

Janne Marie List

# Quadcopter for indoor mapping

Master's thesis in Embedded Computing Systems

Supervisor: Tor Onshus

June 2024

NTNU  
Norwegian University of Science and Technology  
Faculty of Information Technology and Electrical Engineering  
Department of Engineering Cybernetics





Janne Marie List

# Quadcopter for indoor mapping

Master's thesis in Embedded Computing Systems

Supervisor: Tor Onshus

June 2024

Norwegian University of Science and Technology

Faculty of Information Technology and Electrical Engineering

Department of Engineering Cybernetics



Norwegian University of  
Science and Technology



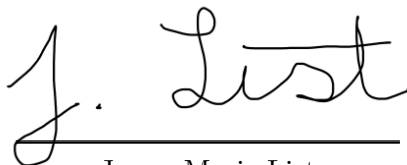
---

## Preface

This thesis concludes a master's degree as part of the European Masters in Embedded Computing Systems (EMECS) programme. During the two-year programme, lectures were attended and projects carried out at the University of Southampton and in the second year at the Norwegian University of Science and Technology (NTNU). This work is part of the course TTK4900 - Engineering Cybernetics, Master's thesis, which is awarded 30 credits - equivalent to a quarter of the degree. The project started in January 2024 and was submitted in early June 2024 after 20 weeks of work.

It is expected that the reader of this report has basic knowledge of computer science, especially low-level programming in C, and a basic understanding of electrical characteristics. The project is based on the nRF52840 system-on-chip microcontroller and follows on from previous work involving the OpenMV Cam M7, the ICM-20948 Inertial Measurement Unit and HC-SR04 ultrasonic sensors for distance measurements. These components as well as a workplace were provided for the duration of the project.

I would like to thank the European Union for funding this study programme and to express my gratitude for being selected as a recipient of the Erasmus Mundus scholarship. My special thanks also go to Tor Onshus - the supervisor of this thesis - for making this project possible and to Rune Mellingseter - an engineer in the electronics lab - for his help in merging the hardware and helping to solve problems with the electronics. Last but not least, I would like to thank my family and friends for their support throughout the duration of this project.

A handwritten signature in black ink, reading "J. List". The signature is written in a cursive style with a large, looped initial "J".

---

Janne Marie List  
Trondheim, June 2024

---

## Original problem description

The mapping of an indoor maze is carried out by the cooperation of ground robots and a quadcopter that hovers above the maze. For this purpose, an embedded camera module with integrated image processing to recognize the walls of the maze as well as an ultrasonic sensor system to measure the position of the drone has been developed previously. The map of the maze is created on a central server, where the ground robots and later the quadcopter will send their measured data.

The position of the quadcopter has previously been calculated by using merely ultrasonic sensors that measure the distance to the surrounding walls of the indoor room. The resulting estimate is quite accurate, but an IMU should be integrated to improve the position estimate based on orientation and acceleration data. The resulting position should be used as reference for the camera module to extract the position of the maze walls. The central server must be modified so that both the image data from the quadcopter as well as the data from the ground robots are taken into account when creating the map of the labyrinth.

- Integration of the existing software for the ultrasonic sensor and the camera module into one system to be used on a quadcopter
- Integration of an IMU into the system to improve the position estimate
- Modifying the communication of the quadcopter system and the server, where the quadcopter sends the position estimate and processed image data, and the server links the received data with the data from the ground robots to generate a joint map of the maze

---

## Summary and conclusion

This thesis is a contribution to the ‘Simultaneous Localisation and Mapping Project’, in which several robots map an indoor environment, or more precisely a maze. The aim was to develop a data processing module for a quadcopter to map the area from above. The module provides a pose estimate for the quadcopter and contains a camera to process images of the maze. It also enables communication with a server so that collaboration with other robots is possible and the processed data can be visualised on a map.

The module is based on the nRF52840 development kit from Nordic Semiconductors, which controls the connected sensors and communicates wirelessly with the server via Thread. Previous work includes a distance measurement system in which ultrasonic sensors are attached to a case to determine the distance to the environment. In addition, a camera module has also been developed that takes pictures on command and extracts the position of the edges of the objects it captures. The robot code that controls the other robots in the project was also available and was used as a basis to establish communication with the server.

After merging the two systems, the previous algorithms were adapted and their performance was improved. An inertial measurement unit (IMU) was integrated and a position estimation algorithm based on the sensor data was developed. The orientation was determined using different methods based on the IMU’s gyroscope data. The extended Kalman filter processed the distance data from the ultrasonic sensors and the orientation provided by the IMU to provide a state estimate relative to the initial position. The resulting system is divided into several tasks that are scheduled using FreeRTOS.

The tests analysed the accuracy of the distance sensors and compared the methods used to determine the module orientation. The response of the system when mapping a simple structure was also analysed, first in a stationary environment where the position is given by the server, then based on the pose estimate of the Kalman filter. The tests show that a rough map of the environment can be successfully created and the correct communication between the tasks of the system and the server was confirmed.

Limitations of the system were bypassed during the tests. These include the slow processing speed of the camera, which limits the movement periods. The angle dependency of the ultrasonic sensors led to limitations regarding the surrounding spatial structure.

In the future, the module will be attached to a quadcopter that flies over a maze to map it while co-operating with other robots within the maze.

To summarise, it can be said that the planned objectives have been achieved. The ultrasonic sensors were combined in a single module with the camera, an IMU was added and communication with the server was established. The tests show that the connection between the different components of the system works and that a rough map of an indoor area can be created.

# Table of Contents

<b>Preface</b>	<b>i</b>
<b>Original problem description</b>	<b>ii</b>
<b>Summary and conclusion</b>	<b>iii</b>
<b>List of Tables</b>	<b>vi</b>
<b>List of Figures</b>	<b>viii</b>
<b>Notation</b>	<b>ix</b>
Basic Rules . . . . .	ix
Symbols . . . . .	ix
Abbreviations and Acronyms . . . . .	x
<b>1 Introduction</b>	<b>1</b>
<b>2 Theory</b>	<b>3</b>
2.1 Real time systems . . . . .	3
2.1.1 Scheduler . . . . .	4
2.1.2 Inter process communication . . . . .	5
2.1.3 FreeRTOS . . . . .	5
2.1.4 Embedded systems . . . . .	6
2.2 Serial Communication . . . . .	6
2.2.1 SPI . . . . .	6
2.2.2 I <sup>2</sup> C . . . . .	7
2.3 State estimation . . . . .	8
2.3.1 Probability distributions . . . . .	9
2.3.2 System Model . . . . .	10
2.3.3 Extended Kalman Filter . . . . .	10
2.3.4 Three-Dimensional Geometry . . . . .	13
2.4 Thread, MQTT and MQTT-SN . . . . .	14
2.4.1 Thread network . . . . .	14
2.4.2 MQTT . . . . .	15
2.4.3 MQTT-SN . . . . .	15
<b>3 System Description</b>	<b>17</b>
3.1 Hardware . . . . .	17
3.2 Software . . . . .	19
3.3 Code structure . . . . .	21
<b>4 Previous Work</b>	<b>23</b>
4.1 Ultrasonic sensors . . . . .	23
4.2 Camera . . . . .	26
4.2.1 OpenMV-code . . . . .	26
4.2.2 Interface with nRF52 . . . . .	27
4.2.3 Limitations . . . . .	28
4.3 Golang Server . . . . .	29



<b>5</b>	<b>Implementation</b>	<b>32</b>
5.1	Separate systems - Initial problems . . . . .	32
5.2	Hardware: Merging and resulting problems . . . . .	33
5.3	Modifications of code for ultrasound sensors . . . . .	35
5.4	Modifications of camera . . . . .	38
5.4.1	Driver setup in SES . . . . .	38
5.4.2	Integration into system . . . . .	39
5.4.3	Modification of communication structure . . . . .	40
5.5	Modifications of server connection . . . . .	42
5.6	Integration of IMU . . . . .	45
5.6.1	Reused functions . . . . .	45
5.6.2	Digital Motion Processor . . . . .	47
5.7	State Estimation . . . . .	48
5.7.1	Model . . . . .	49
5.7.2	Extended Kalman Filter . . . . .	50
5.7.3	Integration in the system . . . . .	51
<b>6</b>	<b>Tests and Results</b>	<b>53</b>
6.1	Ultrasound sensors . . . . .	53
6.2	IMU . . . . .	56
6.3	Camera and server . . . . .	58
6.4	Pose estimate . . . . .	60
6.5	Entire system . . . . .	63
<b>7</b>	<b>Discussion</b>	<b>67</b>
7.1	Comparison to results from previous work . . . . .	67
7.2	Choice of ultrasound sensors for distance measurements . . . . .	68
7.3	Evaluation of IMU . . . . .	69
7.4	Limitations . . . . .	69
<b>8</b>	<b>Further work and conclusion</b>	<b>71</b>
8.1	Further Work . . . . .	71
8.1.1	Hardware . . . . .	71
8.1.2	Software . . . . .	72
8.2	Conclusion . . . . .	72
	<b>Bibliography</b>	<b>I</b>

# List of Tables

2.1	The sampling position of the clock signal (in orange) is shown for each of the four SPI modes used. . . . .	7
4.1	WiFi access data of the Raspberry Pi border router. . . . .	29
5.1	Data structures for saving distance measurements. . . . .	36
5.2	Speed of sound and distance values for specified signal return times at temperatures between 16 and 22 °C. The calculations are based on equation 4.1 and 3.1 and a humidity of $H_T = 0 \text{ g/m}^3$ . . . . .	38
5.3	Format of the line message: The 1-byte message identifier (0x04) is followed by the pose data at which the image was captured $(x, y, \theta)$ . The start and end points of the line messages both contain the int16 $x$ and $y$ value of the associated point. The angle $\Phi$ of the line (thetaLine) is transmitted by the camera and is therefore also included in the line message. The angles are transmitted in degrees, the coordinates in centimetres. . . . .	44
6.1	Mean, variance and error of the distance measurements at different angles. The period of the task was 250 ms and the delay was 5 ms, with the <i>task_distance_meas</i> being the only running task. . . . .	54
6.2	Measured angles in degrees using the raw data and DMP mode for data acquisition at different positions. . . . .	58
6.3	Variances of the state and measurement variables as defined during the pose estimation tests. . . . .	62

# List of Figures

2.1	Value of the result as a function of the response time $t$ for types of real time systems.	3
2.2	Illustration of scheduling algorithms (blue: running, green: ready). The first third shows the arrival time and time needed to finish tasks A, B and C. An example of how the tasks are scheduled with the non-preemptive scheduling algorithm called Shortest Process Next and the preemptive scheduling algorithm Round Robin is shown underneath.	5
2.3	SPI operating principle: The master and slave shift registers are first filled with their respective data (Figure a). After 3 clock cycles, three of the master bits are shifted into the slave register and 3 slave bits are shifted into the master register (Figure b). After 8 clock cycles, the data transfer is complete and the registers have been exchanged (Figure c).	7
2.4	Data transfer using the I <sup>2</sup> C protocol. The levels of the SDA and SCL line during the start and stop condition, acknowledgement and data transfer are illustrated [12].	8
2.5	The normalized product of two Gaussian PDFs is also a Gaussian PDF where the resulting mean is between the original mean values [4].	10
2.6	Markov property: It is shown that the current state $\mathbf{x}_k$ only depends on $\mathbf{x}_{k-1}, \mathbf{v}_k$ and $\mathbf{w}_k$ not $\mathbf{x}_{k-2}$ or older states. The system model is used, which is described in section 2.3.2. [4]	11
2.7	Operating principle of the Kalman filter: If the previous state $\hat{\mathbf{x}}_{k-1}$ passes through the motion model, the predicted state $\check{\mathbf{x}}_k$ is obtained. If the measured state is multiplied by $\check{\mathbf{x}}_k$ , the posterior estimate $\hat{\mathbf{x}}_k$ is obtained. All PDFs are assumed to be Gaussian. [4]	12
2.8	Euler angle "1-2-3" sequence: The initial coordinate system is shown in blue. The first rotation results from the rotation around the z-axis by the value of $\Psi$ (in red). The second transformation is a rotation by $\Theta$ around the resulting y-axis (green). The last transformation in black results from the rotation around the x-axis by the factor $\Phi$ . [8]	13
2.9	Illustration of the MQTT-SN network connection with the MQTT network clients.	16
3.1	Overview of the hardware components and the software used to program them.	17
4.1	Case for the quadcopter module developed by Bjerke [6].	25
4.2	Wiring diagram of the connection of the OpenMV CAM M7 and nRF52840 DK used by Vormdal [23].	27
4.3	Structure of the communication procedure defined by Vormdal between the nRF52 and the OpenMV Cam. The yellow and grey arrows correspond to the capture message, the green arrows indicate the "Don't Care" message and the orange arrows illustrate the camera's response. The internal processes of the components are described next to the blue arrows, which represent the elapsing of time.	28
4.4	Network overview showing communication protocol and components, inspired by [1].	30
5.1	Wiring diagram after merging the camera and ultrasonic sensor systems and including the IMU.	33
5.2	New components and modified case for the quadcopter module.	34
5.3	Distances between sensors and camera origin.	37

5.4	New communication setup between server (left), nRF52 (centre) and OpenMV Cam (right) using either the estimated pose (5.4a) or the command message from the server (5.4b). The dark blue arrows indicate the progress of time. The operations performed on each system are described next to the blue arrows, and the messages sent between the components are shown in light blue (server messages), green (run message), yellow and grey (capture commands) and orange (line messages). The yellow capture command is highlighted in contrast to the grey ones, as it is the one to which the camera responds. . . . .	41
5.5	Polarity of axes for gyroscope and accelerometer of the ICM-20948 module [12]. .	45
5.6	Pose of the quadcopter. . . . .	49
6.1	Distance of the left ultrasonic sensor with a delay of 1 ms (blue), 2 ms (red) and 5 ms (yellow) to previous measurements. The frequency of the task is 250 ms, so that 240 measurements were recorded during the period of 1 minute. The task is not scheduled with other tasks. . . . .	54
6.2	Distances of the right (yellow), front (blue) and left (red) ultrasonic sensors in relation to the origin of the camera lens. The tests are carried out under stationary conditions and with different system settings. The purple line at 90 samples marks the end of the IMU calibration and the start of the position estimator. . . . .	55
6.3	Drift of the IMU under stationary conditions using the raw data (a) and the DMP (b). The results for a window threshold of 0 deg/s (all data are accepted) are shown in blue, for 0.01 deg/s in red and for 0.1 deg/s in yellow. . . . .	57
6.4	Test setup of the camera measurements and images captured by the camera, which were extracted from the OpenMV IDE during the tests. . . . .	59
6.5	Lengths of transmitted line segments at origin position (0,0). . . . .	59
6.6	Map shown on the server after each movement in positive $y$ direction. . . . .	61
6.7	Map shown on the server after each movement in positive $x$ direction (a)-(e) and when the segment is turned by $90^\circ$ (f). . . . .	61
6.8	Test of state estimation within the maze segment. Figure (a) shows the estimated position along a square while pointing straight ahead at the x-wall. Figure (b) shows two test trials of the estimation when changing the angle to $90^\circ$ and back when reaching the corners of the segment. The black square illustrates the actual position along which the module was moved. . . . .	62
6.9	Test setups where the position estimate is used as position reference when creating a map of the room. . . . .	63
6.10	Resulting map of first test. . . . .	64
6.11	Resulting map of second test. . . . .	65

# Notation

Throughout this thesis the notation will follow some basic rules. These rules, all used variables and parameters, as well as the most relevant abbreviation and acronyms will be described in this chapter.

## Basic Rules

In order to make the reading of this thesis as comfortable as possible for readers the code description and mathematical notation follow same basic rules that are listed here:

- Scalar functions are written as lowercase, non-bold letters. The discrete time index is  $k$  while the continuous time is indicated with the letter  $t$ .
- Vectors are displayed using bold, lowercase letters, such as the state of a system  $\mathbf{x}$ .
- Matrices are represented as uppercase, bold letters.
- Gaussian distributed variables follow the notation  $x \sim \mathcal{N}(\mu, \sigma^2)$  where  $\mu$  is the mean of  $x$  and  $\sigma^2$  the variance.
- File names and tasks are notated in slanted teletype font such as `task_pose_estimator.c` and specific functions are represented by the upright teletype font, e.g `getDistance()`. Newly defined datatypes are written in slanted font, like `ultrasonic_distance_t`.

## Symbols

Symbol	Meaning
$a$	Acceleration
$d$	Distance measurement
$e$	Error
$E[x]$	Expectation operator
$f$	Frequency
$f()$	Motion model function
$\mathbf{F}$	Jacobian of the motion model with respect to the system state
$g$	Gyroscope data
$g()$	Observation model function
$\mathbf{G}$	Jacobian of the observation model with respect to the system state
$H_r$	Relative humidity
$K$	Number of samples

Symbol	Meaning
$K$	Kalman gain
$N$	Dimension of a vector
$n$	Measurement noise
$P$	Covariance matrix of state estimate
$q$	Quaternion vector
$Q$	Covariance matrix of process noise
$R$	Covariance matrix of measurement noise
$T$	Temperature
$t_{\text{return}}$	Return time of a sound signal
$p(x)$	Probability density of variable $x$
$v$	velocity
$\mathbf{v}$	Inputs of a system
$v_{\text{sound}}$	Speed of sound
$w$	Process noise
$x$	x-coordinate
$\hat{\mathbf{x}}$	Posterior state estimate
$\tilde{\mathbf{x}}$	Prior state estimate
$y$	y-coordinate
$\mathbf{y}$	Measurements (for state estimation)
$z$	z-coordinate
$\mu$	Mean
$\Sigma$	Covariance matrix
$\theta$	Heading of the module
$\Theta$	Pitch rotation
$\Psi$	Yaw rotation
$\Phi$	Roll rotation

## Abbreviations and Acronyms

	Meaning
ACK	Acknowledgement (I <sup>2</sup> C protocol)
BLE	Bluetooth Low Energy

	Meaning
CPHA	Clock Phase (SPI mode configuration)
CPOL	Clock Polarity (SPI mode configuration)
CPU	Central Processing Unit
CS (or SS)	Chip Select (SPI line)
DK	Development Kit
DMP	Digital Motion Processor (of ICM-20948 IMU)
DoF	Degrees of Freedom
EKF	Extended Kalman Filter
FIFO	First In First Out
FPS	Frames Per Second
GPIO	General Purpose Input/Output
GSD	Ground Sample Distance
GUI	Graphical User Interface
I <sup>2</sup> C	Inter Integrated Circuit protocol
ID	Identity
IDE	Integrated Development Environment
IEKF	Iterated Extended Kalman Filter
IMU	Inertial Measurement Unit
IR	Infra-Red
LIDAR	Light Detection and Ranging sensor
LiPo	Lithium Polymer battery
MISO	Master In Slave Out (SPI line)
MOSI	Master Out Slave In (SPI line)
MQTT(-SN)	Message Queue Telemetry Transport (for Sensor Networks)
NTNU	Norwegian University of Science and Technology
PCB	Printed Circuit Board
PDF	Probability Density Function
PWM	Pulse Width Modulation signal
RAM	Random Access Memory
RTOS	Real Time Operating System
R/ $\bar{W}$	Read not Write bit (I <sup>2</sup> C protocol)
SCLK	Serial Clock (SPI line)
SCL	Serial Clock (I <sup>2</sup> C line)
SDA	Serial Data (I <sup>2</sup> C line)

	<b>Meaning</b>
SDK	Software Development Kit
SES	Segger Embedded Studio IDE
SLAM	Simultaneous Localisation and Mapping
SoC	System on Chip
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
USB	Universal Serial Bus
VS	Visual Studio
WiFi	Wireless Fidelity



# 1 Introduction

## Motivation

The mapping of an environment is an essential task to ensure safe access to deserted or unsafe areas. Often, rescuers are put in harm's way when attempting to rescue victims without knowledge of the surroundings and potential hazards. Autonomous robots can be used to map the area before entering it to gain information about possible obstacles and risks. Mapping robots can also be useful when exploring unknown areas that are inaccessible to humans (e.g. caves). A system for mapping such environments requires low energy consumption to be able to penetrate far into the field, wireless communication to provide real-time updates on the current status and preferably a low cost solution as loss of equipment is not unlikely. Circumstances such as poor visibility should also be taken into account.

As these situations are very complex, a more restricted use case is analysed in this thesis, which can, however, serve as a basis for the scenarios described. The focus of this work is on mapping an indoor maze. The map enables the user to find the optimal route to a destination or can serve as a basis for other autonomous tasks that use the map as a location reference. In this case, the low power requirements and low-cost hardware are less crucial, but still highly desirable for the system to be viable.

## Problem statement

This work is part of the "SLAM Robot Project" at the Department of Cybernetics at the Norwegian University of Science and Technology (NTNU). It was launched in 2004 and has since been regularly expanded by students from the department. Co-operating robots are used to autonomously map an indoor labyrinth. The low-end robots send their recorded position and measurement data wirelessly to a central server via MQTT-SN/Thread. The server creates the map and can be used to send command instructions to the robots. In addition to several two-wheeled ground robots that map the maze from the inside, a drone will be used to provide additional information about the entire maze structure from above. While this project is being carried out, there are four functioning ground robots that communicate via a Golang-server.

This work focusses solely on the development of a module for data acquisition, processing and distribution for a quadcopter. The module developed here will eventually be mounted on the underside of a quadcopter and will also be responsible for controlling the drone in future work. Previous work for the quadcopter includes a module with four ultrasonic sensors (type HC-SR04) to measure the position of the drone (June 2022, [6]) and a camera module, using an OpenMV Cam M7, that takes images and extracts line segments from them (December 2022, [23]). Both modules are based on the nRF52840 System On Chip (SoC) from Nordic Semiconductor.

The aim was to merge these modules into a single system, integrate an inertial measurement unit (IMU) (ICM-20984) into the system, improve the state estimate and establish a communication link with the Golang-server. The data from the ultrasonic sensors is processed to provide the coordinates of the module while the IMU measures its orientation, fully characterising the pose of the module. The pose is sent to the camera, which processes the image locally and calculates the global position of the image pixels. The lines from the image are extracted and the global coordinates are first sent to the nRF52 and then to the central server via MQTT-SN/thread. The server merges these lines with the data collected by the ground robots and creates a unified map of the maze.

## **Sustainability**

The United Nations' 17 Sustainable Development Goals adopted in 2015 are intended to ensure the sustainable development of our planet. It is the policy of NTNU that all research should contribute to these goals.

This work contributes to goal number 9, which describes the need to "build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation" [22]. The development of an energy-efficient embedded system for mapping indoor areas and the associated automation of otherwise tedious processes will promote both innovation and sustainable industrialisation.

## **Structure of the report**

This report begins with an overview of the theoretical background of real-time systems, serial communication, state estimation (focusing on the extended Kalman filter) and the MQTT-SN and Thread protocol. This is followed by a description of the hardware components and software environment used for this project. Their main features are listed and their role in the overall project is illustrated and an overview of the structure of the developed code is given. The previous work is summarised, including its unresolved problems. The implementation process and details are then presented. This includes the merging of the previously developed modules (both on the hardware and software side) and the integration of the IMU. The improvement of the state estimation through an extended Kalman filter, the modification of the communication with the OpenMV-Cam and the set-up of the communication with the central server are also presented. The test setup and their executions are described in chapter 6. The test results are discussed in chapter 7. Finally, ideas for further work are presented in chapter 8.

## 2 Theory

This chapter presents the theoretical background, specifying the principles and algorithms relevant to this thesis. The requirements and functionality of real-time systems are described and important features of FreeRTOS are mentioned. Since the design is subject to power and hardware constraints, the term embedded system is explained. In addition, the implemented serial communication protocols SPI and I<sup>2</sup>C are described. The theory of state estimation, including the extended Kalman filter algorithm, is presented. The Thread and MQTT-SN network architecture is also described.

### 2.1 Real time systems

"A real-time system is any information processing activity or system which has to respond to externally generated input stimuli within a finite and specifiable delay." This is Young's definition of a real-time system from 1982 [26]. As the definition states, the main goal of a real-time system is to behave in a predictable manner so that deadlines are not violated and calculated results are valid for defined periods of time. For example, when sensor data is collected, we need to ensure that the algorithms work with valid data so that the results are conclusive. There are different levels of time constraints for data and tasks [24]:

**Hard constraints** If a system cannot complete a task within a hard time frame, this leads to catastrophic errors such as environmental damage or, in the worst case, loss of life. These deadlines must therefore always be met. An example would be the control of a drone's rotors to ensure that the drone does not crash over a densely populated area. (See Figure 2.1a)

**Firm constraints** The result of these tasks no longer has any value once the deadline has passed. This is often the case with outdated sensor data. (See Figure 2.1b)

**Soft constraints** If the time limit has a soft constraint, it means that the data is still valid after the deadline has passed, but loses value. For example, when communicating with a server, the immediate transfer is often not important, but the data could be less relevant as time goes on. (See Figure 2.1c)

Depending on the time constraints of a task, it is assigned a criticality level. The levels can be categorised into safety-critical, mission-critical and non-critical tasks. Safety-critical tasks are

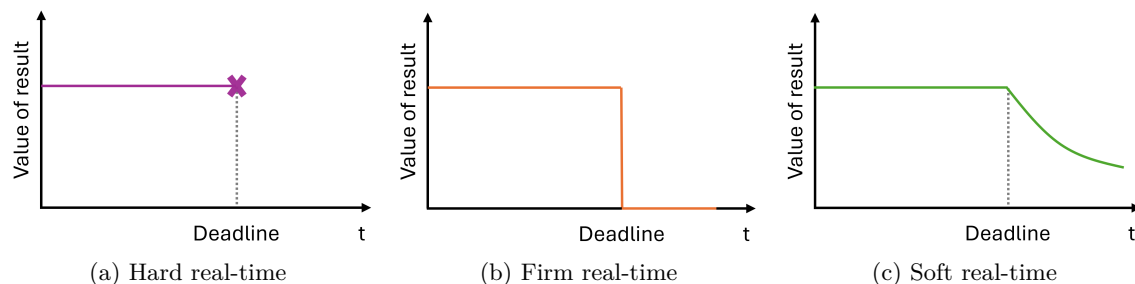


Figure 2.1: Value of the result as a function of the response time  $t$  for types of real time systems.

usually tasks that prevent the loss of human life or tasks that protect the system, e.g. if they are subject to hard constraints. If the purpose of the task contributes to the objective of the mission, e.g. to collect data, it would be categorised as mission critical. All other tasks fall into the category of non-critical tasks.

In order to structure the system code, it is divided into several tasks, each of which fulfils a specific function. This can be done by using several processes or threads. Each process has its own data set, including the program code and the stack. It also has a process control block, which contains an identifier, status, priority and program counter [25].

Threads are entities that exist within a process. They share the program code and data with other threads of a process, but have their own registers and their own stack. Switching between threads is faster than between processes, as not all data needs to be restored. It also follows that it is easier to communicate between threads and synchronise them as they share data. However, there is less protection between threads, which can lead to unintentional interference between them.

Only one process at a time can be executed by the processor, while the others have to wait. The status of a process describes which queue the process is in. In simple systems, there are three main system states: running, blocked (waiting for an event) or ready for execution. The running process is selected out of the ready queue by a scheduler. Semaphores, mutexes or queues can be used to forward messages between tasks and to protect memory areas during read and write operations. Real-time operating systems (RTOS) are implemented to integrate these operations into a system. FreeRTOS is one of these operating systems, which was developed for systems with limited resources. These terms are further explained in the following paragraphs.

### 2.1.1 Scheduler

The scheduler is the instance that decides which process or thread is executed by the processor over time. Scheduling can be either preemptive or non-preemptive. An example of the execution of methods representing each technique is shown in Figure 2.2. With a preemptive scheduler, a process is executed for a certain period of time and then interrupted by the scheduler. While a new process is selected, the state of the current process is saved and it is moved to the end of the ready queue. The interruption can be caused by the arrival of a new process, a periodic interruption by a timer or a system call. In a non-preemptive environment, a running task cannot be replaced by the scheduler unless it finishes, yields or gets blocked [25].

One of the most common preemptive scheduling algorithms is called Round Robin. Here, each process is assigned a time slice, after which it is replaced by the process that has been in the ready queue the for the longest time.

If tasks have different levels of criticality, each process can be assigned a priority. Each priority is then assigned its own queue in the ready queue of the system states. When deciding which process should be executed next, the scheduler first checks the queues with a higher priority before moving on to the lower priorities. If the higher priority tasks are scheduled at a too high rate, the system is not able to execute the lower priority tasks. This occasion is called 'starvation' and needs to be prevented by ensuring reasonable execution times and task frequencies.

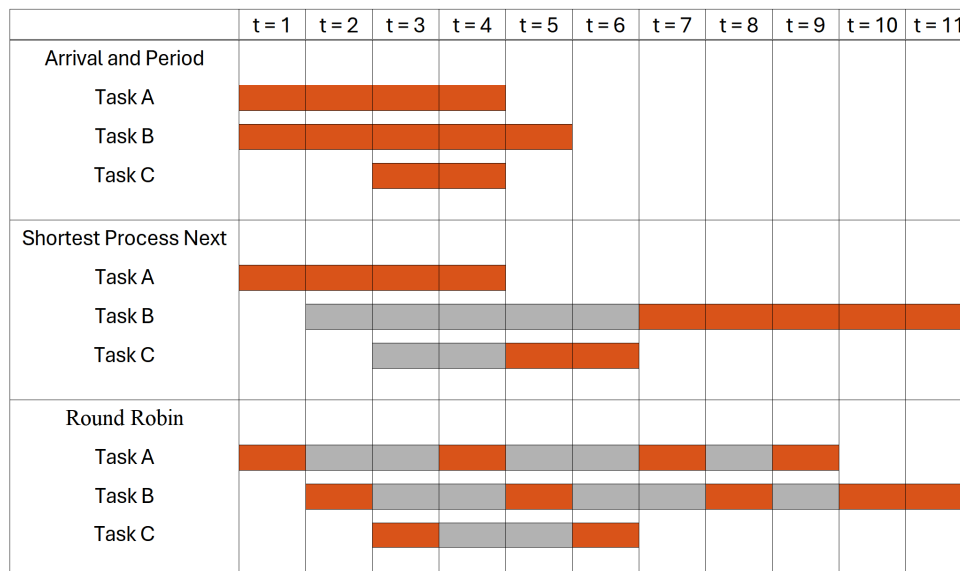


Figure 2.2: Illustration of scheduling algorithms (blue: running, green: ready). The first third shows the arrival time and time needed to finish tasks A, B and C. An example of how the tasks are scheduled with the non-preemptive scheduling algorithm called Shortest Process Next and the preemptive scheduling algorithm Round Robin is shown underneath.

### 2.1.2 Inter process communication

For synchronisation and information exchange between threads, methods, such as semaphores, mutexes or the message passing via queues, can be used [25].

If two threads try to access the same data at the same time, they can interfere with each other. When one thread reads while the other one writes to the same memory region, the result may depend on the order of thread execution. This is also known as a race condition. The memory area must therefore be protected while writing and reading. Semaphores are values that are linked to memory areas to ensure their protection. If a thread attempts to access the memory area, it can reduce the value of the semaphore and thus block the area (wait operation). Any other thread attempting to access this area changes to the "blocked" status during the wait operation. As soon as the thread has completed its operation, it increases the value of the semaphore (signal operation) to release the memory area and grant other threads access.

Mutexes are types of semaphores with the concept of ownership. They can therefore be used to bind a memory region to a specific thread, so that only the thread that locks the mutex can unlock it again. While semaphores can also be used for signaling between threads, mutexes are solely used to protect shared resources.

Queues are to pass messages between threads. A thread can attach messages to a queue that is read by another thread. Usually, a FIFO queue (First In First Out) is implemented here.

### 2.1.3 FreeRTOS

FreeRTOS is a small, open-source real-time operating system [10]. Due to its small size, it is often used for embedded systems running on microcontrollers. As in most simple operating systems, only one task (equivalent to one thread) is executed at a time. The tasks have

no knowledge of the scheduler's activity. Each task is assigned a priority between 0 and a user-defined maximum priority. The scheduler can work either preemptively (with round robin) or co-operatively (non-preemptive). When using the preemptive scheduler, a timer interrupt is required to re-evaluate which task is currently being executed with each timer tick. The task model comprises four states: ready, running, blocked and stopped. A task can be blocked if it is waiting for an external event or by calling `vTaskDelay()`.

FreeRTOS uses queues (commands: `xQueueSendToBack()` and `xQueueReceive()`), semaphores (commands: `xSemaphoreTake()` and `xSemaphoreGive()`) and mutexes (`xSemaphoreCreateMutex()`) for communication between processes. The capacity of the queue is defined when the queue is created (`xQueueCreate()`).

### 2.1.4 Embedded systems

An embedded system is a computer system that is defined and optimised for the execution of a specific task [25]. Embedded systems are usually part of a larger system consisting of multiple components, including sensors. They are often constrained by hard or soft real-time requirements, have limited memory and run on limited power sources such as batteries. In contrast, open systems for general computing, such as a typical desktop PC, are designed to be flexible, to fulfil high performance requirements and do not have strict power and size limitations.

## 2.2 Serial Communication

When designing an embedded system, all components, such as sensors and processors, must be wired to a common power supply. To enable data transfer between the components, they are connected to each other via communication buses. At board level, the most commonly used method is direct serial communication, where a master controls communication with one or more slave controllers. The Serial Peripheral Interface (SPI) and the Inter-Integrated Circuit (I<sup>2</sup>C) protocol are two of such interfaces for synchronous data transmission [7].

### 2.2.1 SPI

SPI is a simple, synchronous communication interface with four bus lines: MOSI (Master Out Slave In), MISO (Master In Slave Out), SCLK (Clock) and CS (Chip Select). It can be configured so that a master controller, which has control over the clock, can be connected to several slave devices. The slave selection is determined by the CS line, not by an address as is usually the case. It is a full duplex protocol, which means that the data flows simultaneously in both directions. The data rate of the interface depends on the capabilities of the master and slave controllers. Rates between 125 kbps and 16 Mbps are usually supported.

SPI communication functions like a cyclic shift register. A block of 8 bits is loaded into the register of the master. With each clock cycle, one bit of the master register is shifted out to the slave via the MOSI line and one bit of the slave register is transferred to the master via the MISO line. After eight clock cycles, the data is completely exchanged between the master and slave (see Figure 2.3).

During a write operation, the master can simply ignore the received byte. When reading from the slave, the master must send a dummy byte in order to receive the slave's data. SPI has

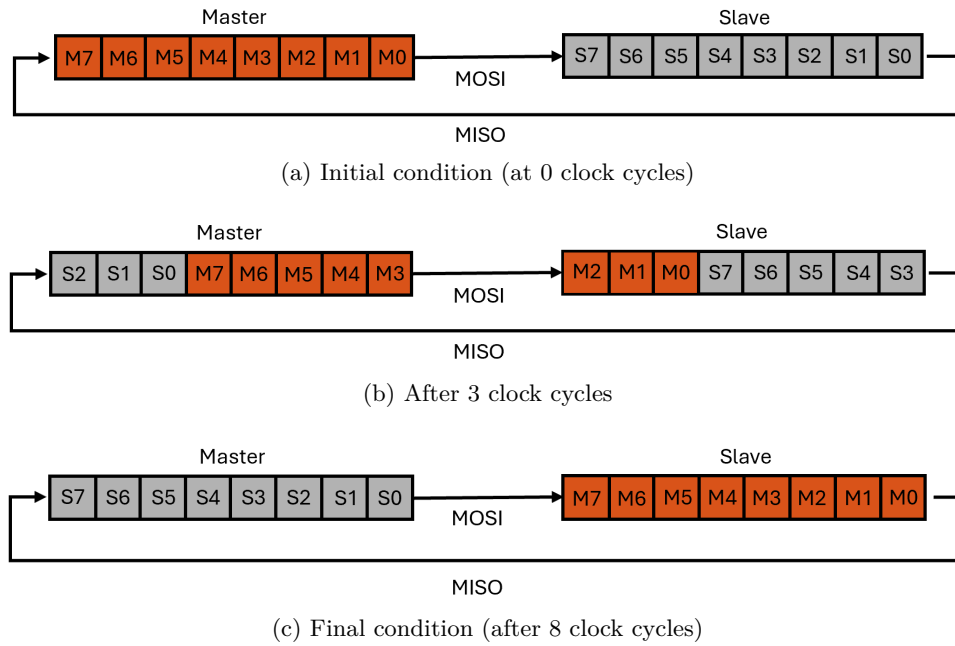


Figure 2.3: SPI operating principle: The master and slave shift registers are first filled with their respective data (Figure a). After 3 clock cycles, three of the master bits are shifted into the slave register and 3 slave bits are shifted into the master register (Figure b). After 8 clock cycles, the data transfer is complete and the registers have been exchanged (Figure c).

four operating modes, which are defined by two configuration bits: The clock polarity CPOL determines the idle state of the shift clock, the clock phase CPHA decides the clock edge at which data is to be sampled. The SPI modes are shown in the Table 2.1.

	CPHA = 0	CPHA = 1
CPOL = 0		
CPOL = 1		

Table 2.1: The sampling position of the clock signal (in orange) is shown for each of the four SPI modes used.

### 2.2.2 I<sup>2</sup>C

I<sup>2</sup>C is another serial protocol that uses only two wires, which are, however, bidirectional: SDA (Serial Data) and SCL (Serial Clock). It is a synchronous interface using a clock frequency of 100 kbits/s in standard mode and 400 kbits/s in fast mode, so it is generally slower than the SPI interface. Another difference to SPI is that I<sup>2</sup>C can have several master controllers. A master

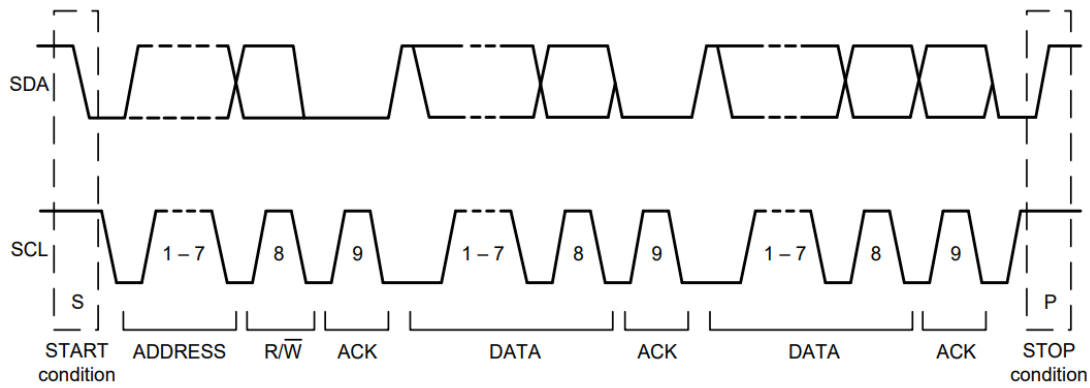


Figure 2.4: Data transfer using the I<sup>2</sup>C protocol. The levels of the SDA and SCL line during the start and stop condition, acknowledgement and data transfer are illustrated [12].

is defined by the device that controls the clock and initiates and terminates the communication, while the slave is the device that is addressed. To enable multi-master arbitration, the devices can only pull the lines down. Therefore, the state of the bus lines is defined by passive pull-up resistors that set them high when not in use.

Each I<sup>2</sup>C-compliant device has been given a 7-bit (or 10-bit) address for identification. In a network, all addresses must be unique to ensure error-free communication.

The operation is defined by the level of the lines (illustrated in Figure 2.4). When the lines are idle, both are in the high state. The master signals the start of a transmission on the line by pulling SDA to low while SCL remains high. Vice versa, the stop condition is signalled by releasing SDA while SCL is high. All other transmitted bits are sampled on the rising edge of SCL and must therefore be placed on SDA while SCL is low. After each 8-bit data transmission, the receiver confirms the information by sending an acknowledgement (ACK). This is indicated by releasing the SDA line while SCL is low.

When writing to a slave register, the master must first send the address of the slave, followed by a read or write bit (R/ $\bar{W}$ ), which is 0 for writing. After this has been acknowledged by the slave, the address of the slave register is sent, followed by the respective data. Most devices allow a burst transfer, so that several registers can be written during the same procedure. So, if more than one byte of data is sent, the slave automatically increases the register address so that the following registers can be modified too.

When reading from the slave, the initial procedure is the same. The bit R/ $\bar{W}$  is set to 1 to indicate the read operation. The master sends the address of the register and then releases the line. The slave responds by sending the corresponding data. If burst transmission is activated, it continues to send the data of the following registers until the master terminates the connection.

## 2.3 State estimation

The state of a robot defines the position and orientation of all its body parts. The state variables can be determined by analysing the structure and kinematics of the robot. When the robot moves, the state changes. Therefore, a motion model that models the movement of the robot is used to calculate the next state of the robot. Sensor data, such as distance measurements, are used to correct the estimated state (observation model).

The state estimation problem is defined as follows: "The problem of state estimation is to come up



with an estimate,  $\hat{\mathbf{x}}_k$ , of the true state of a system, at one or more timesteps,  $k$ , given knowledge of the initial state,  $\check{\mathbf{x}}_0$ , a sequence of measurements,  $\mathbf{y}_{0:K}$ , a sequence of inputs  $\mathbf{v}_{1:K}$ , as well as knowledge of the system's motion and observation models" [4]. Additionally, the uncertainty of the resulting estimate should be quantified.

### 2.3.1 Probability distributions

System models must be analysed from a probabilistic point of view. Every measurement in a system is influenced by noise, sensors have limitations and models are inaccurate. Therefore, any estimate that is calculated from these measurements is uncertain and must be expressed as a probability density function (PDF)  $p(\mathbf{x})$ , where  $\mathbf{x}$  is a vector of dimension  $N$ . As it is difficult to model PDFs directly with a computer, approximate values, such as their moments, are used instead. The more moments are calculated, the better the PDF is characterised. The first moment is the mean value,  $\boldsymbol{\mu}$ , and the second central moment is the covariance matrix,  $\boldsymbol{\Sigma}$ :

$$\boldsymbol{\mu} = E[\mathbf{x}] = \int_a^b \mathbf{x} p(\mathbf{x}) \, dx \quad (2.1)$$

$$\boldsymbol{\Sigma} = E[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T] = \int_a^b (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T p(\mathbf{x}) \, dx \quad (2.2)$$

If the dimension  $N = 1$ , the covariance matrix is called the variance  $\sigma^2$ , where  $\sigma$  is the standard deviation. Since computers cannot handle infinite amounts of data required to calculate actual integrals, they use the sample mean and the covariance. The integral is replaced by a sum over the number of samples  $K$  and the result is divided by  $K$  or  $K - 1$  for the covariance. The third and fourth moments are referred to as skewness and kurtosis. These are however rarely used.

To represent the noise, Gaussian probability functions are used. The multivariate Gaussian PDF, where  $\mathbf{x} \in \mathbb{R}^N$  is given by equation 2.3. As shown in the equation the distribution is fully characterised by its mean and covariance, therefore we write:  $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ .

$$p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^N \det \boldsymbol{\Sigma}}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right) \quad (2.3)$$

An important property of Gaussian PDFs is that when two of these PDFs are multiplied, the normalized result is also a Gaussian PDF (see Figure 2.5). This is utilised when fusing information in state estimation algorithms. Most methods for state estimation are based on Bayes' rule (see equation 2.4, 2.5). Looking at this from the perspective of the state estimation problem,  $p(x)$  can be expressed as the prior density without including data,  $p(x|y)$  as the posterior density (including data), while  $p(y|x)$  represents the sensor model.

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x) \quad (2.4)$$

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \quad (2.5)$$

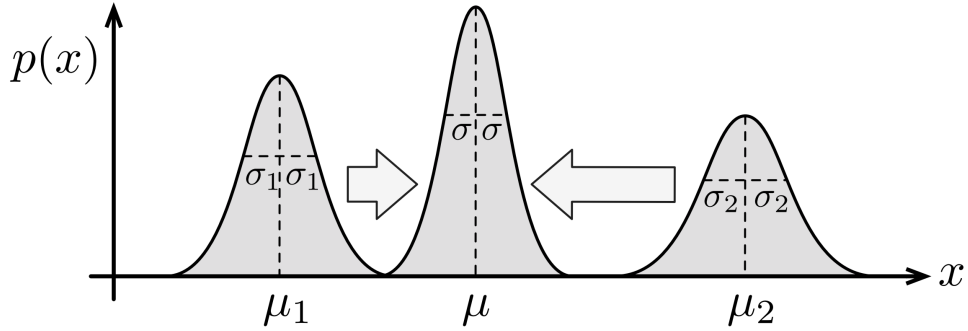


Figure 2.5: The normalized product of two Gaussian PDFs is also a Gaussian PDF where the resulting mean is between the original mean values [4].

### 2.3.2 System Model

A system is characterised by its motion and observation model. For non-linear systems, these are described by equation 2.6 and 2.7, respectively.

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{v}_k, \mathbf{w}_k), \quad k = 1 \dots K \quad (2.6)$$

$$\mathbf{y}_k = g(\mathbf{x}_k, \mathbf{n}_k), \quad k = 0 \dots K \quad (2.7)$$

$K$  is the maximum time and  $k$  is the discrete-time index. The system state is denoted by  $\mathbf{x}_k$ , the system input is  $\mathbf{v}_k$  and the process noise is  $\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k)$ . The observation model takes into account the measurement  $\mathbf{y}_k$  and the measurement noise  $\mathbf{n}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k)$ .

Both models can be simplified for linear systems.

### 2.3.3 Extended Kalman Filter

The extended Kalman filter (EKF) recursively determines the estimate of the state using a linearised version of the non-linear motion and observation models in an online procedure. The EKF works with the knowledge of the initial state  $\check{\mathbf{x}}_0$  and its uncertainty  $\check{\mathbf{P}}_0$ , the inputs  $\mathbf{v}_k$ , the covariance of the process noise  $\mathbf{Q}_k$  as well as the measurements  $\mathbf{y}_k$  and their associated covariance  $\mathbf{R}_k$  at time  $k$ . Previous and future inputs or measurements are unknown, only the previous state estimate  $\mathbf{x}_{k-1}$  is relevant. This works as long as the stochastic process fulfils the Markov property (illustrated in Figure 2.6). It states that for a current state, the PDF of future states can be characterised without knowledge of other past states and are therefore independent of the older states [4].

Since the system model is non-linear, some assumptions and approximations must be made in order to estimate the state of the system. First of all, both the belief function for  $\mathbf{x}_k$  and the noise variables are constrained to be Gaussian.

$$p(\mathbf{x}_k | \check{\mathbf{x}}_0, \mathbf{v}_{1:k}, \mathbf{y}_{0:k}) = \mathcal{N}(\hat{\mathbf{x}}_k, \hat{\mathbf{P}}_k) \quad (2.8)$$

This is necessary in order to obtain a quantifiable result when applying Bayes' equations. As described above, the product of two normally distributed variables is also Gaussian and can therefore be fully described by its mean and covariance. This is not true for most other PDFs, so this assumption is necessary to obtain a realisable estimator.

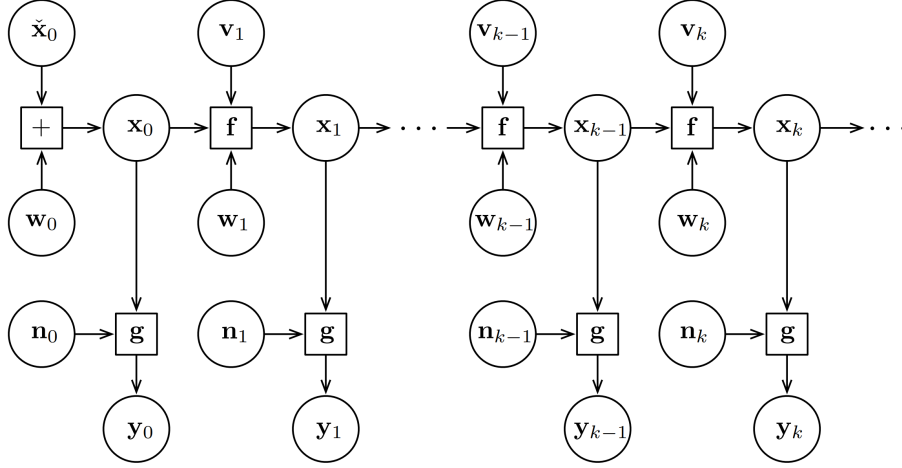


Figure 2.6: Markov property: It is shown that the current state  $\mathbf{x}_k$  only depends on  $\mathbf{x}_{k-1}$ ,  $\mathbf{v}_k$  and  $\mathbf{w}_k$  not  $\mathbf{x}_{k-2}$  or older states. The system model is used, which is described in section 2.3.2. [4]

To simplify the the motion and observation model, they must be linearised around the current state estimate  $\hat{\mathbf{x}}_{k-1}$ :

$$\mathbf{f}(\mathbf{x}_{k-1}, \mathbf{v}_k, \mathbf{w}_k) \approx \check{\mathbf{x}}_k + \mathbf{F}_{k-1}(\mathbf{x}_{k-1} - \hat{\mathbf{x}}_{k-1}) + \mathbf{w}'_k \quad (2.9)$$

$$\mathbf{g}(\mathbf{x}_k, \mathbf{n}_k) \approx \check{\mathbf{y}}_k + \mathbf{G}_k(\mathbf{x}_k - \check{\mathbf{x}}_k) + \mathbf{n}'_k \quad (2.10)$$

where

$$\check{\mathbf{x}}_k = \mathbf{f}(\hat{\mathbf{x}}_{k-1}, \mathbf{v}_k, \mathbf{0}) \quad \check{\mathbf{y}}_k = \mathbf{g}(\check{\mathbf{x}}_k, \mathbf{0}) \quad (2.11)$$

$$\mathbf{F}_{k-1} = \left. \frac{\partial \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{v}_k, \mathbf{w}_k)}{\partial \mathbf{x}_{k-1}} \right|_{\hat{\mathbf{x}}_{k-1}, \mathbf{v}_k, \mathbf{0}} \quad \mathbf{G}_k = \left. \frac{\partial \mathbf{g}(\mathbf{x}_k, \mathbf{n}_k)}{\partial \mathbf{x}_k} \right|_{\check{\mathbf{x}}_k, \mathbf{0}} \quad (2.12)$$

$$\mathbf{w}'_k = \left. \frac{\partial \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{v}_k, \mathbf{w}_k)}{\partial \mathbf{w}_k} \right|_{\hat{\mathbf{x}}_{k-1}, \mathbf{v}_k, \mathbf{0}} \mathbf{w}_k \quad \mathbf{n}'_k = \left. \frac{\partial \mathbf{g}(\mathbf{x}_k, \mathbf{n}_k)}{\partial \mathbf{n}_k} \right|_{\check{\mathbf{x}}_k, \mathbf{0}} \mathbf{n}_k \quad (2.13)$$

$$\mathbf{Q}'_k = \mathbb{E}[\mathbf{w}'_k \mathbf{w}'_k{}^T] \quad \mathbf{R}'_k = \mathbb{E}[\mathbf{n}'_k \mathbf{n}'_k{}^T]. \quad (2.14)$$

Here, the current state estimate is given by the posterior, which is denoted by a  $(\hat{\cdot})$ . By passing the posterior belief of the previous state through the linearised motion model, the prior belief is obtained, which is represented by a  $(\check{\cdot})$ . The estimate of the measurement is calculated by passing the prior state through the linearised observation model.

The Kalman filter works as follows: The current posterior state  $\hat{\mathbf{x}}_k$  is calculated based on the prior state estimate  $\hat{\mathbf{x}}_{k-1}$  and the associated covariance matrix  $\hat{\mathbf{P}}_{k-1}$ . First, the prior belief is obtained in the prediction step. This step is called prediction because we only use the current inputs  $\mathbf{v}_k$  into the system to estimate the next state of the system. The prior belief is then corrected by calculating the estimated measurement from this belief and comparing it to the actual measurement (merge). The result is the posterior estimate  $\hat{\mathbf{x}}_k$ . The uncertainty of the predicted and the corrected state are expressed by the corresponding covariance matrices  $\check{\mathbf{P}}_k$  and  $\hat{\mathbf{P}}_k$ . This procedure is illustrated in Figure 2.7.

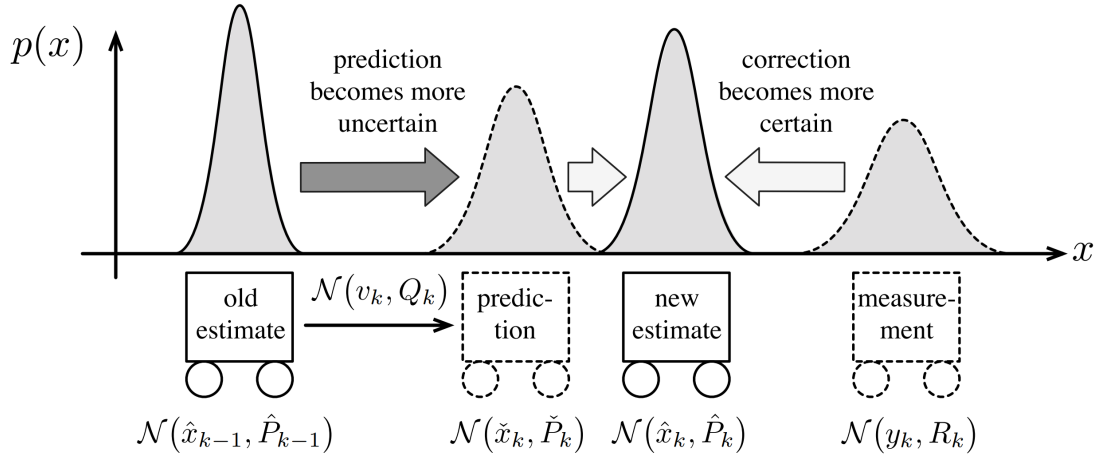


Figure 2.7: Operating principle of the Kalman filter: If the previous state  $\hat{\mathbf{x}}_{k-1}$  passes through the motion model, the predicted state  $\check{\mathbf{x}}_k$  is obtained. If the measured state is multiplied by  $\check{\mathbf{x}}_k$ , the posterior estimate  $\hat{\mathbf{x}}_k$  is obtained. All PDFs are assumed to be Gaussian. [4]

When this is expressed in equations, the process is also divided into prediction and correction. The prediction equations are shown in equation 2.15 and 2.16 and the corrector equations in 2.18 and 2.19. The Kalman gain weights the correction against the prediction using the uncertainty of both. It is given by equation 2.17.

$$\check{\mathbf{P}}_k = \mathbf{F}_{k-1} \hat{\mathbf{P}}_{k-1} \mathbf{F}_{k-1}^T + \mathbf{Q}'_k \quad (2.15)$$

$$\check{\mathbf{x}}_k = \mathbf{f}(\hat{\mathbf{x}}_{k-1}, \mathbf{v}_k, \mathbf{0}) \quad (2.16)$$

$$\mathbf{K}_k = \check{\mathbf{P}}_k \mathbf{G}_k^T (\mathbf{G}_k \check{\mathbf{P}}_k \mathbf{G}_k^T + \mathbf{R}'_k)^{-1} \quad (2.17)$$

$$\hat{\mathbf{P}}_k = (\mathbf{1} - \mathbf{K}_k \mathbf{G}_k) \check{\mathbf{P}}_k \quad (2.18)$$

$$\hat{\mathbf{x}}_k = \check{\mathbf{x}}_k + \mathbf{K}_k (\mathbf{y}_k - \mathbf{g}(\check{\mathbf{x}}_k, \mathbf{0})) \quad (2.19)$$

The EKF can diverge wildly, become biased or inconsistent as we linearise around the mean of the estimated state, not the true state. So if the estimated state deviates too far away from the true state, incorrect assumptions are made. The performance can deteriorate even further if the actual belief function is further away from the assumed normal distribution.

### Other nonlinear state estimators

To improve the estimate from the EKF, the iterated extended Kalman filter (IEKF) can be used. The correction step is performed several times using a linearised version of the observation model around a working point  $\mathbf{x}_{\text{op},k} \leftarrow \hat{\mathbf{x}}_k$ . While the EKF does not correspond to a defined value of the real posterior, the IEKF converges to a local maximum of the true posterior.

Other filters for non-linear recursive estimation, such as the Sigmoid-point-Kalman filter or the particle filter, are not considered in this thesis as they are difficult to implement in a low-level programming language such as C. Both filters are not based on linearisation, but on sampling, which can lead to more accurate results.

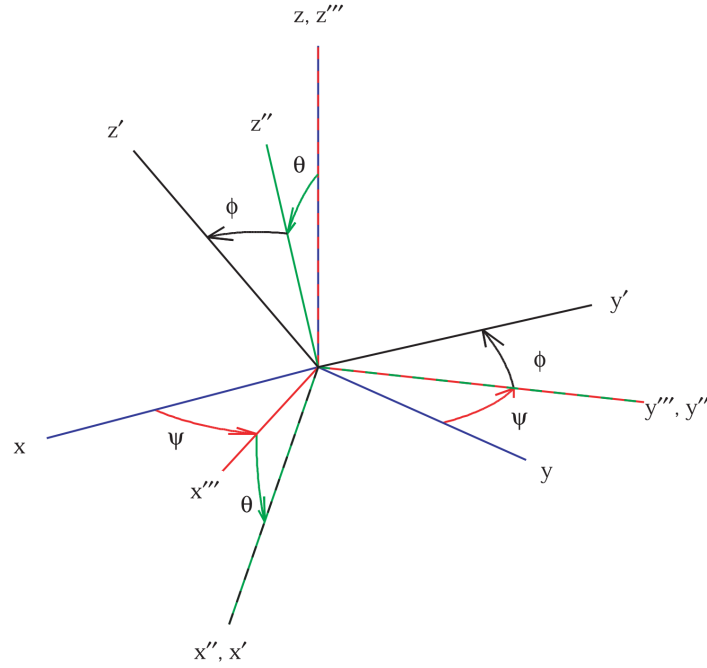


Figure 2.8: Euler angle "1-2-3" sequence: The initial coordinate system is shown in blue. The first rotation results from the rotation around the z-axis by the value of  $\Psi$  (in red). The second transformation is a rotation by  $\Theta$  around the resulting y-axis (green). The last transformation in black results from the rotation around the x-axis by the factor  $\Phi$ . [8]

### 2.3.4 Three-Dimensional Geometry

If the estimation takes place in a three-dimensional space, as is the case with a flying object, special aspects must be taken into account. The 3D position of an object is classified by six degrees of freedom (DoF): 3 in translation ( $x, y, z$ ) and 3 in rotation. Several different methods can be applied to represent the orientation.

Rotations are usually expressed using rotation matrices. A  $3 \times 3$  rotation matrix has 9 parameters, but only 3 DoF and 6 constraints, as the individual vectors are orthonormal to each other (the inverse of the matrix is equal to its transpose).

Alternatively, Euler angles can be used to describe the rotation. There are various sequences that specify the order of rotation. The 3-1-3 sequence is characterised by three angles  $\psi, \gamma, \theta$ . The rotation  $\psi$  takes place around the original 3rd axis, the rotation  $\gamma$  around the intermediate 1st axis and the rotation  $\theta$  around the transformed 3rd axis. Another sequence is given by the "roll-pitch-yaw" or "1-2-3" sequence. The "Yaw" rotation  $\Psi$  takes place around the original 3rd axis, the "Pitch" rotation  $\Theta$  around the 2nd axis in between and the "Roll" rotation  $\Phi$  around the transformed 1st axis. This is shown in the Figure 2.8. Although Euler angles have only three parameters and could therefore be assumed to be perfectly suited to express the rotation in the pose, they unfortunately suffer from singularities. For example, if  $\gamma = 0$  and the 3-1-3 sequence is used, the other two angles cannot be uniquely determined as they refer to the same axis.

Another representation for rotation are unit quaternions  $\mathbf{q} \in \mathbb{R}^4$ . In contrast to the Euler angles, they are independent of the order of the transformations and are therefore easier to handle.

$$\mathbf{q}^T \mathbf{q} = 1 \quad (2.20)$$

The four parameters of a quaternion can be calculated by specifying an axis of rotation with unit length specified by  $x, y, z$  and an angle  $\alpha$  that specifies the rotation around the axis. The result of the equation 2.21 then provides the quaternion value  $(q_0 + q_1i + q_2j + q_3k)$ .

$$\mathbf{q} = \cos(\alpha/2) + \sin(\alpha/2)(xi + yj + zk) \quad (2.21)$$

When rotating an arbitrary point  $\mathbf{p} = (x_1x_2x_3)$  by the specified angle around the specified axis, we must calculate

$$\mathbf{p} \rightarrow \mathbf{q} \cdot \mathbf{p} \cdot \mathbf{q}^{-1} \quad (2.22)$$

$$= (q_0 + q_1i + q_2j + q_3k)(x_1i + y_1j + z_1k)(q_0 - q_1i - q_1j - q_3k) \quad (2.23)$$

to obtain the coordinates of the rotated point. Calculations with quaternions are based on special mathematical principles, which are described, for example, in chapter 6 of the book [4]. Quaternions have 4 parameters for 3 DOF, so that this representation is also not optimal for the visualisation of rotation in 3D space. The Euler angles of the 1-2-3 sequence can be computed from the quaternions as follows [8]:

$$\begin{bmatrix} \Phi \\ \Theta \\ \Psi \end{bmatrix} = \begin{bmatrix} \arctan \frac{2q_0q_1 + 2q_2q_3}{q_0^2 - q_1^2 - q_2^2 + q_3^2} \\ \arcsin \frac{2q_0q_2 - 2q_1q_3}{q_0^2 + q_1^2 - q_2^2 - q_3^2} \\ \arctan \frac{2q_0q_3 + 2q_1q_2}{q_0^2 + q_1^2 - q_2^2 - q_3^2} \end{bmatrix} \quad (2.24)$$

## 2.4 Thread, MQTT and MQTT-SN

Wireless communication methods are used to transmit data over long distances and to exchange information with multiple devices. Cables restrict the freedom of movement of robots and are therefore unsuitable for most cases. This section introduces a low-power wireless network called Thread. It defines the data transfer, the network architecture and the roles of the network devices and is part of the network layer. Application layer protocols, such as MQTT, operate above the network layer. They define the communication flow between the connected devices. MQTT-SN is a special version of MQTT for sensor networks, both are presented here.

### 2.4.1 Thread network

Thread is a low-power, low-latency mesh networking protocol designed for wireless Internet-of-Things applications [18]. It runs on the 2.4 GHz frequency like other commonly used protocols, including WiFi and Bluetooth LE. Thread networks are scalable and do not depend on a central router, so there is no single point of failure. The network nodes themselves can act as routers depending on the active network structure. This and the most relevant other roles that network nodes can fulfil are shortly described:

**Router** The router nodes are responsible for forwarding messages between network devices. They are enabling other nodes to join the network and keep their transceiver enabled at all times.

**End devices** End devices only connect to a single router and do not forward messages. They can turn their transceiver off, to save energy. End devices that are router eligible can be promoted to a router when a new node close to them tries to join the network.

**Leader** The leader manages all routers in a network. It distributes configuration information between the network nodes.

**Border Router** The border router configures the network for external connectivity. It can forward messages between a Thread network and a non-Thread network, like WiFi or Ethernet.

Thread uses IPv6. The ID of an end device is a combination of the ID of the router and the child ID from the router's perspective.

## 2.4.2 MQTT

The Message Queuing Telemetry Transport (MQTT) protocol is based on a publish/subscribe procedure. The central server, also known as the broker, manages the connection to all devices in the network, processes subscription requests from clients and distributes the published messages to clients that have subscribed to the corresponding topic. Clients can publish and subscribe to topics in the network. The topic is defined by the topic name. The names can be structured in levels by inserting a slash "/" in the name to add subtopics and organise the communication. The server forwards all published messages of a topic to clients who have subscribed to the topic as well as the respective subtopics.

## 2.4.3 MQTT-SN

MQTT for sensor networks (MQTT-SN) is a low-power version of MQTT. MQTT-SN does not require TCP support like MQTT and can therefore be used with serial protocols such as Thread. An MQTT-SN gateway is created so that devices from underlying networks that use different protocols can communicate via MQTT. The gateway enables the nodes of the underlying network (like Thread) to connect to the same broker as the MQTT nodes and communicate with each other as if they were part of the same network. This is illustrated in Figure 2.9.

### Gateway discovery

When a new client attempts to establish a connection to an MQTT-SN network, it must send an advertise or search gateway message. When the message is received, the gateway responds with a gateway info message confirming the connection.

### Topic registration

The MQTT-SN protocol stipulates that topics must be registered before information is published on the respective topic. While only the topic name is published in MQTT, a 2-byte topic ID is used in MQTT-SN. When translating between the MQTT and MQTT-SN networks, the topic ID and name must be exchanged. To register a topic, the client sends a registration message to the gateway, specifying both the topic name and the topic ID. This is confirmed by the gateway's 'registration acknowledged' response.

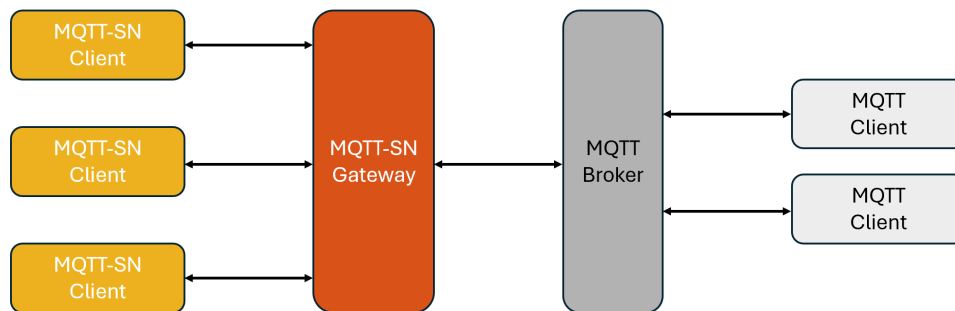


Figure 2.9: Illustration of the MQTT-SN network connection with the MQTT network clients.



### 3 System Description

This chapter presents the components used for the quadcopter module and the hardware required for communication with the server. Their main features and their purpose in the overall system are explained. The IDEs for programming the modules and other software tools used during the work are introduced. Finally, the structure of the quadcopter code developed as part of this work is presented. The implemented tasks and their interconnection are described. Since a large number of components and software applications are involved, Figure 3.1 schematically summarises which hardware is controlled by which software application.

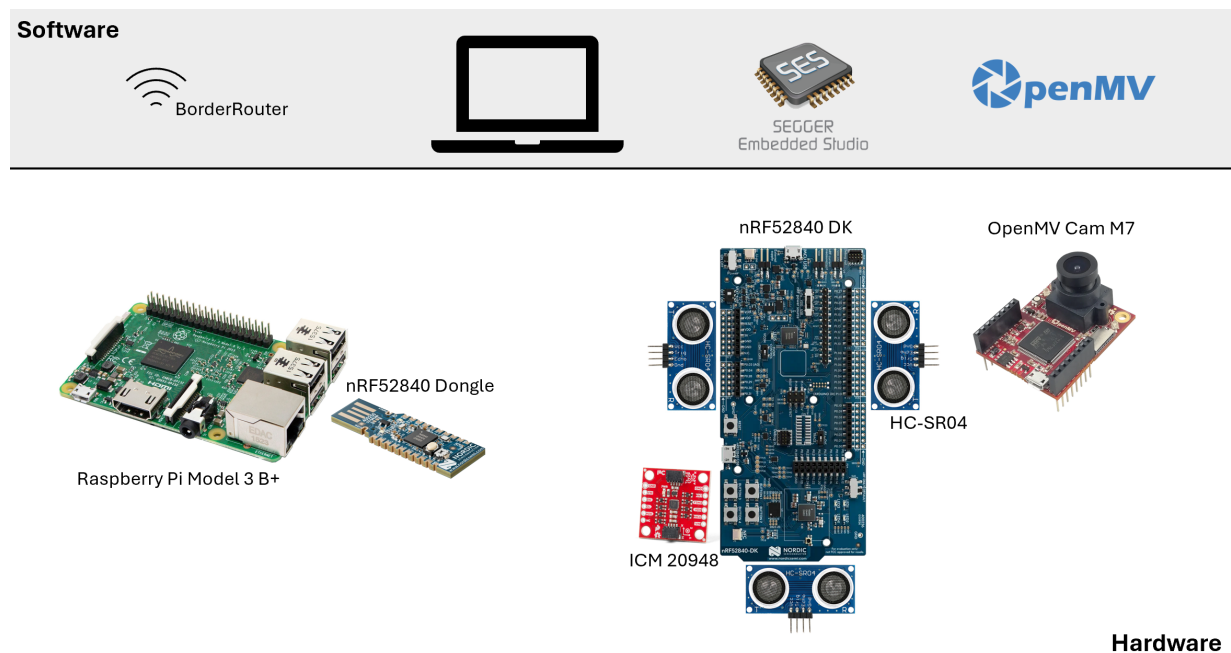


Figure 3.1: Overview of the hardware components and the software used to program them.

#### 3.1 Hardware

The microcontroller used for the ground robots and the quadcopter module is the nRF52840 development kit from Nordic Semiconductor. All sensors are connected to it via the pins provided by the module. The four ultrasonic sensors (HC-SR04) measure the distance to the left, right, front and bottom of the module. The DHT22 is a digital humidity and temperature sensor used to calculate the speed of sound. The images are recorded by the OpenMV Cam, which also processes them locally. The ICM-20948 Inertial Measurement Unit is used for calculating the orientation of the module. Detailed information on the individual sensors can be found below. A Raspberry Pi with the nRF52840 Dongle serves as a border router for forwarding messages between the nRF52 and the Golang-server. Their main purpose is briefly explained at the end of this section.

### **nRF52840 Development Kit (DK)**

The nRF52840 from Nordic Semiconductor is a system-on-chip (SoC) designed for ultra-low power applications. It contains an Arm Cortex M4F CPU (32-bit) running at 64 MHz. Wireless communication protocols with frequencies of 2.4 GHz such as Bluetooth Low Energy (BLE) and Thread are supported. The device has 1 MB flash memory and 256 kB RAM. The peripherals include serial communication interfaces such as I<sup>2</sup>C, SPI, UART, 5 separately configurable timers and analogue-to-digital converters (ADC) [16].

The nRF52840 Development Kit (DK) integrates the SoC and can be powered via a micro USB cable, a LiPo battery cell or by connecting an external power source that supplies between 1.8 and 3.6 volts. The DK has 48 configurable GPIO pins [3]. The main programme, called 'quadcopter code' runs on the nRF52, which controls all connected devices and establishes and maintains the connection to the server.

### **HC-SR04**

The HC-SR04 ultrasonic sensor measures the distance using acoustic waves. Distances from 2 cm to 400 cm can be detected with an accuracy of  $\pm 3$  mm for distances under 1 m [9]. The four pins include power supply (5 V), trigger, echo and ground. It can measure distances reliably as long as the measurement angle is less than 15°. If it is larger, the sound waves are reflected in a different direction, resulting in an invalid measurement. The distance is calculated by measuring the return time  $t_{\text{return}}$  of the triggered signal:

$$d = \frac{t_{\text{return}} \cdot v_{\text{sound}}}{2} \quad (3.1)$$

The time is multiplied by the speed of sound and the result is halved, as the signal travels twice the distance (forwards and backwards). As the sensor uses low-frequency waves, a measuring period of 60 ms is required to avoid interference. Four ultrasonic sensors are connected to the DK, which measure the distance to the left, right, front and bottom of the module. The measurements serve as a position reference.

### **DHT22**

The DHT22 (or AM2303), a digital sensor for relative humidity and temperature, is used for temperature measurement. The sampling time is 2 seconds with a temperature accuracy of  $\pm 0.5^\circ$  C. The device is supplied with an input voltage of 3.3 to 5.56 V and has one data port [2]. The sensor is connected to the DK and was previously used to measure the temperature and humidity in order to calculate the speed of sound required for the ultrasonic measurements.

### **OpenMV Cam M7**

The OpenMV Cam M7 is a low-energy camera module with an STM32F765VI ARM Cortex M7 processor that can execute basic computer vision algorithms. The processor runs at a frequency of 216 MHz and has 512 KB RAM and 2 MB flash memory. The maximum image resolution of the OV7725 image sensor is 640x480 for greyscale images at a rate between 75 FPS and 150 FPS depending on the resolution. The camera can be connected and programmed via USB and has a

serial interface for SPI (54 Mb/s), I<sup>2</sup>C and CAN (up to 1 Mb/s). The pins are 5 V-tolerant with 3.3 V output.

The camera module is connected to the DK. It captures images on command and processes them to extract the line segments. It then calculates the global coordinates of the lines based on the originally transmitted position and returns them to the DK.

### **ICM-20948**

The ICM-20948 Inertial Measurement Unit (IMU) is used to determine the orientation of the module. This low-power 9-axis motion measurement device developed by IvenSense comprises a 3-axis gyroscope, a 3-axis accelerometer, a magnetometer, a compass and a digital motion processor (DMP). The DMP offloads computation from the host processor and the provided data can be accessed via the FIFO buffer. The sensor can be connected via I<sup>2</sup>C (up to 400 kHz) or SPI (7 MHz). The operating voltage range is between 1.71 V and 3.6 V [12].

### **Raspberry Pi Model 3 B+**

The Raspberry Pi is used as a Thread Border Router. It is the communication hub between the robots (using Thread and MQTT-SN) and the central server (connected via WiFi using MQTT). It uses a 1.4 GHz, 64-bit quad-core processor and can communicate via WIFI, Bluetooth 4.3, BLE and Ethernet [19]. The operating system is loaded via a micro SD card.

### **nRF52840-Dongle**

The nRF52840-Dongle is a USB dongle to support wireless communication protocols such as BLE, Thread or Zigbee [17]. It is connected to the Raspberry Pi via USB and enables the a communication link between the robots (nRF52) and the Raspberry Pi.

## **3.2 Software**

This section contains a brief description of the software used in this work. They are used either to program the hardware, to create data logs, to develop algorithms or to track the changes made during development. Both the features and the intended use of the software are explained.

### **Segger Embedded Studio**

Segger Embedded Studio (SES) is an IDE for the development of embedded systems [20]. Version 5.66 of SES for ARM was used in this work for programming the nRF52. The IDE has integrated project management and options for executing and debugging the programme when the device is connected to the J-Link software via USB. The debugging options include the use of breakpoints or the ability to print values on the debugging terminal. A log file can be created in which the terminal data is saved in order to extract data from the system.

### **nRF Connect for Desktop**

nRF Connect is a tool framework developed by Nordic Semiconductor which contains specific software for their products. It includes a programmer with which .hex files of compiled programs can be flashed directly to the connected hardware. This is used to test supplied examples that implement specific functions or to flash the nRF52840-Dongle with the appropriate software. nRF Connect also provides a toolchain manager that is required to install the nRF Connect SDK.

### **nRF Connect SDK**

The nRF Connect SDK is a software development kit for Nordic Semiconductor products that can be used as an extension in Visual Studio Code (VS Code). It is based on the Zephyr RTOS and can be used as an alternative to Segger Embedded Studio [15]. It is easy to use and implements up-to-date software development tools. The memory layout of the connected hardware can be viewed and programmes can be flashed directly or uploaded in debugging mode.

### **OpenMV IDE**

The OpenMV IDE is used for programming the OpenMV Cam. The tool contains a serial terminal for debugging as well as an frame buffer viewer and a histogram display. The frame buffer displays the recorded images and can also contain processed image data. To run the code, the camera must be connected to the computer via USB. When the "Connect" and "Play" buttons are pressed, the currently open main file is uploaded to the camera. If the code for the camera is divided into several files, the non-"main" files must be loaded manually into the file folder of the connected openMV Cam.

### **Jlink RTT-Viewer**

The Jlink RTT viewer is used to create logs of the output data of the nRF52. These can also be created with SES, but debug mode must be activated for this. The debug mode leads to delays in data acquisition, which can be problematic when recording real-time data published at high frequencies.

### **MatLab**

MatLab was used to analyse and visualise data and to develop the extended Kalman filter algorithm. MatLab from MathWorks is a programming platform and language that contains numerous functions for data processing and has very good documentation. As vector data can be easily processed in Matlab, it is used to design algorithms before they are implemented in C. When comparing the output data of both versions, it is easier to recognise errors in the C version.

## GitHub

GitHub is mainly used for version control and code documentation. Being able to revert to previous versions of the code to find errors proved very useful. Furthermore, branches can be created when working on different aspects of the code at the same time or implementing new features. The entire SLAM robot project, including the robot code and the Golang-server, is published on GitHub, which has been helpful for comparing the developed code with the work of other students.

## 3.3 Code structure

The quadcopter code is based on the Golang-server version of the robot code, the software for controlling the ground robots. It is written in C and was developed with SES. The code is uploaded to the nRF52 DK, to which the ultrasonic sensors and the IMU are connected. The system operates on FreeRTOS and is divided into several tasks. Some of these tasks originate from the original robot code, others have been adapted or newly created. The tasks communicate via queues and use mutexes to protect shared data from interference by other tasks. The scheduler operates on a frequency of 1024Hz and the `vTaskDelay` function is used to define the frequency of the tasks based on the last tick value from the scheduler. The structure of the final version of the quadcopter code is presented in the following description.

***main.c*** The main function initialises the tasks of the system. It defines the stack sizes of the tasks, their priorities, initialises semaphores and starts the task execution and the FreeRTOS scheduler. The threads relevant for this work are: *task\_mqttsn*, *task\_imu\_data*, *task\_distance\_meas*, *camera\_mapping* and the *task\_pose\_estimator*. The previously used *task\_sensor\_tower*, which was mentioned during the description of the workflows, was replaced by the IMU and distance measurement tasks. The *task\_pose\_controller* and *task\_motor\_speed\_controller* used in the robot code are not relevant as the quadcopter module is not yet connected to a drone and therefore no control is established.

***task\_mqttsn.c*** This task initialises and maintains the MQTT-SN and Thread connection to the server and was only minimally adapted in the course of this work. The topic names are defined and the subscription and publishing procedure is implemented. During the initialisation of the connection, other tasks must wait if they require access to the server. As soon as a new message arrives from the server, it can be retrieved via queues with the corresponding topic handle. Other tasks can send messages by accessing the respective publishing functions.

***task\_sensor\_tower.c*** The sensor tower is a rotating tower with infrared sensors that the ground robots use to measure the distance to their surroundings. The task collects the IR measurements and controls the rotation of the tower. The measurements are sent to the server as an update message.

Since the quadcopter does not use the tower, the quadcopter version used this task at the beginning of development to perform the IMU and ultrasonic measurements. However, due to the incompatible measurement rates, the sensors were assigned separate tasks. Therefore, this task was removed from the system.

**task\_imu\_data.c** This task is used to retrieve the IMU data. In the course of this work, three different methods for obtaining the measurements were implemented, which can be selected at the top of the task. One of these methods is based on the previous robot code, which defined the measurement process in the position estimation task. After the IMU connection has been initialised and configured, the sensor is calibrated. During calibration, the steady-state offset of the sensor measurements is acquired by calculating the average value of 300-1000 samples. This bias is then removed from the gyroscope and acceleration readings and the readings are stored in a local variable. The sum of the gyroscope data, which corresponds to the angle of the module, is also recorded. The task uses the *ICM\_20948.c* driver, which defines all the functions required to access the IMU data as well as the *i2c.c* driver to connect to the IMU. It is scheduled every 40 ticks which corresponds to a frequency of approximately 25 Hz.

**task\_distance\_meas.c** The ultrasonic sensors are controlled in the distance measurement task. The task initialises the sensor pins, the timer, the data and accesses the speed of sound. It also defines the period of a measurement cycle (250 ticks) in which all four sensors are accessed. The *HC\_SR04.c* driver provides the measurement functions and the local variable for saving the obtained data.

**camera\_mapping.c** The camera mapping task is not based on the code from the ground robots but on other previous work. It uses the newly defined *spi.c* driver to access the OpenMV camera. The communication procedure with the camera is defined such that the position of the module is sent to the camera and the position of the line segments of the captured snapshots are obtained. The position of the module is specified either by the estimated position from the pose estimation task or by a message sent by the user of the server. When line segments arrive from the camera, they are transmitted directly to the server.

**task\_pose\_estimator.c** In this task the acquired sensor data from the ultrasound sensors and IMU is used to estimate the position of the module. After initializing the pose estimate and the static variables of the EKF, the estimator waits for the IMU to finish calibrating. It uses this time to calculate the initial distance to its surroundings, so that future estimates can be described relative to the origin. Once the calibration is finished, the newest sensor data is acquired and the extended Kalman filter defined in the *ekf.c* file is executed. The *ekf.c* reuses parts of the Kalman filter used by the ground-robots and accesses the *matrix\_operations.c* file. The resulting state is saved and an update message is sent to the server. The task runs at a frequency of 25 Hz.

**Other files** The files *defines.h* and parts of *functions.c* are used to convert angles between degrees and radians. The *globals.h* file defines the queues and mutexes. The pins of all connected sensors are specified in the *robot\_config.h* as well as global variables and modes that can be configured. The files are based on the predefined robot code and have only been slightly modified.

In addition to the quadcopter code, the Golang-server code and the OpenMV-code was also adapted. The original structure is explained in the next chapter and the changes are described in the chapter on implementation. However, the underlying structure of these programs remains identical.

## 4 Previous Work

This thesis is based on earlier work carried out by Master's students from the Department of Engineering Cybernetics at NTNU. The SLAM robot project, in that this work is embedded, has been carried out since 2004. The aim is to map a maze (in an indoor area) by using co-operating robots. For the majority of the project, ground robots were used to map the labyrinth from the inside. These are two-wheeled robots originally based on the programmable robots from LEGO®. Their components were replaced piece by piece and the software was continuously optimised until the current version of the robot code with the nRF52 DK as processor was reached. At the moment four of the six available ground robots are operational.

The server, which serves as the communication and control platform between the robots and the user of the system, was also adapted during the course of this project. The current server, which is implemented in the "go" programming language and is referred to as Golang-server, was developed by W. Kloze in 2023 [13]. Older versions of the server were implemented in Java and C++ and were communicating with the robots via BLE. A variety of different tasks were realised with these robots, including line tracking, obstacle avoidance, autonomous backtracking, etc.

In autumn 2021, the idea was formed to integrate a drone into the project, which would hover above the maze and process images of the maze. Jonas Øygaard Bjerke [6] was the first to work on the idea. He researched methods for estimating the position of a quadcopter. In particular, the distance to the ground is required so that the images can be linked to a position in space. For this, Bjerke used the ultrasonic sensors described in the previous chapter. His work was later improved by Kristian Gulaker [14], who also integrated a communication link to the C++ server used in 2022.

In a separate project carried out by Marcus Steffensen Vormdal [23], an image processing algorithm was implemented using the OpenMV Cam. The camera was connected to the nRF DK so that the processed data can be transmitted. The module containing the ultrasonic sensors, the development of the camera and the setup of the golang server are discussed in more detail in this chapter. Specific focus lies on the software and hardware implementation details, as they are further improved or adjusted in the course of this thesis.

### 4.1 Ultrasonic sensors

Jonas Øygaard Bjerke has developed a module based on ultrasonic sensors that enables it to measure the distance to its surroundings. The aim is to place this module under a quadcopter that takes images of the labyrinth with a camera and can estimate the placement of the images based on the distance measurements.

To achieve this, Bjerke first analysed different types of sensors in his specialisation project at NTNU [5]. The sensors were compared in terms of their sensing range, accuracy, measurement frequency, cost and availability. Since infrared (IR) sensors do not have a sufficient measuring range and LIDAR sensors, although very suitable for long range, precision and high frequency measurements, fail in the price category, Bjerke decided to use the available HC-SR04 ultrasonic sensors. The decisive factors for this decision were the low price, the efficiency and the measuring range. Other advantages of ultrasonic sensors are that they are not affected by smoke or dust like light sensors.

Problems can occur when measuring distances to angled or soft surfaces, as the sound is reflected or absorbed. This was confirmed in the tests carried out with OptiTrack, a motion capture system. Scenarios such as the rising up and down of the sensor, the reaction when aligned with

soft surfaces and the measurement of distance through a net hanging from the ceiling were tested. The sensor was first addressed with an Arduino board and later connected to the nRF52 DK. A driver for the ultrasonic sensors was implemented in Segger Embedded Studio by using a timer that measures the return time of the transmitted sound signal.

Bjerke continued work on the ultrasonic sensors in his master's thesis in spring 2022 [6]. It focussed on improving the previous implementation, measuring the accuracy of the ultrasonic sensors, designing and 3D printing a case for the ultrasonic sensors and the nRF52 DK, and integrating the system into the robot code so that a connection to the Java-server could be established. To improve distance measurements, the DHT22 temperature and humidity sensor was integrated into the system. With the measured temperature  $T$  and the extracted relative humidity  $H_r$ , the speed of sound required for the distance measurements can be calculated by an approximation 4.2.

$$v_{\text{sound}} = \sqrt{\frac{C_p}{C_v} \frac{R}{M} T} \quad (4.1)$$

$$\approx 331.4 + 0.606 \cdot T + 0.0124 \cdot H_r \quad (4.2)$$

$C_p$  is the specific heat at constant pressure,  $C_v$  is the specific heat at constant volume,  $R$  is the universal gas constant and  $M$  is the molecular weight of the gas. OptiTrack is also used here to determine the accuracy of the sensors. The tests showed that the measurement error  $\pm$  is 3 cm, where it is less than 1 cm 85 % of the time. If the module remains stationary, the error is less than 1 cm.

## Hardware

Bjerke developed a case to hold the sensors in place. The housing contains four ultrasonic sensors, the temperature and humidity sensor and the nRF52 microcontroller. A voltage divider on a breadboard is also connected to reduce the input voltage for the ultrasonic sensors from 5 V to 3.4 V. The case was printed from polylactic acid (PLA), has a length of 15.5 cm and weighs about 249 g, including all components. A picture of the case can be seen in Figure 4.1.

## Software

As part of Bjerke's implementation was the connection to the Java-server, the resulting code was based on the robot code for the ground robots. These functions integrated the ultrasonic and temperature sensors:

- The `HC_SR04.c` file is the driver for the HC-SR04 sensor. It contains functions for timer initialisation and a separate `getDistance` function for each of the four ultrasonic sensors. In addition, a function for reading the data from the DHT22 sensor is implemented and the function `find_speed_of_sound` for calculating the speed of sound is defined.
- The `main.c` file initialises the ultrasonic and temperature sensor pins, starts the timer for the ultrasonic measurements and calls the `find_speed_of_sound` function.



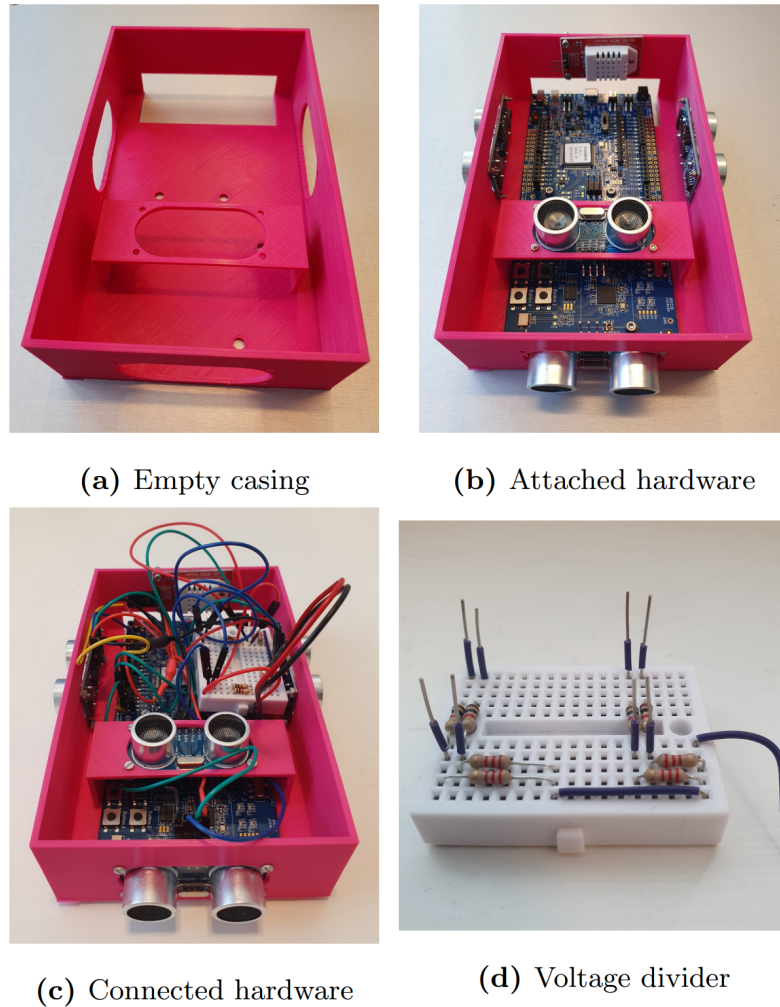


Figure 4.1: Case for the quadcopter module developed by Bjerke [6].

- The *SensorTowerTask.c* initialised in the main function has been adapted to include the ultrasonic measurements. The measurements of the IR sensors used by the ground robots have been replaced by the `getDistance` function. The task sends an update message to the Java-server containing the four distance measurements instead of the four IR values. The rest of the code remained the same as the original version of the robot code.

### Other work

Bjerke's design was improved by Kristian William Macdonald Gulaker in his specialisation project in autumn 2022 [14]. The system was adapted so that it can communicate with the C++-server via Thread and the integration of an IMU was considered theoretically. A description of the structure and the connection to the thread border router using the Raspberry Pi Model 3 B+ and the nRF52 Dongle is provided. The `MainComTask`, which handles communication with the servers, has been extended and is described in detail in Gulaker's report. Possible state variables and measurement models for IMU integration are specified and suggestions are made on how they can be integrated with the ultrasonic sensors:

- Roll and pitch are assumed to be zero, the pose only specifies the  $x, y$  and  $\theta$  (heading) position.
- If the distance of the quadcopter to its surroundings exceeds 4 m, the system must rely solely on the integrated IMU measurements, which are less accurate and drift.
- The ultrasonic measurements are invalid if they hit the wall at an angle, so in this case the IMU measurements must also be favoured.

Gulaker suggested that the next steps to improve the quadcopter system are to include an IMU, improve the cabling of the housing developed by Bjerke and integrate a camera into the system.

## 4.2 Camera

To create a map of the labyrinth from above, the quadcopter must be equipped with a camera. For this purpose, Marcus Vormdal designed a system with the nRF52 DK and the OpenMV Cam M7 in his specialisation project in autumn 2022 [23]. The aim was to extract the line segments from an image, which ideally represent the labyrinth walls, calculate the position of the lines based on the position of the module and return the segments to the nRF52 DK. The OpenMV Cam was chosen as the camera as it contains a processor. The nRF52's limited memory prevents it from processing high-resolution images, and it must adhere to certain time constraints when scheduling safety-critical tasks, which would be exceeded if computationally intensive image processing algorithms were applied at the same time. The nRF52 DK was chosen as the host controller as the module should later be integrated into Bjerke's design.

### 4.2.1 OpenMV-code

The code that is executed by the OpenMV Cam was developed in the OpenMV IDE and is referred to in this paper as OpenMV-Code. It runs locally on the camera and can be viewed and updated by connecting the camera to the PC via USB. After the camera has received a capture command and extracted the position from the command, it starts the line extraction with the help of a Line Segment Detector (LSD). The camera sensor is calibrated and a snapshot is taken. The image is divided into segments, which are processed individually to find line segments within them. The lines are appended to a buffer after the lines that are too short have been removed. The image has been analysed segment by segment, so the lines need to be merged and matched afterwards. As this procedure is beyond the scope of this thesis, it will not be explained further, but can be found in chapter 4 of Vormdal's report [23]. For each of the resulting lines, the global coordinates are calculated based on the position of the camera. They are displayed in the frame buffer of the IDE and returned to the host processor.

### Transformation into global coordinates

The global coordinates of the lines are calculated using the Ground Sample Distance (GSD), which describes the scale between the pixel and the real distance. The GSD is calculated from the pixel pitch  $p$ , the distance between the camera and the ground plane  $H$  and the focal width

$f$  of the camera:

$$\text{GSD} = \frac{p \cdot H}{f} \quad (4.3)$$

While  $p$  and  $f$  are constants,  $H$  is given by the received  $z$  position. The scaled coordinates are mapped to the global coordinate system using a homogeneous transformation matrix, which contains the translation and rotation of the image. When applying the algorithm, objects that protrude from the base surface result in an error that is proportional to the protrusion. Vormdal tested the accuracy of the transformation at different angles and heights and calculated the error between the real positions and the calculated distances. The results show that the calculated lengths of the segments exceed the real lengths, so the GSD was reduced by 8.7% at  $H = 150$  cm and by 5% at  $H = 120$  cm to compensate for the error. The final results showed an error of max. 3.5%.

#### 4.2.2 Interface with nRF52

The camera is connected to the nRF52 DK via SPI. This requires four communication lines as well as a ground connection. The OpenMV Cam is configured as a slave, while the nRF52 is the host that initiates the transmissions. Both the slave and the host are configured to use SPI mode 0 and expect the most significant bit (MSB) first. The clock frequency is set to 4 MHz. The cable connection is shown in Figure 4.2.

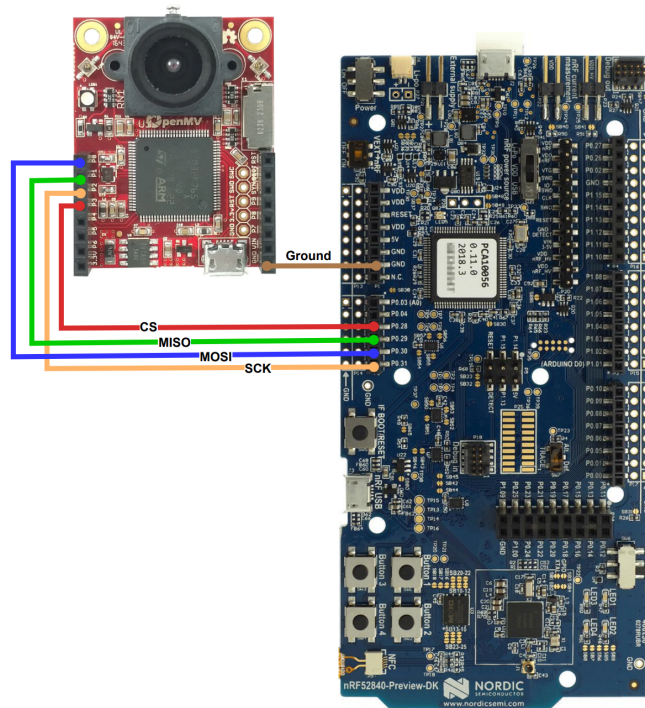


Figure 4.2: Wiring diagram of the connection of the OpenMV CAM M7 and nRF52840 DK used by Vormdal [23].

The camera driver on the nRF52 DK was programmed with the nRF Connect SDK in Visual Studio Code. It is not integrated into a larger system and is therefore the only task that runs on the microcontroller. First, the SPI connection is configured and initialised with the Zephyr SPI driver. The driver then enters a while loop in which, depending on the "waiting" variable, a

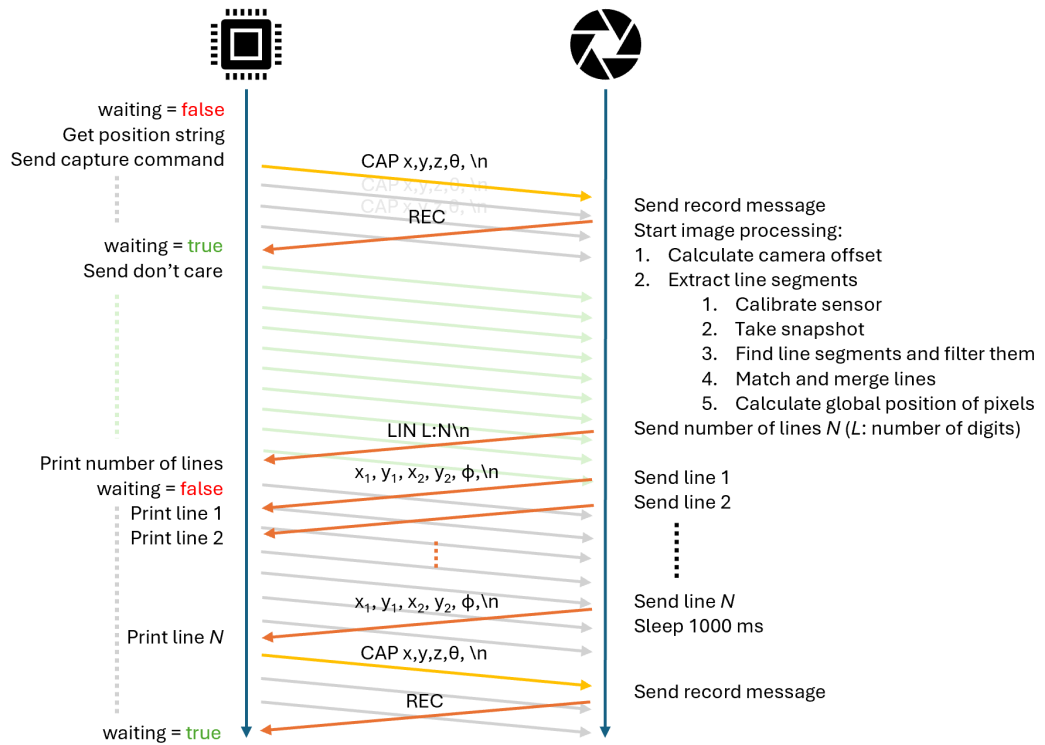


Figure 4.3: Structure of the communication procedure defined by Vormdal between the nRF52 and the OpenMV Cam. The yellow and grey arrows correspond to the capture message, the green arrows indicate the "Don't Care" message and the orange arrows illustrate the camera's response. The internal processes of the components are described next to the blue arrows, which represent the elapsing of time.

"don't care" message or a capture command with the position of the module is sent. The position is a fixed string defined in the code. After each transmission, the programme waits 100 ms before re-evaluating the "waiting" variable.

The new message is simply copied to a transmission buffer in order to send it to the camera. A received message appears in the receive buffer and is evaluated in the code. When the camera receives the capture command, it responds with a REC message (recording). If the REC message is recognised in the receive buffer, the driver sets the waiting variable to true so that the don't care message is sent. When the camera has finished processing, it first responds with a line message indicating the number of lines and then sends the lines one after the other. The driver on the nRF52 sets the wait variable to false as soon as the first line message is recognised and stores the expected number of lines in a count variable. When the line segments are received, the count is reduced and the coordinates are converted into integer variables and output to the console. The communication procedure is illustrated in Figure 4.3.

### 4.2.3 Limitations

Possible sources of error were highlighted in Vormdal's report:

- The labyrinth segments protrude out of the ground plane, therefore the GSD is offset, which leads to errors.

- The error of the GSD increases with the distance to the ground plane. This is due to the wide field of view of the curvature of the camera lens. To correct this error, the image would have to be cropped, which leads to a loss of information.
- The runtime of the algorithm for matching and merging is strongly influenced by the number of detected segments. This can lead to a long delay, which can violate the desired time specifications.
- The delay between messages during SPI communication is very high and can slow down the entire system.

## 4.3 Golang Server

The Golang-server is responsible for forwarding messages between the cooperating robots and the user and creates the map of the environment based on the incoming data messages. It replaces the previously used Java- and C++-servers and is integrated into the latest version of the robot code. The connection between the robots and the Golang-server is realised via MQTT/MQTT-SN/thread. The server must connect to the Border Router WiFi generated by the Raspberry Pi to access the Broker of the network. The WiFi access data is specified in the Table 4.1.

WiFi	Password
BorderRouter	12345678

Table 4.1: WiFi access data of the Raspberry Pi border router.

### Connection setup

The setup of the Raspberry Pi is described in Gulaker and Andersen's report [14],[1]. Gulaker and Andersen have set up the connection with the C++-server, but since both the C++-server and the Golang-server use the Thread/MQTT network, the setup can be adopted here. The Raspberry Pi with the nRF52840 Dongle is used as a thread border router. The robots send messages via the thread network layer and the MQTT-SN application layer to the Raspberry Pi, which forwards them via MQTT to the WiFi network to which the server is connected. The network connection between the robots, Raspberry Pi and server is illustrated in Figure 4.4.

The Raspberry Pi is configured with an image specially developed by Nordic Semiconductor for the border router and connected to a local router via Ethernet. The dongle is flashed by Nordic with a .hex file to receive the thread messages and is connected to one of the Raspberry Pi's USB slots. To configure the Raspberry Pi, one must connect to the Border-Router WIFI via ssh: `ssh pi@10.42.0.1`. The IP address of the broker is specified in the file `paho-mqtt-sn-gateway.conf`.

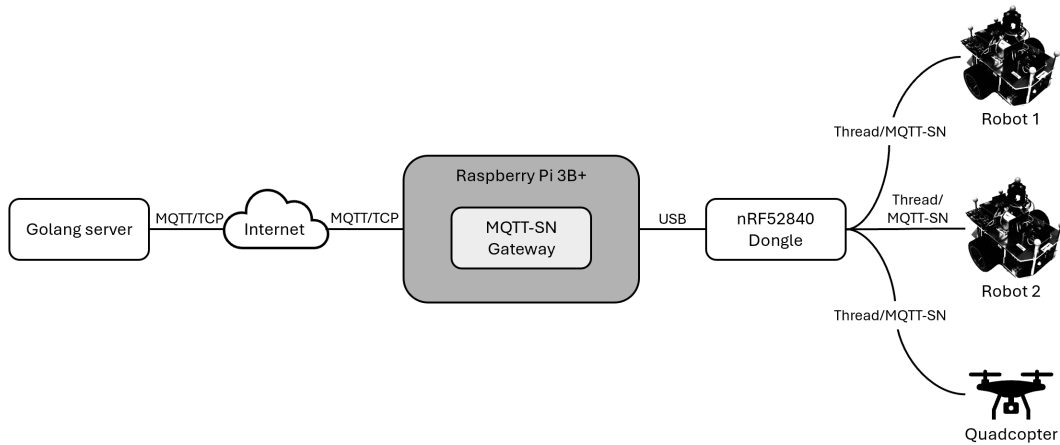


Figure 4.4: Network overview showing communication protocol and components, inspired by [1].

### Server-code structure

The golang-server was developed by Klose [13]. As the name of the server suggests, it is programmed in the programming language "go". To compile the server, fyne must be downloaded and installed. The server can be started by navigating to the src code folder and entering **go build .** and **go run .** in the command line. When the connection to the BorderRouter WiFi is established, the server's GUI opens in a separate window. The position of the robot is initialised in the GUI and the received data is visualised in the mapping window. In addition, the received messages are displayed in the command line. The GUI can also be used to send commands (positions) to the connected robots. As parts of the server were modified in the course of this work, the original structure of the server code is briefly outlined here.

- The *main.go* file defines the internal and external channels for forwarding messages between the server layers and the robots. The backend is initialised and mqtt communication is started. As soon as the connection is established, the graphical user interface is opened so that the map and command interface is displayed.
- The gui files specify the structure and appearance of the map and the server's user interface. The user interface includes a tab for initialising the robot, an automatic command tab and a manual command tab. In each of the tabs,  $x$ ,  $y$  and  $\theta$  can be specified in centimetres or degrees. Another row of tabs below shows the connected robots.
- The processing of the received data and the updating of the robot status and the map is carried out by the server backend. It accepts several channels as input and processes the data depending on the status of the channels. The initialisation and command messages specified in the user interface are processed here. If the automatic command is activated, the backend finds the robot that is closest to the specified position and sends this position it. If a manual command is entered by the user, the message is sent to the specified robot. When the update message is received from the robots, indicating the position and the measured distance from the IR sensors of the ground robots, the state of the robot is updated. The map is updated to show both the new position of the robot and the measured distance in the form of a line from the robot to the specified distance value. The line is generated using the Bresenham algorithm, which calculates the pixels on the line based on the start and end position of the line in pixels. The received message is also printed on the

terminal.

- The communication via mqtt is defined in the *mqtt.go* file. It contains the publish and subscribe functions. The `Subscribe` function is called by the main file at the start of communication in order to subscribe to the update message. The `ThreadMqttPublish` function transfers the messages set by the backend. The adv message handler for processing the update messages is also defined here. The message is unpacked, attached to the incoming message channel and printed on the console.
- The parameters broker name, port, mapsizes as well as gui parameters are defined in the *configuration.go* file. The incoming and outgoing message types and their structure as well as the robot status are specified in the *types.go* file.

## 5 Implementation

The aim of this thesis is to merge Vormdal's camera system and Bjerke's ultrasonic sensor system into a single module, to establish communication with the Golang-server and to integrate an IMU and develop a state estimation algorithm. The limitations and suggestions to improve the systems described in the last chapter were taken into account during the design phase. The resulting module is to be carried by a quadcopter in later projects. The problem was addressed by first separately testing the camera and ultrasonic module described in the last section. Initial problems encountered during this phase are described in the first section of this chapter. After the first tests of the system were successful, their hardware was merged and the IMU was connected to the module. The procedure and the resulting system are described in the second section.

To enable communication with the server, the current version of the robot code of the ground robots is used as the basis for the software integration. Both Vormdal's code and that of Bjerke's system were integrated into this code. The modifications to the ultrasonic sensor software, the server connection and the camera code are presented section by section. Finally, the integration of the IMU and the implementation of the EKF for state estimation are explained.

### 5.1 Separate systems - Initial problems

The first problem encountered when trying to access Bjerke's system was determining the appropriate version of Segger Embedded Studio to compile the code. The latest version of SES is version 8.10, but it turned out that version 5.66 was used in the project. If a newer version is used, the IDE cannot compile the code due to linking errors. This is mentioned here as a note for those who continue this work in the future so they do not waste time configuring the code setup.

After fixing this issue, the module was connected to the PC and the software was uploaded. The next problem was that the calculation of the speed of sound  $v_{\text{sound}}$  was not successful and the system got stuck in an endless loop. It turned out that the temperature sensor (DHT22) was not responding. This was bypassed temporarily by using a fixed value of  $v_{\text{sound}} = 343 \text{ m/s}$  and thus disregarding the sensor. As a result the code executed without errors and the distance measurements of the sensors could be accessed. At least all except for the upper sensor. It was assumed that this was due to a faulty wire, so it was decided to fix this during the hardware merge of the systems, as it was planned to replace all wires anyway.

Installing the software required for the camera and setting up Vormdal's system was straightforward. The nRF Connect SDK for VS Code was used instead of SES in this case. Both the camera and the nRF52 DK were connected to the PC via USB. After starting the camera and nRF software in the correct order (first nRF, then camera), messages were successfully transferred between the modules. Problems occurred when capturing images of complex structures, which caused the camera to run out of memory and crash while processing the images. There were also occasional errors in the transmission of the line segments, which caused the camera programme to freeze. However, as the system was working sufficiently, the merging procedure was initiated. How these problems were resolved later is described in section 5.4.



## 5.2 Hardware: Merging and resulting problems

The first step to merge the systems is to include the OpenMV Cam (and an IMU for later use) in the case designed by Bjerke, presented in section 4 and shown in Figure 4.1. Each of the four ultrasonic sensors has a power and ground connection to the DK, and is accessed by an input (trigger) pin and output (echo) pin. The DHT22 sensor is connected to power and ground and uses a bidirectional line for data exchange.

The connection of the OpenMV Cam to the nRF52 DK used by Vormdal has been shown in Figure 4.2. It includes a power and ground connection, as well as four wires for the SPI communication: MOSI, MISO, SCLK and SS. The wiring diagram showing the connection of the sensors to the DK after the merging is shown in Figure 5.1.

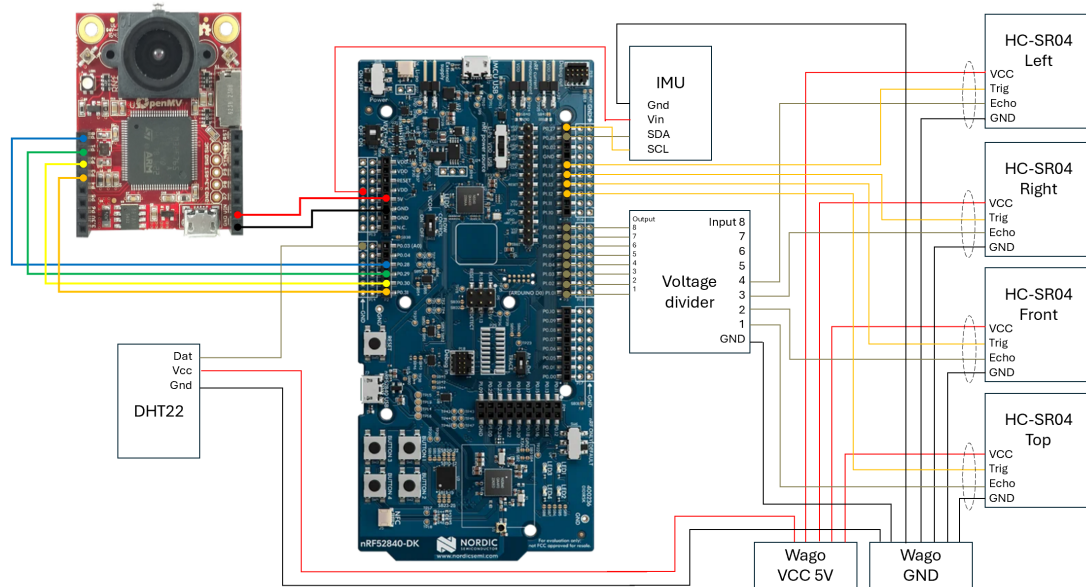
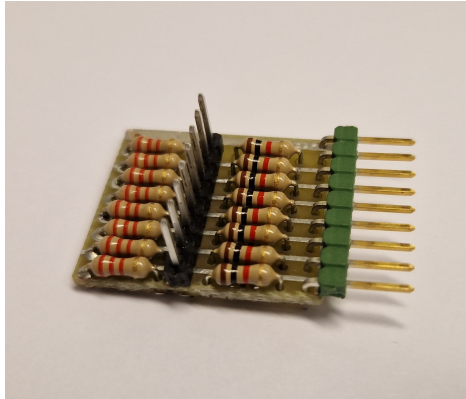
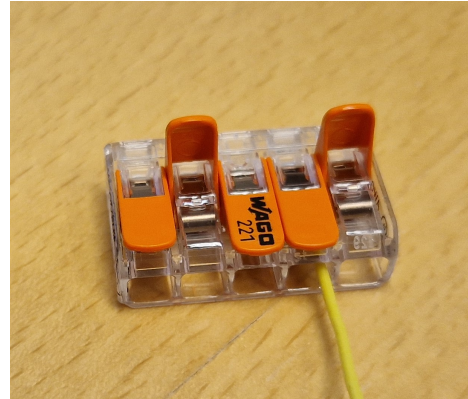


Figure 5.1: Wiring diagram after merging the camera and ultrasonic sensor systems and including the IMU.

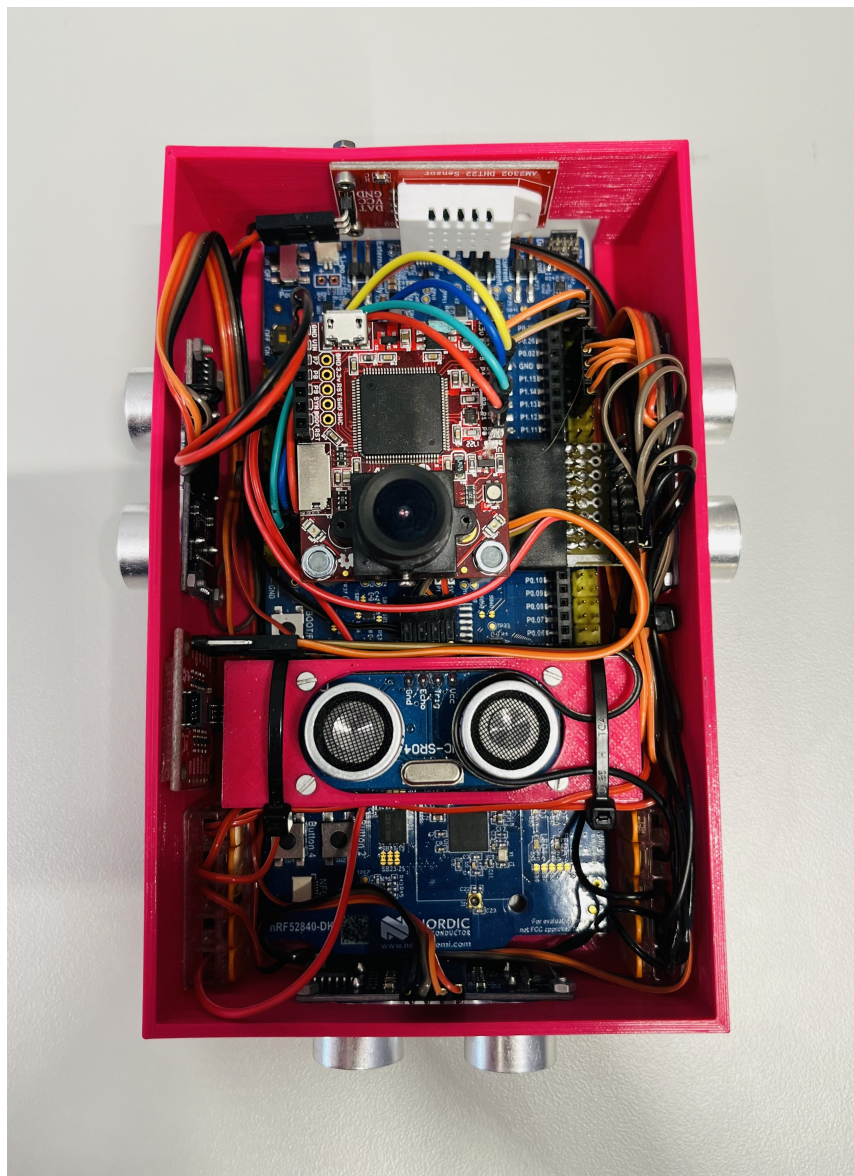
The camera was mounted in the centre of Bjerke's housing. Both the SPI connection to the camera and the trigger inputs for the ultrasonic sensors originally used PINs 28-31 of the nRF52 DK. The trigger pins of the ultrasonic sensors were therefore moved to pins 12-15 to free up the connection for the camera. Additionally, an IMU has been included (description in section 5.6). It was originally planned to connect it to pins 7-10 via SPI, but due to problems with the driver implementation, it was connected via I<sup>2</sup>C and pins 26 and 27 instead. The voltage divider on the breadboard was replaced by a card in which the resistors of the voltage divider are soldered to the inputs (see Figure 5.2c). WAGO clamps connect the sensors and the camera to the power supply in order to occupy fewer pins of the DK (see Figure 5.2b). The assembly of the hardware itself was carried out by Rune Mellingseter, a technician in the electronics laboratory of the Department of Engineering Cybernetics. An image of the modified module can be seen in Figure 5.2.



(a) Voltage divider card



(b) Wago Clamp



(c) Modified module

Figure 5.2: New components and modified case for the quadcopter module.

## Problems

When testing the modified module, a problem with the power supply was discovered. The ultrasonic sensors were not providing any measurement data as opposed to before the merge, so an oscilloscope was used to test the connection. As all four sensors were not responding at all, it was concluded that the power was the problem. Initially it was assumed that the USB connection was not providing enough power to supply all the sensors together. Therefore, an external power supply was connected to the Wago clamps as an interim solution. After using this for a while, it turned out that the Wago clamps were simply not connected to the 5 V and ground pin of the DK. Connecting them solved the problem so that an external power supply is no longer needed. The problem with the upper ultrasonic sensor has been resolved during the merge.

There were also some problems with the new camera connection. Transmitted messages were faulty and could not be decoded. As it turned out, the problem was due to both the ground connection and the wires used for the SPI connection. The ground wire had a pin which was too short on the side of the camera, so there was no contact. The other wires either had the same problem or were not suitable for a high frequency communication link.

The problems with the temperature sensor were not solved by using new cables. The inputs and outputs were analysed with an oscilloscope and no response signals could be detected. The sensor is most likely defective. Methods to work around this problem by using other available sensors are discussed at the end of chapter 5.3.

## 5.3 Modifications of code for ultrasound sensors

To combine the software of Bjerke's ultrasonic sensor module, the camera and the server communication, the latest version of the robot code used for the ground robots serves as basis. The code has been modified to accommodate the quadcopter application and all the features of Bjerke and Vormdal's work were integrated into this code. In this section, the integration of Bjerke's code (i.e. Gulaker's code) into the robot code as well as all modifications made to improve the measurements and code structure are presented.

### Integration of ultrasonic sensor code in overall system

Including the code for the ultrasonic sensors into the robot code was unproblematic as it was based on an earlier version of it. The driver for the HC-SR04 ultrasonic sensors *HC-SR04.c* and *HC-SR04.h* were copied to the application folder and the definition of the sensor pins was added to the *robot\_config.h* file. Since Bjerke uses one of the timers to measure the response of the ultrasonic sensors, timer 2 was reassigned for this purpose. It was originally used as a generator for the PWM signal for the ground robot's motor, which is not used by the quadcopter. The initialisation of the sensor pins and the start of the timer were called in the main file in Bjerke's code. To make the code more concise, an initialisation function *HC\_SR04\_Init* was added, which implements this functionality and is called before the task used for the ultrasonic measurements is started.

At the beginning of the development process, the measurements of the ultrasonic sensors were accessed in the *task\_sensor\_tower*, as Bjerke had implemented it. However, when implementing the algorithm for position estimation, the measurements were outsourced to a separate task dedicated to this purpose. This was done due to the long measurement cycle for each ultrasonic

sensor of up to 60 ms. Considering that four ultrasonic sensors need to be addressed, the entire cycle is 240 ms.

The algorithm for determining the state estimate and accessing the IMU data should run at a much higher frequency, so including the distance measurements in one of their tasks would slow down the system and starve other tasks in the system. The starvation is caused by the need to access the ultrasonic sensors with high priority so that the timer interrupt, which measures the return time, can be evaluated quickly. All lower priority tasks will starve if the frequency of the task is higher than the time required to execute the task once. By moving the ultrasonic measurements to a separate task with a slower frequency and a priority of 4 (highest priority, which both the IMU task and the position estimator task have), the starvation problem was solved.

However, when testing the system, the distance measurements were still very noisy and numerous invalid values were received. Initially, it was suspected that there was a problem with the measurement frequency, causing the signals to interfere with each other. After testing various settings and seeing no improvement in performance, an attempt was made to give the task a higher priority, in this case 5. This fixed the invalid measurements so that they could be accessed reliably. A description of the tests evaluating these conditions is given in the next chapter 6.

### Measurement functions

In Bjerke's version of the code, the ultrasonic measurements are accessed by calling a `getDistance` function, which is defined individually for each of the 4 ultrasonic sensors. The functions are called one after the other with a delay between the individual measurements. The delay is necessary to ensure that the sound waves decay before a new measurement is started. Since the four functions implement exactly the same thing for each sensor and only control different pins, they have been replaced by a single function that receives the trigger pin, the echo pin and the data pointer for the results as input.

While the original functions returned an 8-bit integer value, the new function `getDistance` returns the structure `ultrasonic_distance_t` that contains both the floating point distance value and a valid bit (see Table 5.1). The data is invalid if it takes too long to register the reflected signal. This is the case if the object in front of the ultrasound is more than 4 m away or if the sound hits the object at an angle so that the signal is reflected in a different direction. The result is also invalid if the resulting distance is less than 2 cm according to the specification of the sensor [9].

<code>ultrasonic_distance_t</code>	float distance bool valid
<code>ultrasonic_data_t</code>	<code>ultrasonic_distance_t</code> front <code>ultrasonic_distance_t</code> left <code>ultrasonic_distance_t</code> right <code>ultrasonic_distance_t</code> top

Table 5.1: Data structures for saving distance measurements.

Another function `getUltrasonicData` has been added that calls the `getDistance` function

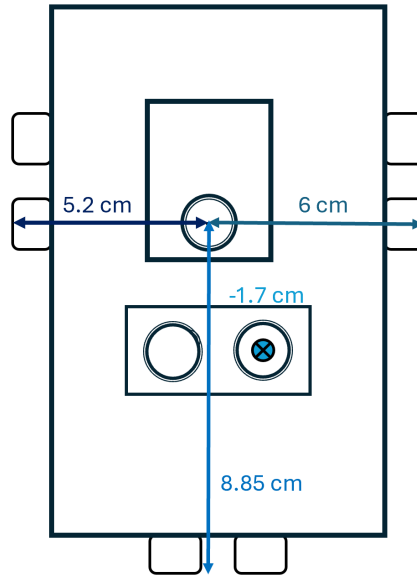


Figure 5.3: Distances between sensors and camera origin.

for all sensors in a row with a delay of 10 ms in between and saves the result in another data structure *ultrasonic\_data\_t*. In addition, the distance of each ultrasonic sensor to the centre of the camera lens is added to the measurements so that they refer to a common origin point (see Figure 5.3). The value of the distance measurements is also stored in a local static variable and protected with a mutex during the write process. Each task can access this variable by calling the `getDistanceMeasurement` function. This way, the value can be updated by a task that calls the `getUltrasonicData` function (in our case, the `task_distance_meas`) and by another task that accesses the measurement using the `getDistanceMeasurement` function (e.g. the `task_pose_estimator`).

### Integration of Timer

Before implementing a separate task for distance measurement, an attempt was made to address the ultrasonic sensor via a timer with a fixed frequency. At that time, it was assumed that the 20 Hz measurement frequency mentioned in Bjerke's report would apply to each sensor individually, not to all sensors together. As this did not work, the code was discarded, but as it took some time to implement, it is mentioned in this section.

Timer 3 was configured to run at a frequency of 500 kHz, which corresponds to one tick every 2  $\mu$ s. To do this, the prescaler register is set to 5 so that the timer base frequency of 16 MHz is divided by  $2^5$ . To obtain a measurement frequency of 20 Hz per distance sensor, the timer must be reset after 25000 counts. Since the sensors use the same timer to measure the return time of the sound wave, they are started at different intervals. Therefore, four capture compare registers are set to 6250, 12500, 18750 and 25000 so that the four distance measurements (front, left, right, top) start at equal intervals ( $25000/4 = 6250$ ). When the timer reaches the counter reading of a comparison register, a flag is set in the timer interrupt handler and the distance measurement is executed.

## Speed of sound

As mentioned in section 5.1 and 5.2, the temperature sensor for calculating the speed of sound did not work. As a temporary solution, the default value of  $v_{\text{sound}} = 343 \text{ m/s}$  at room temperature is used instead, but it is questionable whether this can be a permanent solution. Table 5.2 shows how strongly the distance measurements are influenced by a change in temperature. The distance difference itself is about 4 cm at a maximum distance of about 4 m when the temperature is changed from 16 to 22 °C. This is quite a long distance considering that these measurements are used as a positional reference for mapping a structure. However, it can be assumed that the temperature value in an indoor space is relatively constant, so the fixed value can be adjusted accordingly.

Temperature [°C]	Speed of sound [m/s]	Distance [m] $t_{\text{return}} = 23 \text{ ms}$	Distance [m] $t_{\text{return}} = 10 \text{ ms}$	Distance [m] $t_{\text{return}} = 4 \text{ ms}$
16	341.096	3.923	1.705	0.682
19	342.914	3.944	1.715	0.685
22	344.732	3.964	1.724	0.689

Table 5.2: Speed of sound and distance values for specified signal return times at temperatures between 16 and 22 °C. The calculations are based on equation 4.1 and 3.1 and a humidity of  $H_T = 0 \text{ g/m}^3$ .

The IMU and the nRF52 itself also contain a temperature sensor, but this is merely used to measure the die temperature of the chip. They are therefore only relative sensors that measure the temperature difference and not the absolute temperature. They have an accuracy of about  $\pm 5^\circ\text{C}$ , which makes them unsuitable for our purpose. No new sensor has been purchased yet, so the code is working with the fixed value for now. If another sensor of type DHT22 is used in the future, the variable `DHT_ACTIVE` can be set in the `robot_config.c` file to use the original temperature measurement procedure.

## 5.4 Modifications of camera

As described in the section 4.2, the camera code is divided into two systems. One part of the code (OpenMV code) executes image processing software on the camera itself, while the other is the driver that connects the camera module to the nRF52 and processes the data locally. The camera driver was developed with the nRF Connect SDK and not with SES. As the systems are based on different operating systems and libraries, the driver could not be integrated directly with the robot code. The setup of the new driver and the changes made to the communication and structure of Vormdal’s code to integrate the system with the server are presented in this section.

### 5.4.1 Driver setup in SES

The camera driver initialises the SPI connection with the camera and defines the communication procedure. When designing the new driver in SES, the original communication sequence which was presented in Figure 4.3 was first implemented. The spi connection with the camera is defined

by the driver `spi.c` and `spi.h`, which uses the library `nrfx_spim.h`. Here, an initialisation function specifies the parameters of the connection, e.g. the pins used, the spi mode and the transmission frequency. These parameters were adopted from the previous driver.

A transmission data buffer and a receive data buffer are required for sending messages with the NRF52 driver. Data transmission is implemented in the `spi_transmit` function, which copies the command to the transmit buffer and sends it to the slave. If the transmission is not successful, an error message is issued. Incoming messages are processed by the `spim_event_handler`. If the receive data buffer contains data, it is printed for debugging purposes and the function `camera_message_receive` for decoding the message is called. During the first task setup using the new spi driver, no changes had to be made to the OpenMV-code and the messages could be transmitted successfully. However, the errors in the original version have not been resolved.

### 5.4.2 Integration into system

Having added the camera driver to the quadcopter code in SES and achieved the same performance as Vormdal's version with nRF Connect, plans were made on how to integrate the camera with the server and the other tasks of the system. Since the camera is not similar to any of the existing tasks in the robot code, a new FreeRTOS task called `camera_mapping` was designed. The task has a priority of 2, as the mapping is not safety-critical.

#### Inputs

The camera expects a position command, which is needed to calculate the global coordinates of the pixels. Therefore, the input of the task must be the current position of the module. Vormdal used fixed position strings, as communication with a server or distance sensors were not integrated at that time. This is not the case in this project, so two ways of determining the position have been implemented.

The first version uses the current estimated pose from the `task_pose_estimator`, which is constantly updated. To avoid interference with the estimation task, a mutex is used when reading the data. The other option is to use a command sent by the server that returns the  $x, y$  values. The  $z$  coordinate is obtained from the position estimate, which corresponds directly to the measured distance of the upper ultrasonic sensor. The direction  $\theta$  is set to a fixed value in the code. Later, a third mode was implemented that uses the server command to trigger a record message but uses the state estimate from the estimator task.

The resulting position is then transformed to the camera's coordinate system by swapping the  $x$  and  $y$  coordinate. The capture command that is sent to the camera is generated by rounding the floating point values of the position, converting them into a character string and concatenating the character strings. The resulting message format is "CAP x, y, z, theta".

If the current estimate is used, the camera automatically captures new images once it has finished processing the previous one. When the server is used, the camera waits for the next message from the server in the meantime.

#### Output

After receiving the position, the camera takes a snapshot of the area under the module and processes it to extract the line segments of the image. The global coordinates of the start and

end pixels of each line are calculated based on the transferred position. If the recorded structure is a maze, the position of the labyrinth walls is determined in this way. The extracted lines are sent back to the *camera\_mapping* task.

In the previous version of the code, the driver only printed each line message on the console. Here we want to send the lines to the server so that it can add them to the map created by the ground robots. Therefore, the output of the camera task is an MQTT-SN message containing the coordinates of each received line.

Initially, this was implemented by counting the received line messages and storing them in a buffer. When the expected number of lines is captured, the bundle is sent together to the server. This has been changed as the buffer has a limited size and it is easier to send the messages directly after receiving each line individually. However, the buffer structure is retained in the code so that it can be used in future work if required. Before the line is transmitted, the coordinates are transformed back into the module's coordinate system.

## Task Setup

The *camera\_mapping* task first initialises the SPI connection. Depending on the mode used, either the current position is transmitted continuously or a run command is transferred until a new position arrives from the server. In continuous mode, the camera accesses a queue to wait until the pose estimation is ready at the start of the task. The task must transmit messages regularly, otherwise the slave cannot respond with data. The communication structure for both operating modes is shown in Figure 5.4 and explained in the next section.

### 5.4.3 Modification of communication structure

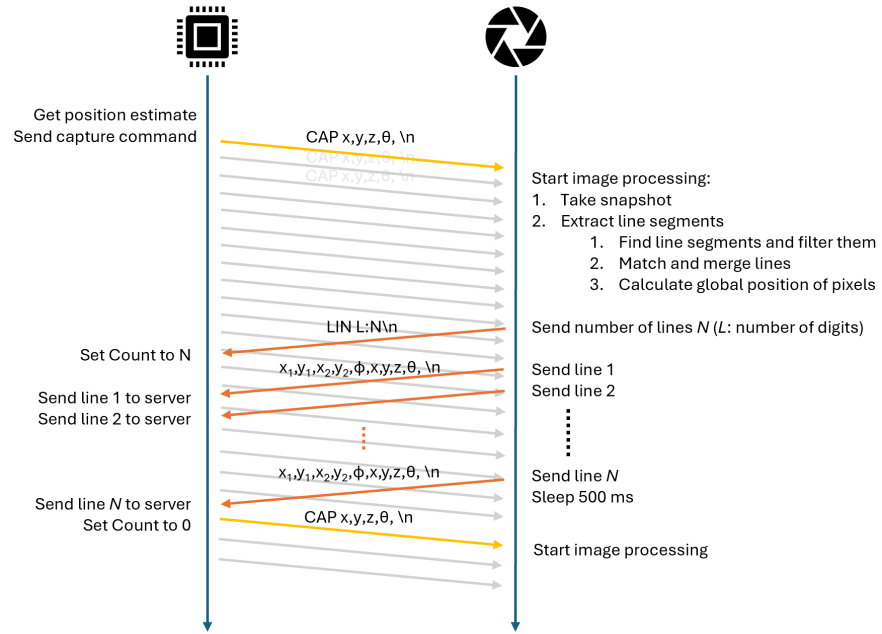
In order to enable error-free communication with the camera and improve the performance of the system, adjustments had to be made to the driver structure and the OpenMV code (as illustrated in Figure 5.4). The reasons for the changes are explained in this section.

#### Modified communication structure

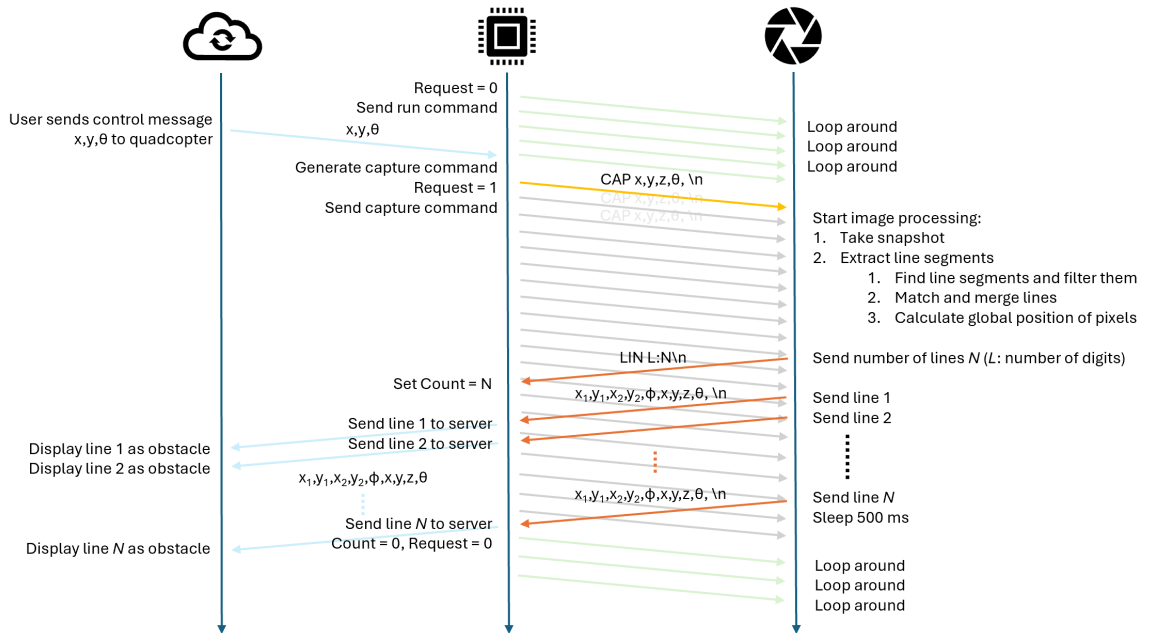
To resolve the errors that occur during communication with the camera, the communication procedure has been adapted. As shown in Figure 4.3, the camera responds with a REC message upon receiving the record command, and the driver goes into wait mode and sends idle messages (don't care). The problem that led to occasional system crashes was that the REC message interfered with the LIN messages, so that the first line message could not be processed by the driver and the next record command was not sent. The camera expects a new valid command shortly after sending the last line message, and if this does not arrive, the system crashes. The fault must have been a problem with remaining data in the transmission buffers. As this problem could not be resolved, it was decided to remove the REC response.

As a result, the waiting mode was removed and the capture messages are transmitted continuously. The link between the capture command and the received line message is established by a change in the line message format. The new format contains the position string from the capture command. The resulting message is " $x_1, y_1, x_2, y_2, p_{lin}, x_{pos}, y_{pos}, z_{pos}, \theta_{pos}$ ", where  $p$  is the line angle  $\Phi$ . As the SPI transmission is fast, the few additional bytes do not affect the time specifications and only the buffer size for incoming messages needs to be adjusted. Due to the





(a) Communication setup when position estimate from pose estimator task is used.



(b) Communication setup when position is send from the server.

Figure 5.4: New communication setup between server (left), nRF52 (centre) and OpenMV Cam (right) using either the estimated pose (5.4a) or the command message from the server (5.4b). The dark blue arrows indicate the progress of time. The operations performed on each system are described next to the blue arrows, and the messages sent between the components are shown in light blue (server messages), green (run message), yellow and grey (capture commands) and orange (line messages). The yellow capture command is highlighted in contrast to the grey ones, as it is the one to which the camera responds.

stable communication, the delay between the transmitted messages could also be reduced from 100 ms to 50 ms.

When using the position estimate as a position source, it makes sense to send the recording command continuously with an updated position. In this way, there are no delays in the system and the camera always works with the latest position. This structure is shown in Figure 5.4a.

When using the position sent by the server, the camera only needs to take snapshots when such a message arrives. In the meantime, it should remain in a standby state. As described above, errors occur if no valid message is sent to the camera. Therefore, the RUN message had to be added. It is similar to the "Don't Care" message, but the OpenMV code has been adapted to recognise it and loop while waiting for the next capture command. A capture message is sent until all processed line messages have been received. It is then replaced by the RUN message until a new server command arrives. The resulting new communication structure is shown in Figure 5.4b.

### **Modifications of OpenMV-code**

These changes also led to changes to the OpenMV code. However, these only affect the communication structure. The image processing methods such as line extraction, matching and merging have not been modified.

- As described above, the OpenMV code has been adapted so that it accepts the RUN message. A local run variable is set here to ensure that the main loop does not get stuck.
- The changed format of the line message leads to an increase of the transmission buffer size. The position from the capture message is appended to the line message by concatenating the character strings of the values.
- In the original version, the image sensor is calibrated before each image is captured. Parameters such as the pixel format, image size, contrast and brightness are set. To avoid delaying the operation, the function is called when waiting for a new command.
- As the system is likely to move during the process, the image must be taken as soon as possible after receiving the position report for the position to remain valid. The original code calculated offsets and parameters before a snapshot was taken, causing a significant delay of about 2s. This has been corrected so that the image is captured before any other processing begins, reducing the delay a little.
- When sending the line messages, the code inserted delays of 100 ms between each line and 1000 ms after sending all lines. These delays are halved to reduce the overhead.
- In the original version, an offset was added to the camera position, which was used to test the design. This was set to zero as the transferred position refers to the centre of the camera lens.

## **5.5 Modifications of server connection**

The connection to the Golang-server via MQTT-SN/thread has already been established in the robot code that serves as the basis for this project. The new goal is to display the line segments

extracted from the image captured by the camera at the correct position on the map. The other ground robots will publish their data on the same server and the map will take their data and the quadcopter's lines into account when creating the map. The `task_mqttsn` has barely been changed, only the topics and message formats have been adapted. The server's backend and mqtt message processing were modified to handle both the ground robot messages and the new line messages. To enable communication with the server, the hardware had to be set up in the right way.

### Hardware setup

The hardware for the connection to the server consists of the Raspberry Pi Model 3B+ and the nRF52840 USB dongle. When in range of the WiFi generated by the Raspberry, it is possible to connect to it using the credentials given in the Table 4.1. If the setup used by the ground robots (Pi1) is available, the server will find the broker and can run without any problems. As this was not always the case, the Raspberry Pi (Pi2) configured by Gulaker served as a substitute. While Pi1 configured the broker address as 127.0.0.1, Pi2 connects to the IP 137.135.83.127. By changing the file `/etc/paho-mqtt-sn-gateway.conf` on the Raspberry Pi, the IP can be adjusted, but the password used to connect to Pi2 has been lost. To avoid resetting the Pi2, the host file `C:\Windows\System32\drivers\etc\hosts` can be edited under Windows. If the line "137.135.83.127 slam" is inserted, the server connects to the correct broker. Alternatively, the server configuration file can also be changed. The variable Broker must then be set to this IP.

### MQTT-SN task and communication

The mqtt-sn task manages the connection to the server. Essentially, the task searches for an open gateway, topics are registered and subscribed to. The topic names, the associated message format and the message ID are defined in the task header file. The function `mqttsn_client_is_connected` can be called by other tasks to check whether the gateway has been found and communication is ready. The `publish_update` function can be used to send a position update message to the server and the `publish_line` function to send a line message to the server. The line message was implemented by Andersen, a former student working on the ground robots [1]. Although other topics are also registered by the `task_mqttsn`, they are not yet recognised by the Golang-server. The only outgoing messages from the server are the controller and initialisation messages. They can be accessed by accessing the corresponding queue, which is found by calling the function `get_queue_handle`.

As described in the previous section, the controller message is used by the server to send a position string to the `camera_mapping` task. This is done in the server's graphical user interface by opening the Manual tab and specifying the desired position. For this purpose, a new command topic `COMMAND_TOPIC_CAM` is defined in the `robot_config` file using the line topic identifier and added to the initialisation procedure of the `task_mqttsn`. The camera driver accesses the command topic queue to see if there are any new messages. The `publish_update` function is used by the ground robots to send their measurements to the server. This topic was initially reused by the quadcopter code to send the ultrasonic sensor data to the server and debug the sensors instead of printing the values on the debugging console in SES. In the final version of the code, the estimated position is transmitted via this message. To send the line messages to the server, the topic and message format defined by Andersen has been changed (see new format in Table 5.3). The `publish_line` function is called when each line segment is received from the

camera.

Datatype	Value	Number of bytes
uint8_t	identifier	1
int16_t	xdelta	2
int16_t	ydelta	2
int16_t	thetadelta	2
coordinate_t	startPoint	4
coordinate_t	endPoint	4
int16_t	thetaLine	2

Table 5.3: Format of the line message: The 1-byte message identifier (0x04) is followed by the pose data at which the image was captured ( $x, y, \theta$ ). The start and end points of the line messages both contain the int16  $x$  and  $y$  value of the associated point. The angle  $\Phi$  of the line (thetaLine) is transmitted by the camera and is therefore also included in the line message. The angles are transmitted in degrees, the coordinates in centimetres.

## Server code

The Golang-server code described in section 4.3 had to be adapted to process the line message. The processing of the update message was used as a reference for the implementation, as the procedure is comparable. The changes to the original code are listed here:

- A data structure has been added to the *types.go* file that specifies the format of the line message.
- A *lineMsgUnpacking* structure has been added to the *mqtt.go* file, which specifies the number of bits for each element of the line message. The *lineMessageHandler* unpacks the message, saves it in the *lineMsg* structure and prints the received line on the server's terminal. The subscribe function has also been modified. In addition to the update message, it takes a *lineMsg* channel as input, subscribes to the line message topic and specifies its handler. The successful subscription is displayed on the terminal.
- The line message is processed in the *backend.go* file. It is added as input for the *ThreadBackend*. The first line message registered by the server enables the initialisation of the position offset. After initialisation, it is possible to send update messages to this topic. The position of all lines that are received afterwards are displayed relative to the initialised position. The display on the map is done using a modified version of the existing *addLineToMap* function. The map indices (pixels) of the start and end points are calculated and the already implemented Bresenham algorithm is used to calculate the point positions that characterise the entire line. The resulting points are printed on the map as obstacles in black colour.
- The *main.go* file initiates the *lineMsg* channel. It is included when the *ThreadBackend* and *Subscribe* functions are called.

The procedure for accessing the server is not altered: When the server code is executed, the graphical user interface and the terminal in which the received messages are displayed will open. When the quadcopter publishes messages on the update or line topic, the initialisation of the robot position for these topics is activated. The robot is added to the map and updated when new messages are received. A control message can be sent to the robot by selecting the "Manual" tab and specifying the topic. The  $x, y$  position may be set in the corresponding fields and will be transmitted to the robot.

## 5.6 Integration of IMU

By establishing the connection with the server, the process of hardware and software merging between of ultrasonic sensor module, the camera and the Golang-server has been completed. In order to obtain the correct position of the module and the image lines, the ultrasonic sensor data needs to be processed and a measure of orientation must be added to the system. The orientation is obtained by integrating the ICM-20948 Inertial Measurement Unit into the quadcopter module. This IMU was already used by the ground robots, so the connection was partly provided by the files *ICM\_20948.c*, *ICM\_20948.h*, *ICM\_20948\_ENUMERATIONS.h* and *ICM\_20948\_REGISTERS.h*. As part of the expansion with additional functions, these files were modified and the file *ICM\_20948\_DMP.h* was added. The connection to the IMU and the implemented options for retrieving and processing the data are described in this section.

### 5.6.1 Reused functions

The original driver for the IMU provided the I<sup>2</sup>C connection (*i2c.c* and *i2c.h*) as well as read and write functions for initialising the IMU registers and accessing the gyroscope and accelerometer data registers. The gyroscope provides the angular velocity around the  $x, y$  and  $z$  axes of the module and the accelerometer provides the acceleration along the three axes (see Figure 5.5). The connection of the IMU and the processing of the raw measurement data was adapted for the quadcopter.

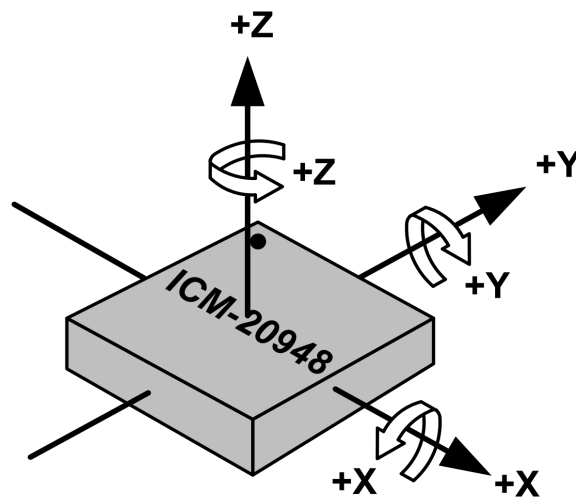


Figure 5.5: Polarity of axes for gyroscope and accelerometer of the ICM-20948 module [12].

## Connection

The original plan was to connect the IMU to the nRF52 DK via SPI. Due to problems with the integration of the SPI driver for the IMU and the realisation that an I<sup>2</sup>C driver for this IMU is already included in the robot code, the connection interface was changed to I<sup>2</sup>C. Pin 27 of the DK is connected to the SCL line and pin 26 to SDA (see circuit diagram 5.1). As these are the same pins that are used to connect the IMU in the ground robots, the addressing in the code did not have to be changed. Only structural changes were made to the I<sup>2</sup>C driver, such as adjusting the buffer lengths, moving the pin definitions to the *robot\_config* file and removing obsolete functions from the header file.

## Raw Measurement Data

The ground robots use the raw data from the gyroscope and accelerometer to determine the pose of the robot. The IMU was therefore included in the *task\_pose\_estimator.c* file. In the code of the quadcopter, a separate task called *task\_imu\_data* was defined for this purpose. During the development process, three modes for accessing the IMU data were integrated.

The first mode is more or less identical to the already existing procedure and uses the originally given functions. The IMU is initialised by selecting the clock source, activating the raw data interrupt, configuring the accelerometer and gyroscope and activating the low-pass filters for the sensor data. The sampling rates and scales for the sensors were experimented with, but otherwise no changes were made to the initialisation function.

Before the sensor data is analysed, the IMU is calibrated. One thousand samples of the *x*, *y* and *z* acceleration and gyroscope data are summed up. The average is calculated and saved as an offset. When the calibration is complete, the estimator task is notified by a queue message and the normal operation of the IMU is resumed. Each time a new sample is ready, the offset is subtracted from the data and the result is filtered using a window function, setting values below a threshold to zero. This is done to reduce the drift of the module. Different values for the threshold of the window function are tested and compared in chapter 6. The result is stored in a local variable that can be accessed via functions defined in the file *ICM\_20948.c* and is protected with a mutex. In addition, the gyroscope data is multiplied by the sampling period, converted into radians and added to the sum of the previous values. The result represents the angle relative to the stationary position during calibration. The task is configured to run at a period of 40 ticks ( $\approx 40$  ms).

One disadvantage of using these raw measurements is that the data must be read at a sufficient frequency. If the frequency is lower than the sampling rate set when initialising the sensors, data is skipped and the orientation estimate loses precision. In addition to the lack of accuracy, the drift of the IMU is also a major problem when precise orientation data is required. Another problem encountered during the implementation of the state estimation task is that the tasks interfere with each other. When the priority of the distance measurement task was increased to 5 to ensure accurate distance data was obtained, it was observed that the calculated heading was incorrect. The *task\_imu\_data* has a priority of 4, but when this is also increased, the distance measurements again lose precision. To overcome this problem, other options for accessing the IMU data were investigated. The IMU datasheet mentions an internal processor, the DMP, which uses a FIFO buffer and it was decided to test this.

## 5.6.2 Digital Motion Processor

The Digital Motion Processor (DMP) is a processor embedded in the ICM-20948 that is used for "offloading computation of motion processing algorithms from the host processor" [12]. It calibrates the accelerometer and gyroscope in the background and can process the data to provide quaternions. The results of the DMP can be accessed via the FIFO buffer registers. This method is being investigated because a high-frequency measurement method is advantageous when integrating the data to obtain the orientation from an IMU. The use of the FIFO buffer also enables the data to be accessed at a lower frequency which might solve the problem with the task interference. Access to quaternion data has also aroused interest, as many orientation calculations are based on quaternions or Euler angles. How exactly the processing is implemented and which algorithms are used by the internal processor is unfortunately not mentioned in the datasheet, but curiosity demanded that this method should be examined further.

The DMP has not yet been implemented, which is probably due to the very poor documentation. Although the existence and benefits of the DMP are mentioned in the IMU datasheet, the registers and memory files needed to configure the DMP are not provided. The manufacturer Ivensense provides an implementation of the DMP, but it is very difficult to follow and cannot be used without serious adaptations to the environment. As this problem is known, several users have uploaded a simplified version of the DMP configuration to GitHub. The version provided by SparkFun [21] includes partial support for the DMP. The DMP has been integrated into the quadcopter code by implementing the necessary functions from the SparkFun library while translating them from C++ to C. The functions are added to the file *ICM\_20948.c* and the data structures and variables for accessing the registers of the DMP are given by the file *ICM\_20948\_DMP.h* provided by the library. The most important functions are briefly described here:

**DMP Initialization** The initialisation function was translated directly from the library. It is quite complex and several other functions had to be implemented to configure specific elements of the sensor. In addition to selecting the clock source and configuring the interrupts, the accelerometer and gyroscope are activated and their sampling rate is configured. The DMP firmware is loaded by copying the binary data of the file *icm20948\_img.dmp3a.h* into the memory of the IMU. The gyroscope and accelerometer scale is set in the DMP's internal memory, as are other parameters for configuring the sensors.

**Enable Sensors** This function selects which sensors are activated and what type of data should be published in the FIFO buffer. The options include the compass, the gyroscope, the accelerometer, the game rotation vector and the orientation sensor. The game rotation vector and the orientation sensor provide quaternions. The memory registers for data output control, data ready status and control memory registers are set.

**DMP Setup** This is the function used to initiate the IMU with the DMP. The I<sup>2</sup>C driver and the DMP are initialised, the acceleration sensor and the gyroscope or the game rotation vector sensor are activated. The respective sensor period is selected and the FIFO and DMP are activated and reset.

**Read data** If the DMP is properly initialised and the sensors are activated, the data can be read from the FIFO buffer. The FIFO count register is examined to determine whether data is present. The type of data provided is specified by the header, the first element in the FIFO. For some sensors, the accuracy of the sensors and the calibration data are included. The function saves the data provided in a data structure and sets a flag if further data is

available in the buffer.

**Process output** Quaternions are retrieved when the game rotation vector or the orientation sensor is activated. This function calculates the Euler angles from the quaternions using the equation 2.24 and returns the result in a data structure.

### Integration in IMU Data task

As already mentioned, 3 different modes for accessing the IMU data were implemented in the task. The first mode was described in the previous section, the other two modes use the data provided by the DMP. While the second mode accesses the acceleration and gyroscope data published by the DMP, the third mode uses the orientation sensor to work with the processed data in form of Euler angles.

Instead of the `IMU_Init` function, the `DMP_Setup` function is used to configure the sensors. If the second mode is used, the offset value of the acceleration and gyroscope data is calculated in the same way as in the first mode, except that the FIFO buffer is accessed and the values are read until the buffer is empty. The same applies to the normal operation. The data is saved and can be accessed in the same way as in the first mode.

The third mode uses a different method, as the angles are returned instead of the angular velocity. The system waits until the results are stable and the resulting values are stored as an offset. The acquisition of the values is unproblematic, but the results are erroneous. Although angle values are returned and the drift of the angles is within an acceptable range, the system does not scale them correctly. If only the yaw angle is analysed, rotating the module by  $90^\circ$  results in an angular change of about  $55^\circ$ . A further rotation of  $90^\circ$  results in a change of  $120^\circ$ . The roll value also changes if the module is only rotated in the  $x, y$  plane, which should not happen. Due to the lack of documentation, it is difficult to find the source of the error. Since only the heading  $\theta$  is needed and roll and pitch are assumed to be zero, the Euler angles are not necessarily needed, so this mode is disregarded when implementing the pose estimator and the second option is used instead.

## 5.7 State Estimation

The integration of the IMU provides access to a measure of the orientation and the change in position. Together with the distance of the ultrasonic sensors, the results are used to estimate the pose of the module. The estimation is performed using an extended Kalman filter, which was developed in Matlab and later integrated into the `task_pose_estimator`. The resulting pose is sent to the server and used by the camera to calculate the global position of the extracted lines.

When designing the filter the ideas suggested by Gulaker in as were described in section 4.1 were taken into account. The angular dependency of the ultrasound sensors were problematic, as the ultrasonic distance is only valid for a measurement angle of  $\pm 15$  degrees. This means that if the sensor is not aimed straight at a wall, the measurements are distorted. This effect was only discovered during the tests of the first version of the filter. The results were too noisy and the filter simply could not work with the distance data in this way. The filter was therefore revised and a different model, which is described here, was implemented. It is very limited by the requirement that the ultrasonic measurements are only used when the estimated angle is



between  $-10^\circ$  to  $10^\circ$ ,  $80^\circ$  to  $100^\circ$ ,  $-80^\circ$  to  $-100^\circ$  and  $-170^\circ$  to  $170^\circ$ . This requires a rectangular room structure. The calibration must be carried out with a direct, straight gaze at a wall, so that the angle is correctly initialized at zero degrees. The estimation of the angle must also be very precise in order for this model to work. The acceleration data serves as a backup if the distance values are compromised and to improve the estimate.

The resulting model is described in the next section. This is followed by a description of the implementation process and the structure of the *task\_pose\_estimator*.

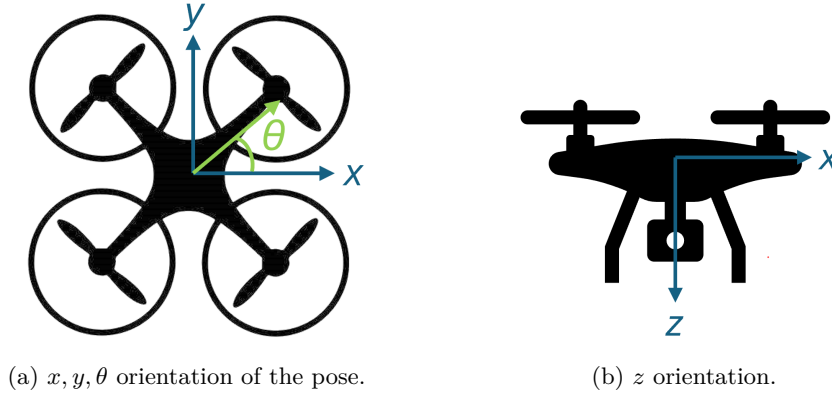


Figure 5.6: Pose of the quadcopter.

### 5.7.1 Model

The state of the system provides a measure of the position  $x, y, z$  and the heading  $\theta$  of the quadcopter (see Figure 5.6). As Gulaker suggested before, roll and pitch of the module are neglected and assumed to be zero. The acceleration data is used to determine the change in position and must be integrated twice. Therefore, the velocities  $v_x, v_y, v_z$  and the angular velocity  $v_\theta$  were added to the state variables. The resulting linear motion model is given in equation 5.1.

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ z_k \\ \theta_k \\ v_{x,k} \\ v_{y,k} \\ v_{z,k} \\ v_{\theta,k} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & dt & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & dt & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & dt & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & dt \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{F}_{k-1}} \mathbf{x}_{k-1} + \mathbf{w}_k \quad (5.1)$$

The variable  $dt$  is the rate of the estimation task. If the angle is fed into the system instead of the angular velocity, the value in row 4, column 8 must be set to zero.

The observations of the systems include the acceleration  $a_x, a_y, a_z$ , the angle or angular velocity  $g_x$  and the distance from the module to the front, left, right and top (or bottom)

$d_{\text{front}}, d_{\text{left}}, d_{\text{right}}, d_{\text{top}}$ . It is assumed that the distances refer to a common point on the module and that they are zero if the measurement angle is not within the valid range. The pose is estimated relative to the origin  $(0,0,0,0)$ . The distance values from the origin position are given by  $d_{\text{front, o}}, d_{\text{right, o}}, d_{\text{left, o}}$  and  $d_{\text{back, o}}$ . Even if the distance measurements are only valid within a narrow angular range, the angle still has an effect on the recorded distance. This has been incorporated into the resulting observation model

$$\mathbf{y}_k = \begin{bmatrix} a_{x,k} \\ a_{y,k} \\ a_{z,k} \\ g_{x,k} \\ d_{\text{left},k} \\ d_{\text{front},k} \\ d_{\text{right},k} \\ d_{\text{top},k} \end{bmatrix} = \begin{bmatrix} v_{x,k} \\ v_{y,k} \\ v_{z,k} \\ v_{\theta,k} \\ l_{\text{left}}(x_k, y_k, \theta_k, d_o) \\ l_{\text{front}}(x_k, y_k, \theta_k, d_o) \\ l_{\text{right}}(x_k, y_k, \theta_k, d_o) \\ z \end{bmatrix} + \mathbf{n}_k \quad (5.2)$$

where

$$l_{\text{left}}(x_k, y_k, \theta_k, d_o) = \begin{cases} (d_{\text{front, o}} - x) \sin \theta & -100 < \theta < -80 \\ (d_{\text{left, o}} - y) \cos \theta & -10 < \theta < 10 \\ (d_{\text{back, o}} + x) \sin \theta & 80 < \theta < 100 \\ (d_{\text{right, o}} + y) \cos \theta & 170 > \theta > -170 \\ 0 & \text{otherwise} \end{cases} \quad (5.3)$$

$$l_{\text{front}}(x_k, y_k, \theta_k, d_o) = \begin{cases} (d_{\text{right, o}} + y) \sin \theta & -100 < \theta < -80 \\ (d_{\text{front, o}} - x) \cos \theta & -10 < \theta < 10 \\ (d_{\text{left, o}} - y) \sin \theta & 80 < \theta < 100 \\ (d_{\text{back, o}} + x) \cos \theta & 170 > \theta > -170 \\ 0 & \text{otherwise} \end{cases} \quad (5.4)$$

$$l_{\text{back}}(x_k, y_k, \theta_k, d_o) = l_{\text{front}}(x_k, y_k, \theta_k - 180, d_o) \quad (5.5)$$

$$l_{\text{right}}(x_k, y_k, \theta_k, d_o) = l_{\text{left}}(x_k, y_k, \theta_k - 180, d_o). \quad (5.6)$$

## 5.7.2 Extended Kalman Filter

Since the observation model is non-linear, the extended Kalman filter algorithm has been used to update the state of the system. The observation model is linearised by calculating its derivative with respect to the current state  $\mathbf{x}$ . The resulting Jacobian matrix is:

$$\mathbf{G}_k = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \frac{\partial l_{\text{left}}}{\partial x} & \frac{\partial l_{\text{left}}}{\partial y} & 0 & \frac{\partial l_{\text{left}}}{\partial \theta} & 0 & 0 & 0 & 0 \\ \frac{\partial l_{\text{front}}}{\partial x} & \frac{\partial l_{\text{front}}}{\partial y} & 0 & \frac{\partial l_{\text{front}}}{\partial \theta} & 0 & 0 & 0 & 0 \\ \frac{\partial l_{\text{right}}}{\partial x} & \frac{\partial l_{\text{right}}}{\partial y} & 0 & \frac{\partial l_{\text{right}}}{\partial \theta} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (5.7)$$

Based on these equations, the EKF was initially implemented in MatLab. The system was tested with measurement data obtained from a log file generated with the J-Link RTT Viewer. Once the desired performance was achieved, the algorithm was translated into C and implemented in the `ekf.c` and `ekf.h` files. The C code is based on the extended Kalman filter functions used in the robot code, but has been modified to reflect the new model.

When initialising the Kalman filter, the constant matrices  $\mathbf{Q}$ ,  $\mathbf{R}$  and  $\mathbf{F}_{k-1}$  as well as the constant part of the  $\mathbf{G}_k$  matrix are defined. The variance matrices  $\mathbf{Q}$  and  $\mathbf{R}$  contain variances  $\mathbf{w}$  and  $\mathbf{n}$  on their diagonal, which were adjusted during the tests of the system. The initial uncertainty  $\mathbf{P}$  is initialised as an identity matrix, all other matrices and vectors, including the pose, are initialised with zeros. When the algorithm is executed, it is split into two parts:

The first part calculates the predicted state  $\check{\mathbf{x}}_k$  using the motion model  $\mathbf{F}_{k-1}$  (5.1) and the equation 2.15. The second part is based on the observations. The expected measurements and their derivatives are calculated as a function of the angle  $\check{\theta}_k$  predicted in the first step. The measured distances  $\mathbf{y}_k$  are filtered in the same way, so that they are zero, when the angle is out of their measurement range. As soon as the predicted measurement and  $\mathbf{G}_k$  are set, the Kalman gain is calculated. To determine the inverse, an algorithm is used that decomposes the Cholesky matrix and back-substitutes the corresponding vector which was provided by the robot code. Finally, the predicted state and its covariance are determined with the equation 2.18 using the measured distance, the estimated distance and the Jacobian of the observation model  $\mathbf{G}_k$ .

### 5.7.3 Integration in the system

The EKF is integrated into the `task_pose_estimator`. It communicates with the other tasks in the system via queues and uses semaphores when it accesses shared resources. When the task is initialised, the EKF is initiated and the pose estimate is set to zero. As the filter cannot work without the IMU data, it waits for the IMU calibration to be completed. In the meantime, the origin distance  $\mathbf{d}_o$  is determined by averaging the distance samples from the ultrasonic sensors. As there is no sensor pointing to the back of the module, only 3 out of 4 origin distances can be calculated.

As soon as the IMU has signalled that the data is ready via a queue, the state estimator checks whether the connection to the server has been established. If this is not the case, the task waits. If the connection to the gateway is established, the task notifies the `camera_mapping` task that valid pose estimates are arriving and begins with the estimation.

The task accesses the current samples of the ultrasound measurements  $d_{\text{front}}, d_{\text{left}}, d_{\text{right}}, d_{\text{top}}$ . As described in section 5.3, the distance data is already transformed to refer to the centre of the camera lens, so no additional transformation is required here. A mutex is set before the data is read in order to prevent interference with other tasks. If the measured angle is in the range of  $90^\circ$  or  $-90^\circ$ , the ultrasonic data provided is used to determine the distance of the back from the origin  $d_{\text{back,o}}$  using the equation 5.8. Before entering the data into the Kalman filter, it is filtered so that the previously determined distance is used if the data obtained is invalid.

$$d_{\text{back,o}} = (d_{\text{right,k}} + d_{\text{left,k}}) \cos(90 - \theta) - d_{\text{front,o}} \quad (5.8)$$

The acceleration data, the raw data from the gyroscope and the angle supplied by the *task\_imu\_data* are retrieved in the same way. As the acceleration data has not been pre-processed, it is multiplied by the sampling period of the state estimator and by the gravity constant  $g = 9.81 \text{ m/s}^2$ . The result is added to the current velocity estimate to obtain the current velocity input for the Kalman filter  $a_x, a_y, a_z$ . The angle or angular velocity ( $g_x$ ) has already been processed in the *task\_imu\_data*, so these are not processed further.

The resulting measurement data is passed to the EKF which returns the pose estimate. The estimate is stored in a variable so that the camera task can access it, and the position is sent to the server every 20 cycles to update the map. The message uses the update topic as the line topic is used by the *camera\_mapping* task and the messages should be separated for now to better analyse the system.

The *task\_pose\_estimator* has a priority of 4 and a period of 40 ticks which corresponds to a frequency of approximately 25 Hz.

## 6 Tests and Results

This chapter analyses the quality of the state estimation algorithm and the accuracy of the mapping. To achieve the best result, the sensor processing was optimised first. In the first section of this chapter, the performance of the ultrasonic sensors is presented for different system settings and sampling rates. It also looks at how the measurements behave when they are angled towards a wall, as this had a significant impact on the design phase of the Kalman filter. The precision of the IMU sensor in relation to the angle measurements is then examined, comparing the different modes implemented as part of this work. The state estimation algorithm is evaluated based on the optimal parameters determined during these tests. Subsequently, the line extraction of the camera is tested in a stationary environment to see how well the mapping works under optimal conditions. This chapter is concluded by showing how the mapping performs in a real environment using the pose estimate.

### 6.1 Ultrasound sensors

The ultrasonic sensors are used to determine the distance to the module's surroundings. The highest possible accuracy should be achieved so that the extended Kalman filter can generate a good estimate of the position of the quadcopter. The settings of the *distance\_meas* task, which include the frequency of the measurements and the priority of the task, as well as the delay between consecutive measurements by different sensors, were analysed in the following tests. The system was tested in terms of accuracy and reliability.

#### Measurement frequency and delay

First of all, it was necessary to determine the optimal measuring frequency of the ultrasonic sensors. If the frequency is too low, the positioning system cannot work with the latest data, which greatly affects the performance of the overall system. If the frequency is too high, the measurements obtained may be invalid because the sound has not yet decayed before a new measurement is started. The data sheet for the HC-SR04 sensors specifies the measurement duration of 60 ms for one sensor. As there are four sensors, the time period for the distance measurement task can be set to 240 ms or 250 ms to simplify the calculations. The time a measurement takes varies as it depends on the distance the sound signal has to travel. During the implementation phase, an attempt was made to incorporate a timer that starts the measurements at a regular interval. This failed as it was assumed at this stage that the 16 measurements per second corresponded to 16 measurements per sensor and not for all four sensors together. However, by adding a delay between the completion of one measurement and the start of the next, the same performance should be achieved as long as there is sufficient time for all measurements to be completed within the tasks duration. For the test, the module was placed in a box and all four sensors were activated. The length of the delay was varied and the results from the left sensor are given in Figure 6.1.

With a delay of 1 ms, the distance fluctuates between 22.36 cm and 23.65 cm. The maximum error during the measurement period is 1.45 cm. If the delay is increased to 2 ms, the average error decreases, but some outliers can still be recognised. The delay of 5 ms provides good results with a maximum error of 0.16 cm.

If the delay is increased to a very high value such as 40 ms, the performance decreases again as

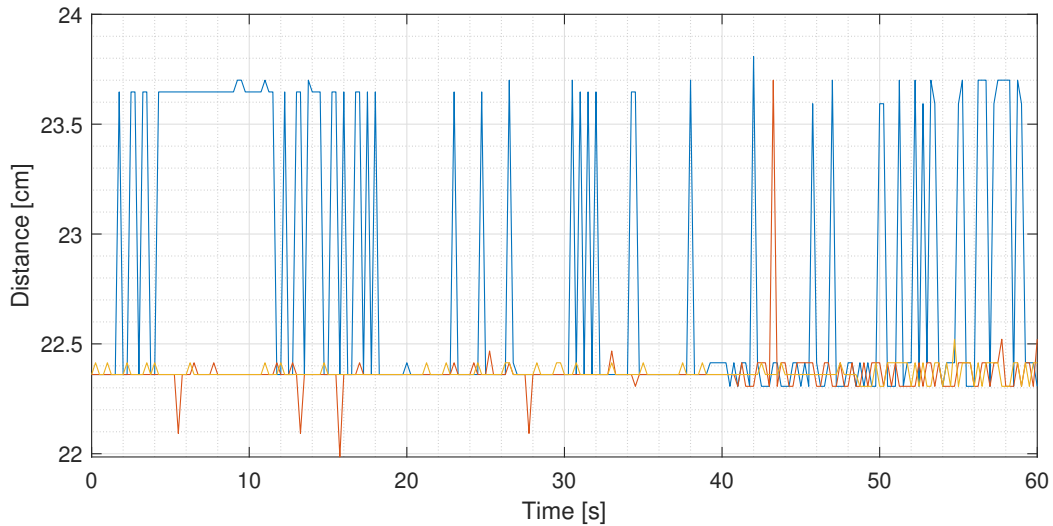


Figure 6.1: Distance of the left ultrasonic sensor with a delay of 1 ms (blue), 2 ms (red) and 5 ms (yellow) to previous measurements. The frequency of the task is 250 ms, so that 240 measurements were recorded during the period of 1 minute. The task is not scheduled with other tasks.

the tasks period is 250 ms and therefore most of the time is spent waiting. In the tested scenario, the walls were very close to the distance sensor, so the return time was quite fast. If the system is tested in a larger environment, the performance will probably already decrease at lower delays. Therefore, the delay of 5 ms was chosen.

## Angle

The next step was to investigate the effect that occurs when the sound signal hits the wall at an angle. The measured distance, the actual distance and the error are shown in the Table 6.1. The task duration is 250 ms and the measurement delay is 5 ms. Although all sensors were accessed, only the data from the front ultrasonic sensor is analysed here. It can be seen that the error gets worse with increasing angle and reaches a maximum of 3 cm at 45°. The variance of the measurements is also highest at this angle. At 70° to 90°, the variance is rather high in as well, even though the average error is very low. So, there does not seem to be a specific correlation between variance of the measurements and the angle.

	0°	10°	20°	45°	70°	80°	90°
$d_{\text{real}}$	27	27.5	28.8	33.2	25	23.6	23.3
$\mu_{d,\text{meas}}$	26.891	27.844	27.85	36.427	23.69	24.056	23.384
$\sigma_{d,\text{meas}}$	0.0017	0.0057	0.0108	0.1740	0.0438	0.0461	0.0361
$e$	-0.1095	0.3442	-0.9497	3.2271	-1.3076	0.4549	0.0839

Table 6.1: Mean, variance and error of the distance measurements at different angles. The period of the task was 250 ms and the delay was 5 ms, with the `task_distance_meas` being the only running task.

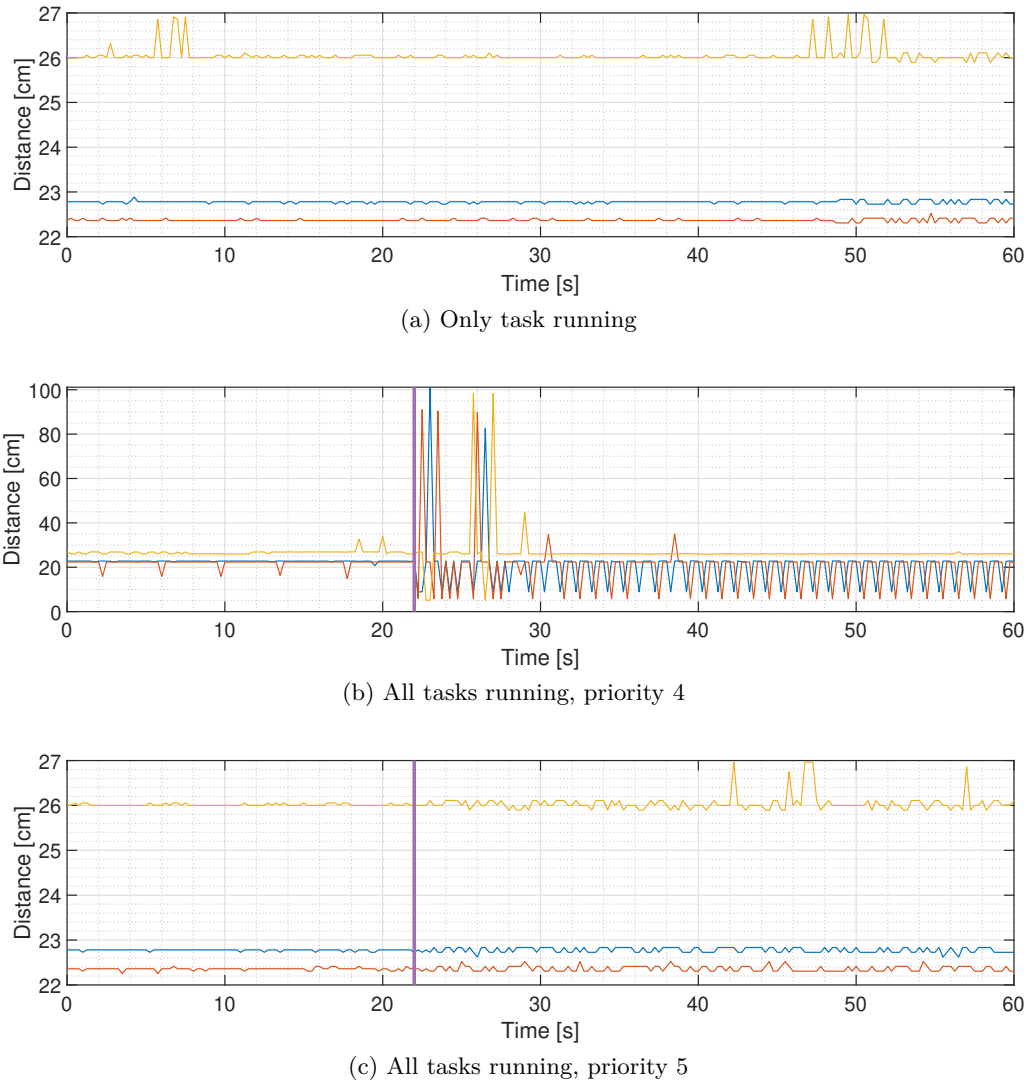


Figure 6.2: Distances of the right (yellow), front (blue) and left (red) ultrasonic sensors in relation to the origin of the camera lens. The tests are carried out under stationary conditions and with different system settings. The purple line at 90 samples marks the end of the IMU calibration and the start of the position estimator.

### Scheduling with other tasks

When testing the state estimation algorithm, which is based on the distance measurements, unexpected errors occurred. While the measurements of the sensors were very accurate when it was the only running task, they deteriorated significantly when they carried out together with the rest of the system. The measurement results for different task settings are compared in Figure 6.2.

For the test, the module was again placed in the box while pointing straight at the wall. The data was collected for 1 minute in a stationary state and the task rate was set to 250 ms with a delay of 5 ms between each measurement. The distance shown was already transformed to the origin of the camera, so a distance of 5 cm for the left sensor corresponds to an invalid measurement of zero as it is equal to the offset.

Figure 6.2a shows the performance when the `task_distance_meas` is the only running task. The measurements are very stable with an error of less than 1 cm. The state when the task is running together with other tasks is shown in Figure 6.2a. Errors of up to 78 cm are received and many invalid measurements are obtained. In particular, after the IMU has completed its calibration, which is indicated by the purple line at 90 samples, the measurements regularly exceed the expected distance or are invalid. The solution of this problem was to increase the priority of the distance measurement task to 5. The results shown in Figure 6.2c are very similar to the original measurements. The error is only slightly higher and only starts after the pose estimation task begins its execution.

## 6.2 IMU

The IMU provides acceleration and gyroscope data. The Kalman filter uses both, but the gyroscope data is integrated beforehand to obtain the angle of the module. In this section, different settings for processing the gyroscope data are analysed.

### Frequency

Two different modes have been implemented for accessing the data. In both cases, the `task_imu_data` runs at a frequency of 25 Hz (40 ticks period). So if the raw measurement data is retrieved directly from the registers of the ICM-20948, a new measurement is recorded at this frequency. When using the DMP, the sampled values are published in a buffer that can be accessed with a lower period. The DMP can publish the gyroscope data at a maximum rate of 1.1 kHz, but the system is too slow to process this frequency. If the frequency is lowered to 275 Hz or 220 Hz, the data can be accessed, but lower priority tasks will starve in the meantime. Therefore, the frequency of 100 Hz was chosen for the gyroscope measurement values. The acceleration data is published to the buffer at a frequency of 56 Hz as it is not directly integrated.

### Window

Once the calibration of the IMU has been completed, the current angles can be calculated by summing the gyroscope values. This integration leads to a drift in the angle. To reduce the drift, a window function is used that excludes inputs below a certain threshold. Figure 6.3 shows the drift of the angles over a period of 5 minutes when different window thresholds are applied. During the tests, the module remained stationary. The graphs only show one test cycle, so they do not represent the expected drift, but give an indication of how the drift is affected.

Nevertheless, conclusions can be drawn from the results. It can be seen that the drift is in most cases lower when using the raw data than when using the DMP. This can be explained by the lower frequency of raw data collection, as fewer samples correspond to less drift. The use of a window function significantly improves the stability of the measurements. However, it must be ensured that the window is not too large or in valid data will be lost. Then the change in angle can no longer be reproduced accurately.



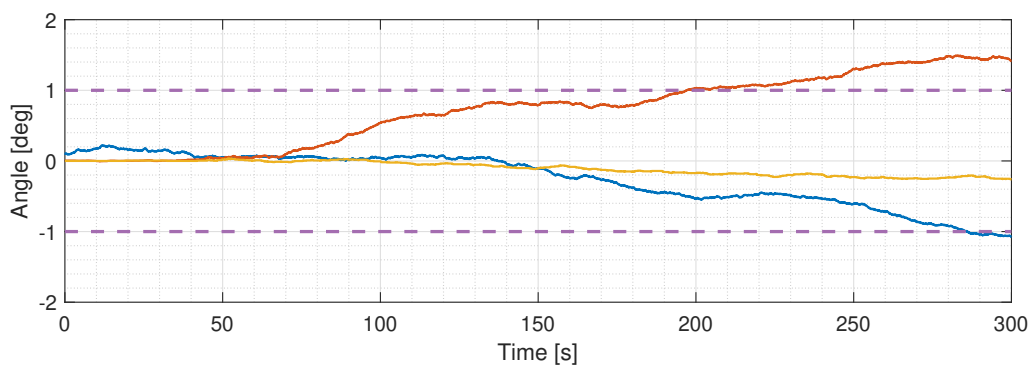
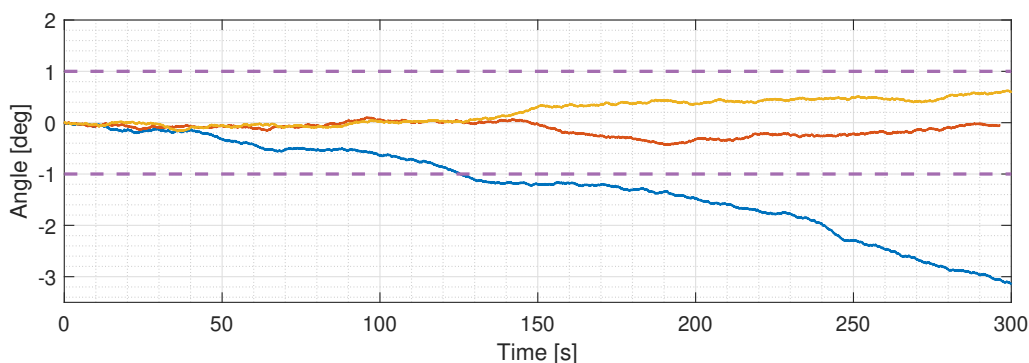
(a) Raw data,  $f = 25$  Hz(b) DMP,  $f = 100$  Hz

Figure 6.3: Drift of the IMU under stationary conditions using the raw data (a) and the DMP (b). The results for a window threshold of 0 deg/s (all data are accepted) are shown in blue, for 0.01 deg/s in red and for 0.1 deg/s in yellow.

## Precision

The accuracy of the calculated angles was analysed by positioning the module at different angles and noting the respective measurement results. All tests were carried out with a window limit of 0.1 deg/s, as this provided good results in the previous test. The priority of the `task_imu_data` also plays a role here. In the Table 6.2 you can see the measurements in raw data mode at different angles.

The first row shows the results when the task priority is 5, the same as the `distance_meas` task. There is an offset of  $2^\circ$  for an angle change of  $90^\circ$ . However, if the task is executed with this priority, there are problems with the distance measurements, which become very unstable. The second line shows what happens when the priority is set to 4. The measurements show a rather large error of  $13.2^\circ$  with an angle change of  $90^\circ$ . In the last row it is shown that the DMP can work reliably at priority 4. The processor ensures a precise measurement frequency and access can be delayed without causing inaccuracies. The error is in most cases below  $1^\circ$ .

Mode	Priority	10°	20°	45°	70°	80°	90°	180°
Raw data	5	9.5	18.9	44.5	67.3	77.9	88.6	173.4
Raw data	4	9.2	18.1	36	56.8	66.4	76.3	140.3

Mode	Priority	10°	20°	45°	70°	80°	90°	180°
DMP	4	9.7	19.6	44.7	71.9	82.1	90.9	176.8

Table 6.2: Measured angles in degrees using the raw data and DMP mode for data acquisition at different positions.

As the module was placed by hand, small deviations of the angle from the desired value are to be expected. As with drift, the response at different angles varies slightly between tests. The speed of movement when carrying out the tests also influences the measurements.

### 6.3 Camera and server

A milestone of this work was the visualisation of the lines received from the camera on the server's map. In this section, the accuracy of the camera mapping is evaluated using the height measured by the upper ultrasonic sensor and the  $x$  and  $y$  coordinates provided by the server's control message. By merging measurements at different positions, the correct transformation from server to camera coordinates is also validated. In all tests, the *task\_pose\_estimation* and the *task\_imu\_data* were disabled. The results represent the state that can be achieved with an optimal pose estimate.

#### Test setup

The test setup for the camera is shown in Figure 6.4a. The module was attached to an overhanging beam so that the camera points downwards towards a maze segment. The camera is located at a height of 1.12 m, and the labyrinth segment is 53 cm wide, 42.8 cm long and 30 cm high at the outer edges. The walls of the segments are 1.5 cm thick. For the first test, the segment was placed directly under the camera (Figure 6.4b), for the second test it was moved forwards and backwards (Figure 6.4c). During the tests, the nRF52 and the camera are connected to a PC so that the images from the camera can be visualised in the OpenMV IDE and data can be accessed for debugging purposes.

When the module was pointed downwards, problems with the camera connection were detected. The camera system stopped responding from time to time. This problem already occurred during the implementation and merging process, but should have been fixed by the new communication structure. The errors are therefore most likely due to poor cable connections between the camera and the nRF52.

#### Camera precision

The first test aimed to analyse the accuracy of the line extraction of a single image processed by the camera. The position of the camera is sent via the server command and was set to (0,0). The corresponding snapshot was exported from the framebuffer in the OpenMV IDE and is shown in Figure 6.4b. The line data transferred to the server was extracted and displayed in MatLab (see Figure 6.5). The length of each line is given in the legend next to the image.

The black line at the top represents the length of the box (40.11 cm) and has an error of 2.8 cm.

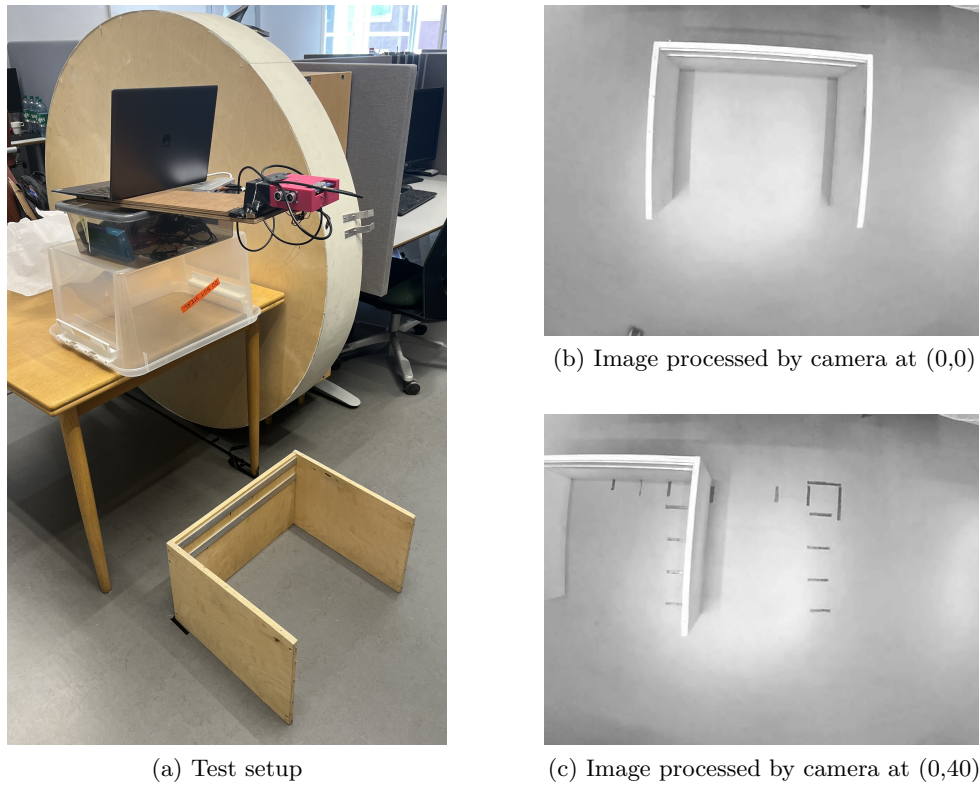


Figure 6.4: Test setup of the camera measurements and images captured by the camera, which were extracted from the OpenMV IDE during the tests.

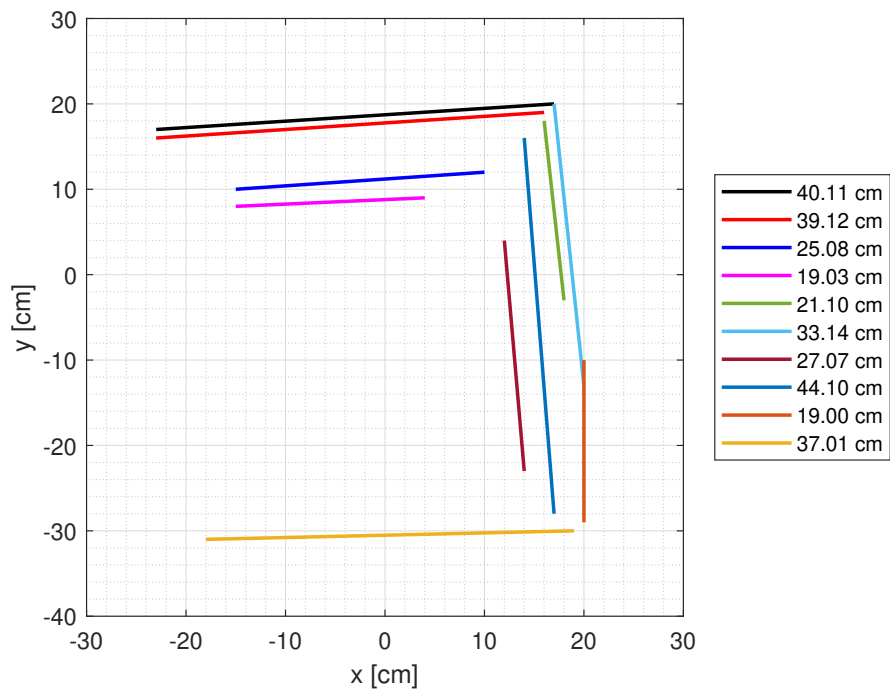


Figure 6.5: Lengths of transmitted line segments at origin position (0,0).

The sum of the length of the light blue line and the dark orange line is 52.1 cm, which corresponds to an error of 0.9 cm in width. The camera also extracts parts of the lower edges of the segment and the shadow of the wall. These are represented by the dark blue, pink, dark red and medium blue lines. Ideally, they should not have been recognised. It can be concluded that the accuracy of the measurements is sufficient. As the test setup has shortcomings in terms of lighting conditions and the straight positioning of the segment, better results were not expected.

### Merging images

To see what results can be achieved when the system is based on an optimal pose estimate, the same test setup was used. However, the line segment was moved in  $x$  and  $y$  directions, and the position change was specified by the server message sent to the camera. In this way, from the camera perspective, it looks as if the segment is stationary and the camera is moving. After each movement of the segment, the position is sent, a picture is taken and the map is updated.

The results of the tests are shown in the Figures 6.6 and 6.7. As before, both the upper and lower edges of the segment are recognised in the images at the origin. If the position is changed by 10 cm in the  $y$  direction, the extracted lines overlap. The further the segment moves, the more the bottom edges of the segment are recognised. At the end position (0.60), only the right edge of the maze segment is in the image frame, so only the right wall is updated. In general, the performance is quite good. Small errors are recognisable, but these were expected. The structure of the maze segment is clearly discernible.

If the  $x$  position of the segment is changed, similar results are achieved. The bottom edges are recognised at all five positions. Changing a direction by 90 degrees was also tested (see Figure 6.7f). It proved difficult to rotate the segment around the origin of the camera lens, so the placement of the segment is not good. The map shows that the new image is shifted too much to the left. However, it can be seen that the orientation transformation works as the segment is rotated correctly on the map.

## 6.4 Pose estimate

The pose estimation is based on both the angle from the IMU data and the distance from the ultrasonic sensors. The model described in the implementation chapter intended to use the accelerometer data to determine the position, in addition to the distance data. Unfortunately, the accelerometer data could not be processed correctly. The value of the integrated data was constantly increasing, so that the estimate deviated greatly from the actual position. Therefore, the accelerometer data was ignored in the tests and only the ultrasonic data was used to determine the position.

The measurement model was also adjusted so that the derivation of the distance measurements with respect to the heading was not taken into account. It led to a strong deviation of the angle and was therefore ignored. The angle was determined using the DMP version of the `task_imu_data`.

Figure 6.8 shows two different tests regarding the estimation. The module was placed inside the maze segment and the distance to all three walls was measured. During the tests, the module was moved in a rectangle given by lines drawn on the ground between the walls. This made it possible to compare the results with the real position. The tests started and ended at the bottom left corner of the box.

The first test estimated the position while the module was facing the front wall and did not

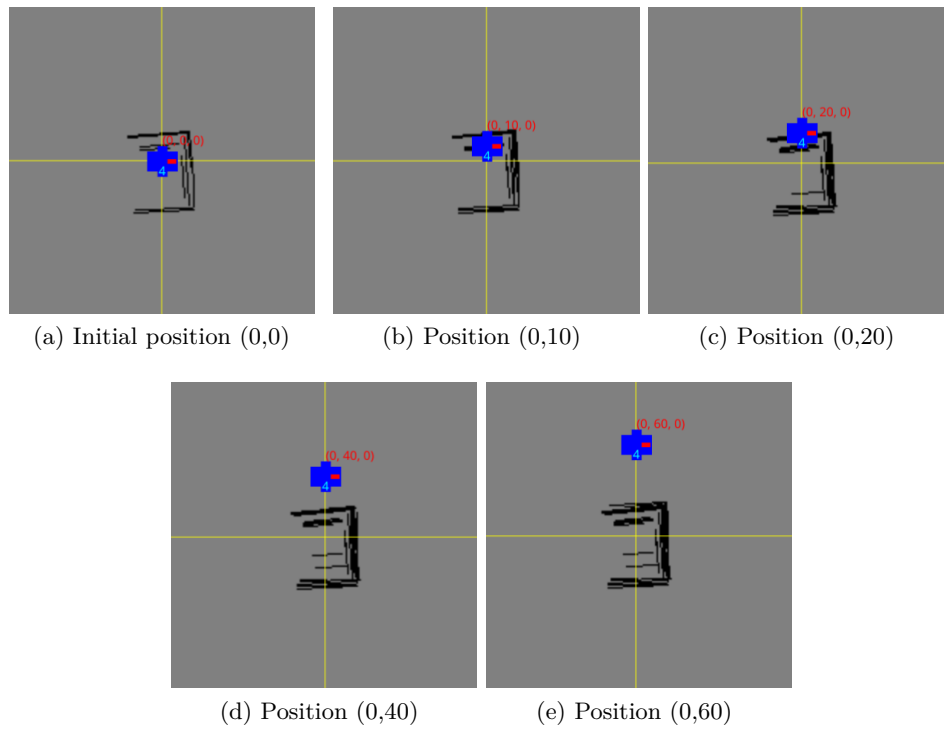


Figure 6.6: Map shown on the server after each movement in positive  $y$  direction.

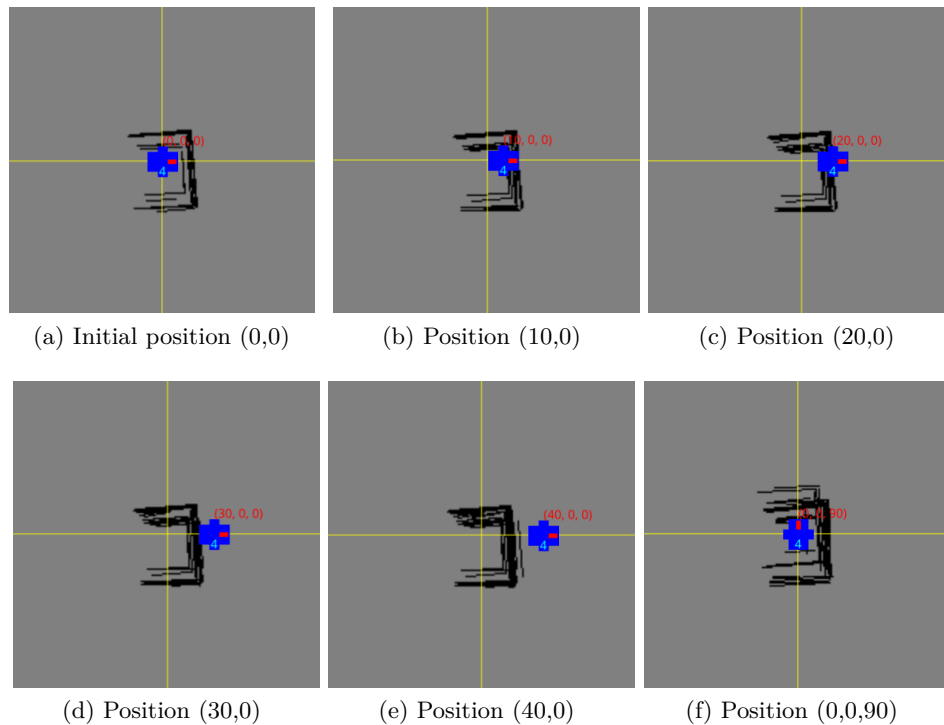


Figure 6.7: Map shown on the server after each movement in positive  $x$  direction (a)-(e) and when the segment is turned by  $90^\circ$  (f).

rotate. In the second test, the angle was changed. For this, a fourth wall was required at the rear of the segment in order to be able to determine the distance to the rear during the rotation. Therefore, another wooden wall was attached to the open side of the labyrinth segment. The module was first moved along the x-axis, looking forwards. When it reaches the corner, it is rotated  $90^\circ$  in the positive y-direction. This direction is maintained until the segment reaches its starting position, where it is rotated back by  $90^\circ$ . This test was used to check the accuracy of the angle estimation and the accuracy of the calculated distance between the origin and the back of the module.

The variances of the model parameters used in the tests are given in Table 6.3. These were selected based on the results of the tests for different settings.

	$x, y, z$	$\theta$	$g_x$	$d_{\text{front}}, d_{\text{left}}, d_{\text{right}}, d_{\text{top}}$
<b>Variance</b>	0.0000001 m <sup>2</sup>	0.0001 deg <sup>2</sup>	0.01 deg <sup>2</sup>	0.001 m <sup>2</sup>

Table 6.3: Variances of the state and measurement variables as defined during the pose estimation tests.

The results of the first test (see Figure 6.8a) show that the position estimate has a maximum error of around 1.5 cm. There are some outliers, but the estimate is mostly very close to the actual position. For the second test (see Figure 6.8b), two test runs are shown. In both tests, there are some outliers with an error of 5 cm. However, the change in angle does not seem to have a large impact on the measurements as they were performed in the lower right corner and the lower left corner.

Overall, the performance of the state estimation is not outstanding, but sufficient considering that an error of 1 cm can be expected from the ultrasonic sensors themselves and the system is additionally affected by movements.

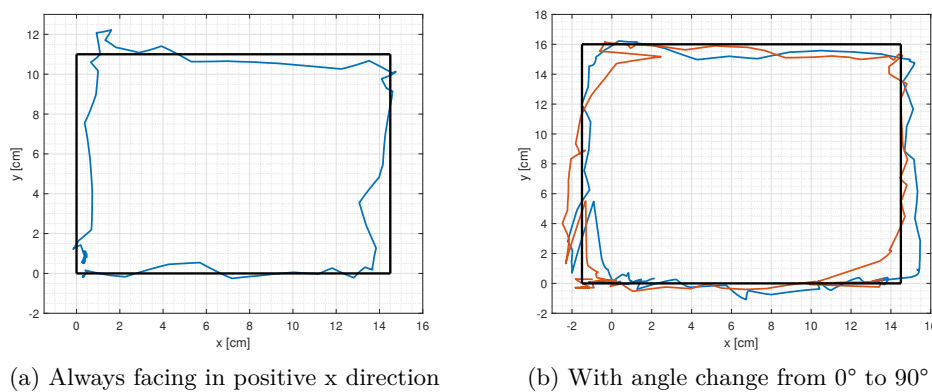


Figure 6.8: Test of state estimation within the maze segment. Figure (a) shows the estimated position along a square while pointing straight ahead at the x-wall. Figure (b) shows two test trials of the estimation when changing the angle to  $90^\circ$  and back when reaching the corners of the segment. The black square illustrates the actual position along which the module was moved.

## 6.5 Entire system

Finally, the entire system was tested together. The camera operates with the estimated position to calculate the coordinates of the lines that are displayed on the server. The tests were carried out in a room with a width of 2.42 metres. A length of 2 m was not exceeded during the tests. The room contained some elements protruding from the walls, such as windows, a heater and a blackboard. The distance measurements are therefore affected by these circumstances. It can lead to an error of up to 10 cm if the module measures the distance towards these elements instead of the wall.



Figure 6.9: Test setups where the position estimate is used as position reference when creating a map of the room.

The module was mounted on a beam so that the camera and the upper ultrasonic sensor have a clear view of the floor. Both the camera and the module were connected to a laptop, which was

also placed on the beam (see Figure 6.9a). This allows the person holding the beam to control the programme.

The camera mode was set to continuous operation so that the next image is captured and processed immediately after an image has been analysed. The parameters and settings of the position estimator are the same as in the previous section. An attempt was made to keep the orientation of the module close to zero so that the position determined by the ultrasonic sensors remains valid throughout the test. The three test setups are shown in Figure 6.9b-d. In the first test, the maze segment was placed in front of an upturned table so that the distance to the front sensor was not affected by the protruding windows in the background. The module was moved forwards and backwards and from left to right, but so that the table always remained in sight of the front sensor.

The resulting map after a few frames each is shown in Figure 6.10. Comparing the first four images, the left and right walls are quite well aligned. The mapping of the front wall is less successful. Three lines are recognised here that are very close to each other, namely the outer edge at the top, the inner edge at the top and the inner edge at the bottom. When moving forwards, the outer edge at the bottom is also recognised. As these lines do not overlap perfectly on the map, the result is a tangle of lines.

Figure 6.10e shows a case in which the module has measured the distance to the wall of the segment and not to the floor. The distance is reduced from 120 cm to 90 cm, so that the size of the segment is assumed to be smaller than it actually is and therefore appears shifted in the map.

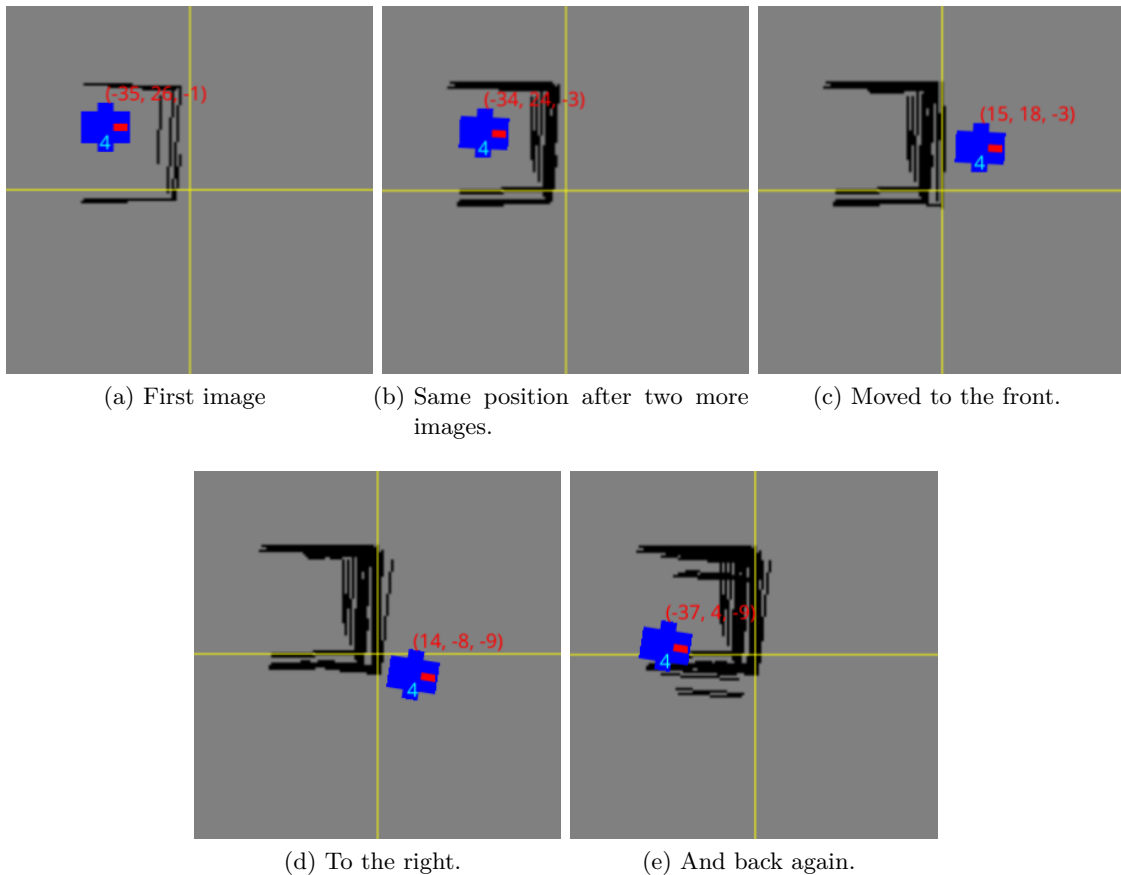


Figure 6.10: Resulting map of first test.



The second test uses the setup in Figure 6.9c. Here, the maze segment was placed in the centre of the room and the position estimate is based on the distance to the walls on the left, right and front. The results are shown in Figure 6.11.

As can be seen, two robots are displayed on the map. The one that moves between the different images is based on the position sent with the line messages from the camera. The other one should represent the estimated position which is transmitted using the update topic on the server. This obviously failed as the robot remains stationary. The error was that the server was expecting the position in millimetres, while the position estimator was sending the position in centimetres. This problem was fixed after the tests were performed.

The room was walked through twice and one to three snapshots were recorded at each position at a height of approximately 120 cm. The fact that it takes a short time for a snapshot to be taken after the position has been sent to the camera caused difficulties. The module must remain stationary during this time and can only be moved while the image is being processed. An attempt was made to take this into account during the tests, but it could not always be adhered to. An example of this is shown in Figure 6.11e near the origin of the map. The module was moved from the centre close to the front wall to the top left corner before the snapshot was taken. Therefore, the line segments that should represent the left wall are displayed in the centre of the room.

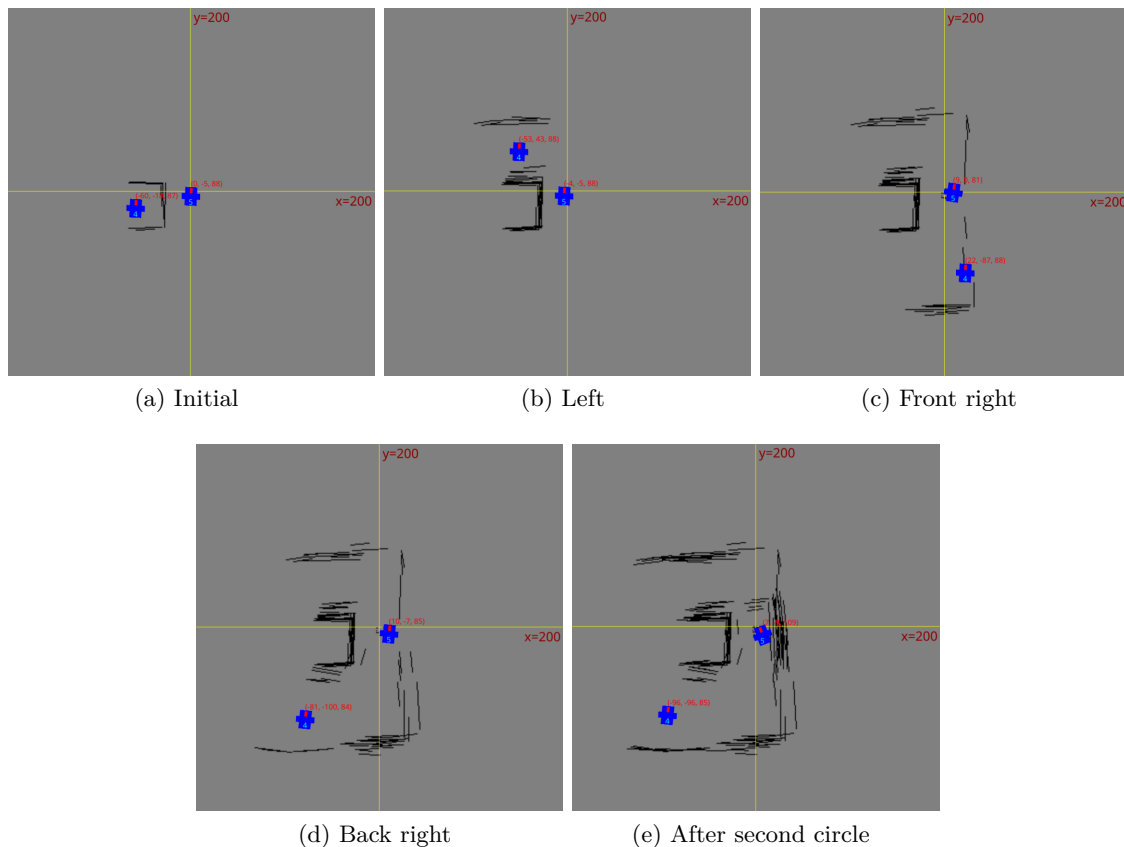


Figure 6.11: Resulting map of second test.

In the last test, an attempt was made to map the structure of smaller objects in addition to the maze segment. This was not successful as the objects were too small and were filtered out during image processing. The resulting map shows some independent lines in the room, but does not depict the objects correctly. As the result is otherwise similar to the results of the second test, it

is not shown here.

In general, it can be said that the mapping works as long as the objects are large enough. The spatial structure and the maze segment are clearly recognisable in the map created in the second and third test. If only one image was taken at one position, the mapping would probably improve as the overlap would be reduced.

## 7 Discussion

The objective of this work was to combine the previous work into a single system that can be used to map a structure in an indoor environment and can be carried by a quadcopter. This was achieved - with limitations. The performed tests clearly show that the estimated pose for the camera is precise enough to process the images of the maze in such a way that a rough map of the space can be created. The developed module is limited by several factors, which are now mentioned to assess the expected performance of the system:

- The system is executed by a microcontroller with limited memory and limited processing speed. This restricts the possibilities of what kind of algorithms can be implemented. The fact that the entire system is written in C also makes it difficult to integrate complex processing techniques. For example, vector and matrix calculations are very tedious to implement, other than in Python, for example. Advanced state estimators such as the iterated extended Kalman filter (IEKF) are also complicated to implement due to the limited processing speed and the risk that the frequency of the estimator must then be reduced.
- The ultrasonic sensors and the IMU are at the lower end of the price spectrum and therefore only have limited capabilities. Whether ultrasonic sensors are even suitable for this task is discussed in a section below, but it should be noted that it is a challenge to design a system with components that are subject to both frequency and angle constraints.
- The hardware uses the standard development kit instead of a customised PCB for the system, so that the sensors are connected to the pins on the DK using standard PCB jumper cables. The WAGO clamps used as power distributors are also theoretically optimised for thicker cables. These circumstances restrict the reliability of the connections, so that a lot of time had to be spent debugging the hardware - especially the camera connection.

Given these limitations, the expectations of the system's performance were limited. It was clear that developing this embedded system would be a challenge, but one that was gladly embraced. An attempt was made to offload as much processing as possible by utilising the DMP and internal CPU of the OpenMV Cam to reduce the task volume of the microcontroller. This will hopefully help when implementing the control of the quadcopter in future work.

Nevertheless, several errors were detected during the tests, which will be discussed in the next sections. The advantages and disadvantages of using acoustic sensors for pose estimation and the design decisions when integrating the IMU are critically evaluated after comparing the results with previous work. Finally, possible consequences and limitations of the design for future work are highlighted.

### 7.1 Comparison to results from previous work

The test results of the ultrasonic measurements and camera accuracy are compared with the results of previous work presented in chapter 4. Although the developed system is based on this work, it has been significantly extended so that differences should be discovered. Ideally, the accuracy should have been improved by the integration of additional functions.

Bjerke has discovered an error of 1 cm in stationary tests and an error of 3 cm when moving the

module based on the measured distance of the ultrasonic sensors. The actual position during the tests was determined using a motion capture system, so it can be assumed that the results are more accurate than the manually performed tests in this thesis.

The steady-state error when the task `distance_meas` is performed as part of the overall system is between 0.16cm and 1cm. Therefore, the results match. If the data is used as part of the EKF and the system is moved, the pose estimate based purely on this data is around 1.5 cm. Changing the angle of the system increases the error to about 5 cm, which exceeds Bjerke's measurements. However, the average measurements have an error of less than 1.5 cm. The outliers shown in Figure 6.8b could also have been caused by the inability to precisely follow the predetermined line that defines the reference values in the corners of the box.

All in all, the results are similar and the average error has improved slightly.

The accuracy of the line lengths determined by the camera was around 3.5% in Vormdal's tests. To achieve this percentage, the GSD was manually adjusted for different heights of the system. The current code contains an offset of 30 cm from the  $z$ -coordinate to compensate for elements protruding out of the GSD. Considering the results from Figure 6.5, the maximum error was about 6%. The system used the distance from the top ultrasonic sensor as the height instead of the exact distance and did not adjust the GSD. Even though the error is worse than described in Vormdal's report, the error of 3 cm in one frame is acceptable.

## 7.2 Choice of ultrasound sensors for distance measurements

The use of acoustic sensors to determine the distance to the module's surroundings definitely had some disadvantages and complicated the design process of the state estimator. First of all, the measurement frequency is very low. Four measurements per sensor per second is a very low rate considering that the pose estimator operates at a frequency of 25 Hz. Normally, the sampling rate of a sensor should be around 20 Hz to allow accurate measurements when a system is moving. At the current measurement frequency, it is questionable whether the system can provide accurate enough data to serve as a position reference for a quadcopter flying at higher speeds. The frequency of access to the ultrasonic data could have been improved by assigning a separate timer to each sensor. However, all timers would then be occupied and would not be available for new tasks. In addition, the sound would interfere with the other measurements, which - even though it has not been tested - would very likely lead to many invalid measurements. Another problem is that the sound moves slowly. If the quadcopter is moving, the return signal may not be recognised. Therefore, the position could only be detected when the quadcopter is hovering at a stationary position.

The most troublesome property of the ultrasonic sensor is its angular dependence. This made the development of the state estimator a laborious endeavour and led to undesirable limitations with regard to the environment. One idea was to recognise the invalidity of the acquired data by calculating the variance of the measurements. This was based on the idea that the reflections must lead to more inconsistent data. However, the test to analyse the angular dependence showed a higher variance even at small angles, so this idea was not pursued further. However, this test was performed in a box. It would probably have been more meaningful to analyse the measurement if it had been directed at a long wall. The corners of the box reflect the signal back to the sensor, so the distance received is still quite accurate. If another test measuring the reaction against a wall is successful (in the future), the possibility of using the variance to determine when the data is valid could be considered.

The main reason for choosing ultrasonic sensors was primarily the price of the sensors, which is indeed very low. Another argument was that the distance can also be measured through smoke, but since this is unlikely to be the case indoors and the camera algorithm would not work under these conditions anyway, this is a very weak reason. Whilst it is interesting to develop systems with higher limitations, consideration should be given to changing the sensor type. One idea would be to use a LIDAR, even if this is quite expensive. The estimation of the pose would most likely improve significantly.

## 7.3 Evaluation of IMU

A great deal of time and effort was invested in obtaining an estimate of the orientation from the IMU. In particular, the integration of the DMP was complex and time-consuming. The advantages and disadvantages of the DMP are discussed in this section.

The main reason for implementing the DMP was to outsource the data processing and to have access to the quaternion vectors. What kind of data processing the DMP performs remains unclear, and the quaternion vectors were not used in the final version because the scaling was wrong - or simply not understood correctly. Ultimately, the DMP was used primarily because it publishes the data on the FIFO buffer so that it can be accessed at a lower frequency. The same could probably have been achieved by configuring the FIFO buffer so that the raw acceleration and gyroscope data could be accessed through it without using the DMP. This was not considered before the DMP was integrated.

The DMP requires a lot of memory from the nRF52, as an image of the internal memory must be loaded into the IMU when the DMP is initialised. In addition, the register definitions in the dmp header file alone comprise 700 lines of code and the functions for accessing the DMP are implemented in 1500 lines of code. That is a lot of data for a minimal benefit. If the quaternions had been used, it would have been worth it, as access to these is otherwise difficult to provide and they can also be used to estimate the pitch and roll of the quadcopter. These parameters are completely ignored in the current state estimator, but could be needed as the movement of the quadcopter leads to a change in both parameters. For the current application, it must be said that the DMP is overkill for the constrained environment.

Furthermore, the time needed to integrate the DMP to improve the orientation estimation would have been better invested in processing the acceleration data. No accurate velocity could be extracted from the data, so the state estimator has to rely solely on the ultrasonic data. The additional information from the accelerator could have been used very well to compensate for the low measurement frequency and the angular dependence of the ultrasonic sensors.

## 7.4 Limitations

The current system has the limitation that the area in which it is used must be rectangular and should be surrounded by hard walls that do not absorb the sound. The module must be placed so that it is pointing straight at a wall so that the angle used for the ultrasonic sensors is correctly calibrated. One way to simplify the algorithm, remove the angle restriction for the ultrasonic sensors and improve pose estimation would be to restrict the movement of the quadcopter so that it is always facing forwards. As it has four rotors and can therefore fly in all directions without turning, this could be a viable option.

Another requirement is that the quadcopter hovers in a stationary position when the camera is in operation. This is due to the long delay that exists in the camera between receiving the capture command and the time the snapshot is taken. The ultrasound data should also be captured in a stationary state as this would be more accurate.

The camera does not recognise objects shorter than about 20 cm when it is at a height of 120 cm. This can be configured in the OpenMV code settings. If the variable is reduced, image processing takes longer and the processor could reach its limits. Another problem is the resolution of the captured images, which is currently limited to 640 x 480 pixels. Small objects cannot be recognised and exact line coordinates cannot be calculated if the height is too large. To improve the resolution of the images, a newer version of the camera with significantly more RAM and computing power could be considered. However, this will not help if the state estimation does not improve as well.

The system should also be limited to process only one image at one position. The lines would overlap less on the map so that the structures would be more clearly recognisable. Of course, there would still be problems at the edges of the images, but overall the map would probably be improved.

One final comment should be made regarding the hardware connections between the module's sensors: As noted with the camera connection during testing, the connections are not the most reliable. Especially when you consider that the module is to be attached to a quadcopter. This will be exposed to vibrations and stronger forces, it is questionable whether the module will hold as it is designed now. It should therefore be considered to revise the structure of the case before attaching it to a quadcopter.

-

## 8 Further work and conclusion

There are several improvements that can be made to the final version of the presented module. Some include new components, others modifications to the current hardware or changes on the software side. These are presented in this chapter, followed by a brief conclusion.

### 8.1 Further Work

#### 8.1.1 Hardware

The hardware of the module has worked so far, but some components should be replaced in the future. For example, the DHT22 temperature sensor is broken and needs to be replaced. The addressing for the sensor is already implemented and it is very cheap, so the integration should be effortless. The error caused by the temperature change is not large, but should be taken into account if possible.

The sockets for the pins on the OpenMV Cam M7 are not well connected and should also be replaced to ensure that communication is not interrupted or fails, as was observed during the tests. If the module is connected to a quadcopter, the housing should be improved and it should be ensured that all connections between the modules are stable and not affected by the vibrations of the drone. In addition, a battery should be added to the module so that it can be used as a stand-alone system.

Consideration should also be given to installing the newer version of the camera (the OpenMV Cam RT1062). It has significantly more memory so that images can be recorded and processed at a higher resolution. This would result in a more accurate map, but this would only be of benefit if the state estimation is also improved.

#### Distance sensors

Another way to improve the state estimation would be to use other sensors for distance measurement. For example, a LIDAR could be used instead of the ultrasonic sensors to obtain more accurate distance data. The low measurement frequency and the angle dependency of the ultrasonic sensors would then no longer be a problem. LIDARs have a higher measuring frequency, can measure large distances and are very precise. To use the LIDARs to determine the position of the module, 3 sensors should be connected (one pointing down, one pointing left and one pointing right). One could also argue to use only two LIDARs (one pointing left, one pointing right) and keep one ultrasonic sensor for height measurement. Models can be purchased at prices between 40\$ and 150\$, which is a considerable increase over the 5\$ HC-SR04 sensors. The ‘Garmin LIDAR-Lite v4 LED’ [11] could be considered. It has the following features:

- 5 cm to 10 m range
- $\pm 1$  cm accuracy, for measurements below 1 m to a 90% reflective target
- I<sup>2</sup>C interface for accessing the measurement data in centimetres
- Frequency of 200 Hz

As the results of this work show, satisfactory performance can also be achieved with the ultrasonic sensors. If further work is to be carried out with them, a fourth ultrasonic sensor should be added to the rear of the module. This would compensate for the errors of the front sensor, increase the range of wall detection and simplify the state estimation algorithm. The disadvantage would be that the measurement frequency of all ultrasonic sensors would be reduced even further.

### 8.1.2 Software

On the software side, several improvements can be made with regard to the state estimation. Firstly, an attempt should be made to correctly process the acceleration data from the IMU and include it in the EKF. If the module is connected to a quadcopter, the signals for controlling the drone can also be used to further improve the state estimate, as the motion model could be extended. Furthermore, it might be necessary to include roll and pitch orientation in the model of the system.

The camera could be used to analyse the change between two images to extract a relative position and angle change. However, before implementing such an algorithm, it should be analysed how fast the system would react. The camera has a long delay when capturing an image, so it needs to be investigated whether the resulting position change would still be valid once processing is complete.

The system should also be tested together with the other ground robots. It would be interesting to see whether cooperation them is possible.

## 8.2 Conclusion

All in all, the objectives stated at the beginning of this thesis have been achieved. The ultrasound sensors were integrated into a single module with the camera, an IMU was added and the communication with the Golang-server has been established. The tests show that the interconnection between the different parts of the system work and that a rough map of an indoor area can be created.



# Bibliography

- [1] Thomas Andersen. *Sparse IR sensor EKF-SLAM for MQTT-SN/Thread connected robot*. Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2022. (document), 4.3, 4.4, 5.5
- [2] AosongElectronics. Dht-22 datasheet, 2024. <https://cdn-shop.adafruit.com/datasheets/DHT22.pdf>, Accessed: [06.05.2024]. 3.1
- [3] armMBED. *Nordic nRF52840-DK*, 2019. <https://os.mbed.com/platforms/Nordic-nRF52840-DK/>, [Accessed: 15.02.2024]. 3.1
- [4] Timothy D. Barfoot. *State estimation for robotics*. Cambridge University Press, 2021. (document), 2.3, 2.5, 2.3.3, 2.6, 2.7, 2.3.4
- [5] Jonas Bjerke. *Position measurement for quadcopter*. Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2021. 4.1
- [6] Jonas Øygard Bjerke. *Position Measurement for Quadcopter*. Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2022. (document), 1, 4, 4.1, 4.1
- [7] John Catsoulis. *Designing Embedded Hardware*. O'Reilly Media, 2 edition, 2009. 2.2
- [8] J. Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. 2006. (document), 2.8, 2.3.4
- [9] ElecFreaks. Hc-sr04 datasheet, 2024. <https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf>, [Accessed: 06.05.2024]. 3.1, 5.3
- [10] FreeRTOS. Freertos, 2024. <https://www.freertos.org/>, [Accessed: 19.05.2024]. 2.1.3
- [11] Garmin. Garmin lidar-lite v4 led datasheet, 2020. <https://www.garmin.com/en-US/p/610275#specs>, [Accessed: 02.06.2024]. 8.1.1
- [12] InvenSense. Icm-20948 datasheet, 2016. <https://invensense.tdk.com/wp-content/uploads/2016/06/DS-000189-ICM-20948-v1.3.pdf>. (document), 2.4, 3.1, 5.5, 5.6.2
- [13] W.A.L Klose. *A new server for the slam robot project*. Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2023. 4, 4.3
- [14] Kristian William Macdonald Gulaker. *Quadcopter for indoor mapping*. Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2022. 4, 4.1, 4.3
- [15] NordicSemiconductor. Introduction - nrf connect sdk 2.6.99 documentation, 2024. [https://developer.nordicsemi.com/nRF\\_Connect\\_SDK/doc/latest/nrf/index.html](https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/index.html), [Accessed: 06.05.2024]. 3.2
- [16] NordicSemiconductor. Nordic semiconductor infocenter - nrf52840, 2024. [https://docs.nordicsemi.com/bundle/ps\\_nrf52840/page/keyfeatures\\_html5.html](https://docs.nordicsemi.com/bundle/ps_nrf52840/page/keyfeatures_html5.html), [Accessed: 30.05.2024]. 3.1
- [17] NordicSemiconductor. nrf52840 dongle, 2024. <https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle>, [Accessed: 06.05.2024]. 3.1

- [18] OpenThread. <https://openthread.io/guides/thread-primer/node-roles-and-types>, [Accessed: 06.05.2024]. 2.4.1
- [19] RaspberryPi. Raspberry pi 3 model b+, 2024. <https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/>, [Accessed: 26.03.2024]. 3.1
- [20] Segger. Embedded studio: The multi-platform ide, 2024. <https://www.segger.com/products/development-tools/embedded-studio/>, [Accessed: 06.05.2024]. 3.2
- [21] SparkFun. Sparkfun icm-20948 arduino library, 2021. [https://github.com/sparkfun/SparkFun\\_ICM-20948\\_ArduinoLibrary/blob/main/DMP.md](https://github.com/sparkfun/SparkFun_ICM-20948_ArduinoLibrary/blob/main/DMP.md), [Accessed: 01.03.2024]. 5.6.2
- [22] UnitedNations. United nations sustainability goals, 2024. <https://sdgs.un.org/goals> [Accessed: 06.05.2024]. 1
- [23] Marcus Steffensen Vormdal. *Embedded system for maze-mapping*. Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2022. (document), 1, 4, 4.2, 4.2.1, 4.2
- [24] Burns Wellings. *Real-time systems and their programming languages*. Addison-Wesley, 4 edition, 2009. 2.1
- [25] Stallings William. *Operating Systems: Internals and Design Principles, Global Edition*. Pearson Education Limited, 9 edition, 2017. 2.1, 2.1.1, 2.1.2, 2.1.4
- [26] Stephen J. Young. *Real Time Languages, Design and Development*. E. Horwood, 1982. 2.1

