MASTER

Design and implementation of the real-time control, monitoring and coordination of a wheeled mobile teleoperation setup using haptic force feedback

van de Ketterij, R.L.

Award date:
2021

Declaration concerning the TU/e Code of Scientific Conduct for the Master's thesis

I have read the TU/e Code of Scientific Conduct[i].

I hereby declare that my Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct

Date

2021-10-05

Name

R.L. van de Ketterij

ID-number

1277537

Signature

……..……..

Submit the signed declaration to the student administration of your department.

[i] See: https://www.tue.nl/en/our-university/about-the-university/organization/integrity/scientific-integrity/
The Netherlands Code of Conduct for Scientific Integrity, endorsed by 6 umbrella organizations, including the VSNU, can be found here also. More information about scientific integrity is published on the websites of TU/e and VSNU

Dept. of Mechanical Engineering

Graduation Project

Design and implementation of the real-time control, monitoring and coordination of a wheeled mobile teleoperation setup using haptic force feedback

Master thesis

Author:
R.L. van de Ketterij BSc (1277537)

Supervisors:
Prof. dr. ir. Herman Bruyninckx (Control Systems Technology)
Dr. Regina Luttge (Microsystems/Neuro-Nanoscale Engineering)

Committee:
Dr. Regina Luttge, Chair
Prof. dr. ir. Herman Bruyninckx
dr.ir C.A. López Martínez

Eindhoven, Tuesday 5th October, 2021

Contents

Nomenclature

F	Force
τ	Torque
θ	Angular displacement
x	Position
v	Velocity
d	Distances
s	Seconds
ms	Millisecond
us	Microsecond
Hz	Hertz
kHz	Kilohertz
GHz	Gigahertz
kg	Kilogram
GB	Gigabyte
$DC\ motor$	direct current motor
V	Volt
$^\circ$	Degree
$2D$	Two dimensional
$ASCII$	American Standard Code for Information Interchange
CPU	Central processing unit
IMU	Inertial measurement unit
IP	Internet Protocol
$LIDAR$	Light Detection and Ranging
RAM	Random-access memory
TCP	Transmission Control Protocol

1. Introduction

Mobile teleoperation systems show a significant growth over the last years and promise to take an emerging role in the upcoming years [19]. In mobile teleoperation, a human operator executes a task in a remote environment with the help of a remote manipulator. Mobile teleoperation is widely used in unknown and hazardous environments because it can make use of the cognitive capabilities of the human operator and can perform tasks in large spatial areas. One of the major challenges of mobile teleoperation is to obtain a reliable and good enough perception of the remote environment. Practical examples of mobile robots that are haptically controlled by a remote operator are the inspection of underwater structures on the bottom of the ocean, exploration of hazardous environments, and rescuing people in unknown buildings [1].

Most mobile teleoperation systems use visual feedback as the main source of perception. However, although visual feedback provides much information about the environment, it is not always sufficient because of the limited visual fields, it requires much attention/ training from the operator, and it does not give a perception of the interaction between the mobile robot and the environment [2]. Also, for applications where the light conditions are not optimal or applications with limited network bandwidth, visual feedback could lack in providing the operator with sufficient information.

Haptic force feedback has shown potential as a fail-safe option and in increasing the user's perception of the environment or objects to, for example, minimize the risk of dangerous collisions in hazardous and unknown environments [1], task errors, and operation time [3]. However, in current literature, force feedback is always used as a supplementary cue to help to understand the remote environment, and none has focused specifically on what is necessary to (temporary) fully rely on force feedback when visual feedback is lacking. Currently, in the absence of visual feedback, mobile telerobotic systems have to stop executing their task until the vision information is restored, and this is in many cases undesired. Additionally, no literature is found that describes the software architecture design and implementation of a teleoperation system for mobile robots. Hence, there is a need to further research the potential of haptic force feedback in the teleoperation of wheeled mobile robots.

In this research, as an essential external requirement, an existing generic software architecture library referred to as "template" in this thesis and its concepts have to be used as the guidance for the research approach. The generic approach aims to bring structure to the complex system development process. Including the perspective of a client's goal in introducing this requirement is to try and validate whether and how well these existing software concepts can be used to design and implement a non-trivial application in the scope of this graduation project. Important concepts of this approach are: "(i) The separation of concerns into the five major categories (5Cs): Computation, Coordination, Configuration, Communication, and Composition. (ii) Composability: the extent to which a component makes all of its "5Cs" separately configurable to increase the opportunities to reuse that component in any kind of system. And (iii) Compositionality: the extent to which the behavior of a system can be predicted based on the knowledge of the individual behaviors of the composing components and their interconnections" [4]. This generic approach hypothesizes that such concepts make the system more predictable and easier to build as a composition of components (modularity). All concepts and models behind this approach are described in detail in [4] and briefly discussed in chapter 2. To the best of the author's knowledge, no teleoperation systems exist yet that used these concepts to develop a force feedback application for wheeled mobile robots. This research focuses on the generic development of a haptic teleoperation system of a wheeled mobile robot and how it can be used to increase the operators' perception of the robot's environment with force feedback control.

1.1 Teleoperation system description

The setup used for the teleoperation system for this research can be seen in figure 1.1. The ultimate goal of the teleoperation system is for the user (2) to haptically control the wheeled mobile robot (4) blindly through an unknown environment (6) using only force feedback provided by a haptic device (1). Because the focus of this project is mainly on the generic design and implementation of the haptic teleoperation system using the above-mentioned software models, existing available hardware devices have been chosen that are suited for haptic applications. For the haptic device, the Novint Falcon is used (1). The Novint Falcon is a commercial haptic force feedback device with three translational degrees of freedom. For the wheeled mobile robot, the ropod platform is used (4). The ropod platform is a holonomic automated guided vehicle, consists of four so-called Smart-Wheel units with many actuators and sensors, and has a real-time EtherCAT interface. A distance sensor mounted on the ropod platform is used to perceive the environment of the ropod platform. For the distance sensor, the Sick TIM 561 2D LIDAR sensor (5) is used. Chapter 3 will elaborate more on the generic design approach of the haptic application and chapter 4 on the hardware architecture, describing the individual components and their communications.

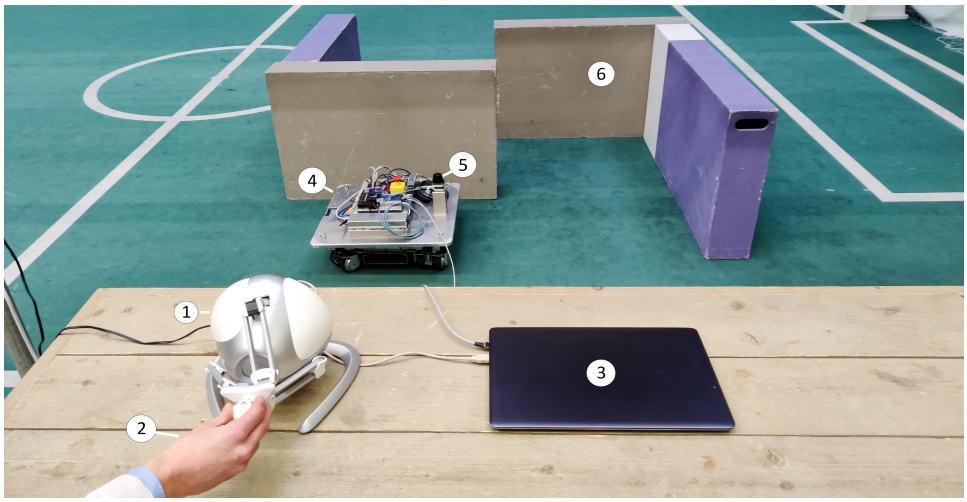


Figure 1.1: Overview of the proposed haptic teleoperation application with numbered components. (1) The Novint Falcon haptic interface, (2) the human operator, (3), the computational computer, (4) the ropod platform, (5) the LIDAR sensor, (6) obstacles in an unknown environment

1.2 Objectives

From the problems stated above, the main research questions for this project can be stated as:

How to design and implement the real-time control, monitoring, coordination, and communication of a wheeled mobile haptic teleoperation system and how can force feedback be used on this system as a fail-safe option for the lack of visual feedback?

In order to reach this goal, the objectives and requirements are stated next. This research adopts the generic modular design approach from the given software template to structure the design process of the haptic teleoperation system of this research. The template provides generic concepts of how to do the real-time control, monitoring, coordination, and communication of a robotic system and has to be entirely specified for this research. Hence, the main aim of this research is to design the system in a modular and predictable manner using the generic approach adopted from concepts of the template. Furthermore, this research aims to investigate what is generic about creating a real-time distributed robotic system and what is specific for this research's haptic force feedback teleoperation application. It also aims to explore how well the template can be used to design and implement the haptic teleoperation system.

The main objectives can be stated as:

1. Determine the hardware architecture for the teleoperation system: Investigate all the hardware and communication resources for the haptic teleoperation system and determine their interactions and information flows.
2. Determine the software architecture for the haptic teleoperation system: Determine how to compose generic software components from the available hardware and communication resources and determine how they can be used to compose an application-specific force feedback teleoperating system.
   2.1. Design the interface of the Novint Falcon, ropod platform, and LIDAR distance sensor devices and integrate their functionality into generic software components so they are ready to be used in the real-time application. This integration also implies the monitoring and coordination of the generic software components.
   2.2. Compose the application-specific software components that uses the generic software components from 2.1 as resources to create the haptic force feedback application. Hence, this application should use the functionality of the Novint Falcon as a haptic interface to control the ropod platform and to perceive information in the form of force feedback.
3. Optional: Guaranteeing the safety of the teleoperation system: Limit the actuation forces and analyse parameters that could cause instability of the control system, like communication delays (latencies).

The overall design and implementation of the haptic teleoperation system should cover the following requirements:

1. The design should to be modular (composable)
2. The design should be predictable (deterministic)
3. The design should provide (re-)configurability at runtime

1.3 Thesis outline

This research is split up into the following chapters. First, chapter 2 will discuss the background for this research. Next, the teleoperation system design approach is described in chapter 3 and the hardware architecture of the teleoperation system will be explained in chapter 4. Furthermore, chapter 5 presents the controller designs and the mappings between different device workspaces. Chapter 6 elaborates on the generic software architecture design of the teleoperation system, and chapter 7 explains the implementation of the generic software architecture design, the implantation challenges, and the real-time performance measurements. Finally, the conclusions, discussions, and related work are elaborated.

2. Background

In this chapter, the work related to this research and some background information is briefly discussed. A comprehensive overview of the conceptual design models applied to the research questions in this thesis can be found in [4]. First, haptics and, more specifically, haptic force feedback is introduced. After that, the main coordination models from the generic software architecture library are listed. Finally, real-time programming and how real-time performances could be measured are described.

2.1 Haptics in mobile telerobotics

The science and technology of experiencing touch sensations are called haptics. Haptics in teleoperation is closely connected with robotic systems by relying on mechatronic devices that generate the experience of touch sensations to the users. Haptic interfaces attempt to enhance or replicate the touch experience of manipulating or perceiving a real environment through haptic devices and control. This research focuses on achieving this goal by applying force feedback.

In force feedback systems, a slave robot acts as a sensor, and the master functions as a haptic force feedback device [5]. This way, the system provides forward and feedback signal pathways from the operator to the environment and back. Force feedback is a very common form of sensory feedback, and the force feedback information may correspond or is obtained by environmental parameters [6]. Forces resulting from the interaction between the slave and the environment are called naturally occurring forces (natural force feedback), and forces generated based on additional information are called virtual forces (assistive force feedback). These forces have to be sent to a haptic force feedback master device so the user can experience/feel the generated forces. For example, the distances to the environment measured by a distance sensor on the slave device can be used to determine the virtual force that the operator feels to perceive the environment better.

Various research related to the concept of adding haptic force feedback in mobile teleoperations has been done already [1]. In [7], for example, the distance from all obstacles around the slave device, measured by a sonar sensor mounted on the mobile robot, is used to compute a virtual repulsive force that is rendered to the operator utilizing a haptic device. The haptic device is also used to control a robot's motion. Haptic interfaces using information from direct force sensors were designed in [6]. Moreover, in [8] shared autonomy using a sensor-based collision avoidance scheme is discussed. The type of controller and control strategy are important decisions when studying the quality of haptic force feedback. In [9] several control command strategies for haptic mobile teleoperations are discussed and compared. Finally, the drivers intentions are estimated more accurately for the haptic control of a powered wheelchair in [10]. The results of all researches confirm that the augmented perception of the environment via force feedback increases task performance and reduces the number of collisions and operator effort and is therefore worth investigating further.

2.2 Software architecture coordination models

As stated in the introduction, a generic software architecture model was selected to be used as the basis for the design approach in this research. With the goal of making the system modular and easier to build as a composition of components (system-of-systems). Detailed explanations of the concepts and models used for this approach, are given by [4]. However, the Life Cycle State Machine, flags, and petri net coordination mechanism are briefly described here because these concepts are important for the software architecture design and implementation of the haptic teleoperation system as described in chapter 6 and 7.

2.2.1 Life Cycle State Machine

"Almost no component (activity) or system can provide its services to other components without making use of the services of multiple resources itself. These resources can be services of other components of physical resources like computer memory" [4]. The Life Cycle State Machine (LCSM) mechanism coordinates the configuration of all resources required for a certain activity before its capabilities can be provided as an external service to other components. An LCSM has the following states, and these are described next: Creation, Resource configuration, Capability configuration, Pausing, Running, Cleaning, Done. These states and the three super states deploying, active, and ready can be seen in figure 2.1. More details on the LCSM mechanism can be found in [11] and [12].

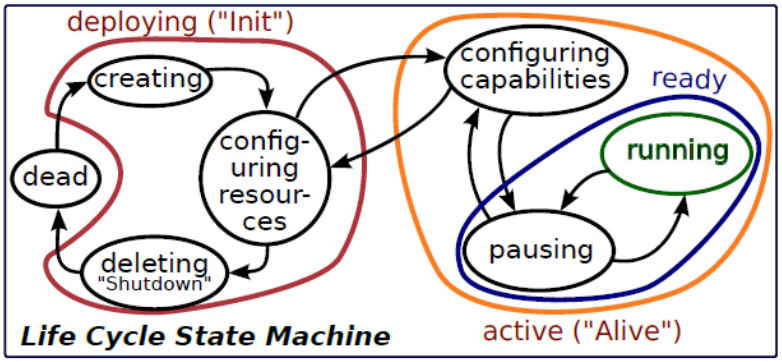


Figure 2.1: Representation of the Life Cycle State Machine coordination model (source [4])

- **Creation:** In this state, the software resources are created, with which the activity does the bookkeeping of its own behavior and of its usage of its external resources.
- **Resource configuration:** The activity configures the service resources it requires for its own operation. This also included re-configuration at run-time, so the latter state can be transitioned to and from multiple times during the lifetime of the activity.
- **Capability configuration:** The activity is not yet providing its services, because it (re)configures its own capabilities to provide a particular service configuration.
- **Pausing:** The provision of the activity's services is put on hold but can resume immediately.
- **Running:** In this state, the activity provides all its services.
- **Cleaning:** The software resources mentioned in the creating state are cleanly removed.
- **Done:** Here, the activity is not able to do anything and must be brought to life by another activity's event.

The three super states are:

- **Deploying:** In this state, the activity does not react to external coordination events and is busy with finding and configuring all the resources it needs to be able to offer its services to others.
- **Active:** Here, the activity react to external coordination events and is visible to other systems.
- **Ready:** In this state, the activity is providing its services to other systems.

2.2.2 Flags

A flag is one of the simplest possible mechanisms to coordinate a control flow of two or more algorithms. The status of a flag can either be true or false. A flag is a simple version of an event case, where handling and firing are the same as setting a flag status to true or false. The goal of using flags is to increase the efficiency of making decisions in a system that has several asynchronous activities that have to work together. An example of the use of flags is to use array of flags as a protocol that coordinates how the mediator and one of the coordinated algorithms go through the coordination. A mediator is introduced to centralize the coordination of a program, for more details of the design of such a system as envisaged in this research, see chapter 6. Another example is that flags can be used to indicate if new sensor data have arrived.

2.2.3 Petri nets

The place/transition net (Petri Net) model is a modeling language for the description of distributed systems. A Petri Net is a mechanism to support the coordination between the control flows in multiple components while decoupling the direct coordination interaction between any two of these components. The added value of a Petri Net model shows up when multiple components need coordination, but they should not be aware of each other and hence should not engage in protocols together. Also, Petri Nets can prevent the occurrence of unpredictable race conditions (different components access shared data simultaneously). Petri Nets models are used for the coordination in the software design as described in chapter 6. Here a brief explanation with a simple, practical example is given, and a detailed explanation can be found in chapter 2.8 of [4].

The core structure of the Petri Net mechanism is a graph data structure that supports the bookkeeping of the coordination behavior. This structure can be composed of a set of behavioral policies, and the resulting composition is a coordination pattern that designers can configure. Figure 2.2 shows an example of the structural model of a simple Petri Net and is used to further explain the concepts of a Petri Net. A Petri Net has the following entities:

- Places: a place is depicted as a blank circle (P_1 and P_2 in the figure) and can contain any number of tokens. A Petri Net must have at least one place and a place is always connected to a transition by an arrow.
- Transitions: a transition is the relation that connects its input and output places. (t_1 in the figure). A Petri Net must have at least one transition.
- Tokens: A token identifies the state of a place (filled or empty) and depicted as black dots in places. A token is connected to one single place at a time.
- Arrow: The arrows are the relation that connects a place to an input or an output of a transition.
- Marking: a marking is the relation that represents one set of tokens in places. It is the state of the data type, after a certain number of operators have been executed.
- Behaviours: A behaviour represents the effect of the fire operator on a transition in the form of the removal and addition of tokens in the input and output places of each transition. The behaviour reaction table and behaviour b_1 of the example in the figure is given here:

input places	**transition**	**output places**
P_1 (SELECT_SCHEDULE_1)	t_1 (with b_1)	P_2 (SCHEDULE_1)

Table 2.1: Places and transitions

Transition behavior b_1	
Fire condition:	Token in all incoming places
Consumption behaviour:	Consume token in all incoming places
Production behaviour:	Produce token in all outgoing places

Table 2.2: Transition behavior

For this example (and commonly) flags are used as a linear Petri Net. That is, flags are directly coupled to the state of a specific place (the presence of a token yes or no). Flags should track the presence of tokens in places and tokens should track the status of flags. This flag-based coordination is coordinated by a third party activity whose only responsibility is to make all decisions for the coordination of the activities involved, and without the coordinate activities being aware of its existence. The third party activity in this research is the application mediator activity of which the design is explained in chapter 6.

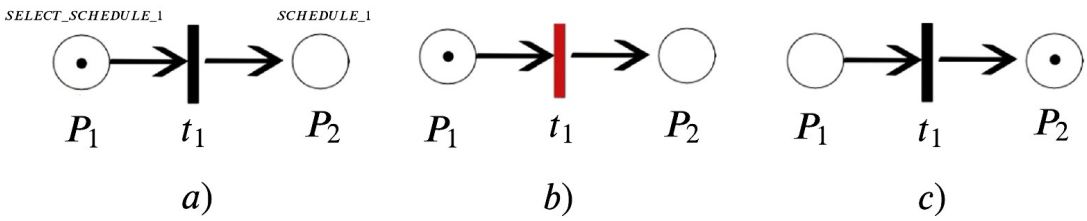


Figure 2.2: Example of a petri net for switching between 2 thread schedulers. a) shows the begin state of the Petri Net model with one token in P_1. In b), the fire condition is met and the transition is firing and c) shows that after the firing, the token in P_1 is consumed and a token in P_2 is produced

This simple example uses the model of a Petri Net in figure 2.2 to switch between two activity schedulers in a single thread and is actually used in the software design in this document (see chapter 6). Let the current scheduler activity be 2 and the goal is to switch from scheduler 2 to scheduler 1 using a Petri Net model. Let transition behaviour b_1 be added to transition t_1, let place P_1 be named SELECT_SCHEDULE_1 and place P_2 be named SCHEDULE_1. Let the presence of a token in place P_1 track the status of $Flag_1$ and let $Flag_2$ track the the presence of a token in place P_2.

The application mediator decides to switch to scheduler 1 and sets the status of $Flag_1$ to true, resulting in the creation of a token in place P_1 (figure 2.2 a). Consequently, the transition behaviour b_1 detects a token in all of its incoming places and triggers the firing of the transition t_1 (figure 2.2 b). The token in incoming place P_1 is consumed and a token in outgoing place P_2 is produced (figure 2.2 c). As a result of the created token in place P_2 the corresponding $Flag_2$ status is set from false to true. Hence, the application mediator now knows that activity scheduler 1 is selected and can now send signals to all activities in this activity scheduler to start their LCSM.

It should be noted that for this simple example, there is not much added value in introducing a Petri Net, but it helps to show the decoupling of the direct coordination interactions. However, it does add value when multiple threads and components need coordination, which is the case in the generic software architecture design as applied and described in chapter 6.

2.3 Real-time programming

Real-time systems are becoming more important as industries focus on distributed computing in automation. Many distributed systems contain one or more activities whose execution must be predictable with respect to the resources they have available. This *predictability* of an activity is often referred to as real-time. A system is said to be real-time if the total correctness of an operation depends not only on its logical correctness, but also on the time in which the operation is performed [13]. Real-time programs must guarantee response within a small tolerance of the ideal instance in time, also known as *deadlines*. Practical examples of real-time systems are medical systems such as a heart pacemaker and a car engine control systems. For these systems, missing a deadline is critical and can lead to physical damage or threat to human life.

The following concepts have to be considered to design an application that meets the real-time requirements as good as possible: the use of an operating system with a preemptible kernel, the chosen scheduling policy and process priority, the communication protocol, the memory management, and the measurements of deadlines. This research focuses on the use of the Linux operating system. The usual sequence for creating a real-time program is to start with a non-real-time program and then create a real-time thread with appropriate resources and scheduling.

2.3.1 Preemption

A preemptible operating system is a system that has the ability to preempt tasks, that is, to stop or pause a currently scheduled task in favor of running a higher priority task. The Linux kernel implements several preemption models, which are described in Appendix A. The fully preemptible kernel model is referred to as the real-time kernel. The real-time kernel is designed to maintain low latency and consistent response time. This kernel allows preemptible priority inheritance and introduces new real-time scheduler policies. A real-time kernel in Linux is achievable by modifying the Linux kernel by installing the so-called PREEMPT_RT patch [14].

2.3.2 Scheduling and priorities

The assigning of resources to tasks and handling process interrupts is done by the operating system's scheduler. The set of rules used to determine when and how to select a process to run is called the scheduling policy. Linux scheduling is optimized for minimizing wait time, maximizing good throughput, and avoiding process starvation. The Linux scheduling policy is based on ranking processes according to their priority. Each process is associated with a value that tells the scheduler how appropriate it is to let the process use the available resources. The Linux scheduling policies can be divided into normal policies (for normal dynamic priority scheduling) and real-time policies (for fixed-priority scheduling), both priority ranges can be found in appendix A. Real-time systems require to meet the timing constraints of the individual tasks and, therefore, that tasks should never be blocked by lower-priority processes. Real-time policies implement the fixed-priority scheduling specified by POSIX[1] The main real-time policy is SCHED_FIFO. Processes using this policy will always immediately preempt any currently running process under the normal scheduling policy and will run until the process goes to sleep or a higher-priority real-time process becomes ready to run. Together with the normal scheduling policies, SCHED_FIFO is described in more detail in appendix A. Also, it is worth mentioning that a real-time thread should never get the highest priority 99 on a system with multiple running processes because this process will never block, and lower priority (system) threads will never get the chance to run.

[1]The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. More info here.

2.3.3 Real-time communication: EtherCAT

A real-time program often has to perform (motion) control tasks and communicate with numerous inputs and outputs (I/O) over a communication network. Achieving deterministic program performance over a network is a challenging task. Ethernet is widely used in industry because of its many benefits over other solutions. By design, traditional Ethernet and fieldbus systems cannot meet the cycle time requirements of milliseconds. However, solutions exist to achieve real-time performances over a traditional network using the Real-Time Ethernet (RTE) protocol. EtherCAT (Ethernet for Control Automation Technology) is a low-cost Ethernet-based RTE fieldbus system invented by Beckhoff. EtherCAT is specialized in real-time applications, allows for *modularity* and easy *reconfigurability*. Furthermore, it combines the advantages of Ethernet with the simplicity of a fieldbus, and it has shown excellent performances in real-time network communications [15]. EtherCAT is developed with the goal to require short data update times with low latency and communication jitter, usually in the range of 100 μs

EtherCAT employs a master-slave principle to control access to the medium. The functional principle of EtherCAT is based on the summation frame principle in which the process data of all slaves on the network are carried together in one (or more) Ethernet frame(s). In contrast to the individual frame approach, wherein each frame carries process data for only one device. With EtherCAT, Ethernet frames are no longer received, copied, and interpreted as process data at every slave in the fieldbus. The EtherCAT slave devices read and insert the data addressed to them while the data passes through the device resulting in processing data "on the fly." Processing data on the fly and using relatively small process data frames results in minimal latency and good system performances. To implement on-the-fly processing, EtherCAT uses logical addressing. All devices read from and write to the same address range of the EtherCAT data message. Logical addressing maps data from the data frame to the slave's local address and memory area also called *I/O-mapping*.

In EtherCAT, there is only one master device. The master is a computer device that sends the EtherCAT frames to the slaves. Also, the master is responsible for maintaining data communication and synchronization between the master and the different slave devices. The synchronization uses a distributed clock mechanism, which leads to a very low jitter (less than 1 μs). Regular broadcasts are sent to counter any internal clock differences to keep the master and slave clocks synchronized after initialization. If a difference is detected, the slave should adjust their internal clock accordingly. EtherCAT has its own state machine, which controllers the states of the slaves. Depending on the state, different functionalities of the slave devices are accessible. The five states of a slave device are Init, Pre-Operational, Safe-Operational, Operational, and Boot. Usually, to check synchronization timeouts and states, and handling slave device errors, a *watchdog* process is implemented in addition to the real-time communication process. An example of an EtherCAT network can be seen in figure 2.3.

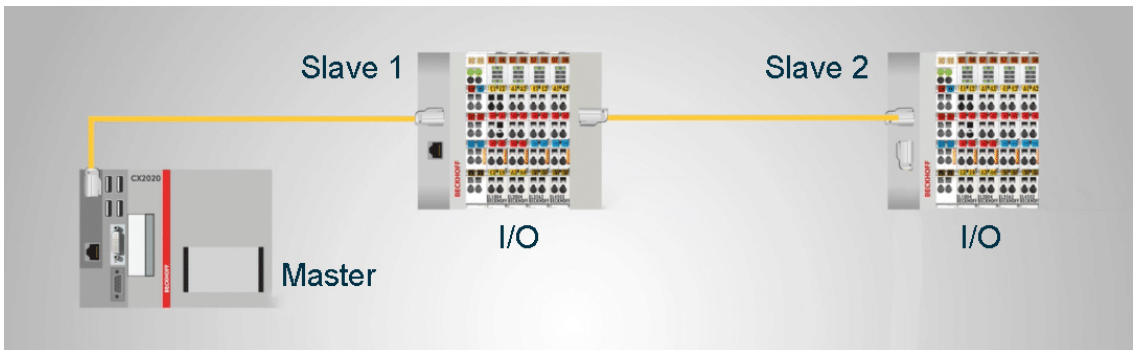


Figure 2.3: Example of EtherCAT network with 2 slaves (source: www.realpars.com)

2.3.4 Memory management

Good memory handling will improve the real-time program's deterministic behavior. The first thing that has to be considered is the locking of memory. Memory locking is a way to ensure the memory of the real-time program stays in the RAM memory of the system. This results in much faster access to those memory locations as disks are extremely slow compared to RAM, reducing the latency for the data accessing. An example of a Linux function that locks the memory for a real-time application is mlockall(). Another important consideration is set the desired stack size of the real-time program if the program becomes very big or a large number or real-time programs are created. However, for this research, the default stack size is sufficient to handle the memory.

2.3.5 Real-time measures

Deadlines define a real-time application and meeting deadlines is essential for the predictability. Latency, together with jitter are the two key performance measures of deadlines in a real-time program [16]. The definition of latency in this research is: the time delay between the occurrence of an event and the time when that event is handled. Latency in itself is not a bad thing, there is always some delay. Latency is bad if it becomes excessive (over time) and therefore, more and more unpredictable. The definition of jitter in this research is: the variation of latency over a specific time period. Jitter describes the amount of inconsistency in latency and in control systems especially the worst values are important because they give rise to the largest disturbances that the execution of the control introduces to the desired controlled system behavior.

One common way of benchmarking a real-time system is the so-called cyclictest. Cyclictest is used for evaluating the relative performance of real-time systems. On Linux, this test repeatedly and accurately measures the difference between a thread's[2] intended wake-up time and the time at which it actually wakes up in order to provide statistics about the system's latencies. It can measure latencies in real-time systems caused by the hardware, the firmware, and the operating system. Because the most important values are the latencies, and in particular the worst-case latency, the system should be in a situation that is as close as possible to its maximum capabilities in terms of memory, CPU, and network use. In this way, the latencies on the system will be influenced by everything that could influence them when running the designed application normally. The system on full load will produce the most realistic latency values. The technical explanation of the cyclictest and how it can be used to measure application performance is described in detail in [18].

[2] "A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. A program can contain multiple threads" [17]

3. Teleoperation design approach

The modular generic design approach of the haptic teleoperation system is based on the architecture and coordination models and their mechanisms mentioned in the introduction and described in detail in [4]. The main idea is first to design all generic components and their interactions independently, such that they are ready to be composed into a larger system. Hence, the components should be ready to be composed into an application-specific haptic teleoperation system. The goal of using components as building blocks and designing their interactions in this way is to bring structure to a system. Furthermore, composing the application-specific system should be more straightforward, and the whole system should better preserve the predictability property. The result is an architectural composition that can be deployed in a system without having to know about the exact workings of these activities and interactions inside of a component but providing how a component interacts from the outside. By designing the system in this way, the coordination of the configuration for the individual components can be designed with LCSMs, and the coordination of the individual components and whole application can be centralized more easily in software. Furthermore, this design approach allows dividing the components into generic and application-specific components more easily, which supports the modularity requirement.

The design approach for the haptic teleoperation system is to first design the hardware architecture of the teleoperation system. That is, thoroughly investigating all hardware devices and communication resources to determine what is needed to provide their basic functionality and map the data flow between them. Secondly, the controllers for the hardware devices need to be designed so that their basic functionality can be coupled to the haptic teleoperation application. Furthermore, the non-trivial mappings between the device's workspaces must be determined to close the haptic loop between the user and the environment. Finally, the software architecture has to be designed. Generic and application-specific software components have to be created using the investigated hardware and communication resources and controllers. To design software components that support the generic and modular approach, the "4C" concerns (computation, communication, coordination, and configuration) and the LCSM states have to be investigated and determined for each software component. Moreover, the coordination between software components and the deployment architecture to deploy the software components on processes, threads, and activities have to be designed. The structure of this thesis is consistent with the design approach. Hence, the hardware architecture design can be found in chapter 4, the controller designs in chapter 5, and details on the software architecture design is described in chapter 6.

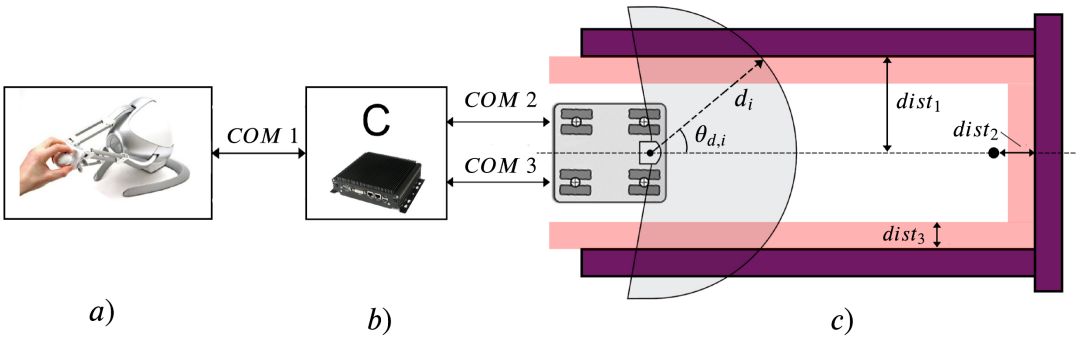


Figure 3.1: Overview of the haptic application. a) The user holds the Novint Falcon haptic interface, b) The computational computer C realizes the communications with the devices and runs the software program. c) The ropod platform with the LIDAR sensor moves in a dead-end hallway. The depicted COMs are the different communications between the devices

The haptic teleoperation hardware is briefly described in the introduction. The haptic application that is chosen for this research and hence, for which the teleoperation system should be designed, can be seen in figure 3.1. The application is designed as a test application to verify how well the force feedback teleoperation is performing in terms of obstacle avoidance, dangerous collisions, and operation time. The application consists of the Novint Falcon that has to be used as an input device to control the motion of the mobile ropod platform. The ropod device is placed in a dead-end hallway and has to be blindly navigated to the end of the hallway, where it has to stop at a fixed distance from the wall. The LIDAR sensor mounted on the ropod platform has to perceive the environment by measuring distances from obstacles. This environmental distance information has to be translated into virtual forces in a helpful way and mapped to the force feedback workspace of the Novint Falcon, so the user feels the interaction with the environment and tries to help the user in blindly navigating the ropod in an "unknown environment".

4. Hardware architecture design

This chapter describes the hardware architecture design, that is, all the details of the hardware devices and communication interfaces necessary to integrate their functionality into generic software components to be used in the composition of the specific haptic teleoperation application. The haptic teleoperation system consists of four primary hardware devices: The Novint Falcon haptic device, the ropod platform, the Sick LIDAR sensor, and the computational computer. The latter is responsible for running all the designed software programs and all the communications between devices. These hardware devices were chosen because they were made available for this project and are suitable for haptic applications. The communication between the components is chosen to be wired, and hence this research assumes that the physical communication layer does not cause any delays in the system. The four devices are described in detail next, and a general overview of the teleoperation system hardware, their communications, and the information over the communications can be seen in figure 4.1.

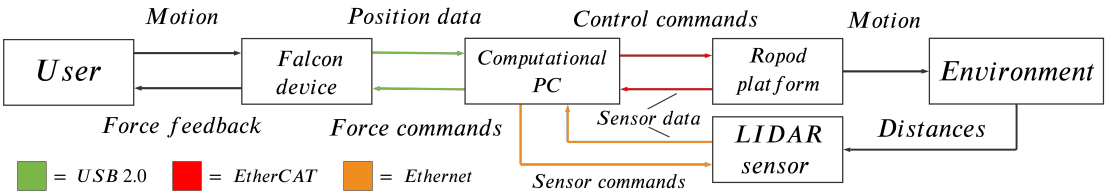


Figure 4.1: General block diagram of the hardware components of the teleoperation system, their communications, and information exchange

4.1 Computational computer

The computational computer in the teleoperation system is the center of the system and responsible for running all the designed software programs. The computer is responsible for providing the following resources: the physical inputs and outputs ports, storage media, computational hardware (CPU cores on a processor), and a user interface.

The computer used in this research is the HP Elitedesk 800 G3 TWR model. It has three Gigabit Ethernet ports on three separate Network Interface Cards (NICs) and four USB 3.2 ports. It also has three DisplayPorts for connecting the computer to a display. The processor in this computer is the Intel Multi-Core i5-7500 with four cores and a clock speed of 3,4 GHz. Furthermore, the memory consists of 16 GB DDR4 RAM and 256 GB Solid State Disk (SSD) memory. As mentioned before in chapter 2.3, the operating system is chosen to be Linux with the real-time patch to convert Linux into a fully preemptible kernel and get access to real-time programming functions.

4.2 Novint Falcon

The Novint Falcon is a low-cost commercial haptic force feedback device with three translational Degrees of Freedom (DOF) and can be seen in figure 4.2. The device has been made available for this project. In this teleoperation design, the Falcon is responsible for interpreting the user's motion of the end-effector and providing the calculated virtual forces back to the user. Its structure and design, the Novint Falcon is based on the 3-DOF Delta parallel robot. The variant that the Novint Falcon uses is most similar to that proposed by Thai [19] where the spherical joints of a traditional delta robot are replaced with single DOF rotary joints, mainly to reduce manufacturing costs. The device is mainly used for gaming. However, the Falcon has already seen various entrees in research applications due to its low costs and accessibility [20]. The Novint Falcon has a removable grip so that different end-effector can be used for different application purposes. The

grip used in this research is the default grip which has four digital buttons.

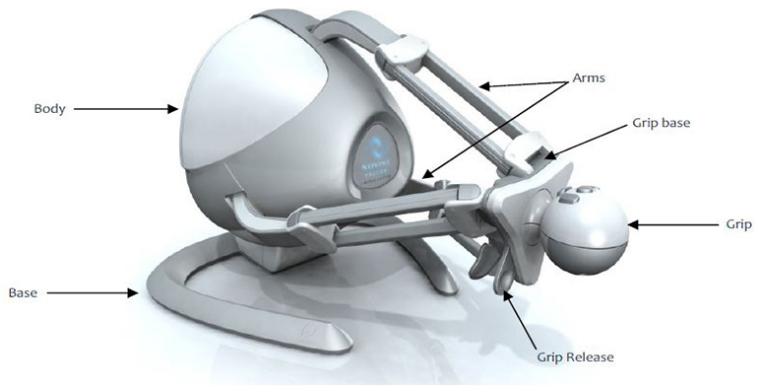


Figure 4.2: Overview of the Novint Falcon haptic device (Source: www.Novint.com)

A disadvantage of the delta-robot form is the limited workspace. The three legs of the Falcon work in kinematic concert to actuate the end-effector, but the reach of the connected linkages limits each leg. The entire 3D workspace of the Falcon is marketed as 10x10x10 cm. However, due to the geometric properties of the Falcon, the workspace is actually bounded by warped tri-hemispherical regions overlapping along the common longitudinal (z) axis. The quantification of the workspace is done in [21] and the plot of the Falcon's achievable workspace can be found in appendix B. The maximum force that the Falcon can deliver in one translational direction is 10 newton. The Falcon is an impedance-type haptic kinesthetic device, which means the user's motion is translated into positions using position sensors (encoders) in the device, the output force is computed accordingly, and that force is sent and felt by the user through the device. The three encoders used to sense the position of each joint are coaxial 4-state encoders with 320 lines per revolution. The motors used to generate the force feedback are three Mabuchi RS-555PH-15280 12 volt DC motors. The coordinate frames conventions used for the Novint Falcon in this research can be seen in figure 4.3.

4.2.1 Communication interface and calibration

The Novint Falcon uses a USB 2.0 serial interface to communicate between the computational computer and the internal communication chip of the Falcon. The internal communication chip of the Falcon is the FTDI FT232R USB to serial UART [1]. The internal communication chip is connected to the internal microcontroller of the Falcon. The data send over the serial USB interface is raw sensory data from the encoders and raw motor commands to provide actuation. The internal microcontroller runs default firmware which interprets the raw sensory data and raw motor commands. The internal microcontroller tries to maintain a 1 kHz update rate with commanded forces maintained by the firmware for 100ms unless overwritten. The firmware is also responsible for the calibration (homing) of the Novint Falcon device using flag bits. The calibration process or the so-called homing-mode of the device is needed to determine the offset of each encoder before using the device. To calibrate the device, the end-effector has to be pulled all the way out and all the way in over the z-axis respectively. The main disadvantages of the USB 2.0 interface is the relatively low communication speed and relatively large latencies, which could cause problems for the 1 kHz communication

[1] A universal asynchronous receiver transmitter (UART) is a hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable.

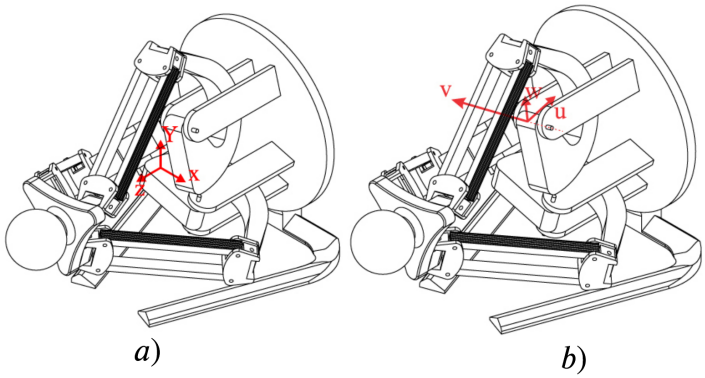


Figure 4.3: Coordinate frames conventions used in this research of the Novint Falcon device. a) The Cartesian coordinate system and b) the UVW coordinate system for one leg (Source: [21])

4.2.2 Novint Falcon kinematics

Before the Falcon can be used and controlled as a haptic manipulator, the kinematic equations and device properties have to be determined. The position of the end-effector is described by the triple [x,y,z] with the x-direction defined as horizontal and the z-direction perpendicular to the base (see figure 4.3). Each leg has an equal angular spacing or $2\pi/3$ around the z-axis. The first step in obtaining the kinematic equations of the Novint Falcon consists in obtaining its geometric and kinematic properties. Next, the forward and inverse kinematic problems have to be solved. The geometric properties, together with the inertial properties and a schematic of each leg of the Novint Falcon, can be found in appendix B.

The forward kinematic problem consists in defining the mapping of the known set of the actuated joint values to the unknown position of the end-effector. The forward kinematics is most useful in haptic applications, which include sensing the position of the user's hand. The forward kinematics must be solved in each control loop to map the encoder space to the user's task space. Solving the forward kinematics problem leads to 16 real solutions when solved for a 3-DOF Delta parallel robot. For the Novint Falcon, only one solution is correct and can be found using the Newton–Raphson iterative approach [22].

The inverse kinematic problem consists of finding the required encoder values of the three joints knowing the position of the center of the end-effector in Cartesian coordinates. Due to the redundant constraints of the Falcon's mechanism, this problem is more straightforward than the forward kinematics. It can be solved by transforming the end-effector location to the frame of a single leg, namely the UVW coordinate frames (see figure 4.3). This then allows the use of the 2D Pythagorean relationships to solve the kinematics analytically [23]. The full derivation of the kinematic equations is not derived here but can be and found in [24] based on the study conducted in [23].

4.3 Ropod platform

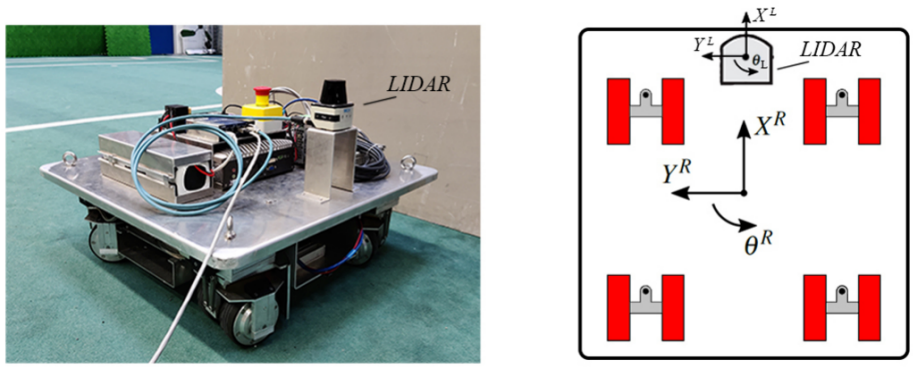
The ROPOD project is a research project with the goal to develop ultra-flat, ultra-flexible, cost-effective robotic pods for handling legacy in logistics [25]. Furthermore, the main concern in the ROPOD project is the modularity of the design. Each robotic pod (ropod) is a platform that consists of four so-called smart-wheel units. A smart-wheel unit, in turn, consists of two separately actuated wheels with embedded brush-less DC motors. Multiple ropod platforms can form a flock of ropods that can cooperate in order to move larger objects. The prototype of the ropod platform has been made available and used for this research. In the teleoperation system, the ropod is the wheeled mobile teleoperated device responsible for moving in an unknown environment using

operator commands. The ropod platform prototype and a smart-wheel unit and can be seen in the left and right picture of figure 4.4, respectively.

a) b)

Figure 4.4: a) The ropod platform prototype. b) 2 perspectives of an individual smart-wheel unit (source: www.cstwiki.wtb.tue.nl)

When mounted on the platform, the most important feature of the smart-wheel units is their back-drivability. This is achieved by the fact that the smart-wheel units have a caster offset with the platform. The platform's base is holonomic in the horizontal plane, such that it can translate and rotate in any direction in this plane, regardless of the orientations of the smart-wheel units. A consequence of the caster offset is that some movements of the platform will require the wheel units to rotate around their pivot axes in order to make the transition from their pulling to pushing configuration. The coordinate frames convention used for the ropod platform in this research can be seen in the right picture of figure 4.5. The front of the robot is in the positive x direction and hence, the LIDAR sensor (see section below) is mounted on the front of the ropod platform.



a) b)

Figure 4.5: a) Ropod platform overview including the LIDAR sensor. b) The coordinate frame conventions of the ropod and the LIDAR sensor

4.3.1 Communication interface and calibration

The ropod platform uses the EtherCAT Ethernet-based fieldbus system (see section 2.3.3) for the communication between the computational computer and the four smart-wheel units. The embedded electronics in the smart-wheel units allow them to act as a standalone EtherCAT slave device. Hence, the embedded electronics handle the real-time communication between the smart-wheel unit and the EtherCAT master device. Furthermore, the smart-wheel units handle the low-level control of the motors themselves. A single smart-wheel unit features extensive sensing and actuating capabilities. It is equipped with multiple encoders, gyroscopes, and an IMU, which allow measuring the individual wheel velocities, pivot velocity, and acceleration of the units. The full list of all sensing and actuating parameters from and to the smart-wheel slave devices can be seen in the EtherCAT smart-wheel data message structure in appendix C. The EtherCAT hardware

on the ropod platform consists of one EtherCAT coupler and two slave terminals. The EtherCAT coupler is the Beckhoff EK1100 and is the link between the EtherCAT protocol at fieldbus level and the EtherCAT terminals. The two EtherCAT terminals are the Beckhoff EL6652 with 2 Ethernet ports each to connect the four smart-wheel slaves and the master device. The above-mentioned computational computer in this teleoperation system is used as the EtherCAT master device for the low-level ropod control. Hence, the low-level ropod control software should also implement The EtherCAT topology of the four slaves, the EtherCAT coupler, EtherCAT terminals, and the master device can be seen in figure 4.6.

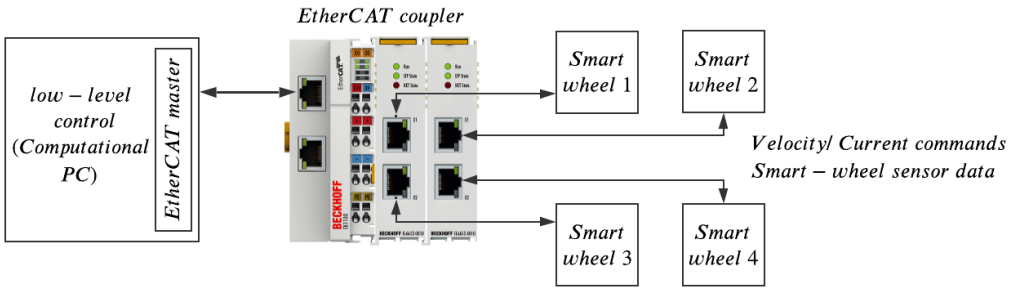


Figure 4.6: EtherCAT network hardware topology of the ropod platform. The EtherCAT coupler is mounted to the ropod platform. The data over the network are velocity or current commands as the input and sensor data as the output of the Smart-wheel unit slaves (source coupler image: www.beckhoff.com)

Before the smart-wheel units can be used in the motion control of the platform, the relative pivot encoders of each unit have to be calibrated. The calibration is handled by the internal firmware on the embedded electronics of the smart-wheel units. To calibrate the encoders, each smart-wheel unit has to turn around its own axis multiple times, which is done by letting the two actuator wheels rotate in opposite directions for a certain period of time.

4.3.2 Ropod platform kinematics

The kinematic equations are essential for the motion control of the ropod platform. Assuming that there is no wheel slip, the relation between the platform velocity and the velocity of the smart-wheel units can be determined using forward and inverse kinematics. The forward kinematic consists of finding the platform velocity $v = [\dot{x}_R, \dot{y}_R, \dot{\theta}_R]^T$ using the measured wheel velocities $\dot{\varphi} = [\dot{\varphi}_{1,l}, \dot{\varphi}_{1,r}, ..., \dot{\varphi}_{N_w,l}, \dot{\varphi}_{N_w,r}]^T$, with $N_w = 4$, the number of smart-wheel units mounted to the ropod platform. In a similar way, the inverse kinematics consists of finding the wheel velocities given the platform velocity. Using the geometric relations of the ropod platform and smart-wheel units, the kinematics problem is solvable by performing two unique matrix manipulations. The full derivation of the kinematic equations is not derived here but can, together with the geometric properties of the ropod, be found in [26]. For the mapping from the Novint Falcon to the ropod platform, is worth mentioning that the total mass of the ropod (36.5 kg) [27] is much bigger than the mass of the end-effector of the Falcon (0.89 kg) [21].

4.4 Sick LIDAR sensor

In this teleoperation design, the unknown environment of the mobile robot is perceived using a LIDAR distance sensor. LIDAR is a method for sensing ranges by targeting an object with a laser pulse and measuring the time for the reflected light to return to the receiver. In particular, the Sick 2D LIDAR TiM561 is used for this research and can be seen in figure 4.5 and 4.7. The TiM561 has a 2D scanning range of 270 degrees with an angular resolution of 0.33 degrees. The scanning frequency is 15 Hz and the working range is from 0.05 m to 10 m. The LIDAR sensor is mounted on top of the ropod platform at a height of 20 cm so the full 270 degrees 2D scanning

plane is not interfered by any hardware. The scanning range and coordinate frame conventions of the LIDAR mounted on the ropod can be seen in figure 4.5 and 4.7.

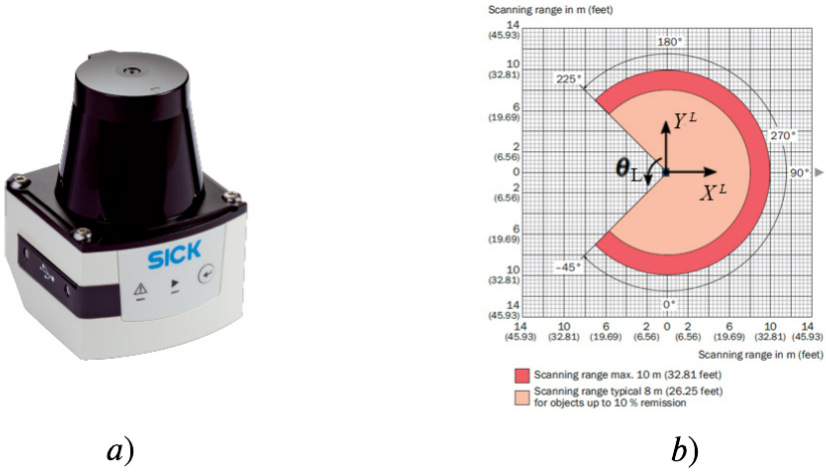


Figure 4.7: a) The SICK TiM561 LIDAR sensor. b) The scanning range and coordinate frame convention of the LIDAR (source: www.SICK.com)

4.4.1 Communication interface

The SICK TiM561 uses an Ethernet connection with the TCP/IP protocol to communicate with the computational computer. The messages send from and to the LIDAR sensor are called telegrams. Embedded electronics in the LIDAR run default firmware to interpret the communication telegrams. Furthermore, the LIDAR can communicate with two types of telegram formats, namely ASCII and Binary. In short, the ASCII protocol uses plain text, and the Binary protocol uses hexadecimal code grouped into pairs of two digits to communicate with the embedded electronics in the LIDAR sensor. The ASCII protocol is chosen for the communication between the LIDAR for this research because it is human-readable and more convenient for debugging purposes. The LIDAR sensor acts as a TCP server and has a fixed IP address which has to be set once while setting up the device. The computational computer is used as the TCP client and has to maintain the connection with the TCP host. The raw data sent from the LIDAR sensor is a stream of raw distance points in polar coordinates.

In conclusion, this chapter describes how the first objective is achieved. The information necessary to create generic software components from the hardware devices and communications between hardware devices is described. Moreover, this chapter sums all the device and communication resource constraints that have to be taken into account. Also, the specifications of the computational computer are given, which are essential for the performance of the whole application and performance measures in chapter 7.4.1. The communication of the Falcon is USB 2.0, and of the LIDAR sensor is TCP/IP over Ethernet. Hence, both communications are asynchronous. The update rate is set to be 1000 Hz for the Novint Falcon and the ropod platform and 15 Hz for the LIDAR sensor. Both the Novint Falcon and ropod platform need to do a calibration process before they can provide their functionality to be used in the overall teleoperation system. All of the above has to be taken into account in designing the software architecture in chapter 6. For the controller designs, this chapter describes the kinematic equation of the Novint Falcon and ropod platform to convert between joint and cartesian coordinates. Furthermore, the coordinate frame conventions and workspace constraints are given for all devices so that they can be constantly used in the further design process.

5. Controller designs

Before the functionality of the ropod platform can be used, a controller to control the platform velocities has to be made. In this teleoperation system design, the motion of the ropod has to be controlled with the use of the Novint Falcon's input values, so the limited workspace of the Novint Falcon has to be mapped to the workspace of the ropod platform. Also, it is chosen to keep the end-effector of the Novint Falcon in the origin of the Falcon's workspace so the input of all the three translational axes can be interpreted equally. On top of that, the method for calculating the virtual environmental forces felt by the operator has to be determined. Hence, the workspace of the LIDAR sensor has to be mapped on the workspace of the Novint Falcon. This chapter elaborates on the designs of the controllers needed in this research and the devices' workspace mappings to close the application-level haptic control loop.

5.1 Ropod platform controller

The design of the ROPOD controller used in this research is based on the kinematic controller with torque distribution described in [26] and slightly modified for this research. In [26], the focus is on an interfacing the ropod platform based on velocity commands for various reasons. The long-term goal of the ROPOD project is among other things to also support force commands and force interaction and to allow, for example, interactions with people working with the platform. However, this falls outside the scope of this research.

Mobile robot control based solely on kinematics is widely used due to its simplicity and intuitiveness. The kinematic platform velocity controller, solely based on the wheel velocities, can be seen in figure 5.1. This approach first transforms the platform velocity setpoint $v_{sp} = [\dot{x}_{sp}, \dot{y}_{sp}, \dot{\theta}_{sp}]^T$ into wheel velocity setpoints $\dot{\varphi}_{sp}$ using inverse kinematics. The wheel velocity $\dot{\varphi}$ for each wheel is then controlled independently using local control loops, resulting in the motion of the platform. The output of the wheel velocity controller is torque, which has to be translated to current because, in normal operation mode, current has to be sent to the smart-wheel units. The embedded electronics in the smart-wheel units handle low-level current control of the motors themselves. However, in this over-actuated system, the wheel velocity controllers can perturb and fight each other because each wheel controller uses only the wheel velocity available locally. Above a platform speed of 0.8 m/s, the wheels begin to resonance, causing oscillations and limiting the system's performance, and even causing instability due to excitation of unmodeled dynamics [26].

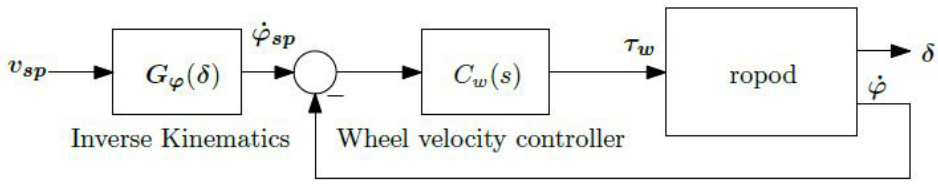


Figure 5.1: Kinematics wheel velocity control (source [26])

Therefore, [26] also proposes a torque distribution algorithm to provide a way to control how the torque will be distributed to all the wheels. Furthermore, a feed-forward control action is added to have a control loop at the platform velocity level and decrease the oscillation problem mentioned above. The final ropod platform velocity controller used for this research can be seen in figure 5.2. The smart-wheels velocities are transformed into platform velocities using forward kinematics and used for the platform velocity controller. The output of the platform velocity controller is the platform wrench $w_p = [f_{xr}, f_{yr}, \tau_{\theta r}]^T$. f_{xr} *and* f_{yr} are the forces applied to the ropod platform, and $\tau_{\theta r}$ is the total torque applied to the ropod platform in local coordinates.

This platform wrench is the input for the torque distribution algorithm, which will distribute this wrench over the available smart-wheel units. The lower-level wheel velocity controllers are kept because it is desired to add damping locally at the wheel to damp out oscillations. However, with the introduction of the torque distribution, the gains of these lower-level wheel velocity controllers are lowered, and the integral actions are removed to reduce the effect of local wheel velocity controllers perturbing each other.

The wheel velocity controller consist of eight lower-level linear velocity controllers (one for each wheel) and the platform controller consists of three lower-level linear velocity controllers (one for $\dot{x}$, $\dot{y}$ and $\dot{\theta}$). All these linear velocity controllers are designed the same with following control actions: a gain action (P), an integral action (I), a lead-lag compensator, a low-pass filter, and a saturation for the integral action. The values for each of these control actions of all lower-level velocity controllers can be found in appendix D.

The forward en inverse kinematics are matrix manipulations and have to be calculated for each control loop. Because matrix manipulations are known to be computationally expensive, this has to be taken into account while implementing this controller. However, the matrices are fixed-size, and therefore, the computational time is expected not to vary that much. Also, it is worth mentioning that the kinematic platform and wheel controllers are a set of linear controllers to control the non-linear dynamics of the ropod platform. Therefore the unmodeled non-linear dynamics can act as disturbances to the system. The implementation of the ropod controller in code can be found in chapter 7.3, where the control blocks are translated to convenient C++ classes and where the controller is coupled to the EtherCAT communication to get the inputs and send the outputs.

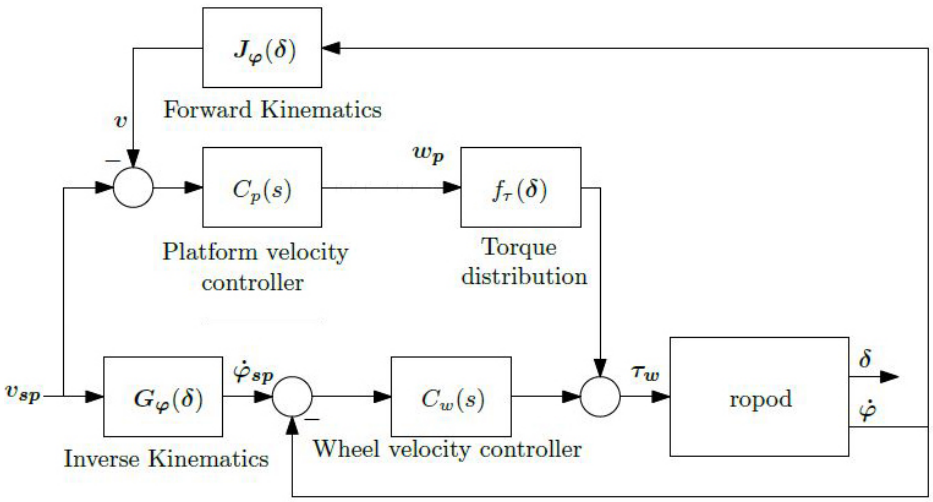


Figure 5.2: Kinematics platform control with torque distribution (source [26])

5.2 Novint Falcon controller

In this research it is chosen to pull the end-effector of the Novint Falcon back to its origin. By doing so, the displacement of the Falcon's end-effector from the origin can be used to generate motion commands for the ropod platform in an intuitive manner. Furthermore, when the user is not moving the end-effector, the ropod will stop its motion, which is desired for safety reasons.

The origin of the Novint Falcon is set to be [x,y,z] = [0,0,0] in meters. Hence the origin (and thus the reference position for this controller) is expressed in the cartesian coordinate frame. To actively control the [x,y,z] position of the Novint Falcon, it can be considered to have three inputs and three outputs. The three inputs for the controller are the motor voltages commands, and the three outputs are the encoder values of each leg. So, the system can be posed as three single-input

single-output (SISO) systems rather than one multiple-input multiple-output (MIMO) system. Because the kinematics of the Falcon is known, the input and output values can be converted into forces and position in cartesian coordinates.

The controller design depends on the implementation options available for interfacing the Novint Falcon through software. For this, the LibNiFalcon development library for the Novint Falcon is used and described in detail in chapter 7. LibNiFalcon is an open source alternative to the official drivers from Novint and is widely used [21], [28], [29], [30]. To keep the end-effector of the Novint Falcon in a desired constant configuration, it is chosen to virtually couple the end-effector and the origin with a spring and a damper. The difference of the actual position in cartesian coordinates [x,y,z] and the origin is measured and a simple discrete linear PD controller is used to calculate the attractive restriction forces in each translational direction to keep the cartesian position of the end-effector in the origin of the Falcon's workspace. The derivative action is added to add damping in the system and increase the stability of the system. Tuning of the derivative action should be carefully done because it could amplify high-frequency noise. However, it is worth mentioning that there is already a low-pass filter present in the internal microcontroller of the Falcon. The cycle time of the position control loop is set at 1 kHz.

The block scheme for the proposed controller can be seen in figure 5.3. The calculated force setpoints for the Falcon are translated into motor torques using the Jacobian matrix. The torques are translated into motor voltages which are sent over the USB 2.0 connection to the internal microcontroller of the Falcon device. The sensed encoder values are then translated into angular positions θ in degrees and used in the forward kinematics function to determine the end-effector position x in cartesian coordinates $[x, y, z]$. The end-effector position is also fed back to calculate the Jacobian matrix. For the calculation of the Jacobian matrix, the LibNiFalcon library uses the calculated angular positions θ_c using the inverse kinematic function **and** actual measured angular positions θ. The above steps have to be executed every control cycle.

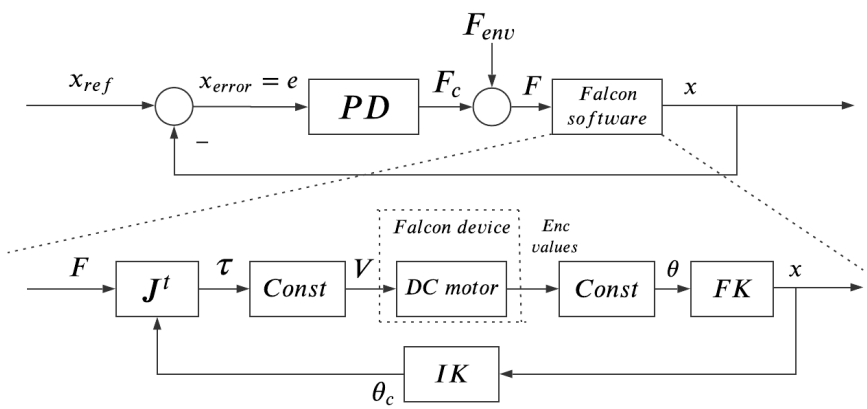


Figure 5.3: Block diagram of Novint Falcon position controller

with,

$$x_{ref} = \begin{bmatrix} x_{ref} \\ y_{ref} \\ z_{ref} \end{bmatrix}, x = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, F = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix}, \tau = \begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix}, V = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix}, \ \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}, and\ \theta_c = \begin{bmatrix} \theta_{c,1} \\ \theta_{c,2} \\ \theta_{c,3} \end{bmatrix}$$

and the numbers 1,2,3 referring to the three motorized joints of the Falcon.

The total force F that is felt by the operator is chosen to be:

$$\underbrace{\begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix}}_{F} = \underbrace{\begin{bmatrix} F_{c,x} \\ F_{c,y} \\ F_{c,z} \end{bmatrix}}_{F_c} + \underbrace{\begin{bmatrix} F_{env,x} \\ F_{env,y} \\ F_{env,z} \end{bmatrix}}_{F_{env}} \tag{5.1}$$

Where F_c is the restriction force that tends to fix the end-effector to the origin and F_{enc} is the force generated by the interaction of the robot with the environment using LIDAR data. Furthermore, e_k is the position error $e_k = x_{ref} - x$ at an instant k. The generation of force F_{env} is discussed in the next paragraph, and F_c at instant k is given by:

$$F_c = K_p e_k + K_d(e_k - e_{k-1}) \tag{5.2}$$

The Falcon has a maximum force it can deliver in each translational direction, which is around 10 Newton. Therefore, the controller outputs for each translational axis also have to be limited accordingly. K_p and K_d are chosen to be diagonal matrices with $K_{p,11} = K_{p,22} = K_{p,33}$ and $K_{d,11} = K_{d,22} = K_{d,33}$. It is difficult to determine the controller gains analytically since the Falcon's characteristics are non-linear due to, for example, backlash and friction and are therefore tuned experimentally. Furthermore, the reference values for the control gains are determined by scaling the maximum reachable position of the end-effector over the [x,y,z] axis and a portion of the maximum force output values of each axis. So that, even when the position of the end-effector is on the limits of one of the axis, the environmental force could still influence the force felt by the operator. Furthermore, the control values depend on the compensation of the gravitational force acting in the y-direction. After tuning, the value of K_p is set to 55, and the value of K_d is set to 0.2.

There also has been looked to control the Falcon using the angular positions to eliminate the use of the non-linear forward kinematics function in each control loop. However, it is chosen to control the cartesian position of the Falcon because the source code of the libnifalcon has to be altered for the direct use of angular positions, which is not in the scope of this research but can be looked at in future work. Also, since the masses and Jacobian of the Novint Falcon are known [21], gravity compensation could be added as a feedforward loop to equalize the controller output values in each direction.

5.3 Haptic application control loop

This section describes the two mappings necessary to close the loop between the human operator and the unknown environment of the haptic application. The first mapping is from the input positions of the Novint Falcon to the motion control setpoints of the ropod platform. The second mapping is from the distances measured by the LIDAR to the forces felt by the operator. One of the biggest and unique challenges in mobile teleoperating systems is the kinematic discrepancy between the master and slave systems. The mappings are not trivial and a lot of different mappings can be designed for numerous haptic applications. The mappings for the teleoperation system of this research can be seen in the block diagram in figure 5.4,

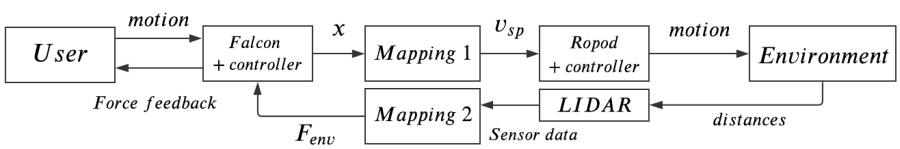


Figure 5.4: Block diagram of closed haptic loop

5.3.1 Velocity setpoints

For the first mapping this research is based on the car-driving metaphor for calculating motion commands for the ropod platform because it is believed that this an intuitive method to start the research in haptic teleoperation with. The car-driving metaphor uses position-velocity kinematic mapping, where the displacement of the end-effector across a horizontal plane (x,z) is mapped to the linear and angular velocities of the ropod ($\dot{x}_{sp}, \dot{\theta}_{sp}$) and can also be seen in figure 5.5 a.

The displacement of the end-effector of the Falcon in the x and z direction with respect to its origin is considered to be proportional to the desired velocity setpoints for the ropod platform, such that:

$$\dot{x}_{sp} = k_1 \cdot q_1 \tag{5.3}$$

$$\dot{\theta}_{sp} = k_2 \cdot q_2 \tag{5.4}$$

with,

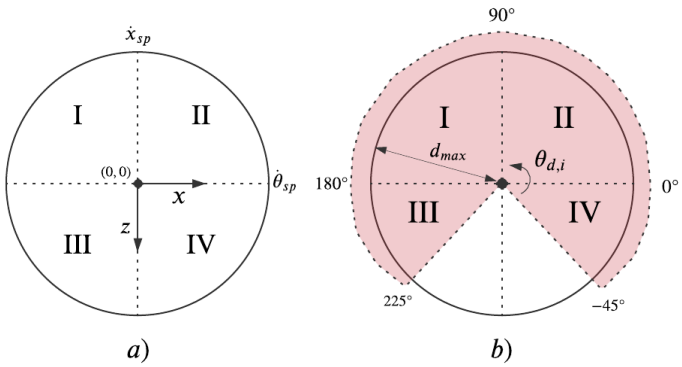
$$q_1 = \begin{cases} -z, \text{if } |z| > z_{dz}, \\ 0, \text{if } |z| \leq z_{dz} \end{cases}, \quad q_2 = \begin{cases} x, \text{if } |x| > x_{dz}, \\ 0, \text{if } |x| \leq x_{dz} \end{cases}$$

where z_{dz} and x_{dz} are positive constants to implement a dead-zone. This dead-zone presents sensitive movements of the ropod due to very small displacements of the Falcon's end-effector. Although linear functions are used here, other monotonic functions such as cubic or quadratic functions can also be used. Note that the speed setpoint $\dot{x}_{sp}$ has positive values for negative z values since a coordinate system is used in which the value of z is negative when pushing the end-effector forwards with respect to the operator. The dead-zone values z_{dz} and x_{dz} are both set to 0.5 cm in this research. For this research, the values k_1 and k_2 are determined as the ratio between the maximum velocity of the ropod and the maximum reachable distances of the Novint Falcon from the origin of its reference frame. Hence, k_1 is set to 28 and k_2 to 10.

5.3.2 Environmental force

The second mapping necessary for closing the haptic loop is to calculate the forces that the operator feels due to interaction with the environment. As mentioned before, the Sick TiM561 LIDAR sensor is used to get distance information of the obstacles surrounding the ropod. The LIDAR sensor data is a list of 811 distance values per scan with their corresponding scanning angle $\theta_{d,i}$ at a scan frequency of 15 Hz. Furthermore, the ropod is modeled as a point rather than the actual shape of the ropod for simplicity and time restrictions of this research, and each scanned point from the LIDAR sensor is considered an obstacle. A small moving average filter with a window of 3 samples (truncated endpoint) is used to reduce the noise of the LIDAR sample data, resulting in a data output size of 809 points per scan ($i = 809$) and a small offset in the scanning angle.

The environmental force should prevent the ropod from moving and turning towards obstacles by giving the operator distance information between the ropod and obstacles in the form of a force. The first approach of calculating is based on the traditional potential force field used for path planning of mobile robots [**?**]. However, the difference is that there is no attraction to a desired goal since the goal position is unknown. Also, only relevant areas are considered; that is, obstacles in the direction opposite to the movement of the ropod are not relevant. The areas that are considered according to the position of the Falcon's end-effector and intended direction of the ropod can be seen in figure 5.5. In this figure $z < 0$ means the forward movement intention of the ropod, $z > 0$ means moving backwards, $x > 0$ means turning right, and $x < 0$ means turning left. Figure 5.5 b also shows the measurement area of the LIDAR sensor covering the considered areas with the convention of the measurement angles $\theta_{d,i}$ corresponding to a measured distance point d_i. Furthermore, the maximum force over all sensed obstacles is used and not the average of the



Use area when:	Area
z < 0	I and II
z > 0	III and IV
x < 0	I and III
x > 0	II and IV

c)

Figure 5.5: a) Considered areas according to the movement of the ropod. b) LIDAR sensor scanning range (red) covering the areas. c) Area conditions

sum of the forces due to obstacles, to prevent collisions. The max() function used below returns the value with the largest absolute value.

The the environmental force $F_{env} = [F_{env,x}, F_{env,z}]^T$ can be calculated using the x,z positions of the Falcon's end-effector and the LIDAR distance points d_i with corresponding scanning angle $\theta_{d,i}$ using:

$$F_{env,z} = \begin{cases} k_3 \cdot z \cdot \big(\max\limits_{i=1}^{n}(\, h_1(d_i) \cdot \frac{d_i - d_{max}}{d_{max}} \cdot sin(\theta_{d,i}) \,) \big), \text{ if } z \geq 0, \\ -k_3 \cdot z \cdot \big(\max\limits_{i=1}^{n}(\, h_2(d_i) \cdot \frac{d_i - d_{max}}{d_{max}} \cdot sin(\theta_{d,i}) \,) \big), \text{ if } z < 0 \end{cases} \tag{5.5}$$

$$F_{env,x} = \begin{cases} k_4 \cdot x \cdot \big(\max\limits_{i=1}^{n}(\, h_3(d_i) \cdot \frac{d_i - d_{max}}{d_{max}} \cdot cos(\theta_{d,i}) \,) \big), \text{ if } x \geq 0, \\ -k_4 \cdot x \cdot \big(\max\limits_{i=1}^{n}(\, h_4(d_i) \cdot \frac{d_i - d_{max}}{d_{max}} \cdot cos(\theta_{d,i}) \,) \big), \text{ if } x < 0 \end{cases} \tag{5.6}$$

where,

$$h_1(d_i) = \begin{cases} 1 \text{ , if } d_i < d_{max} \text{ and } sin(\theta_{d,i}) < 0. \\ 0 \text{ , otherwise.} \end{cases}$$

$$h_2(d_i) = \begin{cases} 1 \text{ , if } d_i < d_{max} \text{ and } sin(\theta_{d,i}) > 0. \\ 0 \text{ , otherwise.} \end{cases}$$

$$h_3(d_i) = \begin{cases} 1 \text{ , if } d_i < d_{max} \text{ and } cos(\theta_{d,i}) > 0. \\ 0 \text{ , otherwise.} \end{cases}$$

$$h_4(d_i) = \begin{cases} 1 \text{ , if } d_i < d_{max} \text{ and } cos(\theta_{d,i}) < 0. \\ 0 \text{ , otherwise.} \end{cases}$$

Where d_{max} is the maximum adjustable distance up to where the environmental force affects the ropod movement. The calculated force due to an obstacle is inversely proportional to the measured distance between the ropod and the object d_i. Moreover, k_3 and k_4 are scaling constants to adjust the influence of environmental force. Note that if these constants are very large, it is harder to control the motion of the ropod since the environmental force is too large. However, if the constants are very small, the human operator does not feel any force relating to obstacles. The values of k_3 and k_4 have to be determined using experiments, keeping the maximum possible output force that the Novint Falcon can provide in mind. Also, note that if there is no object in the considered areas, and if Falcon's end-effector is in its origin, the environmental force is zero. Moreover, the maximum values for $F_{env,z}$ and $F_{env,x}$ are $\pm\, k_3 \cdot z$ and $\pm\, k_4 \cdot x$ respectively.

As an example, when the ropod is moving forward ($z < 0$), the max() part in equation 4.3 defines the maximum value of the z components for all the forces due to the measured obstacles in the front area of the ropod and is used as feedback to the operator. Similarly, the maximum value of the x components is determined using the corresponding areas. Together, they should reduce the commanded velocity commands in the direction of the sensed object.

To conclude this chapter, it is chosen to control the ropod platform on platform-level instead of controlling the wheels individually to reduce undesired oscillations and because it is believed to be more intuitive for the operator. Also, the kinematic ropod controller is described to control the velocities of the ropod on platform-level using the raw data sent over EtherCAT to and from the smart-wheel units. Hence, this controller, along with the EtherCAT communication, ensures that the ropod platform functionality can be used properly. Furthermore, this chapter describes what is needed to convert the raw inputs and outputs of the Novint Falcon to useful forces and positions in cartesian coordinates depending on the interfacing library LibNiFalcon. Hence, these conversions, together with the firmware and USB communication, ensure that the Novint Falcon functionality can be used properly.

Everything concluded up to here describes the *non-application-specific* (generic) parts of the system design and hence, are ready to be composed into generic software components. That is, the hardware devices, communication resources, and their controllers are ready to be integrated into software so the devices' functionality can be used for various applications. The described controller to keep the end-effector in the origin of the Falcon's workspace and the two mappings to generate velocity commands and environmental force feedback are the *application-specific* parts of the system. This division in generic and application-specific parts is essential in the software architecture design to achieve modularity of the system. Furthermore, this chapter describes how the actuation forces of the Novint Falcon to the operator and to the ropod platform are limited and controllable, achieving the first part of the third objective.

The above mappings of generating velocity commands and environmental force are just somewhat simple examples chosen as a start for this research. However, it is expected that these mappings could provide good insights into the effects and performance of the environmental force feedback in the dead-end hallway application. Note that many different mappings could have been used to generate velocity commands and environmental force, but the focus of this research lies more on the generic design approach of the overall haptic teleoperation system. As an extension for future work, for example, the intended trajectory could be generated en let the environmental force act on the intended trajectory, or task-specific requirements could be used to interpret the environmental force accordingly, like for example, driving through a doorway, which is predicted to be very hard using the approach as mentioned above.

6. Software architecture design

This chapter describes the design of the software architecture of the teleoperation system and hence, the design of the software components. Furthermore, this chapter aims to take the predictability and modularity objectives into account for each software component and the overall design of the software process. All the before-mentioned resources, controllers, and mappings in this report must come together in software to design and implement the overall haptic teleoperation system. For the software design of the teleoperation system, it is assumed that the software has to run on and use the resources of the provided computational computer mentioned in chapter 4.1. Hence, for this design, the operating system is Linux with the real-time patch to convert the kernel into a fully preemptible kernel.

The software architecture design for this research approach uses the generic deployment architecture model to bring structure to the overall application. The model allows for the deployment and coordination of threads, activities, and algorithms easily. Furthermore, for the coordination of the application, Life Cycle State Machines (LCSM), Petri Nets, schedulers, and flags are used on the different deployment levels. The software template code given for this research has minimally implemented all the above-mentioned deployment and coordination models on different levels and is used as a basis for designing the software architecture for this research. So for the implementation of the design (chapter 7), this template has to be "filled in" and supplemented correctly. The design helps to determine the generic and application-specific parts of the haptic teleoperation system. It should be mentioned that not all parts of the given template code are modified for this design. However, the template comprises the minimal amount of code needed, and hence only code is added to the template. Furthermore, the template code is written in the programming language C and is therefore chosen as the leading programming language for this design. This chapter describes how the teleoperation design is translated into generic and application-specific software components and their deployment and coordination. More details on the generic software architecture design approach and the definitions used can be found in chapter 11 of [4]. Chapter 2 briefly describes the Life Cycle State Machine (LCSM) coordination mechanism and its states together with an example of the Petri Net coordination model.

6.1 Deployment architecture design

The used generic deployment architecture model advocates a particular pattern about how to compose a generic software architecture. It uses the following main building blocks: a process, threads, activities, activity schedulers, and algorithms. Each **process** executes by design at least one thread, which is so-called the "main thread" or "thread 0" and can have multiple threads. Each **thread** consists of an event loop that triggers the computations in all activities and an LCSM that realizes the decision about which and when activities the event loop has to trigger. The coordination for the thread's LCSM and triggering of activities is realized using a Petri Net coordination mechanism. Each **activity** has one LCSM to coordinate its outwards-facing behavior, one or more algorithms to realize its internal behavior, and zero or more Petri Nets to coordinate these algorithms to coordinate these algorithms.

As an attempt to further explain the above, an overview of the deployment architecture for the generic design of the teleoperation system for this research can be seen in figure 6.1. It can be seen that the design uses one process with four threads and, in total seven activities. Each thread uses an activity scheduler with at most two sets (IDs) of activities. The expected communication data input and output between each thread and hence, which thread has to communicate data with which thread, can also be seen in the figure. This section describes the decomposition into threads and activities. The following section describes the coordination between threads and activities.

6.1.1 Time triggered threads

Creating a thread is considered expensive when designing a software program. It requires a fair bit of work; for example, memory has to be allocated and initialized, and too many threads in a process can be crucial for performance. Therefore, the amount of threads used for the teleoperation design is kept to a minimum. In a distributed system, like this haptic teleoperation system, the asynchronous nature of the hardware resources and their communication interfaces introduces asynchronicity into the system. Therefore, this software design adopts the multi-threading architecture, which is one form of asynchronous programming. Multi-threading allows for multiple threads to be created within one process. The threads execute independently but concurrently share process resources. In asynchronous programming, the occurrence of events is independent of the main process, and it is not known when data will arrive and where it will be used. In synchronous programming, the order in which functions are programmed in the code is also the order in which they will execute, and the data needed for these functions need to be available at any moment.

In this software design, it is chosen to create four time-triggered threads. Time-triggered threads are threads that run their event loop at a given cycle time. In this event loop, the thread first updates its LCSM state and next executes the Communicate, Coordinate, Configure, and Compute (4C) functions corresponding to the LCSM state of the thread, resulting in the triggering of the set of activities scheduled on the thread. After each event loop cycle, the thread sleeps for the remaining time of the set cycle time. The cycle time of a time-triggered thread can be set in nanoseconds using the Linux *clock_nanosleep()* function. This function allows the thread to sleep for an interval specified with nanosecond precision. The choice of cycle time values of each thread is mainly based on the hardware devices' communication interface and desired controller rates and are, therefore, closely linked to activities deployed on each thread.

- **Thread 0** is the "main thread" of the process and takes the responsibility to mediate all process resources in the form of an application mediator activity (see next section). Furthermore, this thread handles the asynchronous interfacing with the Novint Falcon device and runs the haptic application controller (section 5.3). The cycle time for this thread is chosen to be 1000000 ns (1000 Hz) because the internal microcontroller of the Novint Falcon maintains the communication at 1000 Hz and expects a force input at that frequency.

- **Thread 1** is an asynchronous thread that handles the interfacing with the Sick TiM561 LIDAR sensor and the parsing of the communication telegrams to usable distance values. The LIDAR operates at a scan frequency of 15 Hz, and hence, the cycle time for this thread is also set to 15 Hz.

- **Thread 2** is the so-called "real-time communication" thread. This thread handles the EtherCAT fieldbus communication and the ropod platform velocity controller at a cycle speed of 1000 Hz. This synchronous thread needs to be non-blocking (in the running state of the LCSM) by design. EtherCAT is one such communication technology that allows for synchronous execution using memory-mapped I/O. More details on the design of this real-time thread can be found in 6.4. The real-time scheduling priority for this thread is set to 39 (see chapter 2 and appendix A).

- **Thread 3** is the "EtherCAT watchdog thread" and is also designed to be a real-time thread because of its interaction with the EtherCAT fieldbus. The goal of this thread is to monitor the internal states of all the initialized EtherCAT slaves on the EtherCAT network. Furthermore, this thread handles all the slave errors and tries to recover slaves when their state is not in the internal operational state. The real-time scheduling priority for this thread is set to 40. Note that this thread has the highest priority (even higher than thread 2) because the goal is to *monitor* the communication of thread 2.

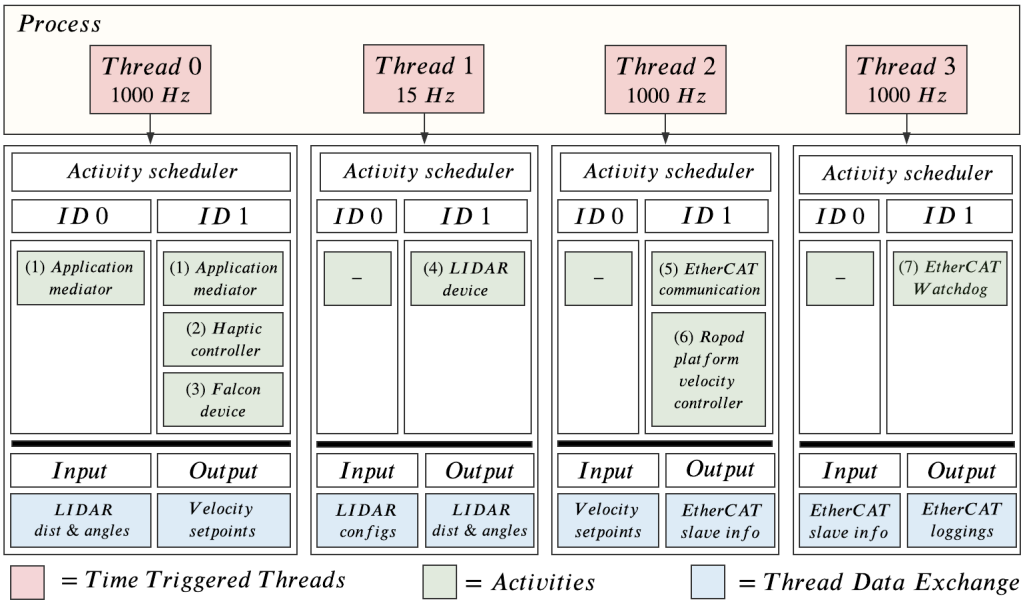


Figure 6.1: Design overview of the deployment of four threads with seven activities for the haptic teleoperation system

6.1.2 Activity schedulers

The activities are executed on a thread according to the activity scheduler. The event loop of a thread coordinates, configures, or computes (triggers) the activity scheduler according to the LCSM state of the thread. Each thread has one activity scheduler that selects which set of activities are deployed on the corresponding thread. The sets of activities are indicated using ID numbers and can be seen in figure 6.1 for this design. For example, if activity set ID 1 is selected for thread 2, the activity scheduler schedules activities 5 and 6 to start (be enabled) at thread 2. However, note that the activities will only start if thread 2 on itself is ready to start the activities (thread 2 is in LCSM running state). Changing between activity scheduler IDs can be done at run-time so that different sets of activities can be executed at different time instances depending on the application task or available resources. This design uses a maximum of two scheduler IDs per thread, and ID 0 is set to be enabled by default at the start of the process. This is done for two main reasons:

1. For thread 0, ID 0 contains the application mediator activity and first has to run solely so that it can create the other threads and is ready to manage the coordination of the whole application (see next section). After that, ID 1 can be selected for thread 0, and other activities can be triggered. Note that ID 1 of thread 0 still contains the application mediator activity because this activity has the unique ownership of the decision-making of the coordination of the whole application and therefore has to run continuously.

2. For the other three threads, ID 0 has no set of activities with the intent to have an option to “disable” the activities. ID 1 contains the set of activities that should run when the activities are not disabled. Hence, the switching between ID 0 and ID 1 in these threads are disabled/enabled triggers that could be used for experimental or debugging purposes.

Activities scheduled on the same thread are triggered to execute in a serialized way and hence, cannot write data to the same memory address at the same time. This can, however, occur between functions in different threads resulting in a race condition which is undesired (see 6.3).

6.1.3 Activities

Activities interact with each other via software and communication services they provide to each other. In this design, a total of seven activities are designed. From here on, the terms activity and component are considered the same. Each activity has to realize a subsystem's behavior, taking into account the available information and resources. The activities are chosen such that each *hardware device interface* and each *controller* has its own activity. This design choice is also directly linked to the design choice made for the number of total time-triggered threads. Note that all threads are time-triggered, so the corresponding activities scheduled in one of those threads are also time-triggered in the same way. Furthermore, as explained above, there should be one application mediator activity to coordinate the whole application and one activity for the realization of the EtherCAT monitoring (watchdog). This makes for a total of seven activities and the activities with their scheduling ID can be seen in figure 6.1. Note that the haptic controller activity (2) and the Novint Falcon device activity are serialized and scheduled within the same schedule ID. The same goes for the EtherCAT communication activity (5) and the Ropod platform velocity activity (6). The reason for this serialization is to keep the controllers close to the controller's input and output values to minimize the unnecessary asynchronous exchange of big data sets between threads. Furthermore, the computations of the serialized activities in the real-time thread must be executed immediately to minimize the latency the EtherCAT communication activity [13]. Figure 6.2 provides an overview of all the activities within the block scheme of the haptic teleoperation design together with the data exchange between activities/threads.

As mentioned above, each activity has one LCSM to coordinate the availability of internal resources for the provision of external services. That is, to make sure that the activity's own capabilities are configured appropriately before it provides services to others. The states of the LCSM pattern of each activity have to be designed, and this is one of the most essential steps in this research. Each state in the LCSM pattern is designed to be a function that holds the code that executes synchronously when the LCSM reaches that state. The application mediator activity coordinates the changes in and switching of the activity's LCSM states. All states together are responsible for performing the overall purpose of the activity. Next, all the above-mentioned activities and their purpose are described. Furthermore, an explanation is given for only the LCSM state functions that are supplemented on top of the minimum required for the LCSM to work correctly. Note that all LCSM state functions in an activity, supplemented or not, have to be implemented for the application to work correctly and minimally consist of the switching options to other states.

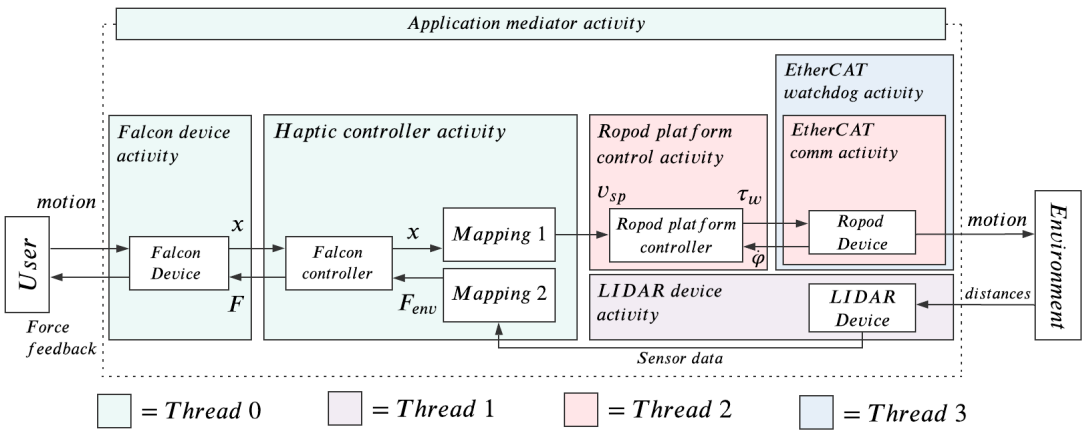


Figure 6.2: Block scheme of the haptic teleoperation design with the subdivision into software components and threads

1. **Application mediator activity:** This activity can be called the brain of the whole application. This activity has to be executed on the main thread that comes with the process. In this multi-activity program, the application mediator activity is introduced to take the responsibility of all the configuration and coordination of all threads and activities and *couples* them together. In this way, the ownership of the decision-making is decoupled and centralized. For the coordination of the threads and activities, this mediator activity uses Petri Nets and flags (see next section). Moreover, this mediator activity is responsible for configuring all process resources like log files (see section 6.5) and data streams. It will also start up the other three time-triggered threads and configure and manage their LCSMs. When all threads are configured, this activity signals the scheduled activities that they can start (creation state of LCSM) on their corresponding thread. Furthermore, this activity has to monitor and coordinate the LCSM of all activated activities. When a shutdown event is received, this activity will correctly shut down all activities before cleaning the threads and process resources. By designing the application mediator in this way, none of the coordinated activities themselves must know about how their execution is coordinated with other activities, and hence, the activities do not have to be changed when integrating them. This approach increases the composability of the activities and hence, the whole application.

   Create: This state has to initialize the amount of threads that have to be created and coordinated, allocate memory for the values of all the activity state structure members (see section 6.3), and allocate memory for the following data streams: The ropod platform velocity setpoint, the LIDAR distance values, the Falcon's end-effector position, the smart-wheel sensory input, and the smart-wheel output control commands. Moreover, this state has to fully initialize the application mediator Petri Net.

   Resource conf: Here the signal for starting the other three threads has to be sent.

   Running: This state has to configure and open the log files for all activities and configure the streams. Furthermore, it has to initialize each thread's coordination Petri Net. When the configurations are successful, this state has to start and link thread 1, and real-time threads 2 and 3 to the main thread and assign the log files to each thread and activity. When a shutdown signal is received, this state has to signal all threads to stop their event loop.

   Cleaning: This state has to be responsible for, cleaning allocated memory (deallocating), destroying the created logfiles and thread coordination Petri Nets. The threads on itself are responsible for destroying the activity scheduler Petri Nets.

2. **Haptic controller activity** This activity uses the Novint Falcon, Sick LIDAR sensor, and the ropod platform devices as a *resource* to close the haptic force feedback loop and to perform the haptic teleoperation application. When the above devices are ready to provide their services (they are in the running LCSM state), this activity is responsible for controlling the end-effector of the Novint Falcon according to the upper control loop of figure 5.3. This activity is also responsible for computing the two mappings discussed in section 5.3. Hence, the environmental force and ropod platform velocity setpoints have to be calculated. Consequently, this activity requires end-effector position input data from the Novint Falcon device activity and measured distance input data from the LIDAR device activity. Both required data streams are asynchronous, and therefore, these data exchanges are designed to be event-triggered. The LIDAR data has to come from another thread and hence have to be exchanged in a thread-safe way. The output data of this activity is the total force that has to be sent to the Falcon device activity and therefore felt by the operator. Furthermore, the platform velocity setpoints for the ropod platform have to be sent to the ropod platform velocity controller activity, which also is scheduled in another thread.

Create: This state has to allocate memory for the PD-controller parameters.

Resource conf: This state has to do the bookkeeping in making sure all the resources needed for the haptic controller are ready to provide their services. These resources are provided by activities (3), (4), and (6), and hence, these activities have to be in the running state of their LCSM. Coordination flags to indicate which activities are in their LCSM running state, have been chosen to realize this bookkeeping, and these are coordinated via the application mediator activity.

Capability conf: This state is responsible for first running the Novint Falcon controller to keep the end-effector of the Falcon in the origin of its workspace. Hence, the activity is first configuring its own capabilities. After this, the activity is now ready to send platform velocity setpoints to the ropod and environmental forces to the user.

Running: This state executes the complete haptic controller. That is, the Novint Falcon controller, the calculation of the environmental force, and the generation of platform velocity setpoints for the ropod platform. Consequently, this state has to handle the incoming events for new Novint Falcon and LIDAR data. When new data from the Falcon arrives, the controller has to update its values, and when new LIDAR data arrives, the environmental force has to be calculated.

Cleaning: Here, the allocated memory in the create state has to be deallocated.

3. **Falcon device activity** This activity should handle the interface with the Novint Falcon device and makes the Falcon ready as a resource for the provision of services for the haptic force feedback controller activity. Furthermore, this activity is responsible for the calculations described in the lower part of figure 5.3 named "Falcon software". Hence, the Jacobian and kinematics are calculated in this activity. For interfacing with the Novint Falcon, the library LibNiFalcon is used and explained in more detail in chapter 7. In short, this library provides the basic functionality to connect to the Novint Falcon (like sending and reading useful device data and kinematic functions) and to load firmware to the internal microcontroller. button behaviour

Resource conf: This state has to set the communication, firmware, kinematics, and gripper behaviors of the main LibNiFalcon class[1], which is the FalconDevice class. This class ties together all of the components in libnifalcon to create a simple, usable single object for accessing and controlling the Falcon. This state also has to set the incoming end-effector position and outgoing force values to zero before using the Falcon for safety reasons. Furthermore, this state has to load and check the firmware on the internal microcontroller.

Capability conf: This state is responsible for the calibration of the Falcon device (see section 4.2.1). The calibration should be brought to this state because from the perspective of the haptic control activity, the calibration of the Falcon means that this activity is configuring its own capabilities before providing its service as a resource to the haptic controller activity.

Running: Here, the so-called "IO loop" has to be executed. This implies that for each loop, it has to be checked if new data is available or can be sent using USB polling, callback functions, and timeouts. The raw data then has to be formatted to useful encoder values and motor voltages. This data has to be used in the kinematic functions to get the positions of the end-effector and forces in cartesian coordinates (see lower part of figure 5.3 named "Falcon software"). Furthermore, these values have to be written to a stream for the haptic controller activity to use them.

Cleaning: This state has to stop and close the USB connection with the Novint Falcon and has to check if the device is closed correctly and has no errors.

[1] A software class is meant here: A C++ class is the building block that leads to Object-Oriented-programming.

4. **LIDAR device activity** This activity should handle the interface with the Sick TiM561 LIDAR sensor device and makes the device ready as a resource for the provision of services for the haptic force feedback controller activity. The official generic C++ software library from the SICK company is used for interfacing with the LIDAR sensor (see chapter 7). This library provides the basic functionalities for connecting with the sensor via TCP Ethernet connection and collecting distance values.

   Resource conf: Here, the LIDAR sensor has to be initialized. The sensor parameters, like the IP address of the device and the protocol for the communication datagram has to be set. The TCP connection has to be initialized and established and and error codes handled correctly. The initialization of the sensor consists of a list of commands that are sent to the sensor to set the desired sensor settings on the internal microcontroller of the sensor before normal operation begins. Examples of settings that have to be set before operation are the maximum output range and angular resolution and have to be send each time the sensor starts up. The full list of commands that has to be send for initialization can be found in appendix E.

   Running: This state checks if there is new data available on the TCP connection and converts that data to useful values like distances, corresponding scan angles, and timestamps. Furthermore, parsed sensor data has to be written to shared memory in a thread-safe manner so that the haptic controller can access these values. The flag for new LIDAR sensor data has to be set high so that the haptic controller knows that new data is available.

   Cleaning: This state has to stop the LIDAR and close the TCP communication and wait for confirmation that this connection is closed.

5. **EtherCAT communication activity** This activity has to be responsible for the real-time communication with the ropod platform device over the EtherCAT fieldbus. As described in chapter 4, the ropod platform has four smart-wheel units, which all act as EtherCAT slaves. This activity has to initialize all of the four slaves and has to send and receive data at a fixed cycle time. For the connection with EtherCAT, the Simple Open Source EtherCAT Master (SOEM) library is used (see chapter 7).

   Resource conf: This state initializes the EtherCAT connection. The network card is bound to an internet socket, and after that, the EtherCAT slaves are searched for and configured when found. The internal state machine of the EtherCAT slaves is triggered and monitored. This state should wait until all slaves are in the operational state of their internal state machines.

   Running: This state transmits and receives processdata from slaves. Furthermore, this state synchronizes with the internal EtherCAT clock.

   Cleaning: This state has to close the EtherCAT communication and trigger the internal EtherCAT slave state machine.

6. **Ropod platform velocity controller activity** The ropod platform controller described in section 5.1 is put in this activity. This activity uses the ropod as one of its resources.

   Resource conf: This state has to initialize all of the controller parameters of the platform velocity controller and the four wheel velocity controllers. It also has to initialize the torque distribution class and the kinematic functions.

   Capability conf: This state has to handle the calibration of the four smart-wheel incremental pivot encoders as described in section 4.3.1. Again, the calibration has to be done in this state because from the perspective of the haptic control activity, the calibration of the pivot encoders means that this activity is configuring its own capabilities before providing its service as a resource to the haptic controller activity.

Running: Here, the actual control values are updated for each controller. The input values from the smart-wheel units are read and used to determine the control command values that has to be sent to the smart-wheels via the EtherCAT communication activity.

7. **EhterCAT watchdog activity** The goal of this activity is to monitor the internal states of all the initialized EtherCAT slaves on the EtherCAT communication maintained by the EtherCAT communication activity. Furthermore, this thread handles all the slave errors and tries to recover slaves when their state is not in the internal operational state.

   Running: This state monitors the internal states of the EtherCAT slaves and handles accordingly. Also, this state logs all the events like errors and state-changes of the EtherCAT slaves. Furthermore, the time of each internal clock for EtherCAT slave are monitored to check if the slaves are still in sync, which is essential for synchronous EtherCAT communication.

6.2 Application coordination architecture

As described in the section above, the application mediator activity is introduced as the owner of all coordinations of the whole application. The mediator can execute the computations of both the monitoring to set flags, and the coordination to make a control flow decision. By doing so, none of the coordinated components itself must know about how its execution is coordinated with other components. This results in a better composable activities because it does not have to be adapted when it is integrated into a larger context.

The coordination architecture of this software architecture design for the haptic teleoperation can be seen in figure 6.3. In total, there are Petri Net coordination models in the design, and together they fulfill the coordination of the whole application. The three Petri Net models are:

1. Application mediator Petri Net: This Petri Net model is responsible for all the coordination of the whole application. It coordinates the LCSMs of all the activities, it coordinates the LCSM of all threads via the Thread LCSM Petri Net, and it coordinates the activity scheduler of each thread via the Thread activity scheduler Petri Net.

2. Thread LCSM Petri Net: This Petri Net model couples the LCSM of each of the four threads to the application mediator Petri Net using Flag map 1 for each thread. The Flag map consist of all flags that are used to coordinate between the two Petri Net models.

3. Thread activity scheduler Petri Net: This Petri Net model couples the event loop (consisting of coordinate, configure, and compute) of the activity schedulers of each of the four threads to the application mediator Petri Net using Flag map 2 for each thread. The contents of Flag map 1 and Flag map 2 can be found in appendix E.

It should be mentioned that the given software template code already has designed and implemented the three Petri Net coordination models above, including the two Flag maps to couple them. However, in the template, only two threads with an activity scheduler each and a total of four activities have to be coordinated. The existing Petri Net models of the given template code are used as a basis. They are extended to four threads with an activity scheduler each and, in total, seven activities. This means that for this design, two threads, two activity scheduler, and three activities have to be added to the template and coordinated. Hence, the Thread LCSM Petri Net model has to be extended with two threads to coordinate their LCSM. Furthermore, for each of the two added threads, an activity scheduler has to be added, and these have to be coordinated by the Thread activity scheduler Petri Net. The application mediator Petri Net has to coordinate all the newly added threads and activity schedulers. Hence, Flag map 1 and Flag map 2 of the two added threads have to be added to the application mediator Petri Net to couple the coordination of the three Petri Net models. Furthermore, new flags have

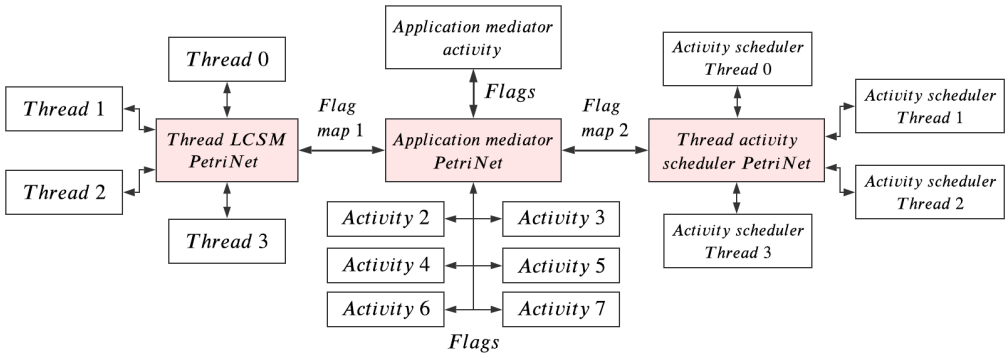


Figure 6.3: Application coordination via the application mediator using flags and flag maps as coordination mechanisms between Petri Net models

to be added to the application mediator Petri Net for the coordination of the LCSMs of the three newly added activities. An example of an added flag used to coordinate the LCSM of the EtherCAT commutation activity is: **disabling_ethercat_communication_activity**.

6.3 Data exchanges between activities

The different activities have to exchange information with each other. For this, so-called activity state structures are introduced. Each activity has at most one activity state structure to store its "state" and could be changed at run-time. These structures are used to store data that has to be exchanged with other activities and used for logging. Data from different activity state structures can be coupled using a stream. The application mediator activity has to be the only activity that can access the activity data structures of all activities. Hence, all streams must be initialized and coordinated by the application mediator activity. Many data stream models exist, but this research uses shared data memory to exchange data between activities. That is, two activities read and write data to the same memory address. For example, the application mediator activity allocates the memory for the data that has to be exchanged. Two activity state structures both have a data slot available for the exchanged data. The application mediator then has to let these two data slots *point*[2] to the same memory address, which is the memory address of the allocated memory for this data exchange by the application mediator activity.

Shared memory is one of the simplest ways to exchange data between activities. However, care is to be taken that one activity does not write data at the same time that another one is reading or writing that same data. This so-called race condition can only happen when two or more threads access shared data and try to change it simultaneously. Two activities that run on the same thread are always executed in a serialized way and hence, can never access the same data simultaneously (they are thread-safe). However, it is possible that activities that are executed on different threads can access and try to change shared data simultaneously, and this unpredictable behavior has to be prevented. This is done using Mutual exclusion locks (mutexes). Mutexes are used to ensure exclusive access to shared data between threads and are one of the simplest forms of a synchronization mechanism. Moreover, mutexes are essential in real-time programming. Typically, a mutex is acquired (locked) and released (unlocked) around the code that wants to access the shared data. Only one thread may have the mutex locked at a given time so that no other thread can access this locked code region. This research uses the mutex functions from to the POSIX thread (pthread) library, and, in particular the pthread_mutex_lock() and pthread_mutex_unlock()

[2]Pointing to data is done using a pointer in C programming, which is a variable whose value is the address of the memory location of another variable.

functions. In this research, the data that has to be exchanged between threads and hence need a mutex are the ropod platform velocity commands and the LIDAR distance measurements from and to the haptic control activity.

6.4 Creating real-time threads

As mentioned above, the EtherCAT communication activity and EtherCAT watchdog activity both run on separate threads. These threads are set to be real-time and have to be created and treated as described in section 2.3. A real-time thread executes all Computations that must be executed immediately, all Communications that are done via non-blocking memory-mapped I/O, and the Coordination that is triggered by the real-time event loop itself and must be dealt with immediately. For example, deciding to switch to a fail-safe control mode. Creating a real-time thread is done in the application mediator activity (just as the non-real-time threads) and has to be done in the following steps:

1. Lock memory using the function mlockall().
2. Initialize pthread attributes to default values using the function pthread_attr_init().
3. Set the specific stack size for the thread using the function pthread_attr_setstacksize().
4. Create the pthread using the specified attributes with function pthread_create() and attach the time-triggered event loop functions of the threads to the created pthread.
5. Set the priority and scheduler policy of the pthread using the function pthread_attr_setschedparam(). The scheduler policy is chosen to be SCHED_FIFO. The priority for the EtherCAT communication thread is set to real-time priority 39, and the EtherCAT watchdog thread is set to real-time priority 40.
6. Check the return value of the function pthread_attr_setschedparam() function to check that no errors have occurred.

Hence, the EtherCAT watchdog thread gets the "highest priority" in the process and can be given to one thread only. If two threads are given the same numerical priority, only one can really be executed first. Note that as a result, the EtherCAT watchdog thread could be called the **hard** real-time thread, and the EtherCAT communication thread could be called the **soft** real-time thread but there is no straightforwardness in these definitions.

6.5 Logging

The given software template code already has a mechanism for logging important events or parameters in activities or threads (loggers). Logging is used for debugging and monitoring purposes. The template provides the facilities to log messages to an output file with five verbosity levels. These levels are: *Always, Error, Warning, Info,* and *Debug* and can be chosen to indicate the type of logged message. The logging mechanism uses the C library function fputs() to write a C string to a standard C stream in a fully buffered way. Because some threads run at 1000 Hz, this logging needs faster access to the Linux file system. Hence, for the logging, a temporary filesystem from the RAM memory of the computational computer should be mounted. This temporarily puts the log files in the faster RAM memory so they can handle the fast cycle times of the time-triggered threads. Another advantage of this logging is that it provides a very accurate timestamp that could be used to track cycle times. As mentioned above, the application mediator activity configures and opens all the file descriptors of the loggers of all threads and activities, so they can be used in the corresponding LCSMs of the threads and activities. In this design, one logger is added to each of the added threads and activities.

In conclusion, this chapter describes the software architecture of the haptic teleoperation system using the modular and generic approach adopted from the template. Hence, this chapter describes how objective 2 is achieved. This chapter describes the deployment and coordination of, in total, one process, four time-triggered threads, and seven activities. The hardware devices, communication resources, and controllers from previous chapters are designed into activities. The activities are divided into generic and application-specific software activities. From figure 6.2 it can be concluded that the Haptic controller activity is the application-specific part of the system, and the other six activities are the generic parts of the system. These activities are deployed on threads using the multi-threading architecture, which provides deterministic ownership of data and resources, allows for real-time performance and reconfiguration of activity deployments. All the threads are deployed on one system process via the main thread.

The deployment architecture in this design clearly requires coordination between threads and activities. The application mediator activity is introduced to take the responsibility of all the configuration and coordination of all threads and activities and couples them together. In this way, the ownership of the decision-making is decoupled and centralized. For monitoring and coordination, the application mediator activity uses Flags, Petri Nets and the LCSMs of each thread and activity. All activities and threads are designed to have one LCSM to coordinate the internal resources for the provision of external services. This results in a modular system since the threads and activities do not have to know about how their execution is coordinated and others do not have to know about the exact workings of these components and interactions inside of the components but only know how it interacts with other components from the outside. Furthermore, knowing the architecture of each individual thread and activity and the behavior of the interactions between them results in a better predictable system. The choice to create two real-time threads for the EtherCAT communication and watchdog activity is also made because of the deterministic nature of real-time threads and the synchronous communication protocols like EtherCAT. To indicate how deterministic these threads are, section 2.3.5 measures the worst-case latencies of the system and a rough approximation of the worst-case latencies of the application. While these two real-time threads can be said to be more deterministic, the other two threads have to deal with asynchronous communication. They are, therefore, less deterministic because of the reliance on the uncontrolled embedded electronics of the hardware devices. To deal with this non-deterministic behavior, coordination flags are introduced via the application mediator activity to signal if new device data is available to other threads so that the system can safely anticipate.

Activity state structures are introduced to centralize the data that has to be accessible by other activities. The activity state structures are coupled via shared memory that is owned by the application mediator activity. The application mediator activity is the only activity that can access the activity state structures of other activities, and hence, again, this results in a more modular and predictable system. Mutexes are used to protect shared data between threads to prevent race conditions, which supports the deterministic requirement. Mutexes are used because of their simplicity, but a disadvantage of mutexes is that it requires waiting time, could force the thread to block, or in worst-case could introduce a deadlock. This is in contradiction with the non-blocking design of the synchronous real-time threads, and hence for future work, all mutexes have to be replaced by lock-free asynchronous streams, like a ring buffer.

The last requirement is that the design should provide (re-)configurability at runtime. All threads and activities are designed with an LCSM. LCSMs have a resource configuration state and can be transitioned to and from multiple times during the life time of a thread or activity to re-configure their resources with the goal to provide other services for other tasks or applications. The template uses c structures across the complete code with function and parameter pointers (like the execute functions for each LCSM state) and could in theory be changed at runtime, but care has to be taken in when and how these datastructures are changed and this should best be coordinated by the application mediator to maintain the predictably requirement. Furthermore this chapter introduces activity schedulers, which also can be changed at runtime so other sets of activities can be triggered to be executed depending on the task or application.

7. Implementation

This chapter describes the implementation of the real haptic teleoperation system design based on the software architecture model as described in chapter 6. The aforementioned given template code is used as a basis and altered for this implementation. The implementation in code of the altered coordination Petri Net models, the LCSMs of the added activities/threads, and the creation of real-time threads are very consistent with the designs described in chapters 5 and 6. Hence, this chapter will mainly focus on the challenges that arose while implementing the controllers and software architecture. Furthermore, this chapter will describe the used external dependencies and the implementation of the ropod platform velocity controller in C++, which was not present in the currently available ropod platform controller code. Also, an attempt is made to measure the latencies of the real-time platform and software application. Finally, the inconsistency in the libusb communication of the Novint Falcon and a convenient shutdown command are described.

For the implementation, it is chosen to use Visual Studio Code to write all the code. This is chosen because of its easy integration with GitHub for backing up the code and doing the version management. CMake is used for compiling and linking the code and for managing all of its dependencies (like libraries). CMake is an open-source, platform-independent tool used to control the software compilation process using simple and compiler-independent configuration files. These configuration files generate the build systems (makefiles) for the code. A build system is a set of text commands for the compilation and linking of software code and turning the code into an executable file. As a result of using CMake, the designed program can be compiled and linked on other computational computers easily.

7.1 Libraries and dependencies

For the interface with the Novint Falcon device, Sick TiM561 LIDAR sensor, and the EtherCAT hardware, it is chosen to not write all the programming code from scratch but to use available open-source libraries. Extensive research has been carried out into which libraries can best be used for the implementation of this research. The main requirements for choosing the correct libraries for this research are that the libraries should be open-source, should be able to be integrated into C/C++ programming language code, and should provide the minimal functionality necessary for the implementation of the haptic teleoperation system described in the designs in earlier chapters.

7.1.1 Novint Falcon interface

For the interfacing of the Novint Falcon, the library LibNiFalcon [31] is chosen and used. LibNiFalcon is an open-source, cross-platform development library for the Falcon and uses a version of the no longer supported official driver from Novint. Other libraries that are looked at are the low-level interfaces Metuunt, CHAI3D, HDAL, and the high-level interface H3DAPI. An overview of the current developments in haptic APIs can be found in [32]. LibNiFalcon is chosen because it is an extensive low-level interface that also includes the driver for the Novint Falcon firmware. This results in high flexibility when programming the Novint Falcon. LibNiFalcon also includes efficient solutions for the inverse and forward kinematic equations as described in chapter 5. Furthermore, LibNiFalcon provides USB communication by using the libusb library and connects to the right USB port by receiving the USB device descriptor from the Falcon. Libusb is a cross-platform C library and provides generic access to USB devices of the internal FTDI chip in the Novint Falcon. For the implementation, the functionalities provided by the library are kept to a minimum. That is, the library is used for loading firmware, handling the USB connection, calibration process, solving the kinematic equations, and converting the raw data to useful voltage and encoder values.

7.1.2 Sick LIDAR interface

The official SICK Scan Base library [33] is a generic C++ library for interfacing SICK laser scanners and is used to interface the SICK TiM 561 LIDAR sensor. The library is also a low-level driver and is maintained by SICK itself. However, all the code of the library is openly accessible and adjustable. For the implementation, the Sick Scan Base library is slightly modified for the particular use of this sensor. By default, the library initializes and parses the interface parameters for *all* possible LIDAR sensors models of SICK, which is not necessary because only one sensor is used. Furthermore, by default, the library outputs an results of a completed scan to an image file, which is also removed to reduce the number of computations needed per scan. Hence, the library is used for connecting with the sensor via Ethernet TCP, sending commands to the sensor in the right datagram protocol, and receiving and converting the raw sensor data to distance and angle values. For this implementation, the IP address of the LIDAR sensor is set to be fixed for the initialization of the TCP communication. The IP address on the internal firmware of the LIDAR sensor is set to "169.254.153.219", and hence, this IP address is used in the LIDAR device activity for the communication with the LIDAR. Note that the IP address could probably also be obtained automatically, which should be more modular, but this is not implemented due to time restrictions.

7.1.3 EtherCAT master

As discussed in chapter 4, the computational computer has to fulfill the task of acting as an EtherCAT master. For this, the Simple Open EtherCAT Master (SOEM) C library is used [34]. SOEM is an open-source, cross-platform EtherCAT master driver and application layer. It provides the following functionalities: binding an Ethernet card to connect to a RAW socket, configuring EtherCAT slaves and coordinating/monitoring their states, and reading/ writing data using I/O mapping for synchronous communication. For the implementation of SOEM in the EtherCAT communication activity, the network card device name is fixed and set to the network card name "enp1s0". Note that this parameter value is hardware-dependent and has to be changed when the program runs on different hardware.

7.2 Combining C/C++

The software template that is given for this research, together with the SOEM library, are written in the programming language C. However, the LibNiFalcon and SICK Scan Base library are only available in C++, and hence, these programming languages have to be "mixed". C++ is a superset of C, and therefore, any valid C program is a valid C++ program, and it is chosen to use a C++ compiler to compile the final code. However, there are differences between the two programming languages that have to be taken into account.

C++ supports function overloading (there can be more than one function with the same name but different parameters) and, as a result, adds additional information to these function names, which is called Name Mangling [35]. C does not support function overloading and hence Name Mangling. To make sure that the functions in C can be used properly in the C++ program, the *extern "C"* linkage specification has to be used when C functions are defined. This specification tells the linker that after the compilation of the program, the functions declared with extern "C" have to be linked without Name Mangling.

The rules for compiling C++ code are stricter than that of C and especially with characters and strings. Therefore, all pointers of type *char* in the C code have to be changed to a pointer of type *const char* when using the C++ compiler. Furthermore, the given C template code uses the following function declaration for the creation (and deleting) a token type in the Petri Net models:

- int create_token_type(place_t *p, const char *typename, int n);

However, the argument *typename* is a reserved keyword in C++ and cannot be used for the

declaration of functions in this way in C++. Hence, all arguments in the C code with reserved keyword parameters (like *typename*) have to be changed when compiling the code with a C++ compiler.

7.3 Ropod platform velocity controller

As described in chapter 5, the ropod platform uses a combination of different controllers to control the motion of the platform. The overall controller consists of the wheel velocity controller with eight lower-level velocity controllers (one for each wheel), the platform controller with three lower-level velocity controllers (one for $\dot{x}$, $\dot{y}$ and $\dot{\theta}$), the torque distribution algorithm, and the kinematic equations (see figure 5.2).

For implementing the ropod platform controller, already existing C++ code from the ROPOD project is used. However, the existing C++ code consists of separate module tests for different parts of the overall controller. The existing code is not yet fully completed and is not yet ready to be implemented as a whole. The parts (blocks) that are already implemented and coupled in code in the form of module tests are: the wheel velocity controller, the torque distribution, and the forward and inverse kinematics equations. Hence, the platform velocity controller has yet to be created and integrated for this research.

In the existing code, the "Wheel_velocity_controller" is a C++ class that consists, among other things, of the class "Wheels_velocity_controllers" and an update function. The class "Wheels_velocity_controllers" is the set of eight linear velocity controllers (one for each wheel), and the update function is used to update all of its eight controllers with one function call. These linear velocity controllers are generic linear controllers with the following control actions: a gain action (P), an integral action (I), a lead-lag compensator, a low-pass filter, and a saturation for the integral action.

The class diagram that describes the used classes and their relationships can be seen in figure 7.1. For the implementation of the platform velocity controller, the class "Wheels_velocity_controllers" is renamed to the more generic and fitting name "Velocity_controllers". This class is reused with a set of three linear velocity controllers (instead of eight) and implemented in a new class called "Platform_velocity_controller". This new class controls the platform velocities $\dot{x}$, $\dot{y}$ and $\dot{\theta}$ using its three velocity controllers. The new class also has an update function to update all of its three velocity controllers with one function call. Furthermore, it calculates the velocity errors on platform-level, which are the input for the three lower-level velocity controllers. The outputs of these controllers are converted into platform wrenches and used as inputs for the torque distribution algorithm.

On top of that, the current code has another convenient class to merge all update functions from the controllers described above, and this class is called "Ropod_Torque_Dist". The update function of this class has to be called each time the overall controller needs to update (in this design, each cycle time of thread 3). The update function from the newly implemented "Platform_velocity_controller" is therefore also added to this convenient class. Hence, the overall controller can conveniently be updated using only one update function call (see figure 7.1). Note that in this research, the name for this convenient merging class is adopted from existing code and should be changed to a more fitting name in future work.

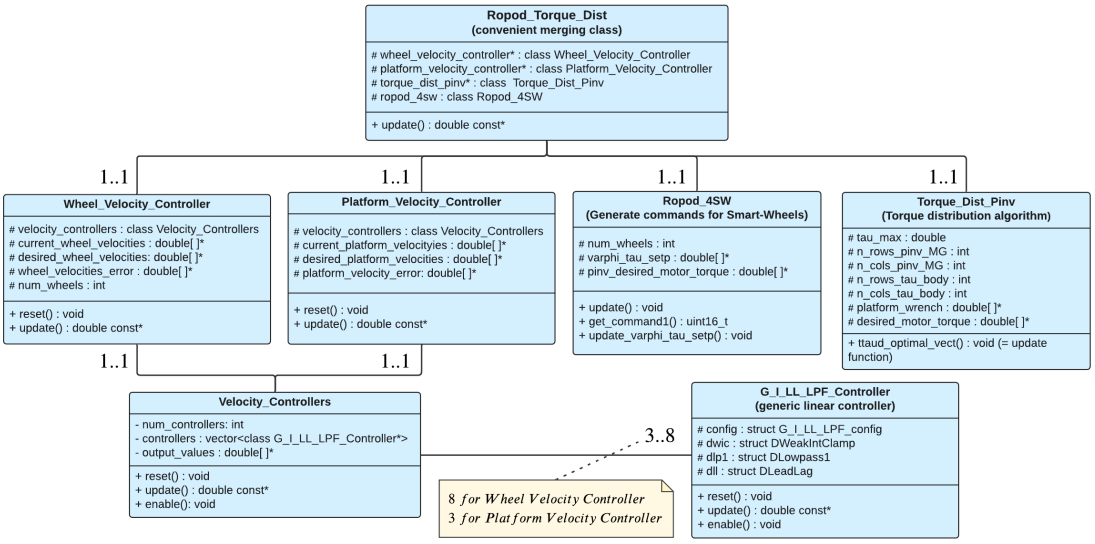


Figure 7.1: Class diagram of the overall ropod platform velocity controller

The generic control actions values of the eight velocity controllers for the wheels differ from the three velocity controllers for the platform. The control action values for both the wheels and platform controllers can be found in appendix D. Moreover, two broken smart-wheel units had to be replaced before the ropod platform could be used for this research. As a result, the pivot offsets of all the smart-wheel units had to be redetermined manually once. These values have to be known before the smart-wheel units can internally calibrate their pivot encoders. The pivot offset values are determined to be: 4.691878, 3.224157, 0.028178, and 5.856820 radians for smart-wheel units 1,2,3, and 4, respectively.

7.4 System and application performance

To determine the real-time performance of the Linux computer and the application, two experiments are conducted. The first experiment is the evaluation of the performance of the real-time Linux environment (kernel) on the computational computer used for this research. The second experiment is an approximation of the real-time teleoperation application performance. The essential parameters in these experiments are the latencies and, in particular, the worst-case latency. The latencies and worst-case latency give an idea of how deterministic the real-time system is. The fastest cycle time or the designed teleoperation system has to be 1000 Hz, which is 1000 us.

7.4.1 Real-time performance Linux kernel

To determine the performance of the real-time system, the latencies are measured under the worst-case system environments by generating different loads on the system. The system on full load will produce the most realistic and worst case latency values. To generate loads on the system, the benchmark hackbench is used to generate heavy CPU load. “Hackbench creates pairs of threads which send data from a sender to a receiver via a pipe or socket.” [36]. Furthermore, I/O load is generated using the Linux function disk usage (du), and network load is generated using the Linux function netperf.

For the measurements of the real-time system latencies under load, the tool Cyclictest [18] is used in the role of the real-time application. Cyclictest is also briefly described in section 2.3.5. Because there are many factors involved in generating latencies, it is important to run the tests as

long as possible because running it for a more extended time usually means that it has a greater probability of detecting the system's maximum latency. Another important consideration is the options chosen for the measurements with Cyclictest, such as the thread wake-up interval, the real-time thread priority, the test duration, etc. All options can be found in [18]. The test duration is chosen to be a hundred million iterations, which is around 13 hours with an interval of 500 us. It is chosen to run one measuring thread on each CPU (which is four) and the clock_nanosleep() option is enabled for shorter wake-up latency. The test has to be executed as Super User and the execute command and chosen options are as follows:

- sudo time cyclictest -t -q -n -l -a –mlockall –priority=80 –interval=500 -l100000000 -h1500

The -h option creates a latency histogram output of all the measured latencies in the time interval. The results of the cyclictest on the Preempt real-time kernel can be seen in figure 7.2 a. For comparison, the same test is also executed on the non-preemptive base kernel of Linux, and the results can be seen in figure 7.2 b. The worst-case latency for the real-time kernel is 53 us and for the base kernel 354 us. The test shows that the worst-case latency measured under load of the real-time kernel is a lot lower and more deterministic than the base kernel. However, the results have to be evaluated carefully, as the worst-case value measured does not necessarily represent the system's worst-case.

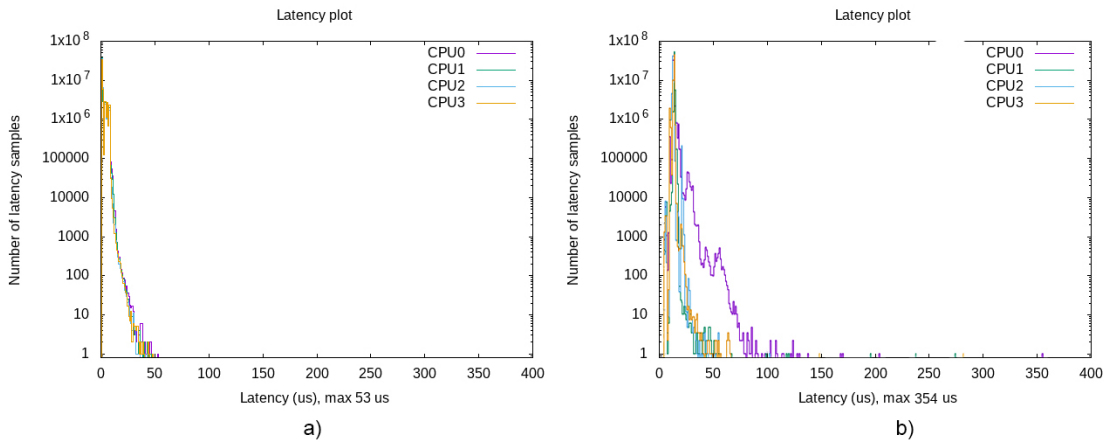


Figure 7.2: Latencies of the Preempt real-time kernel (a) compared to the base kernel (b)

7.4.2 Real-time performance real-time application

The computational load on the Linux system of the haptic teleoperation system is expected to be pretty much the same for each update cycle. All matrix equations are fixed-size and do not grow in size. Also, the LIDAR produces a constant stream with fixed-length data. Hence the computational time of the complete program is expected not to variate much over time. Cyclictest is not intended to measure latencies of applications that run on the real-time system. However, Cyclictest could be used to get an estimation of the real-time application's performance. To get this approximation, two different runs of the Cyclictest have to be performed. The first one is to establish a baseline, and the second is run to determine the changes in the baseline by the influence of the real-time application. The baseline measures the latency caused by the operating system and everything below the operating system. By calculating the difference between the results of the two Cyclictest runs, the approximate maximum latency of the real-time application can then be determined.

The first Cyclictest run determines the baseline and has to have a higher real-time priority than the real-time application whose performance is to be measured. The second Cyclictest run determines the value influenced by the real-time application and has to have a real-time priority that is

slightly lower than the evaluated real-time application. Therefore, the real-time priority of the first Cyclictest is set to 41 and of the second Cyclictest to 38. Again, it is important to run the tests as long as possible to get the best approximating real-time performance. For this test, the Preempt real-time kernel is used because this will be the used kernel for the final operation of the haptic teleoperation system. Concerning the system load for these tests, the system should be running all real-time and non-real-time processes that would usually be executed to bring the load as close as possible to the real-time application for which the system is intended. This system load will provide the most realistic latencies because the latencies on the system will then be influenced by everything that would influence them in the final setup of the teleoperation application. Five tests (so 10 Cyclictests) are conducted with a test duration of two million iterations (about half an hour), and settings similar to the Cyclictest execute command described above except the priority and number of iterations.

The difference between the two Cyclictest runs and hence the rough approximation of the worst-case latencies of the real-time application for the five tests can be seen in table 7.1. It can be concluded that the worst-case latency never exceeds 44 us and hence is more than enough for the requirements of the haptic teleoperation system execution.

Approximation of the application's worst-case latency					
Test number	1	2	3	4	5
Worst-case latency [us]	44	32	40	25	33

Table 7.1: Places and transitions

7.4.3 Novint Falcon I/O iterations

The cycle times of the Novint Falcon I/O loops (USB communication in the Falcon device activity) were briefly measured using the high-precision timer already available in the event-loop in thread 0. At first, the average I/O loop time of around 850 Hz was measured. After investigating and reading notes in the LibNiFalcon library comments, it was concluded that the latency timer on the FTDI chip had a buffer that reached its maximum capacity and, therefore, will send bytes to the host immediately. Without going into detail too much, the read request size was lowered, and after that, I/O loop times of 1000 Hz were measured, which is the intended I/O loop for the Novint Falcon communication. (source: www.FTDIchip.com, and in particular this document)

7.5 Shutdown command

The Linux terminal is used to execute the final program. In the Linux terminal, the process (main thread) can be interrupted ("killed") by sending a terminate signal using the Ctrl+c keyboard shortcut. When the process catches the signal, it will terminate itself and returns to the Linux terminal. However, there is no insight in *how* the process is stopped. For example, direct termination of the communication with hardware devices is undesired because it can cause unpredictable and sometimes even unsafe behavior of the devices. Therefore for the implementation of this research and for debugging purposes, a shutdown mechanism is implemented that customizes the catch of the terminate signal using Ctrl+c. The program will detect if a terminate signal is given, and the application mediator activity will signal all activities and threads to stop. The activities and threads will act on this signal by going through their LCSMs and when all LCSMs are in their done state, the process (main thread) can safely be terminated.

8. Conclusions and Recommendations for future work

This chapter states the conclusions of this research and the recommendations for future work. Note that more detailed conclusions can be found at the end of chapters 4, 5, and 6.

8.1 Conclusions

This work proposes a modular design and implementation for a haptic force feedback teleoperation system based on a given software template and its generic design approach. The main conclusion of this research is that a modular haptic teleoperation system is designed and implemented using components as building blocks. That is, all components and their communications are designed to be individual sub-systems that are ready to be composed into an application-specific system-of-systems, in which the components come together and realize an application together. The application designed for this research uses the Novint Falcon haptic interface to generate velocity commands for the ropod based on the car-driving metaphor to drive to the end of a dead-end hallway. The user perceives the ropod platform's environment using force feedback generated by LIDAR distances up to objects around the ropod and considered areas. Due to time reasons, the environmental force generation based in LIDAR distance points to objects is not implemented. Note that this is one choice of an application that could be done using this modular haptic teleoperation design and that other haptic applications could easily be implemented due to the modular and generic aspects of the design. For example, the generic components could be re-used to create a haptic application in which the ropod is used to haptically move objects, and the user can feel how heavy the objects are using force feedback.

The template is used as guidance for the design and implementation of this research and helped a lot in understanding the generic software architecture concepts thoroughly. The template already had the application mediator activity, and two example threads, four example activities, and two activity schedulers implemented. For this research, these example threads and activities were (re)used to create a total of four threads, seven activities, and four activity schedulers. The application mediator activity with its Flags, Petri Nets, and LCSM was adjusted to add the configuration and coordination of all added threads and activities. It should be mentioned that the template was a bit overkill for the design and implementation of this haptic teleoperation system, and as a result, parts of the template were not used in this research. For example, the threads' event loops and event loop functions remain untouched for this research. The disadvantage of the template from the author's perspective is that the template and its concepts have to be fully understood before it can be used in the way that it is intended. Hence, this also includes parts that are not particularly necessary to be modified for a design.

8.2 Recommendations for future work

This work can still be improved and the current design and implementation of the haptic teleoperation system triggers a lot of new opportunities for further research in the field of haptic force feedback. The first recommendation for future work is to actually close the haptic loop on two sides by implementing the designed environmental force generation from chapter 5. When the haptic loop is closed, experiments should be conducted on how well the haptic teleoperation system performs. The application environment using the dead-end hallway could be used to quantify the transparency and number of collisions of the force feedback (see figure 3.1). Participants could be asked to blindly drive the ropod platform in a straight line at a distance $dist_1$ from the upper and lower walls. Furthermore the ropod should come to a complete stop at a distance $dist_2$ at the end of the hallway. A virtual offset $dist_3$ could be used to determine collisions. Furthermore, the ropod platform full geometry could be used instead of a point for

the generation of the environmental force and the communication could be chosen to be wireless. However, wireless communication introduce delays and probably means trade-offs have to be made in the communication speed and stability/ transparency of the system.

Another recommendation for future work is to look at the use of shared control using the designed haptic teleoperation system. For example, autonomous obstacle avoidance could be implemented where the ropod add extra motions on top of the commands send by the user to avoid obstacles autonomously using the sensors available on the ropod platform.

Bibliography

[1] S. Lichiardopol, "A Survey on Teleoperation," *DCT 2007.155*, vol. 2007, no. 2007, p. 34, 2007.

[2] R. T. Laird, M. H. Bruch, M. B. West, D. A. Ciccimaro, and H. R. Everett, "Issues in Vehicle Teleoperation for Tunnel and Sewer Reconnaissance," *Quantum*, pp. 1–6, 2000. [Online]. Available: http://www.dtic.mil/docs/citations/ADA422071

[3] N. Diolaiti and C. Melchiorri, "Teleoperation of a mobile robot through haptic feedback," *Proceedings: HAVE 2002 - IEEE International Workshop on Haptic Virtual Environments and their Applications*, no. February, pp. 67–72, 2002.

[4] H. Bruyninckx, E. Scioni, H. Nico, J. Philips, F. Reniers, D. Monari, M. Frigerio, S. P. Diez, S. V. Driessche, N. Tsiogkas, and K. U. Leuven, *Building blocks for the Design of Complicated Systems featuring Situational Awareness. Composable components for compositional, adaptive, and explainable systems-of-systems.*, 2021.

[5] B. Siciliano and O. Khatib, *Springer handbook of robotics*, 2016.

[6] H. Roth, K. Schilling, and O. J. Rösch, *Haptic interfaces for remote control of mobile robots.* IFAC, 2002, vol. 15, no. 1. [Online]. Available: http://dx.doi.org/10.3182/20020721-6-ES-1901.00936

[7] S. Lee, G. J. Kim, and C.-m. Park, "Haptic Control of a Mobile Robot: A User Study," no. October, 2002.

[8] S. Lee and G. Kim, "Effects of haptic feedback on telepresence and navigational performance," *Proc. of Intl. . . .*, 2004. [Online]. Available: http://vrsj.ime.cmc.osaka-u.ac.jp/ic-at/papers/2004/S4-4.pdf

[9] I. Farkhatdinov, J. H. Ryu, and J. Poduraev, "A user study of command strategies for mobile robot teleoperation," *Intelligent Service Robotics*, vol. 2, no. 2, pp. 95–104, 2009.

[10] E. V. Poorten, E. Demeester, and C. Diepenbeek, "Estimating powered wheelchair driver intentions more accurately us- ing force feedback information Modeling behaviour of drivers us- ing haptic joysticks Probabilistic driver model," 2020.

[11] H. Bruyninckx, E. Scioni, H. Nico, J. Philips, F. Reniers, D. Monari, M. Frigerio, S. P. Diez, S. V. Driessche, N. Tsiogkas, and K. U. Leuven, "Building blocks for the Design of Complicated Systems featuring Situational Awareness. Composable components for compositional, adaptive, and explainable systems-of-systems." 2021, ch. 2.6.8, p. 53.

[12] J. Wang and W. Tepfenhart, *Formal Methods in Computer Science.* Boca Raton: Chapman and Hall/CRC, 2019.

[13] K. G. Shin and P. Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994.

[14] D. Hart and J. Kacur, "Real-Time Linux Wiki," 2015. [Online]. Available: https://rt.wiki.kernel.org/

[15] G. Cena, I. C. Bertolotti, S. Scanzio, A. Valenzano, and C. Zunino, "Evaluation of EtherCAT distributed clock performance," Ph.D. dissertation, 2012.

[16] K. Langlois, T. Van Der Hoeven, D. Rodriguez Cianca, T. Verstraten, T. Bacek, B. Convens, C. Rodriguez-Guerrero, V. Grosu, D. Lefeber, and B. Vanderborght, "EtherCAT tutorial: An introduction for real-time hardware communication on windows [tutorial]," *IEEE Robotics and Automation Magazine*, vol. 25, no. 1, 2018.

[17] Q. V. Dang, B. Allouche, A. Dequidt, L. Vermeiren, and V. Dubreucq, "Real-time control of a force feedback haptic interface via EtherCAT fieldbus," *Proceedings of the IEEE International Conference on Industrial Technology*, vol. 2015-June, no. June, pp. 441–446, 2015.

[18] Linux_Foundation, "Cyclictest manual," 2018. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

[19] Lung-Wen Tsai, "Multi-degree-of-freedom mechanisms for machine tools and the like." p. 11, 1997.

[20] F. Schill, R. Mahony, P. Corke, and L. Cole, "Virtual force feedback teleoperation of the InsectBot using optical flow," *Proceedings of the 2008 Australasian Conference on Robotics and Automation, ACRA 2008*, no. January 2008, 2008.

[21] S. Martin and N. Hillier, "Characterisation of the Novint Falcon haptic device for application as a robot manipulator," *Proceedings of the 2009 Australasian Conference on Robotics and Automation, ACRA 2009*, 2009.

[22] K. Ouellet, "Projet de n d'etudes en genie de la production automatisee." University of Quebec, Quebec, Tech. Rep., 2008.

[23] R. E. Stamper, "A Three Degree of Freedom Parallel Manipulator with Only Translational Degrees of Freedom," *Engineering*, vol. 1, p. 211, 1997.

[24] N. Karbasizadeh, M. Zarei, A. Aflakian, M. Tale Masouleh, and A. Kalhor, "Experimental dynamic identification and model feed-forward control of Novint Falcon haptic device," *Mechatronics*, vol. 51, no. May, pp. 19–30, 2018.

[25] R. Project, "Ultra-flat, ultra-flexible, cost-effective robotic pods for handling legacy in logistics," 2020. [Online]. Available: https://cordis.europa.eu/project/id/731848

[26] Cesar A. Lopez Martinez, "TRL5 prototype of low-level motion controller for joint level and Cartesian level control of smart wheel," pp. 1–31, 2018.

[27] J. Unkel, "Force sensorless compliant control for mobile robots," Ph.D. dissertation, TU/e, 2012.

[28] A. V. Shah, S. Teuscher, E. W. McClain, and J. J. Abbott, "How to build an inexpensive 5-DOF haptic device using two novint falcons," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6191 LNCS, no. PART 1, pp. 136–143, 2010.

[29] J.-L. Rodríguez and R. Velàzquez, "Haptic Rendering of Virtual Shapes with the Novint Falcon," *Procedia Technology*, vol. 3, pp. 132–138, 2012.

[30] I. Farkhatdinov and J. H. Ryu, "Switching of control signals in teleoperation systems: Formalization and application," *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, pp. 353–358, 2008.

[31] K. Ouellet and A. Barrow, "The libnifalcon Project," 2016. [Online]. Available: https://github.com/libnifalcon/libnifalcon

[32] P. Kadleček, "Overview of current developments in haptic APIs Abstraction layers of Haptic APIs," *Central European Seminar on Computer Graphics (CESCG)*, p. 8, 2011.

[33] SICK, "Sick Scan Base project," 2018. [Online]. Available: https://github.com/SICKAG/sick_scan_base

[34] OpenEtherCATsociety, "SOEM Wiki," 2021. [Online]. Available: https://github.com/OpenEtherCATsociety/SOEM

[35] D. Herity, "C++ in embedded systems: Myth and reality," *Embedded Systems Programming*, vol. 11, no. 2, 2006.

[36] "Hackbench manual," 2018. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/hackbench

[37] "Preemption models," 2020. [Online]. Available: https://github.com/torvalds/linux/blob/master/kernel/Kconfig.preempt

Appendix A

Preemption Models

According to the official Linux kernel documentation [37], the following preemption models can be selected before building the Linux kernel:

- **No Forced Preemption (server):** This is the traditional Linux preemption model, geared towards throughput. It will still provide good latencies most of the time, but there are no guarantees and occasional longer delays are possible.
- **Voluntary Kernel Preemption (Desktop):** This option reduces the latency of the kernel by adding more "explicit preemption points" to the kernel code. These new preemption points have been selected to reduce the maximum latency of rescheduling, providing faster application reactions, at the cost of slightly lower throughput.
- **Preemptible Kernel (Low-Latency Desktop):** This option reduces the latency of the kernel by making all kernel code (that is not executing in a critical section) preemptible. This allows reaction to interactive events by permitting a low priority process to be preempted involuntarily even if it is in kernel mode executing a system call and would otherwise not be about to reach a natural preemption point.
- **Fully Preemptible Kernel (RT):** This option turns the kernel into a real-time kernel by replacing various locking primitives (spinlocks, rwlocks, etc.) with preemptible priority-inheritance aware variants, enforcing interrupt threading and introducing mechanisms to break up long non-preemptible sections. This makes the kernel, except for very low level and critical code paths (entry code, scheduler, low level interrupt handling) fully preemptible and brings most execution contexts under scheduler control.

Scheduling policies and priority ranges

All Linux threads have one of the following scheduling policies:

- **SCHED_OTHER** or **SCHED_NORMAL:** The default policy.
  Min/max priority = 0/0.
- **SCHED_BATCH:** Similar to SCHED_OTHER, but with a throughput orientation.
  Min/max priority = 0/0.
- **SCHED_IDLE:** A lower priority than SCHED_OTHER.
  Min/max priority = 0/0.
- **SCHED_FIFO:** A first in/first out real time policy (real-time).
  Min/max priority = 1/99.
- **SCHED_RR:** A round-robin real time policy (real-time).
  Min/max priority = 1/99.

Appendix B

The schematic figure of one leg, the end-effector base, and end-effector of the Novint Falcon and the geometric dimensions can be seen in figure 1. The values of these dimensions can be found in table 1. Table 2 shows the inertial properties of the Falcon and figure 2 shows the achievable workspace positions.

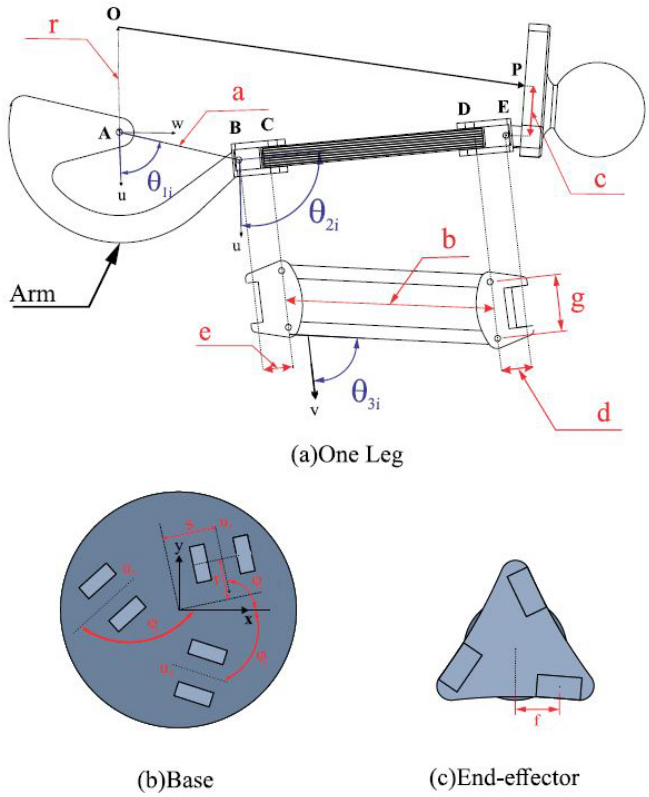


Figure 1: Geometric dimensions of the Novint Falcon (source: [21])

Dimension	Value (m x 10^{-3})
a	60.00
b	102.50
c	14.43
d	11.25
e	11.25
f	25.00
g	27.90
r	36.60
s	23.09

Table 1: Dimension values of the Novint Falcon according to figure 1

	leg	shin bar	shin joint	end effector
mass $\times 10^{-3}$	89.59	8.37	10.42	82.73
C.M.†				
$x \times 10^{-3}$	-3.19	56.35	15.61	15.7
$y \times 10^{-3}$	23.34	0	0	26.2
$z \times 10^{-3}$	0	0	0	48.78
I_{xx}	-	-	-	
I_{yy} $\times 10^{-6}$	-	8.5	-	-
I_{zz} $\times 10^{-6}$	140	8.5	0.7	-

Table 2: Inertial properties for the Novint Falcon links according to [21]. The center-of-mass is defined as [x,y,z] with the x-direction along the line outwards from the base.

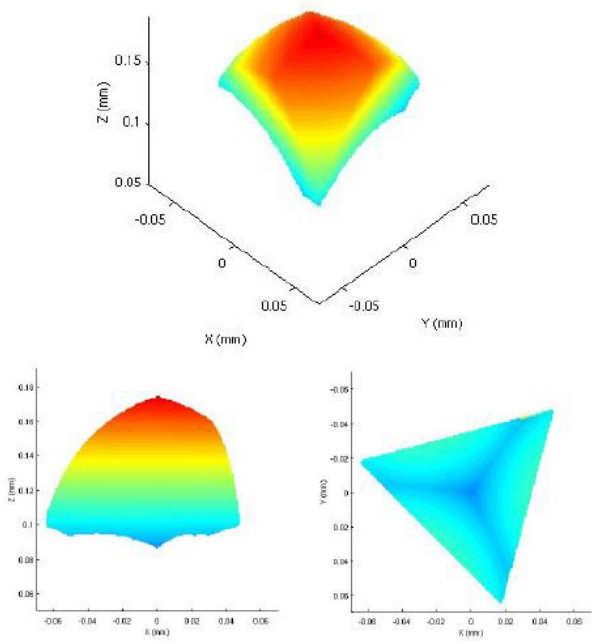


Figure 2: The plot of the Novint Falcon's achievable workspace positions according to [21]. An isometric view (top), side view (bottom-left), and behind view (bottom-right), units are meters

Appendix C

Variable type	Variable name	Description
uint16_t	status1	Status bits as defined in STAT1_
uint16_t	status2	Status bits as defined in STAT2_
uint64_t	sensor_ts	EtherCAT timestamp (ns) on sensor acquisition
uint64_t	setpoint_ts	EtherCAT timestamp (ns) of last setpoint data
float	encoder_1	encoder 1 value in rad (no wrapping at 2PI)
float	velocity_1	encoder 1 velocity in rad/s
float	current_1_d	motor 1 current direct in amp
float	current_1_q	motor 1 current quadrature in amp
float	current_1_u	motor 1 current phase U in amp
float	current_1_v	motor 1 current phase V in amp
float	current_1_w	motor 1 current phase W in amp
float	voltage_1	motor 1 voltage from pwm in volts
float	voltage_1_u	motor 1 voltage from phase U in volts
float	voltage_1_v	motor 1 voltage from phase V in volts
float	voltage_1_w	motor 1 voltage from phase W in volts
float	temperature_1	motor 1 estimated temperature in K
float	encoder_2	encoder 2 value in rad (no wrapping at 2PI)
float	velocity_2	encoder 2 velocity in rad/s
float	current_2_d	motor 2 current direct in amp
float	current_2_q	motor 2 current quadrature in amp
float	current_2_u	motor 2 current phase U in amp
float	current_2_v	motor 2 current phase V in amp
float	current_2_w	motor 2 current phase W in amp
float	voltage_2	motor 2 voltage from pwm in volts
float	voltage_2_u	motor 2 voltage from phase U in volts
float	voltage_2_v	motor 2 voltage from phase V in volts
float	voltage_2_w	motor 2 voltage from phase W in volts
float	temperature_2	motor 2 estimated temperature in K
float	encoder_pivot	encoder pivot value in rad (wrapping at -PI and +PI)
float	velocity_pivot	encoder pivot velocity in rad/s
float	voltage_bus	bus voltage in volts
uint64_t	imu_ts	EtherCAT timestamp (ns) of IMU sensor acquisition
float	accel_x	IMU accelerometer X-axis in m/s2
float	accel_y	IMU accelerometer Y-axis in m/s2
float	accel_z	IMU accelerometer Z-axis in m/s2
float	gyro_x	IMU gyro X-axis in rad/s
float	gyro_y	IMU gyro Y-axis in rad/s
float	gyro_z	IMU gyro Z-axis in rad/s
float	temperature_imu	IMU temperature in K
float	pressure	barometric pressure in Pa absolute

Table 3: The struct values for the Smart-wheel units sensory output data

Variable type	**Variable name**	**Description**
uint16_t	command1	Command bits as defined in COM1_
uint16_t	command2	Command bits as defined in COM2_
float	setpoint1	Setpoint 1
float	setpoint2	Setpoint 2
float	limit1_p	Upper limit 1
float	limit1_n	Lower limit 1
float	limit2_p	Upper limit 2
float	limit2_n	Lower limit 2
uint64_t	timestamp	EtherCAT timestamp (ns) setpoint execution

Table 4: The struct values for the Smart-wheel units command and setpoint inputs

Appendix D

Ropod platform velocity controller and wheel controller values.

Linear controller values for wheels velocity $\dot{\varphi}$ can be seen in table 5.

Control action	Value name	value
Gain	p_gain	0.05
Integrator	i_fhz	0
Lead-lag	ll_zero_fhz	2
Lead-lag	ll_pole_fhz	10
Low pass	lpf_fhz	50
I saturation	i_saturation	7
Step time	t_step	0.001

Table 5: Values for control actions wheels

Linear controller values for platform velocity $\dot{x}$, $\dot{y}$ can be seen in table 6. Linear

Control action	Value name	value
Gain	p_gain	100
Integrator	i_fhz	0.25
Lead-lag	ll_zero_fhz	20
Lead-lag	ll_pole_fhz	20
Low pass	lpf_fhz	100
I saturation	i_saturation	100
Step time	t_step	0.001

Table 6: Values for control actions wheels platform

controller values for platform velocity $\dot{\theta}$ can be seen in table 7.

Control action	Value name	value
Gain	p_gain	755
Integrator	i_fhz	0.3
Lead-lag	ll_zero_fhz	20
Lead-lag	ll_pole_fhz	20
Low pass	lpf_fhz	100
I saturation	i_saturation	100
Step time	t_step	0.001

Table 7: Values for control actions wheels platform

Appendix E

The list of commands that have to be sent for the initialization process can be found in table 8 and the thread & Activity scheduler coordination flag maps can be seen in figure 3

Sending order	Command	Description
1	CMD_SET_ACCESS_MODE_3	Maintenance access mode
2	CMD_SET_TO_COLA_A_PROTOCOL	Set protocol to ASCII
3	CMD_DEVICE_STATE	Check device state
4	CMD_SET_OUTPUT_RANGES	Set the output range
5	CMD_START_SCANDATA	Set scan data format
6	CMD_START_MEASUREMENT	Start continuous measurement
7	CMD_RUN	Log out and save changes

Table 8: LIDAR commands

Thread coordination flag map 1

Flag name	*Handled by* :
Start_thread_n	*Application mediator PetriNet*
Stop_thread_n	
Thread_n_disabled	
Thread_n_error_status	
Thread_n_running	
Thread_n_configures	*LCSM of thread n*
Thread_n_configuration_successful	
Thread_n_configuration_unsuccessful	
Thread_n_disabling	
Thread_n_cleaning_completed	

Activity scheduler flag map 2

Flag name	*Handled by* :
*Select_schedule_*1	*Application mediator PetriNet*
*Thread_n_schedule_*1	*Event loop of activity scheduler*
*Thread_n_schedule_*2	

Figure 3: Flag map 1 and 2 for the coordination between Petri Net models