difference of the robot since latest movement call (in degrees)
float angleDelta;
```

Upon receiving this pack of information, the direction control works through a decision tree. This can be seen in Figure 25.

First, the module's action depends on its current movement state. These can either be IDLE (meaning the robot is not moving or otherwise achieving anything in terms of its goal), MOVE (this state represents the robot moving in a straight line), and TURN (meaning the robot is spinning on the spot).

We consciously do not divide between turn and spin anymore, as we first wanted to test the spinning motion before investing time into performing a turn motion. As we weren't able to even test the prior, we settled for not implementing a turn-motion further (except for the work that had already been done on the controller-side). So from now on, "turn" actually means spinning, as in rotating while staying on the spot.

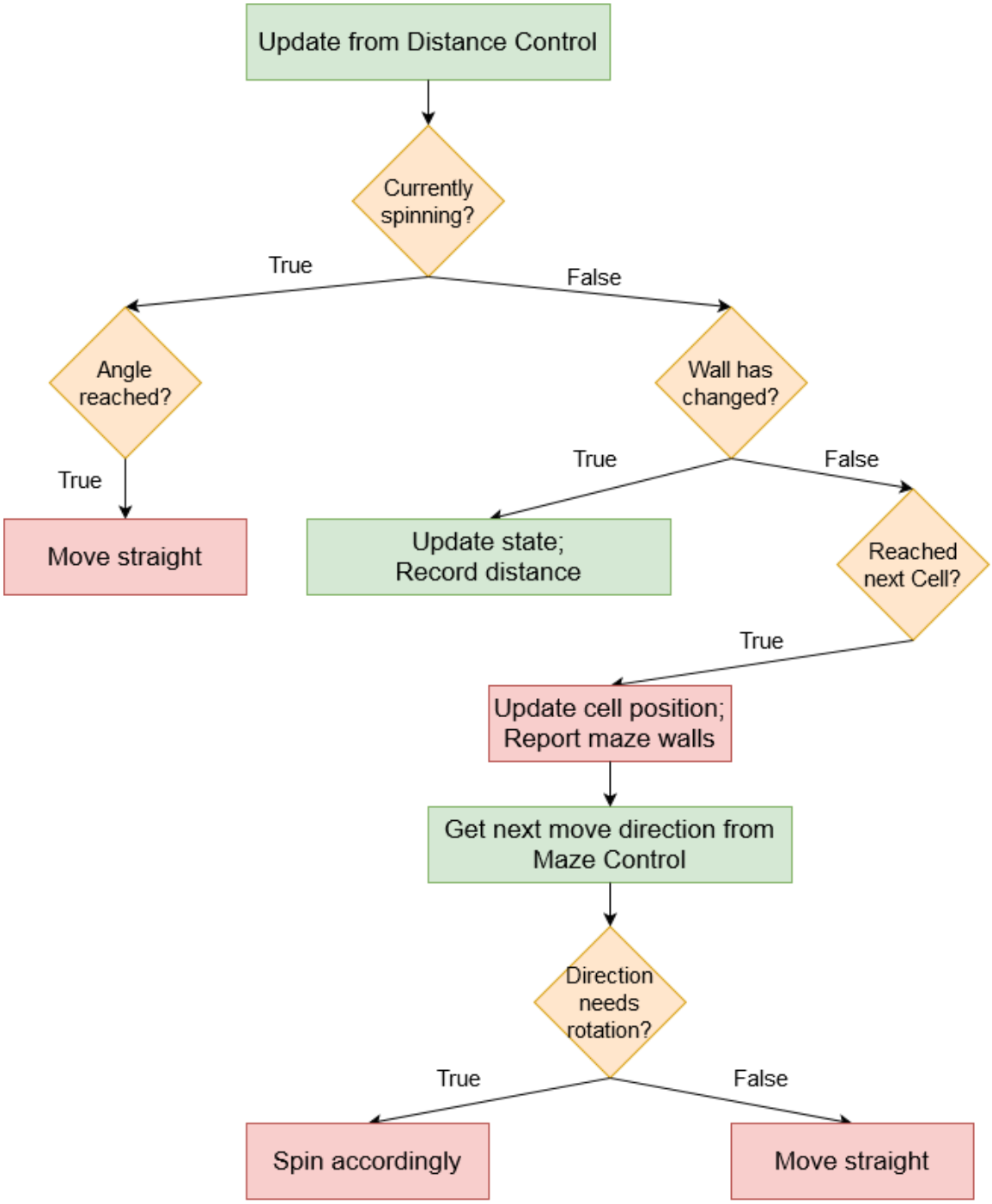


Figure 25: The decision map of the direction control module.

4.4.1 IDLE

If the direction control is currently in IDLE mode, it does nothing. It only leaves this state if a higher level module sends a corresponding signal. When it leaves IDLE, the module sends a signal to the maze solver to receive the next direction the robot should move to. Depending on whether the robot is already facing that goal direction or not, it transitions into the MOVE or TURN state.

4.4.2 MOVE

As the robot is in the MOVE-state (i.e., moving straight), it will check whether the detected walls have changed. If the sensors have picked up a different combination of sensed walls, the direction control saves the `distanceDelta` in that update call. It stands for the distance the robot has moved since the last cell up to the point where the walls have changed (i.e., the front part of the robot, where the sensors are placed, has changed cells).

Then, it is evaluated whether the whole robot has changed cells or not (which is defined by the robot being completely in the new cell). There are two significant conditions that each would reason a cell change. Either the measured total distance passed since the previous cell (represented by the newest `distanceDelta`) is greater than a specified constant, or the distance passed since the latest wall change detection is greater than a specified, lower value. Both constants are set with some margin of error to allow for noise in the motor encoder's data.

When the direction control decides that the robot has changed cell position, it updates the corresponding internal state and sends a signal to the maze control, which returns the next goal direction. Depending on the relation of that direction and the robot's current one, the direction control either stays in MOVE or changes into the TURN-state.

4.4.3 TURN

Whenever the robot is currently spinning, the module receives the `angleDelta` sent by the distance controller and compares it with the relative angle between its old and goal direction. If they are equal; i.e., the robot has reached the right direction, the module changes to the MOVE-state, sending a corresponding call to the distance controller (after a spin, the robot always needs to head down straight to actually reach the next cell).

4.5 Maze Map

While the Micromouse is exploring the labyrinth, the direction control manages the localization of the robot. But for the Micromouse to work properly, the mazes' walls need to be known, as well as which cells are already explored. The maze map module keeps track of those properties. This is done by constructing a 2D list of cells, containing the following properties.

```
0 typedef struct {
```

```
    // determines whether a cell has been explored by the robot
    BOOL visited;
    // an 8 bit integer, where the least significant four bits determine wall
       placement within that cell
    uint8_t wall;
} cell;
```

Walls are saved by utilizing a bit-field approach. Each cardinal direction has its own bit within the 8 bit integer, as the following code fragment shows.

```
    NORTH = 0b0001,
    EAST = 0b0010,
    SOUTH = 0b0100,
    WEST = 0b1000
```

With that, many maze map functionalities can be implemented quite run-time effective. However, there is one drawback. As the module saves walls for each cell in each of the four cardinal directions, there is quite a lot of redundancy. For example, at position `(1, 1)`, a wall to that cells' `EAST` is equal to a wall `WEST` from the coordinates `(2, 1)`. This means each time a maze wall is set, two entries in the maze list need to be updated. As an example, the function to add one maze wall at a certain position `pos` and cardinal direction `direction` becomes:

```
void addMazeWall(position pos, dir direction)
{
    getCellAt(pos)->wall |= direction;
    // need to add wall at the neighbor cell in direction with the inverse
       cardinal direction.
    getNeighborCellInDir(pos, direction)->wall |= getInverse(direction);
}
```

Checking whether a wall exists at a certain position and direction can intuitively be implemented via the following check:

```
// Standard bitfield compare via '&' (AND)
(getCellAt(pos)->wall & direction) == direction
```

Despite the redundancy, we think that this method is quite simple and effective, and therefore decided to not improve it further.

4.6 Maze Control

The maze control is the highest level module in our software design. It is responsible for the macro-level solution. For the Micromouse, this is essentially solving the maze. The base idea is the following: whenever the robot arrives at a new cell (detected by the direction control, see subsection 4.4), it sends a signal with the new position to the maze control. The maze control then decides which cardinal direction the robot should move to next, and sends that back.

There are two unique modes for the maze control: the exploration, and the race phase.

During the exploration phase, the robot is, as the name suggests, supposed to find out where each wall of the maze is placed. This will also lead to the ulterior goal, that is finding a path from the start position to the center of the maze. For this first minimal working prototype, we decided on the "hug-right wall"-algorithm (also known as "right-hand rule"). Within this simple function, the next direction of the robot depends only on the current wall placement and direction. The robot will stick to its current right wall, driving along it. This has some flaws and could, for some mazes, end up in a loop, but we considered it not a priority concern. In later development, of course, this needs to be addressed by implementing an advanced exploration algorithm.

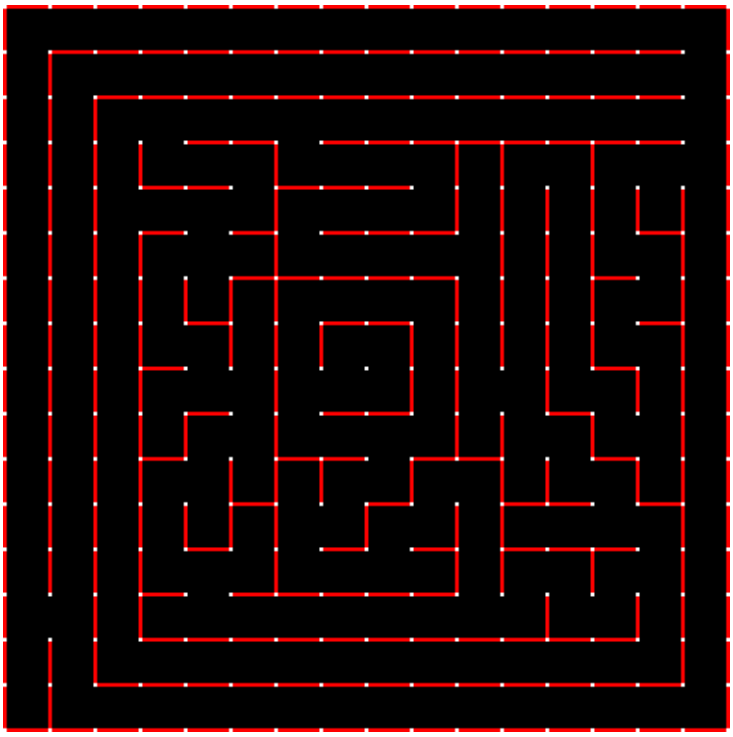


Figure 26: The official micromouse maze from the 2018 Applied Power Electronics Conference [taken from 24].

As soon as the robot reaches one of the goal states, which consist of the four center spaces of the maze, the quickest path from start to finish needs to be computed. Initially, we wanted to program a simulator in Python for this, to evaluate different approaches (like punishing rotations, moving diagonally etc...) and to find the optimal solution. However, two factors changed our mind: the time to develop such tools seemed to be out of proportion for this project (even though the rough structure of the simulator has already been implemented). Second, looking at some of the official Micromouse challenge mazes (for an example, see Figure 26), it seemed like many of our ideas for more efficient algorithms to be in vain. There is seldom room for diagonal movement. In general, there seems to be only one obvious best path, as each different one is much longer. This would render an decision even partly-based on numbers of rotations completely worthless.

In conclusion, the number of cells; i.e., the length of the path, is the most important and only criteria we settled for. Realizing that, we chose a recursive implementation of the

standard floodfill algorithm to find the quickest path from start to finish, as it is simple and easy to implement [27].

The floodfill algorithm is initialized by assigning each cell a cost of `maze_weight` × `maze_height`. It represents the maximum cost to reach each arbitrary cell regardless of wall position (assuming there is a valid path to each cell). Only the goal states in the center of the maze receive a cost of zero.

When the robot is done exploring; i.e., the quickest path needs to be found, the floodfill algorithm is called from each of the four goal position. A pseudo-code of our implementation is as follows:

```
0  void floodFill(position pos)
   {
       // iterate over each cardinal direction
       foreach (direction dir):
       {
5          // get the position in that direction
           position positionInDirection = getPosInDir(pos, dir);
           // there must be no wall obstructing the path, the node need to have
               been explored and its current cost worse
           if (noWallAt(pos, dir) AND isExplored(positionInDirection) AND
               costAt(positionInDirection) > costAt(pos) + 1)
           {
10             setCostAt(positionInDirection, costAt(pos) + 1);
               // recursive call from each position that fulfilled the conditions
                   above
               floodfill(positionInDirection);
           }
       }
15 }
```

This algorithm results in a cost assigned to each explored cell equal to the distance of that cell to the nearest goal position. For instance, a cost of five means that the robot needs to pass at most five cells to reach the closest center position.

The race-mode itself then becomes quite easy. As before, the direction control asks the maze control for the next direction. The maze control computes it by looking at the cost in each direction not obstructed by a wall and returns the one with the lowest, ensuring the robot takes the quickest path known.

We currently do not handle any form of uncertainty on this level. Other improvements can also be made. For instance, one could evaluate the worth of driving into unexplored cells by their euclidean norm [26], caused by the explored path being unlikely long.

These ideas should be considered in the future and were only not done due to the constrained time.

4.7 Commanding and telemetry

As stated in section 2, our mouse uses UART (either wired or via Bluetooth Low Energy) to communicate with the outside world. Such communication happens in two directions: the mouse can receive commands from the external world or send telemetry to it.

The command reception is implemented in the HAL layer, and in particular in the "uart.c" file. Here, the characters received by the microcontroller are analysed to find strings in the form `<command>`. Whenever such a string is obtained, the callback `onCommandReceived` that is defined in "uart.h" is invoked with the `command` that was received as a parameter. This callback is implemented in "main.c" and, depending on the content of the `command` invokes the appropriate methods in "mazeControl.c" to modify the robot state. In order to accomplish that, a hash function was implemented. It assigns each command string a corresponding integer. Though our hash function is limited to five characters that can only consist of the letters A to Z (no lowercase), it was well sufficient for our purposes.

Command	Effect
`<LED>`	Toggles a specific LED (for debugging).
`<SPIN>`	Overrides movement control by ordering to spin (for debugging).
`<GO>`	Activates the direction control, and therefore, the whole process.
`<STOP>`	Stops the direction control, halting the robot.
`<PAUSE>`	Pause/Unpauses the direction control.
`<CLRS>`	Executes a soft reset on the higher level modules (resetting internal position, explored maze walls...)
`<CLRH>`	Executes a hard reset, re-initializing each module.
`<MAP>`	Sends the current maze map over the UART.

Table 2: The list of all commands accepted by our UART module.

A list of all commands we implemented can be found in Table 2. The commanding protocol is rather simple and straightforward, yet covers all of our requirements.

Every few timer interrupts the mouse sends back some telemetry data regarding its current status and parameters like the current goal cell and wall presence/absence as detected by the sensors. This data is sent as ASCII characters that can then be visualised on a laptop.

While the communication via wired UART is handled through Cutecom [17], the communication via BLE requires a different solution. Since the GATT Characteristics exposed by the Adafruit module are custom, we needed to write a small Python script that uses the `bleak` [2] Python module to read from and write to these custom GATT Characteristics. Adafruit does offer an Android app, which we planned to use in initial testing of the module functionality, but it does not allow for an easy export (e.g. to a laptop) of the data received, therefore we needed to develop our own communication software.

5 Tests

Due to the Coronavirus pandemic and the delays we faced in receiving the parts from Mouser, we were not able to perform tests on our Micromouse nor were we able to obtain a final working prototype. Nevertheless, we discovered a design error in our board that we discuss in subsection 5.1.

5.1 The design error

During the software implementation phase of the project, after we had already submitted the board design to the manufacturer, we identified a significant design error in our board. While designing the mouse's wiring schematic, we were convinced that PWM pins on our microcontroller were remappable, but they are not. This lead us to having only one motor (one of the two inputs of the H-Bridge) connected to actual PWM pins: the other one was connected to two reprogrammable pins (`RP23` and `RP24`), which do not have PWM capabilities.

After noticing this problem we quickly identified three solutions:

- **The hardware fix**. From the piece of the board shown in Figure 27, we can see that pin `PWM1H1` and `PWM1L1` are unused and physically close to the H-Bridge input connected to the `RP23/RP24` pair. It should be possible to run wires from the two `PWM1X1` pins to the H-Bridge input and then either configure `RP23/RP24` to be in a high impedance state or physically disconnect them by scratching the copper trace. In this way, we would be able to completely fix our design error.
- **The software fix**. Although PWM pins are not remappable, output compare (OC) pins are. Moreover, one of the modes of the OC pins allows them to behave as PWM pins by using one of the microcontroller's timers as a clock source. Therefore, if the hardware fix is not applicable from some reason, we plan on driving the H-Bridge input by mapping `RP23/RP24` to the `OC1` and `OC2` peripherals, both using `Timer 3` as a clock source. In this case, `Timer 3` would be programmed to run at the same frequency of the `PWM1` generator. By default, the mouse software has been programmed as if we applied the hardware fix. By defining the `SOFTWARE_FIX` macro during compilation, the HAL changes and uses the software fix instead (the API exposed to higher levels remains unchanged). If this solution is chosen over the hardware fix, the mouse should still be fully operational, but it would consume more energy as an extra timer and peripheral are needed with respect to the ideal situation.
- **The last resort**. If, for some reason, both the hardware fix and the software fix fail, we though about a third solution. We observe that pin `RP23` is actually connected to a PWM pin from the `PWM2` unit and that `RP24` is a GPIO pin. By correctly configuring the `PWM2` unit and by using the GPIO capabilities of `RP24` we can still drive the H-Bridge input, although in a limited manner, as we would not be able to select the " Forward PWM, slow decay" and "Reverse PWM, fast decay", but we would still be able to use "Forward PWM, fast decay" and "Reverse PWM, slow decay". Although sub-optimal, this would still allow to drive the mouse, although in a less precise way.

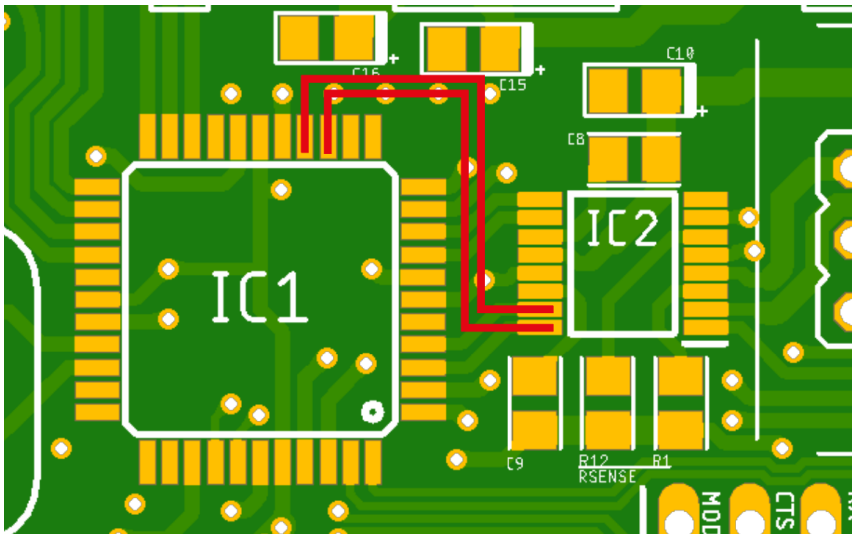


Figure 27: The wires that need to be applied for the hardware fix

In order of preference, we think that the hardware fix is the most preferable solution, followed by the software fix. We don't think that the third method will be needed at all, and we reported it only for completeness.

6 Conclusion

All in all, we successfully managed to design a Micromouse robot from the ground up. For many of us this was the first experience in designing a non-trivial embedded system, and we all managed to learn a lot, from PCB design and 3D printing all the way up to high level control implementation.

Unfortunately the pandemic limited what we could achieve regarding the practical implementation within the given time constraints. We weren't able to actually build the robot, meaning there weren't any practical tests that we would be able to conduct. Although already aware of certain possibilities for improvements, some trial runs would have surely improved both our design and implementation.

Nevertheless, we still consider the project successful and plan to invest some more time on it during the summer semester, assuming that the overall situation regarding the pandemic will allow for it.

References

[1] Adafruit. *Adafruit 2479 UART Service details*. 2021. URL: https://learn.adafruit.com/introducing-adafruit-ble-bluetooth-low-energy-friend/uart-service.

[2] Henrik Blidh. *Bleak*. 2021. URL: https://pypi.org/project/bleak/.

[3] Ramon Cerda. *Pierce-gate oscillator crystal load calculation*. 2004. URL: https://www.crystek.com/documents/appnotes/PierceGateLoadCap.pdf.

[4] D. Christiansen. "Spectral lines: Announcing the Amazing Micro-Mouse Maze Contest". In: *IEEE Spectrum* 14.5 (1977), pp. 27–27. DOI: 10.1109/MSPEC.1977.6367601.

[5] TME electronic components. *POLOLU-1452 wheels 40mm*. URL: https://www.tme.eu/de/details/pololu-1452/robotik-und-rc-zubehor/pololu/pololu-wheel-40x7mm-black/.

[6] Sharp Electronics. *GP2Y0A41SK0F Datasheet*. URL: https://global.sharp/products/device/lineup/data/pdf/datasheet/gp2y0a41sk_e.pdf.

[7] Sharp Electronics. *GP2Y0A51SK0F Datasheet*. URL: https://global.sharp/products/device/lineup/data/pdf/datasheet/gp2y0a51sk_e.pdf.

[8] J.S.T. Deutschland GmbH. *PH Connector*. URL: https://jst.de/product-family/show/119/ph.

[9] J.S.T. Deutschland GmbH. *ZR Connector*. URL: https://jst.de/product-family/show/103/zr.

[10] Rust embedded HAL team. *Embedded-hal Rust crate*. 2021. URL: https://github.com/rust-embedded/embedded-hal/tree/v0.2.4.

[11] Autodesk Inc. *Fusion 360 - Integrated CAD, CAM, CAE, and PCB software*. 2021. URL: https://www.autodesk.com/products/fusion-360.

[12] Texas Instruments. *DRV8833 Datasheet*. 2015. URL: https://www.ti.com/lit/ds/symlink/drv8833.pdf.

[13] Texas Instruments. *LM1117 Datasheet*. 2020. URL: https://www.ti.com/lit/ds/symlink/lm1117.pdf.

[14] Dr. Fritz Faulhaber GmbH & Co. KG. *APTD2012LSURCK Datasheet*. 2020. URL: https://www.faulhaber.com/fileadmin/Import/Media/DE_2619_SR_IE2-16_DFF.pdf.

[15] Kingbright. *APTD2012LSURCK Datasheet*. URL: https://www.kingbrightusa.com/images/catalog/SPEC/APTD2012LSURCK.pdf.

[16] United Kingdom Micromouse and Robotics Society. *Micromouse*. 2014. URL: https://ukmars.org/contests/.

[17] Alexander Neundorf. *Cutecom*. 2021. URL: https://gitlab.com/cutecom/cutecom.

[18] qu1ck. *Interactive HTML BOM plugin for KiCad*. URL: https://github.com/openscopeproject/InteractiveHtmlBom.

[19] STMicroelectronics. *VL53L0X time-of-flight ranging sensor*. 2021. URL: https://www.st.com/resource/en/datasheet/vl53l0x.pdf.

[20] Microchip Technology. *16-Bit 28-Pin Starter Development Board User's Guide*. 2008. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/51656B.pdf.

[21] Microchip Technology. *AN826*. 2002. URL: http://ww1.microchip.com/downloads/en/appnotes/00826a.pdf.

[22] Microchip Technology. *dsPIC33FJ64MC804 Datasheet*. 2012. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/70291G.pdf.

[23] Microchip Technology. *MPLAB PICkit 4 In-Circuit Debugger User's Guide*. 2018. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/PICkit4%5C%20In-Circuit%5C%20DebuggerUG-DS50002751B.pdf.

[24] Jeffrey Alan Weisberg. *Collection of Micromouse Mazes*. URL: http://www.tcp4me.com/mmr/mazes/.

[25] Wikipedia. *Control theory*. 2021. URL: https://en.wikipedia.org/wiki/Control_theory.

[26] Wikipedia. *Euclidean distance*. URL: https://en.wikipedia.org/wiki/Euclidean_distance.

[27] Wikipedia. *Flood fill*. URL: https://en.wikipedia.org/wiki/Flood_fill.

[28] Wikipedia. *Hardware Abstractions*. 2021. URL: https://en.wikipedia.org/wiki/Hardware_abstraction.

[29] Wikipedia. *PID controller*. 2021. URL: https://en.wikipedia.org/wiki/PID_controller.

[30] University of Würzburg. *RODOS (Realtime Onboard Dependable Operating System) Hardware Abstraction Layer*. 2021. URL: https://gitlab.com/rodos/rodos/-/tree/1204ea38ee2bc29dfaeeb51d61e4cde5bcd77d6e/api/hal.

[31] University of York. *Micromouse Maze Solver Rules*. 2013. URL: https://www.cs.york.ac.uk/micromouse/Rules/Maze_Solver_Rules.pdf.