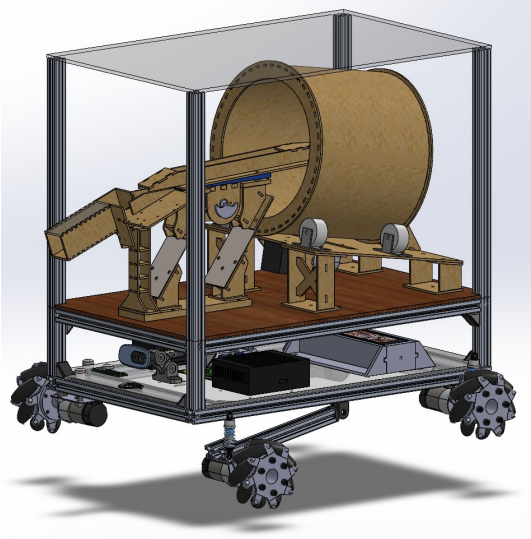
Omnidirectional Process Robot for Swarm Production



ROB6 - Bachelor Project Report
Robots in an Application Context

Group 663

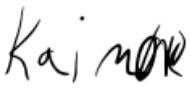
Aalborg University
Robotics
6th Semester

Preface

Aalborg University, June 25, 2022

Andreas Følbæk Gravgaard
<agravg19@student.aau.dk >

Benjamin Oliver Musak Hansen
<bomh19@student.aau.dk>



Kaj Mørk
<kmark19@student.aau.dk>

Kasper Grøntved
<kgrant19@student.aau.dk>

AALBORG UNIVERSITY
STUDENT REPORT

Robotic
Aalborg University
http://www.aau.dk

Title:
Omnidirectional Process Robot for Swarm Production

Theme:
Robots in an Application Context

Project Period:
February 2, 2022 – May 25, 2022

Project Group: 663

Participants:
Andreas Følbæk Gravgaard
Kaj Mørk
Kasper Fuglsang Grøntved
Benjamin Oliver Musak Hansen

Supervisor:
Casper Schou

Page Numbers: 142

Date of Completion:
May 25, 2022

Abstract:

The demand for variable products is forcing manufacturers to be more flexible in production. In this project, current technologies and methods are analyzed and compared with the concept of swarm production. The concept of swarm production is based on process robots, which perform specific manufacturing tasks, and carrier robots, whose task is material transport. Based on a study of current methods for manufacturing and mobile robotics, a design of a process robot platform for use in swarm manufacturing is created. The design includes a generic, cost-effective platform that incorporates a generic design for process robots and a system that uses a Kalman filter to localize robots with ArUco markers. The process robots use IR sensors for obstacle detection and avoidance navigation. To test the design on a real case, a prototype is built from the design. The case on which the prototype is tested is a LEGO *prepackaging* concept, where high variability is required. The prototype has shown that the ArUco markers are a cost-effective solution for localization and navigation that can be used to perform essential functions related to the case, such as transferring a specific number of LEGO bricks to a carrier robot.

Contents

Acronyms

AAU Aalborg University.

AGV automated guided vehicle.

AMCL adaptive Monte-Carlo localizer.

AMR autonomous mobile robot.

API application programming interface.

CP cyber physical.

CPR counts per revolution.

DDS data distribution service.

DOF degrees of freedom.

EOL end-of-life.

FMS flexible manufacturing systems.

FOV field of view.

FPS frames per second.

GPS global positioning system.

I2C inter-integrated circuit.

IMU inertial measurement unit.

IP internet protocol.

IR infrared sensor.

lidar light detection and ranging.

LiPO lithium polymer.

MDF medium density fiberboard.

PE polyethylen.

PP polypropylen.

PWM pulse width modulation.

QoS quality of service.

ROS Robot Operating System.

RPM rounds per minute.

SLAM simultaneous localization and mapping.

SPI serial peripheral interface.

TCP transmission control protocol.

UDP user datagram protocol.

URDF unified robot description format.

UWB ultra-wideband.

XML extensible markup language.

YAML yet another markup language.

Special fonts and markings

- Function names: *function name*
- Transform frames in ROS 2: `frame`
- ROS 2 topics: */ros_topic*
- ROS 2 messages: *ROS 2 message*
- ROS 2 nodes: *nodes_name*
- X distance variable (specific): X
- Y distance variable (specific): Y
- Z distance variable (specific): Z
- Roll distance variable (specific): *roll*
- Pitch distance variable (specific): *pitch*
- Yaw rotation variable (specific): *yaw*

1 Introduction

The paradigm of assembly line production was introduced by Henry Ford in 1913 through the innovation of moving products continuously during their processing. This enabled efficient mass production of products, increasing the production rate while reducing unit costs [1]. Over the years, efficiency and operating costs have been continuously improved, further lowering unit costs and increasing unit production. Today, market needs have changed with the increasing demand for more product innovation to compete globally. This has led to the need for manufacturing techniques to become more adaptable to new and rapid market changes as product life spans become shorter and greater product variety is desired. To accommodate this change, innovative production equipment is therefore needed that can be easily reconfigured to manufacture new products and components [2]. This means that efficiency and flexibility must go hand in hand, which is a challenge with current mass production techniques. To solve this problem, swarm production is proposed as a reinvention of the traditional manufacturing systems with fixed production lines and machines [3].

Swarm production is a concept developed and currently being researched at Aalborg University (AAU). The concept aims to create the necessary flexibility while maintaining the cost efficiency of current production methods. In swarm production, the material flow and processes are made mobile. The mobility of the processes makes it easier to reconfigure the production flow. The current state of the concept consists of two types of units, namely carrier robots and process robots. The carrier robots can transport materials and components and exchange them between the process robots, while the process robots perform tasks such as welding, assembling, and feeding parts to the carrier robots, etc. This new production flow is called "swarm production" because the use of multiple mobile robots in an orchestrated system enables the transport of materials, components and parts, etc. between mobile stations and makes the production flow more flexible and faster to adapt to market changes. The concept of a swarm production system is shown in Figure 1.1.

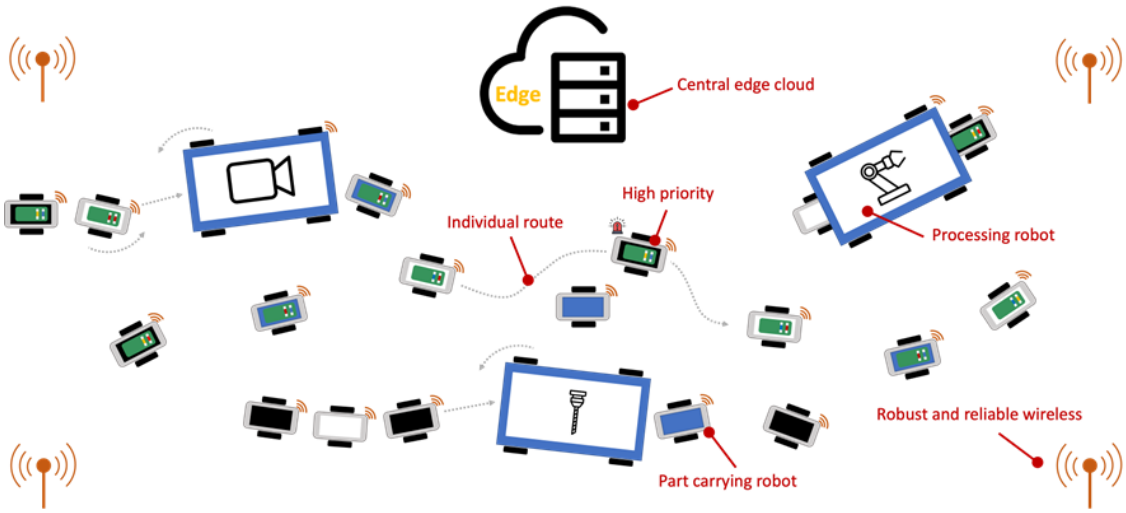


Figure 1.1: Concept of a swarm production system [4].

Swarm production aims to have the same advantages as existing flexible production paradigms, such as matrix production [5] and reconfigurable line production [5], without adopting their disadvantages. Thus, swarm production is a flexible production paradigm that aims at both flexible routing and flexible positioning of workstations. The main advantage is that swarm production can adapt to other topologies, such as line topology and matrix topology according to changes in demand without downtime or human intervention [4]. Since swarm production is still a research topic, there are some challenges that still need to be solved. Some of the biggest challenges are keeping the cost of the mobile robots low by both off-boarding heavy computations to a central computation unit and finding alternative sensors for localization and navigation. By off-boarding heavy computations, another challenge arises, namely fast and robust wireless communication between the robots and the central computation unit. Finally, planning and scheduling solutions must be developed for such a highly flexible system, while maintaining efficiency [6].

As part of the research project at AAU, this report addresses the development of the first prototype of four process robots for use in a swarm production system. This includes the design of the physical structure and software of a process robot and how it interacts with a particular process and carrier robot. In addition, a more cost-effective solution for localization and navigation of the robots will be explored.

Initial Hypothesis

An initial hypothesis is created, accumulating the basic points of the project:

It is possible to develop a process robot platform with alternative cost-effective sensors for localization and navigation, which cooperate with carrier robots in a swarm production system.

---

2 Problem Analysis

In this chapter, the problem described in the introduction is explained in more detail. A further justification of why the problem needs to be solved and why swarm production could be a solution. How swarm production can be implemented is explored using technologies and current solutions. At the end, a hypothesis is made and requirements for a prototype robot are stated.

2.1 Flexibility in Manufacturing

The ongoing implementation of Industry 4.0 is the result of consumers' desire for low prices, a large product variety and short delivery times. Industry 2.0 enabled high volumes and low prices through the invention of mass production. Industry 3.0 introduces high volumes with short delivery times and some product variety through the introduction of digitization and more control over production. However, Industry 3.0 does not offer the product variety that the market needs today. A large variety of products has always been an important factor in maintaining market share. This is illustrated by the example of Ford, whose market share fell from 66% to just 18.9% [7] between the 1920s and 1940s because consumer demand was more diverse than Ford's product line offered. A more recent example is Apple's iPhones. Originally, there was only one iPhone model in one color with fixed specifications, but today there are 80 different variants of the iPhone 13 [8]. This level of variation challenges the current status quo. As a result, Industry 4.0 is now trying to push the boundaries to meet all three desires at once with the introduction of mass customization, as shown in Figure 2.1. However, to achieve Industry 4.0 there is a need for flexibility.

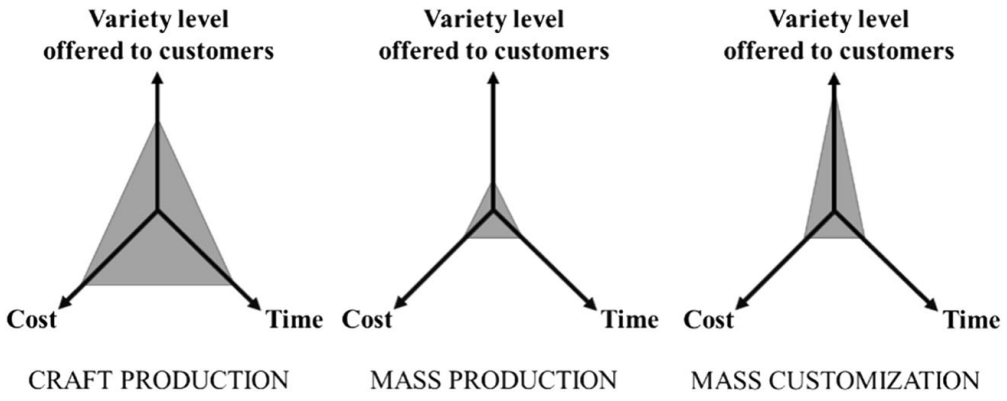


Figure 2.1: Relationship between the level of product variety offered to customers, the time it takes to deliver the product, and the cost of the product through different production types [9].

The required flexibility can be generated by flexible manufacturing systemss (FMS) and/or flexible material flow. FMSs can be divided into two categories: *machine flexibility* and *routing flexibility*. Machine flexibility means that the system can be changed to

produce new types of products and change the sequence of operations performed on a part. Routing flexibility is the ability to use multiple and/or different machines for the same operation on a part and the ability of the system to absorb large-scale changes, such as in volume, capacity, or capability [10, chap.12]. Machine flexibility and routing flexibility are utilized through flexible material flow, i.e., if the machines are able to provide different routes through production, the material flow must also have the ability to change routes to utilize the machine flexibility and routing flexibility.

In terms of manufacturing processes and material flow the number of different parts in production combined with high production volume and changes in production volume is a challenge in mass customization. A common method for handling large volumes of materials/parts is dedicated line production, as shown in Figure 2.2. In dedicated line production, a product is mass produced with few or no changes. This is ideal for a product that needs to be mass produced, but becomes problematic when the market changes over time and a redesign of the product is required. Conveyors are very common in dedicated line production, but this is only a viable solution up to a certain level of part variability [11].

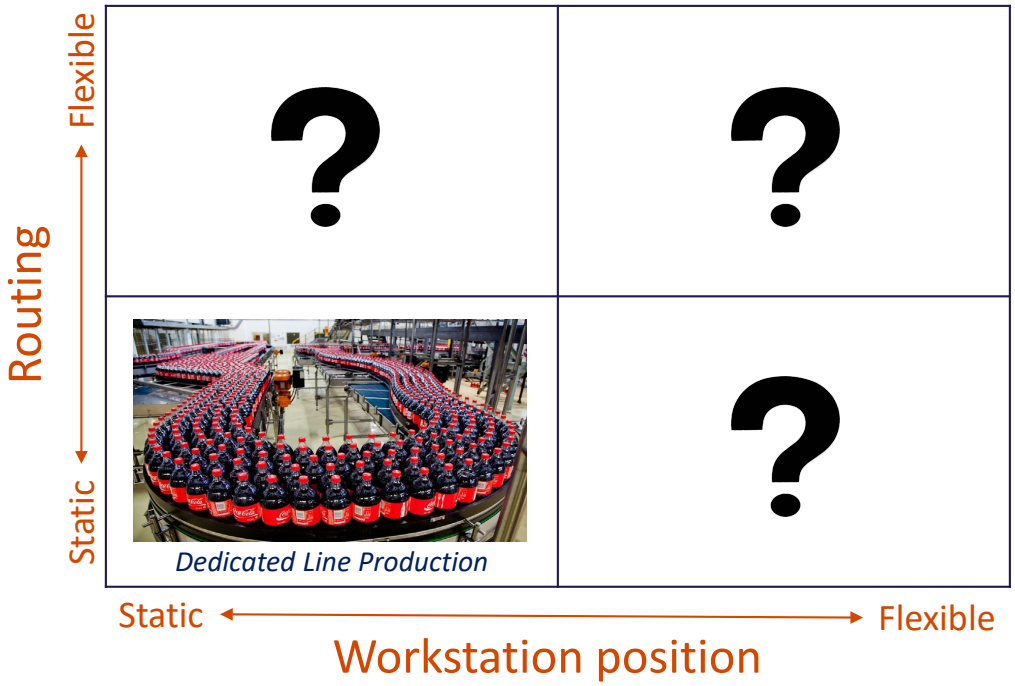


Figure 2.2: Dedicated Routing flexibility vs. workstation position flexibility. Adapted from [6].

Factories have been using mobile robots for some time in production areas where the material flow has low throughput but high variability. One example is LEGO, which has used mobile robots in production since the 1980s [12]. LEGO uses mobile robots to retrieve different LEGO bricks from different molding presses, but uses conveyors in other production areas [13]. Since mass customization is about high volumes while maintaining flexibility, current methods that handle high product variability are being challenged. Another problem with using conveyors and mobile robots in the current paradigm is that they are relatively fixed to one operation. This is also the reason for the problem of changeover times, which is also a bottleneck in mass production. A more

inflexible material flow and manufacturing system means longer changeover times. Decentralization is required for minimum changeover times, where decentralization means that the production system can be changed in real time and that the system is also able to adapt to machine learning, for example [6]. If the system is decentralized, the production can also be non-linear. Tracking all inventory and materials is also an important aspect to ensure decentralization and lean production. The system must be scalable so that production volume can be increased or decreased as demand fluctuates. To meet the challenges of mass customization, a system is needed that has the following characteristics:

- Able to change material routing
- Able to have high volumes in material flow
- Able to have high product variability output
- Non-linear and fully decentralized
- Scalable

2.2 Existing Flexible Paradigms

There are already other paradigms aimed at solving mass customization challenges, and these can be seen in the second and fourth quadrants of Figure 2.3.

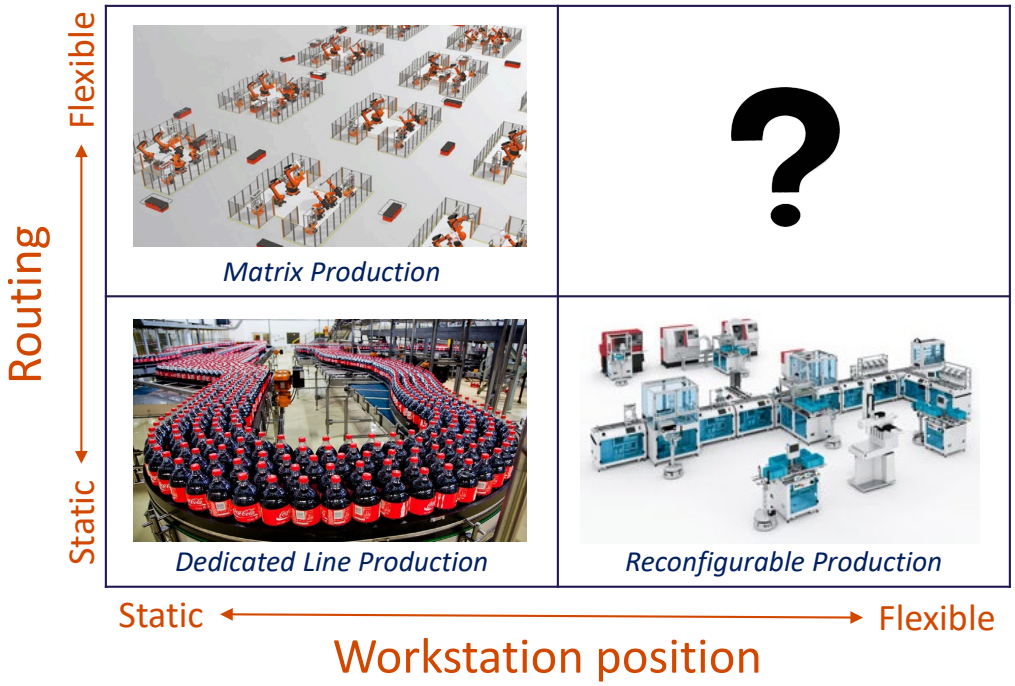


Figure 2.3: Routing flexibility vs. workstation position flexibility. Adapted from [6].

Matrix production (second quadrant in Figure 2.3) aims to solve the static routing problem by dividing the dedicated line processes into modules. The modules are arranged

in a grid where automated guided vehicles (AGV) then transport workpieces and tools to each module, as shown in the Kuka matrix production example in Figure 2.5. Each module can then perform tasks on the workpieces before they are transported to the next module.

Matrix production thus enables a flexible and continuous flow of products [14]. The redundant modules also allow the production system to handle breakdowns by simply passing the AGVs to a working module [15]. By making modules with articulated robots and having the AGVs provide different tools, it is possible to introduce an additional level of customization into the production system [16].

Reconfigurable production lines (fourth quadrant in Figure 2.3) aim to solve the static workstation position problem by using standardized modular and movable conveyor cells to which processes can then be connected. Some reconfigurable production lines, such as Festo's cyber physical (CP) Factory (see Figure 2.4), can also be reconfigured at the routing level by having cells with different conveyor configurations and even making the conveyors configurable [17]. However, all of this must be done manually and products must therefore be produced in batches [5].

Figure 2.4: Festo CP Factory at AAU Smart Lab [18].

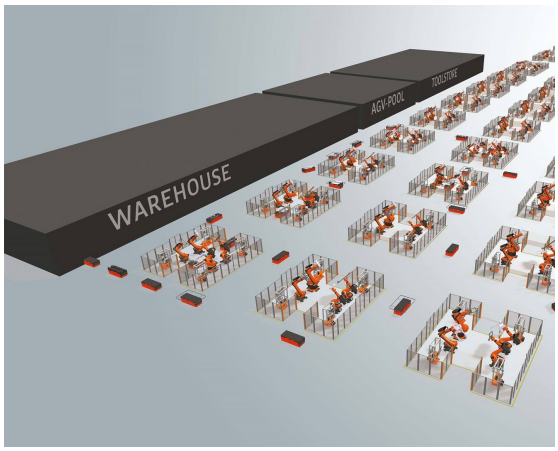


Figure 2.5: Kuka's example of an Industry 4.0 matrix production [16].

2.2.1 The Effects of Using Mobile Robots in Production

Matrix production introduces the concept of using mobile robots in a production system, which is not the case with either dedicated line production or reconfigurable production.

A study evaluates this concept of using mobile robots for material handling and concludes that using multiple AMRs increases flexibility to deal with rapid response times and breakdowns. The study also concludes that using multiple AMRs makes it easier to handle urgent orders, as one of the AMRs can then be selected to perform the prioritized order [19].

A disadvantage of reconfigurable production is that human intervention is required when it needs to be reconfigured. A study examines the concept of a reconfigurable production line consisting of processes made mobile and compares it to a traditional

production line with conveyors and stationary processes. A simulation showed that the line with mobile processes achieved higher utilization, availability and production volume. The study also found that the use of mobile robots resulted in higher reconfigurability, reduced duration of downtime, reduced commissioning time, higher reliability and flexibility [20].

By combining these two concepts, the disadvantages of matrix production and reconfigurable production can be eliminated. The result is the swarm production paradigm was born.

2.3 Swarm Production - A New Paradigm

The concept of swarm production is a research topic currently being researched at AAU. The description of swarm production is inspired by swarm robotics. Swarm robotics is a study on how larger groups of robots can be coordinated by some defined local rules to achieve a desired collective behavior [21]. The behavior is inspired by the insect world and is a combination of swarm intelligence and robotics. Swarm robotics allows multiple robots to work individually toward the same goal, making the path from start to finish much more flexible. To make the process reliable and robust, some rules must be followed:

- Autonomous control for each robot
- It is scalable and can work in great numbers
- Decentralized coordination
- Flexibility

Robots must be able to operate autonomously by interacting with the other robots and the environment. They need to be able to operate in larger groups that allow scalability by making the removal and addition of new robots to the swarm fast and flexible [21]. More robots doing the same task or serving as backups in case the performing robots cannot complete their tasks ensures that a constant flow of robots can work and help each other complete tasks without downtime. This is something that manufacturing can greatly benefit from Industry 4.0. Robots need to be able to make decisions on their own by communicating with other robots and their environment. And, more importantly, they must be flexible and able to create modular solutions for different tasks.

The proposed solution for swarm production, shown in the first quadrant in Figure 2.6, provides the most flexible routing and positioning of workstations. Swarm Production seeks to implement both a flexible material flow and a flexible production topology. It allows for customizable movement of materials and workstations into and out of production without the need to reconfigure a line through manual labor. Another benefit is the backup machines described earlier, which can take over a specific task when a machine needs maintenance or breaks down. Such flexible production also means that

it is possible to switch between different configurations. The swarm production robots can replicate the line topology and the matrix topology when such a system is needed.

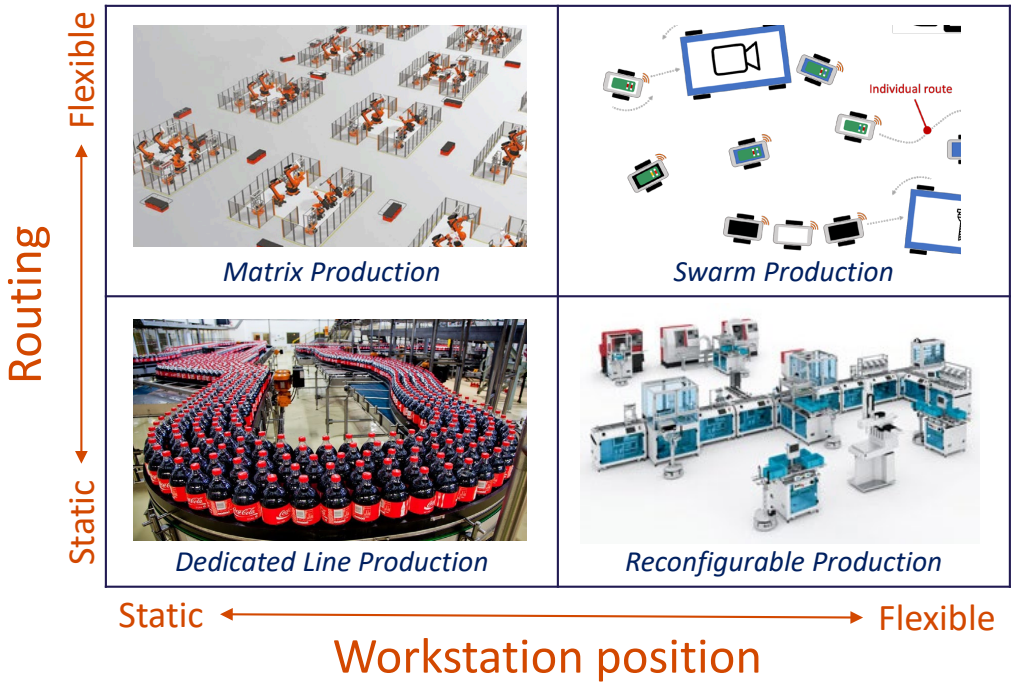


Figure 2.6: Routing flexibility vs. workstation position flexibility. Adapted from [6].

As the production line becomes fully mobile, reconfigurability also allows the system to find a more optimal positioning of workstations and material paths over time. Bottlenecks in the material flow can be eliminated automatically without having to interrupt production to disassemble, move and reassemble conveyors. Figure 2.7 shows a conceptual interpretation to illustrate the difference in operation. Each configuration has advantages and disadvantages, so the decision to use a particular configuration depends on the situation. An ongoing study at AAU, is investigating how the swarm production topology can be advantageously implemented in a production line.

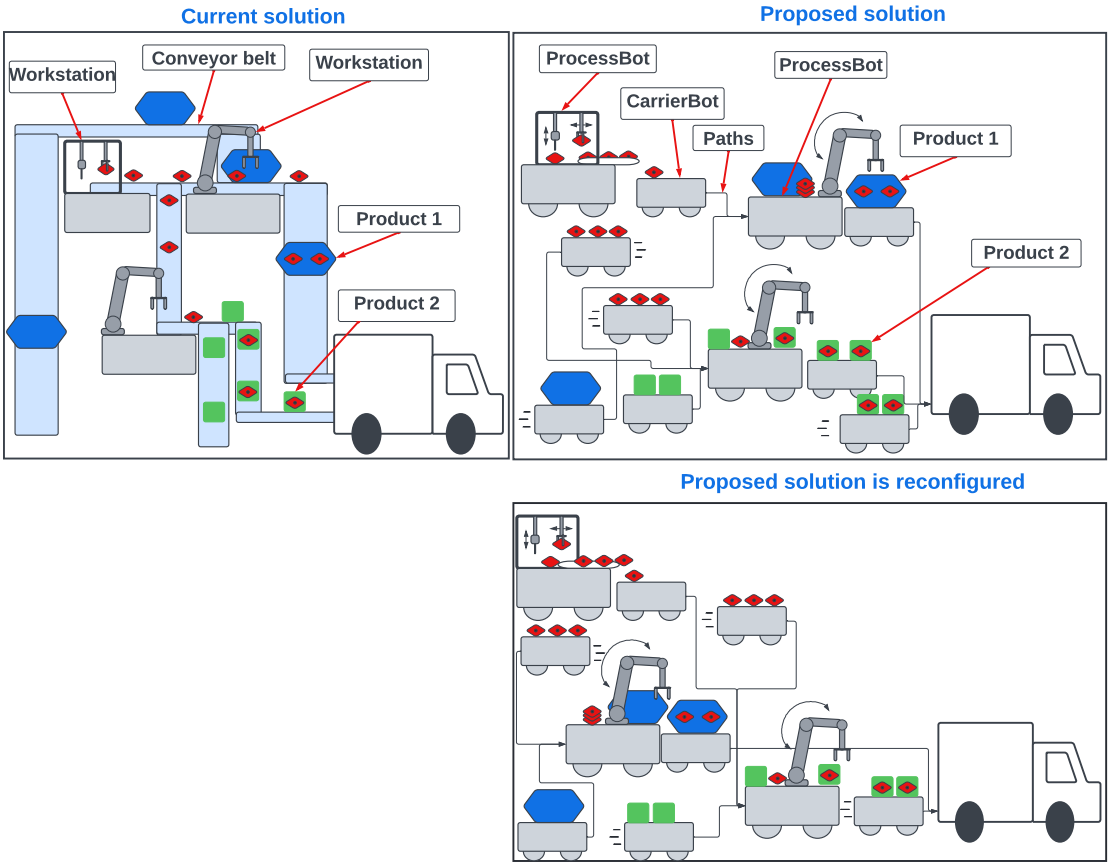


Figure 2.7: Current solution vs. proposed solution. The layout is viewed from the top down, although the images are drawn from the side.

Summary of swarm production

A major challenge at present is to identify the necessary system functions and consider how they can be adapted to existing methods, standards and frameworks that are already widely used in the industry. Another challenge is to develop a control scheme that requires coordinated control and holistic planning. The goal is collective swarm intelligence based on centralized control but with decentralized capabilities for some functions. Industry and process types also need to be explored, to determine a suitable design for the different types of mobile entities and the number of each entity. So far, two types of mobile entities have been determined, the carrier robot platform and the process robot platform. The process robot platforms are an important part of swarm production because they directly impact the control scheme, which functions can be decentralized, and allow real-world swarm production scenarios to be tested.

Due to the nature of swarm production, the overall design of these entities would benefit from being cheap, simple and redundant, as opposed to complex and expensive. These are all topics that are currently being researched, with the exception of the design of the process robot platform [6]. Therefore, this project will explore the design of the process robot and the challenges associated with it.

2.4 Initial Requirements - Interview With Casper Schou

An interview was arranged with Casper Schou [1] to get a better understanding of the basic requirements for the process robot platform. Since the process robot must be built from scratch, a budget must be estimated for the construction costs (not including the process itself). Casper indicated that the budget for a single process robot would be 8000 DKK. In terms of physical aspects, Casper stated that the process robot must not weigh more than 25 kg, that it must be able to carry and move around with a process, and that it must initially have an area of less than 1×1 m. Casper also mentioned that the factory floor at the AAU Smart Lab, which is where the robots will be used, is uneven, with differences of up to 10 mm. To ensure that the wheels are in contact with the floor at all times, the process robot must compensate for this unevenness.

Regarding the accuracy of the robot, Casper mentioned that it should be most accurate when handling the docking sequence between the carrier robot and the process robot. This is due to the fact that the containers might change in size and be quite small, and accuracy of 1 mm is therefore the requirement. For all other scenarios, the accuracy should be sufficient to navigate from a to b without collisions. According to the paper [23], the least achievable accuracy for a mobile robot should be 200 mm, which will therefore be the requirement for all other scenarios. All requirements from the interview are summarized in Table 2.1.

Budget of a single process robot:	8000 DKK
Maximum weight:	25 kg
Maximum footprint size:	1×1 m
Docking position accuracy:	1 mm
Position accuracy for other scenarios:	200 mm
All wheels of the robot must be in contact with the floor at all times thus compensate for a floor unevenness of up to:	10 mm

Table 2.1: Initial requirements from the interview with Casper Schou.

2.4.1 Introduction to Case - LEGO Brick Feeder

As an example of the use of the process robot platform in a specific case, Casper Schou mentioned that at AAU the first case selected for swarm implementation is LEGO's prepackaging line. In a LEGO factory, the process from plastic to product can be divided as follows [12]:

Production: Here plastic pellets are melted and used in injection molding machines to produce LEGO bricks, which are then transported to a storage facility.

Packaging: When a new LEGO set is to be produced, the required bricks are delivered from the storage facility and then placed in counting machines. Then the prepackaging

[1] Casper Schou is an Ass. Prof. at the Department of Materials and Production Robotics and automation group at AAU. His background consists of an MSc. in Manufacturing Technology, a PhD. in reconfigurable collaborative robotics, and he is of now doing research in swarm production [22].

step begins, where the counting machines output the exact number of LEGO bricks into plastic bags, which are finally packed into boxes.

Shipment: After packaging, the LEGO boxes are shipped to the customers.

LEGO's prepackaging system consists of a fixed series of counting machines, each of which counts and outputs the individual bricks needed for a single bag. A LEGO set may contain several individual bags of different LEGO bricks, so hundreds of counting machines are dedicated to packaging a single LEGO set. Each time a new variant is to be produced, LEGO's packaging line must be changed over. In addition, given the increasing need for customization (as mentioned in Section 2.1), LEGO's packaging line must adapt to these changes. For example, LEGO has introduced the *Pick a Brick* and *Build a Minifigure* concepts [24], which allow customers to purchase individually selected bricks or design their own minifigures. These are two new concepts that the old packaging method probably can not keep up with.

By implementing AAU's LEGO prepackaging case, a proof of concept can be created that demonstrates the advantages of swarm production by mounting the counting machines on the process robots and letting the smaller carrier robots do the collecting. In this way, several different LEGO sets can use the same counting machines, reducing the number of counting machines needed and increasing material flow, which in turn increases order flexibility. It also facilitates tasks such as restocking the counting machines and transporting the LEGO sets to the next packaging step. An illustration of the case is shown in Figure 2.8.

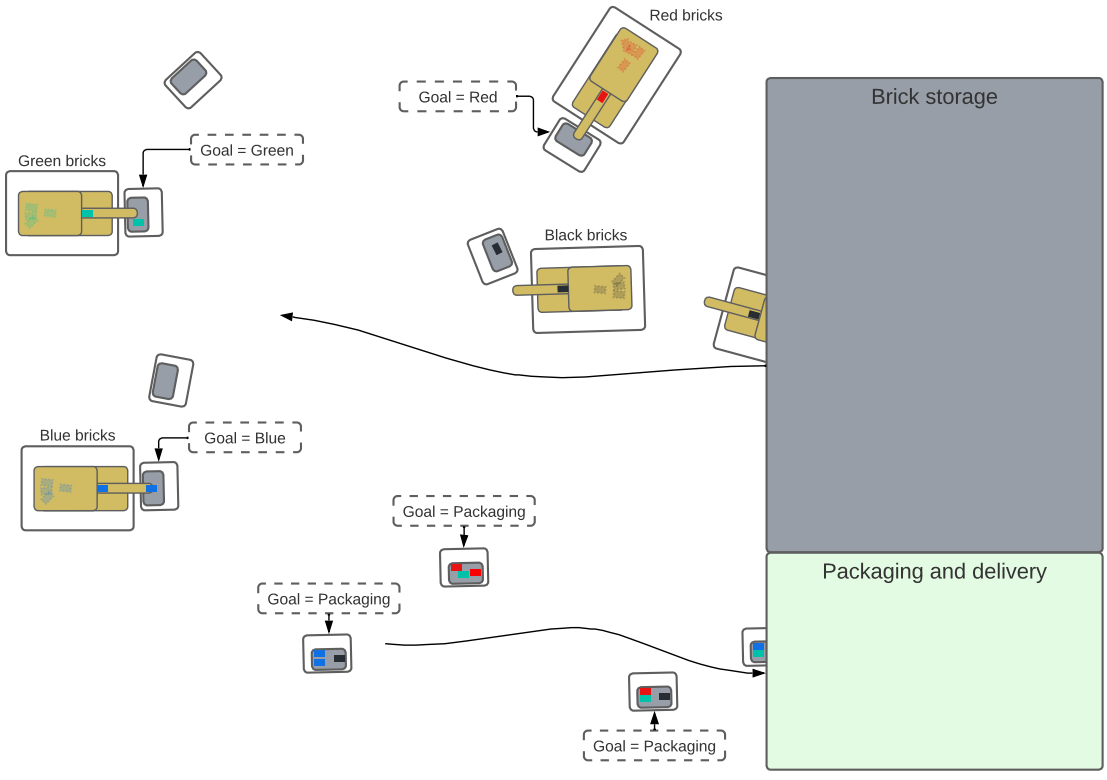


Figure 2.8: Illustration of AAU's LEGO prepackaging case. The process robots are supplied with bricks from the storage facility and feed them into containers on the carrier robots.

AAU's LEGO prepackaging case is complex and is therefore delimited for this project. The delimited case consists of a single process robot with a counting machine mounted on it, which then dispenses LEGO bricks onto a carrier robot, as shown in Figure 2.9.

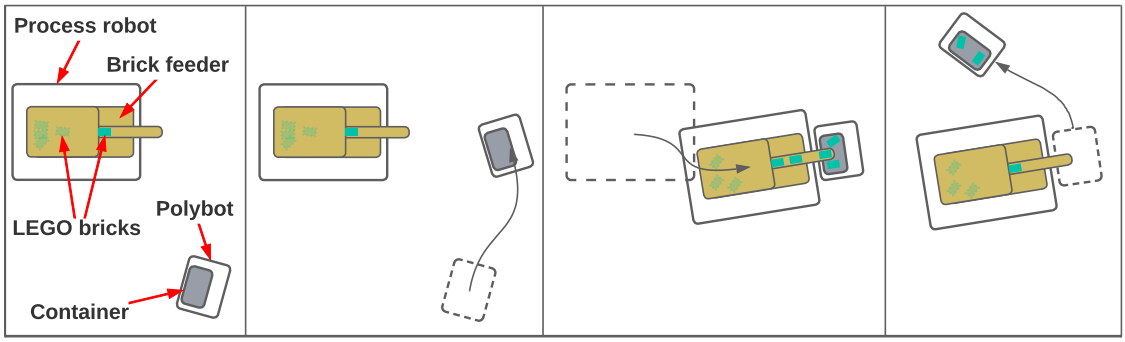


Figure 2.9: Visualization of the delimited case.

The counting machine in this case is a LEGO brick feeder that was built by a 3rd semester project group studying *Machine and Production* at AAU [25]. The LEGO brick feeder is shown in Figure 2.10.

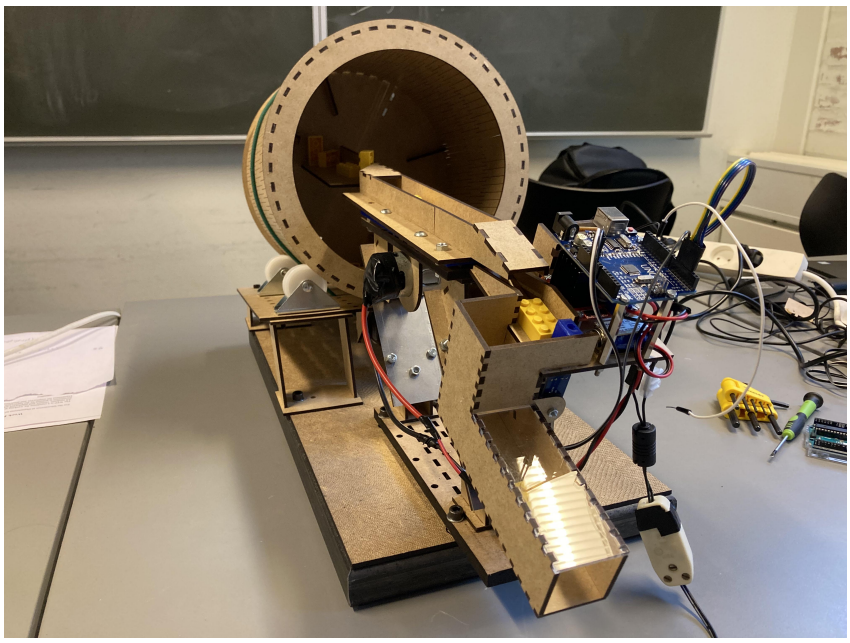


Figure 2.10: LEGO brick feeder [25].

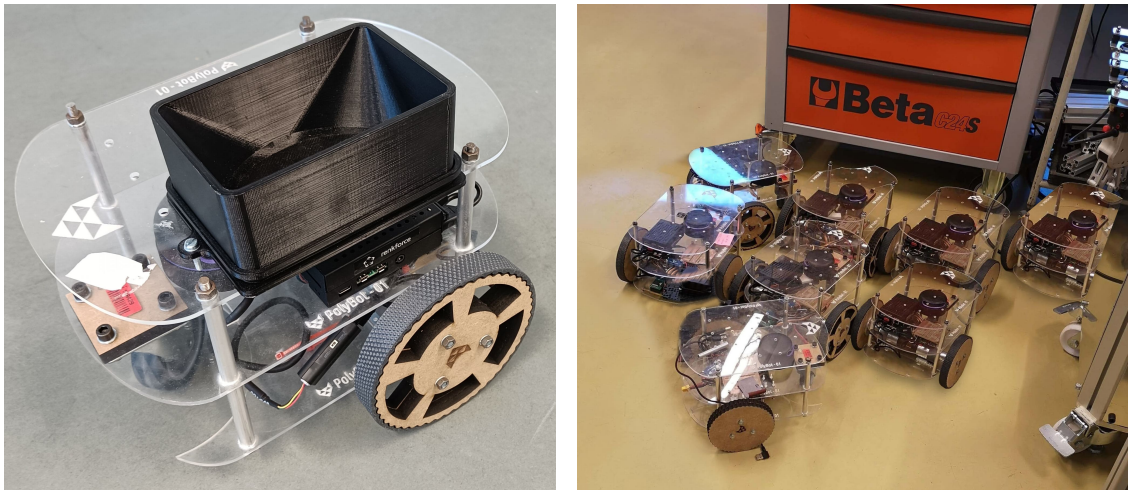
The purpose of the feeder is to transfer 2×4 LEGO bricks from a rotating drum into a small container on the carrier robots. Therefore, the process robot must be tall enough to allow the LEGO brick feeder to reach the height of this container. The process robot and the carrier robot must also be able to navigate to each other so that the LEGO brick feeder on the process robot can feed LEGO bricks into the container on the carrier robot.

The purpose of the case is to prove that the process robot platform can be used to:

- Show that the process robot platform is suitable to mount a given process on.
- Have a process robot navigate in a complex environment.

2.5 Existing Carrier Robot - Interview With Polybot

AAU has purchased eight Polybot V1.2 mobile robots (see Figure 2.11a) from the company Polybot, which can be used for projects or research at AAU (see Figure 2.11b). Polybot is a startup company that aims to develop lightweight and versatile mobile robots. In terms of swarm production research, these eight Polybots will be used as carrier robots in physical use cases such as the one mentioned earlier. Polybot has succeeded in developing a cost-effective (when excluding the light detection and ranging (lidar)) mobile robot that is compatible with Robot Operating System (ROS). It would therefore be useful to better understand what experience they have gained to facilitate the development of the process robots. Therefore, an interview was arranged with laboratory engineer at AAU and co-owner of Polybot, Martin Bieber Jensen [26]:



(a) Polybot V1.2 with a mounted container.

(b) Eight Polybots purchased by AAU.

Figure 2.11: Polybots V1.2.

The Polybot V1.2 is a differential wheeled mobile robot with two Pololu 37D geared motors, both with dual-channel Hall-effect encoders. Each motor is controlled by a Pololu VNH5019 motor driver and these are controlled by a Teensy 4.1 microcontroller. The Teensy 4.1 microcontroller also receives data from a BNO055 9-degrees of freedom (DOF) inertial measurement unit (IMU) and the two encoders. This data is then sent to a Raspberry pi 4 via serial on the appropriate ROS topics. The entire robot is powered by an 11.1 V 5000 mA lithium polymer (LiPO) battery, but the Raspberry pi 4 requires 5 V, so a step-down converter is used to convert the 11.1 V to 5 V. The Polybot V1.2 is also equipped with a Slamtec RPLIDAR A3 360° laser scanner for navigation and localization. To avoid the risk of undercharging the battery, a LiPO battery voltage tester is attached to the balance connectors to alert the user when the battery needs to be changed.

Brick Container
The Polybot has been modified to fit into the LEGO prepackaging case, as it has a container on its top for storing the LEGO bricks. The container has a height of 65 mm, an inner length of 83.4 mm, and an inner width of 131.4 mm. In order for the LEGO brick feeder to feed bricks into the container, the center of the tip of the chute must be placed with an accuracy of -41.7 mm to 10 mm in the x-axis and -49.7 mm to 49.7 mm in the y-axis relative to the center of the container. In addition, the height of the chute must be greater than 252 mm, as this is the height of the Polybot with the container mounted on top. This results in the following case-specific requirements, as shown in Table 2.2.

Docking position accuracy:	-49.7 mm to 49.7 mm in the X-axis and -41.7 mm to 10 mm in the y-axis.
Brick feeder chute height:	Above 252 mm

Table 2.2: Case specific requirements.

Linorobot - Repository:
Polybot's software and firmware is based on the popular and open-source linorobot repository [27]. The linorobot repository is a suite of open.source ROS compatible robots and aims to make it easier for students, developers and researchers to create their own mobile robots. Linorobot supports a variety of existing laser sensors, IMUs, microcontrollers and motor drivers. Linorobot also supports 2WD, 4WD, and Mecanum drive. Since linorobot is open-source, users can customize the software and hardware. Currently, linorobot is no longer under development, but the new linorobot2 is still being maintained (a ROS 2 port of the original linorobot repository), which also uses the successor to ROS-serial, namely micro-ROS to for communication between the microcontroller and ROS 2.

2.5.1 Summary of Interview With Polybot

The interview with Polybot was informative and there were some important aspects to consider. First of all, using the linorobot repository can drastically reduce the overall development time. Also, it would be wise to include some form of battery protection to prevent battery undercharging. For the battery, Polybot prefers to use LiPO batteries. Also, a step-down converter should be included if any of the components rely on a lower voltage than what the battery provides. For the motors, Polybot recommends the Pololu brand, which they have used for all their versions of the Polybot.

2.6 Current Solutions

The focus of this project is on the design of the process robot platform. As mentioned earlier, the process robots will collaborate with the carrier robots and therefore must operate autonomously in a non-static environment inside the factory. The basic functions of an autonomous mobile robot (AMR) are obstacle detection, localization and

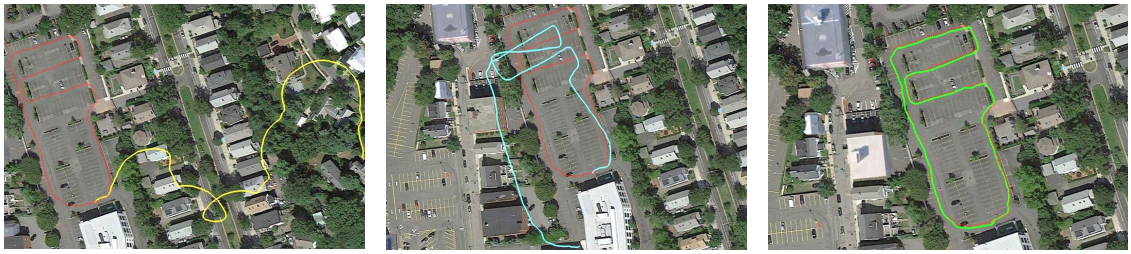
navigation. In addition, the process robot must be able to dock with a carrier robot.

Therefore, this section discusses the current solutions for navigation, obstacle detection, localization, and docking, and then compares the advantages and disadvantages of the solutions. In addition, some existing AMRs are examined to gain a better understanding of the design decisions made. A limiting factor in the choice of method and equipment required is cost. The cost of the equipment is proportional to the number of robots.

2.6.1 Localization Solutions

One of the most fundamental problems in mobile robotics with autonomous capabilities is localization. It is usually divided into two groups: position tracking and global self-localization. Position tracking assumes that the initial position is known and then compensates for any dead reckoning errors, while global self-localization considers the problem without prior position information [28]. There are several solutions to these issues using different sensors and methods. In a survey paper by Durrant-whtye and Bailey [29], it is shown that most simultaneous localization and mapping (SLAM) methods were developed for static environments and use point clouds with particle filters and Kalman filters. Other solutions include inside-out tracking systems, such as visual SLAM [30], or outside-in, such as ArUco markers or optical tracking systems [30], [31].

Robotic localization solutions typically include an extended Kalman filter that fuses data from multiple sensors to produce a position and rotation estimate. Sensor data can come from wheel encoders, IMU, global positioning system (GPS), etc. Each type of sensor data has its own noise characteristics, but when fused, a more reliable and accurate estimate can be obtained. The Kalman filter is able to track the robot from an initially known position. The risk of using Kalman filters is that the uncertainties may become too large and thus the distribution is not unimodal [32]. If the initial position estimate is incorrect, the Kalman filter will not be able to correct for this error. The result of the Kalman filter is shown in Figure 2.12.



(a) Output of the extended Kalman filter (yellow) when fusing only the raw wheel odometry data. The red line is the actual path of the robot [33].

(b) Output of the extended Kalman filter (cyan) when fusing data from wheel odometry and a single IMU. The red line is the actual path of the robot [33].

(c) Output of the extended Kalman filter (green) when fusing data from wheel odometry, two IMUs, and two GPS units. The red line is the actual path of the robot [33].

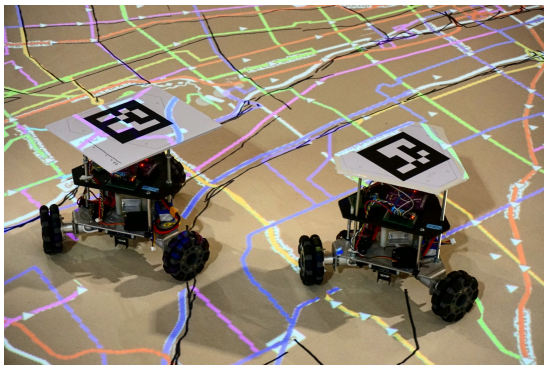
Figure 2.12: Comparison between the outputs of the extended Kalman filter when fusing data from different sources [33].

One method proposed in a paper [28] is the idea of multiple robots use positional tracking with dead reckoning: wheel encoders, IMUs, etc., which help the robot local-

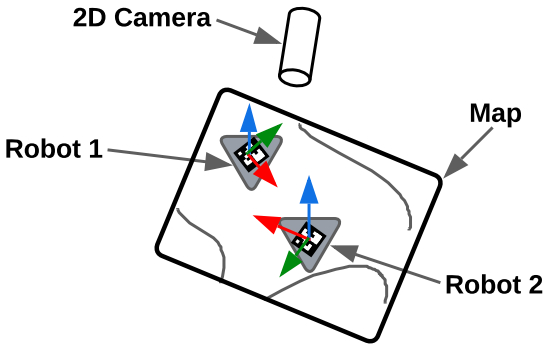
ize itself. Each robot can also localize the other and help reduce inaccuracies by synchronizing localization data, resulting in faster and more robust localization for both parties. This system uses lidar and RGB cameras for robot recognition, a point cloud system, wheel encoders, and distance sensors for navigation and localization. Since the environment is considered static where the robot navigates, this kind of system works well, but might struggle in a busy environment with sparse features to track.

In another paper [34], a visual solution is proposed using color coding for robots tracking. However, as explained in this paper [35], this requires complicated color and camera calibration procedures and is easily compromised by changes in lighting or smooth surfaces that are susceptible to specular highlights. For this reason, a paper [35] presents the so-called "ArUcoRSV", an artificial marker robot localization system developed for soccer robots. The advantages of such a system are that it can remove the constraints imposed by light, perform fast and reliable camera calibrations, refine pose estimation, and significantly improve detection rate with high position accuracy. It is also robust, easily expandable, and can be used for low-cost robots. Disadvantages include visibility of the ArUco markers, complex computations, time required to calibrate the camera, and the need for specific markers with a uniform size for all markers.

Another example of using ArUco markers for localization is a collaboration between two robots drawing a map on a 2-dimensional plane [36]. They achieved an accuracy of about 3 mm when the camera was 4 m away from the markers, with the camera having a resolution of 960×720 pixels while running on a Raspberry pi 3. The size of the markers is unknown. To obtain more accurate measurements, the image is perspective transformed to compensate for any angular errors between the scene and the camera. This method has proven useful in robot control and localization [36], [37]. The concept is shown in Figure 2.13.



(a) Robots with ArUco markers attached, driving on a map [36].



(b) Illustration of the concept.

Figure 2.13: Example of using ArUco markers for localization.

The use of this method to localize the robots scales with the area in which the robots need to be tracked and the accuracy attributed to the camera resolution. Compared to other sensors such as lidars, which scale proportionally with the number of robots, this can be a more cost-effective solution for tracking many robots.

A paper [38] reviews an alternative technology that shows great potential for robot localization, which is called ultra-wideband (UWB). UWB localization systems sends

periodical ultrashort pulses ($<$1 ns) on a wide spectrum ($>$500 MHz), which shows to have stronger anti-interference abilities than Bluetooth and greater accuracy (centimeter) than Wi-Fi. UWB have been used in the military field for a long time, but not in the civilian market. Therefore, is UWB still a topic that is being researched, but companies such as MOTOROLA and Intel have also begun to develop their own UWB technology.

adaptive Monte-Carlo localizer (AMCL) [39] is a widely used localization algorithm that is being used especially with lidars. This localization method is also integrated in ROS and well documented. AMCL is a probabilistic localization system for a robot moving in 2D. It implements the adaptive Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map.

Summary of Localization Solutions

If ArUco markers are attached to each robot and overhead cameras are used, there is a risk of losing visibility. Since AMCL must match sensor readings to a position in a known map, this could theoretically become a problem in a swarm production system because the environment changes frequently with potentially many robots. In addition, lidars, which are commonly used in AMCL, are expensive compared to other solutions, especially since the solution must be scalable. Dead reckoning by itself is not acceptable due to cumulative errors, but a Kalman filter fusing dead reckoning of wheel encoders and IMU with ArUco marker data is considered a viable option for the localization design. If the Kalman filter state estimate becomes inaccurate and unimodal, the pose estimated by the ArUco markers could be used to adjust the wheel odometry data.

2.6.2 Navigation Solutions

Another fundamental problem in mobile robotics with autonomous capabilities is navigation. Robot navigation can be defined as a combination of localization, path planning and map generation/interpretation. Thus, localization depends on the performance and accuracy of these three capabilities. The navigation system should be able to create a collision-free path from one position to another position and follow that path. In swarm production, multiple robots need to plan paths in a confined space at the same time, which poses some problems. For example, if the robots do not know each other's paths, deadlocks can occur because the robots do not know if and when their paths cross. There is also the problem of priority, i.e., which robot goes first. Also, detecting and distinguishing between objects or other robots. Navigating multiple robots is outside the scope of this report, as other research groups at AAU are addressing these issues. However, this project is about enabling the implementation of their solution on the process robot platform. To solve this, ROS and the ROS navigation stack can be implemented.

The ROS navigation stack (or Nav2 for ROS 2) provides all the necessary packages a robot needs to navigate safely in an environment.

Costmap-2D:
This package provides the functionality to map an environment. The map is in 2D

(3D is also possible) and consists of cells, where each cell has a value indicating the possibility of whether the cell is occupied by an obstacle or not. The map is created by subscribing to the sensors of the robots, which usually consist of a laser scanner or an RGB-D camera and odometry. The package is also able to create a costmap for navigation from infrared sensor (IR) and sonar data, which allows the use of cost-effective sensors.

Global-planner:
This package provides the functionality to plan the optimal path from the robot's current position to the goal position given a global costmap. The package also provides a number of different path finding algorithms, such as A* and Djikstra, and even the possibility to implement custom algorithms.

Local-Planner:
This package helps the robot navigate a section of the global path by using the odometry and sensor measurements from the robot, which continuously update the local costmap. The local-planner can therefore avoid dynamic obstacles that are not recorded on the global costmap used by the global-planner, as seen on Figure 2.14.

AMCL:
This package, as mentioned in Section 2.6.1, is also provided by the ROS naviation stack and uses the previously mentioned costmaps to localize the robot.

Gmapping:
This package contains the algorithm Fast SLAM, which creates the 2D costmap from laser scan data and odometry data [40].

Robot_localization:
This package contains an implimentation of an extended Kalman filter [33]. With this, the positional and orientational data from the ArUco markers, IMU data, and wheel encoder data can be fused and used for localization.

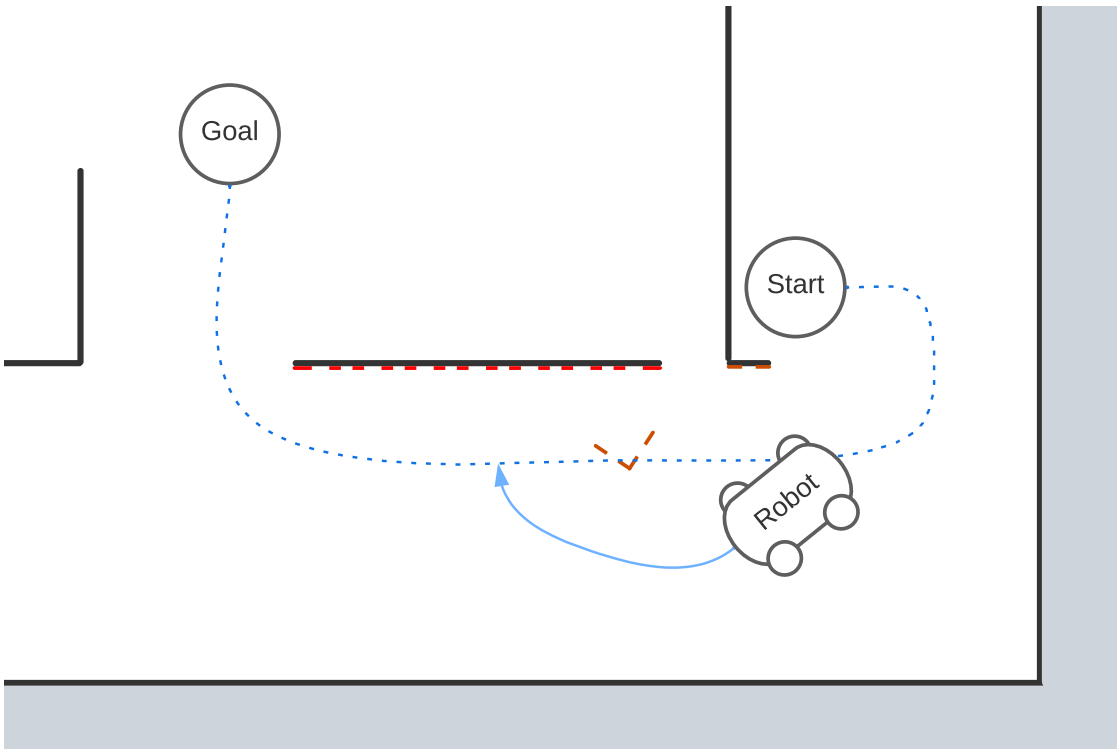


Figure 2.14: Illustration of the global and local planner and costmap in action. The blue dotted line is the global path, the red dotted lines show the laser scanner continuously updating the local costmap, and the blue arrow indicates the local path trying to avoid the dynamic obstacle.

ROS itself is highly configurable, so that the ROS navigation stack is in fact not a single solution, but a variety of different solutions, all of which depend on how the user configures them. For example, the user can decide which sensors and path planning algorithms to use and which sensor data to rely on the most. In addition, all messages in the ROS network can be freely accessed, allowing multiple robots to communicate with each other.

The MiR100 robot is an example of a commercial robot that uses ROS and the ROS navigation stack [41] as its backbone. For the MiR100 robot, users can input the initial position of the robot, the goal position and a prerecorded map, which is then used by the global planner. For the local planner, the MiR100 robot provides sensor data from two SICK-S300 safety laser scanners, two 3D cameras, and 4 ultrasonic sensors, which are also used to localize the robot. For further localization, the MR100 robot also provides sensor data from an IMU and encoders, as shown in Figure 2.15.

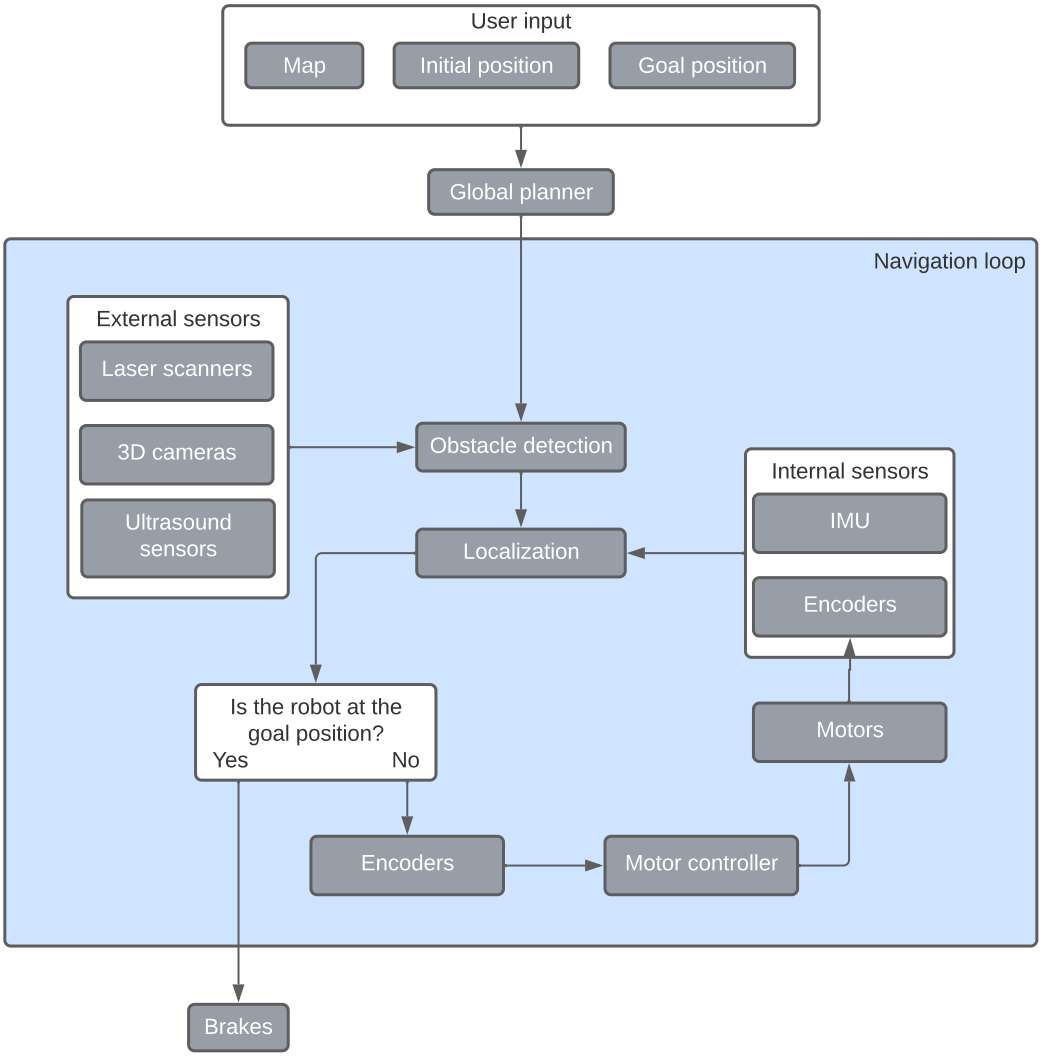


Figure 2.15: Flowchart of the MiR100 navigation system [42]. After the user provides the required input, the robot generates a path and executes the steps of the navigation loop. The loop continues until it reaches the goal position, where it applies the brakes and stops.

2.6.3 Obstacle Detection Solutions

Obstacle detection is critical for mobile robots with autonomous capabilities, as this information can be used to map areas that the robot must traverse. This map can then be used by numerous pathfinding algorithms and even assist in localizing the robot. A paper [43] investigates a range of different types of sensors, which are now examined.

Exteroceptive Sensors

These sensors are used to provide a representation of the environment, usually in terms of visible features or distance to obstacles. This information can help vehicles recognize their surroundings and interact with the real world. Exteroceptive sensors can

include microwave radars, cameras, laser scanners and sonars. Distance sensors such as infrared sensors, tactile sensors, laser scanners, cameras (visible and infrared), and depth cameras can be used to measure distance to obstacles.

Passive and Active Sensors

The above sensors can be further divided into passive and active sensors. Passive sensors do not emit energy and rely solely on external energy sources, such as the sun. Examples of passive sensors include thermal infrared sensors and stereo cameras.

Active sensors have their own energy source and can therefore operate in more versatile conditions than passive sensors, but may have interference issues when used on multiple robots. Examples of active sensors include lidar, sonar, radar, and infrared distance sensors.

Tracking Systems

The paper also mentions the continuous advances in machine vision, which are slowly becoming more important for mobile robots. This is due to the versatility of machine vision, as it can be used to detect objects and people, and even extract three-dimensional data and predict the motion of detected objects and people.

Summary of Obstacle Detection Solutions

Sensors such as 3D cameras or lidars are expensive compared to sonars or IR sensors. Tracking systems are expensive in terms of computing power and cost. For a more robust solution, an active sensor could be the solution. Therefore, IR sensors or sonars are considered as a cost-effective and robust solution. In terms of integration, IR and sonar data can be used in ROS to create costmaps for navigation.

2.6.4 Docking Solutions

Mobile robots can perform a variety of tasks, such as transporting materials or assembling parts. The specific tasks usually require greater accuracy in positioning and a fixed orientation, such as when docking a boat in a harbor. To solve this problem, a docking procedure must be implemented to allow the process robot to dock with the carrier robot, which can be divided into three subprocesses, as shown in Figure 2.16. The requirement for the process robot platform is a position accuracy of 1 mm.

Move close to object: The first subprocess focuses on the approach of the robot to the object and can be achieved by using standard navigation solutions.

Approximate docking: The second subprocess starts with the docking procedure, but with a lower accuracy and a higher velocity. Depending on the accuracy of the sensors used for docking, this subprocess can sometimes be omitted.

High accuracy docking: The last subprocess uses high-accuracy sensors to slowly navigate the robot to the goal pose [44].

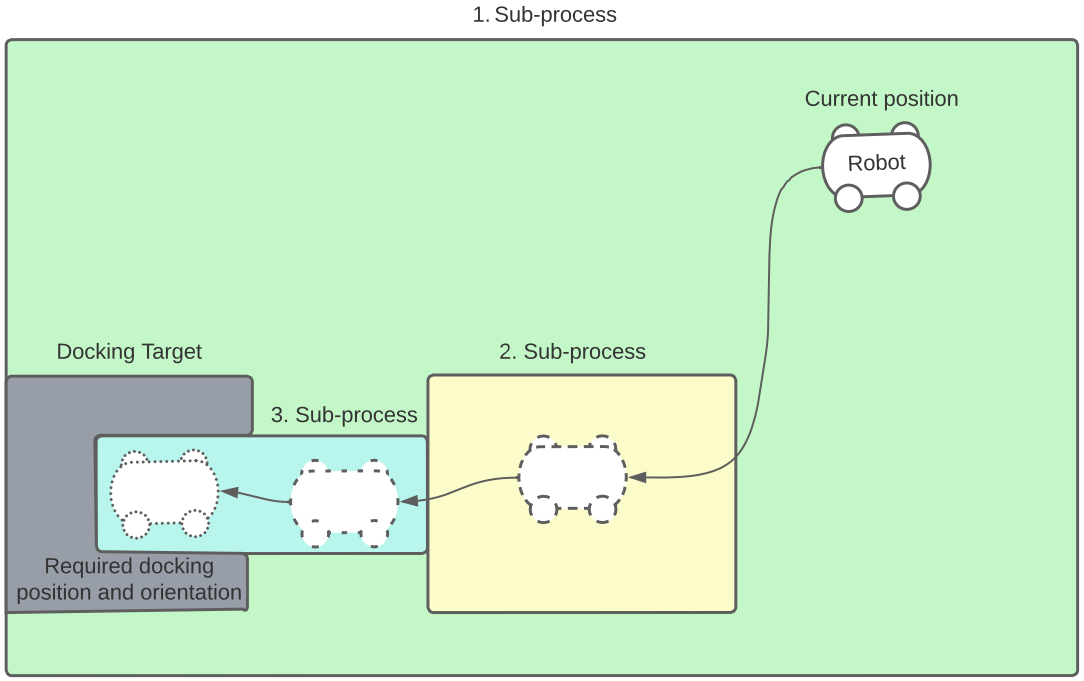


Figure 2.16: Illustration of the docking procedure. The robot starts at its current position and starts subprocess 1. When it reaches the yellow area, it starts subprocess 2 and then subprocess 3 when it reaches the blue area.

When building a docking system [45], sufficient repeatability and accuracy is required to ensure that there is no misalignment between the process robot and the carrier robot [46]. Therefore, various docking solutions as well as methods for accurate localization for robotics are presented.

Docking With Visual Recognition Markers

A 2021 paper [47] presents an algorithm that uses ArUco markers for wheeled robots with differential kinematics. The paper mentions that using IR emitters and sensors is the predominant method for mobile robots such as vacuum cleaner robots. The paper states that accurate docking requires an abundance of sensors that allow for a larger search and docking range. With a localization error of 7 cm and an angular error of about 2°, the abundance of IR sensors and emitters needed can be a major drawback to both the design and cost of the robot. The paper also mentions another solution for docking, namely a lidar that uses the generated point cloud to search for the geometric patterns of the docking stations. This method has a high enough accuracy, but it may give many false positives because the docking success rate depends on the unique geometric pattern of the docking station.

Therefore, this project proposes a different solution for locating the docking station and positioning the robot, namely the use of ArUco markers. ArUco markers have

proven to be very effective in robotics [37]. The advantage of such a system is that it is a robust tracking solution with no false positives. The disadvantage is that the localization of the robot depends on the camera resolution and that the positioning error increases proportionally to the distance between the camera and the marker(s) [47]. As mentioned earlier, the ArUco markers can achieve an accuracy of 3 mm when the camera is placed 4 m away at a low cost [36]. It is assumed that the closer the camera is to the ArUco markers, the higher the accuracy. The same is assumed for a camera with higher resolution. Therefore, the ArUco markers are being considered as a solution for docking. They allow the process robots to detect the ArUco marker on a carrier robot at a greater distance, while maintaining the required accuracy when approaching them. This means that guided docking by physical hardware, such as guide rails or a socket, is not required, further reducing the cost of the robots.

If a lidar is used, higher costs are inevitable, as both high-end and low-end lidars tend to be more expensive than most cameras, depending on the quality of the two components. The lidar used on the Polybots has an accuracy of 1% of the distance between the lidar and the walls, when the distance is less than 3 m [48]. Thus, at a distance of 2 cm, the accuracy is 2 mm, which is 1 mm higher than the required accuracy. In enhanced mode, the operating distance is between 0.2 m and 25 m. The process robot must get closer than 20 cm to dock properly. This means that the lidar would not be a viable option for the docking design, even if it were able to achieve a 1 mm accuracy during docking.

Docking With Mecanum Wheels

The following study uses a sensory fusion method that incorporates sensory data from an RGB-D camera and a laser scanner to dock a Mecanum wheeled robot. The docking sequence begins with the mobile rotating around itself when the docking station is detected. The RGB-D camera provides data such as position, orientation and depth of points of interest in a predefined image pattern. The 2D distance data from the laser scanner is modeled as line segments and all data is used to estimate the pose of the docking station. This data is used to move the robot forward until it reaches the critical region R' (as shown in Figure 2.17), where it then begins to rotate around the center of the docking station until it is in line with it. In the final phase, the robot moves forward against the docking station until a preset distance threshold is reached [49].

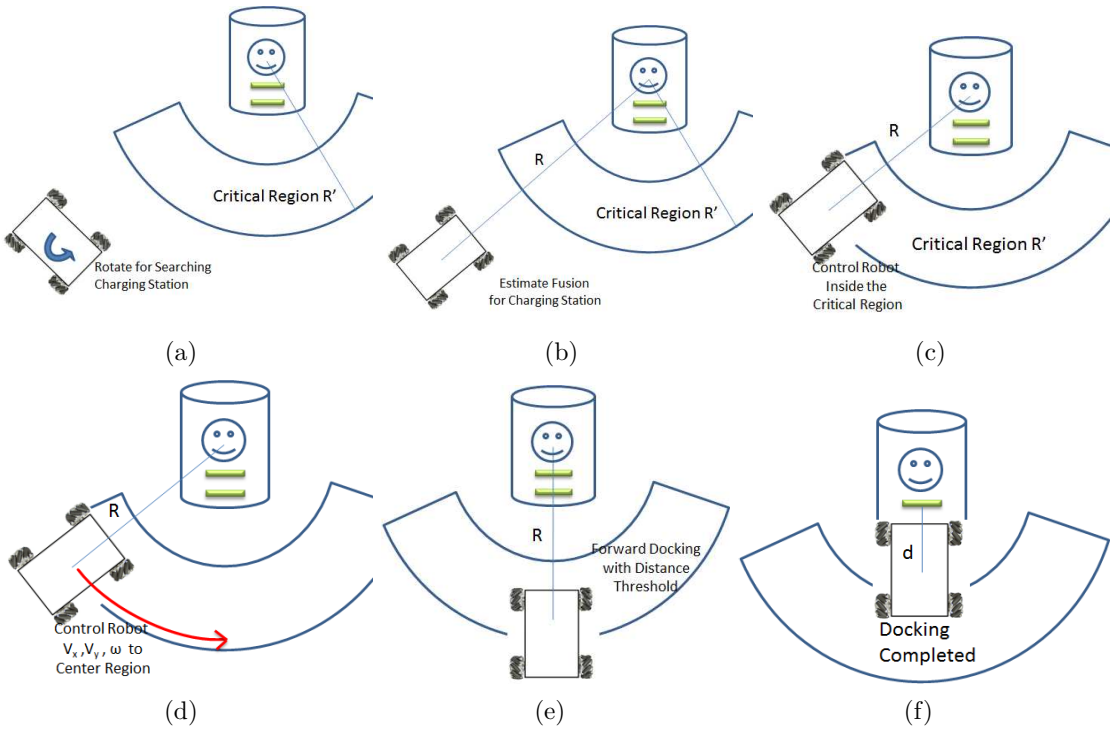


Figure 2.17: The docking sequence of the four-wheeled Mecanum robot. The advantages of Mecanum wheels become apparent when the robot reaches the critical region R', as it begins to rotate around the center of the docking station (a feature differential wheeled robots do not have) [49].

MirCharge 500

MiRCharge 500 is a commercially available charging station developed and sold by MiR and can be used by all AMRs in the MiR family [50].

The docking works by using a VL-marker, which is a V-shaped recess in combination with a flat front plate to the right, as shown in Figure 2.18. This results in a specific shape that the safety scanner can then use to recognize the charging station, as shown in Figure 2.19 [50].

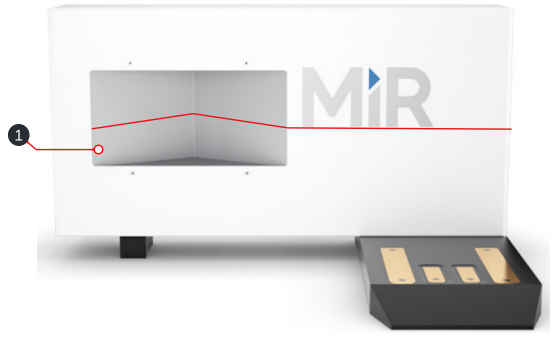


Figure 2.18: The MiRCharge 500 charging station. (1) The red line illustrates the shape of the VL-marker, seen by the laser scanner. Adapted from [50].

Figure 2.19: Laser scan of the MiRCharge 500 while a MIR robot is docking [50].

Summary of Docking

Docking with visual markers, such as ArUco markers, appears to have great potential to provide a accurate yet cost-effective solution to docking. As the paper [36] shows, ArUco markers can achieve an accuracy of about 3 mm, but as Casper Schou mentions, the required docking accuracy should be 1 mm. However, by decreasing the distance and/or increasing the resolution, the ArUco markers have the potential to achieve even higher accuracy.

Using lidar to recognize a geometric pattern seems to be a well researched and even commercially available solution for docking, but fast and accurate lidars are expensive and therefore not a viable solution when considering cost-effectiveness.

In addition, a paper [49] highlights the advantages of using Mecanum wheeled robots for docking, as they allow for omnidirectional movement that cannot be performed by differential wheeled robots. Docking of a Mecanum wheeled robot using ArUco markers is therefore considered in the development of the process robot.

2.7 ROS 1 vs. ROS 2

As described in Section 2.6, an autonomous mobile robot consists of several different functions, such as localization, navigation, obstacle detection, and docking. Developing these functions from scratch takes time. Fortunately, all of these functions are already implemented in third-party software, but even then it would take time to get this third-party software to communicate with each other. The solution to this problem is ROS.

ROS is an open-source robotics middleware suite that provides users with a set of software frameworks for robot development, message passing between processes, hardware abstraction and API calls that can be used by third-party software. There are two versions of ROS, ROS 1 and ROS 2, with the recommended distributions being Noetic Ninjemys for ROS 1 and Galactic Geochelone for ROS 2[51].

Some of the key differences between ROS 1 and ROS 2 are:

- ROS 1 uses the roscpp and rospy libraries, which are completely independent of each other and therefore their API calls are not necessarily identical. ROS 2, on the other hand, uses only one library implemented in C, called rcl. Rclcpp and rclpy (as shown in Figure 2.21) are simply bindings, so the API calls are much more similar and new core features will be available earlier in different languages.
- ROS 2 supports python3, C++ 11 and C++ 14 (with C++ 17 on the roadmap). ROS 1 supports python2 and older versions of C++.
- In ROS 2, the ROS master node no longer exists and nodes are now able to discover each other without using a ROS master. This allows redundancy in communication between nodes, as an entire ROS based system is no longer dependent on a single ROS master to handle connectivity. This difference is visualized in Figure 2.20.

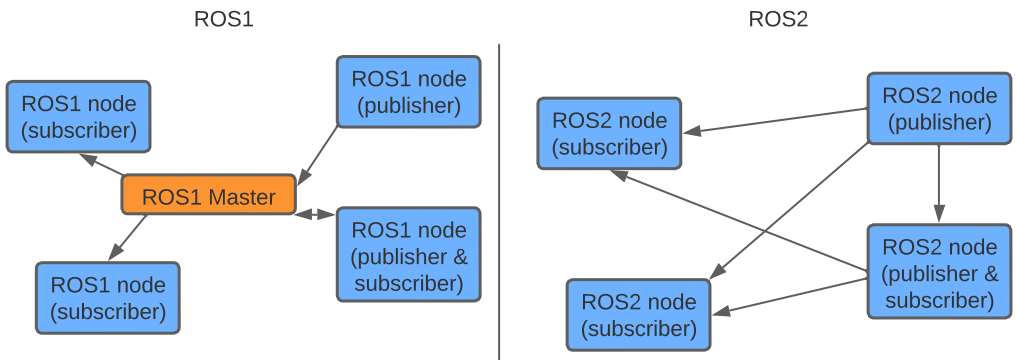


Figure 2.20: Difference in communication in ROS 1 and ROS 2.

- In ROS 2, services are no longer synchronous, but asynchronous. This means that with ROS 2, service clients no longer have to wait for a response from the service server and the main thread can run in the meantime while a callback function waits for the server to respond.
- ROS 2 also implements quality of service (QoS). This allows the user to specify how nodes should handle communications, such as message reception priority, queues, and dropping new messages when the old one is still in progress.
- ROS 1 focuses mainly on Ubuntu, but ROS 2 can also run on macOS and Windows 10 (and others). This would make it possible, for example, to run a mobile robot on Ubuntu, then a PC on Windows for simulation, and a Mac for post-processing the live feed from the mobile robot [52].

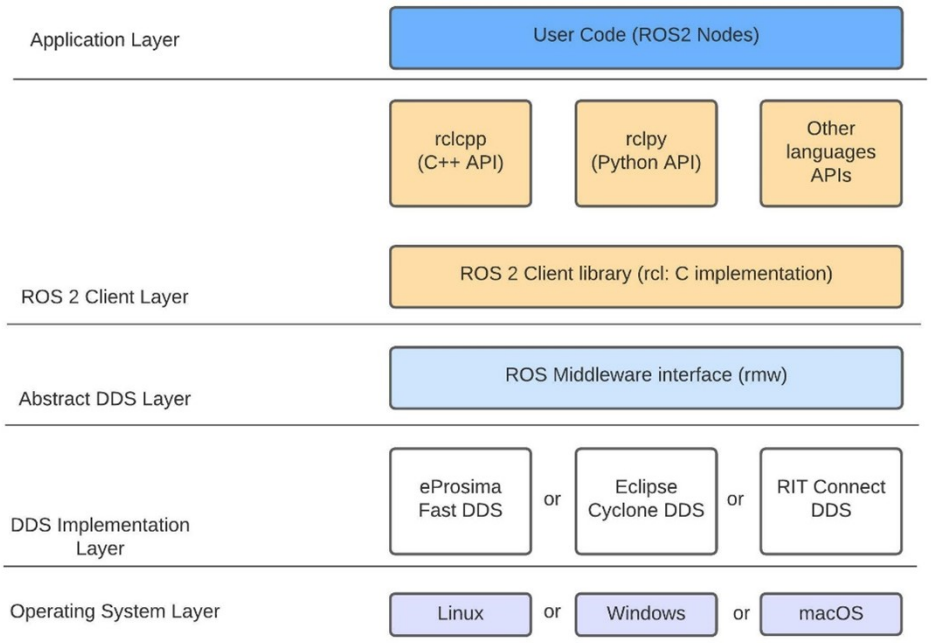


Figure 2.21: ROS 2 layers, adapted from [53].

Summary of ROS 1 vs. ROS 2

ROS 2 is still relatively new, so there is a risk that not all old packages from ROS 1 are supported. In terms of support, the latest and final distribution of ROS 1 (Noetic Ninjemys) has an end-of-life (EOL) in May 2025, so ROS itself recommends switching to ROS 2 as ROS 2 will continue to receive new distributions.

The Polybots acquired for swarm production research use ROS 1, but it is possible to establish bridge communication between ROS 1 and ROS 2 [54].

The mandatory ROS 1 master is a centralized component, in the context of communication ROS 2 has a very big advantage in terms of swarm production, because the swarm system must be a decentralized system.

Due to these facts, and because the majority of other research groups working on swarm production use ROS 2, the process robots are also being developed in ROS 2 (using the Galactic Geochelone distribution as recommended by ROS).

2.8 Proposed Solution

Based on an analysis of the existing solution for docking, robot navigation and localization and the interview with Casper Schou, a solution can be proposed. This solution will be the framework within which the design phase will take place. The goal is to design a process robot that can successfully navigate, be localized using ArUco markers, dock with a Polybot and feed LEGO bricks.

The proposed solution consists of a physical design of the process robots, a software, firmware, navigation, localization, and docking design for the robot, and a firmware design for the process. The hardware should support the software by having sensors that enable the use of the Nav2 stack in ROS 2. An ArUco marker should be placed on the robot and visible to a ceiling. The process robot needs space for a feeder that can feed LEGO bricks to a smaller carrier robot. It should have autonomous capabilities to feed the bricks without human intervention. Therefore, the process robot should be equipped with a camera that allows it to position itself correctly by reading the position of an ArUco marker attached to the carrier robot. It should also be able to feed specific quantities of LEGO bricks to specific carrier robots by using the ID capabilities of the ArUco markers. For better maneuverability, the process robots should have Mecanum wheels. This allows the robots to move in any direction while maintaining their orientation, making docking more efficient by requiring less movement.

Other factors affecting the robot hardware and design include the maximum weight, cost and size. The maximum cost of a robot must not exceed 8000 DKK and it must not weigh more than 25 kg. The size of the robot must not exceed 1×1 m, and it must be level at all times. The docking sequence should also achieve an accuracy of 1 mm to ensure a correct docking sequence with the process robot and the carrier robot.

The robot localizes itself using a Kalman filter that fuses data from the wheel odometry, the IMU, and the ArUco marker and must therefore have an IMU and wheel encoders. The Nav2 stack will be used for navigation. The robots should also all be able to be

displayed to the user via a visualization program that shows what the robot sees, what its navigation stack looks like, and where the robot localizes itself.

2.9 Final Hypothesis

Based on the findings and methods from this chapter, requirements can be made for the system. The following final hypothesis indicates where the project stands and what should be investigated.

It is possible to develop a cost-effective omnidirectional process robot platform in a ROS 2-based swarm production system that uses ArUco markers for localization, navigation, and docking, avoids obstacles using IR sensors, and successfully docks with a Polybot V1.2 using Mecanum wheels.

2.9.1 Project Objectives

Some overall objectives are listed. These objectives are not requirements but informal goals.

- Develop four process robot platforms.
- Develop a tracking system that uses the advantages of ArUco markers using cameras.
- Develop a docking algorithm that uses the advantages of ArUco markers and Mecanum wheels.
- Have the process robots navigate to the carrier robots and dock with them
- Have the process robots automatically feed LEGO bricks.

3 Requirements

The findings from Chapter 2 are now listed as requirements. These requirements define the boundaries of what the system must do. The requirements are divided into two categories, one for generic process robot platform and one for the specific case of the process robot with the LEGO brick feeder.

3.1 Generic Requirements

R 1	The localization and navigation design must allow scalability for multiple process robots.
R 2	The process robot must be able to be commanded to dock with a certain carrier robot.
R 3	The process robot must have Mecanum wheels.
R 4	The software of the process robot must be developed with ROS 2.
R 5	The process robot must not weigh more than 25 kg.
R 6	The size of a process robot must not exceed 1×1 m.
R 7	The process robot must navigate to a pose on a map without collisions.
R 8	The localization design must be able to estimate the pose of the process robot with an accuracy of 200 mm, except during the docking procedure.
R 9	All wheels of the robot must be in contact with the floor at all times thus compensate for a floor unevenness of up to 10 mm.
R 10	The size of the robot platform must be modular to be suitable for other processes.
R 11	The process must be able to communicate with the network independently of the process robot.
R 12	Multiple process robots and processes must be able to communicate over the same network.
R 13	The process robot must have at least the same computer processing power as a Raspberry Pi 3.
R 14	The docking procedure must have an accuracy of 1 mm.

3.2 Case-Specific Requirements

R CS1	A process robot must not exceed the price of 8000 DKK.
R CS2	The design of the process robot must allow for a LEGO brick feeder (as per design, see Figure 2.10) to be mounted on the process robot.
R CS3	The LEGO brick feeder must be able to receive commands wirelessly and feed n number of LEGO bricks.
R CS4	The height of the LEGO brick feeder chute must be at least 252 mm above the ground.
R CS5	When docking, the process robot must have a maximum positional error of -49.7 mm to 49.7 mm in the x-axis and -41.7 mm to 10 mm in the y-axis.

4 Design of Concept

This chapter describes the design process of the generic process robot concept (the design of the case-specific robot follows in the next chapter). This includes the hardware and software of the process robot platform based on the requirements in Chapter 3. The different areas of the system are presented and the methods and solutions used to design the physical structure, and implement a control system and a docking system are explained.

Now an overview of the concept platform is introduced. The generic design of a process robot is shown in Figure 4.1. No specific components are selected for the generic design, nor are the dimensions of the components specified. What is specified are the types of components that are decided and discussed in Chapter 2.

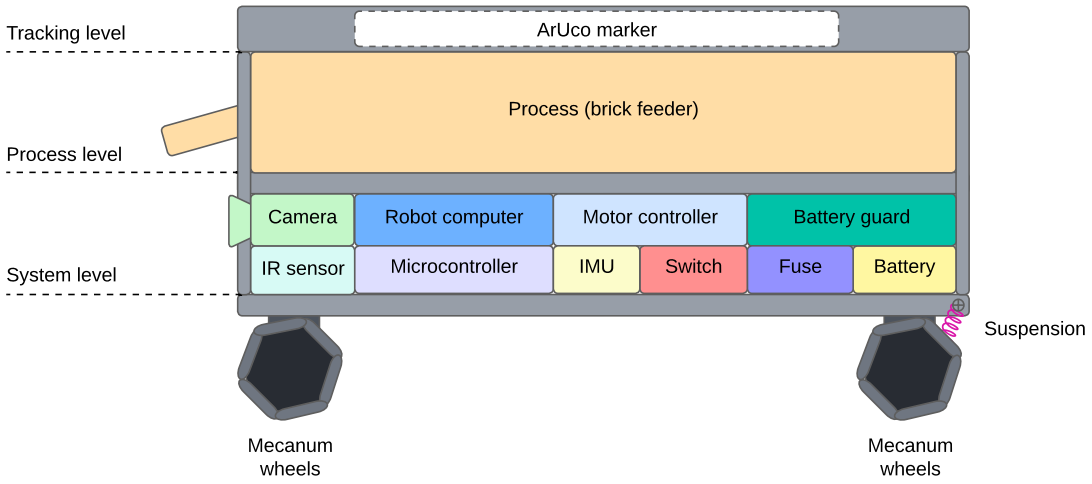


Figure 4.1: Concept design of a process robot.

The concept design of the system platform incorporating an arbitrary number of process robots is shown in Figure 4.2. The design of a process robot is generic and can be reconfigured for different types of processes.

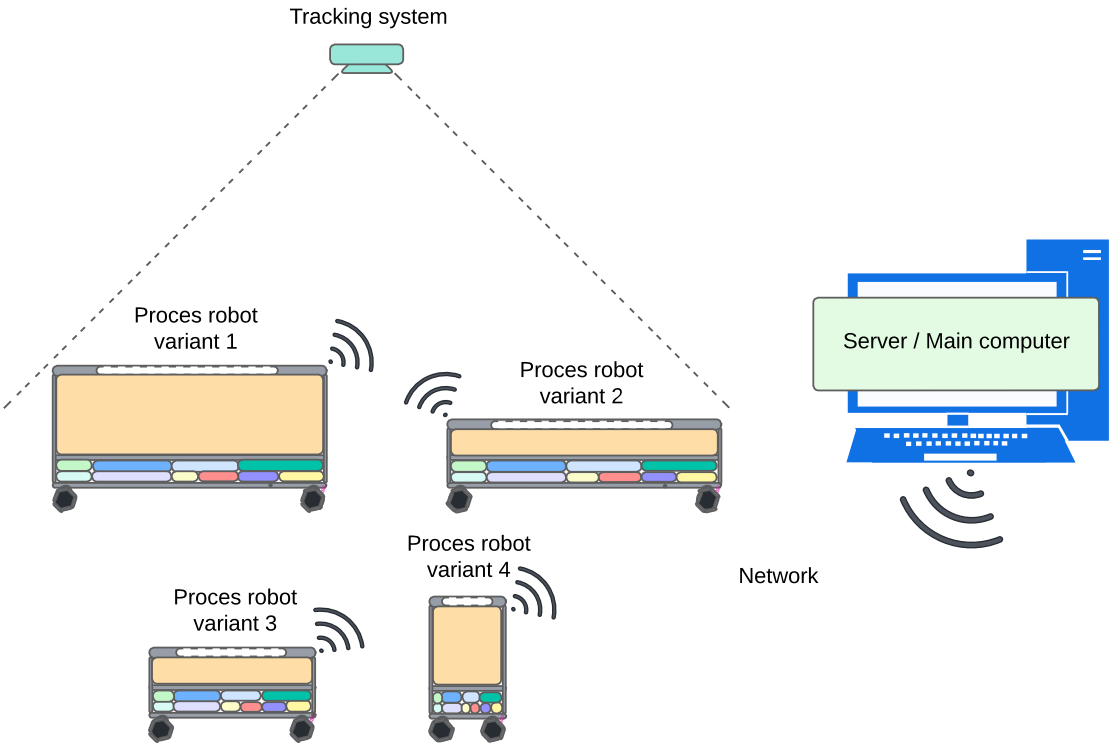


Figure 4.2: Concept design of the system platform.

To use the ArUco markers to localize the process robot, a tracking system must be implemented. The platform consists of a main computer that controls the tracking system, the navigation, the main sequence of actions, and the rest of the swarm control system. A network and a communication design by which the main computer and the robots communicate with each other is necessary in this generic design. The following sections discuss each part of the generic design in detail.

4.1 Electrical Design

The electrical design of the process robot is adapted from the electrical design recommended by linorobot for a 4WD robot, but with some adjustments (see Figure 4.3).

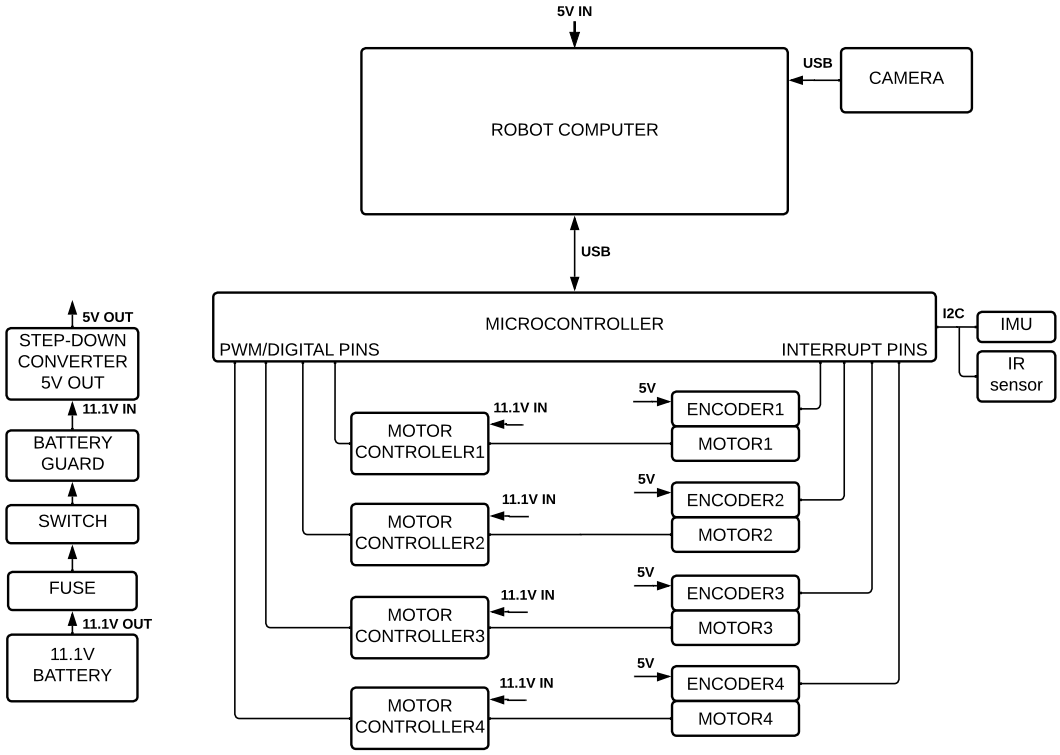


Figure 4.3: Basic circuit design of the process robot adapted from [55].

First, a fuse is placed between the battery and the toggle switch used to turn the robot on/off. The fuse is to ensure that the battery and other electronics are protected in case of a short circuit or at an excessive current draw.

Another safety measure is the use of a battery guard between the toggle switch and the hardware that draws power from the battery. This protects the battery from being undercharged, as the battery guard will cut off power when the battery reaches the minimum allowable voltage.

The motor controller is powered by the battery and controlled by the microcontroller, which also receives data from the IMU and IR sensors via a bus interface such as inter-integrated circuit (I2C). The camera and microcontroller are connected to the robot computer via USB. The motors are controlled by the motor controller and the encoders are powered by a 5 V step-down converter. The encoder data is sent to the microcontroller via interrupt pins, which ensure that each encoder step is read by the microcontroller.

4.2 Software and Firmware Design

To reduce the complexity of this project, most of the software and firmware is not developed from scratch. Instead, the linorobot2 package [56] and the associated linorobot2 firmware [55] are used and modified as needed.

4.2.1 Linorobot2 Firmware:

As shown in Figure 4.4, the linorobot2 firmware provides libraries for mobile robot control, including kinematics, PID control, motor drivers, encoder drivers, and IMU drivers. In addition, the firmware also provides libraries for selecting different motor types, encoders, and IMUs, all integrated into a single configuration file. All of this is based on a precompiled micro-ROS library that transfers all the important core concepts of ROS 2 to microcontrollers, such as nodes, publishers/subscribers, clients/services, node graphs and life cycles. And with the micro-ROS agent, it is possible to access micro-ROS nodes with the familiar ROS 2 tools and application programming interfaces (API) as if they are normal ROS nodes. So the microcontroller can publish odometry data, IMU data, and receive velocity commands as nodes in the ROS 2 network. The firmware will also be publishing the IR sensor data. This data will be used in the Localization Design and Navigation Design.

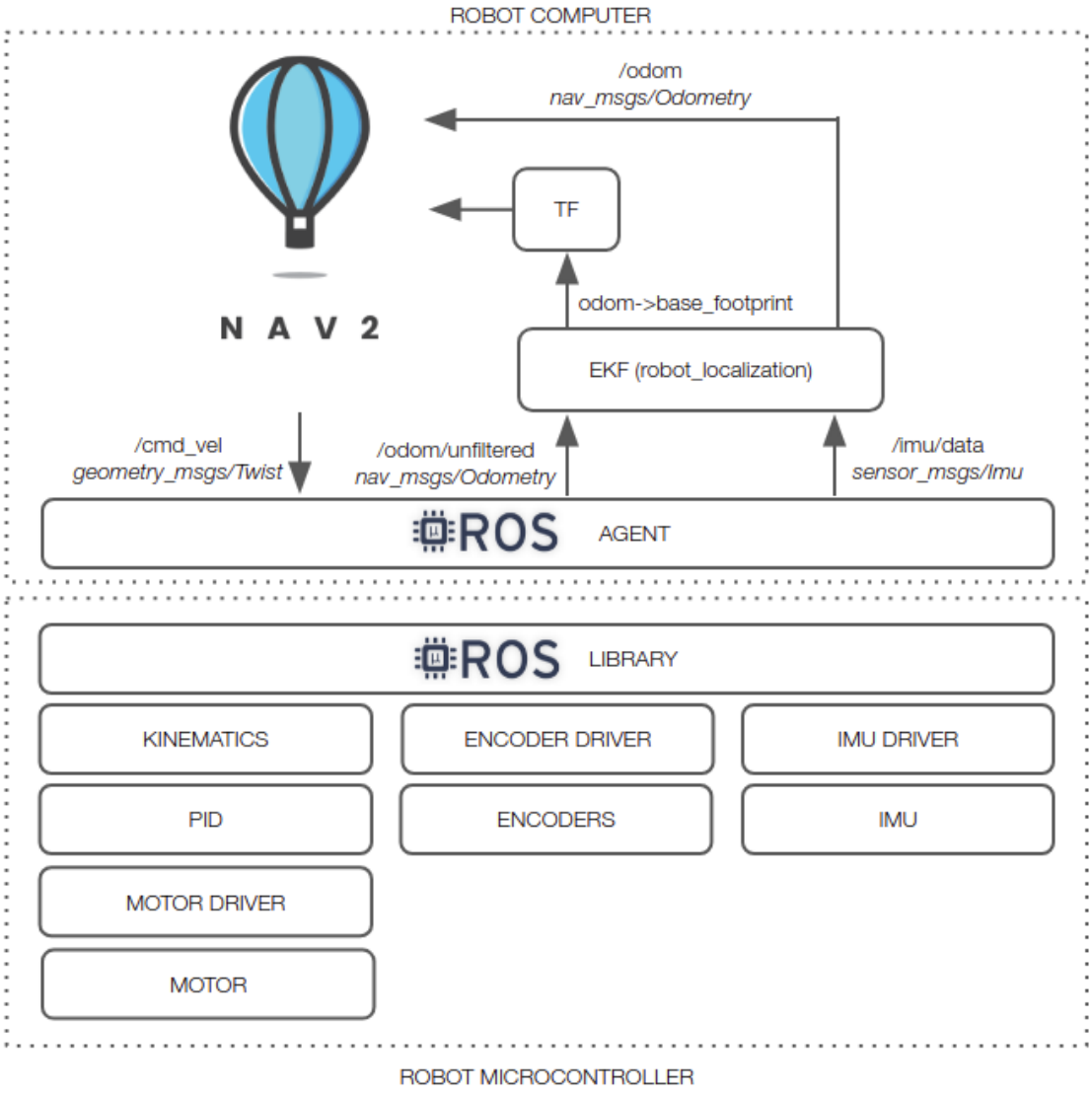


Figure 4.4: Illustrations of the linorobot2 package and the micro-ROS library and how it communicates with ROS 2 [55].

4.2.2 Linorobot2 Software:

Since the firmware design is set up to control a process robot using ROS 2, the next step is to set up the robot for use in a high-level application. The linorobot2 package is used for this purpose.

Nav2, uses URDF files to describe the robot. The URDF files are in an XML format. The URDF files are used to represent models of a robot, i.e., the kinematics, geometry [57], and placement of sensors on the robot. The linorobot2 package provides a generic URDF file that takes the properties of a particular robot from URDF property files. This generic approach fits into the platform because a new robot description can be created simply by creating a new URDF properties file.

Once a URDF description and firmware have been set up, the Nav2 stack can be used. Here, the linorobot2 package is used as inspiration. The use of the Nav2 stack is described later in the Section 4.6 and Prototype.

4.3 Communication Design

To use ROS 2 and send data between the different devices a Communication Design is created

Main computer:
To control the robots, there will be a main computer that handles the navigation, requests docking and feeding, as explained in Chapter 4. Communication within ROS 2 is handled with the data distribution service (DDS) connectivity framework where data is transported either via user datagram protocol (UDP)/internet protocol (IP), transmission control protocol (TCP)/internet protocol (IP) and others [58].

Process:
Since the process is intended to be removable, it is important that it can be transferred to another process robot and continue to communicate over the network. For a microcontroller to communicate with the ROS 2 network, it must have built-in WiFi, Bluetooth, or be connected to the robot computer via USB/serial.

To make it easier to remove the process from the process robot, it is decided that the microcontroller should have built-in WiFi and therefore connect directly to the ROS 2 network via the router. The micro-ROS library can be used for this communication, as it has implemented the *DDS-XRCE protocol*. This allows resource-constrained devices to communicate with the DDS protocol that the ROS 2 network uses, through a client/agent relationship where the agent acts as a broker for bridging the clients with the DDS world. The agent is then set up on the main computer and the client runs on the microcontroller, allowing it to communicate on the ROS 2 network like any normal node. For more information on how micro-ROS and ROS 2 communicate, see Appendix F.

Process robot:
The process robots must be able to communicate with the main computer. Since the robots must move freely and over unknown distances in complex environments, the

communication protocol must be wireless. Since ROS 2 can send data over WiFi via UDP/IP and TCP/IP, WiFi is a suitable solution. Micro-ROS is also used for communication between the robot computer and the microcontroller (to control the process robot hardware). A *Micro XRCE-DDS Agent* is then set up on the process robot to enable serial communication between the computer on the process robot and the microcontroller.

Router:
To facilitate the TCP/IP and UDP/IP communication and routing between the robots, processes and the main computer, a router is used.

The communication design showing how a robot, a process, and the main computer communicate over a network facilitated by a router is shown in Figure 4.5. This design can be scaled to include multiple process robots. An example of a communication design for multiple robots and processes is shown in Figure 4.6. The design includes a router that hosts a network that can be extended by multiple wireless access points. When a robot is out of range of the router, it can connect to the network through the access points.

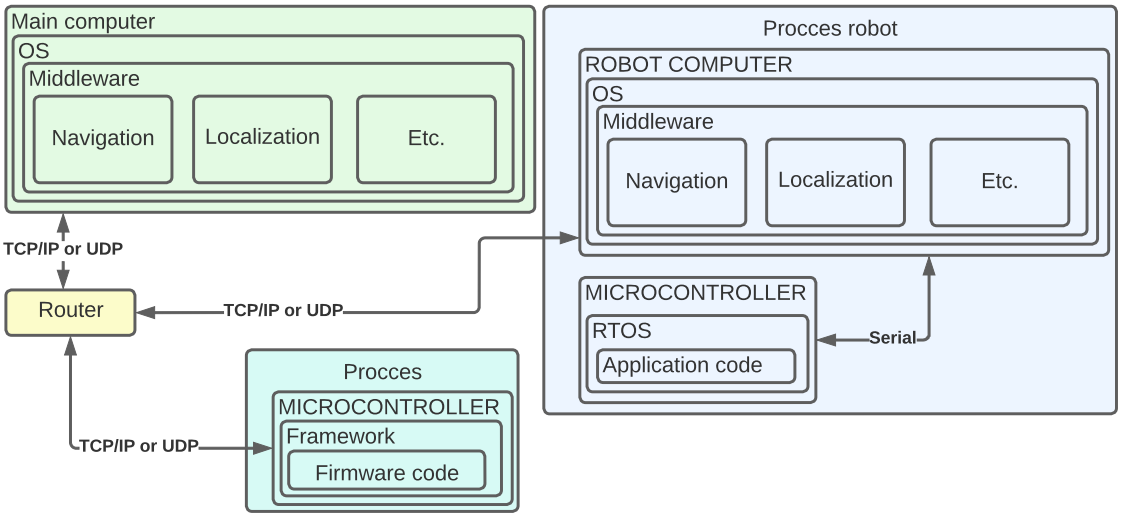


Figure 4.5: Communication design between a single robot, process, router and the main computer.

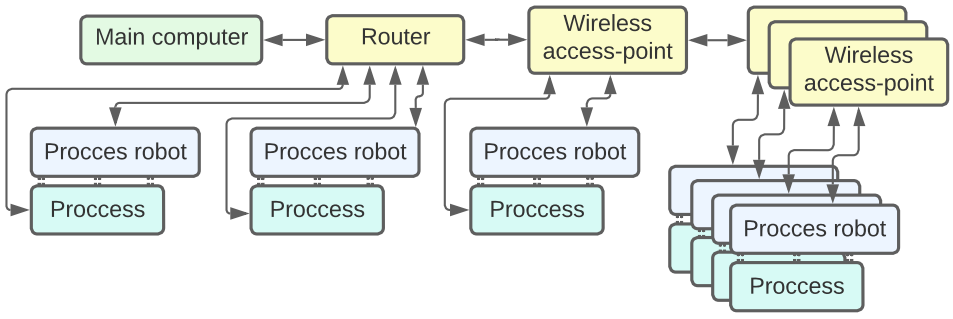


Figure 4.6: Example of a communication design with multiple routers, process robots and processes.

4.4 Tracking System Design

The task of the tracking system is to create a transform between the real world and the virtual world. An outside-in tracking system is used, as shown in Figure 4.7. ArUco markers are placed on top of the robots, and a camera is placed on the ceiling looking down, as shown in Figure 5.22.

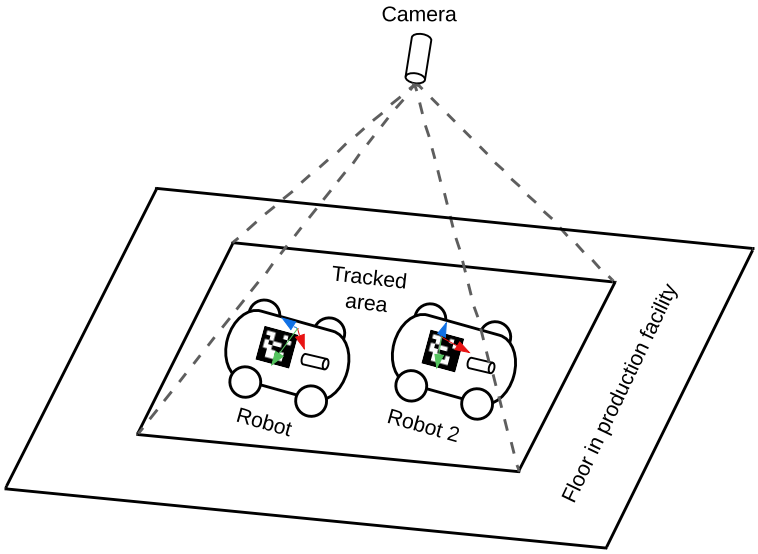


Figure 4.7: Design of the localization system. Two robots are tracked within the FOV of the camera.

The tracking system tracks the ArUco markers that are in the FOV of the camera. Their relative pose to the camera is determined and used for localization in a production room. It is possible to track multiple robots at the same time using only one camera. It is also possible to use multiple cameras placed on the ceiling to have a bigger tracking area, as to ensure that the robots are in the FOV of the cameras at most times.

Calibration and ArUco Markers

An ArUco marker is a square marker consisting of black borders with an inner binary matrix that can give each marker a unique ID, as shown in Figure 4.8. In this way, multiple robots can be tracked since each robot can have a unique ID.

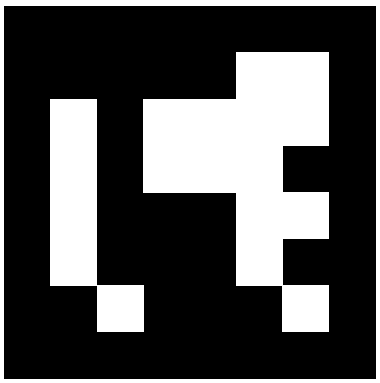


Figure 4.8: A 6×6 binary codification generated with the ID of 1 [59].

The advantage of using ArUco markers is that one marker and its four corners are sufficient to determine the pose of the marker in relation to a 2D camera. The markers have unique IDs through their internal binary codification, which also provides the ability to implement error detection and correction techniques[1]. If one of the corners is not visible to the camera, the marker cannot be tracked because all four corners must be visible.

When using the ArUco markers for a tracking system, the goal is to achieve a transform as close as possible to the real translation between points in real life and 2D points on an image projection. An illustration of a projection is shown in Figure 4.9. It is necessary to know the exact direction vector from the camera sensor to the real space. The figure shows how the corners of the marker correspond to the points projected onto the image plane. This means that there is a correct correlation between the two planes and that a correct pose of the marker can be extracted.

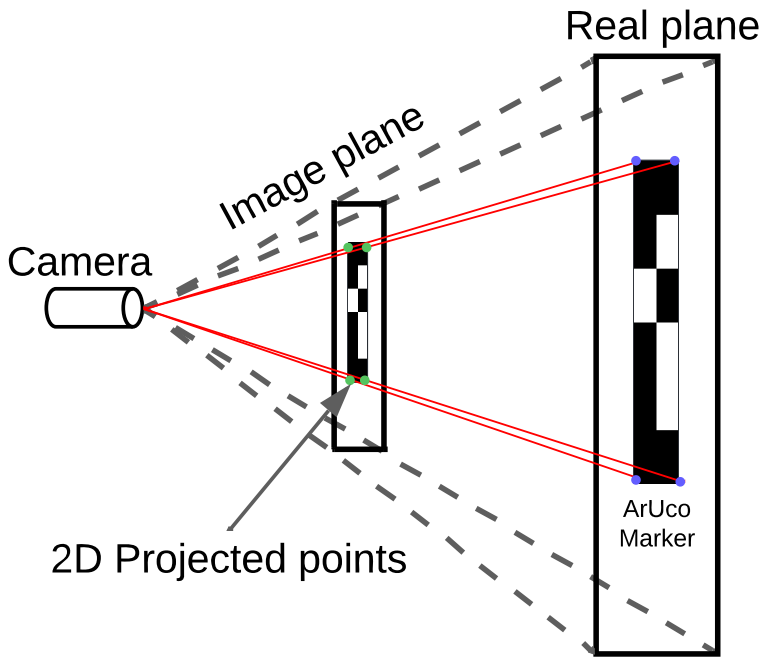


Figure 4.9: Simplified projection of an ArUco marker and its corners onto the 2D image plane.

[1]For a more detailed explanation of ArUco markers and camera calibration, see Appendix D.2.

A higher degree of distortion in an image projection results in a more incorrect calculated pose of the ArUco marker, as shown in Figure 4.10. The corners of the marker are no longer projected correctly and therefore give an incorrect pose. This is because OpenCV2, which is the library used to track the ArUco markers, assumes that the image in which it needs to track the markers is already calibrated.

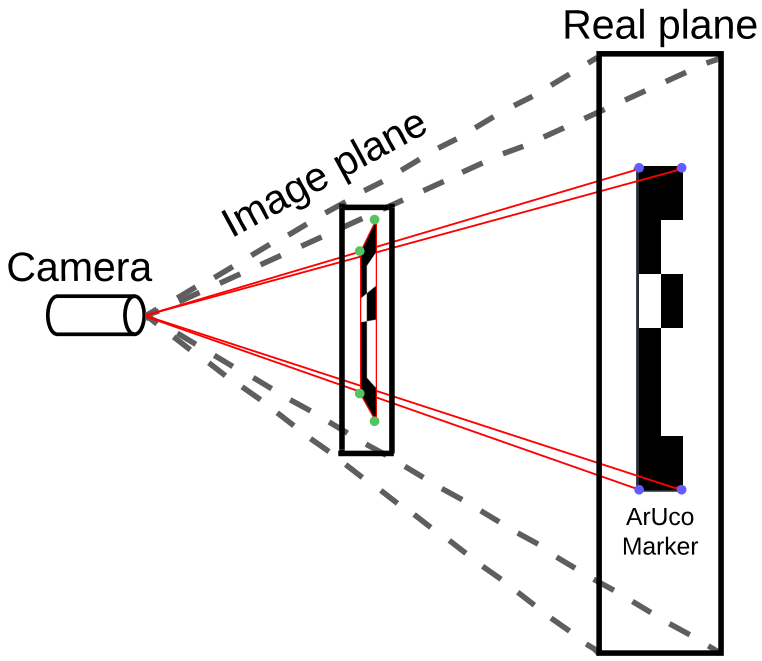


Figure 4.10: Simplified distorted projection of an ArUco marker and its corners onto the 2D image plane.

Calibration for lens distortion is an important step in ensuring that straight lines in reality are perceived as straight lines on the image plane. A fisheye lens or a panoramic camera produces a barrel distortion in any image taken with it. This interferes with the tracking system, as shown in Figure 4.10. The distortion is eliminated by properly calibrating the camera used to track the ArUco marker. OpenCV2 has built-in functions to calibrate the camera using a checkerboard pattern, as shown in Figure 4.11. The calibration allows intrinsic parameters to be created for the particular camera, eliminating the distortions that the camera or the lens have applied to the captured image.

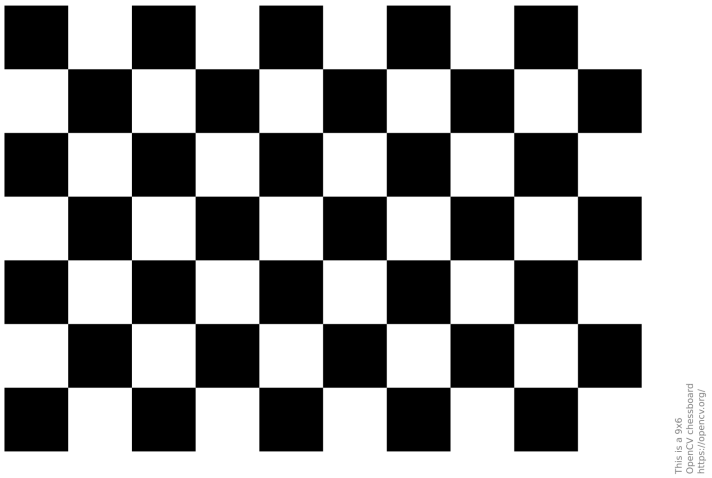


Figure 4.11: Checkerboard pattern used to calibrate for image distortions [60].

4.4.1 Vision System

The tracking design uses ArUco markers. The position of the markers is specified in the form of a frame that describes the pose of the markers relative to the cameras. The ArUco markers provide high accuracy at a low price compared to other sensor solutions [61], [62]. The tracking system also functions as an anchor in the real world, since the fixed positions of the cameras on a map are known. This can be used in conjunction with wheel encoders, IMUs, etc. to create a prediction of the location of the robot on a virtual map. The disadvantage of such a system is the fixed point setup, which requires multiple cameras to be mounted on the ceiling or walls of a room. The possibility of the markers being obscured can also cause problems. This happens when the cameras are not placed properly or the view of the markers is blocked by something. When this is the case, the position of the robot becomes increasingly inaccurate over time, as the wheel encoders and IMU drift exponentially and are the only sensors on which the position of the robots can be based. For this reason, both systems are needed to accurately predict and estimate the pose of the robots. This is discussed in more detail in Localization Design. ArUco marker tracking is more precise than IMUs and wheel encoders in short term as there is no drift. This is because small changes in the pixels can cause the estimated position and rotation of the ArUco marker to jump rapidly between different values. These values increase and decrease depending on the distance between the camera and the ArUco marker and the resolution of the camera. The farther the marker is from the camera, the less accurate the pose estimate becomes.

The choice of ArUco marker size, proper resolution, field of view (FOV), frames per second (FPS), shutter speed, etc. of the cameras depends on several factors:

- What accuracy is needed for the tracking system?
- How powerful is the main server that processes the images when searching for the markers?

- How many cameras can be placed in the production facility?
- What response time is required to have a stable system?
- How much of the production facility needs to be covered by the cameras?
- How long can the robot navigate without being visible to the cameras when using other sensors such as wheel encoders and an IMU?
- What is the maximum distance the ArUco markers can be from the camera without compromising the stability of the tracking system?

All of these factors play a role in the design of the tracking system and how it is implemented. Testing these requirements in different locations could give an idea of what the optimal setup would look like in most cases, but that is not possible within the scope of this project. What *can* be tested is how accurate and precise the tracking system is at different distances. This is explored in the chapter Prototype.

Using several cameras for tracking

The tracking system is designed to track multiple robots within a tracking area while being scalable for an entire production floor. Therefore, multiple cameras are often required to cover the space in which the robots can navigate. This report will cover two ways of setting up the cameras, namely overlapping and not overlapping FOV. Figure 4.12 shows a setup where the cameras are placed side by side without overlapping the FOV. This creates a small dead zone where the cameras FOV hit each other. The whole marker is not visible inside the dead zone, as each camera will only see one half of it each, when it is moving from one camera to the next. To avoid this problem, the different cameras can be *stitched* together and create a larger image. For example, this could be a system that allows up to four cameras in a square to synchronize their shutter and create a composite image. Or, for a larger system, process all the images from the cameras as one large composite image.

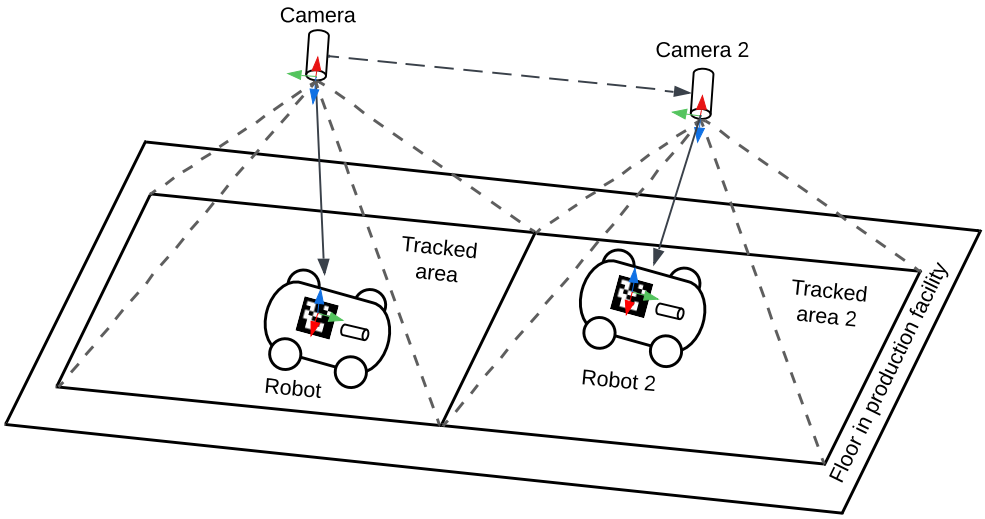


Figure 4.12: Example of tracking two robots with two cameras in a production facility. There is no overlap between the cameras FOV.

Figure 4.13 shows a structure where the cameras have a small overlap, approximately the size of an ArUco marker. This is to ensure that the markers are always visible and can be tracked regardless of position or orientation. This structure costs more depending on how large the FOV of the cameras is and how much they need to overlap. Since there is a loss of FOV when the cameras overlap, this would have to be compensated for by using more cameras in the production room.

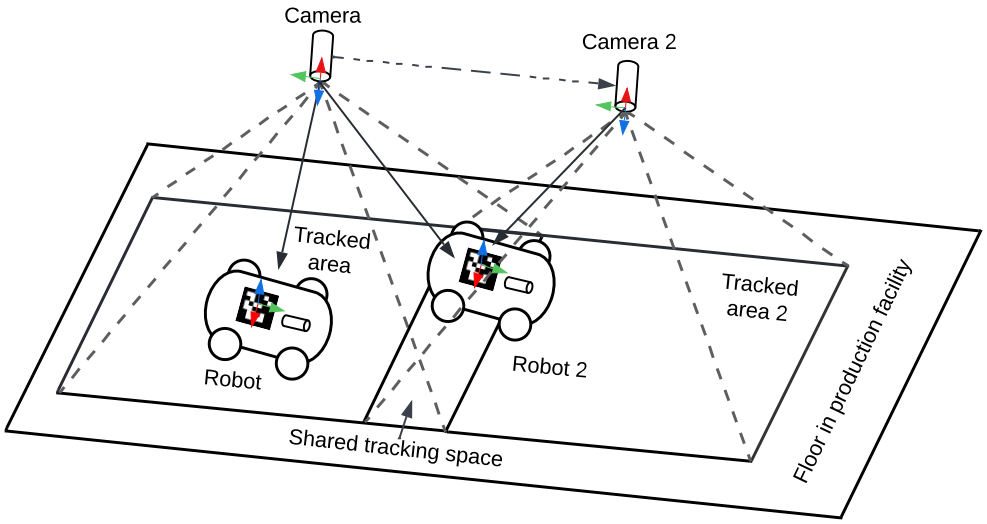


Figure 4.13: Another example of tracking two robots, but with overlapping camera FOV. Both cameras can track the same robot in the overlap.

The decision of which system to implement depends more on the customer than on the implementation of the system. In this project, the focus is on getting the tracking system to work with one camera, so a multi-camera system will not be further explored.

Finding the transform between a map and ArUco markers

To locate a process robot with ROS 2 in a map, it is important to know exactly the orientation of the real world map and the virtual map. If the initial position is off by a certain amount, it could mean that the robot bumps into walls without knowing it, or turns too much and bumps into another wall (see Figure 4.14). Therefore, a system must be developed to ensure that the real world map and the virtual map are properly aligned.

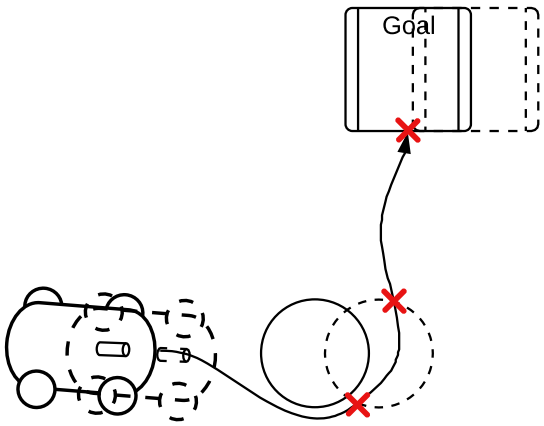


Figure 4.14: Example of misalignment. The dotted lines are the actual positions and the straight lines are the assumed positions. The red crosses are collisions.

It can be difficult to place a camera with an exact angle and position on the ceiling without the proper tools. Therefore, the camera can instead be placed where it has sufficient floor coverage with its FOV, and then calibrated from there using an ArUco marker with extrinsic parameters. Since the translation and orientation of the ArUco markers are derived from the pose of the camera, it is possible to reverse this system and locate the camera using the marker instead. In the localization system, the camera has a fixed known position on a map, but conversely, the ArUco marker has a fixed known position on the map. This now allows the system to determine where the camera is on the map relative to the ArUco marker. When the exact position of the marker on the map and in reality is known, they can be more easily aligned. Since the ArUco marker is on the ground, the alignment only needs to be done in two dimensions x, y instead of three dimensions x, y, z for the camera. The ArUco marker can then act as an anchor for the camera in the real world so that the extrinsic parameters of the camera can be derived.

4.5 Localization Design

The essential function of the localization design is to localize the robot in a known map. Many of the navigation packages in ROS 2 do not require particular sensors for localization, but certain conventions must be followed [63]. For localization, there must be at least one transform tree between the frames, as shown in Figure 4.15.

Figure 4.15: Transform tree convention.

Odometry and IMU data are published by the firmware design. These data are sampled on the main computer and converted into transforms that describe where the robot is in relation to a given point, as shown in Figure 4.16. Both odometry data and IMU data can be used to generate the `odom` → `base_link` transform. Since both transforms

contain error and both represent the same transform, the IMU and the odometry data are fused using a Kalman filter to obtain a better pose estimate.

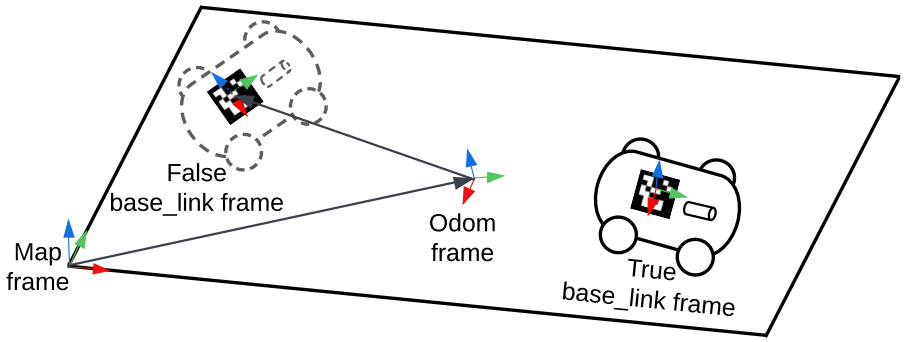


Figure 4.16: Illustration of how the odometry tracks the robot.

4.5.1 Kalman Filter

The odometry and IMU data are sampled over the ROS 2 network on the main computer. Here, a Kalman filter will output a fused estimate of the pose. The Kalman filter takes a 15-dimensional state vector from each sensor:

$$(X, Y, Z, roll, pitch, yaw, \dot{X}, \dot{Y}, \dot{Z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{X}, \ddot{Y}, \ddot{Z})$$

This state vector includes the pose, the respective velocities, and the acceleration of the robot estimated by the sensors. For example, the IMU does not directly provide X, Y, Z. However, it provides $\ddot{X}, \ddot{Y}, \ddot{Z}$, which can be integrated to $\dot{X}, \dot{Y}, \dot{Z}$ and then to X, Y, Z. In this way, two random variables are generated over time, namely the 15-dimensional state vector from each sensor. The robot state is calculated by computing the joint probability distribution of the two random variables. This robot state contains the current estimated X, Y and yaw pose values of the robot, which are used as the `odom` → `base_link` transform.

4.5.2 Resetting Odometry Using the Tracking System

Using only the odometry and IMU data in a Kalman filter for localization is a dead reckoning approach. As explained in Localization Solutions, this dead reckoning approach leads to a cumulative error. In addition, a measurement from the `map` frame to the pose of the robot is required at each startup to relate the `map` frame to the `odom` frame, which is undesirable.

Therefore, the tracking system is now used. The tracking system provides a transform between the ArUco anchor and the ArUco marker on the robot. This transform can also be used as the `odom` → `base_link` transform. The localization system is designed so that the odometry transform converges to the same transform as the inverse transform of the tracking system. An illustration of this approach can be seen in Figure 4.17.

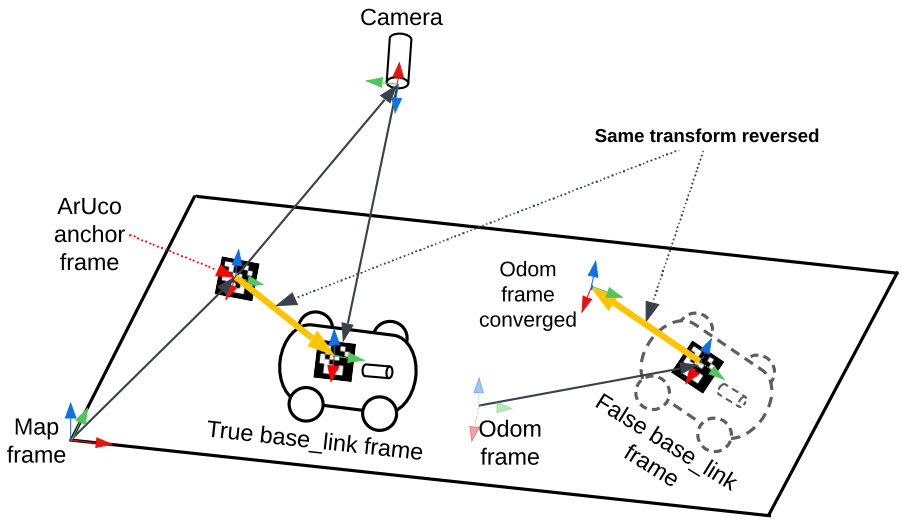


Figure 4.17: Visualization of the converging `odom` frame.

The `odom` frame (converged) is not set in relation to the `map` frame at this time. In the tracking system, the transform between the `odom` frame (converged) and the `map` frame is set equal to the transform between the `map` frame and the `ArUco anchor` frame. This causes the odometry transform to be reset to the transform estimated by the tracking system. Thus, the `odom` frame overlaps the `ArUco anchor` frame, as shown in Figure 4.18:

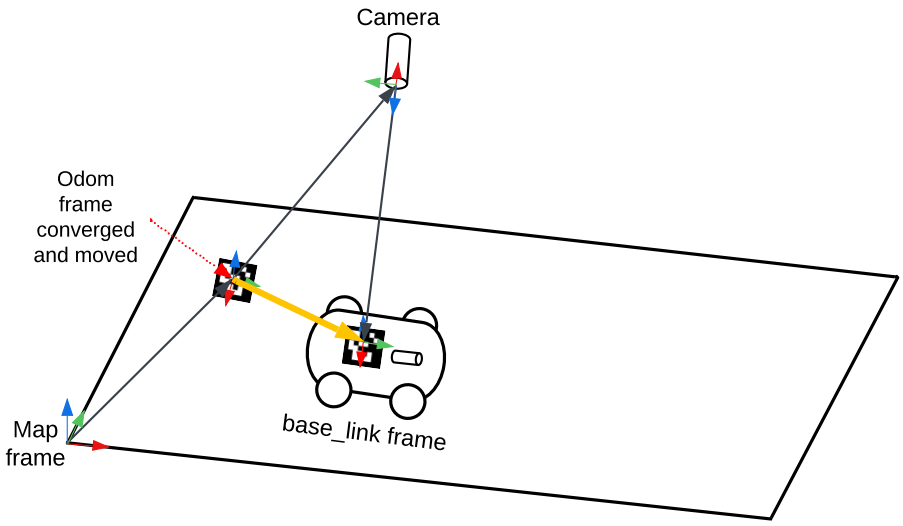


Figure 4.18: Visualization of the relationship between the `odom` frame, the `ArUco anchor` frame and the `map` frame.

In this design, the tracking system would reset the odometry transform to the values output by the tracking system, which is not dead reckoning. This effectively ignores the odometry published by the firmware once the marker is visible to the tracking system. Once the marker is no longer in FOV, the odometry takes over. This design has the advantage that the error in the tracking system is not cumulative, but also if the camera no longer sees the ArUco marker, the dead reckoning approach takes over until the camera sees the ArUco marker again, which then resets the odometry transform and

corrects the dead reckoning error. The tracking system pose estimate is also included in the extended Kalman filter, in the same way as the IMU and odometry. The localization design can be scaled to include multiple robots and cameras, as visualized in Figure 4.19.

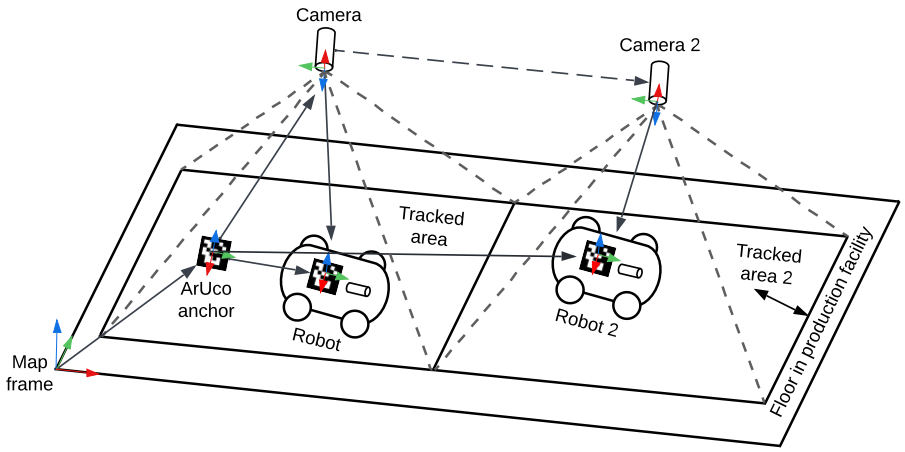


Figure 4.19: Illustration of how the tracking system can be scaled to include multiple robots and cameras.

An illustration of the system architecture so far is shown in Figure 4.20.

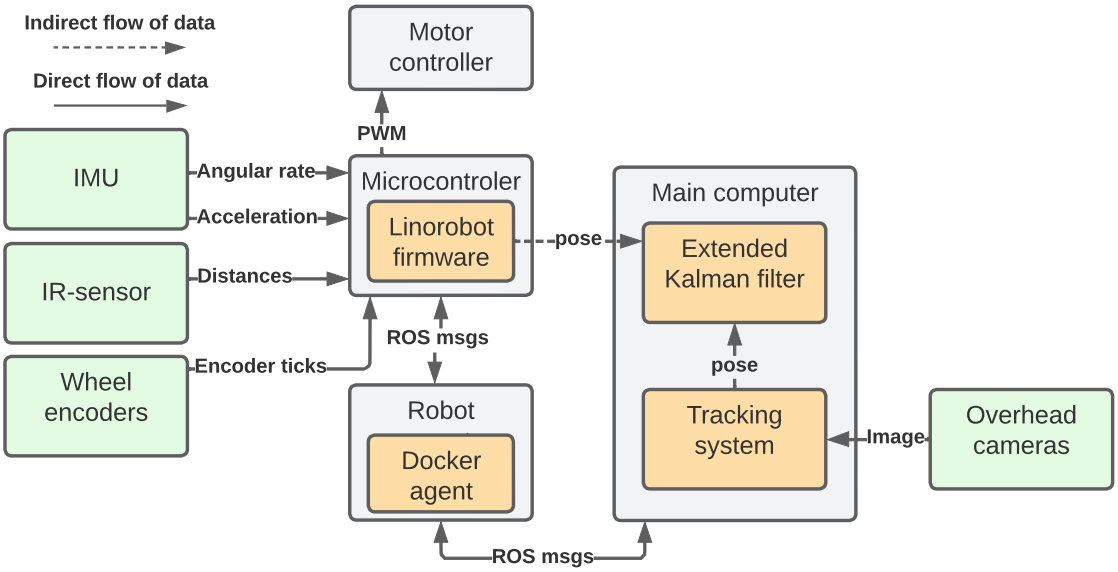


Figure 4.20: Platform system design with the localization implemented.

The localization design is needed for the navigation design, which is described in the next section.

4.6 Navigation Design

The purpose of the navigation design is to generate a collision-free path from the process robot to a position and control the robot to follow the path without collisions, and

then return the result when the position is reached

The Nav2 stack from ROS 2 [64] is used for the navigation design of the process robots. The Nav2 stack runs on the main computer and is used by a main program that also runs on the main computer. A description of how the Nav2 stack works can be found in Appendix D.1. How the Nav2 stack is integrated into the system is shown in Figure 4.21:

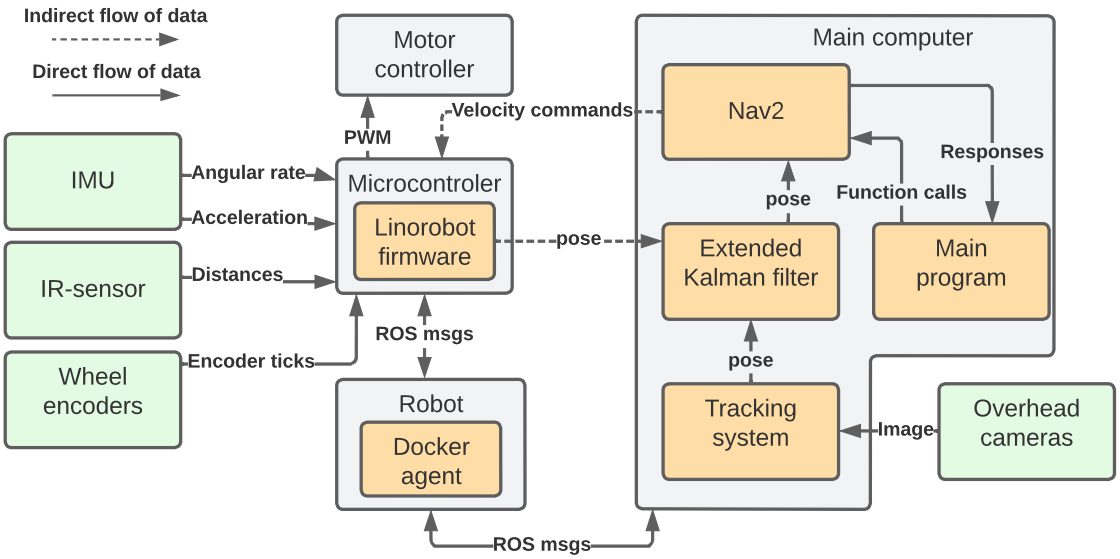


Figure 4.21: Platform system design with the navigation design implemented.

The main program creates pose goals and passes them to the Nav2 stack. The Nav2 stack then uses the current pose from the localization design to generate a collision-free global path from that pose to the goal pose. Once a path is created, the Nav2 stack begins sending velocity commands to the firmware of the microcontroller on the robot. The velocity commands are sent as ROS 2 messages over the ROS 2 network to the robot computer and then via the serial connection to the microcontroller. The microcontroller then uses PWM signals to control the motors that drive the wheels.

This navigation design can be used independently of a docking sequence, e.g. if the process robot just needs to go somewhere, but the main program can also use the navigation design in combination with a docking sequence.

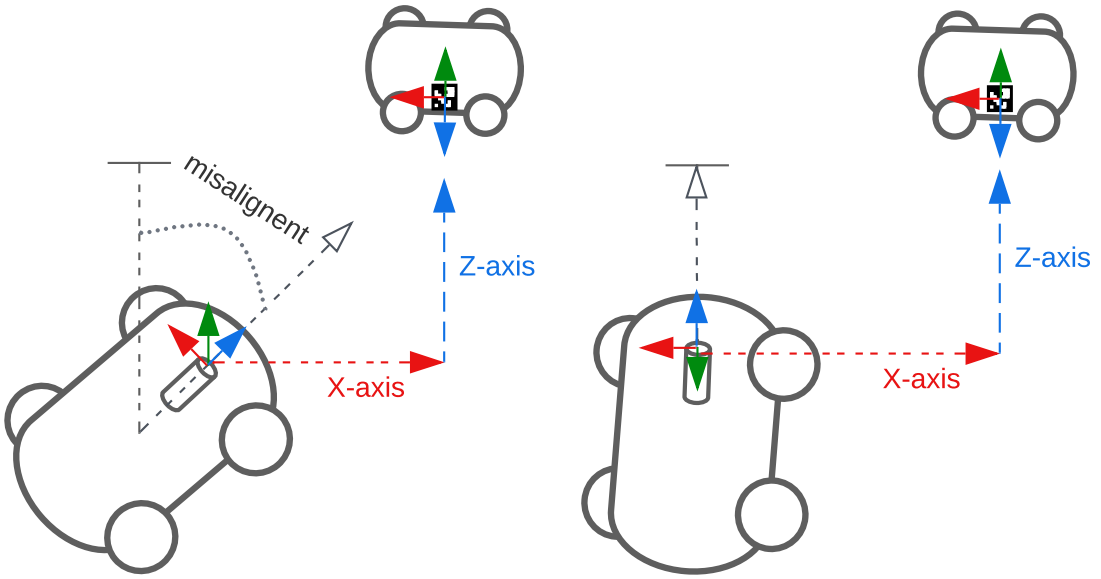
4.7 Robot Docking Design

The docking solution for the process robots refers to the positioning of the process robot to perform a specific task related to the process mounted on them. As described in Section 2.6.4, the docking procedure is divided into three parts. First, the process robot must move close to the carrier robot, then approximate the docking procedure, and finally attempt to position itself by moving slowly and with high precision. This is done to get as close as possible to the final pose. A docking program is developed that is executed on the robot computer.

The docking sequence of the process robot is developed around ArUco markers. The ArUco marker is placed on the carrier robot while a camera is placed on the process robot. If the pose of the camera on the process robot is known, the path to the carrier robot for docking can be estimated.

1. Part of docking sequence
The first part of the docking procedure is performed using the navigation design. The process robot moves from its current position to a position closer to the carrier robot using Nav2. The process robot is oriented in the direction of the carrier robot so that the on-board camera is pointed at the ArUco marker on the side of the carrier robot.

2. Part of docking sequence
The process robot tries to align itself using its camera so that the ArUco marker is in a certain position and at a certain angle relative to the camera. This position and angle are pre-determined depending on the process robot and the type of process it is to perform on the carrier robot. The alignment of the pose of the process robot relative to the carrier robot is divided into three stages, where the previous stage must be successful before the next stage can be tackled: The first stage is angular alignment about the z-axis (yaw). Alignment is accomplished by rotating the process robot about its center point, as shown in Figure 4.22a. This rotation ensures that the ArUco marker is properly aligned with the camera so that the movements in the other axes are parallel. There may be a small misalignment on the z-axis within a threshold that can be set in the software. If the misalignment is within the threshold, it is considered a successful alignment. The misalignment on the z-axis is continuously checked for all stages to avoid fluctuations when the robot is trying to reach an ideal angle but does not have the mobility to do so.



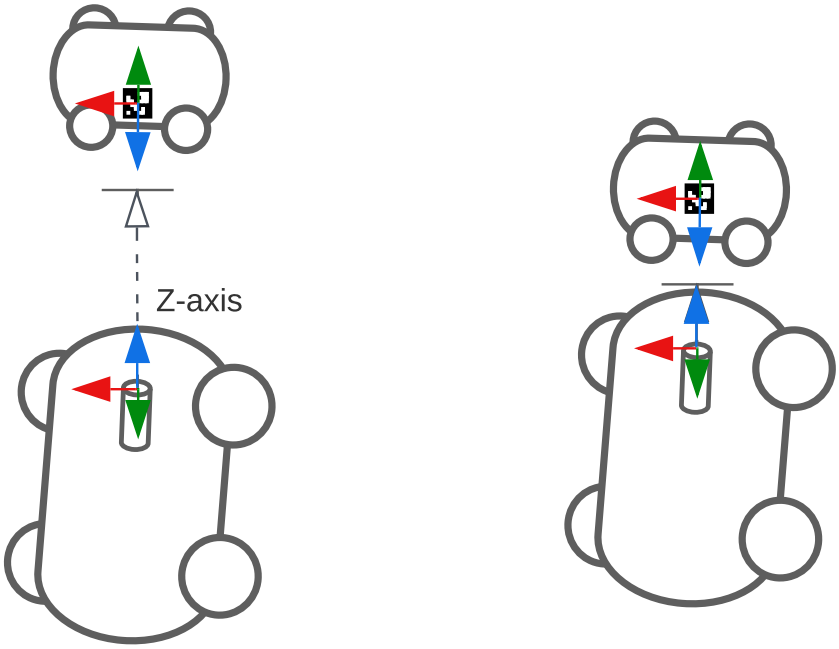
(a) At the first stage the robot rotates until it is parallel with z-axis of the ArUco marker.

(b) The second stage where the robot moves along the x-axis until it aligns with the ArUco markers z-axis

Figure 4.22: First stage of the docking procedure.

If the alignment matches, the next stage is to align the z-axis of the process robot with that of the ArUco marker by moving the robot along the x-axis of the ArUco marker

frame. This can be done by moving the process robot to the right or left. In the final stage, the distance between the process robot and the carrier robot is adjusted when the other axes are aligned. This can be done by moving the process robot forward until the correct distance between the camera and the ArUco marker is reached, as shown in Figure 4.23b. If the stage before the current stage is too misaligned and not within the thresholds, the current stage can be paused and the stage before it will start alignment again. If the stage then successfully returns that it is properly aligned again, the other stage continues.



(a) At the second stage, the robot moves towards the ArUco marker.

(b) Third stage in which the distance between the process robot and the carrier robot is reached

Figure 4.23: Second and third stage where the robot moves close to the carrier robot and docks

3. Part of docking sequence
As the robot approaches the ArUco marker, the misalignment thresholds are adjusted based on how close the robot is to the ArUco marker. The speed of the robot is also reduced as it gets closer. This allows for higher precision. The specified distance to the ArUco marker can be adjusted both in the z-direction and x-direction ($x - y$ plane on the map) to suit the particular application. The same applies to the orientation about the yaw axis of the robot.

4.7.1 Starting the Process

When all stages are successfully completed and the process robot is in the correct pose. Then a response is sent from the robot computer to the main computer. The main program on the main computer can then be set to send ROS 2 messages over the ROS network to activate the process and execute the process task. How the docking design is integrated into the system is shown in Figure 4.24.

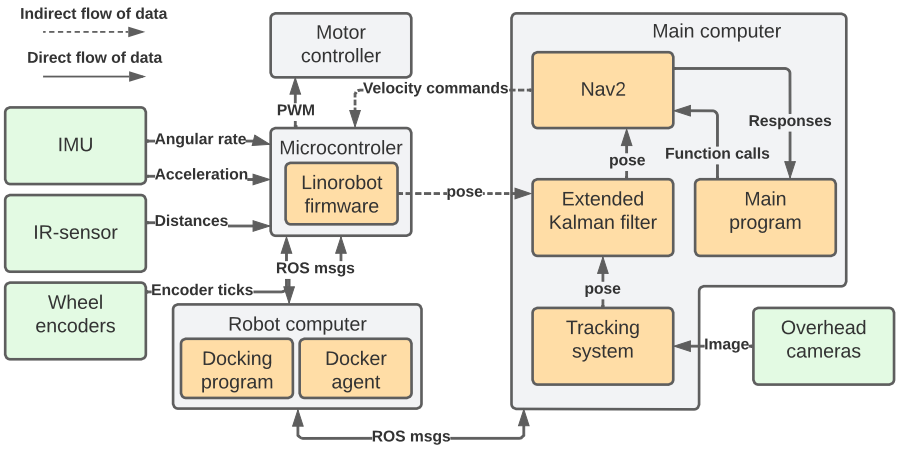


Figure 4.24: The platform system design with implemented docking design.

Summary of Design of Concept

A generic design for a process is not considered possible. However, the requirements are that it must communicate with the main computer, as described in Communication Design. Since a design for a generic system has been developed that fulfills the required functions for the system, the generic design is now used to create a prototype which is described in the next chapter.

5 Prototype

A prototype is now designed based on the concept of a generic process robot platform defined in Chapter 4. The concept is adapted to AAU's LEGO prepackaging case, as mentioned in Section 2.4.1.

5.1 Components Selection

The first step is the selection of the individual components, as this has an impact on the dimensions and layout of the physical design, but also on the implementation of the Linorobot2 firmware, software, and the development of the docking function.

Each component is selected based on knowledge acquired in Chapter 2, availability, and the characteristics of the process mounted on the robot. The selected components and how their interconnection are shown in Figure 5.1. All the component prices and sources for the specifications can be found in Appendix E.6, which is referenced whenever specifications or prices for the components are mentioned.

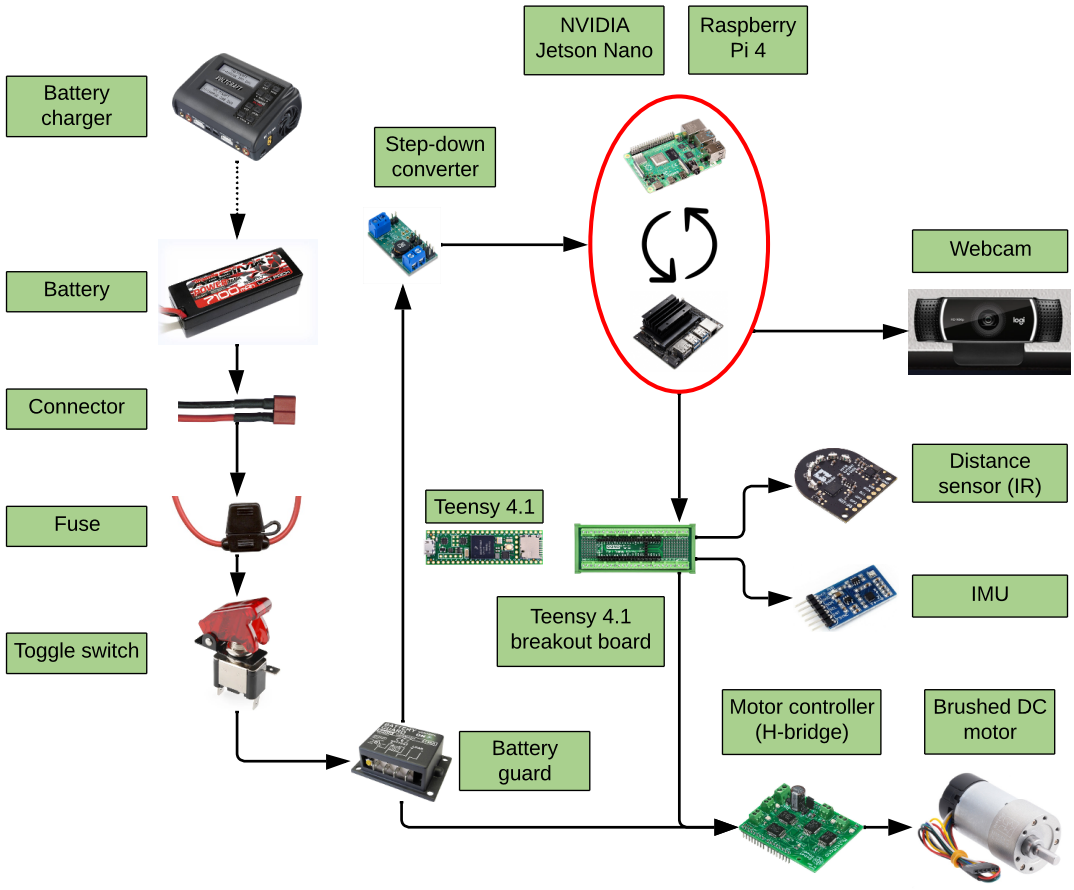


Figure 5.1: All electrical components of the process robot.

Teensy 4.1

The microcontroller is the middleman between the robot computer, the sensors and the motor controllers. It provides real-time communication between them. Therefore, it is important that the microcontroller has sufficient processing power and pinouts.

Linorobot2 supports most Teensy microcontrollers, including the latest Teensy 4.1, which features a Coretex-M7 ARM processor with a clock frequency of 600 MHz and a total of 55 I/O pins that are easily accessible due to its larger form factor compared to the Teensy 4.0. Therefore, the Teensy 4.1 was chosen as the microcontroller.

Teensy 4.1 Breakout Board

For easier and safer access to the pins, a breakout board is used that leads all I/O pins of the Teensy 4.1 to screw terminals.

Brushed DC Motors

The DC motors were among the first components selected because other components depend on their specifications, such as the battery and the motor controller.

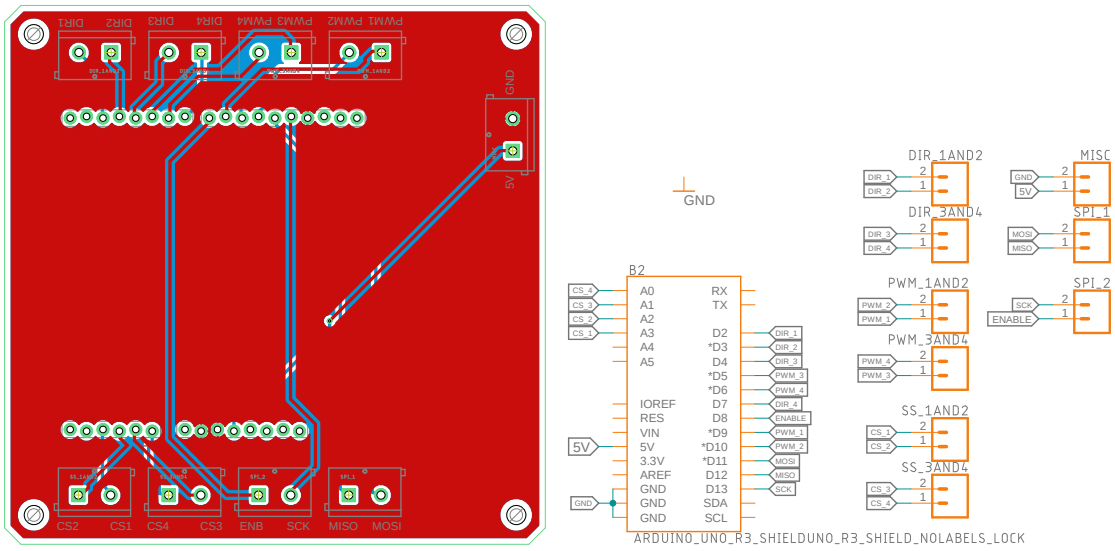
Pololu's 37D and 25D brushed DC motors were considered, because they are also used by Polybots and Martin Bieber Jensen recommended them [26]. Both types of motors are equipped with encoders. The 37D has a 64 CPR dual-channel hall effect encoder, while the 25D has only 48 CPR. The 37D type also has a higher torque to RPM ratio, which is important because the process robot must carry the additional weight of a mounted process.

Therefore, it was decided to use the Pololu 37D motor. It was chosen with a gear ratio of 30:1, which corresponds to 330 RPM and a stall torque of 14 kg (can be found in Appendix E.6). This results in a top speed of 1.728 m/s with a wheel diameter of 100 mm, which is more than sufficient because, according to Casper Schou, the process robots do not need to go faster than the carrier robots. It would also have been possible to buy the same motor, but with a higher gear ratio of 50:1, which would result in a lower top speed, but higher torque and higher counts per revolution for the encoders. The reason why this was not chosen, was that at that time there was no information about the required speed of the process robots and it was assumed that they would need approximately the same speed as the carrier robots.

Motor Controller (H-bridge)

The MultiMoto 4-channel H-bridge was chosen for the motor controller. This motor controller supports an input voltage range of 6 V to 36 V. It has four independently controlled channels, each of which can handle a continuous current draw of 6.5 A and a peak current draw of 8 A. It also features adjustable current limiting, overheat limiting, and short circuit protection for motors and batteries.

One problem, however, is that the MultiMoto controller is designed as an Arduino Uno shield. Therefore, a breakout board was designed to route the motor controller I/O pins to screw terminals, as shown in Figure 5.2.

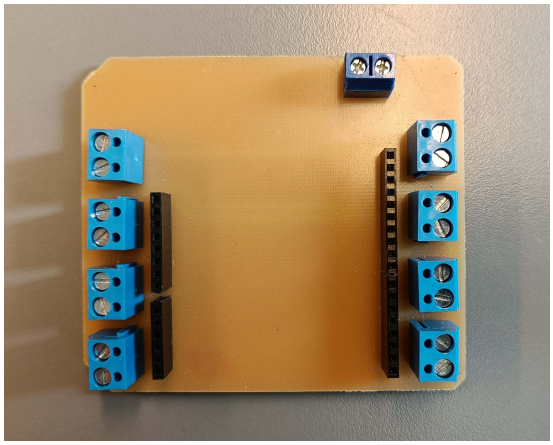


(a) Board design of the breakout board for the Multimoto motor controller.

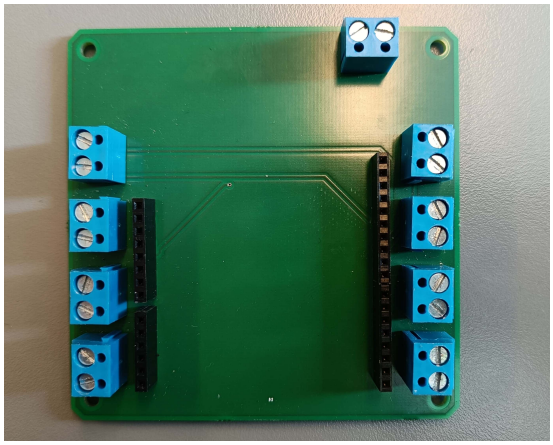
(b) Schematic design of the breakout board for the Multimoto motor controller.

Figure 5.2: PCB design of the Multimoto motor driver breakout board.

A breakout board prototype was made using a CNC milling machine (see Figure 5.3a) to test the design and fix any possible errors before ordering the PCBs online at a PCB manufacturer. The final version is shown in Figure 5.3b.



(a) Prototype of the Multimoto breakout board.



(b) Final Multimoto breakout board ordered from a PCB manufacturer.

Figure 5.3: PCB design of the Multimoto motor driver breakout board.

Robot Computer - Nvidia Jetson Nano

The robot computer is the main processing unit of the process robot. It runs ROS 2, Nav2 and the linorobot2 package, and also receives camera feedback and communicates

with the Teensy 4.1 microcontroller.

Originally, the Raspberry Pi 3 or newer was considered because it is easy to use and has enough processing power to handle the processing of camera feedback (as mentioned in Section 2.6.1). However, due to the current chip shortage on the market, it was not possible to acquire it in time. Instead, four NVIDIA Jetson Nanos were borrowed from CLAAUDIA (AAU's data research service) [65]. The Raspberry Pi 4 and the Jetson Nano are very similar in terms of raw CPU processing power, but the Jetson Nano has an added advantage, namely its 128 CUDA core NVIDIA Maxwell GPU, which makes it better suited for deep learning and image processing [66]. This is actually an advantage since an image processing function has to be implemented to find the ArUco markers in the docking procedure.

Battery

To operate the process robot without a power outlet, a battery is required. This battery powers all components of the process robot and the process itself. The process could have its own battery, but since there was no budget for it at the time of development, it must share the battery with the process robot for now. Therefore, it is important that the battery can handle the power consumption of the entire system.

There are no requirements for how long the process robot must be able to operate on a single battery charge. However, the Polybot has a 5 Ah battery, which is sufficient for about two to three hours, so the goal is to find a battery with a capacity that is roughly equivalent to the runtime of the Polybot. As for the battery type, the Polybots use LiPO batteries, and Martin [26] also recommends them as they can tolerate high charge and discharge rates, so it was decided to use them as well. LiPo batteries often come in one cell (3.7 V), two cells (7.4 V), three cells (11.1 V), and four cells (14.8 V). In this case, a 3-cell battery is desirable because it is closest to the voltage the motors need and because it is not recommended to run motors at a higher voltage than they are designed for.

To choose the battery capacity, it is necessary to know the power consumption of the process robot. A measurement was made to determine the average power consumption of the process robot (including the process itself), and the result was 108.8 W. Therefore, a 29.4 Ah battery is required for three hours of operation. However, due to budget constraints and the lack of high capacity LiPo batteries, it was decided to use LiPO batteries in the 7-8 Ah range.

For safety reasons, it is important to ensure that the battery can handle the worst-case current draw. Batteries often have a C-rating that describes the discharge/charge capability. To determine what C-rating is required for the process robot, the total power consumption of all components must be calculated and an appropriate battery capacity estimated. The C-rating can then be calculated using the following formula [67]:

$$Cr = \frac{\text{current of discharge (A)}}{\text{rated energy (Ah)}} \tag{5.1}$$

The motors each have a stall power consumption of 66 W, and the NVIDIA Jetson Nano has a peak power consumption of 25 W. The brick feeder has a solenoid that consumes

5 W when activated, and the brick feeder also has a stepper motor that spins the drum and a motor that vibrates the lane, both with undefined specifications. However, it has been measured that the two together consume 60 W when operating. This results in a worst-case maximum power consumption of about 354 W, which results in a maximum current consumption of about 32 A for an 11.1 V battery.

The C-rating is calculated for a 7 Ah battery with respect to the worst-case current draw because, as can be seen in Equation 5.1, the C-rating decreases as the rated energy increases but the current of discharge does not. This means that batteries with a capacity higher than 7 Ah, but the same C-rating are guaranteed to be safe to use.

The current of discharge and the rated energy can now be used in Equation 5.1 to determine the minimum C-rating required for the system:

$$\frac{32\,\text{A}}{7\,\text{Ah}} = 4.57\,\text{C} \tag{5.2}$$

In terms of safety, a higher C-rating than required is desirable. For example, Polybots use a 5 Ah battery with a C-rating of 45 C. For the process robot, an Absima (LiPo) 7.1 Ah battery with a C-rating of 60 C was available online.

Fuse

To increase electrical safety, a 40 A fuse was installed between the battery and the toggle switch. It ensures that the battery is safe in the event of an accidental short circuit and does not blow if the worst-case current draw occurs.

Battery Guard

The Kemo M148A was chosen for the battery guard. This battery guard automatically switches off the battery connection when it reaches a low voltage. The threshold is adjustable with a potentiometer and ranges from about 10.4-13.3 V. The selected LiPO battery can be discharged down to 3 V per cell without being damaged [68]. This means that it must be recharged when the voltage of the three cells together reaches 9 V. However, since the lowest threshold of the battery guard is 10.4 V, it could have been lower to better match the 9 V minimum discharge limit for the battery. Since an unbalanced discharge may occur, a safety buffer of 0.1 V per cell was chosen, resulting in a cut-off voltage of 9.3 V.

To determine the resistance required for the desired cut-off voltage, the 100 kΩ potentiometer was removed and a 500 kΩ potentiometer (available at AAU) was connected to the input on a breadboard. The potentiometer was then adjusted until the cut-off voltage reached 9.3 V, whereupon the newly adjusted resistance was measured to be approximately 383 kΩ. The 500 kΩ potentiometer was too large to fit in the battery guard, so a resistor of the same resistance was soldered to the input of the battery guard, as shown in Figure 5.4.

Figure 5.4: The red triangle indicates the new 383 kΩ resistor on the battery guard.

Step-down converter

The Jetson Nano cannot be powered directly with the 11.1 V of the battery because it requires an input voltage of 5 V. Therefore, it is necessary to use a step-down converter that can convert the 11.1 V to 5 V. The step-down converter used in this project can output 6 A, which is sufficient to power the Jetson Nano including additional peripherals.

Webcam

A camera is needed for both the tracking system and each process robot. A HD (720p) webcam came with the NVIDIA Jetson Nanos that was borrowed from CLAAUDIA [65]. It was decided to start with these for the process robot and tracking system, but a fullHD (1080p) webcam was also borrowed from AAU Smart Lab for further testing if a higher resolution is needed.

IMU

Regarding the IMU, linorobot2 supports a few existing IMUs:

- GY-85
- MPU6060
- MPU9150
- MPU9250

The Polybots use an MPU9250 IMU and have had good experiences with it. However, they also mentioned that there is a newer MPU9255 version that is back-compatible with the existing libraries for the MPU9250, but consumes less power.

A Waveshare 10 DOF IMU was selected that includes the MPU9255 plus a BMP280 barometric pressure sensor and a temperature sensor. All the data can be retrieved via an I2C interface, which is also supported by the Teensy 4.1.

Distance Sensor (IR)

For obstacle detection, it was decided whether to use IR distance sensors or ultrasonic distance sensors. Both types have their advantages and disadvantages and depend heavily on the environment and the application of the device. For example, IR distance sensors can be interfered with by other IR sources such as the sun or reflective surfaces. Ultrasonic distance sensors can also be interfered by other ultrasonic sources or by previous pulses reflected back from multiple objects, which can occur in reverberant rooms [69]. Since the process robot has to navigate and detect obstacles in a large production hall (AAU Smart Lab), it was decided to use the IR distance sensors.

The choice fell on a 3-channel IR distance sensor from Pololu. The three channels mean that the sensor has a wide FOV of about 160 deg and can measure a distance of up to 1 m. The sensor data is sent via I2C, so the IR distance sensor and the IMU can use the same two wires for data transmission, since I2C is a bus interface.

5.2 Physical Design

After selecting the components, it is possible to determine the dimensions and layout of the physical design. The entire design was created in Solidworks 3D software, where it was possible to quickly sketch and contemplate about ideas before transferring them to the real world. Manufacturing processes such as additive manufacturing, CNC milling and laser cutting were used to produce the designed parts. The parts that needed to be made from more durable materials were fabricated by AAU's engineering assistants.

5.2.1 Aluminum Frame

The frame of the process robot is made of 20×20 mm aluminum profiles, so it is possible to reconfigure the design during development, as shown in Figure 5.5. In some scenarios, it may also be beneficial to change the composition of the frame to adapt it to different types of processes. The frame has three levels, one nearest the floor for the electronics, a second above for the process, and a third as a roof for the ArUco marker, as shown in Figure 5.5. The levels have an inner dimension of 300×500 mm and an outer dimension of 340×540 mm. The height between the first and the second level is fixed at 100 mm to take into account the feeding height of the brick feeder. The height between the second and the third level is adjustable, as the profiles can be replaced with shorter/longer profiles to suit other processes. In this case, the length of the profiles is adjusted to the height of the brick feeder, which is 365 mm. To create a more robust structure, two support profiles are used to stiffen the frame in the first and second levels (indicated with red boxes in Figure 5.5). These also have another purpose, as they can also be used to mount electronics or parts of the process if they need to be attached to the frame. In this case, for example, four motors on the electronics level and the vibrator part of the process are mounted on support profiles. These support profiles can be moved or more can be added to mount other processes. At each corner of the second level of the process robot, an aluminum profile is mounted vertically to hold the third level on which an ArUco marker is attached.

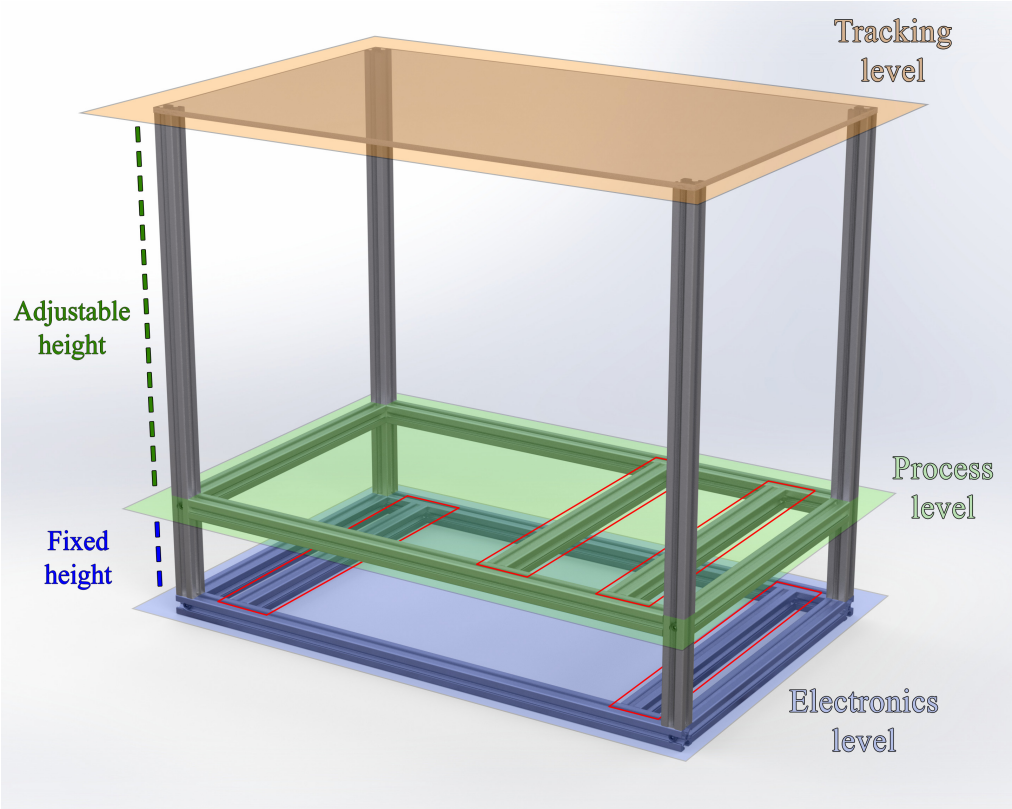


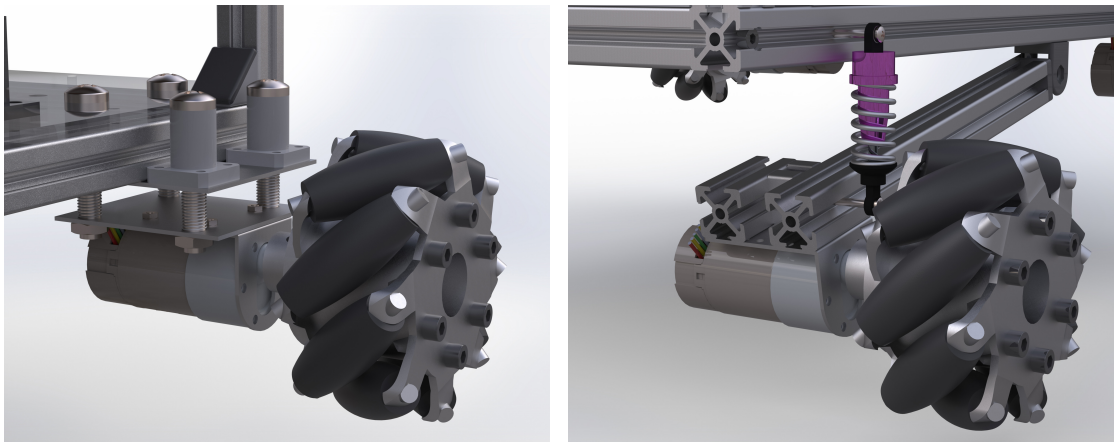
Figure 5.5: Aluminum frame of the process robot. The blue plane indicates the first level for the electronics, the green plane indicates the second level for the process, and the orange plane indicates the third level for the ArUco marker. The red boxes indicate the support profiles.

5.2.2 Suspension

Requirement R 10 describes that all wheels of the robot must be in contact with the floor at all times. To compensate for unevenness of the factory floor of up to 10 mm, it is decided that the two rear wheels are suspended. The front wheels of the robot are not suspended, as it is important that the robot has a fixed height at the front when the process is running. Two suspension designs were made to determine which features should be included to achieve the best solution.

The first design was developed as a compact design to maintain the current clearance below the robot as much as possible. It consists of four linear ball bearings and four chrome rods with flat heads surrounded by springs, as shown in Figure 5.6a. However, the cost of this solution proved to be a large part of the total cost of the robot because of the parts required. The linear ball bearings alone cost 2088 DKK for a robot, which is a quarter of the total budget listed in requirement R CS1. Also, the chrome rods are not commercially available, so they would have to be custom made. In addition, this design contains no damping other than the friction between the linear ball bearings and the chrome rods. Therefore, the suspension can become a bouncy suspension when force is applied if the friction acting as damping is too low. The design also relies heavily on the height adjustment of the four rods being identical. All in all, this design would be too complex to manufacture and is therefore not used in the prototype.

The second design was developed based on other commercially available robots with suspensions. It consists of an aluminum profile attached to a free hinge (354 DKK for a pair) on the frame, with a commercially available suspension in between (129 DKK for a pair), as shown in Figure 5.6b. The price of the suspension per robot is then 483 DKK plus the aluminum profile. This suspension is made with oil damping and has a length of 65 mm and a suspension travel of 10.7 mm. It turned out that this solution increased the clearance under the robot to an extent that was not desired. Therefore, a new iteration was developed from this design to reduce costs while achieving a more compact solution as in the first design.



(a) First suspension design. **(b)** Second suspension design.

Figure 5.6: Two different suspension designs.

The new iteration consists of the same suspension as in the second design (129 DKK for a pair), a commercially available 80×80 mm hinge (58 DKK for a pair), and a custom-made suspension adapter. The price of the design per robot is then 187 DKK plus the custom-made adapter made of a 3 mm aluminum sheet. The upper part of the suspension is mounted further up the frame and the lower part is mounted directly to the motor with the custom-made adapter, as shown in Figure 5.7. This results in a more compact yet cost-effective design, so this iteration is implemented in the final prototype.

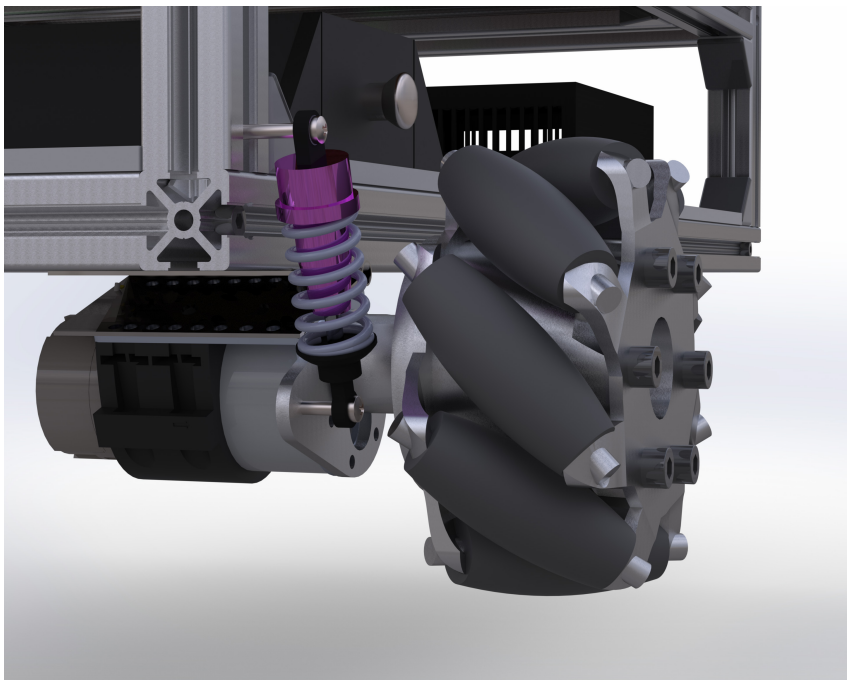


Figure 5.7: New iteration of the second design.

5.2.3 Suspension Adapters

The suspension adapters were developed because the suspensions could not be attached directly to the motors. They were created through iterations where the angle and the length of the adapter were changed between tests to achieve the best possible result. The different iterations were cut out of MDF with a laser cutter to allow for quick fabrication, while the MDF was strong enough to perform the tests, as shown in Figure 5.8.

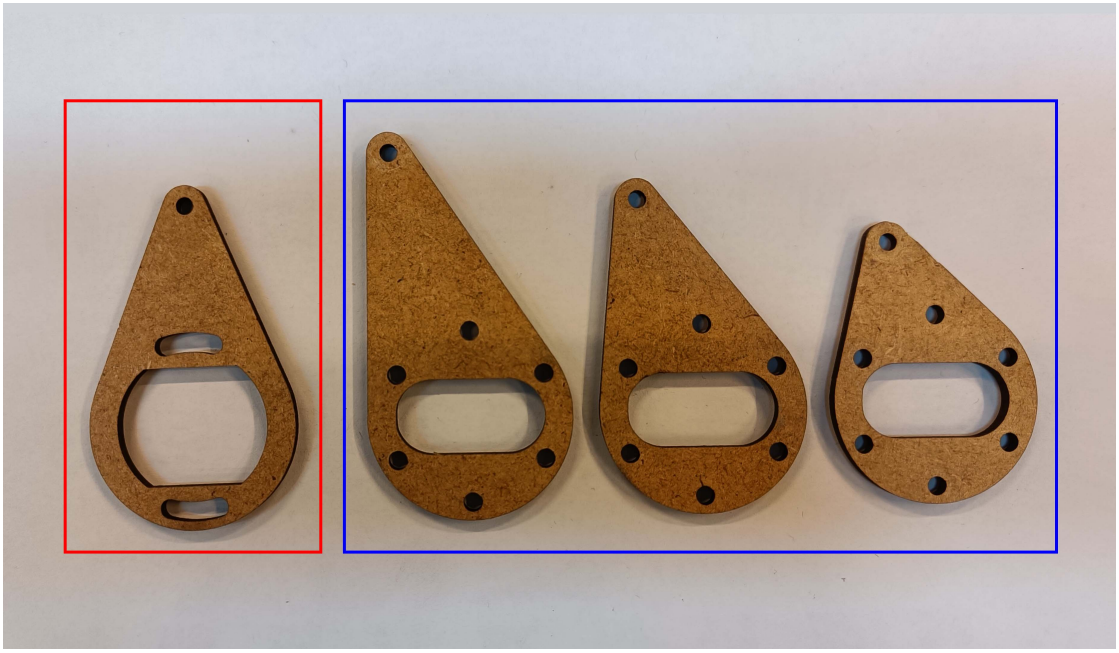
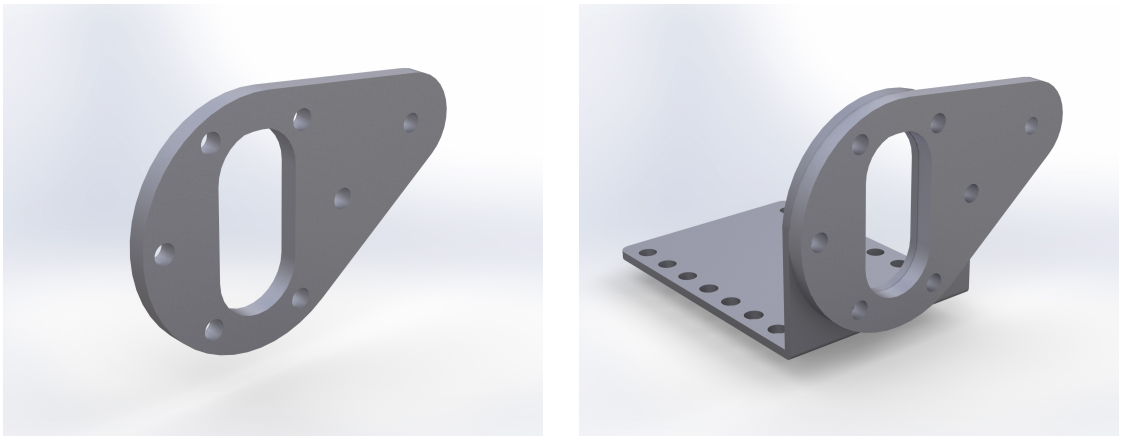


Figure 5.8: Different iterations of the suspension adapter. The red box indicates an angle-adjustable adapter, and the blue box indicates adapters with different lengths.

Once a proper angle and length were determined, the suspension adapters were fabricated in aluminum. The final iteration is shown in Figure 5.9.



(a) Suspension adapter.

(b) Suspension adapter mounted on the Pololu bracket.

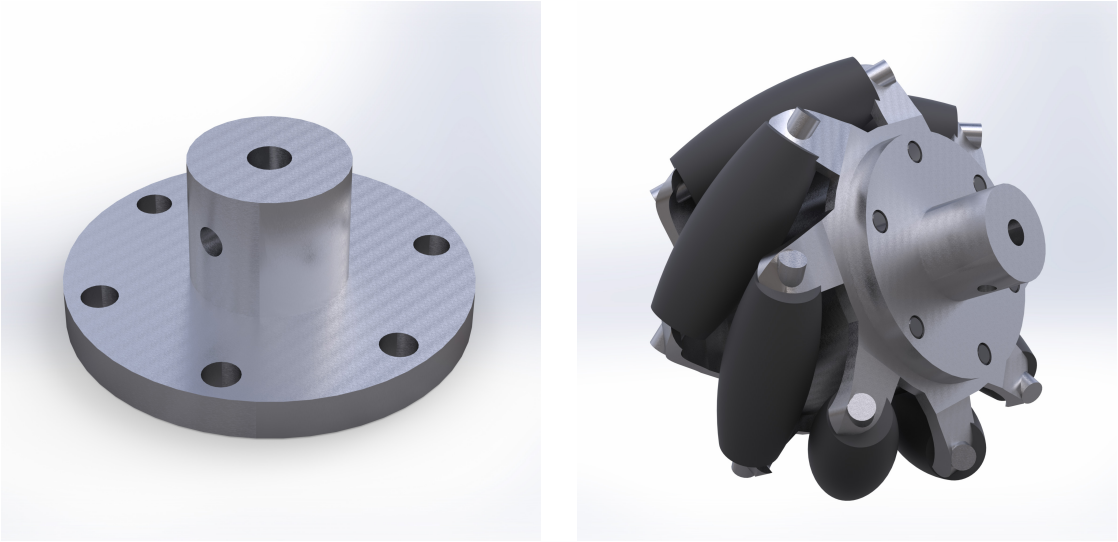
Figure 5.9: Final iteration of the suspension adapter used to mount the suspensions to the motors.

5.2.4 Mecanum Wheels

As decided in Chapter 2, the process robot should be omnidirectional by using Mecanum wheels. When selecting Mecanum wheels, it is important to consider durability, as the process robot can weigh up to 25 kg. Therefore, it was decided to use aluminum Mecanum wheels with a diameter of 100 mm, as shown in Figure 5.10b. The purchased Mecanum wheels each have nine roller wheels made of polypropylen (PP)+polyethylen (PE), two sides made of aluminum alloy, and a nylon spacer in between.

5.2.5 Screw Hub

The Mecanum wheels must be mounted on the shaft of the Pololu 37D motors, which can be done with a screw hub. At the time all the components were ordered, the screw hubs were out of stock. As an alternative, an exact copy was designed and then fabricated in the AAU Makerspace, as shown in Figure 5.10.

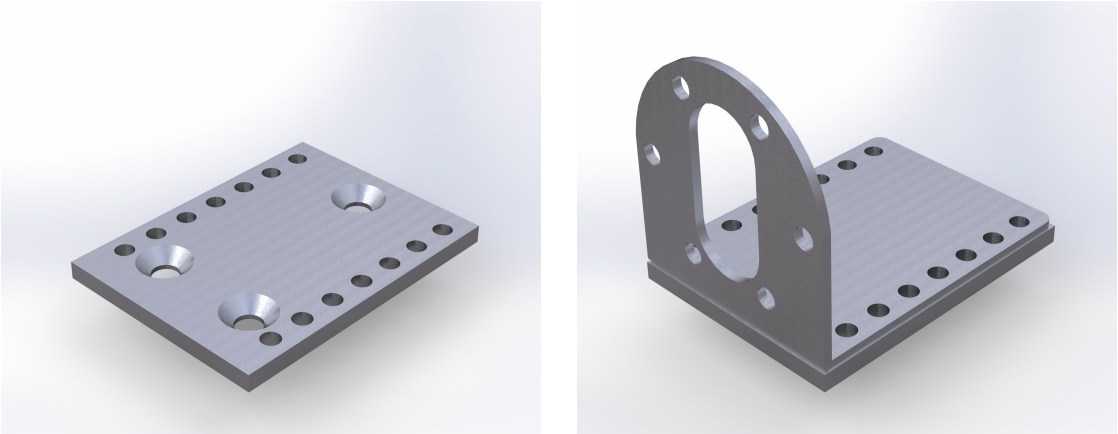


(a) Screw hub. **(b)** Screw hub mounted on a Mecanum wheel.

Figure 5.10: Screw hubs for mounting the Mecanum wheels to the motor shafts.

5.2.6 Motor Adapters

After receiving the motor adapters, it quickly became clear that the motor adapters could not be mounted on the aluminum frame. A solution to this problem was to design a motor adapter, as shown in Figure 5.11a. The motor adapter has three M4 countersunk holes that fit into the aluminum frame of the process robot and 14 M3 threaded holes that match the holes on the Pololu brackets, as shown in Figure 5.11b.



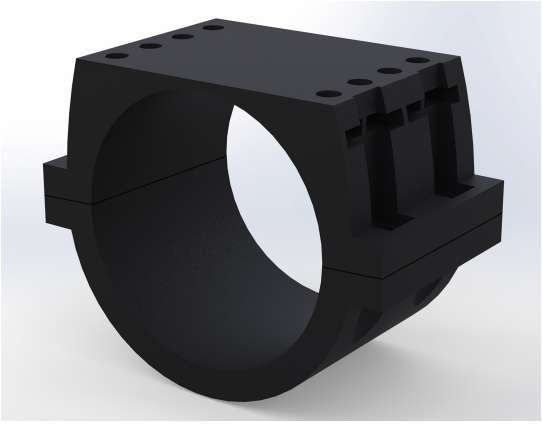
(a) Motor adapter. **(b)** Motor adapter mounted on the Pololu bracket.

Figure 5.11: Mount for the front wheels.

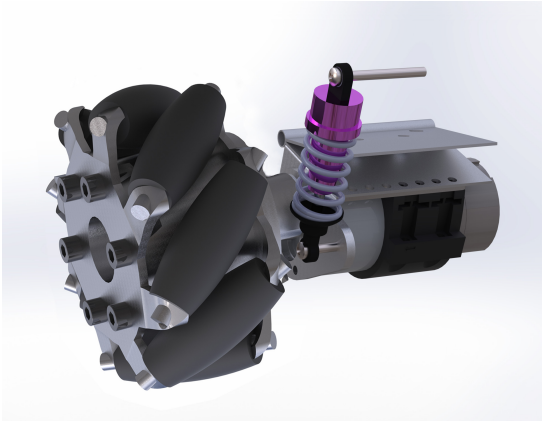
5.2.7 Motor Support Brackets

After mounting all the components and the process on the frame, it was clear that the Pololu motor brackets acquired were not strong enough, as they began to bend. Therefore, support brackets were designed to relieve them. Two different designs had to be made, one for the motors on the static front wheels and another for the motors on the suspended rear wheels.

The first set of support brackets was designed for the two rear motors, as shown in Figure 5.12. Due to the mobility of the rear motors, it was only possible to design support brackets that had to be mounted on the hinges themselves. The support brackets consist of two parts that wrap around the motor and are connected with two M3 nuts and screws on each side. Near the top of the support bracket, eight indents were made for square M3 nuts so that the entire support bracket can be mounted to the hinge using the eight holes on the top of the support bracket.



(a) Support bracket for the rear motors.



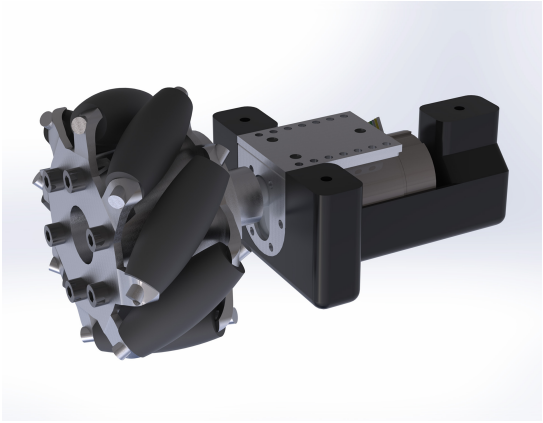
(b) Support bracket mounted on one of the suspended rear motors.

Figure 5.12: Support brackets that stop the suspended rear motors from bending.

The support brackets for the rear motors cannot be used for the front motors because the holes in the motor adapters in the front are threaded. Therefore, it is not possible to tighten the support brackets and the motor adapters together without destroying one of the threads.

The support brackets for the front motors consist of one solid part, as shown in Figure 5.13. They are mounted on the aluminum profiles below the electronics level. The shape is designed so that the support bracket encloses the Pololu brackets and the motor chassis and holds the motor in place by screwing it to the aluminum frame.

(a) Support bracket for the front motors.



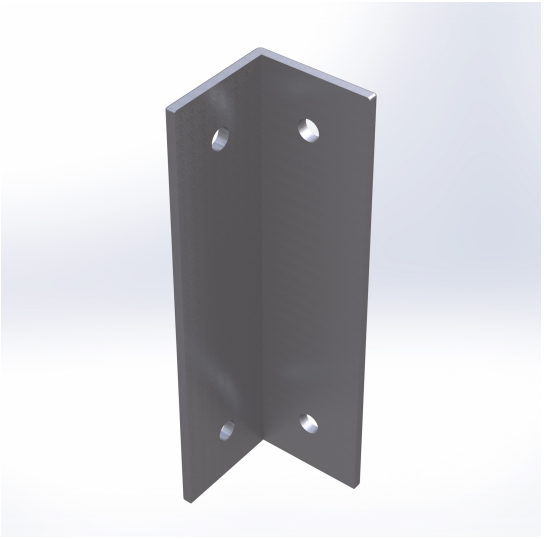
(b) Support bracket mounted on one of the static front motors.

Figure 5.13: Support brackets that stop the static front motors from bending.

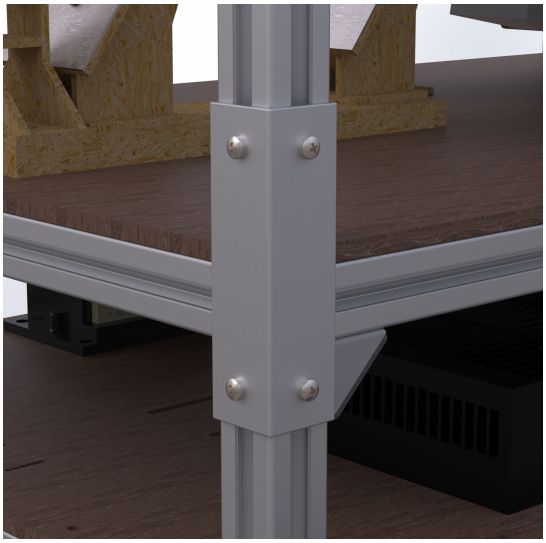
5.2.8 Edge Brackets

Edge brackets were designed to attach the vertical aluminum profiles that hold the tracking level with the ArUco marker to the frame, as shown in Figure 5.14a. The purpose of the edge brackets is primarily to secure the tracking level to the frame. However, they are also designed to allow personnel to easily replace the process. If the process needs to be replaced, only two screws need to be removed from each edge bracket to access the process.

Each edge bracket is attached with four M4 T-nuts, the lower two of which are attached to the electronics level and the upper two of which are attached to the process level, as shown in Figure 5.14b.



(a) Edge bracket.



(b) Edge bracket mounted on the frame.

Figure 5.14: Edge brackets holding the vertical aluminum profiles in place.

5.2.9 Mounting Plates

On each level of the aluminum frame of the process robot there is a mounting plate. As the name implies, these plates are used for mounting the components, processes and ArUco markers. As an example, Figure 5.15 shows the mounting plate with all the electrical components mounted on it. Most of the components were mounted using holes cut into the plate with a laser cutter. All mounting plates were made of medium density fiberboard (MDF), as this is cheaper than acrylic and allows for cheaper reconfiguration of components if needed.

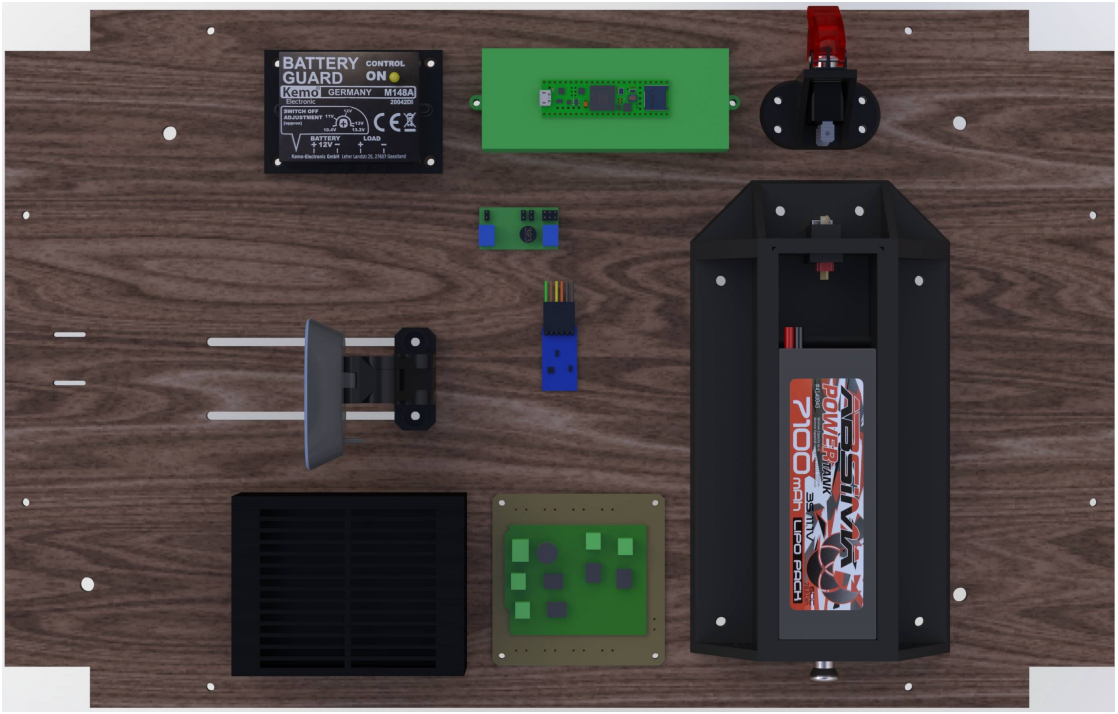


Figure 5.15: Mounting plate of the process robot with all the electrical components.

5.2.10 Battery Drawer

One thing Casper Schou mentioned is that it would be nice to have a quick change system for the battery. This would make it easier to change batteries between runtimes.

Therefore, a battery drawer system was designed and 3D printed, consisting of a battery drawer with a knob, two side brackets with guide rails, and a stop bracket to which the male and the female battery connectors are connected, as shown in Figure 5.16.

Figure 5.16: The blue box indicates the battery drawer, and the green box indicates the connector.

A steel plate was attached to the end of the battery drawer (see Figure 5.17b), and a slot for a neodymium magnet was made in the stop bracket (see Figure 5.17a). This prevents the battery drawer from sliding out during runtime.

(a) The orange box indicates the magnet on the stop bracket, and the green box indicates the connector.

(b) The orange box indicates the steel plate on the battery drawer, and the green box indicates the connector.

Figure 5.17: Battery drawer connector system with magnet.

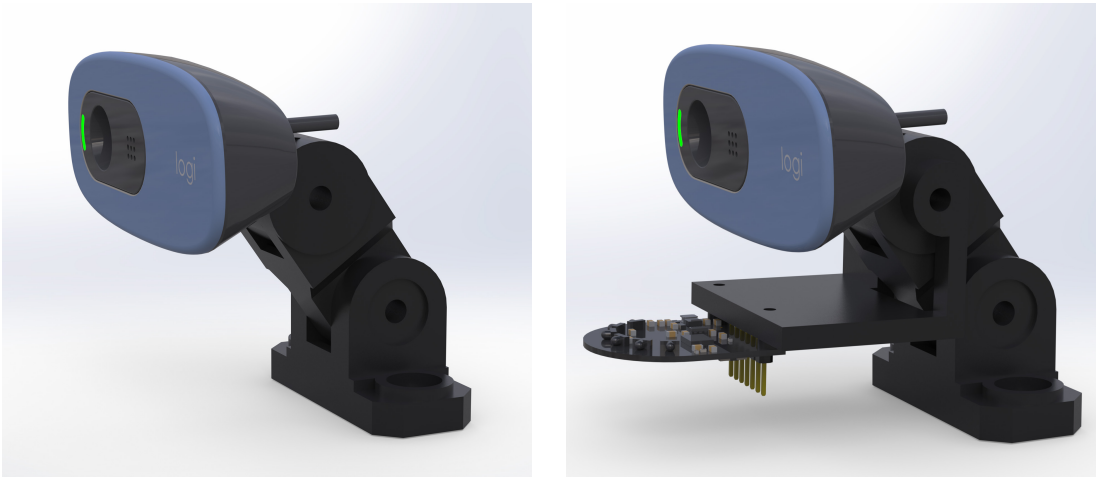
5.2.11 Mounts for IR Sensor, Camera and Toggle Switch

For the IR sensor and camera, it had not yet been decided at what height and distance they would be mounted. Therefore, the mounts were designed to be flexible in both height and distance.

The first version was a combined mount for the IR sensor and camera, as shown in Figure 5.18b. It consisted of three movable joints for the camera, with the IR sensor attached to the second joint. The mount was then mounted on a slider cut out in the mounting plate at the electronics level of the frame, as shown in Figure 5.19. In this way, the IR sensor and camera could be moved back and forth while changing the height, allowing various configurations to be tested.

In some tests, it was found that the IR sensor needed to be moved farther forward and the camera farther back. This resulted in splitting the design, only keeping the

articulated mount for the camera, as shown in Figure 5.18a. The IR sensor was relocated to a slider that was cut out further at the front, as shown in Figure 5.19.



(a) Mount for camera.

(b) Mount for camera and IR sensor.

Figure 5.18: Articulated mounts.

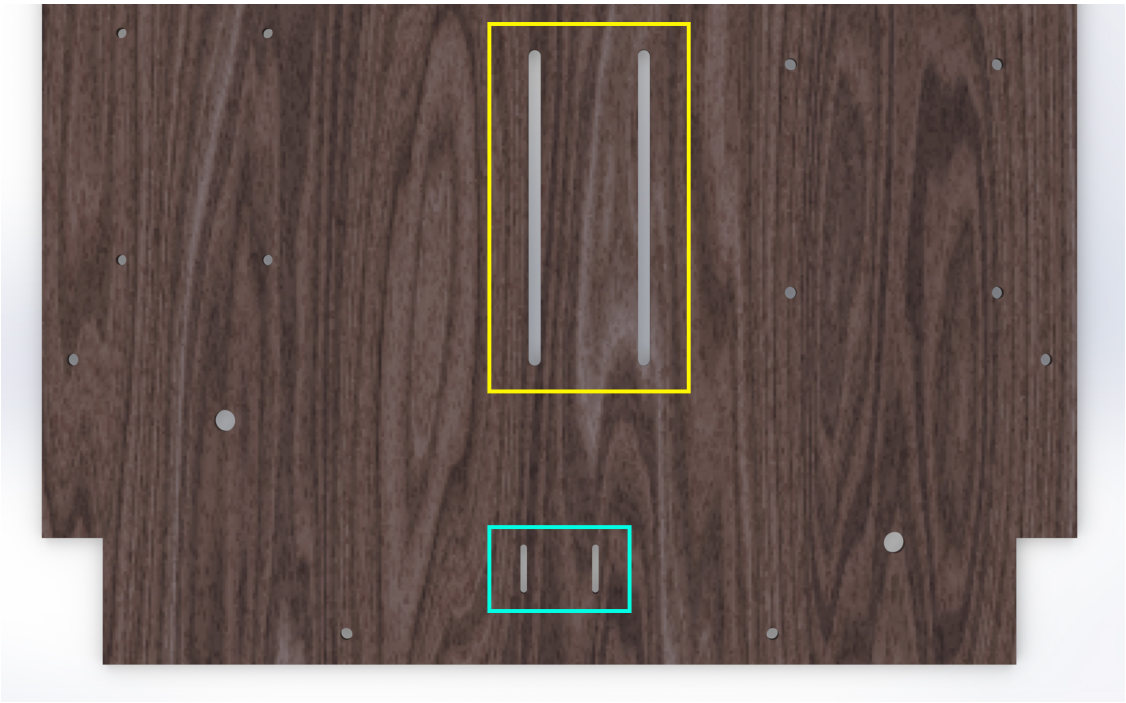


Figure 5.19: Mounting plate of the electronics level. The yellow box indicates the slider for the articulated camera mount, and the green box indicates the slider for the IR sensor.

With the new slider, it became clear that the IR sensor would occupy a large portion of the FOV of the camera. A solution to this problem was to attach the IR sensor to the bottom of the aluminum profile, as shown in Figure 5.20.

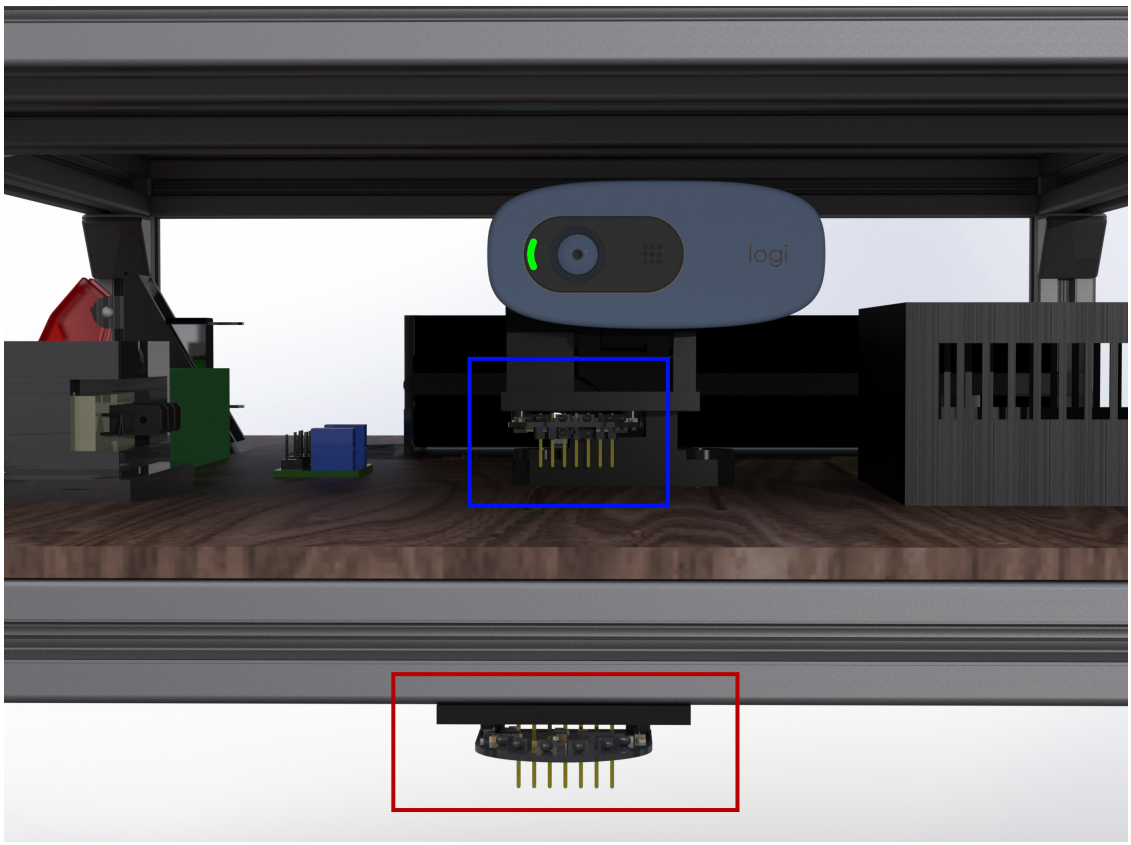
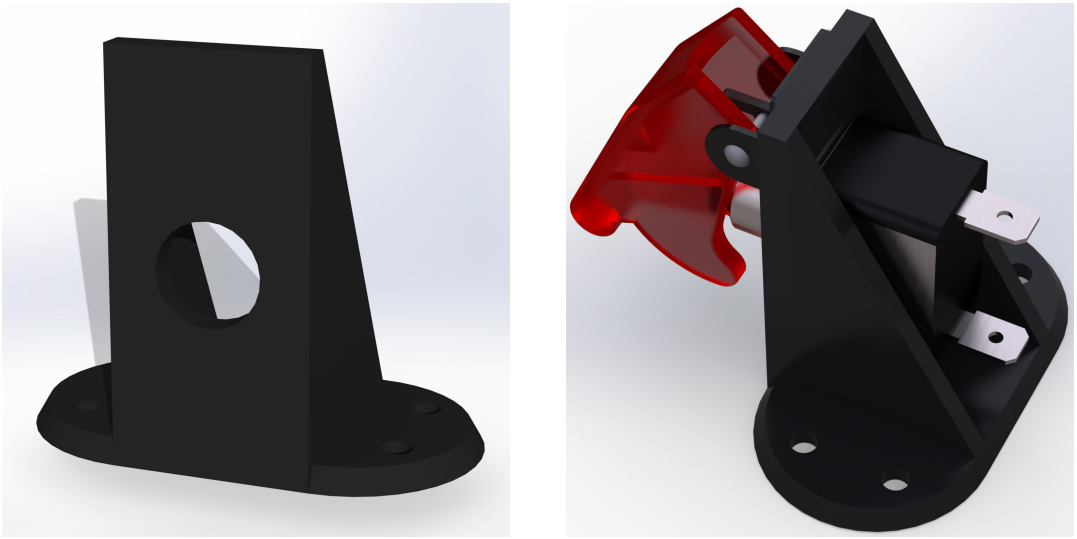


Figure 5.20: Two possible positions for the IR sensor. The red box indicates the IR sensor mounted below the frame, and the blue box indicates the IR sensor mounted on the articulated camera mount.

In addition, a bracket was designed for the toggle switch, as shown in Figure 5.21.



(a) Toggle switch bracket.

(b) Toggle switch mounted mounted on the bracket.

Figure 5.21: Bracket for vertical mounting of the toggle switch.

5.3 MDF Robot Prototype

While waiting for the frame parts to be delivered, a temporary prototype was built on a MDF board (see Figure 5.22) to test the hardware that had already been delivered. This allowed the implementation of software and firmware to begin and be tested before the final process robot was built.

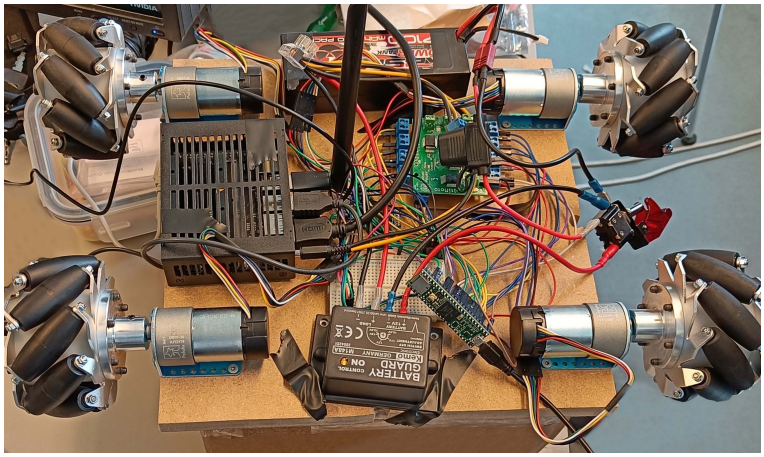
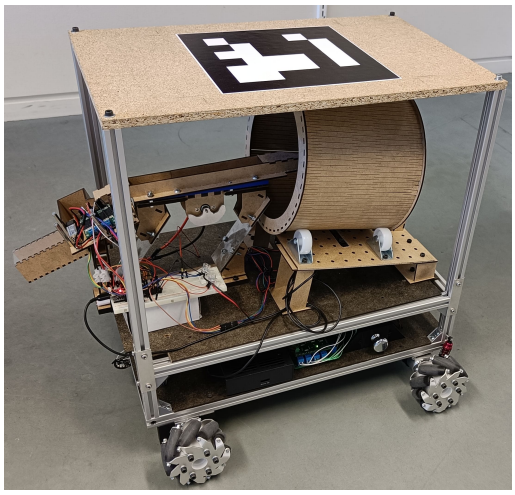


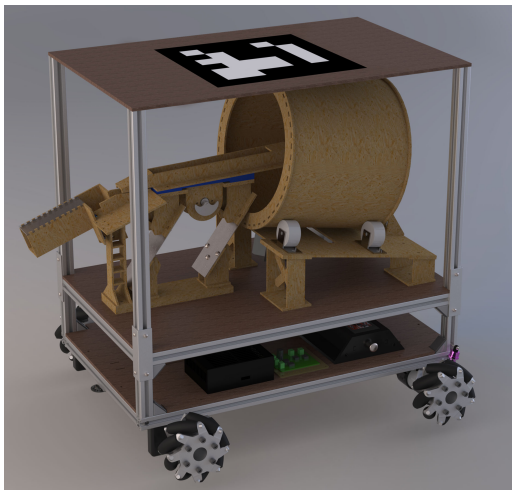
Figure 5.22: Image of the MDF robot prototype.

5.4 Aluminum Robot Prototype

When the frame parts arrived, the final prototype could be assembled (see Figure 5.23a) according to the design (see Figure 5.23b). Due to time constraints, only one of the four process robots were build. The last three will although still be build, just at a later time.



(a) The final prototype in its complete state with the brick feeder process mounted on it.



(b) The rendered version of the final prototype, including the brick feeder process mounted on it.

Figure 5.23: Final prototype and a rendered version of the final prototype.

5.5 Brick Feeder

The brick feeder is the process mounted on the process robot. Since the process is intended to be removable, it is important that the electrical design for the brick feeder is as independent as possible.

However, in the original electrical design, the tasks were split between an Arduino Uno and an Arduino Nano microcontroller, with the Arduino Uno controlling the solenoid and vibration motor and the Arduino Nano controlling the stepper motor. The two microcontrollers did not communicate with each other, so the drum would always spin even if no bricks are dispensed. As for input, the first microcontroller would wait for a serial input in the form of an integer representing the number of bricks dispensed. This makes the brick feeder dependent on the process robot to activate it [25].

In the redesign, the brick feeder will be controlled only by a single but more powerful microcontroller with built-in Wi-Fi. This will make it possible to activate the brick feeder from any device connected to the same network. In addition, the components will be activated only when needed to conserve battery power. It would be best if the brick feeder had its own battery, but currently it is uncertain if this fits the design, so it will share the battery with the process robot, as shown in Figure 5.24.

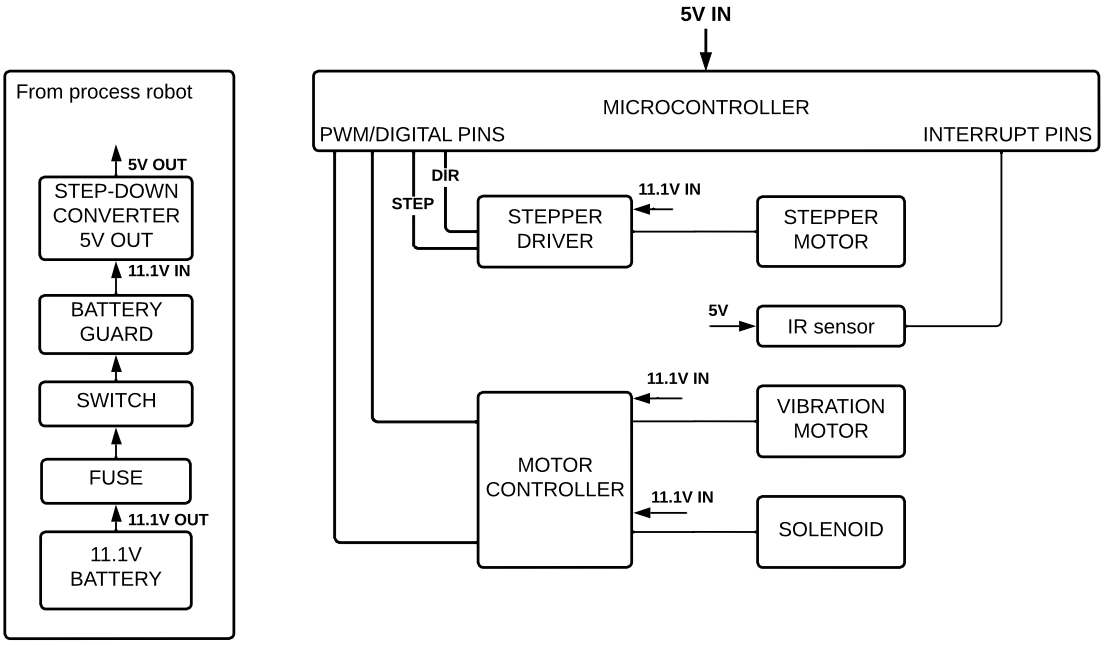
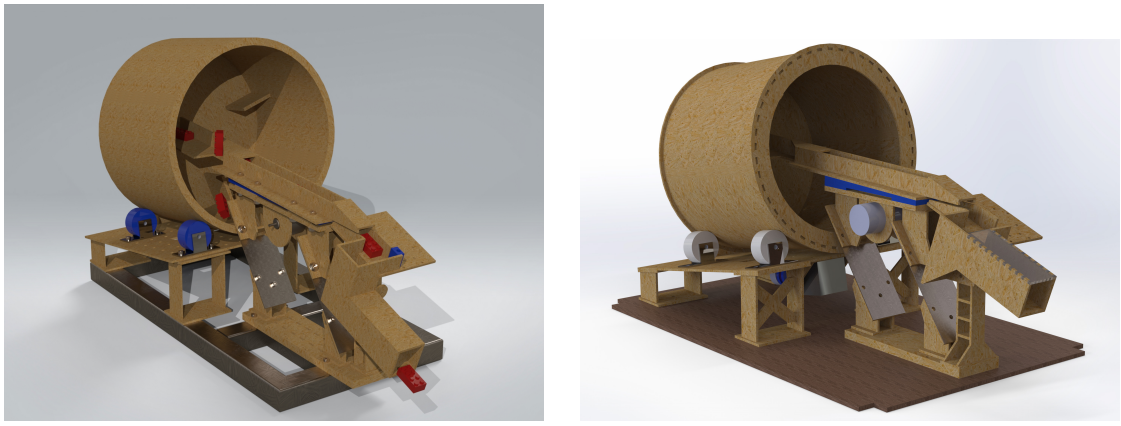


Figure 5.24: LEGO brick feeder basic circuit design.

5.5.1 Reworked Design

The brick feeder was fully functional when it was provided for this project, but several design changes were made to improve its robustness, compactness, functionality, and general characteristics in favor of the robot design. As a result, design changes were made to improve its robustness, compactness, functionality, and general characteristics,

which benefited the robot design. A comparison between the old and the new design is shown in Figure 5.25.



(a) Old design [25].

(b) New design.

Figure 5.25: Old design compared to the new design of the LEGO brick feeder.

After some time, the counter on the foundation slowly began to break off, and the structure that holds the barrel began to warp. Therefore, the general structure in this project was reworked to achieve a more robust design. For example, the bottom of the counter was made trapezoidal rather than straight to increase the contact area (can be found in Appendix E.1). In addition, the structure holding the barrel was provided with cross stiffeners in the legs (can be found in Appendix E.2).

All holes were relocated and the lane for the LEGO bricks was shortened, making the structure 40 mm shorter without losing its functionality (can be found in Figure E.3). A more compact design was desired, otherwise it would become a long and slender robot design that would also make it unnecessarily long.

The chute for the LEGO bricks was also placed unnecessarily far down, wasting some of the potential energy for the LEGO bricks. This also meant that the robot had to be higher than necessary to dispense the LEGO bricks at a certain height. This was also reworked so that the chute was placed further up the counter (can be found in Appendix E.4). This increases the feeding height by 45 mm compared to the old design. Another change to the chute could be to retract it when not in use to avoid possible damage when the process robot is moving. However, as this is outside the scope of this project, this option will not be explored.

Finally, the motor with a shifting rotating weight meant that the direction of vibration was not linear. This meant that the lane for the LEGO bricks also vibrated from side to side, rather than just back and forth. This often caused the LEGO bricks to fall back into the barrel before they reached the lane. The problem was solved by tilting the lane 2° toward the wall so that the LEGO bricks had to fight gravity to fall over the edge (can be found in Appendix E.5). A better solution would have been to replace the motor with an electromagnetic motor that affected the vibrating motion in only one direction.

5.5.2 Communication Design

As mentioned before, the brick feeder should be as independent as possible, which also applies to communication. The brick feeder should not depend on the process robot sending messages on its behalf or receiving messages from the server. By using a microcontroller with built-in Wi-Fi, it is possible to implement micro-ROS in the firmware and then establish a direct wireless connection between the process and the main computer. The process can therefore be easily replaced by another process or transferred to another robot, or even to a stationary workstation.

An action server would be more convenient to implement, but this is not yet available in micro-ROS, so instead a subscriber for receiving the goal and a publisher for sending status messages will be used. The brick feeder will then wait until it receives a goal on the topic */brickfeeder_goal* in the form of the number of bricks it should dispense. It will then start publishing status messages on the topic */brickfeeder_status* about how long it is in the process until it has dispensed all the bricks.

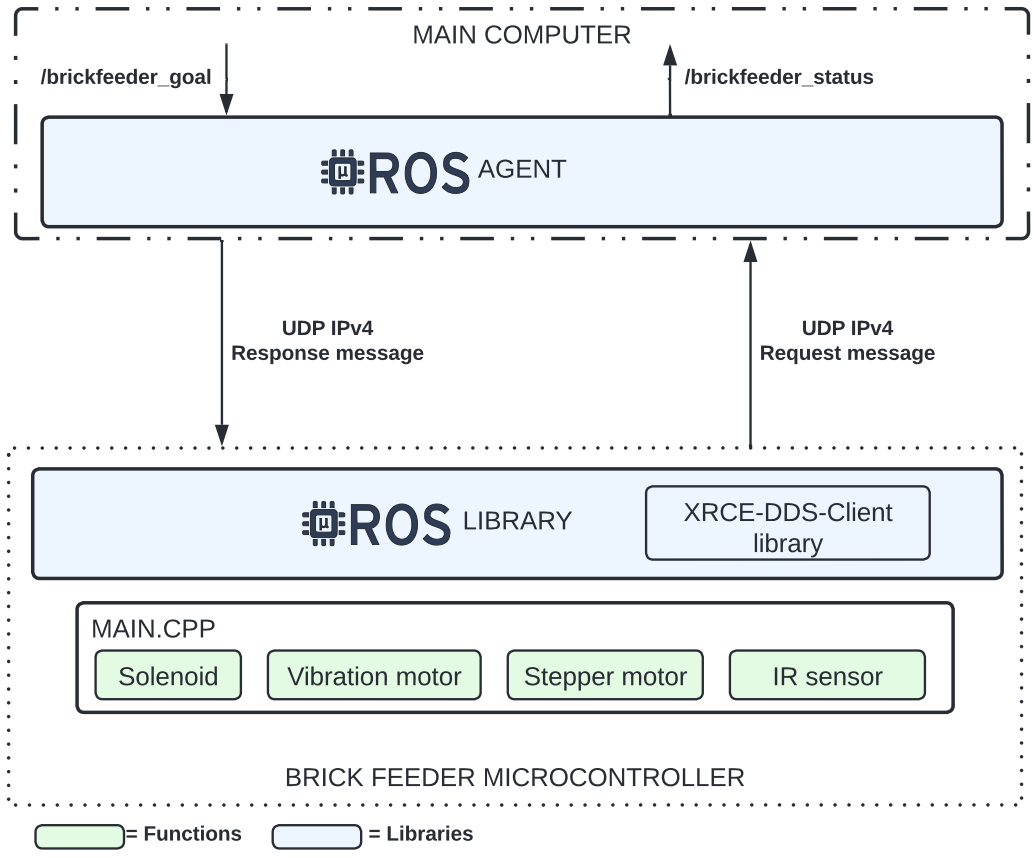


Figure 5.26: Communication design between the ESP32 on the brick feeder and the server.

5.5.3 Firmware Design

The LEGO brick feeder has a total of four components that it must control: a stepper motor to rotate the drum, a vibration motor to move the LEGO bricks, an infrared sensor to detect the LEGO bricks, and a solenoid to dispense LEGO bricks. It also needs to constantly wait for goals on the */brickfeeder_goal* topic and publish its status to the */brickfeeder_status* topic.

Functions must be implemented for each component, the publisher and the subscriber (see Figure 5.26), noting that each component must operate in real time. This means that normal delay functions cannot be used, as this would pause the entire program and risk not detecting or acting upon new goals in time. The solenoid is activated when the IR sensor detects a LEGO brick. So these two components can be controlled by the same function, but the two motors, the publisher and the subscriber should have their own functions. The functions are illustrated in Figure 5.27.

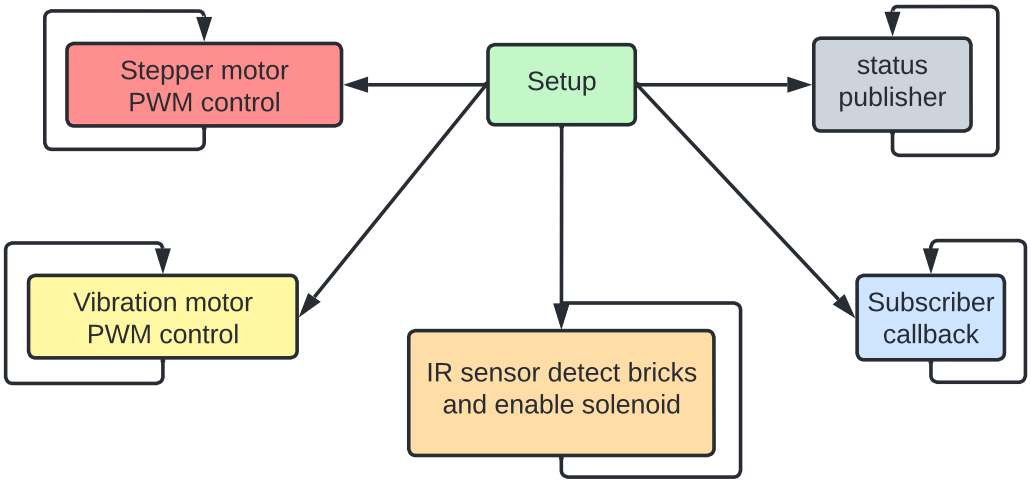


Figure 5.27: Overview of the functions implemented in the firmware for the brick feeder.

5.6 Tracking System

Figure 5.28 shows the setup of the test room. It allows the robot to know its relative position on a map. The tracking system is implemented in Python 3 and runs on the main computer.

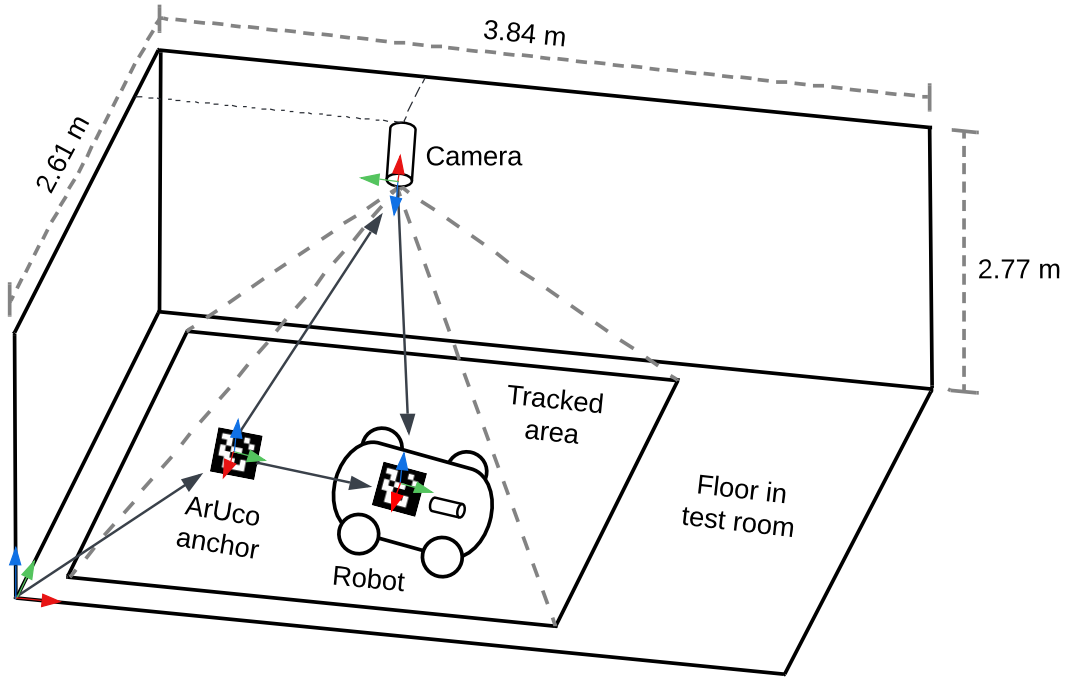


Figure 5.28: Image of the prototype setup and the room in which it is tested.

Figure 5.28 shows how a camera tracks a robot in a test room. The camera is mounted on the ceiling and looks down while an AruCo marker is placed on the floor and another on the process robot. The markers are in the FOV of the camera so that their pose relative to the room can be determined. This is the setup used to test the entire system; the test room has a size of 2.61 m $(L)\times$ 3.84 m $(W)\times$ 2.77 m (H). The tracking system consists of a Logitech C922 HD Pro webcam connected to the USB port of the main computer via a serial port. A 5 m USB extension cable is used to connect it to the computer, as shown in Figure 5.29.

Figure 5.29: Image of the finished process robot prototype. The cyan box indicates the camera mounted on the ceiling, the green box indicates the process robot, and the red box indicates the ArUco anchor.

The ArUco anchor is placed on the ground to align the virtual environment with the real environment. The anchor is used as a reference point to align the two environments

and tell the system how the tracked area is oriented and positioned compared to the real space. This is explained in more detail in Subsubsection 5.6.

The implementation of the tracking system of the robot can be divided into five parts, as shown in Figure 5.30.

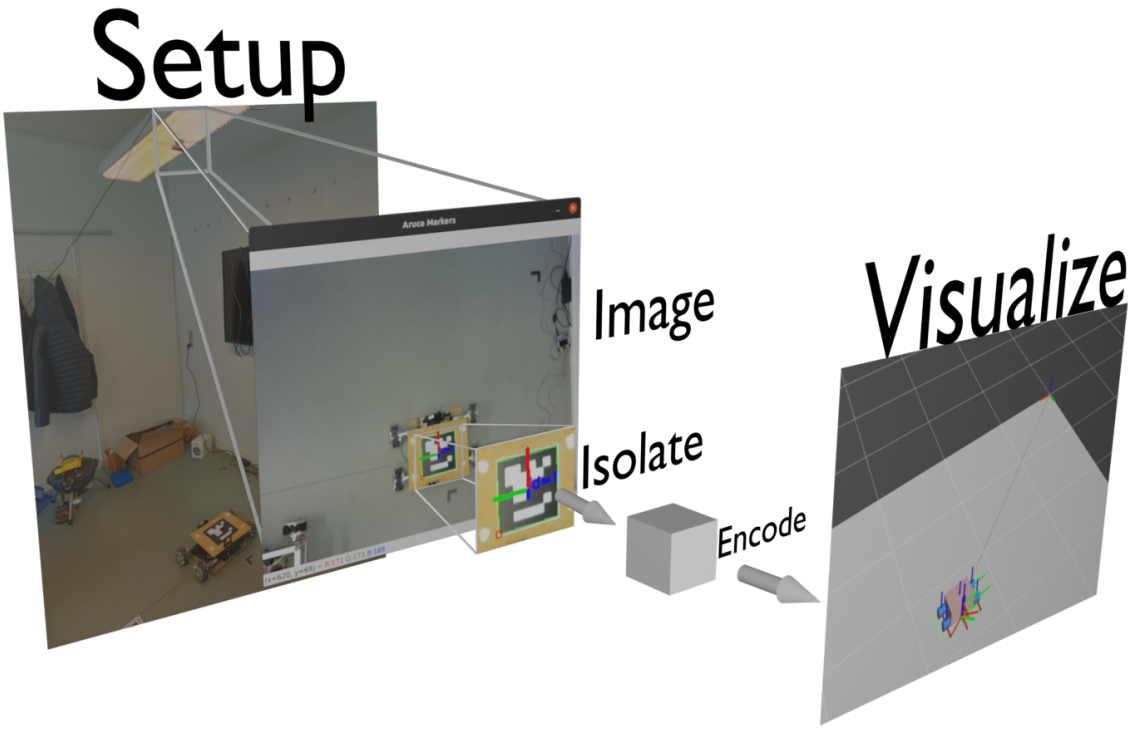


Figure 5.30: The different stages of the ArUco tracking System system presented with the MDF robot. The same system is used for the final prototype of the process robot.

1. The physical *setup* with the camera at the top and the robot at the bottom.
2. The *image* captured by the camera.
3. The software *isolates* the ArUco marker and performs a perspective transform.
4. The pose of the marker being *encoded* and sent to a */pose* topic.
5. The *visualization* of the robot and its pose in real space.

The Physical Setup

The tracking system is set up in a test room. The room cannot be used to its full size because the chairs and tables take up half of the room. This led to some limitations in the tests, e.g., the size of the tracked areas and the navigation goals cannot be further than a maximum of 3.84 m apart. This is sufficient to test the implementation of localization and navigation on the process robot with one camera and one process robot.

When focusing on the camera, the horizontal FOV is 68°. Based on the horizontal FOV, the horizontal length of the tracked area is 3.4 m on the floor. This length is measured without considering the height of the ArUco marker on the process robot. The height

of the ArUco marker above the floor is important because the closer the marker is to the camera, the smaller the area tracked. This is because the diagonal FOV remains the same, but not the distance between the marker and the camera. The measured vertical FOV is 43°, resulting in a vertical length of the tracked area of 2 m. This leaves 0.77 m that is not tracked. This means that 0.39 more Logitech C922 cameras are needed, or rounded up, two cameras that overlap so that the test area is fully tracked. Due to time constraints, the implementation of two cameras will not be investigated further and the system will therefore be built with only one camera. However, if further testing is done, the test room could be a possible room for implementing two or more cameras in the system. The formula for calculating the FOV is as follows [70, p.-180]:

$$FOV = 2 \times tan^{-1}\left(\frac{\frac{Y}{2}}{x}\right) \tag{5.3}$$

Where:

- X is the distance between the floor and camera.
- Y is the length of floor seen in horizontal or vertical direction by the FOV of the camera.

To estimate the distance required between the floor and the camera to cover a given area, the tangent formula can be used [70, p.-180]:

$$\begin{aligned} \text{Adjacent} &= \frac{\text{Opposite}}{tan(\theta)} \\ &\Downarrow \\ X &= \frac{\frac{Y}{2}}{tan\left(\frac{FOV}{2}\right)} \end{aligned} \tag{5.4}$$

For example, to cover the test space of 3.84 m (W) with a diagonal FOV of 68 degrees, without considering the distance between the ArUco marker and the floor, a camera height of 2.85 m would be required:

$$X = \frac{\frac{3.84}{2}}{tan\left(\frac{68}{2}\right)} = 2.85\,\text{m}$$

For future testing, it is possible to determine the height, required FOV of a camera, and number of cameras needed to fully cover a test area using Equation 5.3 and 5.4.

Image Capturing

The process of capturing images includes reading the image stream from the camera and post-processing the images. With the OpenCV function *VideoStream* it is possible

to get video streams from the camera. The function takes an argument "ID" which tells the function which camera to read from. This function also allows OpenCV2 to read from multiple cameras simultaneously, allowing a multi-camera setup for future improvements. In this report, an ID of "0" is used as this is the default camera on the system running the tracking software.

Once the stream is started, it is possible to extract images from the camera. The FPS of the tracking system depends on the camera used and the hardware capabilities of the computer or laptop running the software. The FPS of the software is important because it determines how often the pose of the robot can be updated. The camera must be able to capture up to 30 FPS to avoid excessive motion blur from moving ArUco markers on the robots. The FPS of the Logitech C922 depends on the resolution selected. For example, at 1080p resolution, it provides 30 FPS. If a higher FPS is needed to reduce motion blur, then OpenCV2 can be used to reduce the resolution to 720p HD and increase the FPS to 60.

The Jetson Nano was unable to process the 30 FPS when the resolution was set to 1080p while tracking the ArUco markers. The computations were slow, averaging 10 FPS. Therefore, the resolution was reduced to 720p HD to increase the FPS and the update rate of the tracking system.

Camera calibration
As described in Subsubsection 4.4, camera calibration is important to obtain undistorted images. OpenCV makes implementing camera calibration fairly trivial, since it provides functions based on calibrating a camera with the checkerboard pattern, as shown in Figure 5.31. The calibration function *cv2.calibrateCamera* returns a camera matrix that compensates for lens distortions and distortion coefficients to compensate for tangential distortions. To execute this function, the image must be converted to grayscale, and the corners of the checkerboard must be found with the function *cv2.findChessboardCorners*. The function to find the corners needs the grayscale image as a parameter as well as the number of corners the function should search for on the checkerboard. There are different sizes for the checkerboard, for example an 8×8 grid or 5×5. For this setup, a checkerboard with a grid size of 9×6 was used, as shown in Figure D.5.

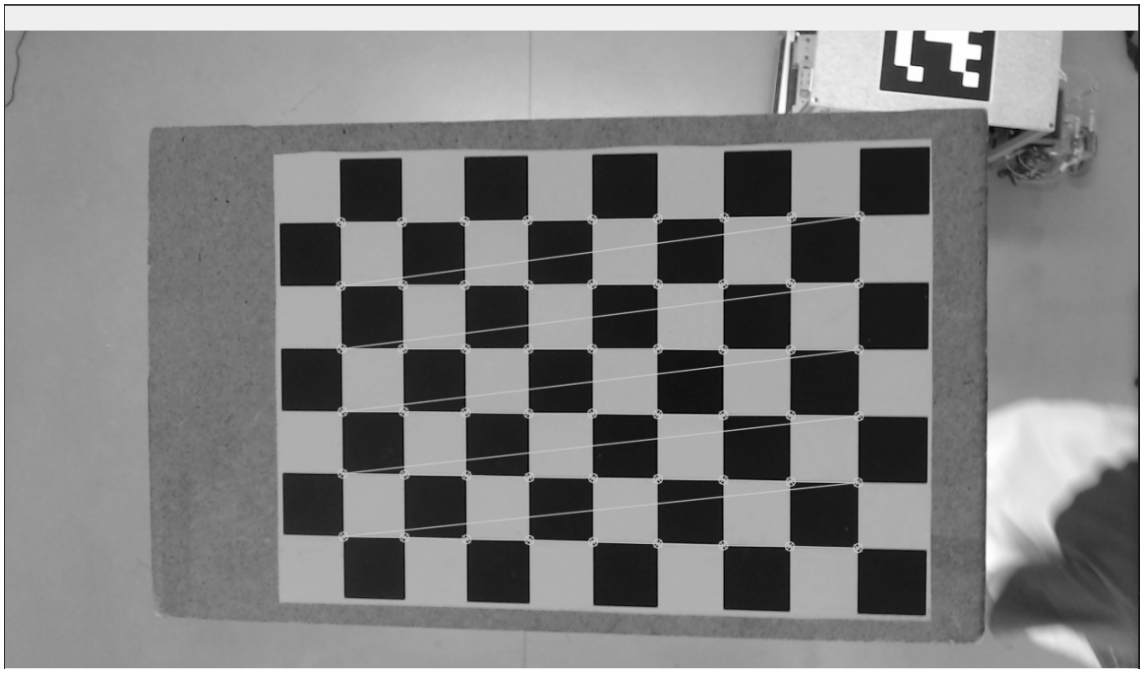


Figure 5.31: Camera calibration of one of the cameras using OpenCV cv2.findChessboardCorners(). The corners are marked as points on the image with lines connecting them.

When running the tracking system, a variable can be changed to switch between tracking the robot and calibrating the camera. When camera calibration is enabled, a window opens showing a live feed from the camera while the camera waits for input. The user can then press "a" on their keyboard, which will capture the current image and append it to a list of images captured during calibration. The corners are found on all images in the list, and then the camera calibration is performed on all images in the list. The result is the distortion coefficients and the camera matrix, which are stored in a yet another markup language (YAML) file for later use in the tracking system and the docking procedure.

All cameras have different coefficients and camera matrices, even if they are the same brand and model. This is because not every camera is made exactly the same and therefore requires its own distortion coefficients and camera matrices. So if a calibrated camera is replaced by another that is not calibrated, and the same YAML calibration file is used for that camera, the projected plane will no longer match the real world. So if the tracking System camera is replaced by a new camera, the new camera has to be calibrated with new parameters. This also applies to the docking procedure, since each robot also has a camera.

Isolation of the ArUco Marker

Once the camera has been calibrated and the parameters have been saved in the calibration file, the tracking system loads the contents of the file into memory before attempting to find markers. This file is then used when the ArUco marker transform needs to be found to ensure that the pose of the markers is correct and without distortion.

With the function *detectMarkers* it is possible to detect the ID and the corresponding corners of each marker. The function takes a grayscale image, an ArUco dictionary (the dictionary is the inner binary matrix codification, see Appendix D.2) and then some detection parameters set by OpenCV2 as inputs. Once these parameters are set and the function is executed, the ArUco markers are isolated and then perspective transformed to preserve the ID as well as the corners. This only happens if the markers are detected in the image. If so, draw their boundaries in an image to show them. Then the function *estimatePoseSingleMarker* is executed. This function uses the markers and their size in meters, as well as the camera calibration parameters, to return the pose of the markers relative to the camera as rotation and translation vectors. The code used for the tracking system can be found in Section 5.6.

The ID capabilities of the ArUco markers have made it possible to implement a system that can distinguish between the process robot and the ArUco anchor. The pose of the process robot and the ArUco anchor can then be used separately in the system while they are in the FOV of the camera. The pose of the process robot is used for the tracking system and the pose of the ArUco anchor is used for the alignment. Now that the pose of the markers have been found and visualized for the user on a 2D image, the information can be encoded for ROS 2.

Encoding for Pose Topic

The localization system will use the data received from the tracking system. Therefore, the data will be published as messages via a ROS 2 topic. ROS 2 uses a message description language to describe the data values (messages). The message type expected by the Kalman filter is positional data. A ROS message type used in this prototype is the *PoseWithCovariance* message type. Therefore, the 2D points projected onto an image by the ArUco markers must be converted to positional and orientational data and then to the *PoseWithCovariance* message type to be sent over the ROS 2 network.

The *PoseWithCovariance* message type provides the filter with a covariance matrix that tells the filter how to weight the positional and orientational data. A *PoseWithCovariance* type requires a header and a pose derived as geometry_msgs. The header describes the timestamp of the frame and its parent frame, while the pose describes a position and a rotation in space. The position is specified as a point $[x, y, z]$, while the rotation is specified as a quaternion $[w, x, y, z]$. This message can then published to the */pose* topic which can then be used by the localization system for navigation.

Visualization of the Robot in Real Space

Visualization of the ArUco markers can be in 3D or 2D, and provides feedback to the user about what is happening within the tracking system. The user is presented with a 2D image stream. It shows what the camera is capturing and how the pose of the ArUco markers is translated from the real world to the virtual world, as shown in Figure 5.32. A 3D representation is also provided by RViz 2, which is a visualization tool used with ROS 2. With this tool it is possible to view the pose of the ArUco markers in three dimensions. This makes it possible to see if the measurements in the real world match

those in virtual world.

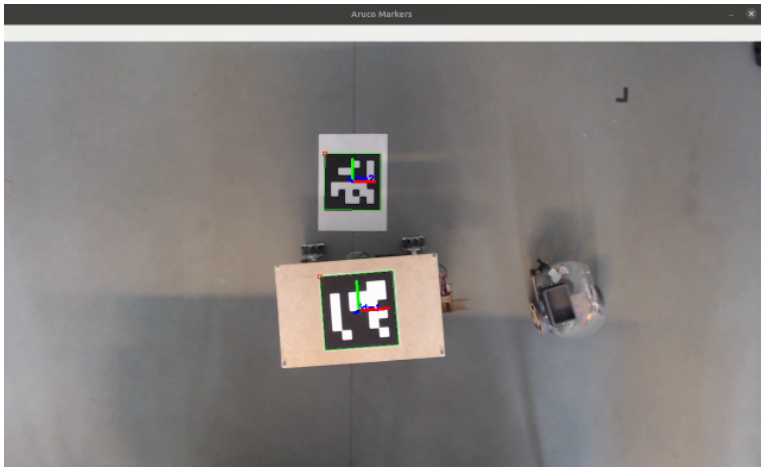


Figure 5.32: Image of two ArUco markers being tracked and their pose visualized in 2D using OpenCV2. The ID is also displayed.

Implementation of Odometry Reset

Once the tracking system is started, it searches for the ArUco anchor ID. Once the marker is found, twenty samples of the transform between it and the camera (`ArUco anchor` → `camera`) are taken and then the average is calculated. This is done to minimize errors during alignment. If only one sample were taken and used for alignment, noise could cause the alignment to be inaccurate.

Once the average is calculated, it is used to subtract the `ArUco anchor` → `camera` transform from the `base_link` → `camera` transform. In this way, the `ArUco anchor` → `base_link` is obtained as described in Section 4.5.2. The `ArUco anchor` → `base_link` transform is then converted to a *PoseWithCovariance* message that is published to the topic */pose*. These steps is shown in Figure 5.33.

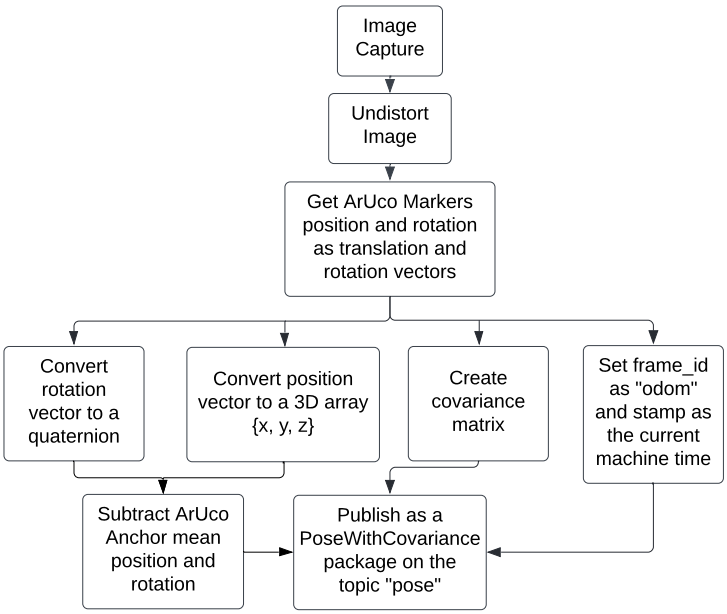


Figure 5.33: How a *PoseWithCovariance* message is created.

5.7 Teensy 4.1 Firmware

As mentioned in Section 4.2.1, the decision was made to use an existing but generic firmware for mobile robots that is based on micro-ROS, namely the linorobot2 firmware repository [55].

A fork of this repository was created for further configuration and development. The forked repository can be found in Appendix A.3, and is what the rest of this section will be referring to.

5.7.1 Setup of Configuration File

When the forked repository is cloned for the first time, the firmware must be configured to fit the electrical components and movement strategy that one have chosen. The firmware consists of three sub-configuration files (robot base type, motor driver type, and IMU type) contained in a larger configuration file called *lino_base_config.h*, as shown in Figure 5.34. In this file, it is possible to configure general motor and encoder specifications, such as the maximum RPM, the number of counts per revolution for each motor, the wheel diameter, the motor encoder pins, the motor PWM pins, and the motor direction pins. All the configurations can be found in Appendix A.3 and the pin setup of the electrical components can be found in Appendix F.1.

Figure 5.34: Illustration of how the *lino_base_config.h* file branches to several other configuration files, which then contain the associated libraries.

5.7.2 Firmware Main Code

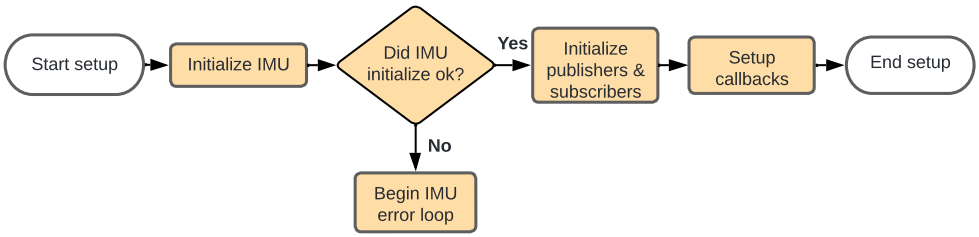
The linorobot2 firmware uses the Arduino framework and therefore consists of a *setup* function and a *loop* function.

The setup function is used to initialize the publishers, subscribers, and the selected IMU, while also setting up the associated callback functions and connecting to the agent, as shown in Figure 5.35a. During setup, the code can end up in three states: successful initialization, error loop due to a failed IMU initialization, and error loop due to a failed micro-ROS API call.

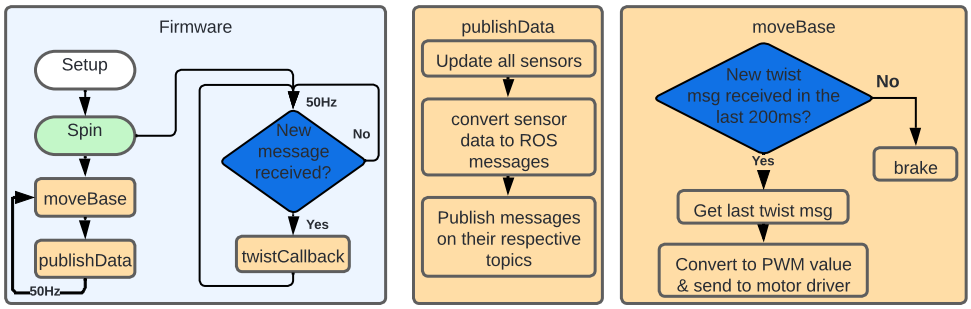
When the setup reaches the successful initialization state, it moves to the *loop* function. Here, the micro-ROS spin function is called at a rate of 50 Hz, which handles the subscribers, publishers, and callback functions that were previously set up.

There are a total of two publishers (IMU and odometry) and one subscriber (twist_msg). The two publishers are handled by a function called *publishData* (see Figure 5.35b), which is set up as a callback function for a timer with a frequency of 50 Hz. This means that every 20 ms the function updates all sensor values, converts them into ROS messages and publishes them in the corresponding topics.

The subscriber is handled by the callback function *moveBase* (see Figure 5.35b), which uses the received *twist_msg* to compute the desired PWM and then sends it to the motor driver.



(a) Setup function that initializes the IMU, publishers, subscribers and sets up the callback functions.



(b) The main code of the linorobot2 firmware (simplified to show the most important functions).

Figure 5.35: Layout of the main code of the firmware.

5.7.3 Additional Functionalities

After setting up the *lino_base_config.h* file, the firmware should be ready to be compiled and uploaded to the Teensy 4.1.

However, in this case, the following components are not among the standard components supported by the linorobot2 firmware:

- Multimoto motor controller.
- 3-channel IR sensor.

Therefore, the corresponding libraries for these two components must be implemented.

In addition, a function to reset the Teensy via the serial interface has been implemented. The reason for this is that whenever the code enters an error loop, a physical reboot of the Teensy is required, which makes troubleshooting faster.

Finally, the firmware code tracks the pose of the robot, but due to wheel slippage, this pose becomes more inaccurate over time. A function was then implemented to correct this using data from the tracking system.

Implementation of Motor Controller Setup via SPI Interface
The purchased Multimoto motor controller contains four L9958 motor drivers that must be configured via SPI during setup. The Multimoto motor controller also needs a pin to enable the motors, which the linorobot2 firmware does not support.

The configuration via SPI and the pin to enable the motor (see Figure 5.36) were implemented at the beginning of the *setup* function in the *main.cpp* file. Then the pin to enable the motor was set high at the beginning of the *loop* function.

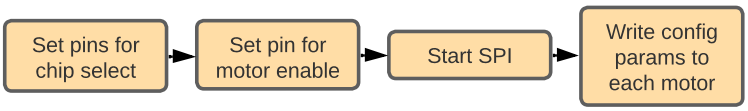


Figure 5.36: Flowchart showing the additional steps added to the *setup* function.

Implementation of 3-Channel IR Sensor Publisher
The 3-channel IR sensor is used to publish distance information used in the costmaps for navigation. The 3-channel IR sensor generates three outputs for distances in different directions (left, middle, and right). This data is sent over the ROS 2 network. For this purpose, three micro-ROS publishers are set up in the firmware to publish messages of type *sensor_msgs/msg/range*. The publishers each publish their separate distance *sensor_msgs/msg/range* messages (left, middle, and right) to their respective topics (*/tri_ir_sensor/left/range*, */tri_ir_sensor/right/range*, and */tri_ir_sensor/middle/range*). The *sensor_msgs/msg/range* message contains information about:

- The type of radiation used by the sensor in this system is IR.
- The FOV of the sensor.
- The minimum range that the sensor can see in this system is 0.
- The minimum range that the sensor can see in this system is 0.6. (Note that tests have shown that this is the maximum range at which reliable measurements have been obtained.)
- The measured range.

This data is published via the *controlCallback* function, which is linked to a micro-ROS executor.

Implementation of Function for Resetting Teensy via Serial
As mentioned earlier, a manual reboot is required if the firmware ends up in an error loop. This means that the Teensy must be powered off and on again by disconnecting the USB cable, since the button on the Teensy is only used to start the program mode.

Fortunately, the Teensy 4.1 supports a programmable reset by writing the hexadecimal value `0x5FA0004` to the address `0xE000ED0C` [71]. This could be implemented into a function that allows to reset the Teensy when it ends in an error loop. The exact function that was implemented is shown in Figure 5.37. First, it checks if the serial port is available. If it is, a check is made to see if the character **R** was received, and if so, the Teensy is reset. This function was then inserted into the error loop so that the Teensy can be reset via the serial interface.

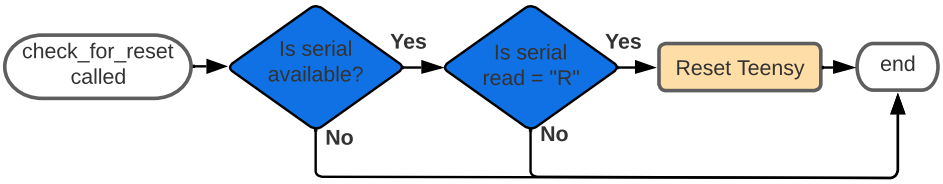


Figure 5.37: Flowchart of the Teensy 4.1 reset function.

Implementation of Subscriber for Pose Correction

Data received from the */pose* topic is used to calibrate the odometry calculated by the wheel encoders as long as data is received from the */pose* topic, i.e., when the ArUco markers are seen by the tracking system. This calibration is added to the firmware running on the Teensy. A subscriber to the */pose* topic is implemented into the firmware. A *correct_pose_callback* function is implemented that is called when a message is published to the */pose* topic. The *correct_pose_callback* function takes the X, Y, and yaw values calculated by the firmware and uses the following function:

$$XPos_{adjusted} = XPos_{adjusted} \times (1 - weightPosition) + (ArUco_x \times (weighPosition)$$

$XPos_{adjusted}$ is the X-position from the odometry data calculated by the kinematics in the firmware. $ArUco_x$ is the X ArUco marker data retrieved from the *PoseWithCovariance* messages published to the */pose* topic. The function takes the previous $XPos_{adjusted}$ and computes the next $XPos_{adjusted}$ by making the value of $XPos_{adjusted}$ converge to the same value as $ArUco_x$. A percentage is set that tells the function how to weight the value of $XPos_{adjusted}$ and $ArUco_x$. This percentage can be adjusted to make $XPos_{adjusted}$ converge to $ArUco_x$ faster. The same approach is used for the Y and yaw data:

$$YPos_{adjusted} = YPos_{adjusted} \times (1 - weighPosition) + (ArUco_y \times (weighPosition)$$

$$Yaw_{adjusted} = Yaw_{adjusted} \times (1 - weightHeading) + (ArUco_{yaw} \times (weightHeading)$$

Figure 5.38 shows an example of how the system works, where the yaw values converge to the pose of the ArUco marker estimated by the tracking system.

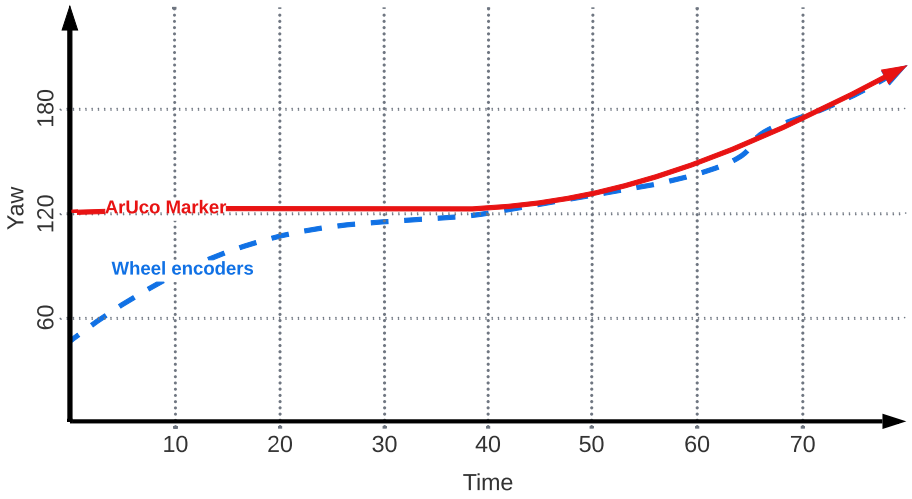


Figure 5.38: Example of how the wheel encoders adjust their values to the ArUco marker.

Figure 5.39 shows a flowchart of how the *correct_pose_callback* function works:

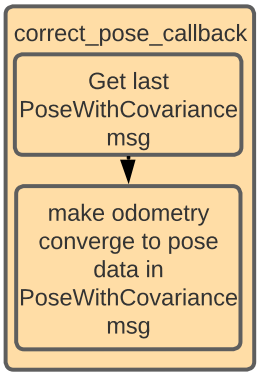
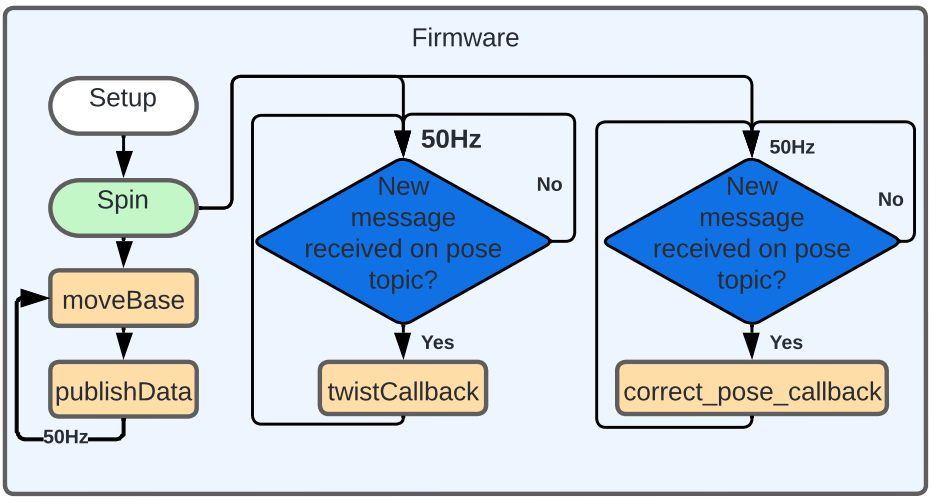


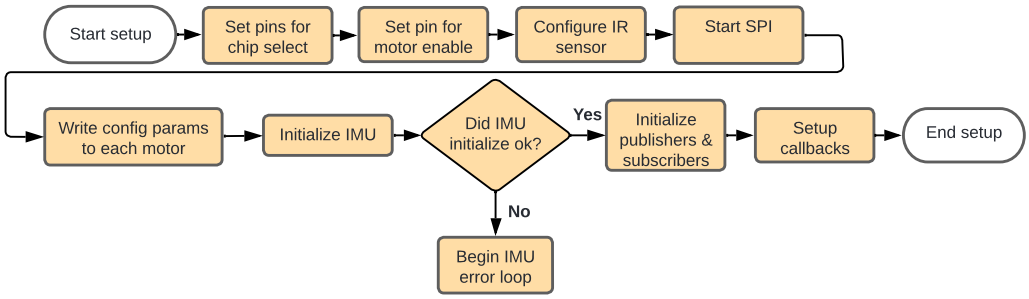
Figure 5.39: Flowchart of the *correct_pose_callback* function.

Overview of the updated Firmware Code

After implementing the additional functions in the *main.cpp* file, the code is now ready to be uploaded to the Teensy. Figure 5.40a shows the new firmware code with the additional implementations. In addition, Figure 5.40b shows the *setup* with the additional code for setting up the pinouts, configuring the motor controller and the IR sensors.



(a) Flowchart of the updated firmware code.



(b) Flowchart of the updated setup function.

Figure 5.40: Main code of the updated firmware layout.

5.8 Localization System

The localization system consists of the ROS 2 node *ekf_filter_node*, which is included in the `robot_localization` ROS 2 package [33]. This node runs on the main computer in the ROS 2 network. The node runs a Kalman filter that fuses odometry data, IMU data and ArUco marker data. The output of the Kalman filter is used as the final estimate to localize the robot.

5.8.1 Implementation of Kalman Filter

A `description_real_robot.launch.py` launch file is created to launch the *ekf_filter_node*. When the node is launched, a configuration file is passed as the launch argument. This configuration file is a .yaml file that contains the settings for how the Kalman filter operates. The code used for localization can be found in Appendix A.2.

From the firmware, messages are published to the topics */IMU/data* and */odom* via the ROS 2 network, as described in Teensy 4.1 Firmware, and from the tracking system

messages are published to */pose* topic via the ROS 2 network, as described in Tracking System. Since all these topics publish the same data about the `odom` → `base_link` transform, these values are fused in the Kalman filter.

Therefore, the configuration file specifies that the node should subscribe to the messages published to the */odom*, */IMU/data*, and */pose* topics. The configuration file also specifies what data in the messages is fused in the filter. This is done by selecting the inputs for the 15-dimensional state vector that the *ekf_filter_node* creates for each sensor:

$$(X, Y, Z, roll, pitch, yaw, \dot{X}, \dot{Y}, \dot{Z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{X}, \ddot{Y}, \ddot{Z})$$

The messages published to the */odom* topic contain data for X, Y, and yaw. These data are fused using the Kalman filter. The */IMU/data* topic provides data for $roll$, $pitch$, yaw, $\dot{roll}$, $\dot{pitch}$, $\dot{yaw}$, $\ddot{X}$, $\ddot{Y}$ and $\ddot{Z}$. Since the robot is assumed to operate in 2D space, the data for $roll$, $pitch$, $\dot{roll}$, $\dot{pitch}$, $\ddot{Z}$ have no relevance to the Kalman filter and are therefore neglected. Tests with the IMU did not show good results when data for yaw, $\ddot{X}$, $\ddot{Y}$ were used. Therefore, only the $\dot{yaw}$ data from the messages, published to the */IMU/data* topic, is fused into the Kalman filter. The */pose* topic provides data for X, Y, Z, $roll$, $pitch$ and yaw. Only data for X, Y, Z, and yaw from the messages, published to the */odom* topic, are fused into the Kalman filter. The Kalman filter calculates the joint probability distribution and outputs this estimate. Figure 5.41 shows an example of this. (Note that the Kalman filter also converges to the same value as the ArUco marker system.)

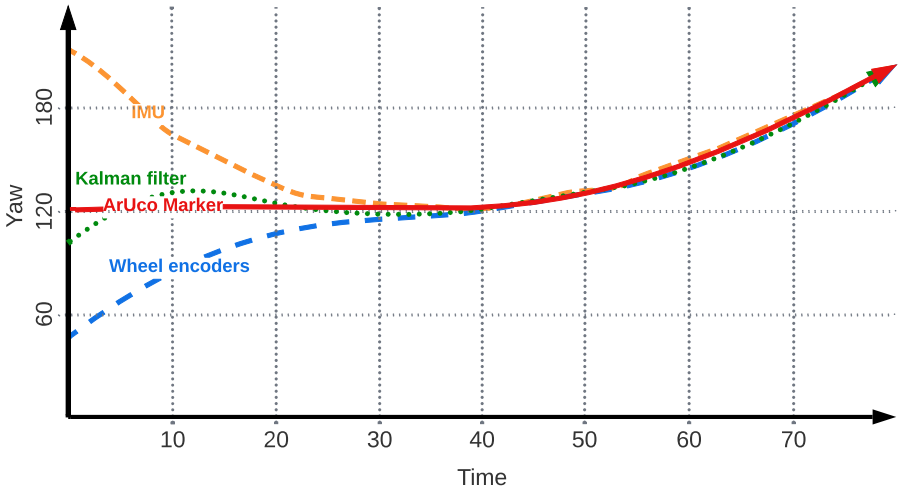


Figure 5.41: Example of how the IMU, the wheel encoders, and the Kalman filter adjust their values to the ArUco marker.

If the ArUco marker is not visible to the tracking system, the Kalman filter calculates the joint probability distribution without the fusing of the data from the ArUco marker and outputs this estimate. An example of this is shown in Figure 5.42.

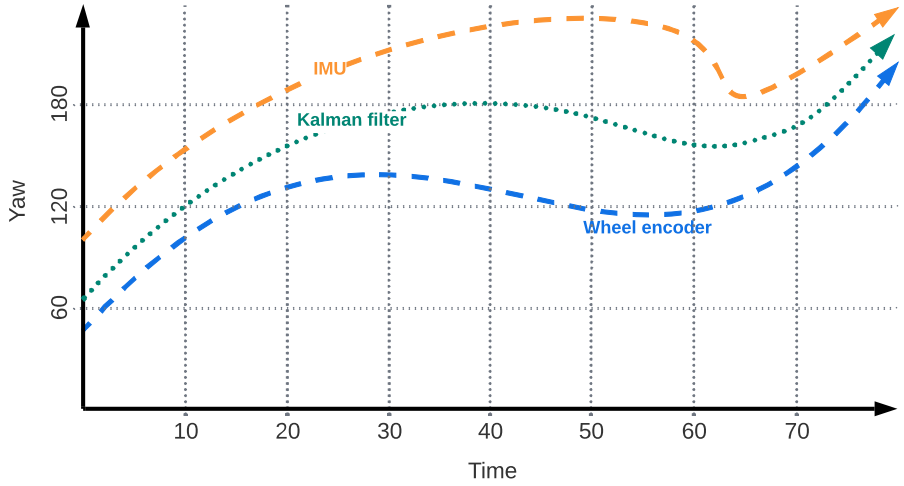


Figure 5.42: Example of how the Kalman filter fuses the estimation from the wheel encoder data and the IMU data.

This implementation ensures that the IMU and the wheel encoders have more to say in shorter periods of time, and the ArUco markers have more to say in the long run. The implementation of `robot_pose_ekf` allows for continuous estimation. The state estimation node in *robot_localization* begins estimating the state of the vehicle as soon as it receives a single measurement. If no data is received from a particular sensor, e.g., the tracking system or the IMU, the filter continues to estimate the state of the robot via an internal motion model, using previous measurements and state estimations and data from sensors that are still publishing data [33].

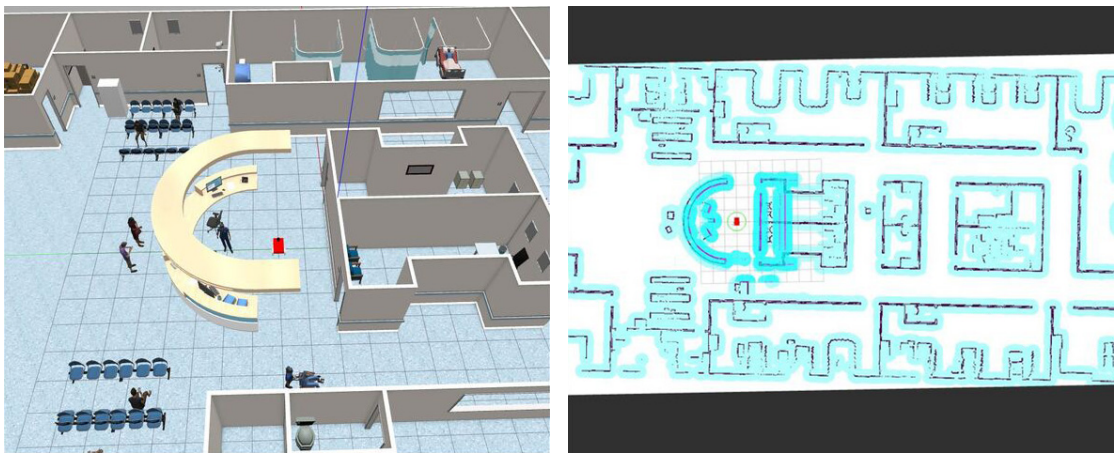
The output of the Kalman filter is published as messages of type *nav_msgs/Odometry* to the

/odometry/filtered topic. These messages are needed in the navigation system.

5.9 Navigation System

The Nav2 stack is used via a main program, which is a Python script running on the main computer.

The transform from the `odom` frame to the `base_link` frame is obtained from the tracking system. The transform from the `world` frame to the `odom` frame is not yet set up. The `world` frame represents a fixed point on a map. A map in Nav2 is defined as an image representation of an area with either free space to move or occupied space. For this project, the map could represent a factory floor. For another example of a map, see Figure 5.43.



(a) 3D simulation of a hospital building [72].

(b) Map of the simulated hospital building [72].

Figure 5.43: Example of a map as an image representation of free and occupied spaces in a simulated hospital.

This project will not focus on how to create these maps, but there are several ways to do so, either using lidars and SLAM or simply by measuring and drawing a map with a sketching software. To use the map, the drawing is converted to a .pgm file format, which is a portable gray map format. Then a .yaml file is used to specify the following:

- The path to the .pgm file.
- Where the `map` frame is anchored in relation to the map.
- What each pixel represents in meters.

In this project, a room at Aalborg University is manually measured and rendered into a .pgm file. The `map` frame is set to be anchored in the corner of the map, as shown in Figure 5.44. A .yaml file containing this information is passed to the Nav2 stack as a launch argument when launching the Nav2 map server. The map server is responsible for loading the map, serving the map, i.e., adding/deleting pixels on the map, and storing created maps.

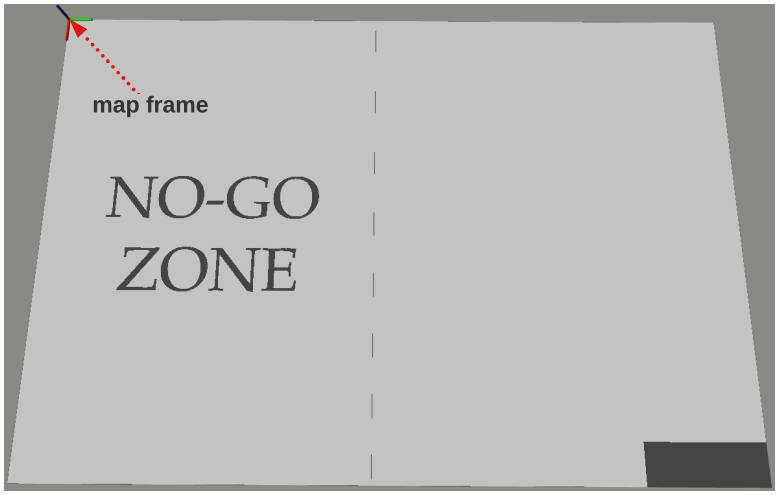


Figure 5.44: The .pgm file loaded in RViz showing the `map` frame linked to the corner of the map.

To create the final link, i.e. the `map` → `odom` transform, the *static_transform_publisher* node is used. The transform from the ArUco anchor to the `map` frame in the physical world is measured. It is important that this measurement is as accurate as possible, as it determines how accurately the map is aligned to the real world. When the *static_transform_publisher* is launched, this transform is passed in the launch argument. The *static_transform_publisher* node will then continuously publish this transform over the ROS 2 network. At this point, the `map` → `odom` → `base_link` transform tree is complete (as shown in Figure 5.45).

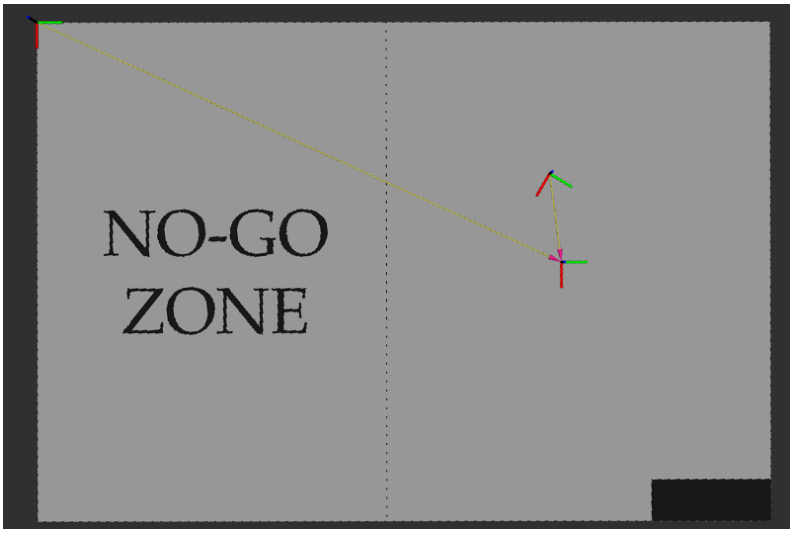


Figure 5.45: `map` → `odom` → `base_link` link visualized in RViz.

When the map server is running and the transforms are published to the ROS 2 network, a robot can be localized on the map. The map does not contain dynamic objects, nor does it take into account the size of the robot. Therefore, a path algorithm would not be able to find a suitable path to follow based on the map alone. Here, Nav2 Costmap 2D is used to create costmaps. The Nav2 stack is used to implement two costmaps: A local costmap and a global costmap. Each costmap consists of multiple layers. In this design, the layers are built as follows:

- Global costmap
  - Static layer
  - Inflation layer
- Local costmap
  - Inflation layer
  - Range layer

Static layer
The static layer is created using the `nav2_costmap_2d::StaticLayer` plugin. This layer takes the map and converts it to a costmap, as shown in Figure 5.46.

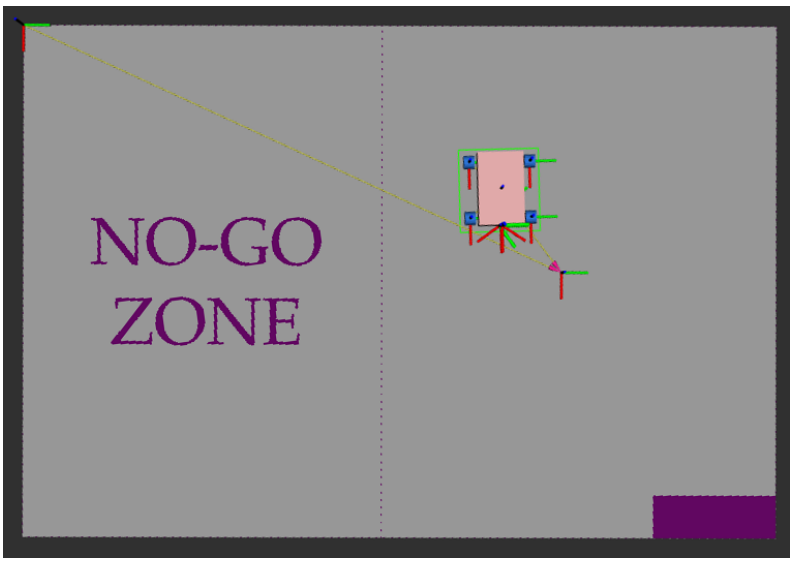
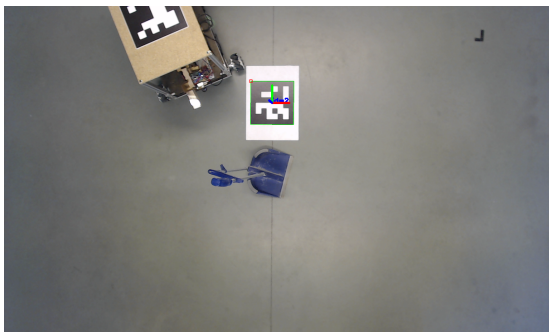


Figure 5.46: Static layer visualized in RViz.

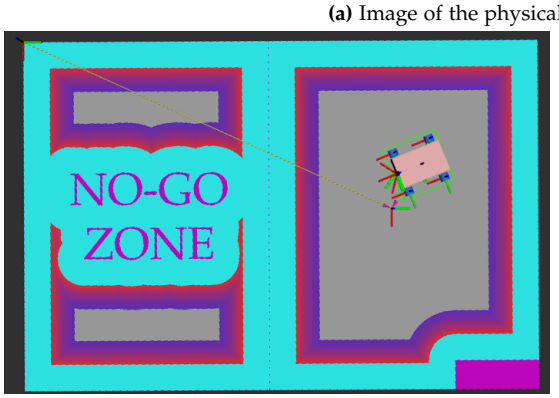
Range Layer
The range layer is created using the `nav2_costmap_2d::RangeSensorLayer` plugin. This layer takes n topics to which *sensor_msgs/Range.msg* messages are published. The *sensor_msgs/Range.msg* contains information about the pose of the distance sensor and the range that the sensor measures. The plugin uses this information to place both static and dynamic obstacles on the costmap. With this plugin, n distance sensors, such as sonars or IR sensors, can be attached to the robot so that the robot can "see" in multiple directions. In Section 5.7.3, the firmware was set up to publish over three topics. The data from these topics is used to create occupied space in the local costmap.

Inflation layer
The inflation layer is created using the `nav2_costmap_2d::InflationLayer` plugin. This layer takes a costmap and inflates it. How much the layer is inflated can be set. This is used to prevent the robot from choosing paths that are too close to walls, objects, etc.

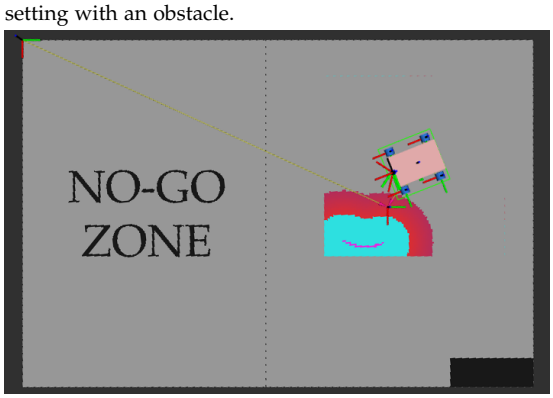
Figure 5.47 and 5.48 shows the result of these layers in the different costmaps.



(a) Image of the physical setting with an obstacle.



(b) Global costmap only running the static layer and the inflation layer.



(c) Local costmap only running the range layer and the inflation layer. (Note that obstacle is shown in a costmap).

Figure 5.47: Example of a setting and RViz showing how the local and the global costmap operate.

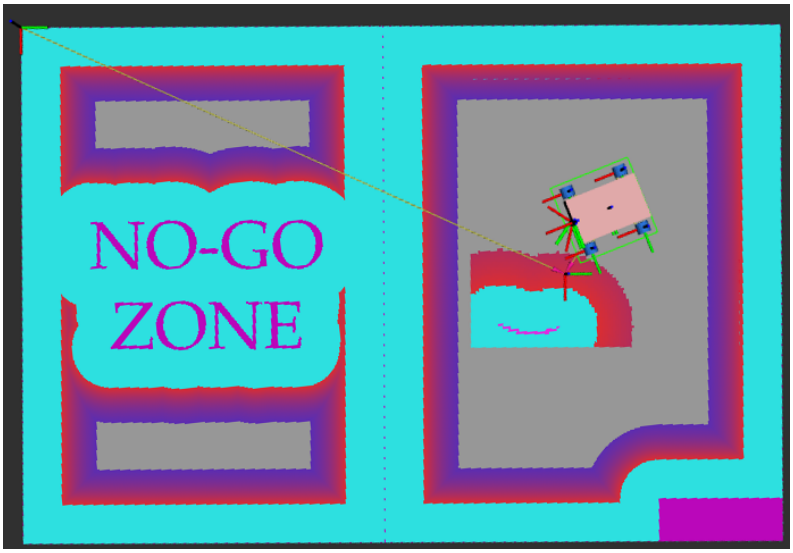


Figure 5.48: Combination of global and local costmap visualized in RViz.

The Nav2 planner server can now generate a path in the global costmap and use the local costmap to create local paths around dynamic objects. The path this design attempts to generate is the shortest possible path from `base_link` (the current pose) to a goal pose. The Nav2 goal consists of an orientation and a position. The planner server can use different algorithms to generate the path. In this project, it is assumed that the floor is flat. Therefore, the planning is done in 2D. There are a number of algorithms

available in ROS 2 and custom algorithms can also be implemented. For this project, the `nav2_smac_planner/SmacPlanner2D` plugin [73] is used. The SmacPlanner2D planner implements the A^* algorithm.

Once the path is generated, the Nav2 controller server is responsible for computing feasible control efforts for the robot to follow. There are many classes of controllers and local planners, but for this project the `dwb_core::DWBLocalPlanner` plugin is used. Finally, the control efforts computed by the Nav2 controller server are published to the robot from the main computer via the */_cmd_vel* topic on the ROS 2 network.

The firmware of the robot subscribes to the */_cmd_vel* topic. From here, the firmware executes the velocity commands via the actuators/motors.

As the robot moves, the localization design tracks the robot and the controller server implements a goal checker. For this project, the `nav2_controller::SimpleGoalChecker` plugin is used. It allows to set an `xy_goal_tolerance` and a `yaw_goal_tolerance`. When the robot approaches the goal, the goal checker checks if the pose of the robot is within the set tolerances. If it is, the robot has completed its navigation with a success result. This result is returned to the main program on the main computer.

The main program can now be used to send Nav2 goals to the robot. If the result is successful, the main program can initiate a docking sequence.

5.10 Docking System

The docking sequence is executed by the Python script *action_server_docking.py* on the Jetson Nano. The Python script handles a custom ROS 2 action server, and a custom action client is set up on the main computer. The action server is used by the main program to initiate the docking sequence. A Docking.action file is created, defined by the messages: *request*, *feedback* and *result*. A *request* message is sent from the action client to the action server to initiate a new goal. The *request* message contains information about the carrier ID, to which the process robot must dock with. The *feedback* messages are sent periodically from the action server to the action client and contain updates about a goal. The *feedback* message contains information about the state of the docking procedure. The *result* message contains information about whether the process robot has docked. Figure 5.49 shows a flowchart of the action server. The code for the docking action server can be found in Appendix A.5.

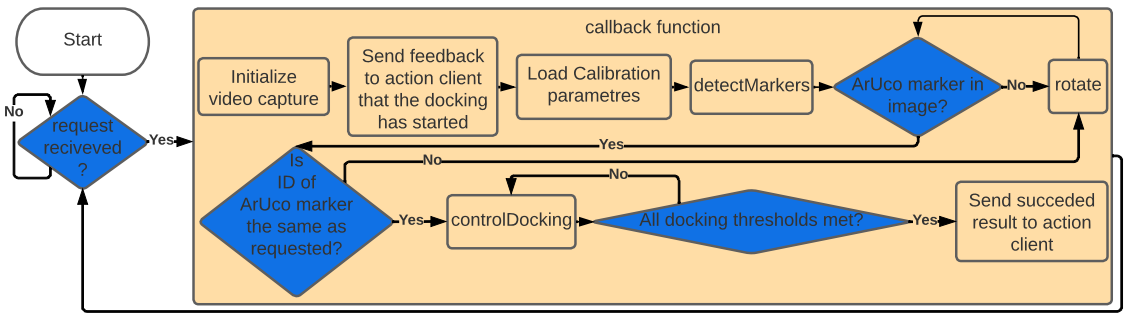


Figure 5.49: Flowchart of the action server.

The code initiates the docking server and runs it in an infinite loop. The server waits for a request from the action client running in the main program. Once a request is received, a callback function is called. This callback function starts the camera on the process robot and sends feedback to the action client that the docking sequence has started. The calibration parameters for the camera is loaded, and a while loop is entered that runs until docking is complete.

The loop runs the *detectMarkers* function, which is used in the same way as in Section 5.6. If there is an ArUco marker in the image, the *detectMarkers* function returns the ArUco marker ID as well as the rotation (*roll*, *pitch*, *yaw*) and translation (*X*,*Y*,*Z*). If an ArUco marker is found with the same ID requested by the action client, the *controlDocking* function is used to perform the docking sequence explained in Section 4.7.

The following thresholds are set in the code (can be found in Appendix A.5).

1. The *yaw* rotation in relation to the ArUco marker in part 2 of the docking sequence.
2. The *yaw* rotation in relation to the ArUco marker in part 3 of the docking sequence.
3. The *X* translation in relation to the ArUco marker in part 2 of the docking sequence.
4. The *X* translation in relation to the ArUco marker in part 3 of the docking sequence.
5. The *Z* translation that part 2 of the docking sequence begins.
6. The *Z* translation considered the final docking distance.

The *controlDocking* function takes the *Z* translations and checks if the robot is in part 2 or part 3 of the docking sequence. Then it checks if the *yaw* angle is correct, and if not a *turnRight* or *turnLeft* function is called, which sends `geometry_msgs/Twist` messages to the */_cmd_vel* topic. In the same way, the *X* translation and *Z* translation are checked and eventually the robot docks. When the robot docks, the docking action server sends the result back to the action client on the main computer. The main computer then publishes messages to the */brickfeeder_goal* topic to start the brick feeder.

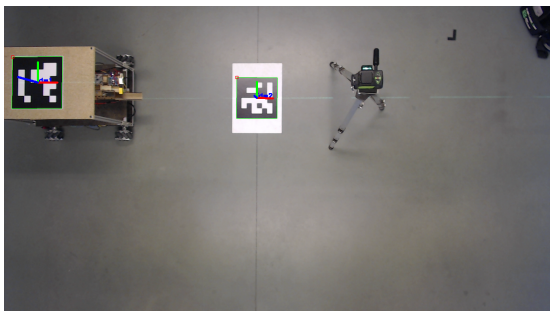
5.11 Tests

These tests are performed to check which requirements are met. Some tests cover more than one requirement, and some were performed to investigate further improvements. The results are presented for each test.

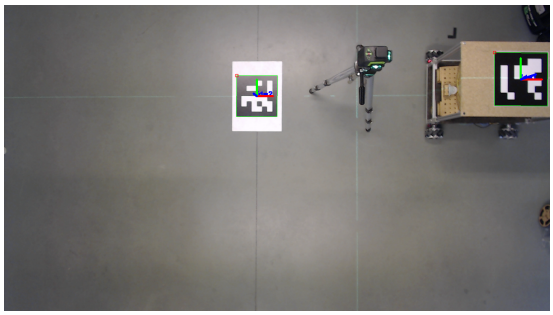
5.11.1 Test of Localization System Accuracy

A test was performed to check the accuracy of the localization system. The robot was manually positioned in two different positions along the *y*-axis of the `odom` frame, as

shown in Figure 5.50. In both poses, the robot was positioned so that the tracking system could track its ArUco marker. In both poses, a laser level was used to find the center of the ArUco marker on the floor. To do this, the laser level was aligned from both sides to make a cross on the floor, as shown in Figure 5.51.

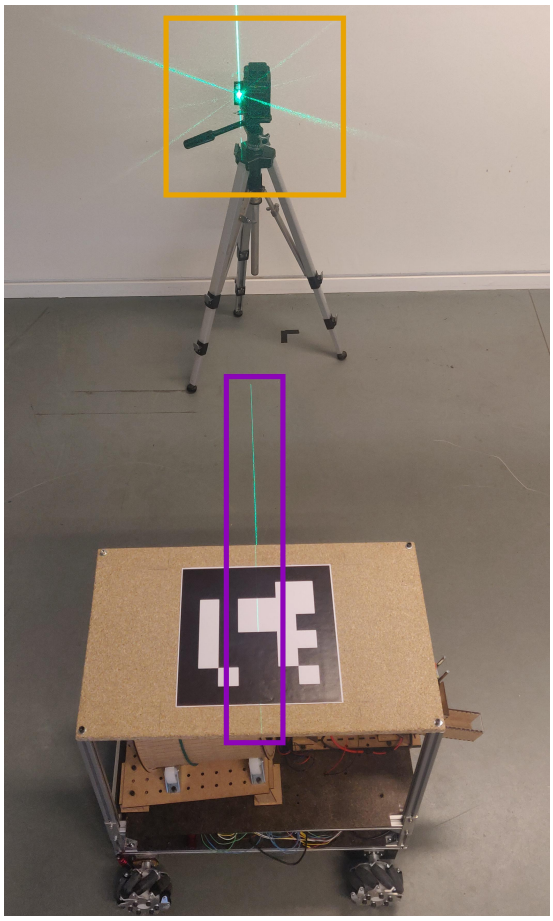


(a) Robot in pose 1 and ArUco anchor in the middle.

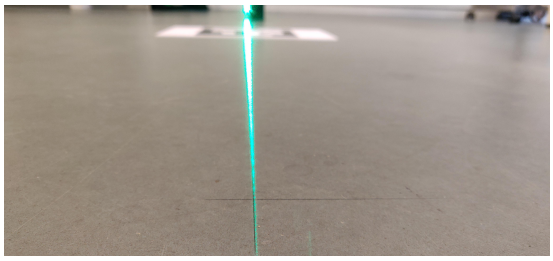


(b) Robot in pose 2 and ArUco anchor in the middle.

Figure 5.50: Robot in both poses used in the test.



(a) The orange box indicates the laser level, and the purple box indicates the laser beam.



(b) Cross made with the laser level representing the center of the ArUco marker on the process robot.

Figure 5.51: Center of the ArUco marker marked on the floor.

A physical measurement ($measurement_1$) between the two crosses on the ground was

considered the true distance traveled:

$$measurement_1 = 2.236\,\text{m}$$

A sampling of the output of the localization system was taken in both poses. The robot was placed as close as possible to the x-axis in both poses.

The X and Y values recorded in pose 1 and pose 2 can be found in Appendix C.1. As can be seen, the values fluctuate and therefore have an impact on the test. The X and Y values in pose 1 and the Y value in pose 2 do not fluctuate more than 1 mm. The X value in pose 2 fluctuates by 4 mm, so an average of 3500 samples is calculated as the X value estimate. In the end, the two position estimates of the localization system are determined as follows:

$$pos_1 = (pos_{1_x}, pos_{1_y}) = (-0.9902\,\text{m}, 0\,\text{m}), pos_2 = (pos_{2_x}, pos_{2_y}) = (1.2790\,\text{m}, 0\,\text{m})$$

This results in an estimation of distance traveled:

$$Distance_{estimated} = |pos_{1_x}| + |pos_{2_x}| = 0.9902\,\text{m} + 1.2790\,\text{m} = 2.2692\,\text{m}$$

Therefore, this test estimates an accuracy of the localization system with this setup:

$$Accuracy_1 = Distance_{estimated} - measurement_1 = 2.2692\,\text{m} - 2.236\,\text{m} = 0.0332\,\text{m}$$

Another measurement ($measurement_2$) from the center of the ArUco anchor (the origin of the localization system) to pose 1 was also physically measured. This showed an accuracy of:

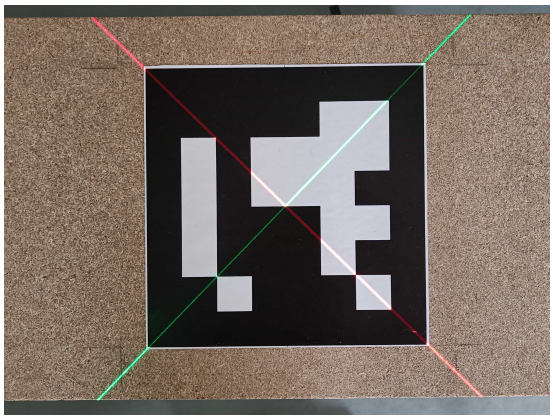
$$Accuracy_2 = |pos_{1_x}| - |Measurement_2| = 0.9902\,\text{m} - 0.9880\,\text{m} = 0.022\,\text{m}$$

This shows that the error increases at greater distances. These accuracies are examples that may change with a different setup, e.g., camera, camera position, ArUco marker size, camera calibration, Kalman filter settings, etc. Errors in the physical measurements may also contribute to an inaccurate estimate of this accuracy.

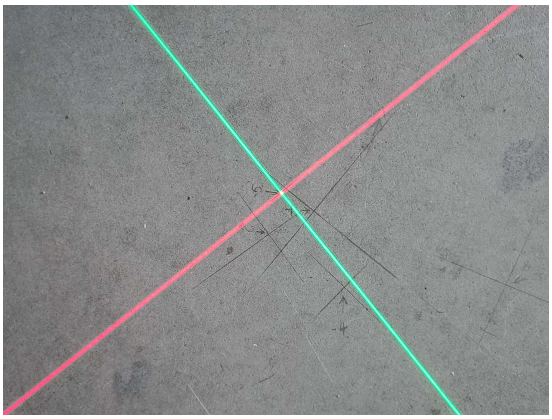
5.11.2 Test of Navigation System Accuracy

To test the accuracy of the navigation system, the main program was used to send a request to the navigation system to navigate the robot to a position in relation to the ArUco anchor. This position was $(X = 1\,\text{m}, Y = 0\,\text{m})$. After the robot moved and the navigation system provided a successful result, the position of the center of the ArUco marker was marked using laser levels, as shown in Figure 5.52. The actual position in relation to the ArUco anchor was measured. A total of five measurements were made, as shown in Figure 5.52c. The measurements are listed in Table 5.1.

(a) Setup for the test of the accuracy of the navigation system by marking the center of the robot with two laser levels.



(b) After the robot reaches the goal, the two laser levels are placed so that they intersect at the center of the robot.



(c) The robot is then moved and the intersection point is marked to be measured relative to the `ArUco anchor` frame.

Figure 5.52: The steps for testing the accuracy of the navigation system.

	X	Y	total
Measurement 1	937 mm	49 mm	938.3 mm
Measurement 2	931 mm	72 mm	933.8 mm
Measurement 3	879 mm	−36 mm	879.7 mm
Measurement 4	897 mm	43 mm	898.0 mm
Measurement 5	950 mm	63 mm	952.1 mm

Table 5.1: Navigation tes results that include coordinate measurements (X,Y) and the distance from the `ArUco anchor` frame to the measured coordinates.

5.11.3 Suspension Test

The suspension system was tested with a piece of wood 10 mm heigh. It was first placed under one of the wheels to which the suspension system is attached, as shown

in Figure 5.53. As can be seen, the other wheels were still touching the ground when the wheel with the suspension was lifted off the ground. Figure 5.54 shows how the plate is placed under the wheel without suspension. The other wheels were still in contact with the ground, which shows that the suspension can successfully compensate for a floor unevenness of up to 10 mm.

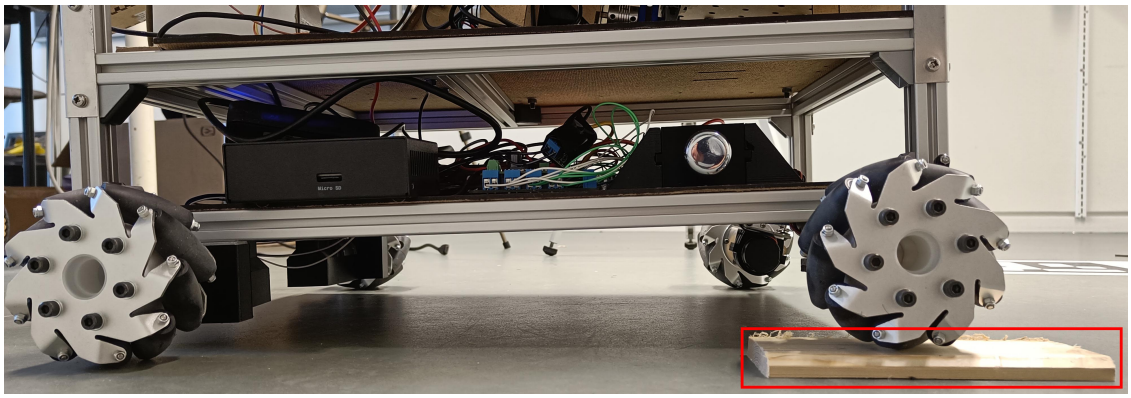


Figure 5.53: The red box indicates the piece of wood with a height of 10 mm under the suspended rear wheel.

Figure 5.54: The red box indicates the piece of wood with a height of 10 mm under the static front wheel.

5.11.4 Case Test

This test is performed to show how the prototype behaves in the LEGO case described in Section 2.4.1. A Polybot with a container is fitted with an ArUco marker on its side, as shown in Figure 5.55.

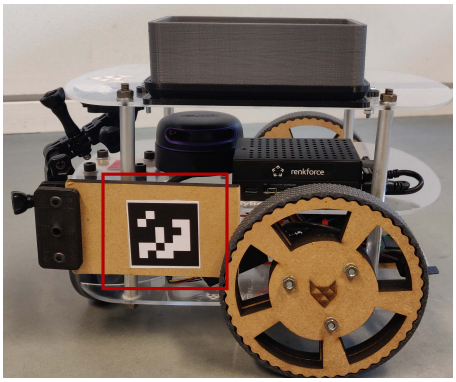


Figure 5.55: The red box indicates the ArUco marker on the Polybot used for the case test.

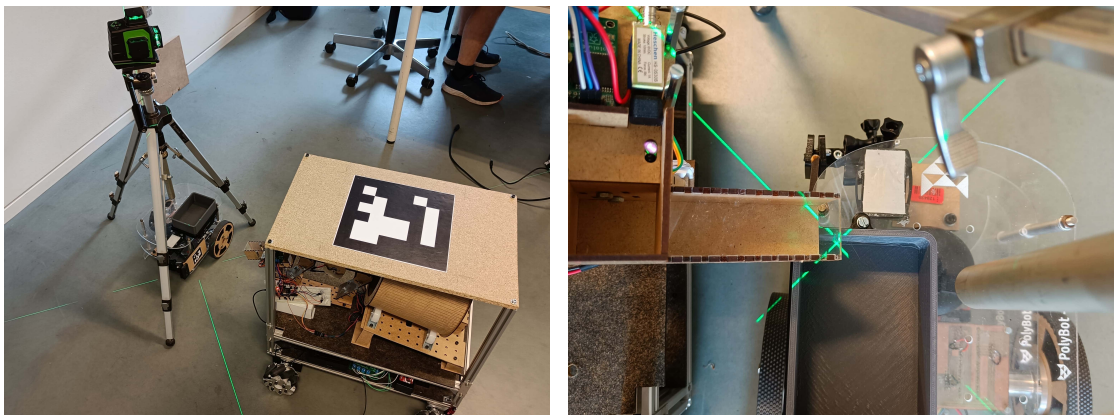
The prototype was driven to an arbitrary location on the map using the navigation system. A Polybot was manually driven to an arbitrary location on the map using a joystick. This was to simulate that the Polybot was controlled by the main computer via the ROS 2 network. The pose of the Polybot was manually estimated in relation to the `map` frame. Again, the pose of the Polybot could be published over the ROS 2 network. Then the main program was started. The main program was set up to use the navigation system to move to a pose in front of the Polybot, then to use the docking system to dock, and then use the brickfeeder firmware to dispense three LEGO bricks. A video showing the result can be found in Appendix B.1. It can be seen that the intended sequence of steps is accomplished.

5.11.5 Test of Docking ArUco ID

The docking sequence was also tested separately to test the ability of the process robot to dock with a certain carrier robot using a specific ArUco marker ID. A video of this test can be found in Appendix B.2. The test setup consists of two ArUco markers with different IDs. The process robot is placed so that both ArUco markers are in the FOV of the camera. Then the docking sequence is started. The result was that the correct ArUco marker was used for docking.

Test of Docking Precision

A test of docking precision was performed to see how close it comes to the 1 mm requirement. The test setup (see Figure 5.56a) involves a process robot being at a certain distance from the Polybot. A docking request is then sent to the process robot, instructing it it to approach the Polybot. Once it has done so, it begins the docking procedure. The first docking result is then used as a reference point by placing a laser level at the end of the feeder, as shown in Figure 5.56b. This reference point is then measured during each test and compared to the position of the feeder at the end of each docking procedure.



(a) Test setup of the docking procedure using a laser level to mark the reference point on the container.

(b) The center of the end on the chutes is then measured relative to the reference point marked with the laser level.

Figure 5.56: Setup of the docking test.

The result of these tests can be seen below in Table 5.2. The origin point, (0,0), is the location of the laser level. The calculated distance is the distance between the origin point and where the feeder landed during the remaining docking tests. A total of seven measurements were taken. The table shows that most of the tests are within a distance of 15 mm, as the average is 14.4 mm. Compared to the rest, there are two larger distances of 28.72 mm and 49.1 mm.

	X	Y	Distance
Measurement 1	7 mm	4.5 mm	8.32 mm
Measurement 2	24.5 mm	15 mm	28.72 mm
Measurement 3	2 mm	8 mm	8.2 mm
Measurement 4	2.5 mm	14 mm	14.22 mm
Measurement 5	5 mm	2 mm	5.38 mm
Measurement 6	11 mm	5.8 mm	12.44 mm
Measurement 7	49 mm	3 mm	49.1 mm

Table 5.2: Summary of the requirements that were met, partially met or not met.

6 Discussion

The prototype solution is discussed and compared to the requirements to determine which requirements are met, partially met, or not met. This is presented in a table after the requirements are discussed. The physical design of the robot, the tracking system, the localization system, and the docking system are also discussed.

6.1 Requirements

R 1: The localization and navigation design must allow scalability for multiple process robots.

Since the platform is built on ROS 2, the communication is based on topics and nodes. All topics and nodes used in localization and navigation can be stored in a so-called namespace. Namespaces modifies the names of rostopic and rosnode for different ROS 2 packages so that multiple instances of the same ROS 2 node can run simultaneously [74]. The prototype did not test this design feature. The ID capabilities of ArUco markers also allow scaling for multiple process robots to be tracked simultaneously. Therefore, R 1 is considered to be met.

R 2: The process robot must be able to be commanded to dock with a certain carrier robot.

As tested in Section 5.11.4 and 5.11.5, the prototype is able to find and dock with the correct Polybot using the ArUco marker ID, while ignoring other ArUco markers. Therefore, R 2 is considered to be met.

R 3: The process robot must have Mecanum wheels.

As designed and tested on the prototype, the process robot is capable of having Mecanum wheels and thus navigating successfully using them. Therefore R 3 is considered to be met.

R 4: The software of the process robot must be developed with ROS 2.

The localization and navigation system is built using ROS 2 and communicates with the hardware via micro-ROS. The prototype was tested with this design with success. Therefore R 4 is considered to be met.

R 5: The process robot must not weigh more than 25 kg.

The process robot itself, without the process, weighs 10.4 kg. Therefore R 5 is considered to be met.

R 6: The size of a process robot must not exceed 1×1 m.

Neither the platform design nor the prototype indicated a need for a footprint greater than 1×1 m. Therefore, R 6 is considered to be met.

R 7: The process robot must navigate to a pose on a map without collisions.

The platform design was tested with the prototype, where the navigation system was set up to add a range layer to the local costmap in the Nav2 stack. This was successfully implemented as described and shown in Section 5.9. However, the Nav2 planner server would not plan around objects in the local costmap, e.g., dynamic objects not included in the map, although the navigation would stop in front of dynamic obstacles. Therefore, R 7 is considered to be partially met.

R 8: The localization design must be able to estimate the pose of the process robot with an accuracy of 200 mm, except during the docking procedure.

As tested in Section 5.11.1, an accuracy of 33.2 mm was achieved. Therefore, R 9 is considered to be met. To further prove this, a navigation test was performed, since the accuracy of the navigation system depends on the accuracy of the localization of the process robot.

By also showing that the navigation system meets the requirements for the localization system accuracy, it can be concluded that the navigation system accuracy of the process robot also meets these requirements.

In Section 5.11.2, the navigation system was given five goals, resulting in five x and y measurements relative to the `ArUco anchor` frame. The distance from the `ArUco anchor` to the position to which the robot navigates is calculated using the x and y measurements.

$$d_i = \sqrt{(x_i)^2 + (y_i)^2} \tag{6.1}$$

The mean value is then calculated using the newly calculated lengths.

$$\mu = \frac{\sum_{i=1}^{N} d_i}{N} \tag{6.2}$$

This results in a mean value of 920.4 mm relative to the `ArUco anchor` frame and a mean value of 79.6 mm relative to the goal. The mean value shows that the robot can navigate to a position with an accuracy of more than 200 mm, which again confirms that R 8 can be considered to be met.

R 9: All wheels of the robot must be in contact with the floor at all times thus compensate for a floor unevenness of up to 10 mm.

The design of the platform was tested on the prototype. A suspension system was implemented on the prototype that allowed the rear wheels to be adjusted in height. The suspension system successfully allowed all wheels to be in contact with the floor when one of the wheels was raised by 10 mm. Therefore, R 9 is considered to be met.

R 10: The size of the robot platform must be modular to be suitable for other processes.

The ability to mount a process on the platform was tested by matching the frame to suit the brick feeder. Therefore, R 10 is considered to be met.

R 11: The process must be able to communicate with the network independently of the process robot.

Section 5.11 tested the system and showed that the communication design could be implemented on the prototype and worked as expected. The ESP32 running the brick feeder firmware was connected to the Jetson Nano via USB for power only. Therefore, R 11 is considered to be met.

R 12: Multiple process robots and processes must be able to communicate over the same network.

The platform was designed to meet this requirement, but it was not tested in the prototype. However, multiple computers were able to communicate via ROS over the network set up in the prototype. Therefore, it is assumed that the system is capable of meeting the requirement. Therefore, R 12 is considered to be partially met.

R 13: The process robot must have at least the same computer processing power as a Raspberry Pi 3.

The prototype used a Jetson Nano as the computer that had sufficient processing power to perform the docking procedure with ArUco markers. Therefore, R 13 is considered to be met.

R 14: The docking procedure must have an accuracy of 1 mm.

The data collected in Subsubsection 5.11.5 is used to calculate the accuracy of the docking procedure, which is calculated in the same way as in Subsubsection 6.1.

The result is a mean of 18.1 mm, relative to the point of origin, which means that this requirement is not met.

R CS1: A process robot must not exceed the price of 8000 DKK.

During the project, a price list was maintained that included all materials and components purchased during the project period (can be found in Appendix E.6). The result is a total cost of 24 425.76 DKK. This gives a unit price of 6196.44 DKK, which means that the requirement is met.

R CS2: The design of the process robot must allow for a LEGO brick feeder (as per design, see Figure 2.10) to be mounted on the process robot.

As demonstrated by the prototype, it was able to mount the LEGO brick feeder on the process robot prototype and use it for the case in Section 5.11.4. Therefore, R CS2 is considered to be met.

R CS3: The LEGO brick feeder must be able to receive commands wirelessly and feed n number of LEGO bricks.

As the test in Appendix B.1 showed, the brick feeder receives commands via ROS topics wirelessly over the network. It was also shown that three LEGO bricks were requested to be fed, which the feeder managed to do. Therefore, R CS2 is considered to be met.

R CS4: The height of the LEGO brick feeder chute must be at least 252 mm above the ground.

Since the test in Appendix B.1 showed that the prototype was able to dock and feed the Polybot, the height of the chute is at least 252 mm. Therefore, R CS4 is considered to be met.

R CS5: When docking, the process robot must have a maximum positional error of -49.7 mm to 49.7 mm in the x-axis and -41.7 mm to 10 mm in the y-axis.

The two axes were tested by performing seven docking procedures and then extracting the precision by testing repeatability and accuracy by testing the pose between the chute and the container after docking. The positional error is shown as a normal distribution in Figure 6.1.

Mean value:

$$\mu_x = \frac{\sum_{i=1}^{N} x_i}{N} \tag{6.3}$$

$$\mu_y = \frac{\sum_{i=1}^{N} y_i}{N} \tag{6.4}$$

Variance:

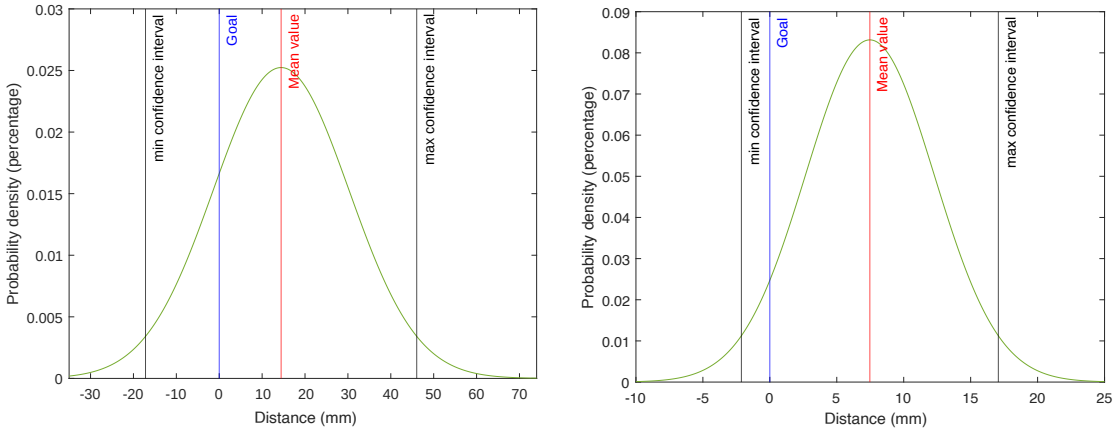
$$\sigma_x^2 = \frac{\sum_{i=1}^{N}(x_i - \mu_x)^2}{N} \tag{6.5}$$

$$\sigma_y^2 = \frac{\sum_{i=1}^{N}(y_i - \mu_y)^2}{N} \tag{6.6}$$

Standard deviation:

$$\sigma_x = \sqrt{\sigma_x^2} \tag{6.7}$$

$$\sigma_y = \sqrt{\sigma_y^2} \tag{6.8}$$



(a) Normal distribution on the x-axis.

(b) Normal distribution on the y-axis.

Figure 6.1: Normal distribution of the docking tests performed in Subsubsection 5.11.5.

Assuming a confidence level of approximately 95 percent, the minimum and maximum confidence intervals can be calculated as follows:

$$CI_x = \mu_x \mp 2 \cdot \sigma_x \tag{6.9}$$

$$CI_y = \mu_y \mp 2 \cdot \sigma_y \tag{6.10}$$

Resulting in a confidence interval of:

$$CI_x = \{-17.2\,\text{mm}, 46\,\text{mm}\}$$
$$CI_y = \{-2\,\text{mm}, 17.1\,\text{mm}\}$$

As shown in Equation 6.9, the confidence interval for the x-axis is within the required range. The y-axis, on the other hand, has a maximum value that is higher than the requirement. Therefore, this requirement is considered to be partially met.

6.1.1 Summary of Requirements

Table 6.1 shows a summary of the generic requirements, and Table 6.2 shows a summary of the case-specific requirements.

	R 1	R 2	R 3	R 4	R 5
Met	X	X	X	X	X
Partially met					
Not met					
	R 6	R 7	R 8	R 9	R 10
Met	X		X	X	X
Partially met		X			
Not met					
	R 11	R 12	R 13	R 14	
Met	X		X		
Partially met		X			
Not met				X	

Table 6.1: Summary of which generic requirements were met, partially met or not met.

	R 1	R 2	R 3	R 4	R 5
Met	X	X	X	X	
Partially met					X
Not met					

Table 6.2: Summary of which case-specific requirements were met, partially met or not met.

6.2 Tracking System

The tracking system was successful in determining the pose of the process robots within the tracking area using an ArUco marker. It can also successfully distinguish the ArUco markers based on their ID and derive their individual pose. This is then used to distinguish between the ArUco anchor and the process robot and then send the appropriate messages via a ROS 2 topic. Feedback to the user is in the form of 2D images showing how the poses of the markers were derived and what ID each marker has.

The problem with using this tracking system is that tracking the process robot becomes impossible when the camera and the ArUco marker are obscured, since the entire marker must be visible to the camera. Therefore, the production facility using such a system would have to clear the space of any paths that the robot might take that obscure the marker.

6.3 Docking

The ArUco markers are visible from a distance, allowing the process robot to begin docking from a greater distance, and they are precise enough for close-range docking.

This is because the ArUco marker gets larger the closer the process robot gets to the carrier robot, making docking more precise.

The precision of the docking system does not meet the requirement of 1 mm. Nevertheless, it can dock successfully and feed LEGO bricks to the Polybot. The problem is also that proper tools to check the precision of the docking system to 1 mm was not available. In the tests performed in Subsubsection 5.11.5, two of the tests were higher than the others. This is due to over-adjustments while the process robot is near the Polybots ArUco marker. To counteract this and achieve faster and more precise docking, it is possible to adjust the movement speed and rotation speed of the docking procedure. The precision of the robot during docking can also be adjusted. A balance between the speed of the process robot, and the final pose it must take to feed the bricks in its final docking pose could result in faster and more precise docking.

6.4 Cost and Cost-Effectiveness

Using the setup from the prototype as an estimate for the cost of a system, functions are created that can be used to determine the cost-effectiveness in a particular case.

Cost of process robot component/raw materials: $C_{process_{robot}} = 6196.4\,\text{DKK}$

Cost pr m^2 using Logitech C922 HD Pro Webcam that covers $3.5\,\text{m} \times 2\,\text{m} = 7\,\text{m}^2$: $C_{cam} = \frac{640\,\text{DKK}}{7\,\text{m}^2}$ [75]

Cost of cheap lidar [26]: $C_{c_{lidar}} = 800\,\text{DKK}$

Cost of expensive lidar [26]: $C_{e_{lidar}} = 4000\,\text{DKK}$

Cost of Polybot component/raw materials [26]: $C_{Polybot} = 5000\,\text{DKK}$

Number of process robots: n_p

Number of Polybots/carrier robots: n_c

Area of floor space in m^2: a

Cost function of swarm of process robots and Polybots tracked with ArUco marker: $cf_1(n_p, n_c, a) = C_{process_{robot}} \cdot n_p + C_{Polybot} \cdot n_c + a \cdot C_{cam}$

Cost function of swarm of process robots and Polybots both using the cheap lidar: $cf_2(n_p, n_c) = C_{process_{robot}} \cdot n_p \cdot C_{c_{lidar}} + C_{Polybot} \cdot n_c \cdot C_{c_{lidar}}$

Cost function of swarm of process robots and Polybots both using the expensive lidar: $cf_3(n_p, n_c) = (\cdot C_{e_{lidar}} \cdot C_{process_{robot}}) \cdot n_p + (C_{e_{lidar}} \cdot C_{Polybot}) \cdot n_c$

6.4.1 AAU Smart Lab Case

At AAU, there is a Smart Production Laboratory (AAU Smart Lab), a platform for teaching in the field of smart production technologies [18], [76]. The AAU Smart Lab is a reconfigurable production line set up for the production of cell phone dummies.

With the current setup (05-2022), seven processes and eight carriers are used. This is used as an example of a case for cost-effectivenes.

The AAU Smart Lab could become a swarm production by exchanging each carrier with a carrier bot and each process with a process bot. This could be done using the ArUco marker or lidar options. Cost functions are used to compare the two options. The differentiation factor is a. If this range is too large, the lidar option is more cost-effective than the ArUco marker option and vice versa.

For the AAU Smart Lab case, the intersections of the cf_1 and cf_2 cost functions are: $solve(cf_1(7,8,a)) = cf_2(7,8)) \Rightarrow a = 131.25\,\mathrm{m}^2$ The ArUco marker is more cost-effective with an area of less than $131.25\,\mathrm{m}^2$. When comparing cf_1 and cf_3 this becomes: $solve(cf_1(7,8,a)) = cf_3(7,8)) \Rightarrow= a = 656.25\,\mathrm{m}^2$

Here, the use of ArUco markers in the AAU Smart Lab Case is considered cost-effective because the Smart Lab area is less than $131.25\,\mathrm{m}^2$. To put these areas into perspective, the average size of a U.S. factory is $1459\,\mathrm{m}^2$ and the median size is $464\,\mathrm{m}^2$ [77]. Therefore, it is also assumed that the use of ArUco markers is cost-effective for median or smaller sized factories.

Considerations

The tracked area refers to the specific prototype tested in this report, but other papers have shown that ArUco markers can be tracked at distances of 4 m [78]. Thus, the camera can achieve a larger tracked area than was used in these calculations, and this could make the ArUco marker option even more cost-effective.

7 Conclusion

This project investigated how to define the requirements for a generic design for a platform that can be used for process robots in swarm production. The requirements and design focused on scalability and the use of cost-effective sensors. Therefore, an ArUco marker-based localization system was used as a critical element to be able to scale costs not only by the number of robots, but also by the size of the factory floor. A lidar is commonly used in combination with AMCL for localization, but lidars are relatively expensive and the number of lidars scales with the number of robots. With an ArUco marker-based localization system, multiple robots can be localized with a single 2D camera, which is cheaper than most lidars. To cover a large area, more cameras must be used. Therefore, the cost functions for using lidars and for using the ArUco marker-based system were investigated. It is found that the design can be more cost-effective than a lidar-based alternative with swarm production.

The design was used to create a prototype to be used in a LEGO case to evaluate if the design could be used in a swarm production environment in a specific case. The prototype showed that the generic design can be used to adapt the robot to a LEGO brick feeder as a process.

Testing showed that the prototype was able to perform essential tasks related to the case, such as localizing a robot, navigating in a confined space, docking with certain carrier robots, and feeding certain quantities of LEGO bricks to carrier robots.

The prototype uses IR sensors for object detection and avoidance. This implementation was successful in some parts, but it also showed that further research and testing needs to be done for the use of IR sensors or other cost-effective sensors for the design to be successful in collision-free navigation.

At this time, this design is only being tested in a small environment with only one camera and one process robot. A prototype that implements the design on a larger scale must be tested before the design can be considered useful for real-world applications.

Chapter 8 discusses the future work that can be done with this project.

8 Future work

8.1 Physical Design

Larger battery capacity and hot-swap:
In the future, a larger battery should be installed so that the robot has a longer runtime equal to that of the carrier robots (as mentioned in Subsubsection 5.1). If it is not possible to find a single battery with the required capacity, the possibility of combining multiple batteries in parallel should be considered. This would also require redesigning the battery drawer to accommodate the larger battery or the multiple batteries in parallel.

Another way to use multiple batteries in parallel is to make them hot-swappable. The process robot could then drive to a battery replacement station and replace the low batteries without having to interrupt the power supply to the entire system, thus reducing downtime.

Combined breakout board for Teensy and motor driver:
The breakout board for the Multimoto motor driver and the Teensy breakout board take up a lot of space and require a total of 22 cables, which can make cable management difficult.

In the future, a combined breakout board could be developed that contains both the Multimoto motor controller and the Teensy, eliminating the need to connect cables between the two. This would result in a smaller footprint and simplify cable management.

8.2 Firmware

Successful calibration of the IMU could not be achieved, resulting in limited usability of data from the IMU. Therefore, the IMU could be calibrated to achieve better performance when the process robot uses dead reckoning.

8.3 Tracking System

The tracking system could be improved or expanded in several ways. The use of multiple cameras needs to be tested on a prototype to assess the scalability of ArUco tracking. In the test setup used to test the tracking system, the camera was only 2.4 m above the floor. Further testing should determine if the camera can be placed higher, as this height is not representative of the height of a factory ceiling. In addition, the camera would then track a larger area. A disadvantage of the higher mounted camera could be the loss of accuracy, which would also need to be tested. More accurate calibration of the camera could result in more accurate and precise localization. Future work on

this system would also include scalability of the camera system so that more cameras can be set up and used for the system.

As shown in Section 2.6.1, it might be possible to synchronize the map between each robot as they interact. The carrier robots could synchronize their map as they interact or pass each other, and also as they interact with the process robot. In this way, problems with misalignment between the real world and the virtual map could be minimized, and the robots could reposition themselves as needed. This would be a viable option if the cameras cannot be placed anywhere in the production room to cover the entire production area. In this case, the IMU and wheel encoders would estimate the position of the robot until it is back in sight of a camera or can synchronize its map with another robot.

To cover a larger area with fewer cameras, a fisheye lens or a panoramic camera could be used. This would increase the FOV of the cameras, allowing one camera to cover a larger area. A library called "ArUcOmni" has been created for such a system and is described in Appendix D.2.2. When motion blur becomes an issue, event cameras have proven to be an effective camera for tracking ArUco markers. Other solutions, such as mounting the cameras on a wall instead of the ceiling, could provide a better FOV. This could be explored in future implementations.

8.4 Localization

The localization met the requirements, but for even better accuracy some things in the design could be tested.

For the wheel encoders, a more accurate calibration of the kinematics could be implemented for better dead reckoning. Regarding the IMU, more accurate calibration could be implemented for better dead reckoning. With respect to the Kalman filter, more accurate calibration of the covariance matrices and settings could be tested and implemented to improve overall tracking. To make the system run faster to perform machine learning, it is possible to implement CUDA into the tracking system. This could drastically improve the performance of the machine vision tasks and allow the robot to process images for docking and obstacle avoidance faster.

8.5 Navigation

The performance of the IR sensors was not reliable enough. Often the sensors gave nothing or a value that did not change (3 m for this particular sensor). Better, cost-effective sensors (IR sensors, sonars, or others) could be tested. This could be implemented with little or no changes to the design of the platform, the same goes for the design of the prototype.

Since the process robots are already equipped with cameras, it would be possible to implement machine vision for obstacle avoidance. This would allow the process robot to better detect the obstacle in front of it and react accordingly to the specific obstacle.

8.6 Docking

The docking system, as it is now, will attempt to dock the process robot indefinitely if it does not dock. Therefore, the docking sequence could include a recovery loop that attempts to dock within a certain amount of time. If it cannot see the ArUco marker on the carrier robot or if docking takes too long, the robot can terminate the docking sequence and ask the navigation system to navigate the process robot to another location. It can also ask the carrier robot to move.

As an alternative to the docking stages, where each axis is set individually, as described in Section 4.7, it might be more efficient to implement a PID control system. The control system would allow all axes to be adjusted simultaneously rather than one at a time. This could reduce docking time and make the system more stable, precise and accurate. When docked, it could have some sort of laser range finder that is only activated when the process robot is in the correct position. Once activated, it tells the process robot that it can turn on the feeder without missing the carrier robot. This could be combined with some smaller guide rails to help the process robot make the final small adjustments to achieve the required docking accuracy. The guide rails could also ensure that the process robots do not need cameras, as the localization system is sufficient to track and dock both the process robot and the carrier robot.

9 Bibliography

[1] Wikipedia. *Production line*. URL: https://en.wikipedia.org/wiki/Production_line. (accessed: 03.05.2022).

[2] Jörg Krüger, Lihui Wang, Alexander Verl, et al. "Innovative control of assembly systems and lines". In: *CIRP annals* 66.2 (2017).

[3] Ignacio Rodriguez, Rasmus S Mogensen, Allan Schjørring, et al. "5G swarm production: Advanced industrial manufacturing concepts enabled by wireless automation". In: *IEEE Communications Magazine* 59.1 (2021).

[4] Casper Schou. *Lecture - Swarm Production at a glance*. AAU University Lecture. (Presentation date: 16.04.2021).

[5] Casper Schou. *Lecture - Production layout for mass customization*. AAU University Lecture. (Presentation date: 26.03.2021).

[6] Casper Schou, Akshay Avhad, Simon Bøgh, et al. "Towards the Swarm Production Paradigm". In: *Towards Sustainable Customization: Bridging Smart Products and Manufacturing Systems*. Springer, 2021.

[7] Yong Yin, Kathryn E Stecke, and Dongni Li. "The evolution of production systems from Industry 2.0 through Industry 4.0". In: *International Journal of Production Research* 56.1-2 (2018).

[8] Hassan Naqi. *iPhone 13 Pro Max Model Numbers And Differences*. URL: https://itsdailytech.com/iphone-13-pro-max-model-numbers-and-differences/2682. (accessed: 06.05.2022).

[9] Cipriano Forza and Fabrizio Salvador. *Product information management for mass customization: connecting customer, front-office and back-office for fast and efficient customization*. Springer, 2006.

[10] Philip Kosky, Robert T Balmer, William D Keat, et al. *Exploring engineering: an introduction to engineering and design*. Academic Press, 2015.

[11] Giuseppe Fragapane, Dmitry Ivanov, Mirco Peron, et al. "Increasing flexibility and productivity in Industry 4.0 production networks with autonomous mobile robots and smart intralogistics". In: *Annals of operations research* (2020).

[12] The LEGO Group. *LEGO Bricks In The Making*. URL: https://www.youtube.com/watch?v=C3oiy9eekzk.

[13] LEGO Family. *LEGO Factory Tour in Billund Denmark with TheDadLab Inside the story of how LEGO is made*. URL: https://www.youtube.com/watch?v=ToVrHzHcsGI.

[14] P Greschke. *Matrix-Produktion als Konzept einer taktunabhängigen Fließfertigung (S. 180)*. 2016.

[15] Rahul Caprihan and Subhash Wadhwa. "Impact of routing flexibility on the performance of an FMS—a simulation study". In: *International Journal of Flexible Manufacturing Systems* 9.3 (1997).

[16] Kuka. *Matrix production: an example for Industrie 4.0*. URL: https://www.kuka.com/en-de/industries/solutions-database/2016/10/matrix-production. (accessed: 04.05.2022).

[17] Festo. *The Cyber-Physical Factory*. URL: https://www.festo-didactic.com/int-en/highlights/qualification-for-industry-4.0/cyber-physical-factory/. (accessed: 04.05.2022).

[18] AAlborg University. *AAU SMART PRODUCTION LABORATORY*. URL: https://www.smartproduction.aau.dk/Laboratory/. (accessed: 04.05.2022).

[19] Adriana F Melo and Lindsay M Corneal. "Case study: evaluation of the automation of material handling with mobile robots". In: *International Journal of Quality Innovation* 6.1 (2020).

[20] George Michalos, Niki Kousi, Sotiris Makris, et al. "Performance assessment of production systems with mobile robots". In: *Procedia CIRP* 41 (2016).

[21] Giandomenico Spezzano. *Special Issue "Swarm Robotics"*. 2019.

[22] Casper Schou. *Casper Schou AAU profile*. URL: https://vbn.aau.dk/da/persons/127366. (accessed: 03.05.2022).

[23] Amalia Lelia Creţu-Sîrcu, Henrik Schiøler, Jens Peter Cederholm, et al. "Evaluation and Comparison of Ultrasonic and UWB Technology for Indoor Localization in an Industrial Environment". In: (2022).

[24] LEGO. *Pick a brick hub*. URL: https://www.lego.com/da-dk/categories/bricks. (accessed: 22.05.2022).

[25] Erik Pagh Pedersen, Jonas Agner Rytter Petersen, Jonas Møller Mikkelsen, et al. "LEGO part feeder". In: (2021).

[26] AAU. *Martin Bieber Jensen*. URL: https://vbn.aau.dk/da/persons/136602. (accessed: 06.05.2022).

[27] Linorobot. *Linorobot*. URL: https://github.com/linorobot/linorobot. (accessed: 12.05.2022).

[28] Dieter Fox, Wolfram Burgard, Hannes Kruppa, et al. "A probabilistic approach to collaborative multi-robot localization". In: *Autonomous robots* 8.3 (2000).

[29] Aishwarya A Panchpor, Sam Shue, and James M Conrad. "A survey of methods for mobile robot localization and mapping in dynamic indoor environments". In: *2018 Conference on Signal Processing And Communication Engineering Systems (SPACES)*. IEEE. 2018.

[30] SYDNEY BUTLER. *What Is Inside-Out Tracking in VR?* URL: https://www.howtogeek.com/756785/what-is-inside-out-tracking-in-vr/. (accessed: 03.05.2022).

[31] Filip Šuligoj, Bojan Jerbić, Marko Švaco, et al. "Medical applicability of a low-cost industrial robot arm guided with an optical tracking system". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2015.

[32] Prabin Kumar Panigrahi and Sukant Kishoro Bisoy. "Localization strategies for autonomous mobile robots: A review". In: *Journal of King Saud University-Computer and Information Sciences* (2021).

[33] Thomas Moore and Daniel Stouch. "A generalized extended kalman filter implementation for the robot operating system". In: *Intelligent autonomous systems 13*. Springer, 2016.

[34] Peter Chondro and Shanq-Jang Ruan. "An adaptive background estimation for real-time object localization on a color-coded environment". In: *2016 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. IEEE. 2016.

[35] Izwan Azmi, Mohamad Syazwan Shafei, Mohammad Faidzul Nasrudin, et al. "Arucorsv: Robot localisation using artificial marker". In: *International Conference on Robot Intelligence Technology and Applications*. Springer. 2018.

[36] Chris Anderson. *Drawing maps with robots, OpenCV, and Raspberry Pi*. URL: https://medium.com/@cadanderson/drawing-maps-with-robots-opencv-and-raspberry-pi-3389fa05b90f. (accessed: 20.04.2022).

[37] OpenCV. *Detection of ArUco Markers*. URL: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html. (accessed: 21.04.2022).

[38] Xinyun Chi, Ge Wu, Jinqiao Liu, et al. "Review on Ultra Wide Band Indoor Localization". In: *Indonesian Journal of Computing, Engineering and Design (IJoCED)* 2.2 (2020).

[39] Sebastian Thrun, Dieter Fox, Wolfram Burgard, et al. "Robust Monte Carlo localization for mobile robots". In: *Artificial intelligence* 128.1-2 (2001).

[40] Lentin Joseph and Jonathan Cacace. *Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System*. Packt Publishing Ltd, 2018.

[41] Lars Dalgaard. *MiR developing a cheap mobile robot with ROS*. URL: https://www.dti.dk/specialists/mir-developing-a-cheap-mobile-robot-with-ros/33795. (accessed: 24.04.2022).

[42] MIR. "User Guide (en) MIR100, Revision v.3.1". In: (2020).

[43] Anca Discant, Alexandrina Rogozan, Corneliu Rusu, et al. "Sensors for obstacle detection-a survey". In: *2007 30th International Spring Seminar on Electronics Technology (ISSE)*. IEEE. 2007.

[44] Ronald C Arkin and Douglas MacKenzie. "Temporal coordination of perceptual algorithms for mobile robot navigation". In: *IEEE Transactions on Robotics and Automation* 10.3 (1994).

[45] Kiran Kumar Lekkala and Vinay Kumar Mittal. "Artificial intelligence for precision movement robot". In: *2015 2nd International Conference on Signal Processing and Integrated Networks (SPIN)*. IEEE. 2015.

[46] SparkFun Electronics. *Robotics 101 - 5 Precision Motion*. URL: https://www.youtube.com/watch?v=We6t-4wmAMU.

[47] Alexey M Romanov and Andrey A Tararin. "An automatic docking system for wheeled mobile robots". In: *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. IEEE. 2021.

[48] SLAMTEC. *RPLIDAR A3*. URL: https://www.robotshop.com/media/files/content/r/rpk/pdf/ld310_slamtec_rplidar_datasheet_a3m1_v1.9_en.pdf. (accessed: 16.05.2022).

[49] Jian-Fu Weng, Chun-Chi Lai, and Kuo-Lan Su. "AUTONOMOUS RECHARGING OF 4WD MECANUM WHEEL ROBOT VIA RGB-D SENSORY FUSION". In: ().

[50] MIR. "MiRCharge500 OperatingGuide, v1.0". In: (2019).

[51] Wikipedia. *Robot Operating System*. URL: https://en.wikipedia.org/wiki/Robot_Operating_System. (accessed: 05.05.2022).

[52] Robotics backend. *ROS1 vs ROS2, Practical Overview For ROS Developers*. URL: https://roboticsbackend.com/ros1-vs-ros2-practical-overview/#ROS1_vs_ROS2_Conclusion. (accessed: 20.04.2022).

[53] Aaron Blasdel Matt Hansen and Camilo Buscaron. *ROS 2 Foxy Fitzroy: Setting a new standard for production robot development*. URL: https://aws.amazon.com/blogs/robotics/ros-2-foxy-fitzroy-robot-development/. (accessed: 20.04.2022).

[54] ROS. *Recording and playback of topic data with rosbag using the ROS 1 bridge*. URL: https://docs.ros.org/en/galactic/Tutorials/Rosbag-with-ROS1-Bridge.html?highlight=bridge. (accessed: 20.04.2022).

[55] Linorobot. *Linorobot2 hardware*. URL: https://github.com/kajMork/linorobot2_hardware. (accessed: 12.05.2022).

[56] Linorobot. *Linorobot2*. URL: https://github.com/linorobot/linorobot2. (accessed: 12.05.2022).

[57] Ioan Sucan and Jackie Kay. *Package Summary*. URL: http://wiki.ros.org/urdf.

[58] DDS FOUNDATION. *What is the DDS standard?* URL: https://www.dds-foundation.org/omg-dds-standard.

[59] Oleg Kalachev. *ArUco markers generator!* URL: https://chev.me/arucogen/. (accessed: 22.04.2022).

[60] asenyaev. *pattern.png*. URL: https://github.com/opencv/opencv/blob/4.x/doc/pattern.png. (accessed: 22.04.2022).

[61] Sara Roos-Hoefgeest, Ignacio Alvarez Garcia, and Rafael C Gonzalez. "Mobile robot localization in industrial environments using a ring of cameras and ArUco markers". In: *IECON 2021–47th Annual Conference of the IEEE Industrial Electronics Society*. IEEE. 2021.

[62] Jingxiang Zheng, Shusheng Bi, Bo Cao, et al. "Visual localization of inspection robot using extended kalman filter and aruco markers". In: *2018 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE. 2018.

[63] Wim Meeussen. *Coordinate Frames for Mobile Platforms*. URL: https://www.ros.org/reps/rep-0105.html. (accessed: 18.05.2022).

[64] Steve Macenski, Francisco Martín, Ruffin White, et al. "The marathon 2: A navigation system". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020.

[65] CLAAUDIA. *CLAAUDIA Data Research Services*. URL: https://www.claaudia.aau.dk/. (accessed: 13.05.2022).

[66] SyonykBlog. *Battle of the Boards: Jetson Nano vs Raspberry Pi 4*. URL: https://blog.adafruit.com/2019/12/16/battle-of-the-boards-jetson-nano-vs-raspberry-pi-4-raspberry_pi-nvidiaembedded-syonykblog/. (accessed: 13.05.2022).

[67] power-sonic. *What is a battery c rating*. URL: https://www.power-sonic.com/blog/what-is-a-battery-c-rating. (accessed: 15.05.2022).

[68] Hyperion. *A guide to understanding LiPo batteries*. URL: https://www.robotshop.com/media/files/pdf/hyperion-g5-50c-3s-1100mah-lipo-battery-User-Guide.pdf. (accessed: 23.05.2022).

[69] University of New Hampshire. *Interference in Ultrasonic and IR Range Sensors*. URL: https://connectivity.unh.edu/blog/interference-in-ultrasonic-and-ir-range-sensors.html. (accessed: 18.05.2022).

[70] Thomas B Moeslund. *Introduction to video and image processing: Building real systems and applications*. Springer Science & Business Media, 2012.

[71] PDOS. *Reboot Teensy programmatically*. URL: https://forum.pjrc.com/threads/59935-Reboot-Teensy-programmatically?p=232143&viewfull=1#post232143. (accessed: 21.05.2022).

[72] Automatic Addison. *How to Load a New Map for Multi-Floor Navigation Using ROS 2*. URL: https://automaticaddison.com/category/robotics/page/10/.

[73] Steve Macenski, bpwilcox, Ruffin White, et al. *Smac Planner*. URL: https://github.com/ros-planning/navigation2/tree/main/nav2_smac_planner.

[74] NVIDIA. *2. Multiple Robot ROS2 Navigation*. URL: https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim/tutorial_ros2_multi_navigation.html. (accessed: 06.05.2022).

[75] videoudstyr.dk. *Logitech C922 HD Pro Webcam*. URL: https://www.fcomputer.dk/logitech-hd-pro-webcam-c920-webcam-960-001055?utm_source=google&utm_medium=organicshopping&utm_campaign=googleorganicshopping&gclid=Cj0KCQjwvqeUBhCBARIsAOdt45busg6HBwbdWD1gV4FSDE2YUoqu15KNjJSQ5J24o4sElJ4GsLNQQNMaAh2wEAwcB. (accessed: 23.05.2022).

[76] Ole Madsen and Charles Møller. "The AAU smart production laboratory for teaching and research in emerging digital manufacturing technologies". In: *Procedia Manufacturing* 9 (2017), pp. 106–112.

[77] U.S. Energy Information Administration. *A Look at the U.S. Commercial Building Stock: Results from EIA's 2012 Commercial Buildings Energy Consumption Survey (CBECS)*. URL: https://www.eia.gov/consumption/commercial/reports/.

[78] Raul Acuna and Volker Willert. "Dynamic Markers: UAV landing proof of concept". In: *2018 Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE)*. IEEE. 2018.

[79] uco. *ArUco: a minimal library for Augmented Reality applications based on OpenCV*. URL: http://www.uco.es/investiga/grupos/ava/node/26. (accessed: 21.04.2022).

[80] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco José Madrid-Cuevas, et al. "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6 (2014).

[81] OpenCV. *Camera Calibration*. URL: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html. (accessed: 22.04.2022).

[82] Wikipedia. *Distortion (optics)*. URL: https://en.wikipedia.org/wiki/Distortion_%5C%28optics%5C%29. (accessed: 22.04.2022).

[83] Jeremy Steward. *Camera Modeling: Exploring Distortion and Distortion Models, Part I*. URL: https://www.tangramvision.com/blog/camera-modeling-exploring-distortion-and-distortion-models-part-i. (accessed: 22.04.2022).

[84] Xianghua Ying - Xiang Me - Sen Yang - Ganwen Wang - Jiangpeng Rong - Hongbin Zha. *SWARD Camera Calibration Toolbox*. URL: https://swardtoolbox.github.io/. (accessed: 03.05.2022).

[85] Jaouad Hajjami, Jordan Caracotte, Guillaume Caron, et al. "ArUcOmni: detection of highly reliable fiducial markers in panoramic images". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2020.

[86] Milo C Silverman, Dan Nies, Boyoon Jung, et al. "Staying alive: A docking station for autonomous robot recharging". In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*. Vol. 1. IEEE. 2002.

[87] Nick Barnes and Giulio Sandini. "Direction control for an active docking behaviour based on the rotational component of log-polar optic flow". In: *European Conference on Computer Vision*. Springer. 2000.

[88] eProsima. *Micro XRCE-DDS Agent*. URL: https://github.com/eProsima/Micro-XRCE-DDS-Agent. (accessed: 14.05.2022).

[89] eProsima. *DDS-XRCE protocol*. URL: https://micro-xrce-dds.docs.eprosima.com/en/latest/introduction.html. (accessed: 14.05.2022).

Appendices

A Github References

A.1 Tracking System

https://github.com/BenMusak/ROB_vis_aruco

A.2 Localization and Navigation

https://github.com/kasperfg16/p6-swarm

A.3 linorobot2_hardware Fork

https://github.com/kajMork/linorobot2_hardware

A.4 Brickfeeder Firmware

https://github.com/kajMork/Brick_Feeder

A.5 Docking Action Server

https://github.com/BenMusak/docking_action_server

B Videos

B.1 Case Test

https://youtu.be/hx6I-bpXJGk

B.2 Docking Test

https://www.youtube.com/watch?v=dUAWVEJqxNg

C Test data

C.1 Location System Test

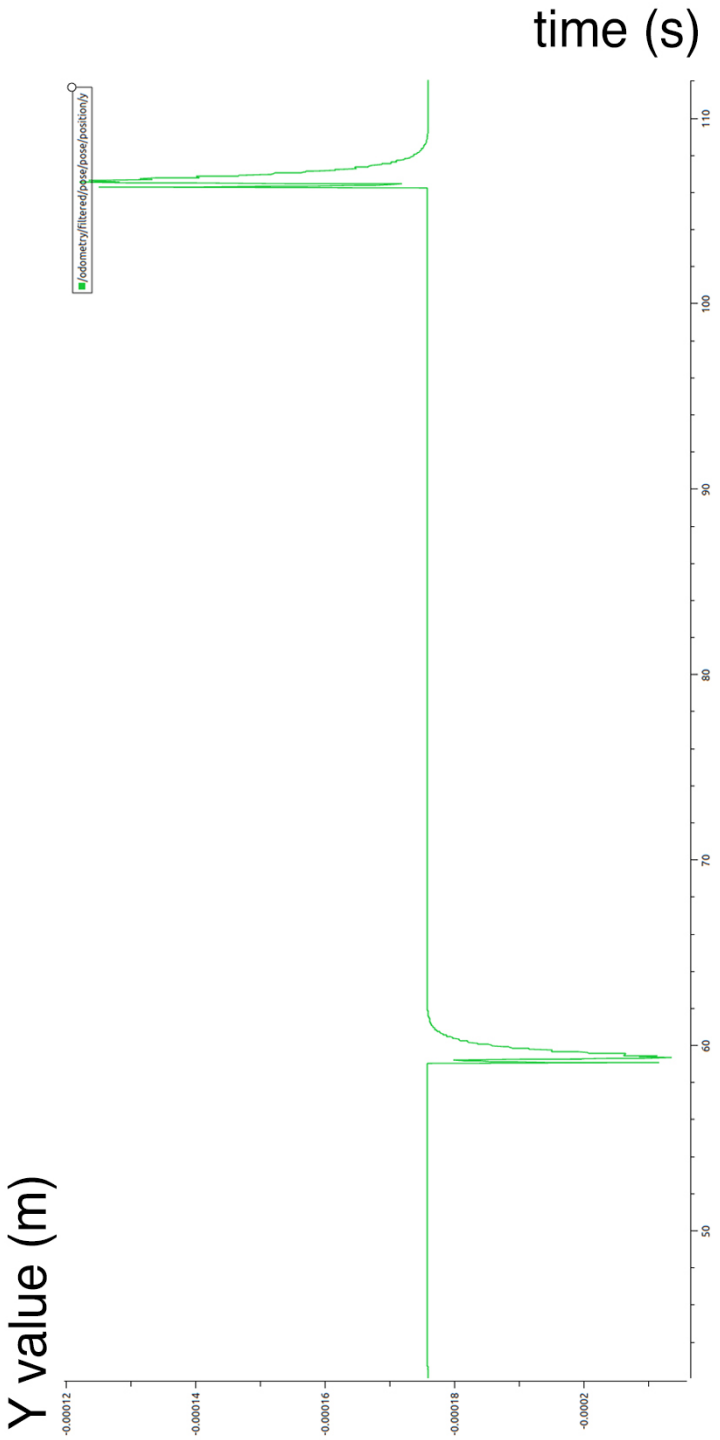


Figure C.1: Plot of Y values in pose 1.

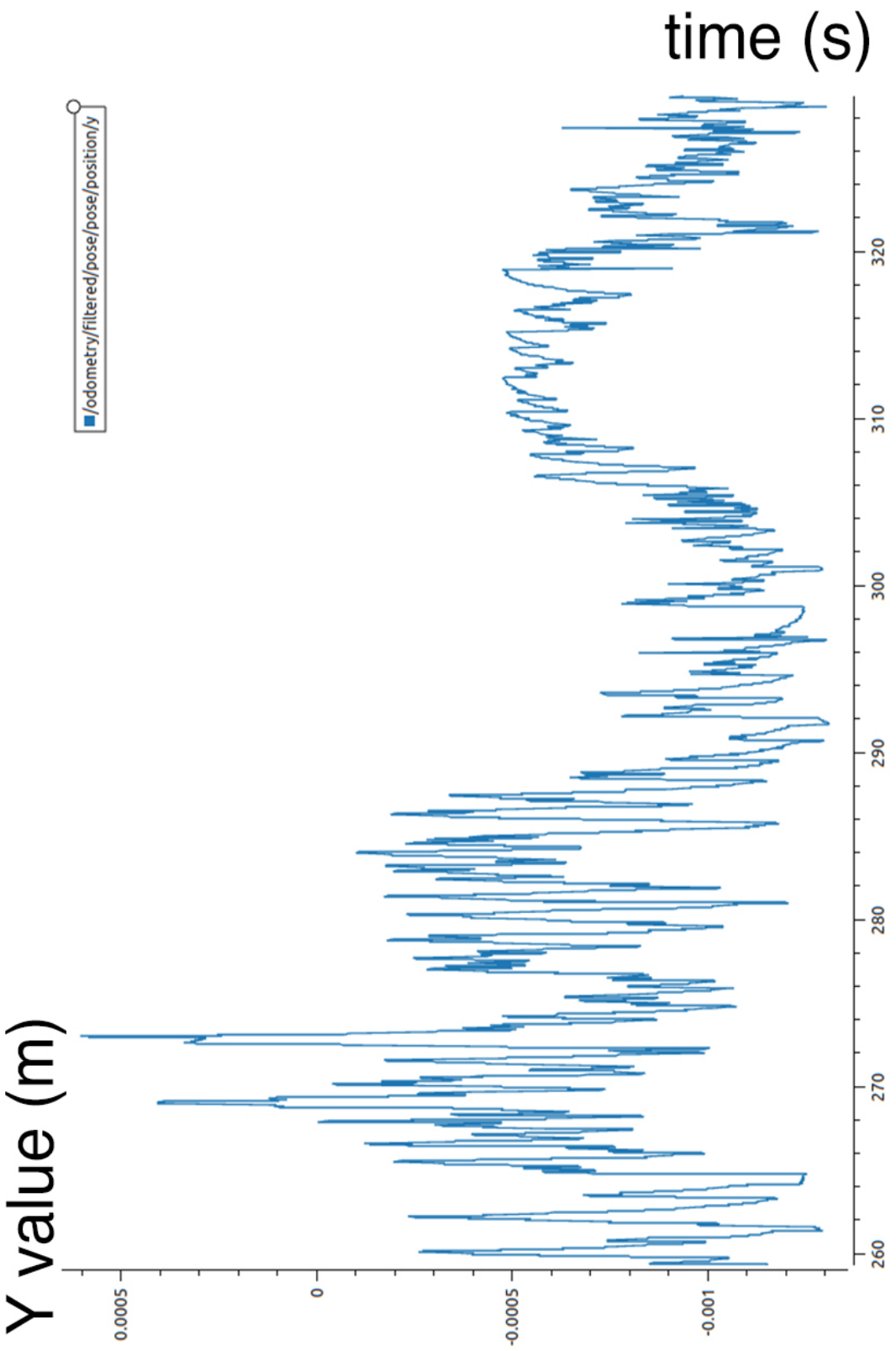


Figure C.2: Plot of Y values in pose 2.

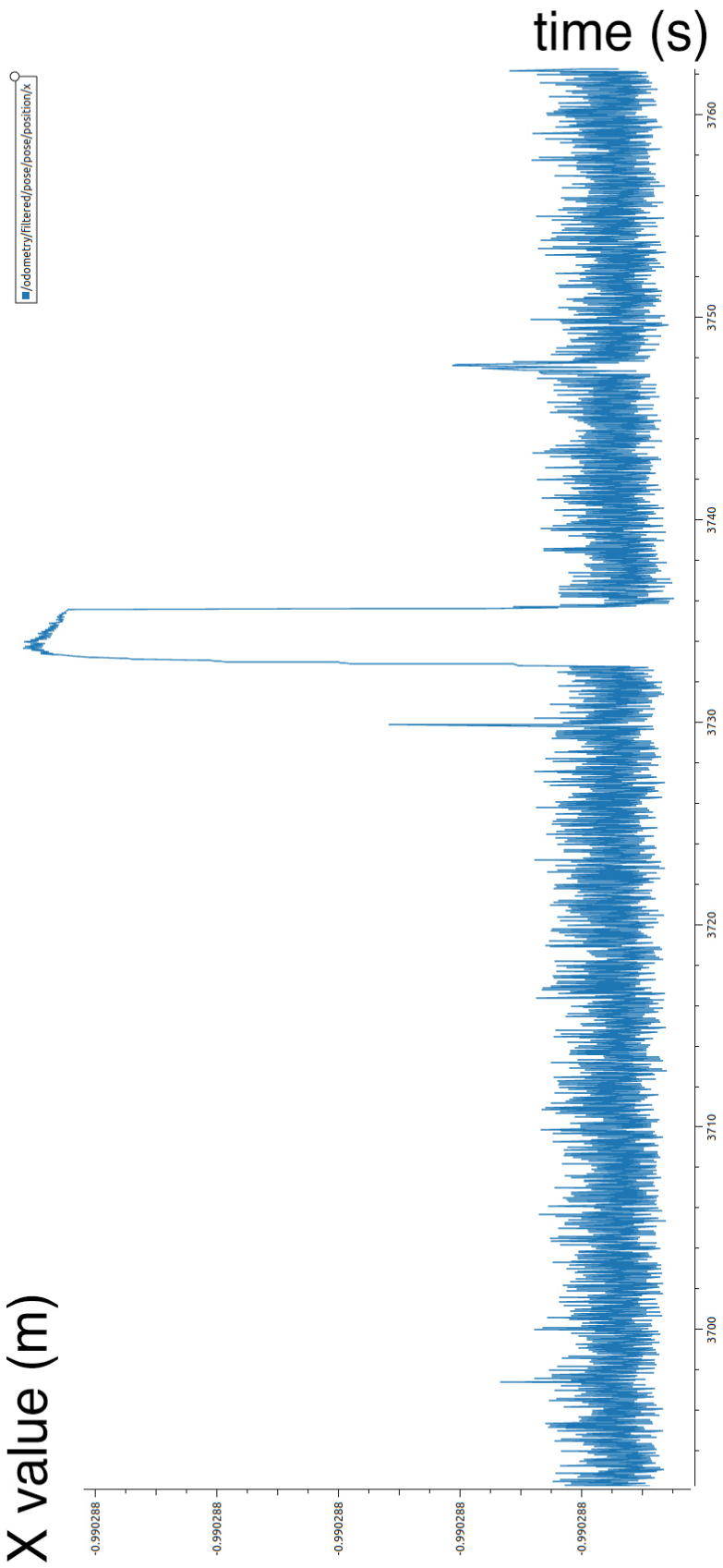


Figure C.3: Plot of X values in pose 1.

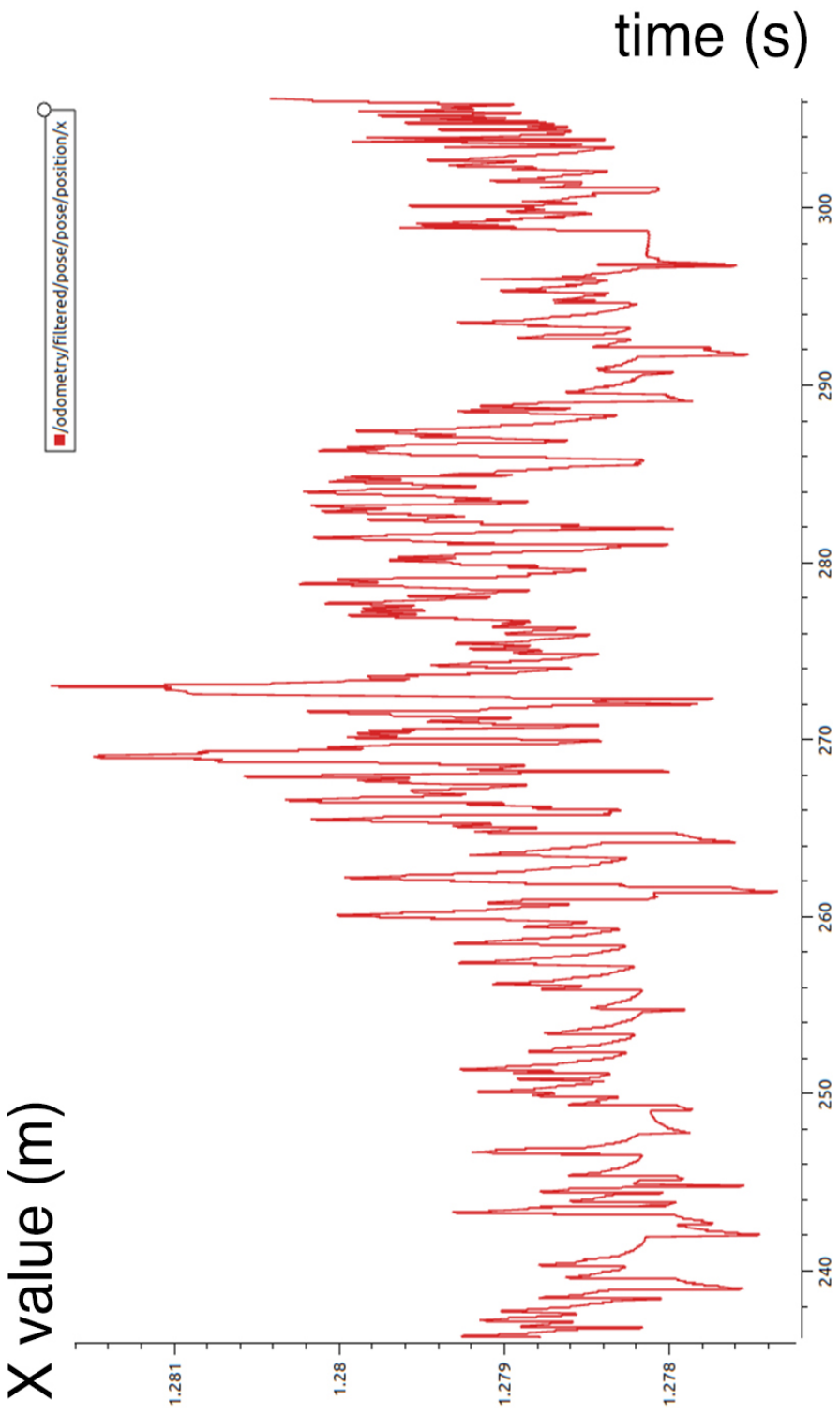


Figure C.4: Plot of X values in pose 2.

D Additional Information

D.1 BT Navigator Concept

The overall architecture of the Nav2 stack is visualized in Figure D.1.

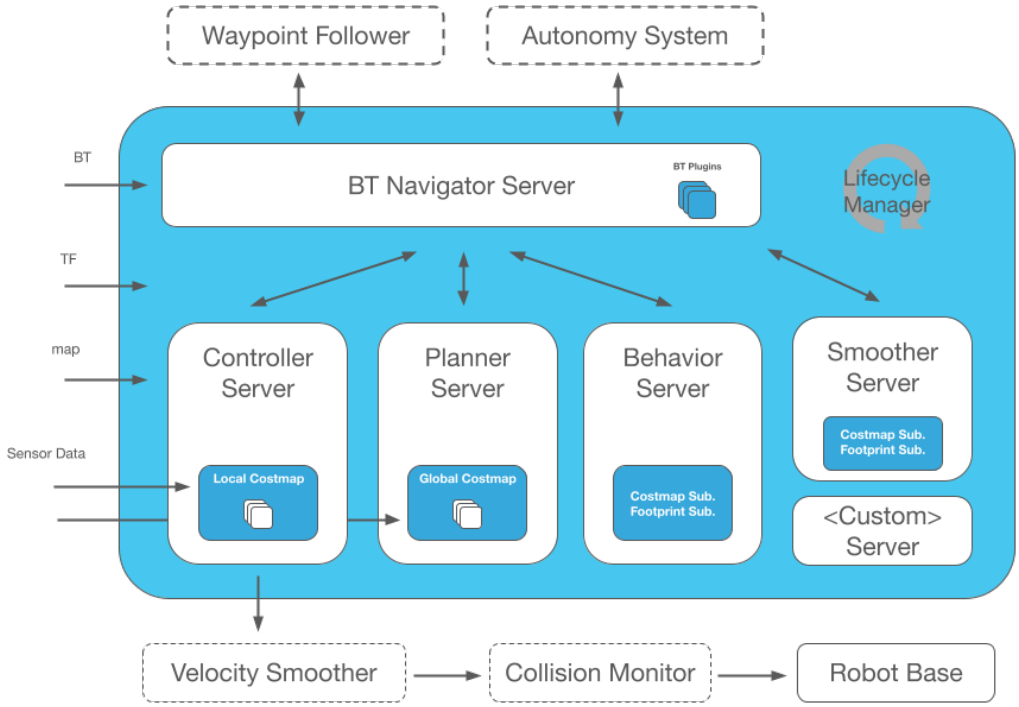


Figure D.1: Nav2 architecture [64].

The Nav2 BT Navigator server uses a behavior tree-based implementation of navigation. That is, a tree structure of tasks to be completed. The BT Navigator server acts as a finite state machine that calls different servers to perform different tasks. For example, the robot is in a planning state before it starts moving. Here, the BT Navigator server calls the Nav2 planner server. The next state is a move state, where the BT Navigator server calls the controller server to control the movement of the robot. This system allows flexibility in the navigation task and provides a way to easily specify complex behaviors of the robot, including recovery.

The Nav2 stack has several underlying tools within [64]:

- **Map server** loads, serves, and stores maps
- **AMCL** localizes the robot on the map
- **Nav2 Planner** plans a path from A to B around obstacles
- **Nav2 Controller** controls the robot as it follows the path

- **Nav2 Smoother** smooths paths to be more continuous and feasible
- **Nav2 Costmap 2D** converts sensor data into a costmap representation of the world
- **Nav2 Behavior Trees and BT Navigator** build complicated robot behaviors using behavior trees
- **Nav2 Recoveries** Compute recovery behaviors in case of failure
- **Nav2 Waypoint Follower** Follows sequential waypoints
- **Nav2 Lifecycle Manager** Manage the lifecycle and watchdog for the servers
- **Nav2 Core** Plugins to enable your own custom algorithms and behaviors

D.2 Aruco Markers for Tracking and Camera Calibration

D.2.1 ArUco Markers

With the open-source tool OpenCV2 and the contribution build it is possible to calibrate the camera and the ArUco marker tracking.

The ArUco module is based on the ArUco library developed by Rafael Muñoz and Sergio Garrido [79], [80] and has proven to be very important for robot navigation [37]. Marker tracking is about finding the correspondence between points in the real world and their 2D image projection. The advantage of using ArUco markers is that one marker and its four corners are sufficient to determine the position and orientation of the camera. The markers are also very robust because they have an internal binary codification that not only gives each marker a unique ID, but also provides the ability to implement error detection and correction techniques [37].

An ArUco marker is a square marker consisting of wide black borders with an inner binary matrix. The matrix determines the unique ID and the error and correction techniques. There is a dictionary for the markers that specifies the binary codification of each marker. This includes the size of the matrix within the marker: 4×4, 5×5, 6×6, 7×7 and the number of markers included in this dictionary [37]. A generated ArUco marker is shown in Figure D.2.

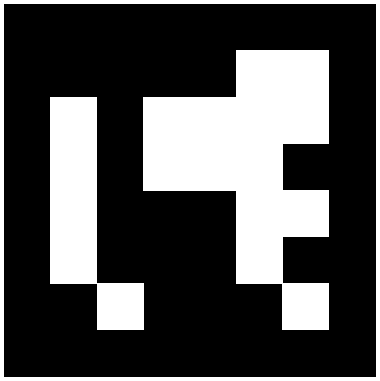


Figure D.2: A 6×6 binary codification generated with the ID of 1 [59].

D.2.2 Camera Calibration

To get the correct position of the points in the real world from the corners of the ArUco markers on the 2D image projection, it is important to know how the camera distorts the image. The distortions present in the image can cause the translation between the 2D image projections and the real world to be inaccurate. This is because the distortions cause unknown displacements between the points in the real world and the projected points on the 2D image plane, which can cause the projected values to differ from the points in the real world.

Some pinhole cameras can distort images significantly, and there are two main types of distortion: radial distortion and tangential distortion [81]. Radial distortions can be symmetrical distortions such as barrel distortion (see Figure D.3) or pincushing distortion (see Figure D.4) [82], [83], and tangential distortions can be decentering distortions caused by manufacturing defects [83].

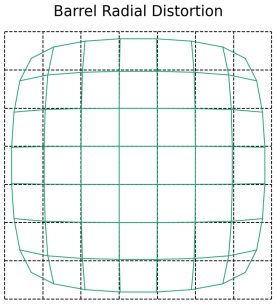


Figure D.3: Symmetrical barrel radial distortion.

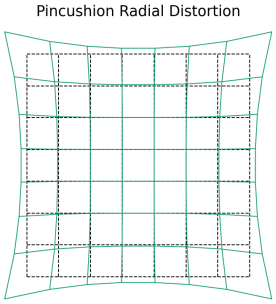


Figure D.4: Symmetrical pincushing radial distortion.

For this reason, OpenCV2 also includes camera calibration functions used to compensate for distortion. It is possible to calibrate a camera with different patterns, but for this report a checkerboard is used.

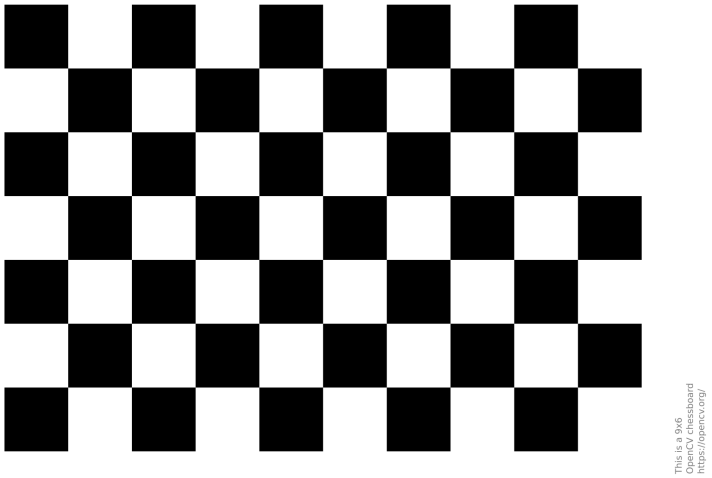


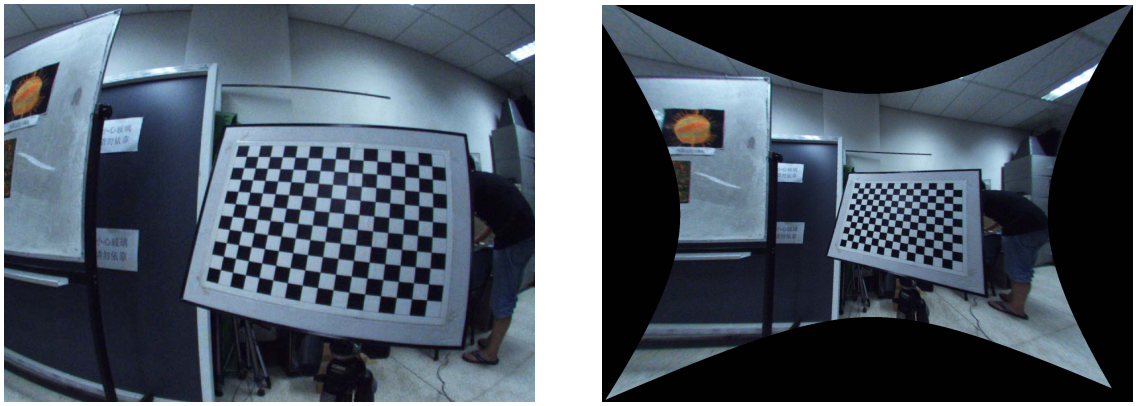
Figure D.5: Checkerboard pattern used to calibrate for camera distortions [60].

The calibration pattern introduces known variables into the image and allows correction of distortion errors when it is known what the pattern should look like and what it actually looks like in the captured images. In this way, two matrices are created that tell the built-in functions how to undistort the images.

Fisheye lenses or panoramic cameras produce a barrel distortion in any image taken with it. The advantage of using such a lens or camera is that it increases the FOV, thus covering a larger area. The disadvantage is that OpenCV only works with undistorted images [1]. If an image with these effects is undistorted, the camera corners of the undistorted image will be so warped after image correction that only the middle part of the image is usable (see Figure D.6a). This means that the additional area covered by the fisheye camera would have to be cropped out after undistortion, making the use

[1] In the case of the ArUco markers with OpenCV, this only works if the lines captured by the cameras are straight.

of such a camera unnecessary.



(a) Image with barrel distortion [84].

(b) Calibrated image from barrel distortion [84].

Figure D.6: Calibration of the fisheye effect as barrel distortion on an image.

Thus, if a larger area is to be covered by the visual system, either more cameras must be placed, or another method of tracking must be used. One such method is presented in a paper [85] that uses a method for detecting ArUco markers in a fisheye or panoramic image. This method is called ArUcOmni. However, for this report, the functions of OpenCV are used and implemented for normal, minor camera distortions without major barrel distortions.

D.3 Docking

A 2002 docking solution uses a combination of several different systems and sensors to ensure successful docking with a charging station [86]. A vision system detects an orange sign attached to the docking station that directs the robot toward the station. Upon its arrival, a laser range finder is activated, which can provide the system with more accurate feedback on the robot's angle to the charging station. This is done by placing a laser beacon at the docking station, which the laser range finder scans and detects. The laser range finder can have an angular error of 12° between it and the laser beacon and still dock successfully. This is possible because the physical setup of the docking station can direct the robot's charging port into the docking station by having angled corners that move the port toward the charger. An IR LED and IR detector is also used to tell the robot when to stop moving forward when connected to the port and to detect a voltage spike in the battery. The robot also has a built-in docking sequence that is activated when docking fails. This helps prevent a stall or misalignment that could cause the robot to run out of battery power.

Another 2002 paper uses Log-Polar Optical Flow [87], which fixes a point and attempts to move to that goal with an uncalibrated visual sensor. The pan/tilt angle of the camera is unknown, as is the speed and direction of motion of the robot with respect to the object. The only knowledge of its environment is whether points in the scene are closer or farther from the fixation point.

E Brick Feeder Rework

E.1 Increased Contact Area

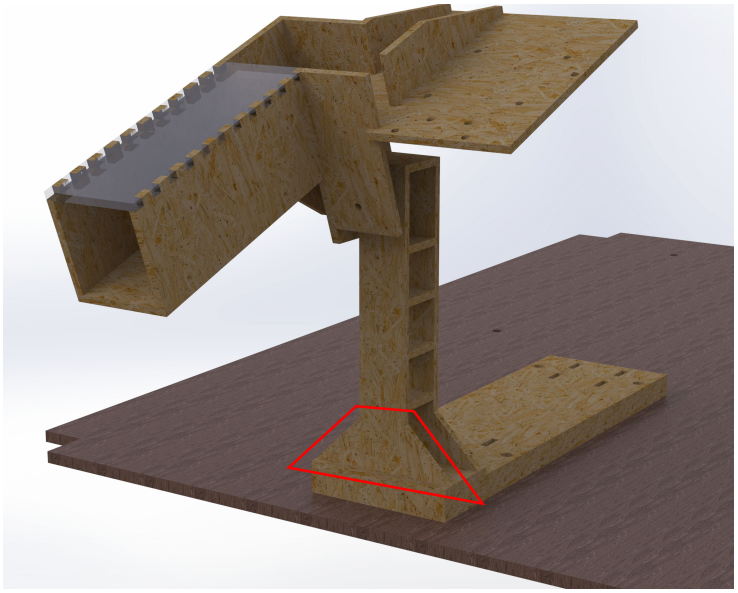


Figure E.1: New trapezoid shape on the counter of the LEGO brick feeder.

E.2 Cross Stiffeners

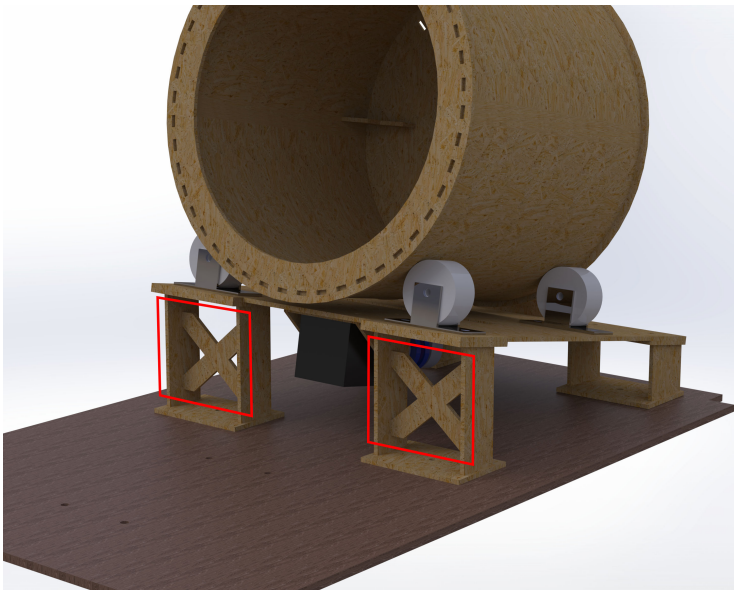


Figure E.2: New cross stiffeners in the barrel structure of LEGO brick feeder.

E.3 Shortened Lane

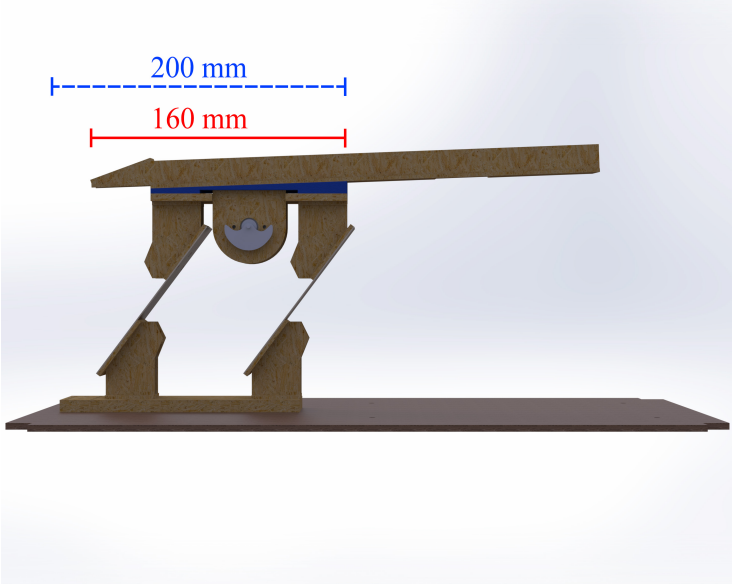


Figure E.3: Lane of LEGO brick feeder shortened by 40 mm. The blue line indicates the old length and the red line indicates the new length.

E.4 Chute Height

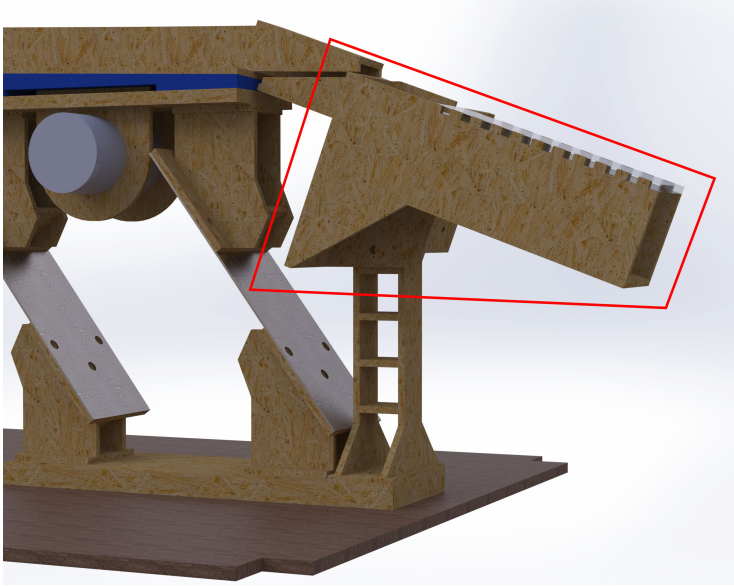


Figure E.4: New chute height of LEGO brick feeder.

E.5 Lane Angle

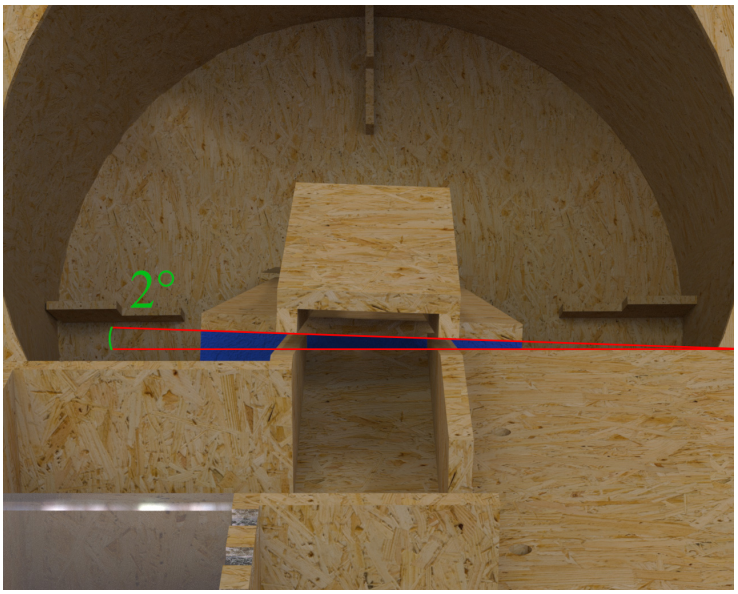


Figure E.5: New 2° angle of the lane for the LEGO bricks.

E.6 Components Price List

Number of robots		TOTAL	Single robot
4		24.425,76 kr.	6.106,44 kr.

Purchase					
Components	Qty.	Price incl. VAT	Price excl. VAT	"x" robots excl. VAT	Shipping
T-nuts	2	609,38 kr.	487,50 kr.	975,00 kr.	- kr.
Vinkel ZN 20x20	32	435,00 kr.	348,00 kr.	1.392,00 kr.	- kr.
SP2000N Profil 20x20 mini	1	566,44 kr.	453,15 kr.	1.812,60 kr.	- kr.
Brushed DC Motor w/Encoder	16	294,48 kr.	235,58 kr.	3.769,28 kr.	74,00 kr.
Motor bracket (pair)	8	65,10 kr.	52,08 kr.	416,64 kr.	- kr.
Distance sensor (IR)	4	245,26 kr.	196,21 kr.	784,84 kr.	- kr.
IMU - MPU9255	4	180,08 kr.	144,06 kr.	576,24 kr.	- kr.
Motor controller (H-bridge)	4	368,43 kr.	294,74 kr.	1.178,96 kr.	- kr.
Step-down DC-DC	4	122,75 kr.	98,20 kr.	392,80 kr.	- kr.
Omnidirectional wheels (incl 4 wheels)	4	509,62 kr.	421,18 kr.	1.684,72 kr.	- kr.
Battery	4	531,00 kr.	424,80 kr.	1.699,20 kr.	- kr.
Battey protection	4	159,00 kr.	127,20 kr.	508,80 kr.	- kr.
Microcontroller	4	269,00 kr.	215,20 kr.	860,80 kr.	- kr.
Microcontroller - Breakout Board	4	157,16 kr.	125,73 kr.	502,92 kr.	63,72 kr.
	95			16.554,80 kr.	137,72 kr.

Purchase (Extras)					
Components	Qty.	Price incl. VAT	Price excl. VAT	"x" robots excl. VAT	Shipping
Battery charger	1	690,00 kr.	552,00 kr.	552,00 kr.	- kr.
Connector for battery	4	36,37 kr.	29,10 kr.	116,40 kr.	- kr.
Toggle power switch	4	24,16 kr.	19,33 kr.	77,32 kr.	- kr.
Fuse holder	4	19,95 kr.	15,96 kr.	63,84 kr.	29,00 kr.
Fuses	1	19,95 kr.	15,96 kr.	15,96 kr.	- kr.
	14			825,52 kr.	29,00 kr.

Available at AAU					
Components	Qty.	Price incl. VAT	Price excl. VAT	"x" robots excl. VAT	Shipping
NVIDIA Jetson Nano	4	1.190,00 kr.	952,00 kr.	3.808,00 kr.	- kr.
Webcam	4	561,00 kr.	448,80 kr.	1.795,20 kr.	- kr.
				5.603,20 kr.	- kr.

Wireless connection					
Components	Qty.	Price incl. VAT	Price excl. VAT	"x" robots excl. VAT	Shipping
ESP32 DEVKITC V4 WROVER B IPEX	1	149,00 kr.	119,20 kr.	119,20 kr.	- kr.
Radxa RockPi_PT WLAN-antenne	1	69,00 kr.	55,20 kr.	55,20 kr.	- kr.
WLAN-adapter	4	215,10 kr.	172,08 kr.	688,32 kr.	- kr.
Suspension	2	129,00 kr.	103,20 kr.	206,40 kr.	- kr.
				1.069,12 kr.	- kr.

Extra Stuff					
Suspension	2	129,00 kr.	103,20 kr.	206,40 kr.	- kr.

F Communication Between Micro-ROS and ROS 2

As mentioned earlier, one of the major changes between ROS and ROS 2 is that the internal communication protocol has been changed from custom built protocols to the existing standardized DDS protocol. ROS 2 aims to support multiple DDS implementations and allows switching between them.

Micro-ROS has taken advantage of this by implementing the *DDS-XRCE protocol*, which allows resource-constrained devices (such as microcontrollers) to communicate with the DDS world just like any other DDS actor. This is implemented as a client/server relationship in the form of two libraries: *Micro XRCE-DDS Client* and *Micro XRCE-DDS Agent* (see Figure F.1). The client library is compiled and executed on the microcontroller and the agent runs on a computer in the ROS 2 network [88].

The *Micro XRCE-DDS Agent* acts as a broker bridging the clients with the DDS world. The *Micro XRCE-DDS Client* sends messages containing a request to the agent to subscribe and publish to topics in the global DDS data space. The agent then processes these requests and responds to the client with the result of the operation status and the requested data if the request is a subscribe/reply operation. Communication between the *Micro XRCE-DDS Client* and the *Micro XRCE-DDS Agent* can be via UDPv4, UDPv6, TCPv4, TCPv6, serial communication or even a custom transport [88].

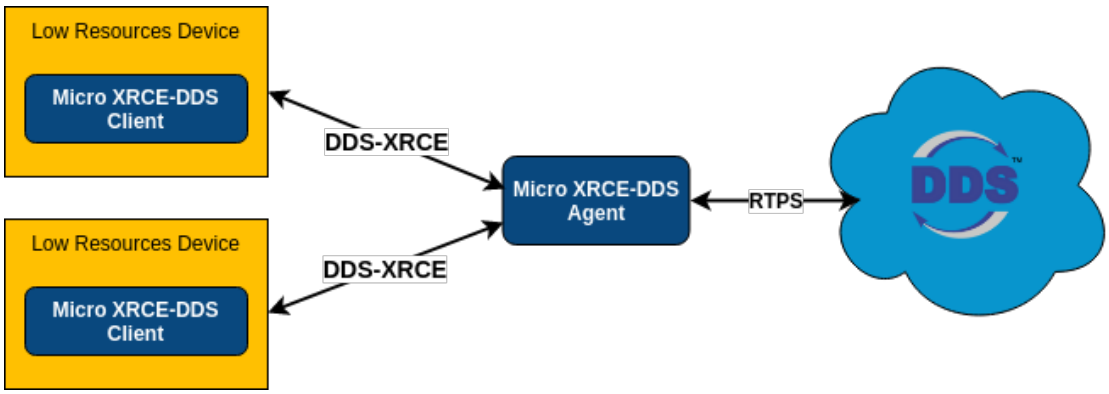


Figure F.1: The XRCE-DDS communication architecture [89].

F.1 Circuit Design of the Process Robot

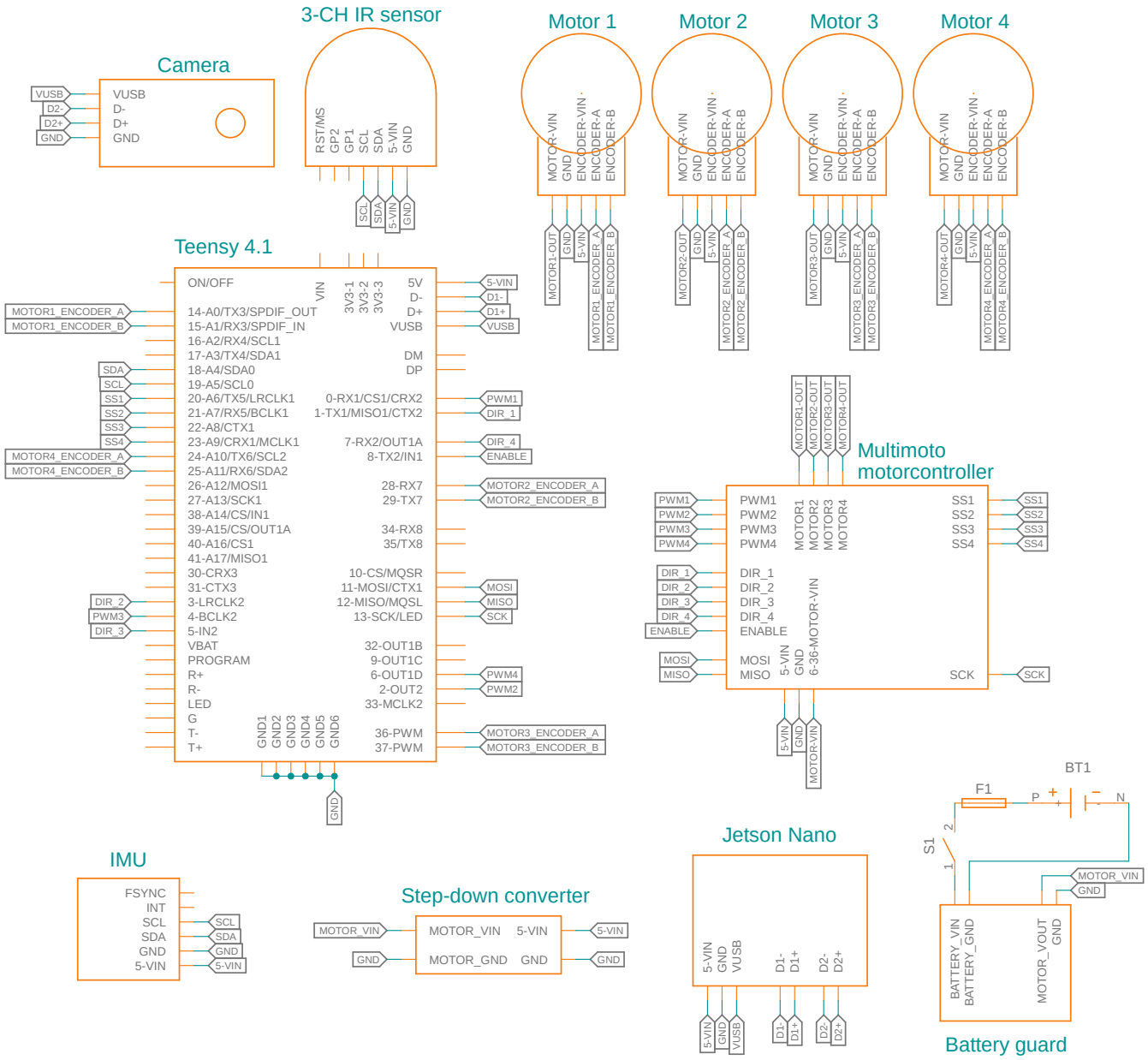


Figure F.2: Circuit design of the process robot.

F.2 Circuit Design of the Brick Feeder

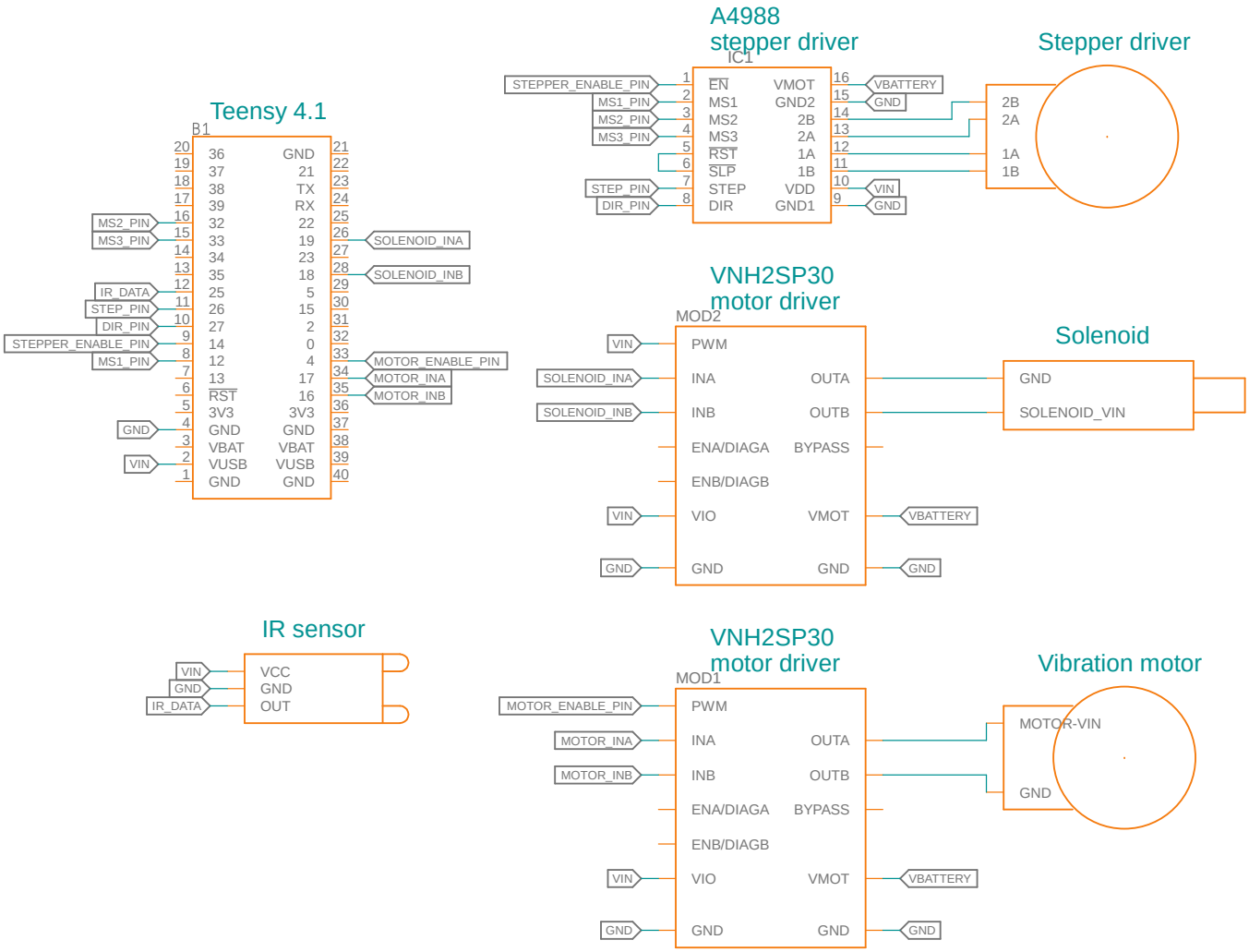


Figure F.3: Circuit design of the brick feeder.