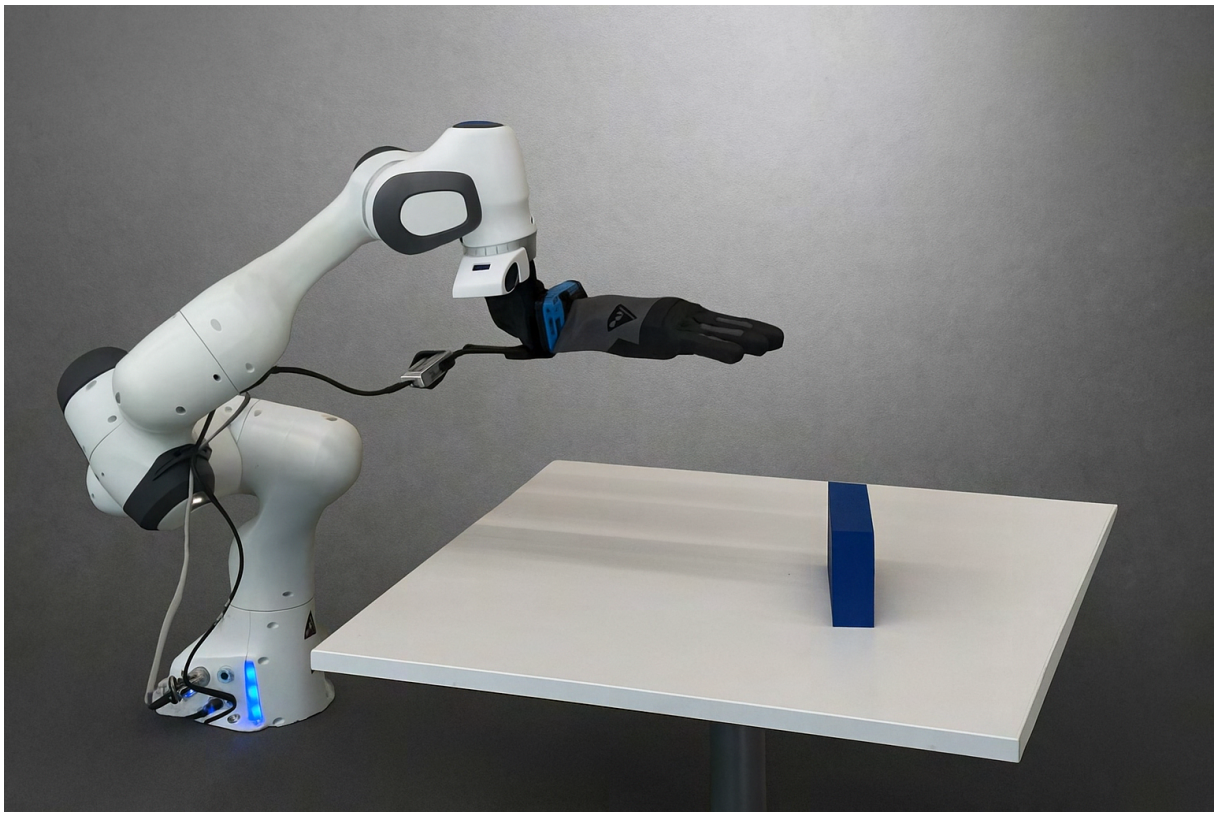


Integration of Robotic Hands and 3D Perception for Manipulation with ROS2

Maxime Lefèvre

ENSTA Bretagne, Örebro Universitet

September 15, 2025

Résumé

Ce rapport présente le travail réalisé au sein du AASS Research Center (AMM lab) de l'Université d'Örebro pour améliorer l'intégration de mains robotisées sur un bras compliant et développer une chaîne perception-vers-prise basée sur ROS 2. L'objectif initial était d'implémenter des interfaces réutilisables pour des mains multi-doigts (notamment la Schunk SIH) dans l'architecture `ros2_control`, et d'évaluer si la préemption de la fermeture de la main pendant le mouvement du bras (*in-motion preemption*) peut surpasser la stratégie classique « atteindre puis saisir ». Après des défaillances matérielles successives de la Schunk SIH, le projet a pivoté vers des expérimentations avec la QB SoftHand puis la conception et l'assemblage d'une main à trois doigts (adaptation du Yale OpenHand Model O).

La méthodologie a consisté à développer des interfaces matérielles `ros2_control` pour plusieurs mains, à décrire et publier les modèles URDF correspondants, et à mettre en place des paquets reproductibles dans un dépôt GitLab. Côté perception, une caméra Intel RealSense D435i a été utilisée pour le filtrage de nuages de points, la segmentation et l'estimation de repères d'objets par ACP (PCA). Pour l'action, un contrôleur de conformité cartésienne (compliance control) pour le bras Franka Emika Panda a été intégré et testé, incluant un contrôle interactif via RViz, la compensation de gravité, un pont capteurs de force/torque et un service de validation de prise basé sur la mesure de forces. L'orchestration globale est assurée par une machine à états finis coordonnant perception, planification et exécution.

Les résultats incluent des briques logicielles modulaires et documentées (interfaces, URDF, scripts de test, FSM), une pipeline perception-vers-grasp opérationnelle et un prototype matériel de main trois-doigts. Ce travail met également en lumière les limites matérielles rencontrées et ouvre des perspectives de recherche pour l'université d'Örebro, notamment l'évaluation de la préemption en mouvement et l'utilisation d'approches d'apprentissage automatique pour la perception.

Abstract

This report presents the work carried out at the AASS Research Center (AMM lab) of Örebro University to improve the integration of robotic hands on a compliant arm and to develop a perception-to-grasping pipeline based on ROS 2. The initial objective was to implement reusable interfaces for multi-fingered hands (notably the Schunk SIH) within the `ros2_control` architecture, and to evaluate whether grasp closure preemption during arm motion (*in-motion preemption*) could outperform the classical reach-then-grasp strategy. After successive hardware failures with the Schunk SIH, the project shifted towards experiments with the QB SoftHand and eventually the design and assembly of a three-finger robotic gripper (adaptation of the Yale OpenHand Model O).

The methodology involved developing `ros2_control` hardware interfaces for several hands, describing and publishing their URDF models, and setting up reproducible GitLab packages. On the perception side, an Intel RealSense D435i camera was used for point cloud filtering, segmentation, and object frame estimation via PCA. On the action side, a Cartesian compliance controller for the Franka Emika Panda arm was integrated and tested, including interactive control in RViz, gravity compensation, a force-torque bridge, and a grasp validation service based on force measurements. The global orchestration relied on a finite state machine coordinating perception, planning, and execution.

The results include modular and well-documented software components (interfaces, URDF, test scripts, FSM), an operational perception-to-grasp pipeline, and a physical prototype of a three-finger hand. This work also highlights the hardware limitations encountered and suggests future perspectives, such as evaluation of in-motion grasp preemption, and use of machine learning for perception tasks.

Contents

Introduction

Robotic manipulation remains one of the most complex and impactful areas in robotics research and application. Enabling a robot to perceive, plan, and execute grasps in a dynamic environment touches upon a wide range of technological challenges: real-time perception, hardware control, mechanical robustness, and software integration. My internship at the AASS Research Center, within the Autonomous Mobile Manipulation (AMM) lab at Örebro University, was situated precisely at this intersection. It aimed to advance the integration of robotic hands on compliant manipulators and to explore perception-driven grasping strategies using ROS 2.

0.1 Context Overview

Örebro University is a relatively young and multidisciplinary Swedish university, with approximately 17,000 students and 1,500 staff members. It has long been involved in robotics and artificial intelligence research. The university actively participates in major national and international projects [1].

Its main robotics research hub is the AASS (Applied Autonomous Sensor Systems) center, which brings together all AI and robotics-related activities at the university. The AASS center at Örebro University serves as a multidisciplinary research environment dedicated to autonomous systems, with a focus on their perceptual and cognitive capabilities. AASS is internationally recognized for its expertise at the intersection of AI and robotics, and it collaborates closely with industry both in Sweden and globally. The center includes about 40 researchers, divided among five thematic laboratories. However, our main focus here lies in robotic manipulation and visual perception, which are covered by the AMM laboratory [2].

The AMM lab is Örebro's research group focused on mobile manipulation. It aims to enable robots to "perform complex interactions with their environment" by combining perception and motion generation for both mobility and manipulation tasks [3]. During the internship, I developed and integrated several robotic hands on this arm, leveraging ROS 2 and the lab's 3D perception tools (e.g., Intel RealSense cameras).

My supervisor is Todor Stoyanov. He is an Associate Professor and the head of the AMM lab within AASS. He specializes in mobile robot autonomy, with a strong focus on perception algorithms and motion generation for manipulation. He leads the scientific direction of the AMM lab and supervises several PhD theses on topics such as learning manipulation skills, object perception, and affordance extraction for whole-body manipulation. In addition to his research, he teaches introductory robotics courses at the master's level and offers final-year thesis projects in robotic manipulation [4].

0.2 Goals of the internship

More specifically, I contributed to the integration of multi-fingered robotic hands (including 5-finger hands) into the lab's ros2_control infrastructure. The work was meant to provide a reusable and modular base for future thesis projects or research activities, with the long-term goal of enabling flexible benchmarking of grasping strategies. One of the core research hypotheses was to evaluate whether in-motion grasp preemption, that is, initiating finger closure during arm motion, could outperform the classical approach of reaching a target pose before grasping. This could be explored using simple baseline algorithms and then compared with more advanced machine learning-based approaches. The initial objective of the internship was to investigate whether initiating grasp closure during arm motion (in-motion preemption) could be effective, by integrating a commercial Schunk SIH hand with a Franka Emika Panda robot using ROS 2, as part of a full perception-to-grasping pipeline.

However, hardware issues quickly reshaped the scope of the project. After two successive hand failures, I shifted towards integrating a soft hand (qb SoftHand) and eventually building a three-fingered robotic gripper. These successive pivots introduced new challenges and learning opportunities: interface abstraction, low-level motor control, mechanical design and high-level state machine design. From a broader perspective, the project also highlighted several human and organizational aspects: the need to adapt to hardware failures, collaborate with international researchers, and contribute meaningfully to a fast-evolving research context.

For the host lab, the internship helped reinforce its experimental manipulation platform and laid the groundwork for future evaluation of grasping strategies. From my own perspective, this experience allowed me to contribute both as a developer and as a research assistant, by building bridges between low-level control, high-level behavior design, and experimental research goals. I believe I was selected for this internship not only to implement technical solutions, but also to bring a critical and creative approach to manipulation challenges in ROS 2.

In what follows, I will describe the technical content of the internship in detail, highlighting the key developments in control, perception, integration, and hardware design, as well as the lessons learned at each stage.

Chapter 1

Initial Robotic Hand Integration: Schunk SIH

1.1 Overview of the Schunk SIH Hand

The Schunk SIH Hand is a robotic end-effector developed as a low-cost, energy-efficient prototype aimed at replicating the human hand at scale. Designed for research purposes, this particular prototype has been made available to Örebro University. As a result, the full technical documentation is not publicly accessible.

Figure 1.1: Schunk SIH Hand

1.1.1 Degrees of Freedom and Tendon-Driven Design

The SIH Hand features five degrees of freedom (DOF), each corresponding to a key motion of the human hand:

- Thumb flexion
- Index finger flexion
- Middle finger flexion
- Coupled ring and little finger flexion
- Thumb rotation

A notable design choice is the coupling of the ring and little finger, which are driven by a single actuator. This reduction in independent DOFs contributes to both cost efficiency and lower

energy consumption. In robotic hand design, it is often unnecessary or impractical to actuate every anatomical DOF independently. Instead, underactuation, especially via tendon mechanisms, offers a robust compromise by allowing complex, adaptive motion with fewer actuators.

Each flexion movement is achieved by pulling a tendon connected to the respective finger, acting against a system of torsion springs integrated at the joints. The motion follows a natural sequence: first, the proximal and distal phalanges bend together via joint J1; with continued tendon pull, the distal phalanx bends further via joint J2. The extension of the fingers is not actively controlled: it is accomplished by simply releasing the tendon tension, allowing the springs to restore the finger to an open position. This semi-actuated nature introduces a hysteresis in the finger trajectory, as the same command value may result in different positions depending on whether the motion is opening or closing.

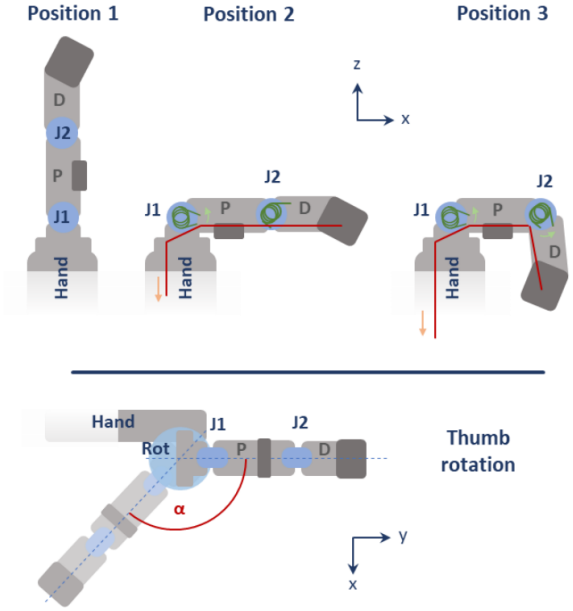


Figure 1.2: Possible finger movements

1.1.2 Human-Scale and Control Interface

Designed to mimic the size of a human hand, the SIH Hand is especially suited for tasks requiring human-like dexterity or interaction with human-designed tools and environments.

From a control perspective, the SIH Hand operates using cyclical actuator data (e.g., for motion control) and acyclical diagnostic queries (e.g., for system status or error reports). It is connected via RS-485 communication through a USB converter and requires a 24V power supply.

As a prototype, the hand is not natively compatible with ROS 2. Therefore, to integrate it into a ROS 2 system, acustom hardware interface must be implemented, particularly to conform with the ros2_control framework.

1.2 Implementation of the Hardware Interface in `ros2_control`

Integration of the Schunk SIH robotic hand into a ROS 2 system requires the implementation of a custom hardware interface compliant with the `ros2_control` framework. As SIH is a prototype and is not natively compatible with ROS 2, this layer ensures communication between robot actuators and the high-level controllers used in ROS.

1.2.1 Overview of ros2_control

The ros2_control framework provides a modular and real-time capable architecture to control robots in ROS 2 [5]. It defines a separation between:

- **Controllers**, which implement control strategies (e.g., position, velocity, effort)
- **Resource Manager**, which loads hardware interfaces from URDF/xacro files (using ros2_control tags), manages multiple hardware instances, provides a unified interface to controllers, and handles resource allocation to prevent conflicts between them.
- **Hardware interfaces**, which handle communication with the robot's actuators and sensors

The hardware interface must implement methods to:

- `read()`: retrieve state data from the hardware (e.g., joint positions, velocities)
- `write()`: send command data to the hardware
- `export_state_interfaces()`/`export_command_interfaces()`: define which data will be exchanged with ROS controllers

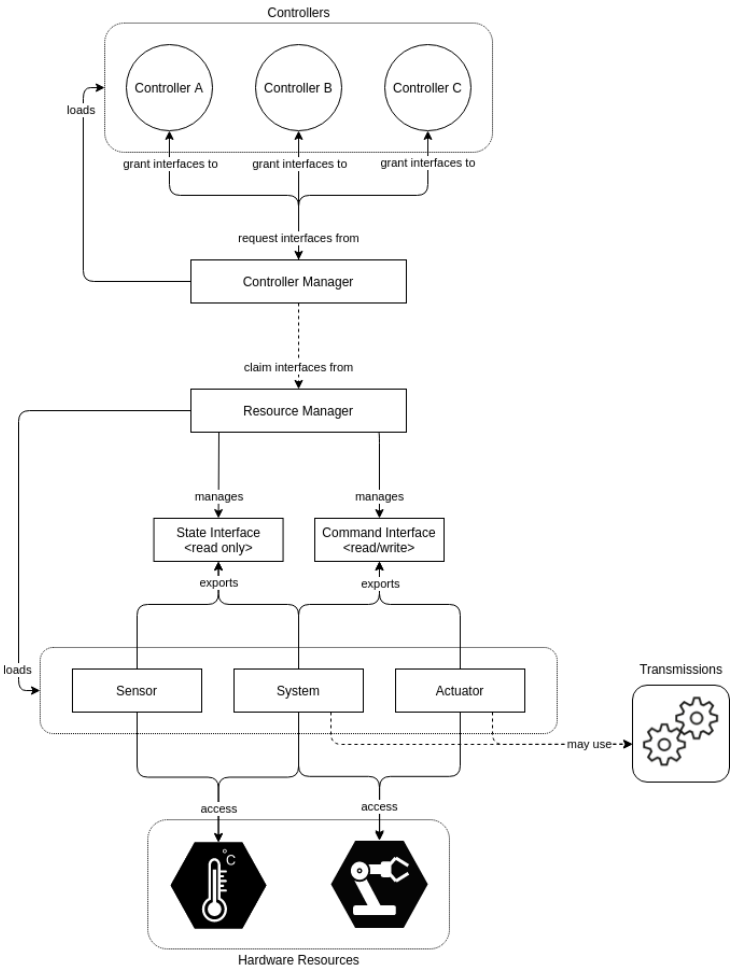


Figure 1.3: Architecture of ros2_control framework

1.2.2 Data Exchange with the SIH Hand

Communication with the SIH hand is established through an RS485 serial interface configured with a baud rate of 1 MBaud, 8 data bits, 1 stop bit, and no parity bit. Moreover, the internal firmware of the SIH hand validates the integrity of the frame by comparing the computed and received checksum values. Only if the values match will the message be processed further. The checksum is computed using the following algorithm:

```
checksum1 = 0;
checksum2 = 0;
for (i = 1; i <= 21; i++) {
    checksum1 = (checksum1 + input_frame_byte[i]) % 255;
    checksum2 = (checksum2 + checksum1) % 255;
}
```

Two types of data exchange are supported, cyclical and acyclic data exchange. For the precise format and byte structure of the cyclic and acyclic frames, see the Appendix B for details.

Cyclical Data Exchange

Cyclical communication is used to send joint commands and receive joint states at a fixed frequency. The SIH uses a predefined frame structure with a fixed length, composed of bytes, words, and double words. Each message must follow this format to be correctly interpreted by the internal firmware.

Data transmitted cyclically include actuator commands (e.g., target positions) and sensor feedback (e.g., encoder positions, motor currents). This exchange is handled inside the `read()` and `write()` methods of the hardware interface implementation.

Acyclic Diagnostic Communication

Acyclic messages are used for diagnostic purposes. The communication involves a request-response protocol. The output data frame for the request is fixed and filled up to the same length as for cyclic data exchange, while the length of the input data (diagnosis message) may vary. However the acyclic messages will not be used in ros2_control.

1.2.3 Integration Strategy

To integrate the SIH hand into ROS 2, I implemented a custom class inheriting from hardware_interface::SystemInterface, as requested by my internship supervisor. The goal was to replace the simple Python script originally provided by the manufacturer with a more robust and ROS 2-compliant C++ implementation. In this class, I defined the `init()`, `read()`, and `write()` methods to handle low-level serial communication with the device via the RS485 interface. This enables direct communication between the ROS 2 hardware abstraction layer and the SIH hand. A demonstration of this integration is available in the following video.

To enable compatibility with ros2_control, a URDF description of the SIH hand was required. Since no official URDF model was provided for this prototype, I started from an existing URDF of the Schunk SVH hand, which has a similar structure but differs in the number and nature of its degrees of freedom [6]. I adapted this URDF by reducing the number of joints and simplifying the mechanical structure so that it would accurately reflect the capabilities of the SIH prototype I was working with. This URDF allowed me to define the necessary joint interfaces and to link them with the hardware interface implementation. The final structure of this adapted model is shown in the Appendix B.3.

1.3 Hardware Limitations and Project Shift

During my development, I attempted velocity control on the SIH hand, a feature that was not supported in the original Python code provided by the manufacturer. Unfortunately, this test resulted in a mechanical blockage when two fingers locked against each other. As the device is a prototype, the damage could not be repaired. This experience highlighted the importance of staying within validated and safe operating conditions when working with prototypes. Unlike industrial products, prototypes are often insufficiently tested and inherently more prone to failure or physical damage.

Following this incident, I decided to stop further development on the hand. However, I ensured that future work could resume smoothly by leaving a clean and well-documented Git repository for the university. This repository includes a checklist detailing completed tasks and remaining work. At the current state, I had developed a functional C++ hardware interface and a custom URDF model adapted for the SIH hand. Some integration steps with ros2_control remained to be finalized. Additionally, I had not yet started implementing a ROS controller node that would allow high-level control of the hand via ROS topics.

Chapter 2

Soft Robotic Manipulation with the QB SoftHand

2.1 Overview of the Soft QB Hand

Due to an unrecoverable issue with the Schunk Hand, I shifted my focus to integrating a different robotic hand: the qb SoftHand developed by qbrobotics. This device offers a significantly simpler design, relying on a single actuator and a tendon-driven mechanism to control all fingers simultaneously. While it lacks the individual finger control of more complex hands, its soft, adaptive structure makes it ideal for robust and safe grasping in a variety of scenarios. The hand is lightweight and compact, yet capable of conforming to diverse object shapes, making it a suitable candidate for quick integration and testing in collaborative robotic applications.

Unlike the previous hand, this time the manufacturer already provided a working ros2_control interface, so I did not need to implement it myself [7]. While it was technically possible to develop a custom controller, I chose to rely solely on the existing one and interact with it using standard ROS topics.

Figure 2.1: Qb SoftHand

2.2 Basic Testing with Topics and Python Scripts

To evaluate the QB Hand in a straightforward and flexible way, I implemented a custom Python script named `control_speed.py`, designed to interface directly with the ROS 2 topics provided by the trajectory controller. This method allowed me to bypass the need for custom controllers while enabling precise control over the single degree of freedom (DoF) of the hand.

The `control_speed.py` script combines trajectory publishing and feedback monitoring. The user specifies a desired target position and a motion speed (expressed in units per second,1 unit corresponds to the full range of motion required to completely open or close the hand.). Based on the current position of the hand, continuously read from the `/state` topic, the script computes an execution time using the formula:

$$\text{execution\_time} = \frac{|\text{target\_position} - \text{current\_position}|}{\text{velocity}}$$

For example, a velocity of 1.0 moves the hand from 0 to 1 in exactly one second, or moves the hand from 0.5 to 1 in exactly 0.5 second.

An other feature of this script is the integrated "Stop" function. When triggered, the current joint position is resent as a new trajectory goal with immediate execution, effectively stopping the hand in place. This functionality was previously tested in a separate file named `stop_on_command.py`, but it is now fully integrated into `control_speed.py`.

This approach proved sufficient for prototyping and manual testing. The dynamic speed control and responsive stopping mechanism allowed for safe and intuitive operation of the QB Hand. A demonstration of this script is available at the following link: `https://youtu.be/ug0_QMClkMg`.

2.3 Practical Limitations for Further Integration

Controlling the QB Hand solely via ROS 2 topics presents several limitations. Primarily, this approach lacks real-time guarantees, which can lead to delays or jitter in command execution, impacting the responsiveness and precision of the hand. Without a dedicated hardware interface integrated into the ros2_control framework, it is difficult to leverage standardized controllers, manage resource conflicts, or synchronize commands with feedback effectively.

Despite these limitations, I chose to control the QB Hand exclusively through ROS 2 topics because the hand only has a single degree of freedom, making a full hardware interface and custom controller unnecessarily complex for this prototype. Using topics allowed for a faster initial integration and simpler implementation, which was sufficient for basic open and close commands. This approach also reduced development time and complexity while providing a functional control method suitable for the current scope of the project.

Chapter 3

Onboard 3D Perception and Object Frame Estimation

3.1 Setup and Use of the Intel RealSense D435i

In this project, an RGB-D camera is essential for detecting and localizing the objects to be grasped. The Intel RealSense D435i was selected for this task due to its robust depth sensing capabilities and its compatibility with ROS 2. This camera provides reliable depth images, which are used to estimate the 3D position of target objects in the environment.

The RealSense D435i benefits from an official ROS 2 wrapper provided by Intel, which significantly simplifies its integration. After installing the realsense2_camera package and launching the camera node, all necessary topics were immediately available and visualizable in RViz2. Among the available topics, we specifically focused on /camera/depth/color/points, which publishes the depth data in the form of a sensor_msgs::msg::PointCloud2 message [8].

This point cloud stream is central to our object detection pipeline. It is processed using the Point Cloud Library (PCL) in a custom C++ ROS 2 node. PCL offers a wide range of tools for filtering, segmenting, and analyzing point clouds, which we leverage to isolate and extract relevant object information [9].

Thanks to the existing ROS 2 support and the powerful features of the RealSense D435i, the setup phase was efficient, allowing us to quickly move on to the development of perception and detection algorithms.

Figure 3.1: RealSense D435i

3.2 Point Cloud Filtering: Basic vs Advanced Approach

3.2.1 Basic Point Cloud Filtering

To extract objects of interest from the depth data provided by the RealSense D435i, we implemented a basic yet effective point cloud filtering pipeline using the Point Cloud Library (PCL). This pipeline consists of three main steps:

Filtering

The raw point cloud received from the camera often contains a large number of points, including irrelevant data outside the robot's workspace. To reduce noise and computational load, we first apply a pass-through filter to retain only the points within defined spatial boundaries (in x, y, and z). A voxel grid filter is then used to downsample the point cloud by reducing its resolution, which helps improve processing speed while preserving the shape of the objects.

Plane Segmentation

Most of the scenes captured by the camera include a table surface on which the objects rest. To remove this dominant plane, we apply RANSAC (Random Sample Consensus) plane segmentation. This method fits a planar model to the point cloud and identifies the inlier points that belong to the table. These points are then removed from the cloud, leaving only the non-planar components—typically the object itself.

Clustering

Once the table is removed, several disconnected groups of points may remain. We apply Euclidean Cluster Extraction to group nearby points into clusters. Among the detected clusters, we select the largest one, which we assume corresponds to the target object to grasp. This assumption holds true in our experimental setup, where only one object is present at a time.

This basic approach provides a fast and reliable method for object isolation, suitable for real-time applications and compatible with low-resource environments.

3.2.2 Advanced Approach with Machine Learning

In parallel, we also experimented with a more advanced, machine-learning-based filtering approach. This solution was developed in collaboration with a PhD student, who provided a containerized service capable of processing point cloud data and segmenting objects more robustly, especially in cluttered environments.

The service is hosted in a container and exposed as a callable endpoint, allowing us to send point cloud data and receive refined 2D object masks. While functional, this method was not extensively used in the project due to its high computational requirements and latency—around 0.5 seconds per image—which made it less suitable for real-time use on our current hardware setup.

Nonetheless, this advanced pipeline shows promise for future developments where more complex scenes or higher precision are required.

3.3 Object Detection Using PCA

Principal Component Analysis (PCA) is a mathematical tool used to determine the main directions of variation within a set of 3D points—in our case, the point cloud representing the object. The first step is to compute the centroid of the point cloud and subtract it from all the points to center the data. Then, we calculate the covariance matrix of the centered point cloud and extract its eigenvectors and eigenvalues. The eigenvectors define the principal axes of the object, sorted by decreasing eigenvalues, which represent the amount of variance along each direction. The first eigenvector points in the direction of the longest dimension of the object (often aligned with its main length), the second corresponds to the secondary spread, and the third is orthogonal to the first two. These three orthogonal vectors form a local coordinate frame centered on the object's centroid. However, in practice, PCA may produce unstable axis directions due to symmetry or noise, for example, when detecting a symmetrical object like a box. To stabilize the orientation

of the object's frame over time, we enforced a consistent direction for the Y-axis. Specifically, in the implementation, we apply the following condition:

```
if (secondary_axis[1] > 0) {
  secondary_axis = -secondary_axis;
}
```

This ensures that the Y component of the secondary axis always points downward in the camera's coordinate system, preventing sudden flips of the frame during motion or visualization. This ensures smoother behavior for downstream tasks such as grasp planning.

This logic is implemented in a custom ROS 2 package named `pca_realsense_pkg`[1], which I developed during the project. The package is versioned on Git and includes detailed installation and usage instructions. It also supports parameter tuning through a .yaml configuration file, allowing users to adjust key processing parameters without modifying the code. This contribution provides a reusable and easily configurable module for future developments in the team.

3.4 Design and Mounting of a Custom Camera Support

To ensure accurate perception, the Intel RealSense D435i had to be positioned at an appropriate distance from the object to be detected. According to the camera specifications, a minimum distance of around 25 cm is required for reliable depth measurements. To satisfy this constraint while maintaining a clear and stable view of the scene, we decided to mount the camera directly on the wrist of the robotic hand.

This setup required the design of a custom 3D-printed camera support. The design needed to meet several key criteria: it had to maintain the correct distance and orientation of the camera relative to the hand, be printable without complex overhangs or support structures, and allow easy modifications if needed. To achieve this, I opted for a modular design composed of three separate parts. The first part ensures a solid mechanical interface with the robotic hand. The second part defines the camera's position and angle, ensuring the required 25 cm spacing is respected. The third part serves as a reliable mounting base for the camera itself.

By separating these functions into modular components, the system allows quick adjustments, by simply reprinting one of the parts, without redesigning the entire support. This approach provided flexibility during the testing phase while keeping manufacturing time and complexity low.

Figure 3.2: Camera support

[1] `https://gitsvn-nt.oru.se/mele/pca_realsense_pkg.git`

Chapter 4

Compliant Control of the Franka Emika Panda Arm

4.1 Overview and Setup of Franka Emika Panda Arm

The Franka Emika Panda arm is a 7-DoF torque-controlled robotic manipulator specifically designed for research applications. Its modular design, precision, and integrated torque sensing capabilities make it particularly suitable for compliant control and advanced manipulation tasks. The manufacturer provides a dedicated C++ library, libfranka, which allows low-level control of the robot and serves as the foundation for integrating the arm into ROS-based systems [10].

Figure 4.1: Franka Emika Arm

To establish communication with the arm, it must be physically connected via an Ethernet cable. Once connected, the robot's web interface can be accessed using its IP address. This interface is crucial for managing the robot's safety settings. From there, users can lock or unlock joints and enable external control via ROS by activating the so-called "Franka Control Interface" (FCI). The robot also features two physical buttons: one for triggering the emergency stop (E-Stop), and another to switch into manual guidance mode, allowing the user to move the arm by hand.

The initial setup and basic testing were straightforward, thanks to the official packages and tools provided by Franka Emika. These packages allow simple motion tests and joint control directly from ROS 2 nodes. However, full integration of Franka's native controllers into the ROS 2 ecosystem remains limited. For this reason, we chose to use the cartesian_compliance_controller,

a community-supported ROS 2 controller, to implement compliant Cartesian control of the end-effector [11]. This approach ensures better compatibility with modern ROS 2 workflows while maintaining the precision and responsiveness required for our manipulation tasks.

4.2 Introduction to Cartesian Compliance Control

For the control of the Franka Emika Panda arm, we chose to implement a Cartesian compliant control strategy. This approach allows us to command the end-effector in Cartesian velocity while also defining its compliant behavior in response to external forces. Thanks to the torque sensors embedded in each joint of the Panda arm, it is possible to estimate external forces and react to them in real time.

Compliance refers to the robot's ability to yield or adapt its motion when interacting with the environment, rather than strictly following a predefined trajectory. In a compliant control framework, the robot behaves like a virtual mass-spring-damper system: when it encounters an obstacle or an object, it does not resist rigidly but instead adapts its motion according to a defined stiffness and damping profile. This is especially useful in manipulation tasks where physical interaction with the environment is expected.

In our use case, compliance enables the robot to approach and gently make contact with an object—such as when attempting to grasp it—without triggering a collision error. This is particularly important when dealing with uncertain object positions or in scenarios that require precise and safe interactions.

4.3 Testing and Integration of the Controller

To integrate and test the Cartesian compliance controller with the Franka Emika Panda arm, I developed a complete ROS 2 package tailored for this task. The package was designed not only to facilitate experimentation during the project but also to serve as a reusable and configurable framework for future research. In this section, I describe the key components and features of this integration.

4.3.1 Interactive Control Using RViz Marker

A core feature of the package is the ability to control the end-effector interactively via RViz. This is done using an interactive marker that can be moved in real-time by the user. The target pose of the end-effector is published as a `geometry_msgs::msg::PoseStamped` on the topic `/cartesian_compliance_controller/target_frame`. The Cartesian compliance controller then commands the robot to follow this target frame while applying compliant behavior. This setup provides a user-friendly way to test the controller's performance and responsiveness directly from RViz, without requiring any pre-defined trajectory or motion planner.

4.3.2 Force-Torque Bridge and Gravity Compensation

To fully enable compliance, the controller requires external force-torque measurements, which are normally provided by a sensor or simulation. For this purpose, I implemented a dedicated ROS 2 node that estimates and publishes the external wrench on the topic `/cartesian_compliance_controller/ft_sensor_wrench`. This node calculates the wrench based on robot state and gravity compensation, simulating what a real force-torque sensor would provide. The gravity compensation is especially important when tools (such as a robotic hand) are attached to the end-effector. The characteristics of the tool—its mass and center of mass—are defined in a configurable .yaml file. This allows for easy adaptation of the system to different end-effectors.

To better understand and tune this part of the system, I used PlotJuggler, which was very useful for visualizing ROS topics and debugging the wrench signal behavior.

4.3.3 Motion Smoothing and Speed Limiting Node

During testing, I observed that the robot could become overly reactive when the target frame moved too quickly or suddenly. Simply tuning the PID gains of the controller was not sufficient to ensure smooth and safe motion. To address this, I developed a motion smoothing node that limits the speed at which the robot can approach the target pose.

Instead of sending the final target directly, the node computes incremental intermediate poses that progressively bring the robot closer to the goal. The step size is configurable and prevents abrupt movements. The logic involves computing the distance to the target and generating a new pose by linear interpolation for position and spherical interpolation (slerp) for orientation.

This solution improved stability and safety during demonstrations and experiments, especially when controlling the arm through RViz.

4.3.4 Grasp Validation Service Based on Force Measurement

To assist with object grasp validation, I implemented a simple ROS 2 service that estimates whether a force is detected along the Z-axis of the world frame. The service returns true if the vertical force exceeds a predefined threshold, indicating that an object might be held by the robot.

While the idea is functional, the sensor readings can be noisy. I observed fluctuations equivalent to approximately ± 400g, which limits the usability of this method for small or lightweight objects.

4.3.5 Contribution and Documentation

All nodes and utilities described above are included in the custom ROS 2 package I developed. The package is versioned on Git and includes detailed instructions for installation, configuration, and usage[1],. Parameters such as tool mass, compliance settings, and speed limits can be easily adjusted via .yaml files. This work aims to help future students and researchers—particularly those migrating from ROS 1 to ROS 2—by providing a ready-to-use, modular control framework for compliant manipulation with the Franka Panda arm.

[1] `https://gitsvn-nt.oru.se/mele/franka_compliance_pkg.git`

Chapter 5

Full Integration: Perception-Guided Grasping

5.1 Unified Robot Description and Visualization

To enable perception-guided grasping, I developed a complete and coherent robot description by merging the individual models of the arm, the hand, and the camera into a single URDF structure. This integration was essential to ensure proper visualization and coordination between all components in RViz2 and for future control pipelines.

The process began by modifying the original URDF of the Franka Emika Panda arm, to which I added predefined named poses for simplified control and debugging during grasping sequences. For the qb SoftHand, I extended its URDF by attaching a 3D camera model (an Intel RealSense D435i), which is physically mounted on the wrist. These modifications were made independently in their respective URDFs.

Next, I reused both individual URDFs inside a unified Xacro file, which combines the arm and the hand-camera subsystem into one complete robot model. This unified description is then passed to the robot_state_publisher node, allowing the system to publish a coherent TF tree that includes the base frame, the arm, the hand, and the camera.

This setup ensures a reliable and accurate robot representation in RViz2, where we can visualize the full kinematic chain and verify grasping configurations in real time.

5.2 Coordinated Joint State Publishing and TF Broadcasting

The system architecture separates control and visualization for robustness and modularity. Each subcomponent publishes its joint states independently. The Franka arm is controlled through a custom node, franka_control2_node, which uses the robot_description_arm and remaps its joint state output to the topic `franka/joint_states`. The qb SoftHand and camera are handled by ros2_control_node under the namespace qbhand1, using the robot_description_hand_camera and publishing joint states to `qbhand1/joint_states`.

To unify these joint state sources, I configured a single joint_state_publisher node with the following parameter:

```
source_list: ['franka/joint_states', 'qbhand1/joint_states']
```

This configuration merges the joint states from both systems and republishes them as if they originated from a single robot. As a result, robot_state_publisher, which consumes the joint data, can compute the transforms for the entire robot described by the unified URDF.

This design enables separate control of the arm and the hand, each using its dedicated URDF and control interface. By merging their joint states, the full kinematic chain can still be visualized consistently in RViz2 through a unified URDF description.

5.3 Integration Process: From Partial to Full System

5.3.1 Initial Setup: Hand and Camera Only

The first stage of integration focused on assembling and testing the Qb SoftHand with an Intel RealSense D435i camera, excluding the robotic arm. This allowed validation of the grasping logic without the complexities introduced by arm control.

To begin with, the camera was physically attached to the hand, and I used the hand-camera URDF. In this model, I manually introduced a hand_frame, positioned relative to the hand's URDF. Then, inside the camera node that publishes the detected object frame, I added a target_hand frame, defined along the object's y-axis and offset by an arbitrary distance. This new frame served as an estimated grasp target for the hand. This frame would later serve as the goal pose to determine when the hand should close on an object.

A basic prototype node was developed to trigger grasping based on frame alignment: when the hand_frame and the target_hand frame became sufficiently close in both position and orientation, the controller would send a command to close the fingers. Despite being a temporary and simplified control strategy, this allowed for early experimental trials with object grasping. These tests confirmed that the grasping logic and hand-camera coordination were functional in isolation. A demonstration video is available here: `https://youtube.com/shorts/Lrl9Op34L_o?feature=share`.

At this stage, the Franka arm was not yet included in the system. By limiting the test setup to the hand and camera, it was possible to rapidly iterate on the grasp detection pipeline, frame definitions, and control signals without additional interference or complexity from the arm's dynamics or its own controllers. This minimal setup played a critical role in exposing early integration challenges in a controlled environment.

5.3.2 Adding the Arm: Coordination Challenges

Once the grasping logic with the hand and camera alone had been validated, the next step consisted in integrating the Franka arm into the system. This integration involved computing a target_arm pose derived from the previously defined target_hand frame, in order to guide the end-effector of the robot arm toward the desired grasp configuration.

However, this new setup introduced several challenges.

The first issue arose from the spatial offset between the target_hand and the corresponding target_arm frame. Since the arm's grasping point is located several centimeters away from the visual target defined near the object, even minor instabilities in the target_hand frame led to amplified, jerky motions of the arm. This instability made smooth motion planning difficult and unreliable. To mitigate this, I implemented a two-stage filter: a low-pass filter to smooth small fluctuations over time, and a threshold-based jump filter to reject sudden, unrealistic changes in the target pose.

The second major difficulty was related to the definition of the orientation of the target_hand frame, especially when dealing with symmetrical objects like spheres or cylinders. Because the orientation was computed via PCA on the object's point cloud, perfectly symmetric shapes often led to unstable or undefined principal axes. This caused the grasp frame to rotate unexpectedly, particularly while trying to approach or interact with such objects.

To address this, I implemented a shape classification function that estimated whether the detected object was a sphere, a cube, or a cylinder. Although the cylinder detection proved unreliable and was eventually unused, the system was able to distinguish spheres from other shapes with sufficient accuracy. When a sphere was detected, a specific logic was triggered to stabilize the frame and avoid unnecessary rotation.

The computation of the grasp frame orientation followed a well-defined objective:

- The $\mathbf{y}_{\text{target\_hand}}$ axis was aligned with the object's main axis, obtained from PCA and normalized.
- The $\mathbf{z}_{\text{target\_hand}}$ axis was defined to point toward the center of the object while remaining orthogonal to $\mathbf{y}_{\text{target\_hand}}$. To minimize the angle with the vertical world axis $\mathbf{z}_{\text{world}}$, the latter was projected onto the plane orthogonal to $\mathbf{x}_{\text{object}}$ and passing through the center of the object. The resulting vector was then normalized and reversed to define $\mathbf{z}_{\text{target\_hand}}$.
- In the rare cases where $\mathbf{y}_{\text{target\_hand}}$ was nearly aligned with $\mathbf{z}_{\text{world}}$, a fallback strategy was used to minimize the angle with $\mathbf{y}_{\text{world}}$ instead.
- The $\mathbf{x}_{\text{target\_hand}}$ axis was computed as the cross product $\mathbf{x}_{\text{target\_hand}} = \mathbf{y}_{\text{target\_hand}} \times \mathbf{z}_{\text{target\_hand}}$, and finally, $\mathbf{z}_{\text{target\_hand}}$ was recomputed to enforce orthonormality.
- The final frame was then translated along the $\mathbf{z}_{\text{target\_hand}}$ axis to ensure a safe offset from the object. The translation distance corresponds to half the object's width plus an additional configurable offset defined in the YAML configuration file.

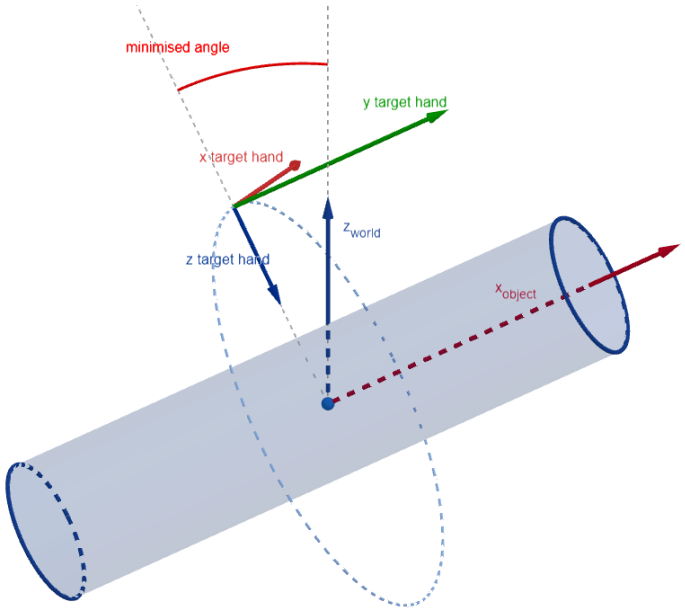


Figure 5.1: The computation of the grasp frame

Finally, another problem emerged during motion: as the arm began to move toward the object, vibrations or partial occlusion from the hand itself would occasionally disturb the object detection. This caused instability in the perceived target frame and sometimes triggered erratic motion. To prevent the robot from reacting to corrupted data, I implemented a safety mechanism that constrained the arm to move only when object_frame remained within a defined workspace zone and remained stable over a short time window. This approach allowed reliable grasp execution without requiring a complex inverse kinematics solver, which would have added unnecessary integration complexity at this stage.

5.3.3 Lessons Learned

This early prototyping phase was very helpful to identify several important issues before moving to the full system.

First, I noticed that sensor feedback could be unstable, especially when the object moved or when the arm itself caused vibrations or occlusions. This made the grasp pose unreliable without proper filtering.

Second, I realized that grasp orientation was not always clear, especially for round or symmetrical objects like spheres and cylinders. I had to add logic to recognize object shape and define a stable orientation for the grasp frame.

Moreover, there was a problem with synchronizing perception and actuation. Sometimes the robot would react to outdated information, especially if the grasp frame moved too quickly or suddenly.

Finally, working with a simple setup made it much easier to test and fix problems one by one. It allowed me to prepare the system step-by-step before moving to the full integration with the robot arm and the state machine.

5.4 Finite State Machine (FSM) Implementation

To coordinate the different stages of the robotic manipulation process, I implemented a Finite State Machine (FSM) in a dedicated ROS 2 node named `robot_fsm_node`. This FSM controls both the arm and the hand of the robot, based on feedback from TF frames, joint states, and external services. The main objective was to define a reactive and modular behavior for object grasping, leveraging the real-time feedback from the perception system.

5.4.1 State Description

The FSM consists of the following four states:

- **ToSecurityPose**: the robot moves to a predefined safe pose, using a TF frame named `target_security`. This pose ensures that the arm is away from obstacles before attempting any grasping operation.
- **ToTargetPose**: the robot attempts to reach the pose defined by the `target_arm` frame, which corresponds to the desired grasp position in front of the object. If the pose is not valid (i.e., outside the workspace or misaligned), the system returns to the security pose.
- **ClosingHand**: once the grasp pose is reached, the hand is closed to attempt grasping the object. The system monitors the joint position of the hand to confirm the closure.
- **ToLiftedPose**: after grasping, the robot lifts the object to a new pose defined by the `target_lift` frame. Once the lift is complete, the grasp is validated using a service that checks the force along the $\mathbf{z}_{\text{world}}$ axis. Finally, the hand is opened again to release the object and return to the target pose if the cycle is to continue.

5.4.2 Implementation Details

Each transition between states is driven by a timer callback running at 10 Hz. The FSM uses TF lookups to check whether the robot has reached the desired pose, based on both position and orientation thresholds. It also monitors the hand's joint state to confirm grasping and uses a ROS service to validate whether the object is securely held. The use of TF frames such as `target_security`, `target_arm`, and `target_lift` allows flexible spatial configuration through external perception modules.

Parameters such as the workspace limits and the alignment threshold are configurable via a YAML file, allowing the FSM to be adapted easily to different setups.

This FSM allowed modular testing of each phase of the manipulation process. It facilitated debugging and tuning by isolating perception, motion, and grasping into well-defined

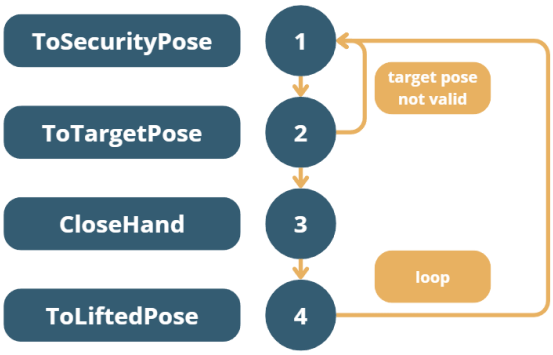


Figure 5.2: FSM

states. This design also prepared the system for future extensions, such as failure handling, retries, or multi-object picking strategies. This logic is implemented in a custom ROS 2 package named `franka_compliance_pkg`[1]. A demonstration video is available here:`https://youtu.be/fYOYV-FDOlU`.

[1] `https://gitsvn-nt.oru.se/mele/franka_compliance_pkg.git`

Chapter 6

Designing a Custom Robotic Hand (Yale OpenHand Model O)

6.1 Project Scope Adjustment

During the integration of the qb SoftHand, a tendon rupture occurred. This issue can be explained by the fact that the hand was several years old, and the tendon had become frayed over time. Given the prohibitive cost of repair and the risk of similar failures, the decision was made to reorient the project towards building a new hand based on the Yale OpenHand Model O design [12].

The Yale Model O is a three-fingered robotic gripper that, like the qb SoftHand, relies on tendon-driven actuation. However, in contrast to the qb SoftHand where all fingers are coupled, each finger of the Model O can be controlled independently. In addition, the two adjacent fingers are mounted with a relative rotation, resulting in a total of four degrees of freedom. This design provides greater versatility: the hand can grasp large objects using thumb opposition, but it can also perform precision tasks by exploiting the relative rotation of the paired fingers, enabling two-finger pinch grasps for smaller objects.

By adopting this design, the laboratory gains a low-cost and robust platform that enables continued research on robotic hands with more than two fingers, while also expanding the range of feasible manipulation strategies.

6.2 Model Adaptation for European Standards

Since the Yale OpenHand Model O was originally developed in the United States, all of its components and CAD models were designed using imperial dimensions [13]. Furthermore, most of the original hardware (e.g., dowel pins, screws) is only commercially available in the U.S. To make the design compatible with European suppliers, the CAD model had to be adapted to the metric system. This involved resizing holes to accommodate standard metric screws (M2, M3, etc.) and adjusting dowel diameters.

In parallel with these modifications, I was responsible for preparing a complete bill of materials for my supervisor, I focused on two websites, RS and Mouser to make ordering easier. Even after 3D printing the first version, several 3D parts had to be reworked to properly fit the newly selected components. While additional time spent on CAD refinement could have reduced the need for reprints, the project was constrained by a strict four-week timeline to deliver the robotic hand. Therefore, rapid iterations and early ordering of components were prioritized over perfect upfront modeling.

Another important aspect was the actuation system. The original Yale design relies on Dynamixel XM430-W350 motors, which we were also able to procure in Sweden. Afterwards, powering and controlling four motors in daisy-chain configuration required careful consideration

of the electrical setup. After evaluating the expected current draw, the final solution consisted of a Power Hub supplying 12 V at 5 A, combined with a U2D2 interface to enable communication between the motors and the control PC. This configuration provided both sufficient power and reliable communication for the new hand. In practice, the finger flexion motors are expected to draw up to 1.5 A each, corresponding to a peak torque of approximately 2.45 N·m, to give an idea this torque corresponds approximately to holding a 2.5 kg weight at the end of a 10 cm lever. In contrast, the rotation motor requires only minimal current due to the low-friction design; experiments showed that as little as 0.03 A was sufficient to operate it reliably. To ensure that these limits are not exceeded, it can be set using Dynamixel Wizard 2.0. [14] B.3.

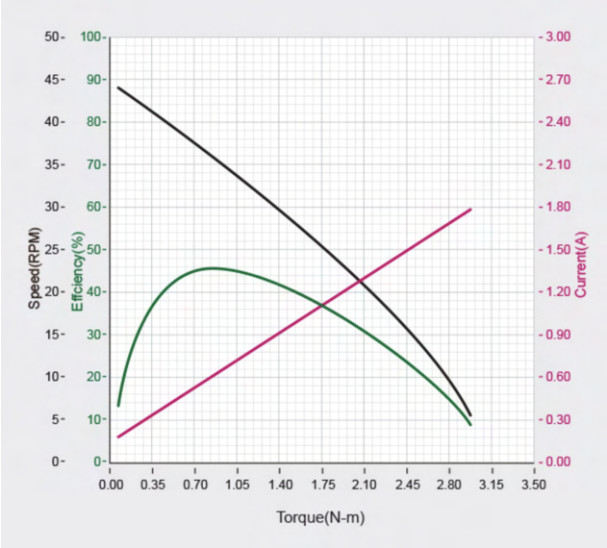


Figure 6.1: Current vs Torque characteristics

Eventually, a custom enclosure was added to the side of the hand to host the Power Hub. The decision to integrate the Power Hub directly with the hand was motivated by communication reliability: over longer distances, RS-485 cables can become prone to signal degradation. By embedding the power distribution hardware close to the motors, the overall setup becomes more compact, reduces the risk of communication errors, and simplifies cabling during experiments.

Figure 6.2: 3D model of the box

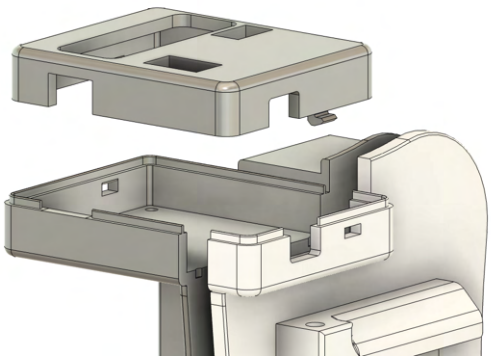


Figure 6.3: Integration of the box into the structure of the hand

6.3 Mechanical Assembly and Motor Wiring

Preparation of parts

Once all the 3D-printed and purchased components were prepared, the mechanical assembly phase could begin. Achieving a functional hand required particular attention to tolerances and surface finishes. Several holes had to be reworked using a drill press to ensure tight fits where rigidity was needed, while preserving clearance in specific joints to allow smooth finger flexion and extension. In addition, some surfaces required polishing to obtain the smoothness necessary for low-friction rotations and gear meshing.

During this phase, special attention was also given to the finger extension springs. Due to limited options from suppliers, it was necessary to adapt the springs that were available. A single spring size was iteratively cut and tested on a prototype finger to achieve two functional characteristics: one spring strong enough to resist gravity for the first phalange, and a second slightly stiffer spring so that the last phalange would close in a second stage. This iterative approach ensured that the existing spring stock could be effectively repurposed to achieve the desired finger motion sequence.

For the fingers, a moulding process was used. A customised 3D-printed mould was designed and manufactured, into which resin was poured to create the flexible bodies of the fingers. This step was essential to reproduce the grip that fingers can have.

Figure 6.4: Moulding of the fingers using resin

Figure 6.5: All parts before assembly

Assembly

The final stage consisted of assembling all the components into a coherent mechanism, integrating the structural frame, finger modules, gears, and tendons [15]. The Dynamixel XM430-W350 motors were then mounted and wired in a daisy-chain configuration, following the electrical setup defined previously. This allowed the hand to be fully operational, both mechanically and electrically.

A custom mounting support was designed to attach the hand to the robot arm (cf. Figure 6.6). The support consists of two main parts: the first part fits onto the arm and is secured with two screws, while the second part is attached to the hand and slides into the first part. A long screw is then used to fasten both parts together, providing a stable yet easily detachable connection between the hand and the arm. The final assembly of the Yale OpenHand Model O is shown in Figure 6.7.

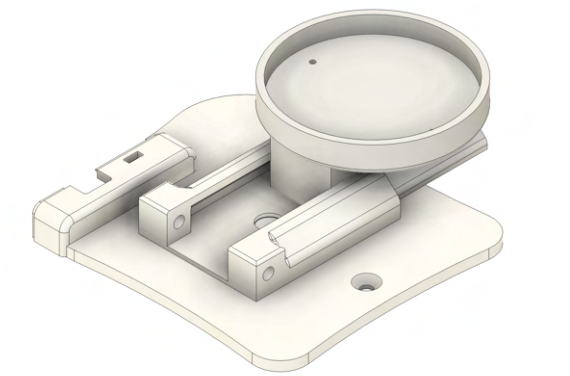


Figure 6.6: Attachment mechanism between the hand and the arm

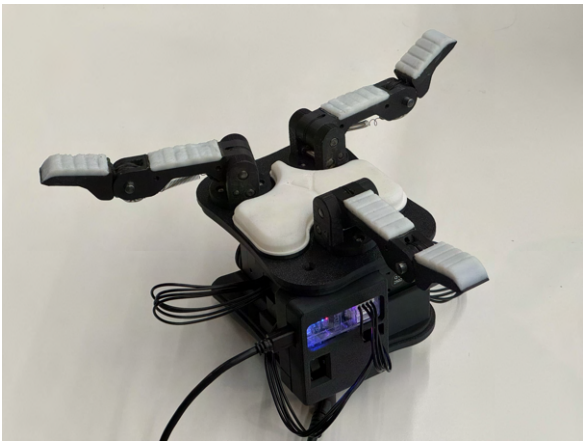


Figure 6.7: Final assembly of the Yale OpenHand Model O

Chapter 7

ROS 2 Control Interface for the Custom 3-Finger Hand

7.1 Preliminary Motor Testing

Before integrating the custom hand into the ROS 2 control framework, it was essential to gain familiarity with the Dynamixel motors and their available control modes. To this end, a standalone C++ node was developed using the official *Dynamixel SDK* provided by Robotis [16]. This library offers low-level access to the communication protocol of Dynamixel actuators and supports configuration, command transmission, and state reading. The node was connected to the four motors through the U2D2 USB interface in a daisy-chain configuration, which allowed sequential addressing of each motor over a single serial line.

This preliminary node provided a convenient test bench to evaluate the different control modes available in the Dynamixel XM430-W350 motors:

- **Position mode:** controls the motor to reach a target angular position within a limited range of motion (typically a single revolution).
- **Velocity mode:** directly sets the rotational speed of the motor shaft, without a specific position target.
- **Current mode:** regulates the output torque indirectly by controlling the current supplied to the motor.
- **Extended position mode:** allows continuous multi-turn operation by extending the position range beyond the standard 360° limit.
- **Current-based position mode:** similar to extended position control, but with the addition of a user-defined current limit to prevent excessive forces and protect mechanical components.

After testing all modes, the *current-based position* control was selected for this project. This mode combines the extended range of multi-turn operation with an adjustable safety limit on the applied torque. It therefore ensures both functional flexibility and mechanical protection, which are crucial for tendon-driven robotic hands. In addition to clarifying the most suitable control strategy, this standalone node also provided valuable practical experience in addressing, commanding, and monitoring multiple Dynamixel motors in daisy-chain configuration, laying the groundwork for the subsequent integration into ROS 2 Control.

7.2 Integration into ROS 2 Control

Once the motors were successfully tested in standalone mode, the next step consisted of integrating the hand into the ROS 2 control framework. This required adapting the preliminary C++ node into a standardized `ros2_control` hardware interface, defining the robot description in URDF format, and selecting an appropriate controller to execute commands. Together, these steps enabled seamless integration of the custom hand into the ROS 2 ecosystem.

7.2.1 Development of the Hardware Interface

The logic initially developed in the standalone test node was encapsulated into a dedicated hardware interface compatible with the `ros2_control` framework. This interface managed the communication with the four Dynamixel XM430-W350 motors using the Dynamixel SDK, while exposing a simplified control abstraction to higher-level controllers.

Commands were normalized between 0 and 1, representing the fully open and fully closed states of each finger joint. The hardware interface converted these normalized values into corresponding Dynamixel commands, taking into account the specific tick ranges of each motor. In the reverse direction, the interface provided motor feedback including position, velocity, and current readings.

This modular implementation not only simplified controller design, but also ensured that the interface could be reused or extended by future developers.

7.2.2 URDF Description of the Hand

To allow ROS 2 to understand the structure and capabilities of the custom hand, a URDF (Unified Robot Description Format) model was created. The description was organized into two distinct parts:

- **Structural model:** This portion of the URDF defines the physical links, kinematic joints, and associated meshes for the three-finger hand. It provides the structural hierarchy required for visualization in RViz and ensures that joint motions are represented consistently with the real hardware.
- **ros2_control section:** This portion specifies the control-related parameters of the system. It defines the four joints associated with the Dynamixel motors, their command and state interfaces, as well as the hardware-specific parameters such as the USB port, baudrate, motor tick ranges, and default current limits. By providing this information, the URDF establishes the link between the physical robot and the hardware interface implementation.

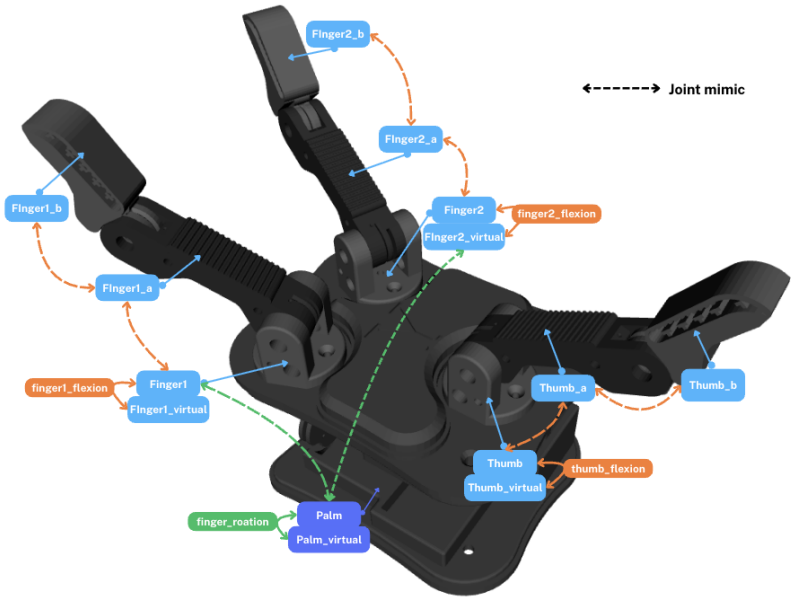


Figure 7.1: URDF Model of the Yale OpenHand

This dual structure enabled both accurate visualization and effective control integration. Mimic joints enable the command between 0 and 1 to be converted into radians using a multiplier coefficient.

7.2.3 Controller Selection and Launch Configuration

With the hardware interface and URDF in place, the final step was to select an appropriate controller within the `ros2_control` framework. Among the available options, the `JointTrajectory Controller` was chosen. This controller allows execution of position trajectories over time, thereby combining both position and implicit velocity control. Its integration was straightforward, as it is a widely used, pre-configured component of `ros2_control`.

This choice also ensured continuity with previous work on the qb SoftHand, where the same controller had been successfully deployed. The familiarity of its configuration and its capability to execute timed joint trajectories made it well suited for the three-fingered hand.

A dedicated launch file was created to bring together the hardware interface, URDF description, and trajectory controller. Once initialized, this setup provided a functional ROS 2 pipeline for sending commands to the hand and receiving real-time feedback, marking the transition from isolated motor testing to a fully integrated robotic system.

7.3 Final Deliverables and Reuse Potential

The outcome of this work is not limited to a functional prototype of the three-finger hand. Particular care was taken to ensure that the developments are reproducible, maintainable, and extendable by other researchers or students. This section outlines the repository organization, example packages, and the potential for reuse in future projects.

7.3.1 GitLab Repository Structure

All software developments were consolidated in a dedicated GitLab repository[1]. The repository follows a clear structure, with each package corresponding to a specific functionality:

- **Hardware interface:** Implements the `ros2_control` interface for Dynamixel motors, enabling normalized command inputs and safe state feedback.
- **URDF description:** Contains the structural model and control-related description of the hand, including meshes, joint definitions, and parameters.
- **Launch files:** Defines entry points for testing the hardware interface, visualizing the URDF in RViz, and ready-to-use configuration files for the `JointTrajectoryController`.

Each package is documented with a `README` file describing its purpose, usage instructions, and dependencies, ensuring that new users can quickly replicate the setup. A demonstration video is available here:`https://youtu.be/Dt2UnCSzhbA`.

7.3.2 Example Packages and Launches

In addition to the core packages, example packages were developed to illustrate integration with other robotic components[2]. The most significant demonstration consists of a launch configuration that simultaneously loads the Franka Emika arm and the custom three-finger hand.

Within this setup, an interactive marker was employed in RViz, allowing simultaneous control of the arm's Cartesian pose and the hand's grasp configuration. This interactive environment provides an intuitive interface for quickly testing grasping scenarios and validating the coordinated operation of both systems.

Compliance controller by providing such examples, the repository offers not only isolated hand control, but also a model for combining it with other robotic platforms in a modular manner.

7.3.3 Reuse Potential

Beyond the scope of this project, the deliverables represent a solid foundation for future research and student projects. The modular hardware interface, the URDF description, and the example controllers form a reusable base that can be adapted to different hardware setups or research objectives.

Potential applications include:

- Adapting the hardware interface to other Dynamixel-driven mechanisms.
- Extending the URDF description to incorporate additional sensors or end-effectors.
- Using the example packages as starting points for advanced experiments in manipulation, or grasping strategies.

By providing well-documented, modular, and reproducible software, this work ensures that the custom hand is not an isolated prototype, but rather a platform that can be leveraged and extended by the broader robotics community. A demonstration video is available here:`https://youtu.be/ZBFgaiWKOdc`.

[1] `https://gitsvn-nt.oru.se/mele/yale_hand_ros_pkg.git`
[2] `https://gitsvn-nt.oru.se/mele/franka_yale_hand_pkg.git`

Conclusion

7.4 Summary of objectives and contributions

The primary objective of this internship was to advance the integration of multi-fingered robotic hands into the laboratory's ROS 2-based control stack, to experimentally investigate whether initiating hand closure during arm motion (*in-motion preemption*) can outperform the traditional "reach-then-grasp" strategy. However, several hardware issues necessitated a redefinition of the internship objectives. While the first two months were largely devoted to pursuing the initial goal with minor adjustments, the focus during the final month shifted toward the design and construction of a three-fingered robotic hand, intended to serve as a platform for future research within the robotics laboratory.

This work delivered a set of concrete, reusable contributions: (i) hardware interface implementations compatible with `ros2_control` for the Schunk SIH, and a custom three-fingered hand (adapted from the Yale OpenHand Model O); (ii) URDF models and well-documented GitLab packages including example launch files and test scripts; (iii) a perception pipeline based on point cloud filtering, plane segmentation and principal component analysis (PCA) for object frame estimation; (iv) integration and testing of a Cartesian compliance controller for the Panda arm, including RViz-based interactive control, gravity compensation, a force/torque bridge and a grasp validation service; and (v) a prototype mechanical assembly of a three-finger hand together with preliminary ROS 2 control tests.

7.5 Critical assessment

The developed software components are modular and documented, which facilitates reuse by future students and researchers. The perception-to-grasp pipeline proved functional in controlled experimental scenarios and the compliance controller enabled safe, interactive tests with the Panda arm. However, the central hypothesis — that *in-motion preemption* can materially improve grasp performance — could not be conclusively validated within the available timeframe and experimental budget. Quantitative benchmarking (success rates, cycle times, robustness across object sets and pose uncertainty) remains to be performed. Hardware reliability issues experienced with the Schunk SIH and the QbHand necessitated scope shifts that reduced the time available for the main objective.

Immediate next steps should focus on standardizing an experimental protocol to compare *in-motion preemption* against the baseline strategy, including well-defined metrics and a diverse object set. Improving perception robustness using learning-based segmentation and pose estimation, reinforcing the mechanical reliability of the hands, and automating experiment runs will accelerate evaluation.

7.5.1 Personal and professional development

This internship meaningfully strengthened both my technical and professional skillset. I gained hands-on proficiency with ROS 2—especially the `ros2_control` framework—and learned to integrate it reliably with physical hardware while adopting a modular approach to software design

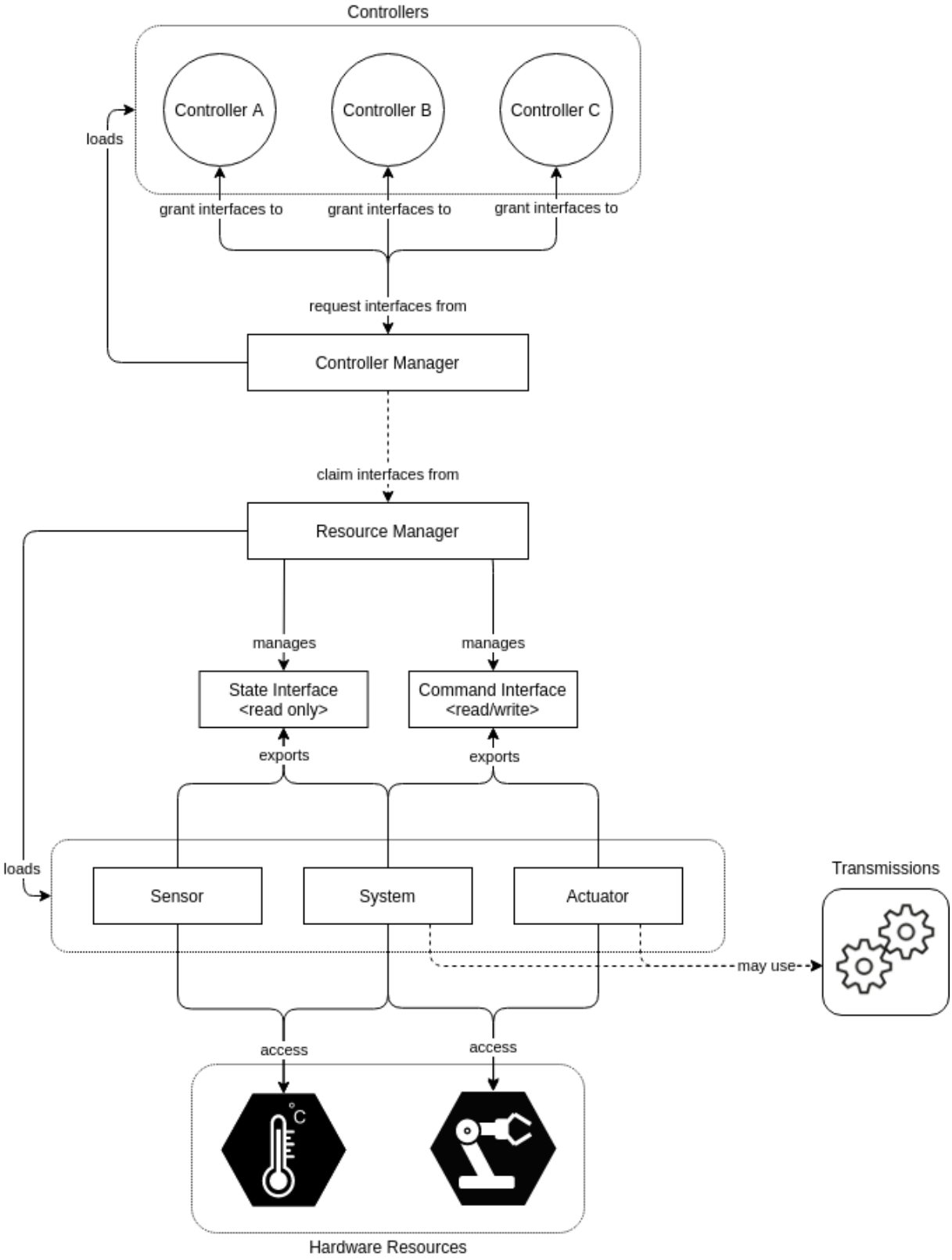
for easier reuse and maintenance. My CAD and mechanical-prototyping abilities improved, and I became more efficient at experimental troubleshooting, which shortened iteration cycles during integration. At the same time I developed greater autonomy by setting and meeting intermediate goals while keeping weekly meetings with my supervisor for guidance, and my technical English improved through regular documentation and collaboration. Working closely with a team largely composed of PhD students exposed me to the discipline and rigor of academic research, including the expectations around reproducibility and publishing. Together, these experiences have made me more organized, effective, and prepared for multidisciplinary projects in robotics.

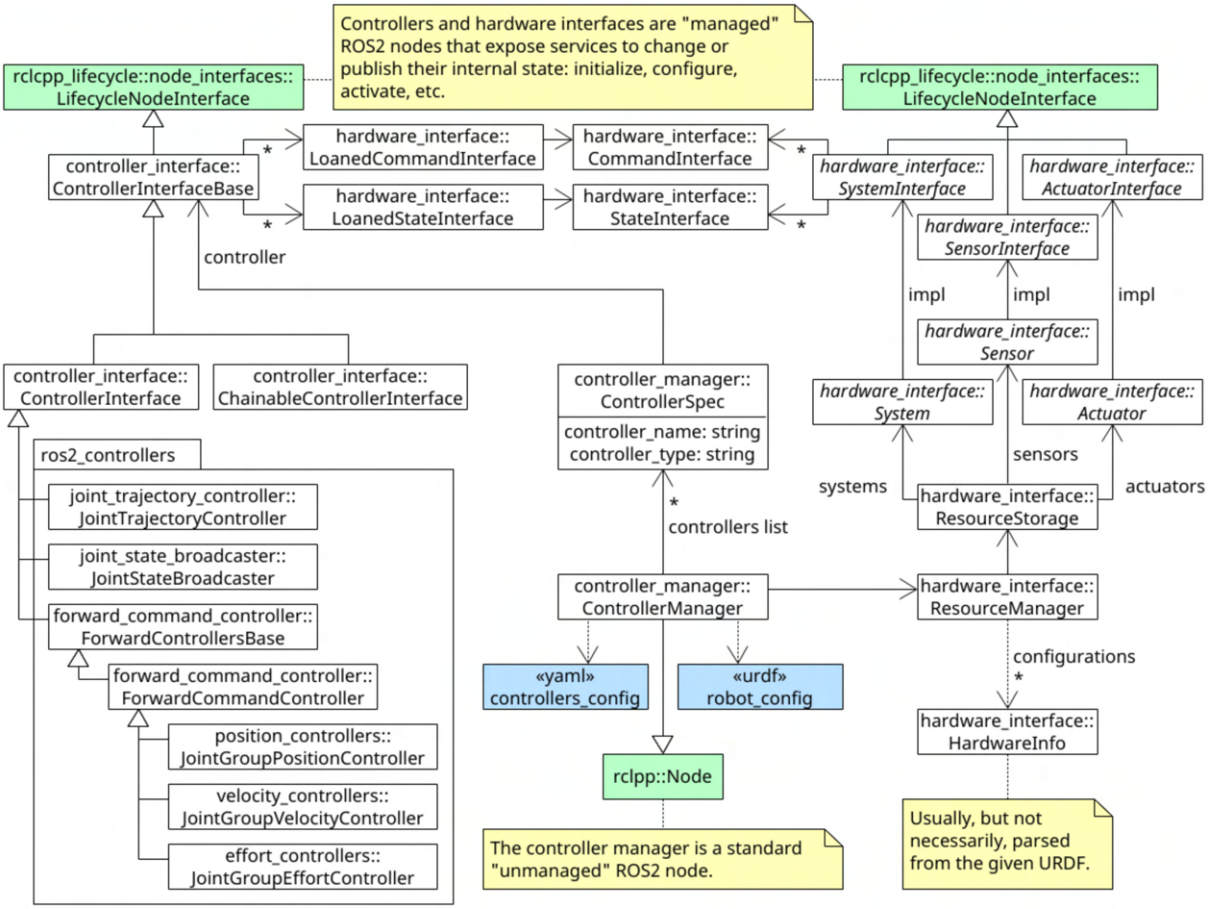
7.6 Closing remark

In sum, the project produced a robust software and hardware foundation for perception-guided grasping on compliant manipulators and outlined clear, actionable directions for validating and extending the core research hypothesis. The artifacts and lessons resulting from this internship form a reusable basis for future experimental work and potential thesis research within the laboratory.

Appendices

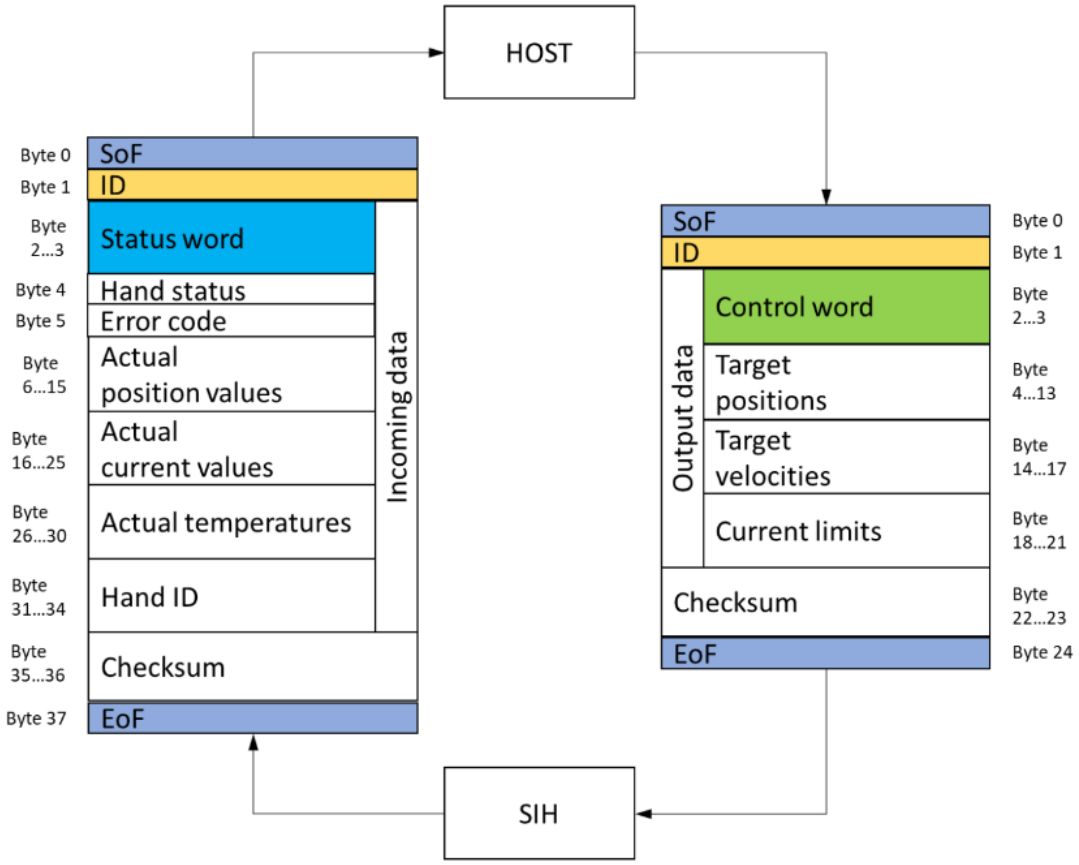
A Ros2_control



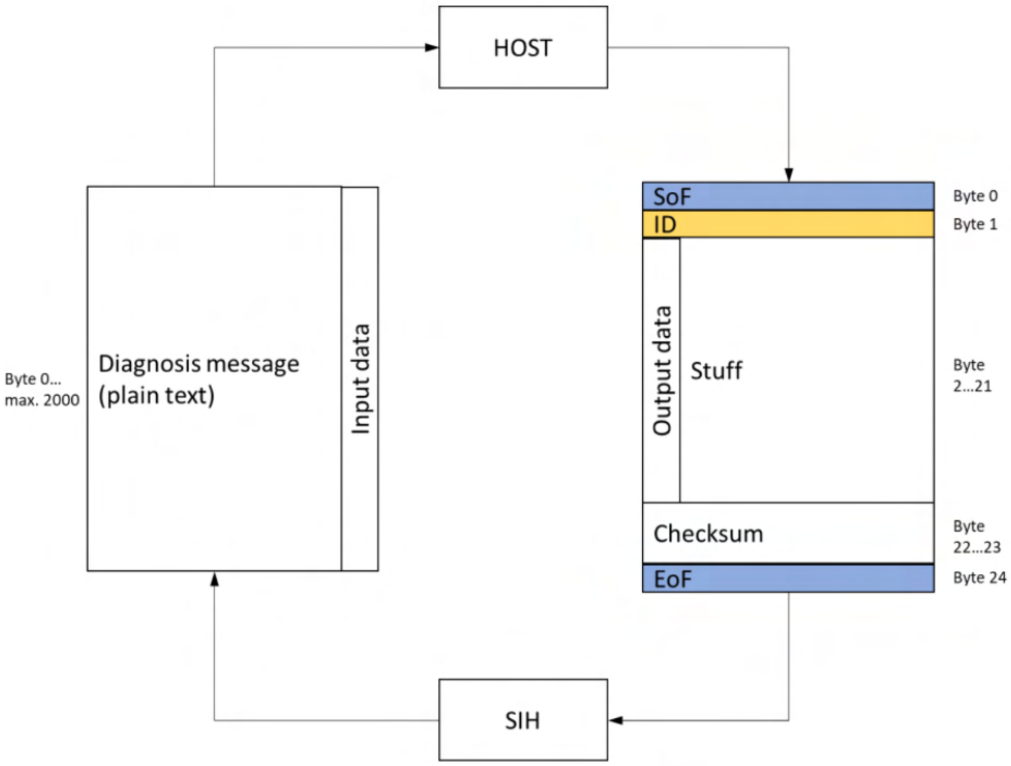

Controllers and hardware interfaces are "managed" ROS2 nodes that expose services to change or publish their internal state: initialize, configure, activate, etc.
rclcpp_lifecycle::node_interfaces::
LifecycleNodeInterface
rclcpp_lifecycle::node_interfaces::
LifecycleNodeInterface
controller_interface::
ControllerInterfaceBase
hardware_interface::
LoanedCommandInterface
hardware_interface::
CommandInterface
hardware_interface::
LoanedStateInterface
hardware_interface::
StateInterface
hardware_interface::
SystemInterface
hardware_interface::
ActuatorInterface
hardware_interface::
SensorInterface
controller
impl
impl
impl
hardware_interface::
Sensor
controller_interface::
ControllerInterface
controller_interface::
ChainableControllerInterface
controller_manager::
ControllerSpec
controller_name: string
controller_type: string
hardware_interface::
System
hardware_interface::
Actuator
ros2_controllers
joint_trajectory_controller::
JointTrajectoryController
joint_state_broadcaster::
JointStateBroadcaster
forward_command_controller::
ForwardControllersBase
forward_command_controller::
ForwardCommandController
position_controllers::
JointGroupPositionController
velocity_controllers::
JointGroupVelocityController
effort_controllers::
JointGroupEffortController
systems
sensors
actuators
hardware_interface::
ResourceStorage
*
controllers list
controller_manager::
ControllerManager
hardware_interface::
ResourceManager
configurations
*
«yaml»
controllers_config
«urdf»
robot_config
hardware_interface::
HardwareInfo
rclpp::Node
The controller manager is a standard "unmanaged" ROS2 node.
Usually, but not necessarily, parsed from the given URDF.

B Schunk Hand

B.1 Cyclic Data



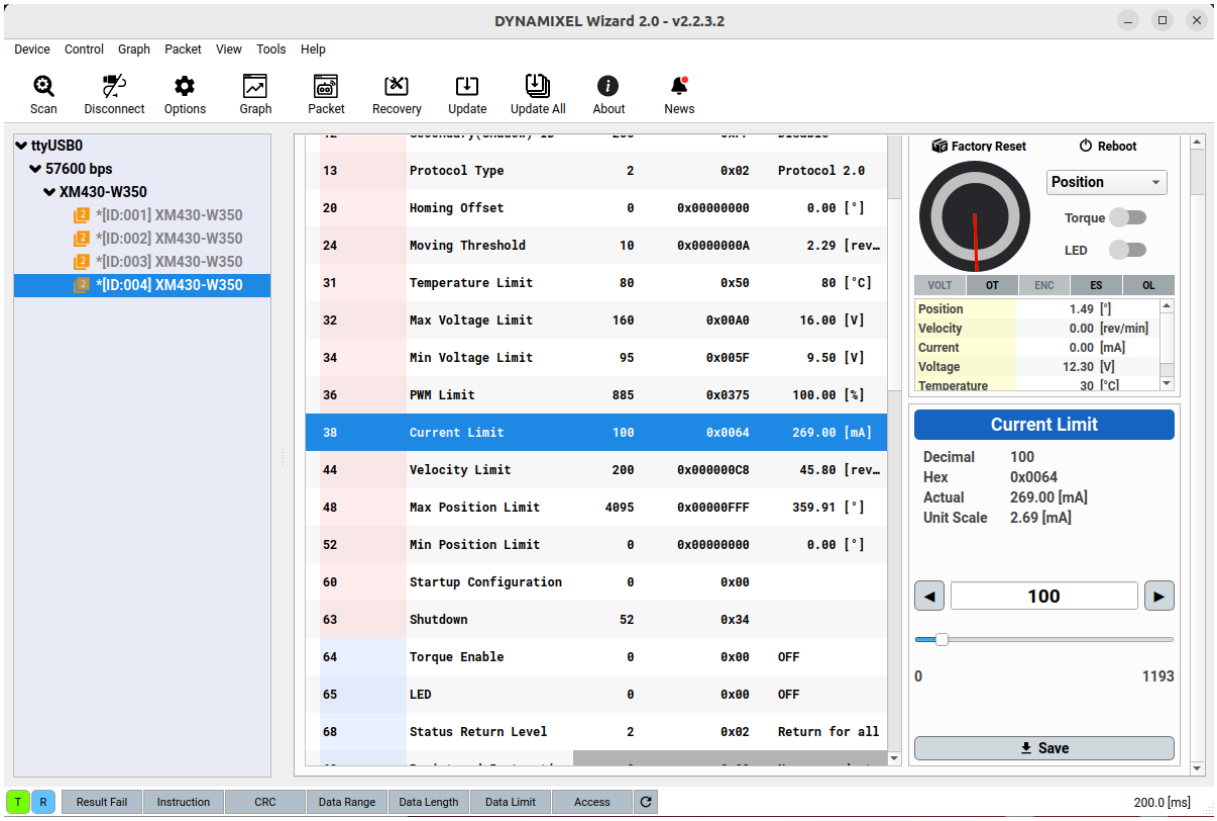
B.2 Acyclic Data



B.3 URDF

C Yale OpenHand Model o

C.1 Dynamixel Wizzard 2.0



Bibliography

[1] Örebro University, *About örebro university*, Accessed: 2025-07-29, 2024. [Online]. Available: https://darko-project.eu/partners/1-orebro-university/#:~:text=%C3%96rebro%20University%20is%20young%2C%20broad,1%2C500%20employees%20and%2017%2C000%20students.

[2] Centre for Applied Autonomous Sensor Systems (AASS), *Research at aass*, Accessed: 2025-07-29, 2024. [Online]. Available: https://www.oru.se/english/research/research-environments/ent/aass/#:~:text=Image%3A%20AASS%20logoThe%20Center%20for,Sweden%20and%20around%20the%20world.

[3] T. Stoyanov and Autonomous Mobile Manipulation Lab, *Autonomous mobile manipulation research group*, Accessed: 2025-07-29, 2025. [Online]. Available: https://amm.aass.oru.se/#:~:text=Welcome%20to%20the%20website%20of,at%20%C3%96rebro%20University%2C%20Sweden.

[4] T. Stoyanov and Autonomous Mobile Manipulation Lab, *Todor stoyanov*, Accessed: 2025-07-29, 2025. [Online]. Available: https://amm.aass.oru.se/people/todor-stoyanov/#:~:text=I%20am%20an%20Associate%20Professor,defended%20in%202012%2C%20on%20the.

[5] ros2 control Development Team, *Ros2 control, humble, getting started*, Accessed: 2025-07-30, 2025. [Online]. Available: https://control.ros.org/rolling/doc/getting_started/getting_started.html#architecture.

[6] F. F. Informatik, *Schunk svh ros description package*, https://github.com/fzi-forschungszentrum-informatik/schunk_svh_driver/tree/master/description, Accessed: 2025-08-01, 2017.

[7] qbrobotics, *Qb softhand ros 2 package*, https://bitbucket.org/qbrobotics/qbhand-ros/src/production-humble/, Accessed: 2025-08-01, 2024.

[8] Intel RealSense, *Realsense-ros*, https://github.com/IntelRealSense/realsense-ros, Accessed: 2025-08-01, 2024.

[9] Point Cloud Library (PCL), *Point cloud library (pcl)*, https://pointclouds.org/, Accessed: 2025-08-01, 2024.

[10] T. Stoyanov, *Franka arm ros 2 integration*, https://github.com/tstoyanov/franka_arm_ros2, Accessed: 2025-08-01, 2024.

[11] FZI Forschungszentrum Informatik, *Cartesian controllers for ros 2*, https://github.com/fzi-forschungszentrum-informatik/cartesian_controllers, Accessed: 2025-08-01, 2024.

[12] Y. G. Lab, *Yale openhand project – model o*, 2025. Accessed: Sep. 7, 2025. [Online]. Available: https://www.eng.yale.edu/grablab/openhand/model_o.html#about.

[13] Y. G. Lab, *Openhand hardware repository – model o*, 2025. Accessed: Sep. 7, 2025. [Online]. Available: https://github.com/grablab/openhand-hardware/tree/master/model%20o.

[14] J.
bibinitperiod M. Mistakes, *Robotis e-manual xm430-w350*, https://emanual.robotis.com/docs/en/dxl/x/xm430-w350/, Accessed: 2025-09-07, 2025.

[15] Y. G. Lab, *Fabrication manual – model o 2.0*, 2025. Accessed: Sep. 7, 2025. [Online]. Available: https://www.eng.yale.edu/grablab/openhand/model%20o/Fabrication%20-%20Model%200%202.0.pdf.

[16] Robotis, *Dynamixel sdk*, https://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_sdk/overview/, Accessed: 2025-09-07.