A Framework For Motion Analysis Of Anthropomimetic Robots

MASTER THESIS IN INFORMATICS

submitted by
Simon Trendel

DEPARTMENT OF INFORMATICS

ROBOTICS AND EMBEDDED SYSTEMS
Technische Universität München

Prof. Dr. Alois Knoll

Abstract

Musculoskeletal robots are a unique variant of soft-robots that imitate biological musculoskeletal systems by replicating the overall structure of muscles with tendons attached to bones with passive joints. While such robots exhibit interesting dynamic properties, they are hard to control. The non-linearities resulting from their actuation call for new adaptive control strategies as purely model-based control hardly copes with the complexities of effects such as muscle-wrapping, hysteresis from friction and the co-actuation of several muscles per joint - at times even bi-articular, influencing several bones per muscle. Amalgamating the results of the original robot Roboy as well as the EU FP7 project MYOROBOTICS, a humanoid robot called Roboy 2.0 was developed at TUM. This thesis describes the development and evaluation of the low level control infrastructure, necessary to achieve high-frequency, high-fidelity control of the robot. Towards this goal, an external tracking system based on the consumer virtual reality system HTC Vive was reverse engineered. It can be used to track arbitrary objects in a room-scale setup. The external tracking provides a cheap, yet accurate alternative to expensive commercial tracking systems like camera-based infrared retro-reflective marker tracking systems. Conclusively, the combination of the new control infrastructure coupled with this external visual tracking allowed for external verification and calibration of the robots' proprioceptive control system.

I would like to thank my supervisor Rafael Hostettler, for the opportunity to work on such an interesting and multifaceted project and for his support and advice throughout.

Additionally I would like to thank Jörg Zelenka from B&R Automation for his continuous support on the openPowerLink implementation.

I would also like to thank my parents. Without their endless loving support this thesis would not exist.

Contents

Chapter 1

Introduction

1.1 Musculoskeletal Robots

Building robots that exhibit comparable dexterity and flexibility to animals and humans are a longstanding goal of robotics. Already, our economy hinges centrally on the automation of manual tasks that made mass-production of goods - from cars to consumer electronics - possible. However, to date we still heavily rely on human labor whenever the task is not repetitive to an extent where the robot's end-effector trajectory can be programmed statically.[1] Similarly, robots that cope with our everyday environment are still out of scope, albeit with recent promising results in robots with legged locomotion.[2, 3] In contrast, every animal is able to survive in its native environment, even when the control system is simplistic such as in cockroaches.[4]

To shed light on this gap in capabilities and extract the essential control paradigms that allow for such versatile interaction with the environment, a useful approach is to mimic the structure of the capable system and build a model that implements key similarities.[5] To this end, musculoskeletal robots comprise a skeleton, passive joints and actuators that exhibit a behavior similar to biological muscles.[6]

The robot project Roboy Junior[7] is such a musculoskeletal robot and was built as a successor to the ECCE robots[8], developed in the EU-FP7 Project ECCEROBOT, between 2012 and 2013 at the Artificial Intelligence Laboratory of Prof. Dr. Rolf Pfeifer in Zürich, Switzerland. MYOROBOTICS - an EU-FP7 project succeeding the ECCEROBOT projects as well - is based on the same results, however with the goal to provide researchers with a modular toolkit for musculoskeletal robotics[9] rather than creating an integrated humanoid robot.

1.2 Low-level Motor Control System

Roboy 2.0 unites the learnings from both of these projects and advances the technology. One of the key limiting factors of Roboy Junior was that its CAN based bus system was severely limited to below 10 Hz overall control frequency due to the 48 motors that had to be controlled concurrently[7]. MYOROBOTICS addressed this by replacing the daisy-chained motor driver boards with an intermediary control board, termed MyoGanglion, that directly controlled four motor boards through an SPI bus and was connected to other MyoGanglia as well as the controlling PC by the industrial field bus FlexRay. This allowed for a 500Hz overall control frequency of up to 24 Motors. [9]

However, as the FlexRay network was statically configured and MYOROBOTICS only implemented and exposed a subset of the FlexRay standard to users, the network was difficult to extend. Furthermore, Roboy 2.0 has more than 24 motors - which is not addressable by the previous system.

As FlexRay has lost importance in recent years due to the advent of Ethernet based buses such as EtherCAT[10] or Powerlink[10] that provide higher data rates and flexibility, replacing the infrastructure with a more flexible low-level control system was one of the central goals for the project.

The technical requirements for the low-level control of Roboy 2.0, deduced from the experience with MYOROBOTICS and Roboy Junior were:

1. Deterministic control - ensuring repeatability of experiment
2. At least 1kHz control frequency over all motors - allowing for fast, centralized control from the controlling PC and therefore for easy controller development.
3. At least 50 motors addressable - allowing to control all motors of Roboy concurrently
4. Simple extensibility and integration of additional sensors - addressing the research nature of the system
5. Co-existence with Robot Operating System (ROS) User Datagram Protocol (UDP)/Transmission Control Protocol (TCP) packages - allowing the non-real-time communication for higher-level control to use the same cabling as the real-time bus
6. Open-source, royalty free protocol - keeping the development of Roboy unrestricted

The first and fifth requirement together enforce Ethernet based bus systems, leaving EtherCAT as well as PowerLink as possible candidates that fulfill all the requirements 1 - 5. However, there are no free EtherCAT implementations, leaving PowerLink as the sole candidate short of implementing a custom bus system. The

development of a PowerLink based low-level motor control system is outlined in Section 2.2 and 3.3.

1.3 External Tracking System

Furthermore, another key aspect to evaluate the performance of a musculoskeletal robotics toolkit is to verify its performance in terms of repeatability, control precision and accuracy. Additionally, having an external reference of a robots state in space is highly useful for control development.

Generally, external tracking can be divided into two variants, model-based and marker-based tracking. In model-based tracking, a 3D model of an object of interest is required. Algorithms such as Efficient Perspective-n-Point Camera Pose Estimation (EPnP) [11] or Dense Articulated Real-Time Tracking (DART) [12] can be used to estimate the 6-degrees of freedom (DOF) pose of a model from 2D monocular images or depth maps, respectively.

In marker-based tracking, an object of interest is physically augmented with some form of markers, that are distinguishable by the tracking system. The markers can be colored balls in combination with monocular cameras. Commercially available systems typically use retro-reflective markers in combination with infrared spotlights and infrared sensitive cameras. These systems enable very accurate tracking down to several millimeters accuracy [13]. Depending on the tracking area size, many infrared cameras with spotlights are required, which makes theses systems very expensive (ten thousands of Euros). Another disadvantage is the required calibration of these systems. Whenever a camera is moved, even slightly, the systems needs to be recalibrated. The price and the static nature of those systems restrict their usage to research labs with a dedicated tracking space.

In robotics and especially for Roboy, it is necessary to change the tracking environment frequently, for example when testing new behaviors and interactions with changing environments. Therefore, a static system like the camera based marker systems was not very appealing. Recently, HTC in a cooperation with Valve has released a commercially available virtual reality (VR) equipment called Vive [14]. The Vive provides a VR experience, where in contrast to so far available systems, the user can freely move around in a designated room-scale area. The tracking is very accurate, which the Vive achieves by using specialized hardware consisting of infrared emitters and infrared sensitive sensors. The Vive tracking system needed to be highly mobile, since it is a commercially available VR equipment for the consumer market, the customer could not be bothered with time intensive calibration routines whenever the system is set up. The Vive system circumvents the calibration elegantly, which was the reason we chose to reverse engineer this system for application in Roboy and robotics in general.

Chapter 2

Theoretical Background

2.1 Myorobotics and Roboy 2.0

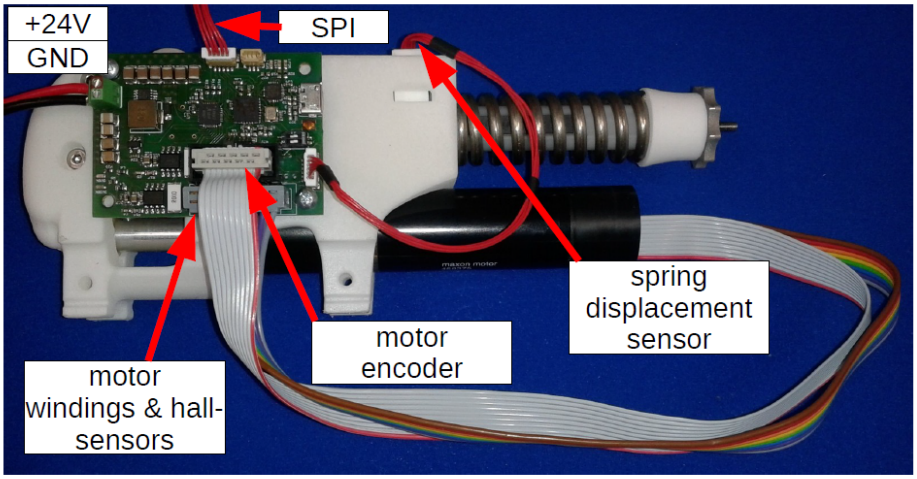
In summer semester 2017 the second version of Roboy was developed by the Roboy student team at TUM. Significant improvements compared to Roboy Junior have been implemented. The upgrades cover the mechanical design, electronic hardware, and control which will be outlined in the following sections.

2.1.1 Hardware

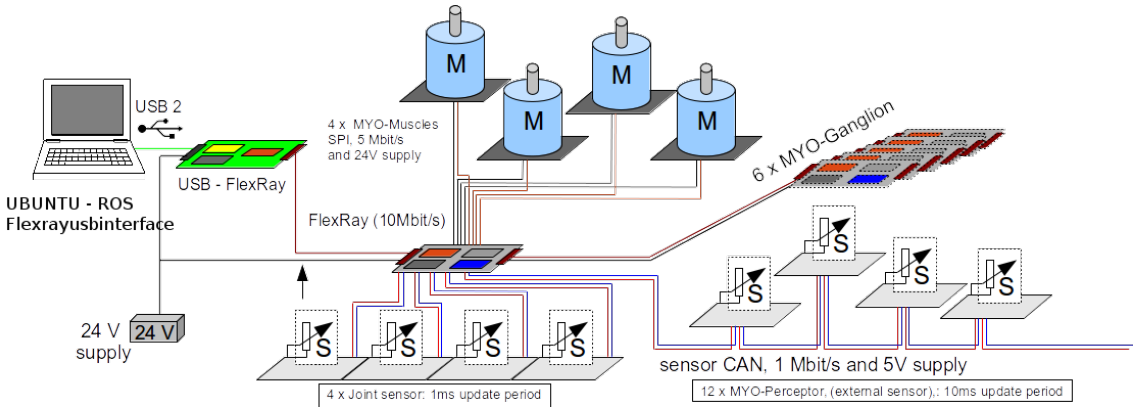
Roboy 2.0 uses hardware and electronics from the MYOROBOTICS toolkit (MyoToolKit)[15]. The MyoToolKit was designed as a modular toolkit for tendon based robots. The actuators are called muscle units. The muscle unit depicted in Figure 2.1a shows its main components. A Brush-less Direct Current Motor (BLDC) is connected to a motor board which controls the motor. The motor rolls up a tendon, which is routed via a pulley connected to the spring shaft. When load is applied on the tendon, the spring is compressed. On one hand, this protects the motor gear box from destruction by jerks, on the other hand this gives the muscle unit an inherent elasticity and recuperation characteristics, similar to a mammal muscle. The spring displacement is measured with a sensor and gives a direct feedback on the force applied on the tendon.

Apart from the muscle unit, the MyoToolKit consists of the following components, ordered the way they are connected from the host PC to the robot, as depicted in Figure 2.1b.

- USB-FlexRay Bridge: A custom PCB that can be connected to the host PC via USB. It features a FlexRay bus (common in the automobile industry) providing hard real-time communication with the MyoGanglia at up to 10Mbit/s.



(a) muscle unit with motor board



(b) system overview

Figure 2.1: MYOROBOTICS muscle unit and system overview (images from [9])

Additionally, it has a CAN bus. The board is depicted in Figure 2.2a.

- MyoGanglion: A custom PCB featuring a TMS570LS20216 from Texas Instruments running at 140Mhz. This floating point processor implements communication and control for up to four muscle units connected to it. It handles the FlexRay communication and runs four PID controller. The communication with the motor board is via SPI. The MyoGanglion is shown in Figure 2.2b on the bottom.
- Motor board: Another custom PCB that drives the BLDC motor of the muscle units. The motor board is depicted in Figure 2.2b on the top. The motor board reads the following pieces of information about the muscle unit:
  - motor position: An optical encoder attached to the back of the BLDC motor measures the motor position in encoder ticks.

– motor velocity: The motor positions between two consecutive queries from the SPI master are used to calculate the motor velocity in encoder ticks per second.

– motor current: The current consumption is measured via shunt resistors. The minimal measurable current is 16mA.

– spring displacement: the displacement of the muscle unit spring is measured via a hall sensor with a magnetic encoder attached to the spring shaft.

2.1.2 Communication and Control

Communication: The communication between the motor board and the MyoGanglion is the fastest in the overall system and runs at 2.5kHz. The communication with the FlexRay bridge runs at 1kHz. The bottleneck in the system communication is the USB connection to the FlexRay bridge which has a nondeterministic nominal speed of 500Hz. This sets the maximal achievable control frequency of a robot controlled with this system to 500Hz.

Control: The MyoGanglion implements four PID controller, one four each muscle unit connected to it. The PID block diagram is shown in Fig. 2.3. The setpoint *sp* coming from the host control PC is compared to a measurement *pv*. The resulting error is fed through a standard PID scheme. An optional feedforward FF gain can be added. Optionally a deadband and output saturation can be applied before the resulting pulse width modulation (PWM) command is sent to the motor board.

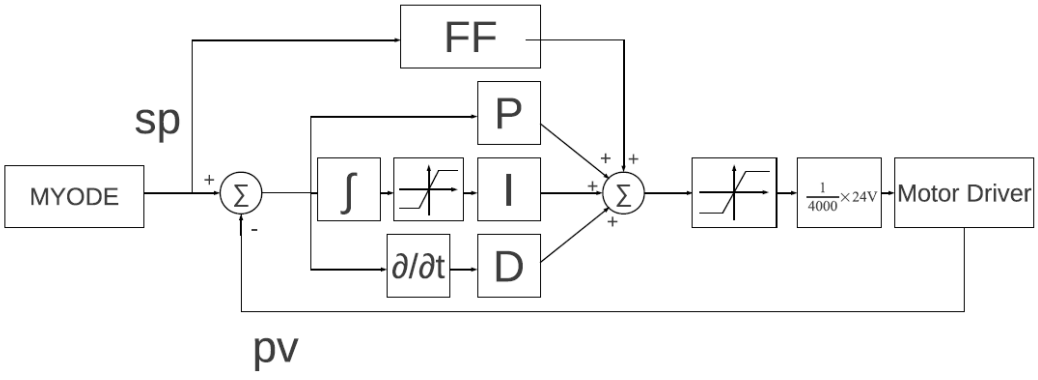
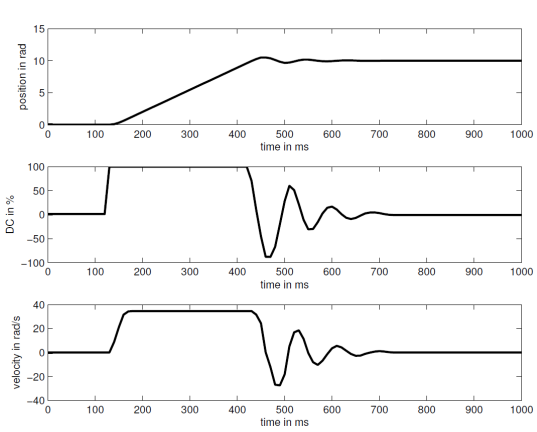


Figure 2.3: MyoGanglion PID controller block diagram (image from [9])

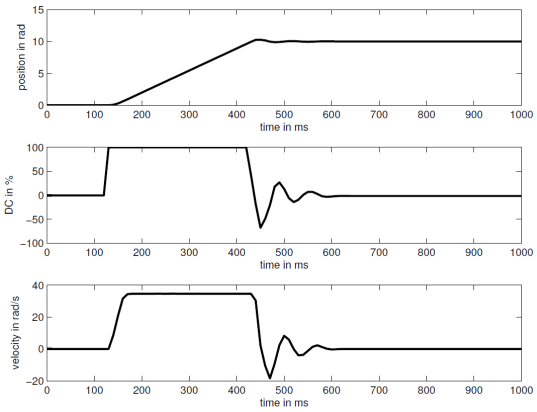
The MyoGanglion supports four control modes. Depending on the control mode, the measurement *pv* fed to the PID controller is changed accordingly:

- **raw**: open-loop raw PWM control
- **position**: the motor position is controlled
- **velocity**: the motor velocity is controlled
- **force**: the displacement of the spring is controlled. The force can only be controlled if the spring parameters are known or have been measured. The parameters are communicated to the MyoGanglion via FlexRay using a high-level config file, where these parameters are stored.

The control capabilities were investigated using a control frequency between motor board and MyoGanglion of 100Hz, driving the motor from 0 to 10 rad in position control mode. The only load acting on the motor, in this case, is the gearbox. In Fig. 2.4a the chosen P gain causes secondary oscillations around the target setpoint. An additional D gain was added which reduces the secondary oscillation as can be observed in Fig. 2.4b.



(a) P gain causing overshoot



(b) Additional D gain reduces the secondary oscillations

Figure 2.4: MyoGanglion motor position control response at 100Hz update frequency, starting from 0 rad with target setpoint 10 rad (images from [9])

2.2 OpenPOWERLINK

OpenPOWERLINK is a communication profile that extends the IEEE 802.3 Ethernet standard [16] to enable hard real-time communication. Hard real-time communication means that data is transferred with "predictable timing and precise synchronization" [17]. OpenPOWERLINK is used primarily in the automation industry for control of sometimes hundreds of distributed motors with very high precision and speed requirements.

2.2.1 Communication

An openPOWERLINK network can consist of up to 240 nodes. An openPOWERLINK network always consists of one Managing Node (MN) and typically several Controlled Node (CN)s. Figure 2.5 shows two separate openPOWERLINK networks. The nodes inside the networks are internally connected via a hub. The two openPOWERLINK networks are connected via routers to a high-level network. Communication with the openPOWERLINK networks is established via legacy Ethernet, such as TCP and UDP.

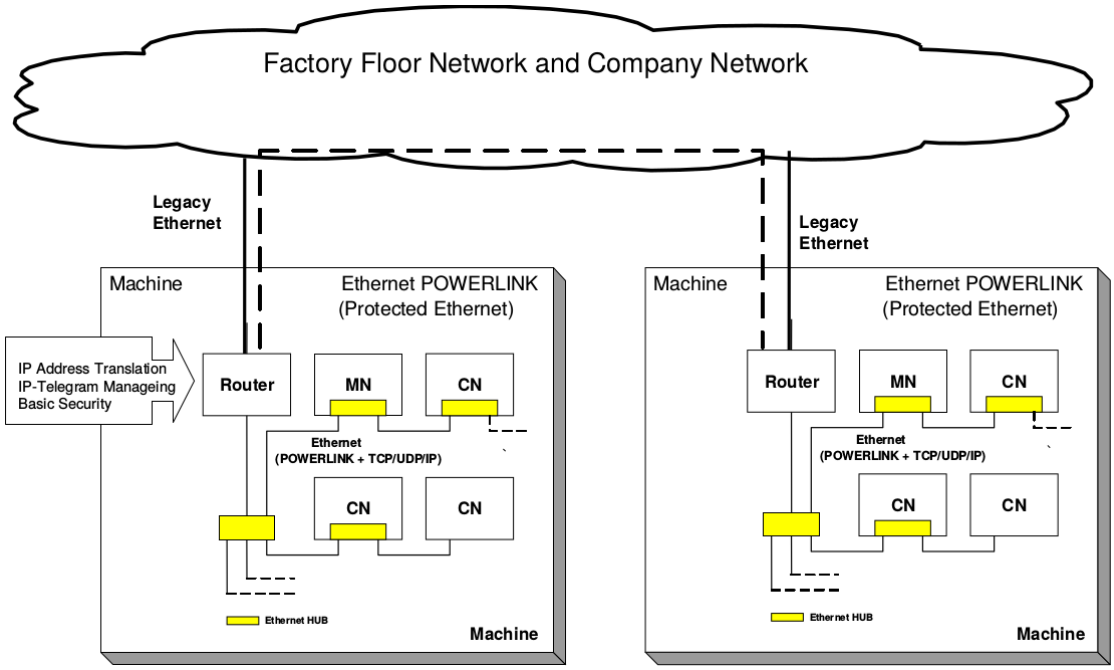


Figure 2.5: Typical openPOWERLINK network topology (image from [17])

The openPOWERLINK stack can be described in the ISO OSI reference model structure as depicted in Figure 2.6. The physical layer (PHY) uses standard Reduced Media-Independent Interface (RMII) or Media-Independent Interface (MII) Ethernet transceivers and RJ-45 magnetic coupling hubs connected with Category 7 Ethernet cables to the network. The openPOWERLINK stack implements modules directly on the data link layer, which parse the incoming data packets. Automated direct response packets enable ultra-low jitter down to $< 1\mu s$ at this stage. The process data is typically exchanged with an application running on the application layer via some form of shared memory. Because openPOWERLINK extends the IEEE 802.3 Ethernet standard, UDP and TCP are supported and also used for initialization of the nodes by the MN.

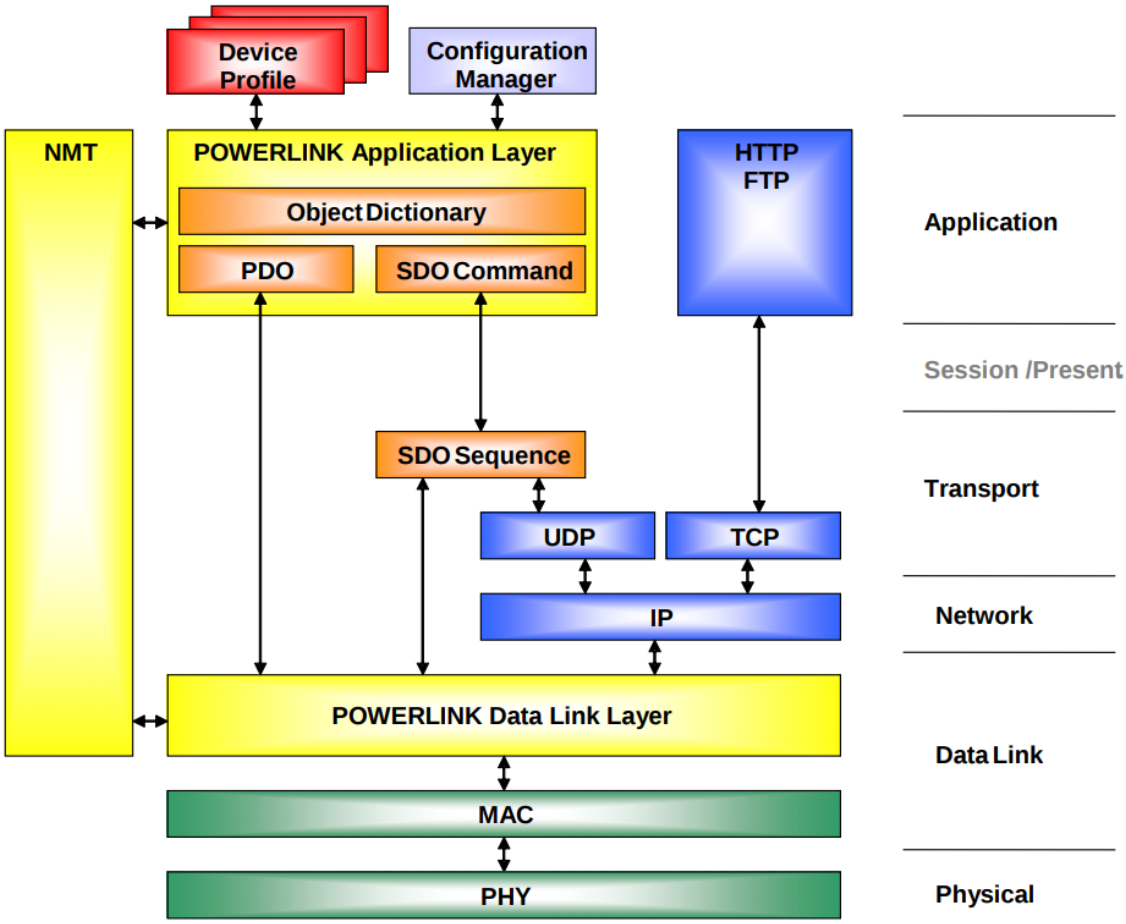


Figure 2.6: OpenPowerLink ISO-OSI Reference Model (image from [17])

The communication inside an openPOWERLINK network is orchestrated by the single MN. The MN keeps track of all connected CNs in the network. In an openPOWERLINK network, the CNs can be hot plugged, meaning that any CN can de disconnected or connected to the network at run time. The MN detects if a CN is connected to the network and initializes the CN according to a device profile. The device profile defines the data that is exchanged between all nodes. This is defined in the so-called Object Dictionary, which "is essentially a grouping of objects accessible via the network in an ordered, pre-defined fashion. Each object within the dictionary is addressed using a 16-bit index" [17]. The Object Dictionary defines data types and required objects for the openPOWERLINK communication, but also reserves index ranges for specifying the data that should be exchanged for each participating nodes in an openPOWERLINK network. The device profile can be edited with a plug-in for the eclipse Integrated development environment (IDE) called openCONFIGURATOR.

A typical communication cycle in an openPOWERLINK network is illustrated in

Figure 2.7. The MN starts the isochronous phase of the cycle with a start of cycle (SoC) frame. In the isochronous phase, the individual CNs are then sequentially polled with a poll request (PReq) frame and each CN has to respond within a specified time with its poll response (PRes) frame. After all CNs are queried, the MN starts the asynchronous phase with a start of asynchronous phase (SoA) frame. In the asynchronous phase, legacy Ethernet packets can be transmitted, such as UDP and TCP.

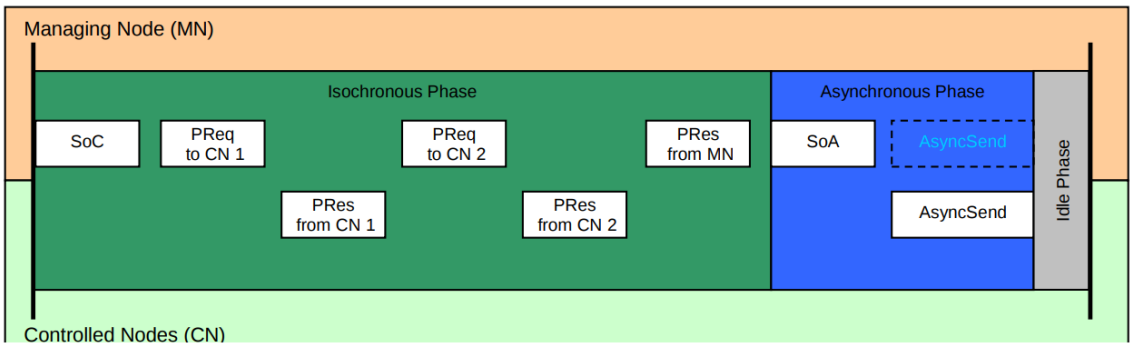


Figure 2.7: OpenPowerLink communication cycle (image from [17])

2.2.2 Hardware

OpenPOWERLINK builds on top of the standard Ethernet and therefore does not require specialized hardware. However, high-performance applications require specialized hardware. Commonly, field programmable gate array (FPGA) are used. These are connected directly to the Ethernet transceiver PHY. The cycle length can then be as low as $400\mu s$ with ultra-low jitter $< 1\mu s$. Additionally, a hub should be used to connect the individual nodes of the openPOWERLINK network. The reason for this is the reduced frame jitter and path delay value compared to switches or routers.

2.3 Lighthouse Tracking

2.3.1 Hardware

The HTC Vive is a VR equipment available commercially. It consists of a head mounted display (HMD), two controllers (one for each hand) and two base stations. Its main application is for gaming, where a user, wearing the HMD, can be tracked in space within the visible area covered by the two base stations. This tracking information can be used to render a virtual reality in the HMD. Because the users'

movements can be accurately tracked, the VR feeling is quite immersive. Additionally, the two controllers can be tracked by the system, enabling interactions in the virtual environment. Recently, additional tracking donuts where released. These can be mounted on arbitrary objects, which in turn can then be tracked in space by the SteamVR system. In the following sections, the tracking hardware is examined in more detail. The tracking system is patented [18], therefore most of the informations presented in this section stem from the publicly available patent, while the protocols described in 2.3.2 were reverse engineered by the open-source community.

Lighthouse

The two base stations, from here on referred to as lighthouses, are positioned around the tracking area, typically mounted on the room walls or camera stands to face each other. Figure 2.8 shows a disassembled first generation lighthouse. Each lighthouse contains two infrared lasers. The lasers are mounted at 90 degrees with respect to each other. Each laser beam is focused onto a 45 degree mirror inside a cylinder. This is depicted in Figure 2.9.

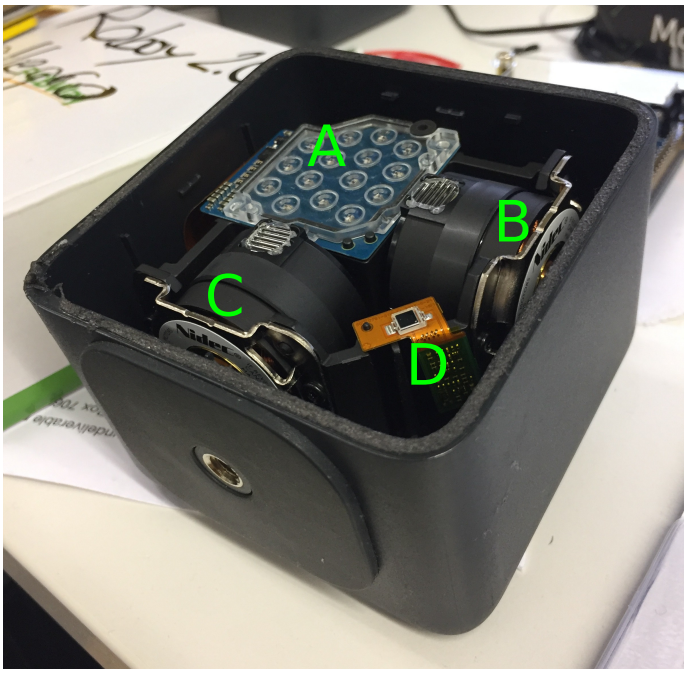


Figure 2.8: A disassembled lighthouse with the main components: A - LED grid, B and C - cylinders with fresnel lenses attached to motors, D - optical sensor

The cylinder has a Fresnel lens mounted on its circumference (Figure 2.8, B, and C). The Fresnel lens transforms the laser beam coming from the mirror into a plane. Each cylinder is connected to a motor, such that the infrared laser plane is sweeping the tracking area. Additionally, the lighthouses contain a grid of powerful infrared LEDs (Figure 2.8, A). The emitted infrared light is modulated with several MHz, which makes the system more robust against parasitic radiation from sunlight or other infrared sources. However, the system can not be used outside in full sunlight.

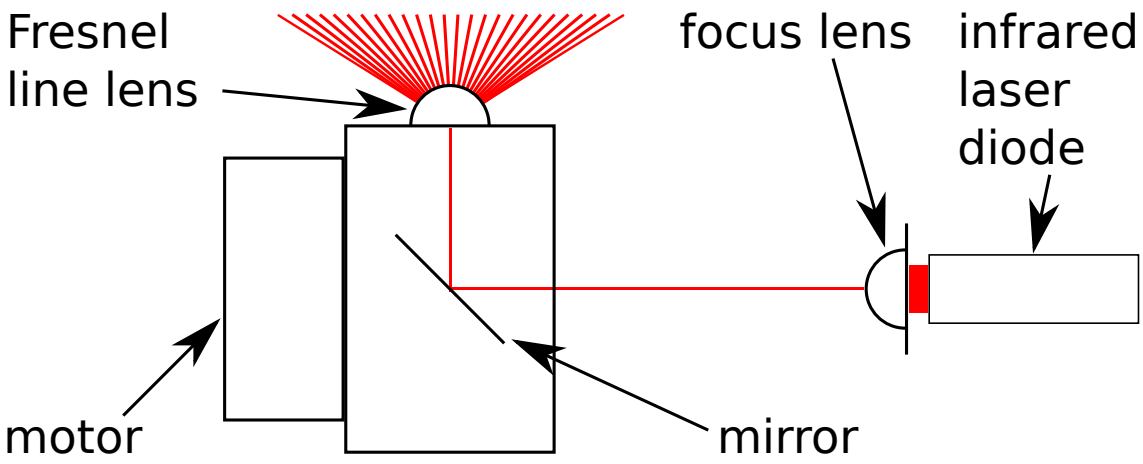
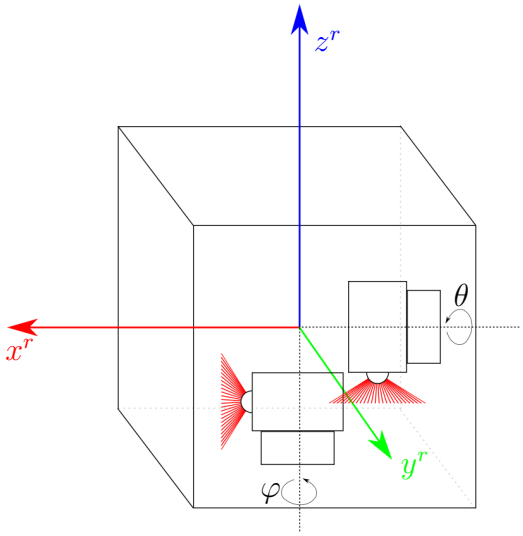
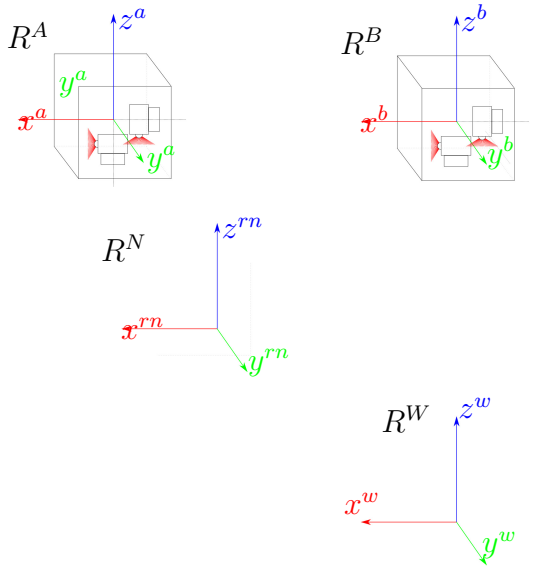


Figure 2.9: Lighthouse optical model

The coordinate system as depicted in Figure 2.10a is used as the local coordinate frame of each lighthouse. The azimuth φ is valid from $0 - 180$ degrees and starts from the x-axis. The elevation θ is valid from $0 - 180$ degrees and starts from the negative z-axis. Because of the housing of the lighthouse, the actual field of view (FOV) is smaller.



(a) Lighthouse local coordinate frame



(b) Global coordinate frame. R^A and R^B represent the pose of the lighthouses in the world coordinate frame R^W, while R^N represents the pose of the N-th object

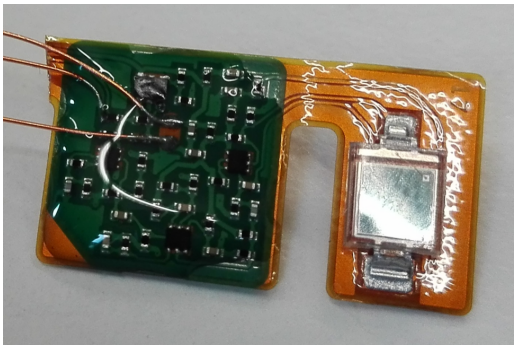
Figure 2.10: Lighthouse tracking coordinate frames

The world coordinate frame R^W is depicted in Fig. 2.10b. It is assumed that

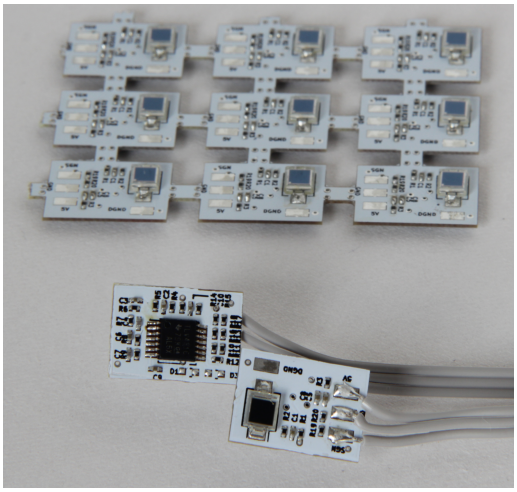
one or two lighthouses are used with coordinate frame R^A and R^B, respectively. The coordinate frame of a tracked object is referred to as R^N, where N defines the number of an object that is tracked by the system. $R^{TargetOrigin}$ shall denote the transformation from one coordinate frame to another. Thereby, as an example the transformation from the local coordinate frame of lighthouse A to object five would be R^{5A}.

Sensor

Every component of the HTC Vive system contains specialized optical sensors. These are able to detect the infrared signals emitted by the lighthouses. While each lighthouse contains only a single sensor (Figure 2.8, D), the HMD and controllers contain up to 24 sensors each. The sensors consist of an infrared sensitive photodiode plus circuitry to filter and amplify the infrared signals emitted by the lighthouses. The infrared signals are received by the photo-diode. This signal is bandpass filtered with the filter tuned to the modulation frequency of the infrared light. The filtered signal is then amplified in multiple stages. The result is a digital signal which is transmitted over a wire to a microcontroller.



(a) Lighthouse sensor disassembled from HTC Vive controller and interfaced using copper wires



(b) Custom lighthouse sensor

Figure 2.11: Original HTC Vive lighthouse sensor and the custom lighthouse sensors used for our tracking system

Extracting and preparing the disassembled sensors from the Vive controller as depicted in Figure 2.11a was very time-consuming. Another student in Roboy team (Luis Vergara) therefore developed a custom version of the sensor which we were able to produce on our own. The custom sensors are depicted in Figure 2.11b. They are double-sided printed circuit design (PCB)s, with an operational amplifier, filtering and amplifying the signal of the photo-diode in multiple stages.

2.3.2 Software

The tracking software, called SteamVR is proprietary, and the code is not publicly accessible. Development with the SteamVR system is typically done in Unity on Windows using the HMD and controllers tracking information to build VR applications. However, the open source community has reverse engineered the protocol [19] the lighthouses communicate with and how the sensor locations in space are measured. The following section describes the protocols used by the Vive tracking system in detail.

Lighthouse Sensor Protocol

The motors inside a lighthouse rotate at a constant speed of 60 Hz, with a phase offset of 180 degrees between the two motors. Every time the line lens of a motor passes the zero degrees mark, the infrared LED grid is flashed for a specific amount of time. The length of this pulse, in the following referred to as sync pulse, encodes three bits of information. The values are listed in Table 2.1.

sync pulse width [μs]	**skip**	**data**	**axis**
62.5	0	0	0
72.9	0	0	1
83.3	0	1	0
93.8	0	1	1
104	1	0	0
115	1	0	1
125	1	1	0
135	1	1	1

Table 2.1: lighthouse sync pulse encoding [19]

The skip bit encodes whether the current motor skips the current cycle or is active. The axis bit encodes which motor is currently active (vertical: 0, horizontal: 1).

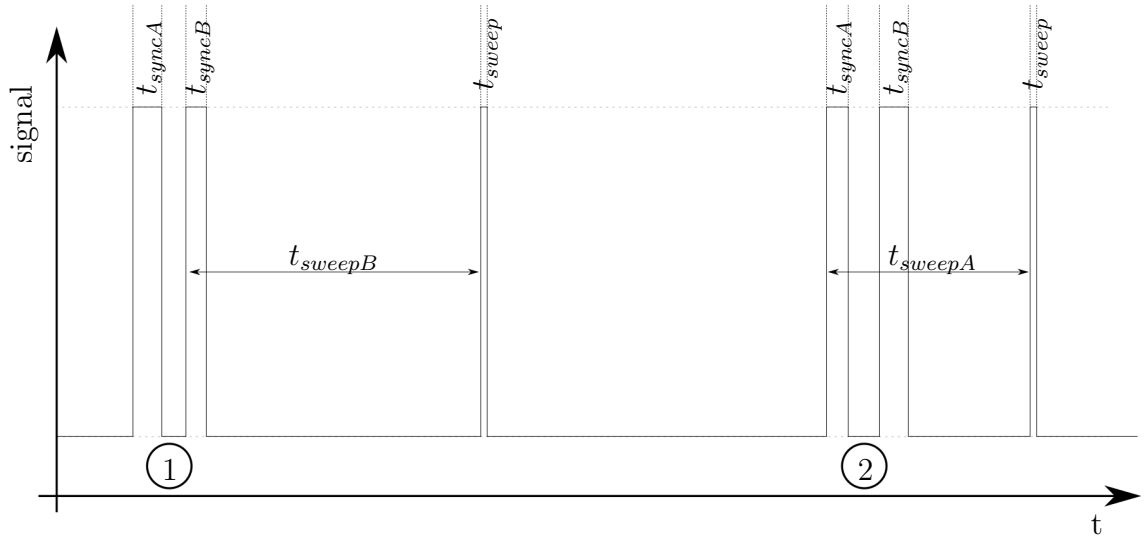


Figure 2.12: Lighthouse sensor signal protocol

A typical signal trace for a lighthouse sensor is illustrated in Figure 2.12. In Figure 2.12 two lighthouses are present. One after the other they flood the room with infrared light. The pulse width t_{syncA} and t_{syncB} encodes which lighthouse is active and with which motor (as described in Table 2.1). In cycle 1, lighthouse B is active and starts sweeping the room with its' infrared laser plane. After t_{sweepB} time, the sensor is hit by the sweep plane, which is indicated by a very short signal pulse t_{sweep}. The time t_{sweepB} can be directly used to calculate the motor bearing angle when the sensor was hit by the infrared plane. In cycle 2, lighthouse A is active. The lighthouses sequentially scan the room with their laser planes.

The sweep times t_{sweepA} and t_{sweepB} can be used to calculate the bearing angle for the currently active motor (elevation θ for vertical motor, azimuth φ for horizontal motor). Equation 2.1 and 2.2 show how this is calculated from the constant 60 Hz motor velocity, which equals $\approx 8.333ms$ for 180 degree rotation of a motor.

$$\theta = t_{sweep}/0.008333s * \pi \tag{2.1}$$

$$\varphi = t_{sweep}/0.008333s * \pi \tag{2.2}$$

Since the motors rotate at 60 Hz and there are two lighthouses, this gives an update rate of 30 Hz for each lighthouse with the vertical and horizontal angles interleaved sequentially.

The data bit is an additional bit sent with every sync pulse. The data bits, consecutively sent by each lighthouse, form the so-called Omnidirectional Optical Transmitter (OOTX) frame. The OOTX frame is shown in Figure 2.13.

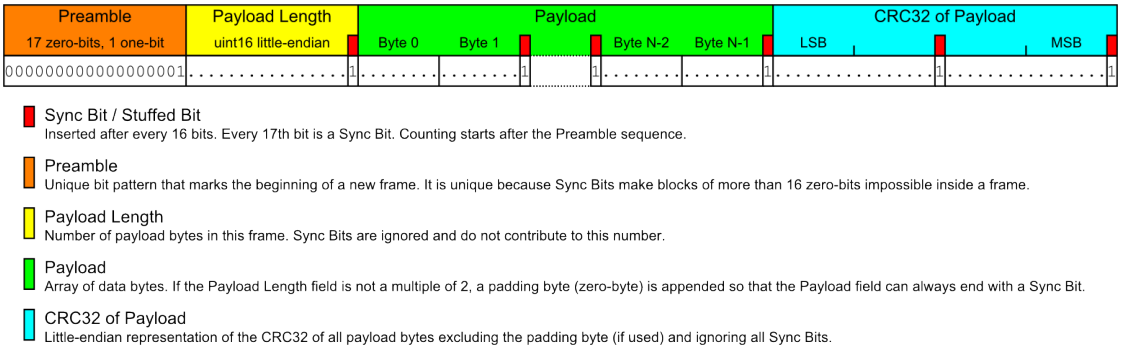
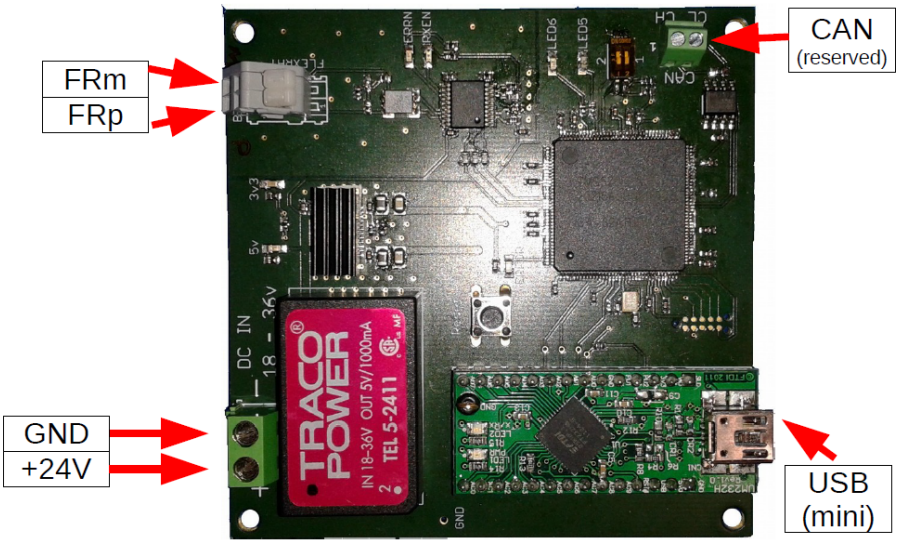


Figure 2.13: OOTX frame (according to [19])

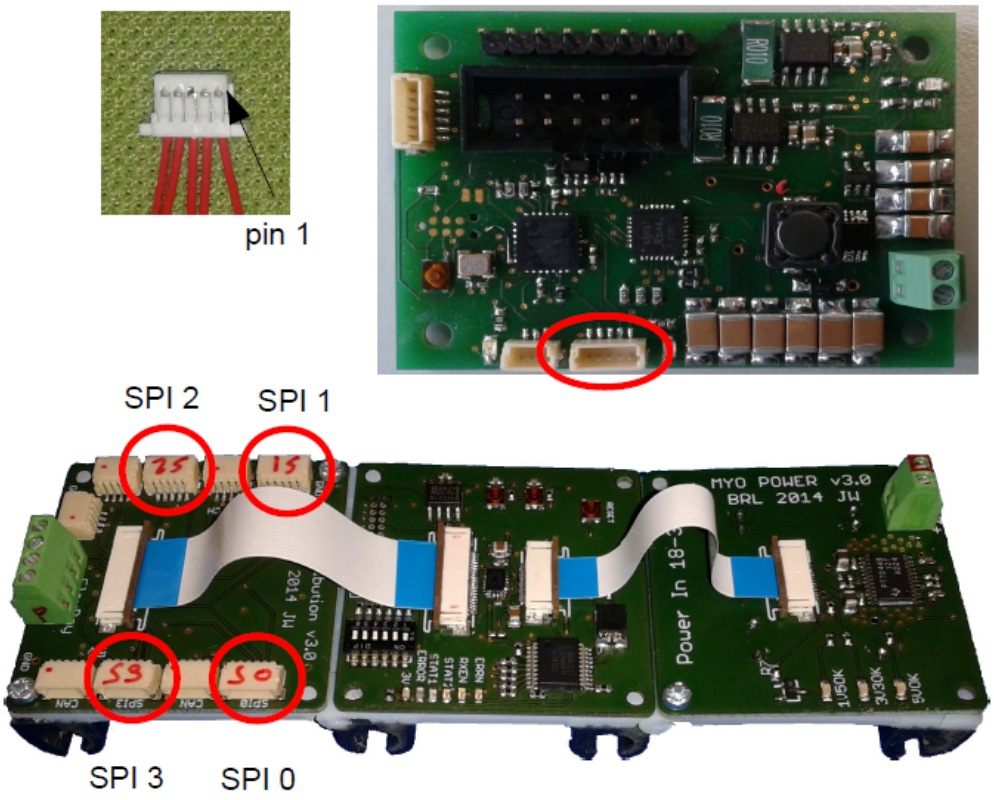
The payload of the OOTX frame contains information about each base station as listed in Table 2.2.

offset	**type**	**name**	**description**
0x00	uint16	fw_version	Firmware version and protocol version
0x02	uint32	ID	Unique identifier of the base station
0x06	float16	fcal.0.phase	”phase” for rotor 0
0x08	float16	fcal.1.phase	”phase” for rotor 1
0x0A	float16	fcal.0.tilt	”tilt” for rotor 0
0x0C	float16	fcal.1.tilt	”tilt” for rotor 1
0x0E	uint8	sys.unlock_count	rotor desynchronization counter
0x0F	uint8	hw_version	Hardware version
0x10	float16	fcal.0.curve	”curve” for rotor 0
0x12	float16	fcal.1.curve	”curve” for rotor 1
0x14	int8	accel.dir_x	”orientation vector”
0x16	int8	accel.dir_y	”orientation vector”
0x16	int8	accel.dir_z	”orientation vector”
0x17	float16	fcal.0.gibphase	”gibbous phase” for rotor 0
0x19	float16	fcal.1.gibphase	”gibbous phase” for rotor 1
0x1B	float16	fcal.0.gibmag	”gibbous magnitude” for rotor 0
0x1D	float16	fcal.1.gibmag	”gibbous magnitude” for rotor 1
0x1F	uint8	mode.current	selected mode (0=A, 1=B, 2=C)
0x20	uint8	sys.faults	”fault detect flags” (should be 0)

Table 2.2: Lighthouse OOTX info block (according to [20])



(a) USB-FlexRay Bridge



(b) MyoGanglion (bottom) with BLDC motor board (top)

Figure 2.2: MyoRobotics hardware (images from [9])

Chapter 3

System Design

This chapter describes the system design for MyoControl (Section 3.2), PowerLink (Section 3.3) and LighthouseTracking (Section 3.4). Every system implemented for this thesis makes extensive use of FPGAs. The DE10-Nano-SoC [21] is a FPGA development board (depicted in Figure 3.1) from Altera. It was chosen for this thesis for the following reasons:

- versatility: The Intel Cyclone V SE5CSEBA6U23I7 chip combines a 800MHz Dual-core Advanced RISC Machine (ARM) Cortex-A9 with a FPGA (with 110000 logical elements (LEs), 120 digital signal processing blocks (DSPs)). This combination allows very flexible system designs
- availability: The development board is available off the shelf
- prize: The price is around 100 Euro per board
- size: Compared to other FPGA development boards it is quite small, measuring 100mm x 70mm x 25mm

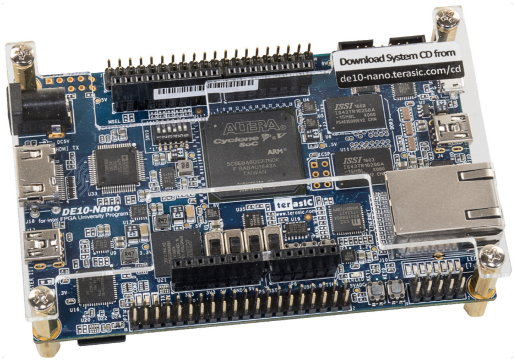


Figure 3.1: DE10-Nano-SoC board (image from [22])

The ARM cores and the FPGA portion of the Cyclone V chip are connected via

Alteras' high-speed Advanced eXtensible Interface (AXI). Thereby, the modules synthesized for the FPGA can be controlled from user applications running on the ARM cores. Hence, low-level hardware control can be outsourced to the FPGA. The ARM cores are alleviated from this workload and can be used for high-level control and communication with the host PC.

3.1 Overview

This section gives an overview of the overall system architecture for Roboy 2.0. Roboy consists of six body parts, and each body part contains a DE10-Nano-SoC FPGA. The communication architecture is illustrated in Figure 3.2. The FPGAs of each body part are connected to an Ethernet hub and form a hard real-time openPOWERLINK network. External control of Roboy is established via a legacy Ethernet connection from a PC/laptop using ROS messages and services. Each body part contains specialized electronics hardware as required by its' function. The following list summarizes the main components:

- head: Contains a laser projector for projecting his face, a stereo ZED camera, two loudspeakers, a microphone and an Odroid development board for control of these multimedia devices. Four muscle units are used to actuate his head.

- torso: A flexible spine actuated by six muscle units.

- left/right arm: the shoulder is a 3-DOF ball in socket joint which is actuated by nine muscle units. The elbow is a 1-DOF joint and actuated by two muscle units. The forearm can be rotated via two muscle units attached to his upper arm. The forearm contains 20 servo motors for actuating the hand.

- left/right leg: Roboy 2.0 is supposed to ride an electric bike; therefore the legs have been designed to enable the pedaling movements. Each leg contains six muscle units.

The low-level control of the muscle units was named MyoControl and Section 3.2 describes the implementation. The six FPGAs of Roboy 2.0 were connected using the hard real-time Ethernet extension openPOWERLINK. High-level control of Roboy is established using ROS via legacy Ethernet connection to the openPOWERLINK network. The implementation of the openPOWERLINK network is described in Section 3.3. An external tracking system was implemented, called DarkRoom lighthouse tracking. The implementation is described in Section 3.4. The pin assignment for the DE10-Nano-SoC FPGAs of Roboy 2.0 is shown in Appendix B.

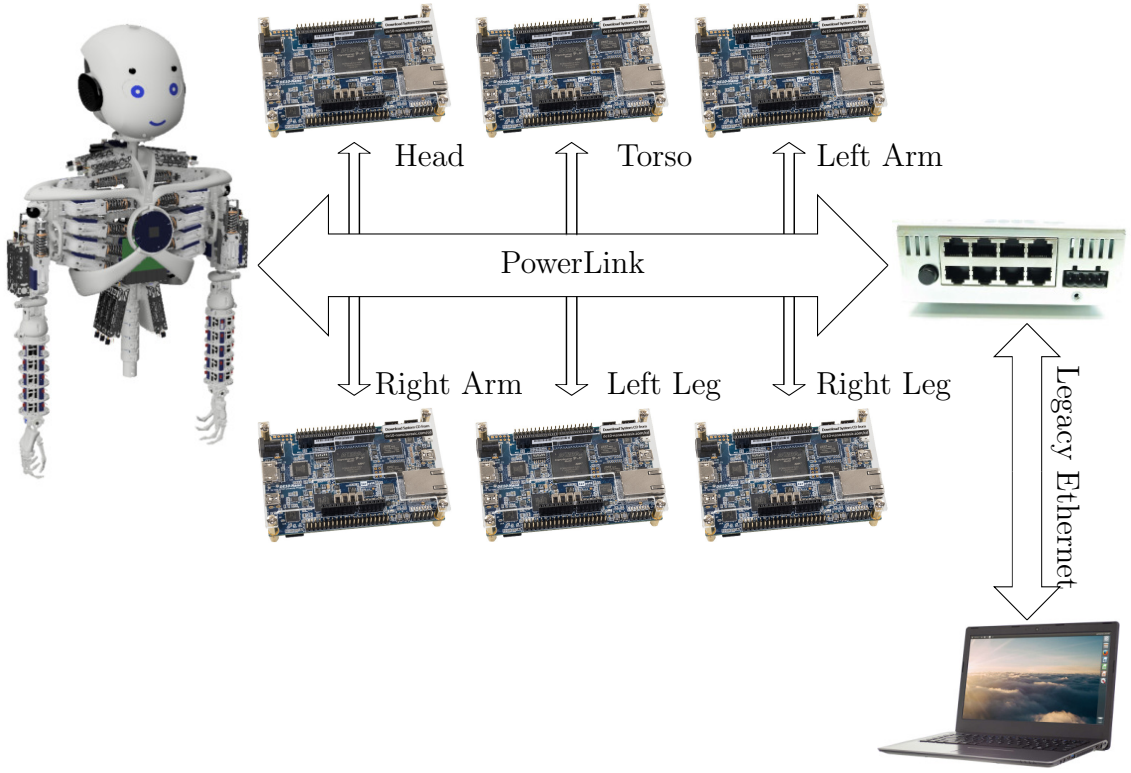


Figure 3.2: Roboy 2.0 communication architecture (Image from [22])

3.2 MyoControl

The goal of the MyoControl subsystem was to replace the MyoGanglion from the MyoToolKit. There are three main reasons, why this was favorable:

1. One MyoGanglion could only control up to four muscle units.

2. A total of six MyoGanglions could be used on one FlexRay bus. This totals to 24 muscle units that could be controlled on one FlexRay bus. Roboy 2.0 contains more than 50 motors, therefore at least two FlexRay buses would have been necessary.

3. The custom PCB was not available off the shelves, since the PCB needed to be manufactured and then assembled. This together with the low volumes made these boards prohibitively expensive.

The MyoControl system can be divided into two main components, the communication with a motor board and the PID control. The first step was to establish communication with a motor board using a FPGA.

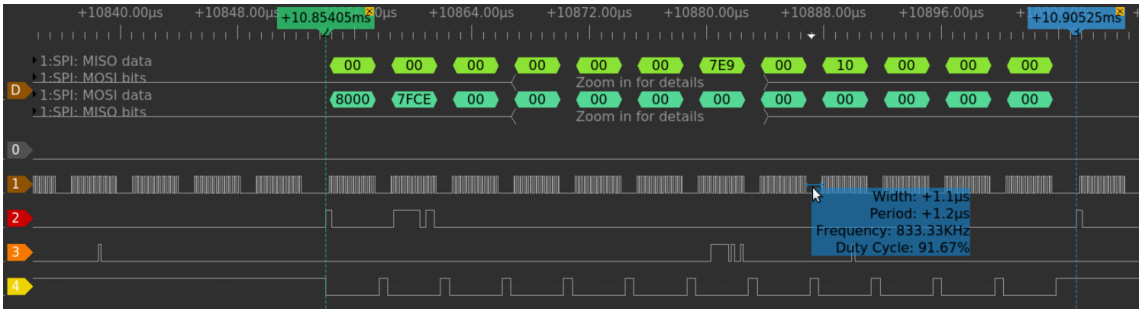
3.2.1 Communication

The communication with the motor boards works via Serial Peripheral Interface (SPI). An open-source Verilog SPI implementation was therefore used from opencores.org [23]. The motor boards have specific timing requirements for the SPI communication which include the custom parameters listed in Table 3.1.

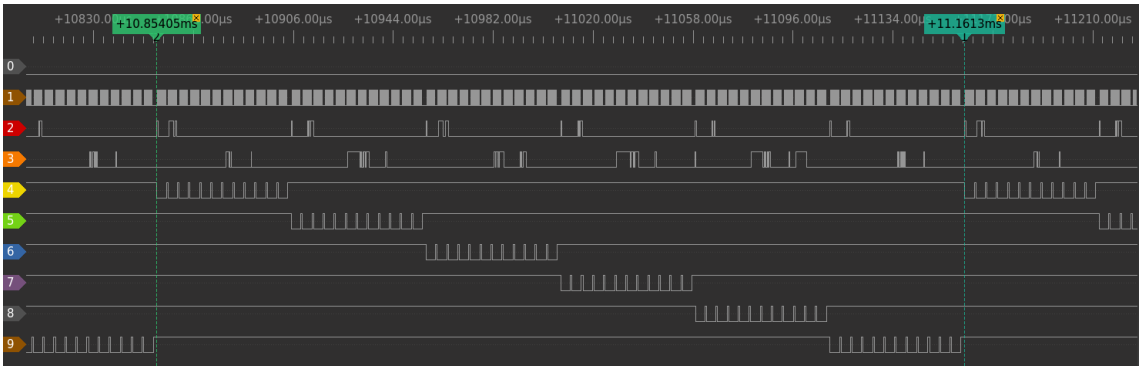
clk frequency	clk polarity	clk phase	word length	target delay
2 Mhz	0	1	16 bit MSB	$1.2\mu s$

Table 3.1: Motor board SPI requirements

(a) SPI frame for communication with motor board (image from [9])



(b) SPI communication frame with a single motor board

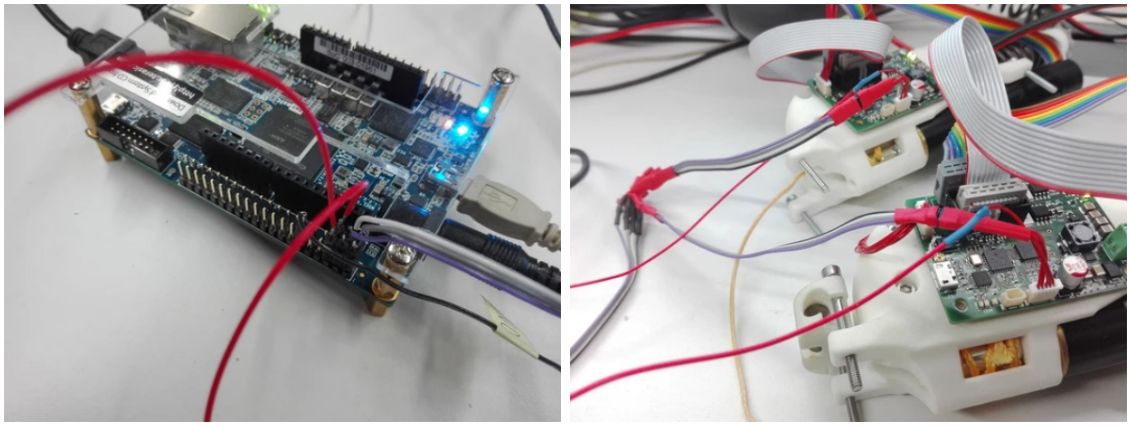


(c) SPI communication with six motor boards

Figure 3.3: MyoControl SPI communication frame and live data recorded with a logic analyser

A distinct SPI frame was defined in the MyoToolKit which is shown in Figure 3.3a. Data is transmitted in 16-bit blocks as specified in Table 3.1. Each 16-bit word contains one piece of information, except for the motor position which is a

concatenation of two 16 bit words. An example communication with one motor board is shown in Figure 3.3b which was recorded using a logic analyser. The FPGA starts the communication with the motor board by pulling the respective slave select line low (in Figure 3.3b this is the yellow channel four), then starts clocking in the data via the master out slave in (MOSI) line (in Figure 3.3b this is the red channel two). The start of a MyoToolKit SPI frame is always initiated with a 0x8000, signaling the start of the frame. After transmission of a 16 bit word, the slave select line is pulled high for the target delay of $1.2\mu s$. Then the slave select line is pulled low again and the next 16 bit word of the SPI frame is transmitted. This is repeated until the 12 words of the frame have been transmitted. The SPI transmission is defacto half-duplex, because the master first transmits four words (message type, PWM, ctrl flags 1, ctrl flags 2), then the motor board transmits it's data to the FPGA (motor position/velocity, current, displacement, sensor 1, sensor 2, error flags) using the master in slave out (MISO) line (in Figure 3.3b this is the orange channel three).



(a) SPI bus and two red slave select cables connected to general-purpose input/output (GPIO)s of DE10-Nano-SoC

(b) two muscle units sharing the SPI bus

Figure 3.4: MyoControl SPI bus connecting two muscle units to the DE10-Nano-SoC

The logic for the transmission of the MyoToolKit SPI frame was implemented in Verilog in `SpiControl.v`. The target delay is handled by a delay counter between consecutive SPI transmissions. The sequential transmission of the frame was implemented using a word counter which is latching in or out the current data to the SPI module. For the legacy MyoGanglion, each motor board was connected to a separate SPI bus. For the new communication, an arbitrary amount of motor boards can be connected to a single SPI bus and separate slave select lines connected to the GPIOs of the FPGA. Figure 3.4a shows the DE10-Nano-SoC with the SPI bus and two red slave select cables connected to its' GPIOs. Figure 3.4b shows the other end of the SPI bus, where two muscle units are connected to the same SPI bus and the slave select cables, one for each motor board.

The control logic for the sequential communication with each motor board was implemented in Verilog in `MYOControl.v`. The MyoControl module has the following features:

1. motor number: It parametrizes the number of motors connected to one module. The MyoControl module is implemented as an Altera Qsys intellectual property (IP), such that the number of motors can be easily configured when the Qsys design is generated, without having to change the Verilog code directly.

2. Avalon interface: It exposes the PID parameters and motor values via the lightweight AXI bridge. The PID parameters for each motor can be read and written.

3. control frequency: The maximal cycle frequency is limited by the number of motor boards connected to one SPI bus. For six motors this was approx. 3250 Hz, which can be seen in Figure 3.3c. While generally, the fastest possible cycle frequency is desirable, MyoControl was augmented with a delay counter, such that the cycle frequency can be set to any value below this maximal frequency.

3.2.2 PID control

The legacy MyoGanglion from the MyoToolKit implements four PID controller for controlling each motor board in different control modes (as described in Section 2.1.2). The PID controller, implemented in C++ for the MyoGanglion, was ported to a Verilog version in `PIDController.v`. One major difference between the legacy PID controller and the new controller synthesized on the FPGA was the usage of integer arithmetic only. The overhead of fixed point or even floating point arithmetic for the FPGA seemed unreasonable because the values to be controlled were integers anyway. The motor position is measured in encoder ticks, the motor velocity in encoder ticks/s and the displacement is also measured in ticks.

The new PID controller features the following control modes:

- position: motor position control in encoder ticks
- velocity: motor velocity control in encoder ticks/s
- displacement: spring displacement control in encoder ticks, where one tick equals a displacement of 0.1 mm

The force control mode was omitted, because it can be easily dealt with in high-level control, where the spring parameters are used to estimate the necessary displacement for a given force.

One particular requirement was to send the data in the big-endian format because the legacy MyoGanglion is big-endian. This was easily done in Verilog because for the FPGA there is, in fact, no difference between little or big-endian. The difference in the code was a mere switch of indexing.

3.3 openPOWERLINK

The openPOWERLINK stack can be built for Ubuntu 16.04 using libpcap. This limits the minimum cycle length to approx. 20 ms, which is not fast enough for accurate motion control of 50 motors. Alternatively, the kernel part of the openPOWERLINK stack can be synthesized in FPGA. This enables very small cycle lengths with ultra-low jitter, down to $400\mu s$ and $< 1\mu s$, respectively.

3.3.1 Porting to DE10-Nano-SoC

The openPOWERLINK examples available for Cyclone V are for different FPGA development boards. Adapting the modules for our DE10-Nano-SoC was necessary. Specifically, the hardware peripheral system (HPS) and the connections to the peripherals of the DE10-Nano-SoC had to be adapted.

The lowest layer in the ISO/OSI network standard is the PHY. In the openPOWERLINK stack, a FPGA module called OpenMac is directly connected to an Ethernet transceiver PHY. For the DE10-Nano-SoC this is the KSZ9031RNX chip from Micrel. This chip is connected to the HPS and has no direct connection to the FPGA portion of the Cyclone V. It is possible to multiplex the Reduced Gigabit Media-Independent Interface (RGMII) and Management Data Input/Output (MDIO) interface of the Ethernet transceiver to the FPGA portion, using so-called loanios. Unfortunately, the OpenMac module only supports MII or RMII. These are not compatible with RGMII. There exists an Altera IP for converting RGMII to RMII, called "Intel FPGA GMII to RGMII Converter Core". This did not function as expected, and the communication failed. It was, therefore, necessary to use an external RMII Ethernet transceiver. We chose a small external board from Waveshare with a DP83848 transceiver chip from Texas Instruments, a RJ45 connector and an on-board 50 MHz oscillator. The transceiver board was directly connected to the GPIOs of the FPGA. The external transceiver board is shown in Figure 3.5 with its RMII and power connections to the GPIOs of the FPGA.

The 50 MHz for the RMII transmission needed to be in sync with the OpenMac module synthesized for the FPGA, therefore the on-board oscillator was desoldered from the transceiver board and the 50 MHz system clock of the FPGA was routed to the oscillator pin of the external transceiver.

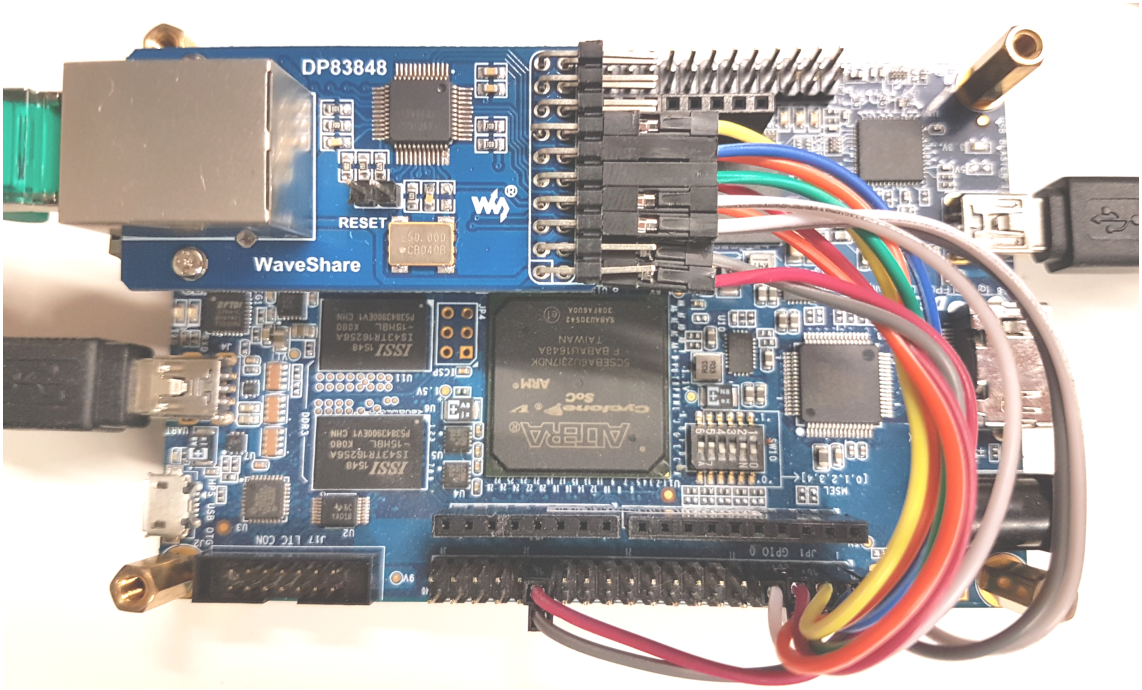


Figure 3.5: DE10-Nano-SoC board with the external Ethernet transceiver and the connections to the GPIOs of the FPGA

The kernel part of openPOWERLINK is implemented in the FPGA portion using a reduced instruction set computer (RISC) called Nios2, which is a licensed IP from Altera. The license for the fastest version of this RISC has to be obtained from a regional distributor. For evaluation, the Nios2 core can be used as long as the Joint Test Action Group (JTAG) connection with the host is established. The Nios2 handles the low-level initialization and communication for openPOWERLINK. The user application is implemented as a bare-metal program running on one of the ARM cores. The communication data, received by the Nios2, is shared with the ARM core using a shared memory region.

3.3.2 Network Profile

The next step was to define a communication profile for Roboy 2.0. openCONFIGURATOR (a plugin for the eclipse IDE) was used to configure the openPOWERLINK network. The network was configured to contain one MN and five CNs as required by the design of Roboy 2.0 (cf. Figure 3.2).

The Object Dictionary was augmented with entries for control of 12 motors connected to each FPGA. Since an external transceiver was unavoidable, the integrated Ethernet transceiver of the DE10-Nano-SoC could be used to transmit lighthouse sensor data, visual data from the stereo camera and data from any other future

sensors of Roboy 2.0. Also, the high-level cognition control could be sent via the integrated Ethernet transceiver of the DE10-Nano-SoC. The communication profile was therefore exclusively reserved for motor control.

3.4 Lighthouse Tracking

This section describes the implementation of our lighthouse tracking system from low-level sensor decoding to high-level pose estimation.

3.4.1 Lighhouse Sensor Protocol

The lighthouse sensor protocol can be decoded with a standard microcontroller. The signal line of a lighthouse sensor is connected to a GPIO of the microcontroller. Interrupts on rising and falling edges of the signal trigger an interrupt service routine (ISR) which uses a timer to timestamp the changes in the signal. Unfortunately, this works only for a limited amount of sensors, because the microcontroller is interrupted too much if a critical amount of sensors is exceeded. The critical amount depends on the microcontrollers interrupt capabilities and speed. For estimating the 6-DOF pose of an object, at least four 3D positions are necessary. This already exceeded the compute capabilities of the boards we tested (Intel Edison, Arduino MKR1000). This was the reason we chose to implement the lighthouse protocol decoding in FPGA.

The incoming sensor signal is illustrated in Figure 3.6. This shows the typical scenario when two lighthouses are active. However, the mechanism is the same when only one lighthouse is used. As can be observed in Figure 3.6, the two lighthouses sequentially flood the room with infrared light, and the sync pulse width encodes if a lighthouse will skip or will be active in the current cycle. The time span t_{b0} between the sync pulse of lighthouse B and the moment the sweeping laser plane hits the sensor can be used to calculate the bearing angle of the currently active motor for lighthouse B (as described in Section 2.3.2). The time span t_{sync} between two pulse widths signaling an active motor can be used to differentiate between the two lighthouses. This is illustrated in Figure 3.6 by the green and red arrows. If a lighthouse stays active, the time span t_{sync} is approx. $8333\mu s$ (which is the cycle time when the motors run at 60 Hz). In Figure 3.6, lighthouse B stays active in cycle 1, which is marked with a green t_{sync} arrow. In cycle 2 the time span t_{sync} will be smaller than $8333\mu s$ by a constant time, which signals that lighthouse A becomes active (which is marked with a red arrow). In cycle 3, lighthouse A stays active. In the next cycle lighthouse B will become active again, which is indicated by t_{sync} being longer than $8333\mu s$ by a constant time.

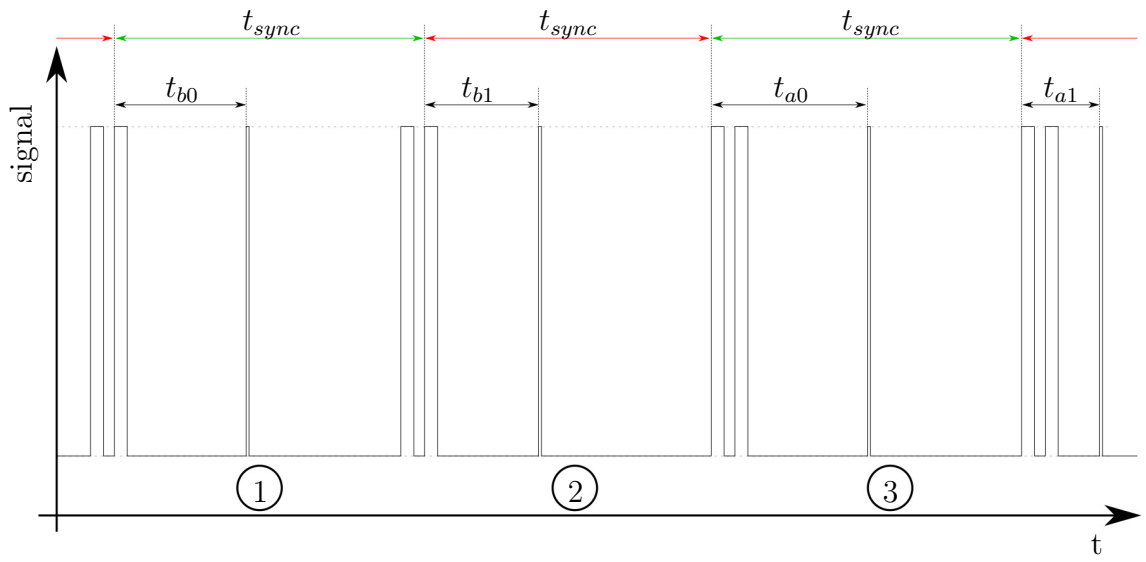


Figure 3.6: Lighthouse sensor signal protocol illustration

The logic for decoding this protocol was implemented in Very High Speed Integrated Circuit Hardware Description Language (VHDL) in `lighthouse_sensor.vhdl`. Each lighthouse_sensor module decodes the signal of one sensor. All synthesized modules run in parallel on the FPGA, which means the number of sensors, that can be decoded is merely limited by the size of the FPGA and the amount of GPIOs it has. At 50 MHz system clock, each clock tick is $1/50\mu s$. For $8333\mu s$ for the 180 degrees at 60 Hz motor speed, this gives a resolution of approximately 0.00043 degree/tick.

The decoded sensor value is packed into a 32-bit field. This is illustrated in Figure 3.7. The field values encode the following data:

- bits 18-0: The decoded bearing angle in ticks of the clock at which the lighthouse_sensor module runs. For the DE10-Nano-SoC we connected the system clock running at 50 MHz.
- bits 28-19: The sensor ID which is a number starting from 0 to the number of sensors that were chosen to be generated. Each sensor has its own unique ID.
- bit 29: Indicates if the decoding is valid. The lighthouse_sensor module checks if the sweep duration is in the range $(15000, 400000)$, which equals a valid bearing angle in the range of approx. $(6, 172)$ degrees. A zero indicates invalid bearing angle, and a one indicates valid bearing angle.
- bit 30: Motor axis, zero for the horizontal motor, one for the vertical motor.
- bit 31: lighthouse ID, zero for lighthouse A and one for lighthouse B

A module called DarkRoom was implemented in Verilog in `DarkRoom.v`, which serves three purposes:

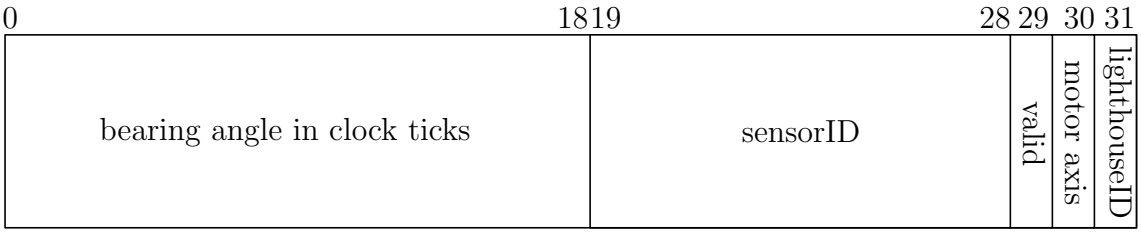


Figure 3.7: lighthouse_sensor result 32-bit field

1. lighthouse_sensor generation: It parametrizes the number of sensors, that should be synthesized. The DarkRoom module is implemented as an Altera Qsys IP, such that the number of sensors can be easily configured when the Qsys design is generated, without having to change the Verilog code directly.

2. Avalon interface: It exposes the decoded signals of the lighthouse_sensor modules via the lightweight AXI bridge. A single read can retrieve each sensor result 32-bit field via this interface.

3. ESP8266 SPI interface: For mobile applications, the DarkRoom module implements a SPI master that triggers the SPI transmission of all sensor values when any sensor detects a non-skipping sync pulse. The SPI transmission is compatible with the low-cost and small ESP8266 chip. The SPI timing was chosen accordingly. Each SPI frame consists of 256 bits, which equals up to eight 32-bit sensor results. All sensor values are transmitted in consecutive frames. An Arduino sketch called `darkroom_esp8266.ino` was written for the ESP8266, which packs the values into UDP packets and transmits them wirelessly to a target PC. The SPI transmission is optional and can be selected in the DarkRoom IP properties in Qsys.

A module called lighthouse_ootx_decoder was implemented in VHDL in `lighthouse_ootx_decoder.vhdl`, which decodes the OOTX frame for two lighthouses. It was implemented using a finite state machine (FSM). As described in Section 2.3.2, the data bits encoded in consecutive sync pulses form the OOTX frame. The lighthouse_ootx_decoder module first detects the OOTX preamble, then makes sure the frame format is valid. The decoder uses a single sensor channel for decoding. If decoding was unsuccessful, because of occlusion or a defective sensor, the sensor channel can be changed.

A module called DarkRoomOOTXdecoder was implemented in Verilog in `DarkRoomOOTXdecoder.v`, which serves two purposes:

1. Avalon interface: It exposes the decoded OOTX frame values via the Avalon lightweight AXI bridge. The sensor signal channel used for decoding the OOTX frame can be changed via this Avalon interface as well.

2. universal asynchronous receiver-transmitter (UART): A UART module was

downloaded from nandland.com [24]. The DarkRoomOOTXdecoder module implements control logic, that transmits the decoded OOTX frame in 8-bit big-endian chunks via UART. An arduino sketch called `ootx_frame_uart.ino` was implemented that receives the frames via UART and checks with the cyclic redundancy check (CRC)32 checksum if they were correctly decoded. The UART transmission is optional and can be selected in the DarkRoomOOTXdecoder IP properties in Qsys.

3.4.2 DarkRoom

Lighthouse ray calculation

The normal vectors of the horizontal and vertical sweep planes (n_h and n_v) can be calculated from the bearing angles using the axis offset d_{axis} for the respective motor as described in Equation 3.1 and 3.2. The parameters $tilt_\theta$ and $tilt_\varphi$ are lighthouse calibration parameters.

$$n_h = \begin{bmatrix} d_{axis} \\ \sin(\theta) \\ -\cos(\theta) \end{bmatrix} \times \begin{bmatrix} \cos(tilt_\theta) \\ 0 \\ \sin(tilt_\theta) \end{bmatrix} \tag{3.1}$$

$$n_v = \begin{bmatrix} \cos(\varphi) \\ \sin(\varphi) \\ d_{axis} \end{bmatrix} \times \begin{bmatrix} \sin(tilt_\varphi) \\ 0 \\ \cos(tilt_\varphi) \end{bmatrix} \tag{3.2}$$

The ray p pointing at the sensor can be calculated from the intersection of these two plane normals:

$$p = (n_v \times n_h) / \left\| n_v \times n_h \right\|^2 \tag{3.3}$$

The position of the sensor P in 3D space with respect to a lighthouse can be represented using the ray p:

$$P = R * p \tag{3.4}$$

In Equation 3.4 R represents the distance of the sensor to the virtual optical center of a lighthouse as constructed by the two intersecting sweep planes. The rays of two lighthouses seeing the same sensor can be used to triangulate the 3D position of the sensor.

Pose estimation

Two methods were implemented to estimate the 6-DOF pose of a tracked object. Both methods require the knowledge of the lighthouse sensor locations on a tracked object, relative to an arbitrarily defined local coordinate system. The construction of a tracked object can be done in computer aided design (CAD). The CAD program

of choice was Fusion 360 from Autodesk. It is the same program Roboy 2.0 was designed with. Fusion 360 exposes most of it's functionality in a python application programming interface (API). Custom plugins can be implemented for Fusion 360. For this thesis two plugins were implemented/augmented:

1. DarkRoomGenerator: Enables the user to define construction points with a specific naming scheme, by simply selecting the desired lighthouse sensor location on a mesh constructed in Fusion 360.
2. SDFusion: A plugin for exporting a constructed robot from Fusion 360 into a format compatible with the open-source simulator Gazebo. This plugin was developed in previous semesters. It was augmented for exporting a configuration file containing the relative lighthouse sensor locations by parsing the construction point names as generated by the DarkRoomGenerator plugin or by hand. Our lighthouse tracking system can directly load this configuration file and use it for pose estimation.

The first method referred to as triangulation pose estimation (TPE) from here on, depends on a minimum of four sensors to be visible by two lighthouses. The sensor positions are triangulated. A non-linear optimizer is used to estimate a transformation matrix which maps the relative sensor positions, as exported from Fusion, onto the triangulated sensor positions. The objective function is shown in Equation 3.5, where T represents the transformation matrix of interest, P_r contains the relative and P_t the triangulated sensor positions.

$$0 = (T * P_r - P_t) \tag{3.5}$$

Both P_r and P_t contain the 3D positions as homogeneous coordinates, such that the dimensions are 4xM, where M represents the number of visible, triangulated sensors. The transformation matrix T is a 4x4 homogeneous matrix, with 3x3 rotational matrix and 3x1 translation vector. In the objective function of the non-linear optimizer, the rotational part is represented by a rotation vector with enforced constraints using quaternion unit sphere projection as described in [25]. The Levenberg-Marquardt non-linear optimizer from the C++ Eigen library was used.

The second method referred to as multi lighthouse pose estimation (MLPE) from here on, was adapted from [26]. As described in [26], the relation between the 3D position of a sensor in the world coordinate frame and the pseudo pixel coordinates of a respective lighthouse satisfy:

$$Z_c \begin{bmatrix} u_i \\ 1 \\ v_i \end{bmatrix} = K_k \begin{bmatrix} R^{kW} & T^{kW} \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} R^{WN} & T^{WN} \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} x^{rN} & y^{rN} & z^{rN} & 1 \end{bmatrix}^T \tag{3.6}$$

The transformation of interest is represented by R^{WN} and T^{WN} in Equation 3.6 which is the transformation from the local coordinate system of the N-th tracked

object to the world coordinate frame (as defined in Figure 2.10b). In Equation 3.6 x^{rN}, y^{rN} and z^{rN} are the relative sensor locations as exported from Fusion 360. The transformation represented by R^{kW} and T^{kW} define the world pose of the k-th lighthouse. In Equation 3.6, u_{iN} and v_{iN} represent the pseudo pixel of the i-th sensor for the respective N-th tracked object in the x-z-plane of the k-th lighthouse (as defined in Figure 2.10a) and can be calculated from the bearing angles in the following way:

$$u_{iN} = \tan(\pi/2 - \varphi_{iN}) \tag{3.7}$$
$$v_{iN} = \tan(\theta_{iN} - \pi/2) \tag{3.8}$$

The focal length is set to one and no distortion is assumed which gives the unit matrix for the intrinsic camera matrices K_k for all lighthouses. Equation 3.6 can then be simplified (for brevity only one tracked object is considered):

$$Z_c \begin{bmatrix} u_i \\ 1 \\ v_i \end{bmatrix} = R_k M \begin{bmatrix} x_r & y_r & z_r & 1 \end{bmatrix}^T \tag{3.9}$$

$$= \begin{bmatrix} P_{k0}^T \\ P_{k1}^T \\ P_{k2}^T \end{bmatrix} \begin{bmatrix} M_0 & M_1 & M_2 & M_3 \end{bmatrix} \begin{bmatrix} x_r & y_r & z_r & 1 \end{bmatrix}^T \tag{3.10}$$

$$= \begin{bmatrix} P_{k0}^T M_1 & P_{k0}^T M_2 & P_{k0}^T M_3 & P_{k0}^T M_4 \\ P_{k1}^T M_1 & P_{k1}^T M_2 & P_{k1}^T M_3 & P_{k1}^T M_4 \\ P_{k2}^T M_1 & P_{k2}^T M_2 & P_{k2}^T M_3 & P_{k2}^T M_4 \end{bmatrix} \begin{bmatrix} x_r & y_r & z_r & 1 \end{bmatrix}^T \tag{3.11}$$

In Equation 3.9, P_k represents the pose matrix of lighthouse k and M represents the pose of interest of the tracked object. In Equation 3.10 and 3.11 P_{kl}^T denotes the l-th row of P_k and M_j represents the j-th column of M. Eliminating Z_c in the first and third row of Equation 3.11 results in:

$$0 = (P_{k1}^T u_i - P_{k0}^T)(M_0 x_r + M_1 y_r + M_2 z_r + M_3) \tag{3.12}$$
$$0 = (P_{k1}^T u_i - P_{k2}^T)(M_0 x_r + M_1 y_r + M_2 z_r + M_3) \tag{3.13}$$

Denoting $P_{k1}^T u_i - P_{k0}^T$ by C_i and $P_{k1}^T u_i - P_{k2}^T$ by D_i, Equations 3.12 and 3.13 can be

expressed in the form of a non-homogeneous linear equation $Ax = b$:

$$\begin{bmatrix} C_i[0]x_r & D_i[0]x_r \\ C_i[1]x_r & D_i[1]x_r \\ C_i[2]x_r & D_i[2]x_r \\ C_i[0]y_r & D_i[0]y_r \\ C_i[1]y_r & D_i[1]y_r \\ C_i[2]y_r & D_i[2]y_r \\ C_i[0]z_r & D_i[0]z_r \\ C_i[1]z_r & D_i[1]z_r \\ C_i[2]z_r & D_i[2]z_r \\ C_i[0] & D_i[0] \\ C_i[1] & D_i[1] \\ C_i[2] & D_i[2] \end{bmatrix}^T \begin{bmatrix} r_{00} \\ r_{10} \\ r_{20} \\ r_{01} \\ r_{11} \\ r_{21} \\ r_{02} \\ r_{12} \\ r_{22} \\ t_0 \\ t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} -C[3] \\ -D[3] \end{bmatrix}$$

The 2x12 A matrices and 2x1 b vectors can be stacked for all sensors visible to any amount of lighthouses. For an unambigous result, at least six sensor values must be available. The same non-linear optimizer was used as above to find the pose vector x with respect to $Ax - b = 0$. Again quaternions with unit-sphere projection as described in [25] were used to enforce valid rotations.

The advantage of this approach over the first is that a sensor does not need to be visible by two lighthouses. Additionally, sensor values from one to possibly many lighthouses can be fused into one coherent pose estimate. The only prerequisite for both approaches is that the global lighthouse poses are known.

Calibration

Due to fabrication tolerances, the optical system as depicted in Figure 2.9 is far from perfect. The influences are different for every lighthouse and must be compensated. The uncompensated bearing angles will be referred to as pseudo bearing angles. There are four main influences:

1. phase offset: The moment the LED grid is flashing to signal the zero crossing of a motor has a constant offset from the true moment in time
2. tilt: Due to manufacturing tolerances, the motors are not perfectly perpendicular to each other
3. sweep plane curvature: The sweep plane is not a straight line but curves to the sides, which is caused by the low quality of the Fresnel lens
4. lens eccentricity: The lenses are not perfectly aligned, which causes a varying sweep plane offset depending on the current motor position

In the OOTX frame, transmitted by the lighthouses regularly, there are ten focal parameters. These were assumed to represent the factory calibration values. Although the exact meaning was not known, there existed some hypothesis about what they might mean. The following non-linear correction functions were chosen for this thesis.

The correction function used for compensation of the phase offsets are simple offset angles $phase_\theta$ and $phase_\varphi$ for the vertical and horizontal motor, respectively:

$$\theta' = \theta + phase_\theta \tag{3.14}$$
$$\varphi' = \varphi + phase_\varphi \tag{3.15}$$

For compensating the curvature of the sweep planes, a quadratic function was chosen, which curves from the optical center to the sides using the offset corrected bearing angles from Equations 3.14 and 3.15:

$$\beta_\theta = curve_\theta * (\cos(\varphi') * \sin(\theta'))^2 \tag{3.16}$$
$$\beta_\varphi = curve_\varphi * (-\sin(\varphi') * \cos(\theta'))^2 \tag{3.17}$$

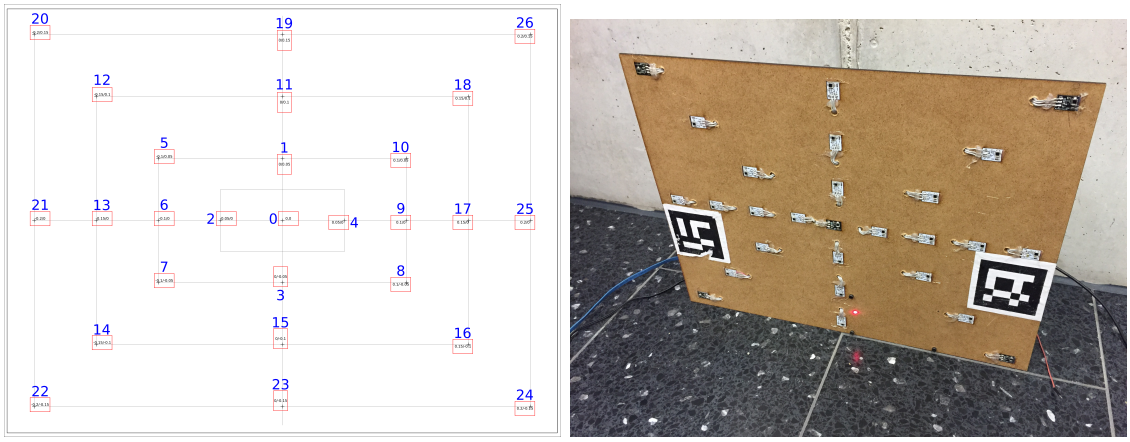
The eccentricity can be compensated using the following equations, where the bearing offsets vary depending on the current bearing angle of a motor:

$$\gamma_\theta = gibmag_\theta * \cos(\theta' + gibphase_\theta) \tag{3.18}$$
$$\gamma_\varphi = gibmag_\varphi * \cos(\varphi' + gibphase_\varphi) \tag{3.19}$$

The tilt of the motors is corrected using the parameters $tilt_\theta$ and $tilt_\varphi$ in Equations 3.1 and 3.2.

The parameters $phase_\theta$, $phase_\varphi$, $tilt_\theta$, $tilt_\varphi$, $curve_\theta$, $curve_\varphi$, $gibmag_\theta$, $gibphase_\theta$, $gibmag_\varphi$ and $gibphase_\varphi$ are different for every lighthouse and have to be estimated.

Two strategies were implemented for estimating these calibration parameters. The first strategy involved the creation of a calibration object with many sensors, where all sensors lie in a common plane. Figure 3.8a shows the sensor placement with dimensions in meter. Figure 3.8b shows the manufactured calibration object made from medium-density fibreboard (MDF). The sensor slots where laser cut. The calibration object was then placed at a known distance relative to a lighthouse using a laser distance ruler. The calibration object and the lighthouse were facing each other. Using the seams on the floor, we were trying to position them as parallel as possible with respect to each other. This way the true sensor positions and therefore the true bearing angles were known. A non-linear optimizer was used to estimate the calibration parameters for both lighthouses, such that the pseudo bearing angles were approaching the true bearing angles in the least squares sense.



(a) sensor placement (b) laser cut from MDF

Figure 3.8: lighthouse calibration object

Instead of trying to position the calibration object relative to a lighthouse as good as possible, the idea behind the second strategy was to collect an extended set of sensor values, where the calibration object was randomly moved in front of a lighthouse. We were then using MLPE to estimate the relative pose of the calibration object to our lighthouse for every recorded frame. The pseudo bearing angles were used for the pose estimation. Using the estimated poses, the bearing angles were calculated for every frame, and the calibration values were estimated such that the pseudo bearing angles would match the calculated ones. More sets were recorded, and over time we were hoping, the pseudo bearing angles would approach the calculated angles, and the estimate for the calibration values would become more accurate. This was only implemented in Matlab for simulated sensor data and has not been tested with real sensor data, yet.

Implementation

The lighthouse tracking was implemented as a ROS package in C++. The following lists all classes and their functionalities:

- TrackedObject:
  - Receive sensor values via ROS message or UDP
  - Load calibrated trackedObject configurations
- Transform:
  - implements helper functions for receiving and sending ROS tf frames

  - implements helper functions for converting ROS tf frames to and from transformation matrices or pose vectors

- Triangulation:
  - implements helper functions for triangulation of sensor positions from bearing angles
  - applies lighthouse tilt calibration values
- Sensor:
  - container class for the lighthouse bearing angles, relative sensor location, triangulated 3D position and distance to a lighthouse of a tracked object sensor
  - implements helper functions for determining if a sensor is active, calibrated or has new data
- Utilities:
  - implements helper functions for reading and writing trackedObject information from and to file
  - implements helper functions for reading and writing lighthouse calibration values from and to file
- LighthouseSimulator:
  - publishes simulated sensor values the same way as the FPGA
  - the pose of a simulated object can be changed at run time
  - calibration values can be changed at run time
  - loads the mesh of a trackedObject and checks if the sensor is visible by the simulated lighthouse
- LighthouseEstimator:
  - all functions for lighthouse tracking, such as triangulation and pose estimation strategies
  - applies lighthouse calibration values
- PoseEstimatorSensorCloud:
  - TPE strategy
- PoseEstimatorMultiLighthouse:
  - MLPE strategy
- InYourGibbousPhase:

– lighthouse calibration value estimator

Extensive usage of ROS visualization messages visualizes the tracking status in the ROS visualizer (rviz). Every pose is published as a ROS tf frame. This was very helpful in the development process of the different tracking and pose estimation algorithms. The resulting pose from our lighthouse tracking system is published to another standard ROS package called robot_localization. The TrackedObject class inherits from the extended kalman filter (EKF) class of this package. The results of our tracking system can, therefore, be fused with additional sensory data, such as interial measurement unit (IMU) data in a future project.

Figure 3.9 shows a prototype of the shoulder of Roboy 2.0 with a custom sphere tracker attached to its end-effector. The sphere tracker was designed in Fusion 360 and the relative sensor positions exported with the respective plug-ins. This sphere tracker or any other custom object can be tracked by our system.

3.4.3 DarkRoom GUI

The lighthouse tracking code as described in Section 3.4.2 implements the tracking functionality. For high-level control, it was necessary to design a GUI. The following features where implemented:

- adding/removing tracked objects
- simulate tracked objects and make them move in virtual space
- start/stop tracking algorithms
- monitor information about sensors, such as the current bearing angles and the update frequency
- start/stop calibration, monitor parameters, change parameters for simulated calibration
- display information about lighthouses as decoded from OOTX frames

The GUI was written in C++ using the Qt framework. The DarkRoom GUI is a ROS package as well and a rqt plugin. Rqt is a window manager from the standard ROS installation. Plugins written for rqt can be added and arranged in the rqt GUI.

Figure 3.10 shows the control panel of the DarkRoom GUI. It provides access to the main control of our lighthouse tracking system. The individual elements provide the following functions:

A Convenience function for adding all tracked objects Roboy 2.0 consists of

B Creates a UDP socket for receiving sensor data broadcast messages on the given broadcast IP and port

C lighthouse rays are visualized in rviz if toggled

D distances between sensors are visualized in rviz if toggled

E clears all visualization markers in rviz

F switches lighthouses

G the lighthouse poses are reset to their initial values

H relative pose estimation for a tracked object is used to correct the pose of the second lighthouse

I instead of using our lighthouse poses, use the poses from SteamVR

J toggles recording of pose estimations from our system and the SteamVR system

K aligns the coordinate frame of a tracked object to that of a Vive controller

L toggles recording of lighthouse bearing angles

M toggles pose estimation using TPE

N toggles lighthouse pose correction using relative distance estimates (slow, inaccurate)

O toggles pose estimation of a tracked object using EPnP

P toggles pose estimation using MLPE

Q toggles triangulation of sensor positions (will be automatically activated when using TPE)

R estimates the distance of all visible sensors to each lighthouse individually

S measures the triangulated sensor positions for a couple of seconds. Rejects outliers and generates relative sensor positions which are written to a configuration file

T Any tracked object can be simulated using this panel. When adding a tracked object, tick the simulate box. The object will be moved in space when random pose is ticked. The movements can be accelerated using the slider and constrained using the respective tick boxes.

The rest of the DarkRoom GUI panels are described in Appendix A.

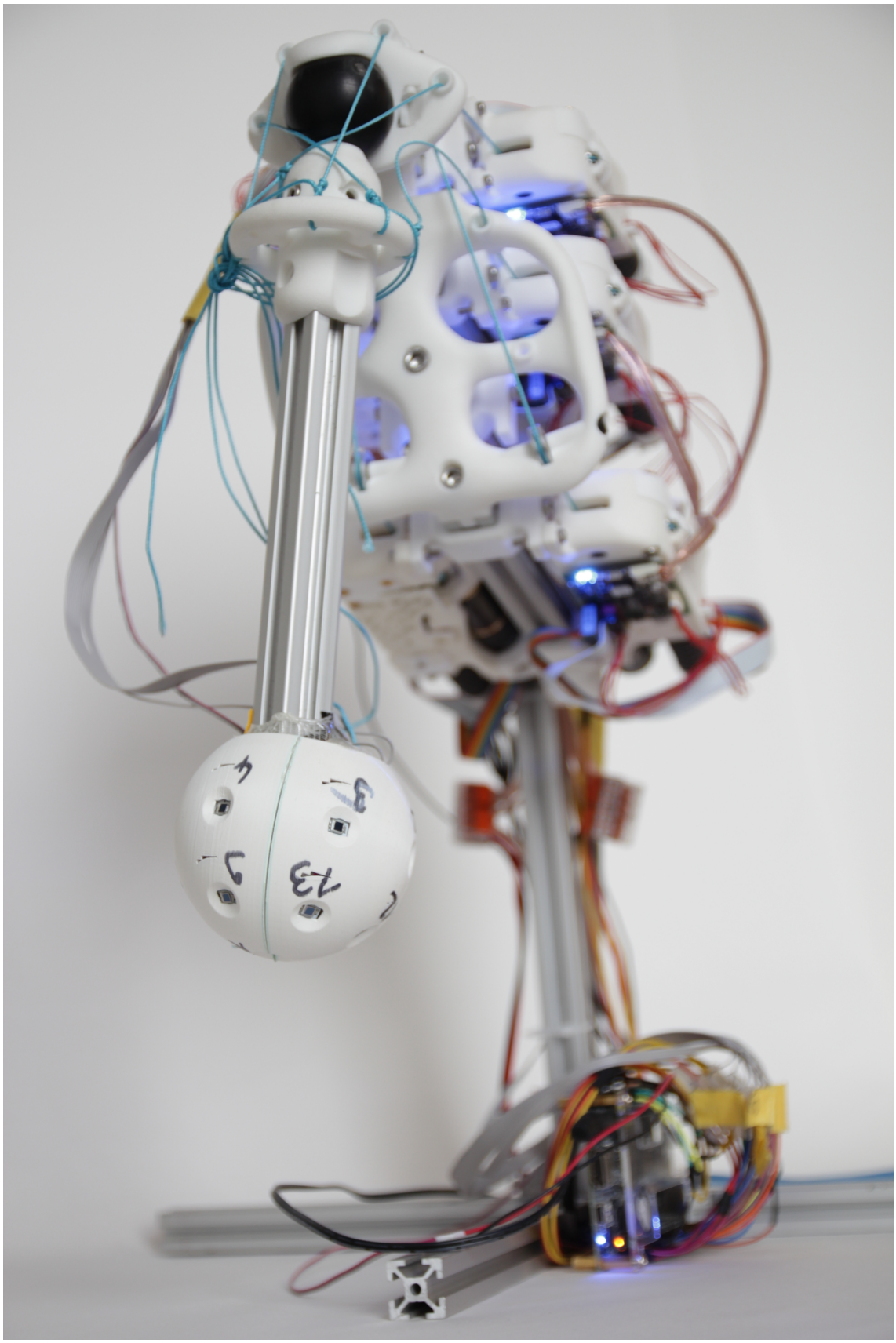


Figure 3.9: A custom sphere tracker attached to a prototype of the shoulder of Roboy 2.0

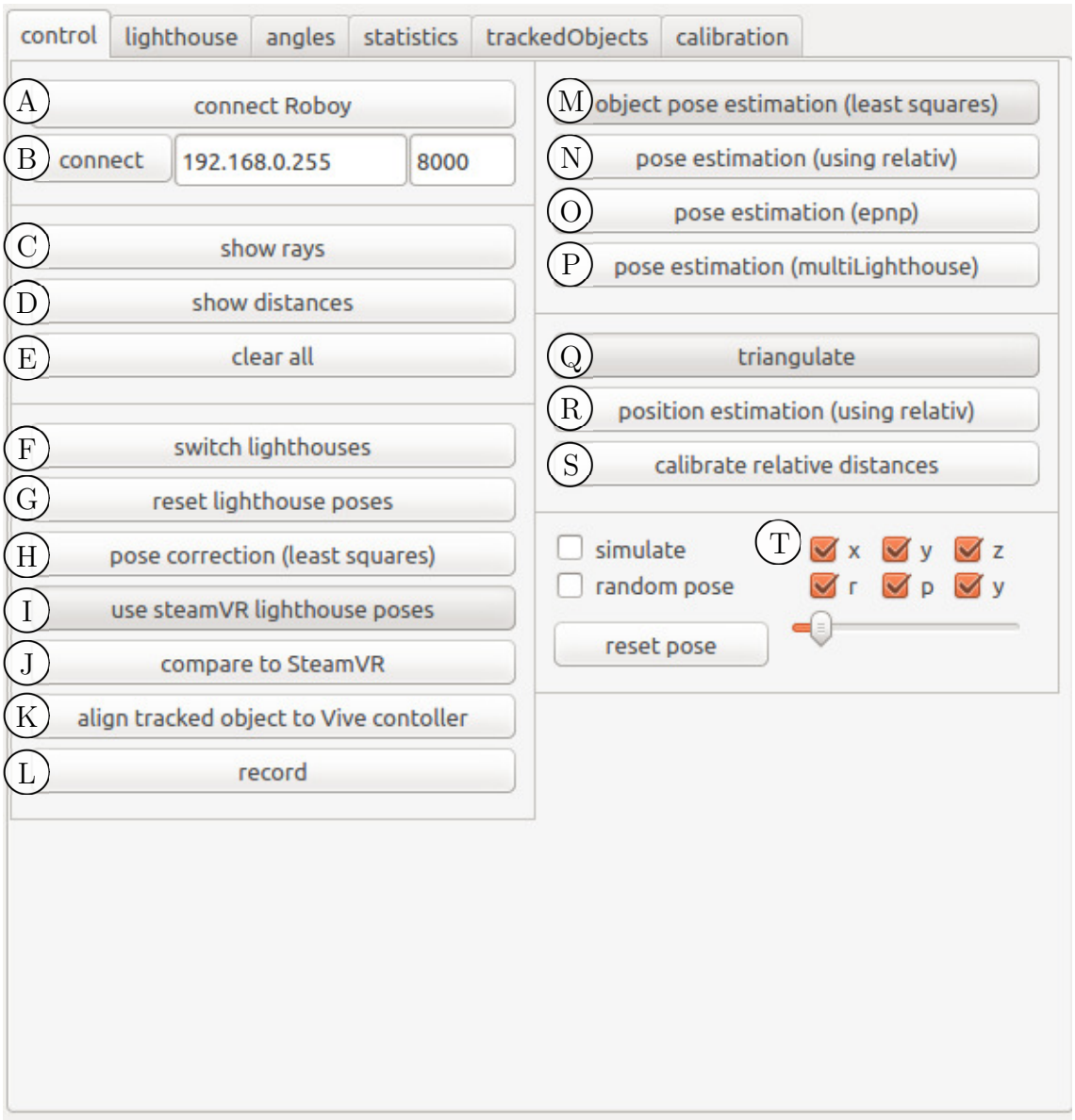


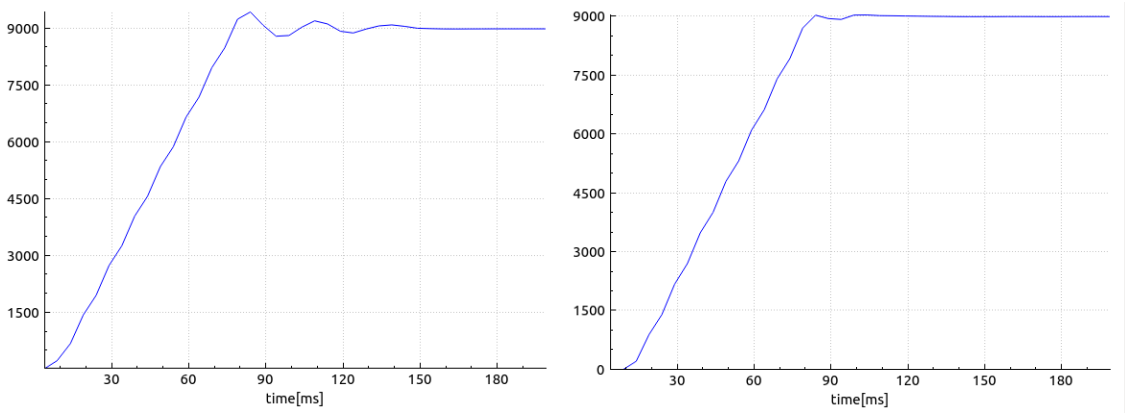
Figure 3.10: Control panel of DarkRoom GUI

Chapter 4

Results

4.1 MyoControl

The PID controller response was evaluated in position control. The only load acting on the motor was the gearbox. The control frequency of MyoControl was reduced to 100 Hz, to make the PID controller response visible. The motor started from position 0 and a set point of 9000 was commanded (this equals approx. 30 degrees). In Figure 4.1a the motor reaches the target position at about 85 ms. The motor overshoots and the secondary oscillations decay until the motor settles at about 150 ms to the target set point. Adding D-gain to the controller can compensate the secondary oscillations. This is shown in Figure 4.1b.



(a) P-gain only

(b) P-gain and D-gain

Figure 4.1: MyoControl PID controller evaluation with different gains, commanding a motor position of 9000 encoder ticks

4.2 Lighthouse Tracking

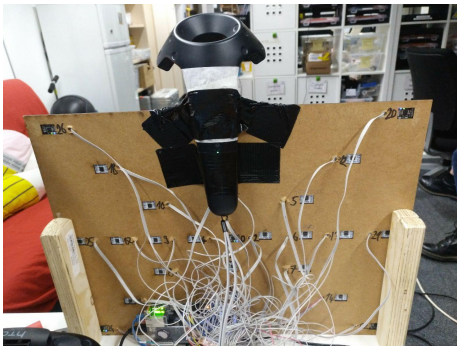


Figure 4.2: calibration object with mounted Vive controller

For the evaluation of our lighthouse tracking system, a direct comparison with the HTC Vive tracking system was chosen. A Vive controller was mounted onto our calibration object, which can be seen in Figure 4.2. The SteamVR system, running on Ubuntu, was used to get the poses of the lighthouses and the Vive controller. For evaluation of the position and orientation errors, the pose of the first sample was used to align our coordinate system with that of the Vive controller. The root mean square error (RMSE) was used to compare the position and angle errors.

Figure 4.3 shows recorded trajectories using our pose estimation algorithms TPE and MLPE with uncalibrated and calibrated lighthouses in a direct comparison with the position estimates from the Vive system. Figure 4.8 shows different views of the trajectories shown in Figure 4.3. The solid lines are the result from our tracking using calibrated lighthouses, while the dashed lines show the position estimates using uncalibrated lighthouses. The position errors between the two results are summarized in Table 4.1.

method	**RMSE [m] x-axis**	**RMSE [m] y-axis**	**RMSE [m] z-axis**
TPE uncalibrated	0.0633	0.1097	0.0326
TPE calibrated	0.0276	0.0446	0.0211
MLPE uncalibrated	0.0850	0.1386	0.0677
MLPE calibrated	0.0402	0.0896	0.0376

Table 4.1: Position errors for trajectories in Figure 4.3

For evaluation of the orientation, the orientation in quaternion representation was converted to euler angles. Figure 4.5 show the euler angles of our system using TPE (dashed line calibrated, dash-dot line uncalibrated) and the Vive system (solid line). The calibration object was rotated around its' x-, y- and z-axis individually. The angle errors are summarized in Table 4.2 using RMSE in [degree].

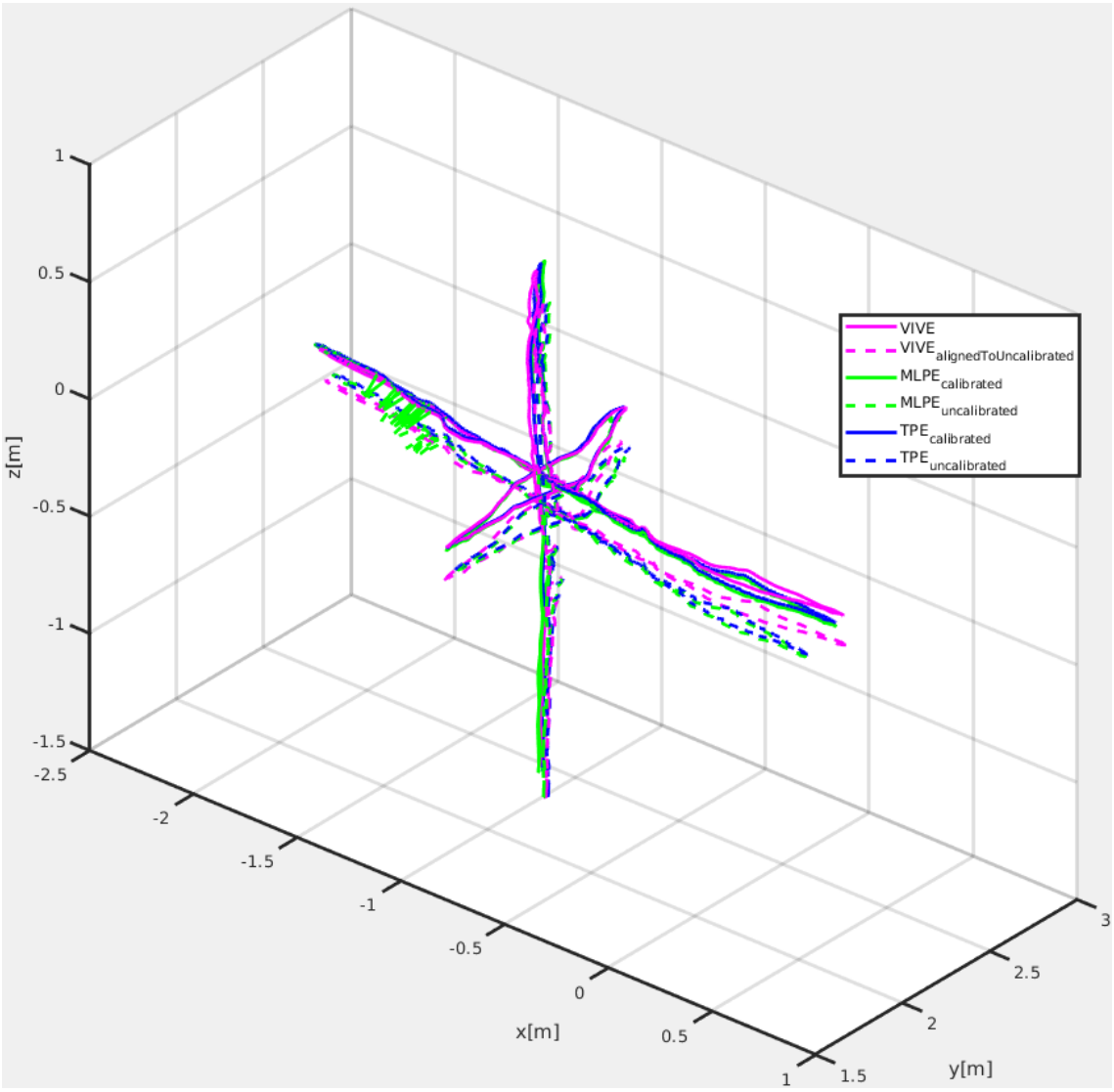
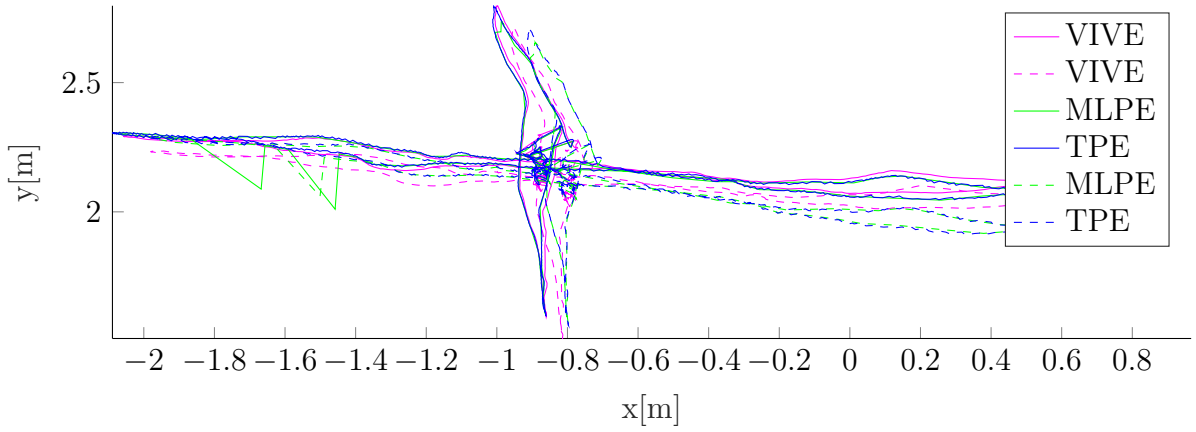


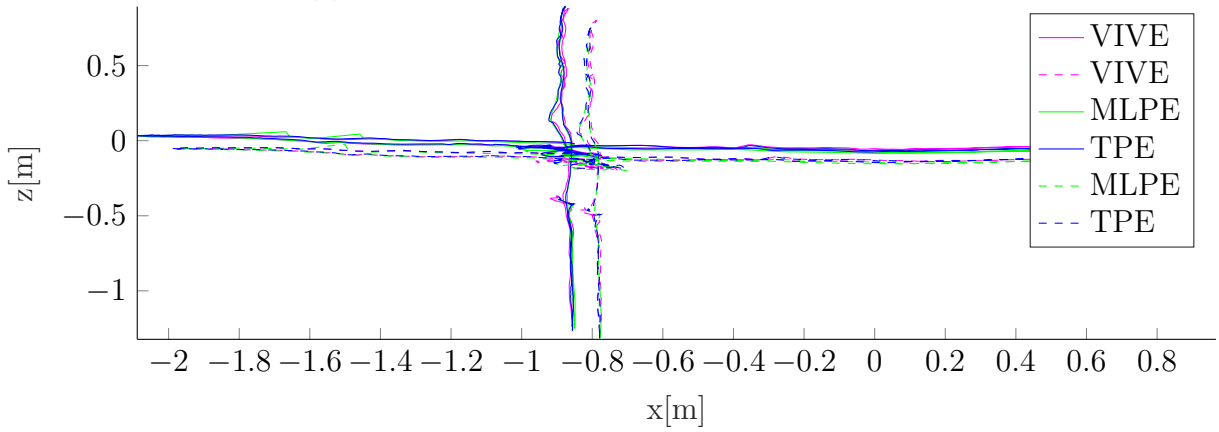
Figure 4.3: Comparison between HTC Vive position tracking (magenta lines) and our position tracking (blue TPE, green MLPE) with uncalibrated (dashed) and calibrated (solid) lighthouses

MLPE was tested in a direct comparison with SteamVR tracking the Vive controller in single lighthouse mode. The same experimental setup as above was applied but using only one lighthouse. Figure 4.7 shows the trajectories for the movement of the calibration object in x-, y- and z-axis. The position errors are summarized in Table 4.3.

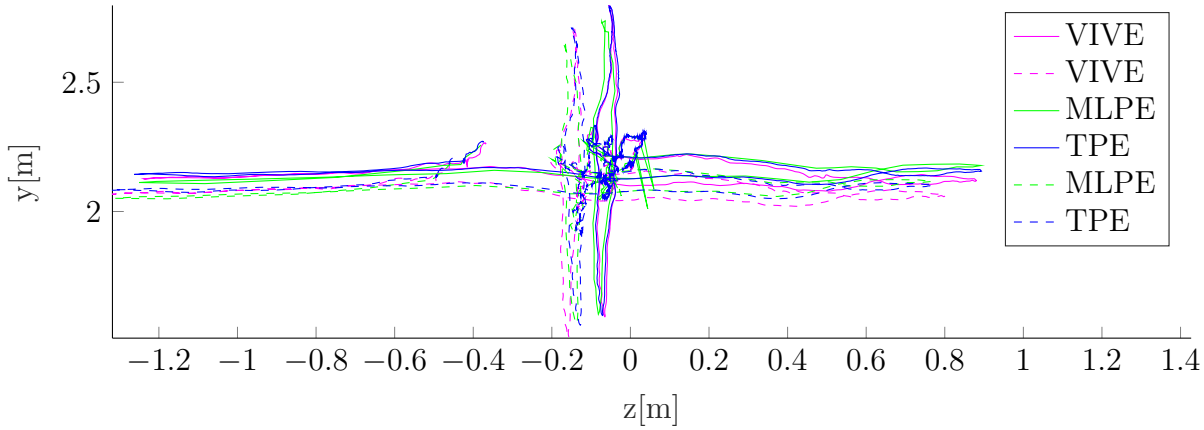
Figure 4.9 shows the orientation of the Vive controller (solid line) and the orientation estimates using MLPE (dash-dot line). The orientation errors are summarized in Table 4.4.



(a) XY-view of trajectories in Figure 4.3



(b) XZ-view of trajectories in Figure 4.3



(c) ZY-view of trajectories in Figure 4.3

Figure 4.4: Comparison between HTC Vive position tracking (magenta lines) and our position tracking (solid line calibrated lighthouses, dashed line uncalibrated lighthouses)

method	rotation	**RMSE pitch**	**RMSE roll**	**RMSE yaw**
TPE uncalibrated	x	1.3324	2.9058	1.2022
MLPE uncalibrated	x	8.1578	3.2872	1.2208
TPE calibrated	x	0.8529	0.2765	0.2358
MLPE calibrated	x	4.8689	0.7650	0.4417
TPE uncalibrated	y	0.3004	3.2919	0.4199
MLPE uncalibrated	y	1.2706	5.4928	0.9116
TPE calibrated	y	0.2609	0.8213	0.2230
MLPE calibrated	y	0.8622	2.3605	0.4162
TPE uncalibrated	z	1.2140	4.1603	0.8138
MLPE uncalibrated	z	1.3045	4.4979	5.1384
TPE calibrated	z	1.1776	1.2260	0.6407
MLPE calibrated	z	1.1575	1.5522	2.5226

Table 4.2: Orientation errors in degree for trajectories in Figure 4.5 and 4.6

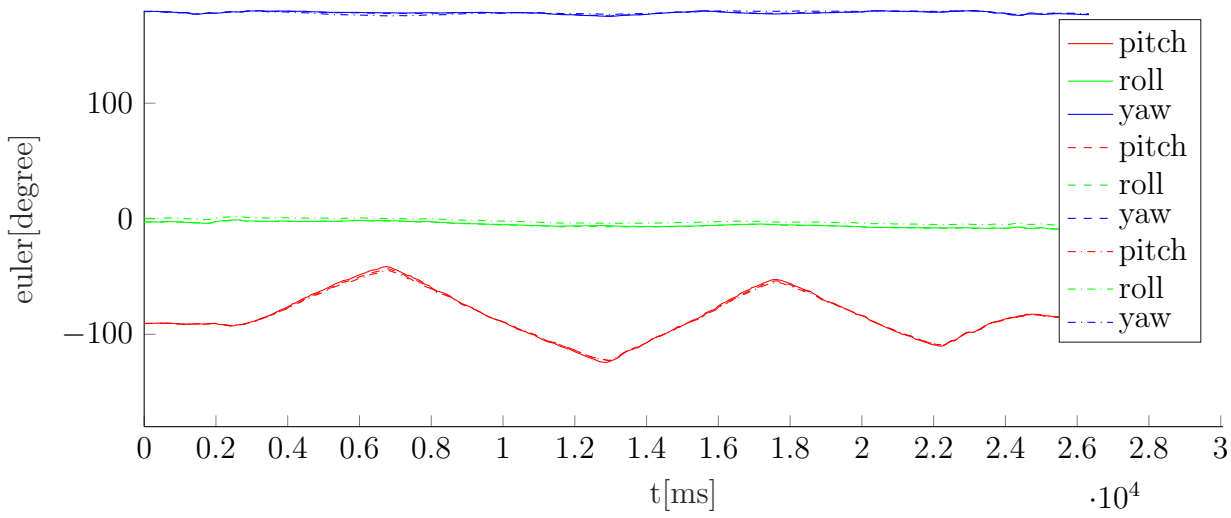
method	**RMSE [m] x-axis**	**RMSE [m] y-axis**	**RMSE [m] z-axis**
MLPE uncalibrated	0.1803	0.5065	0.2585
MLPE calibrated	0.0588	0.1742	0.0249

Table 4.3: Position errors for trajectories in Figure 4.7

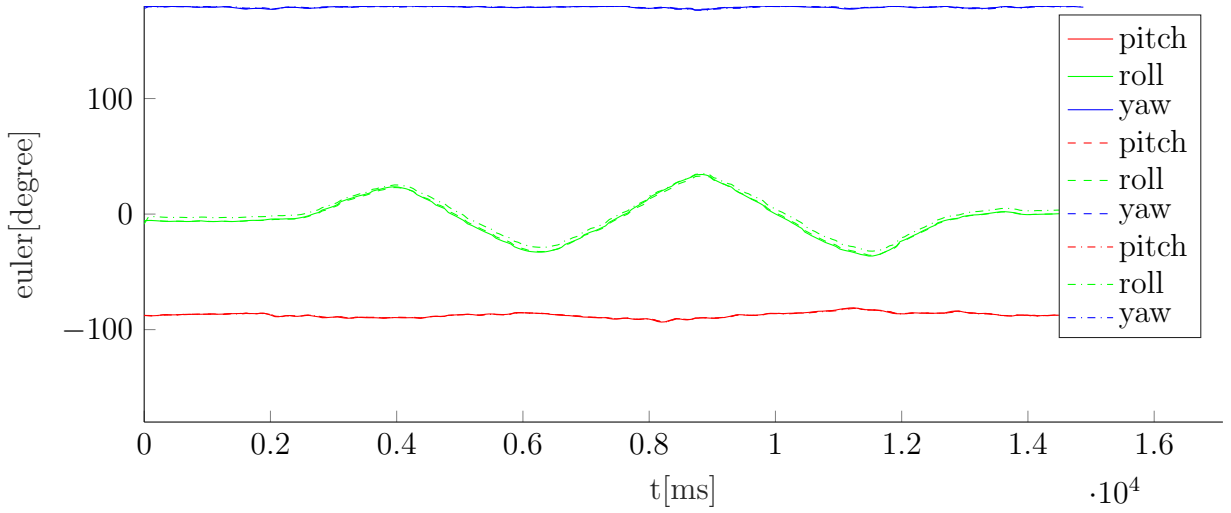
method	rotation	**RMSE pitch**	**RMSE roll**	**RMSE yaw**
MLPE uncalibrated	x	13.6079	4.0370	1.4443
MLPE calibrated	x	13.8810	3.2597	1.0838
MLPE uncalibrated	y	5.5131	33.4763	2.2119
MLPE calibrated	y	5.6269	34.2024	2.5119
MLPE uncalibrated	z	8.0032	10.1888	11.1724
MLPE calibrated	z	9.0688	15.0548	2.9446

Table 4.4: Orientation errors in degree for trajectories in Figure 4.9

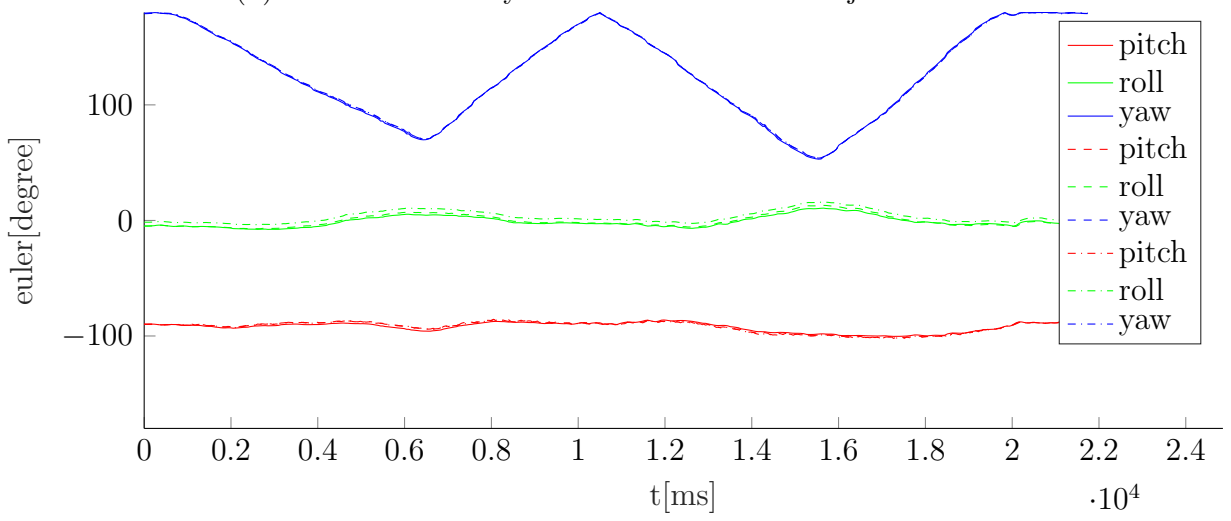
The position tracking for fast movements was analyzed for each axis. The calibration object was moved along its' x-, y- and z-axis in separate recordings. The recorded positions are shown in Figure 4.10. The nominal velocity of the Vive controller together with the RMSE between our position estimates and the Vive positions are shown in Fig. 4.11-4.13.



(a) Rotation around x-axis of the calibration object

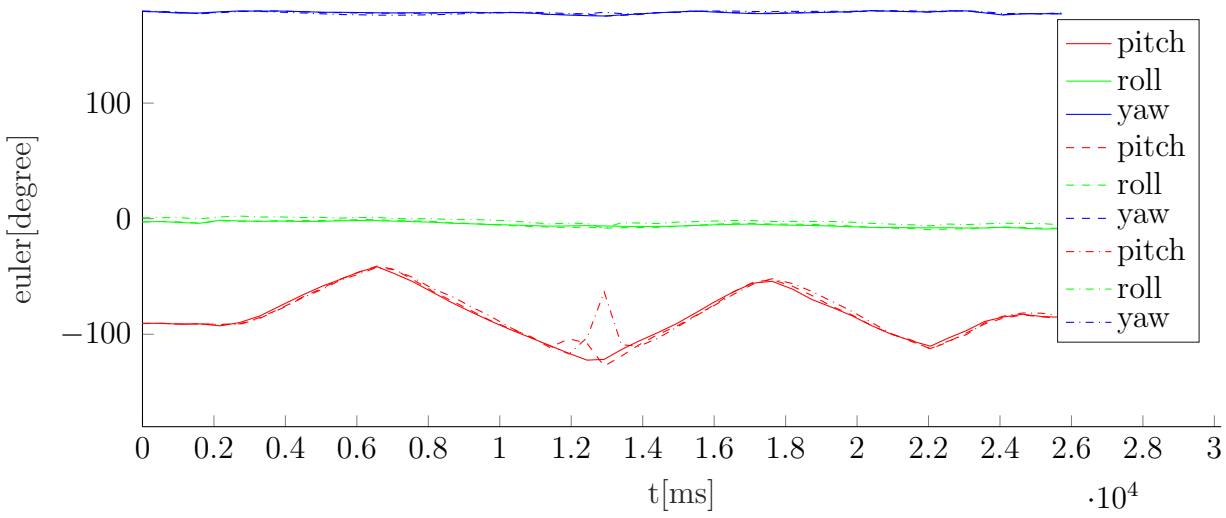


(b) Rotation around y-axis of the calibration object

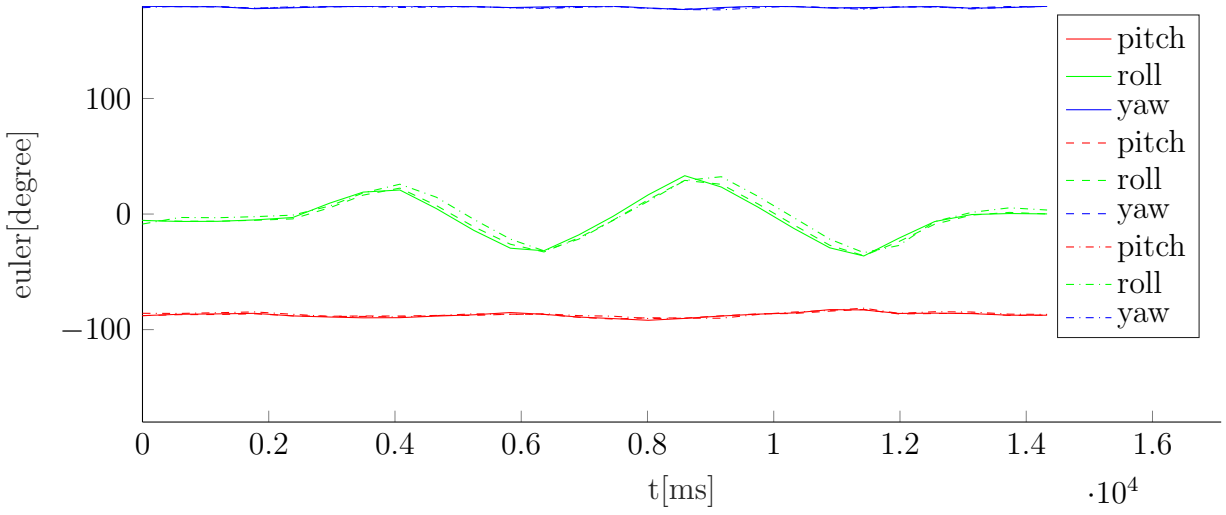


(c) Rotation around z-axis of the calibration object

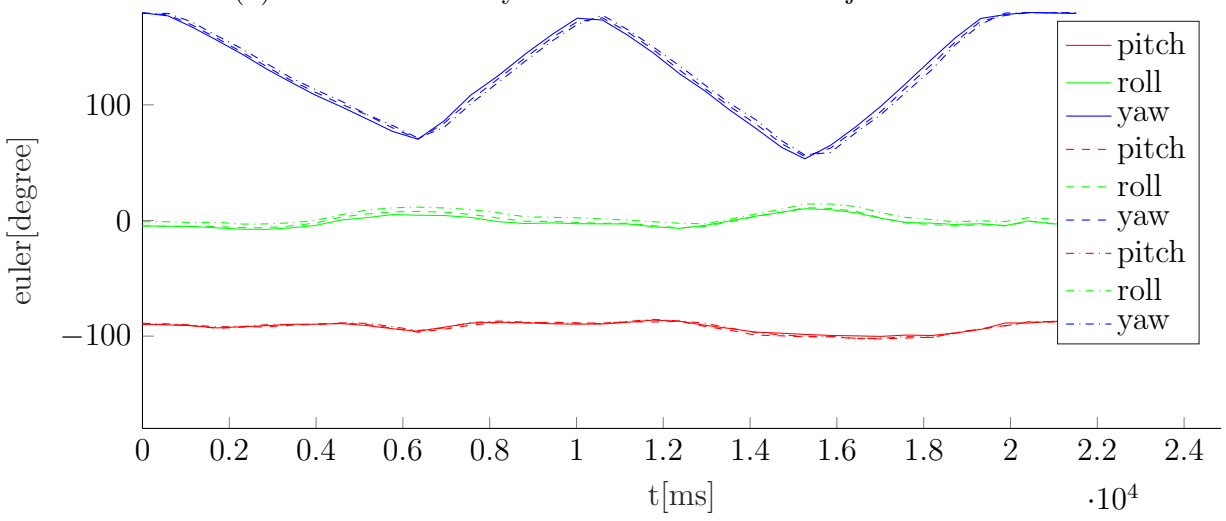
Figure 4.5: Orientation comparison between our system using TPE (dashed line calibrated, dashed-dot uncalibrated) and the Vive system (solid line)



(a) Rotation around x-axis of the calibration object



(b) Rotation around y-axis of the calibration object



(c) Rotation around z-axis of the calibration object

Figure 4.6: Orientation comparison between our system using MLPE (dashed line calibrated, dashed-dot uncalibrated) and the Vive system (solid line)

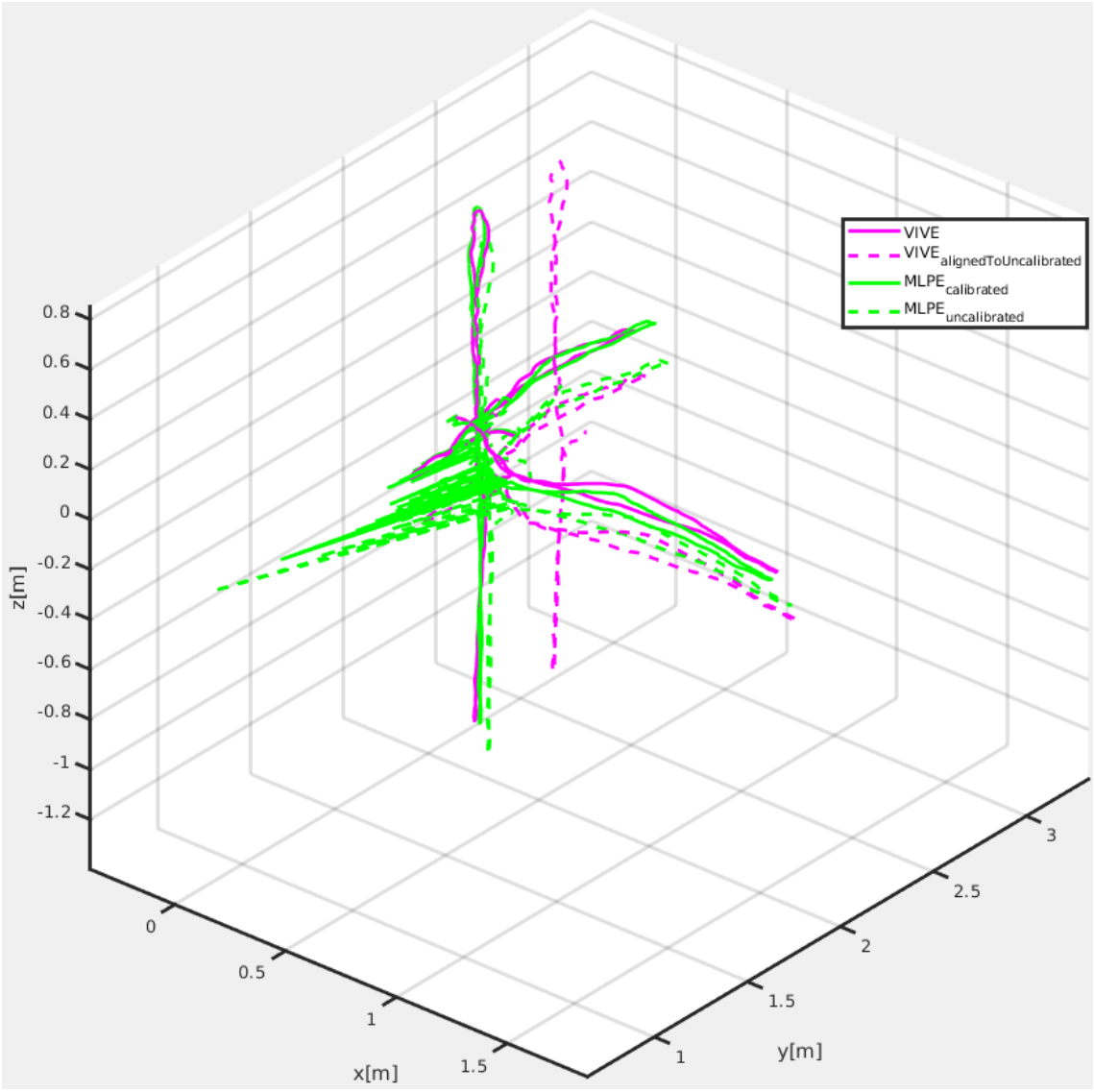
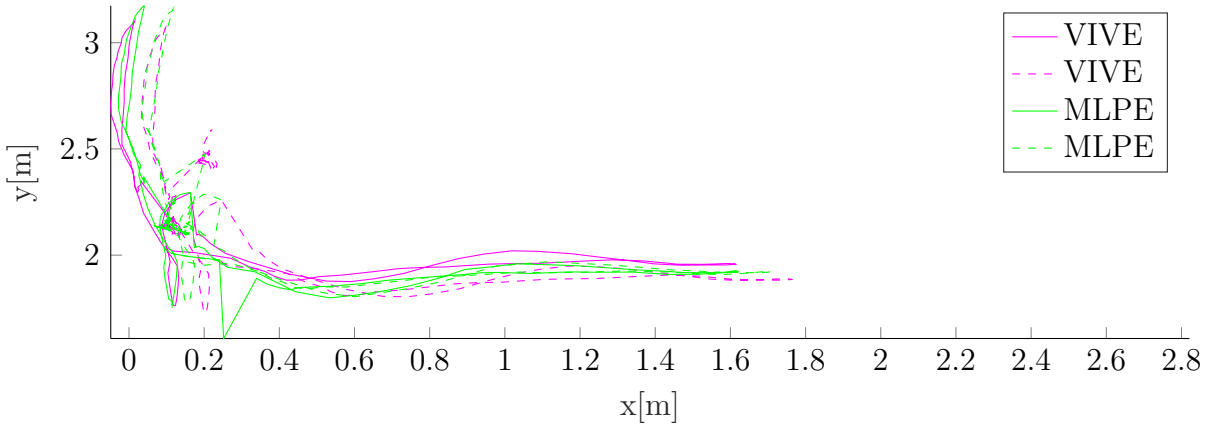
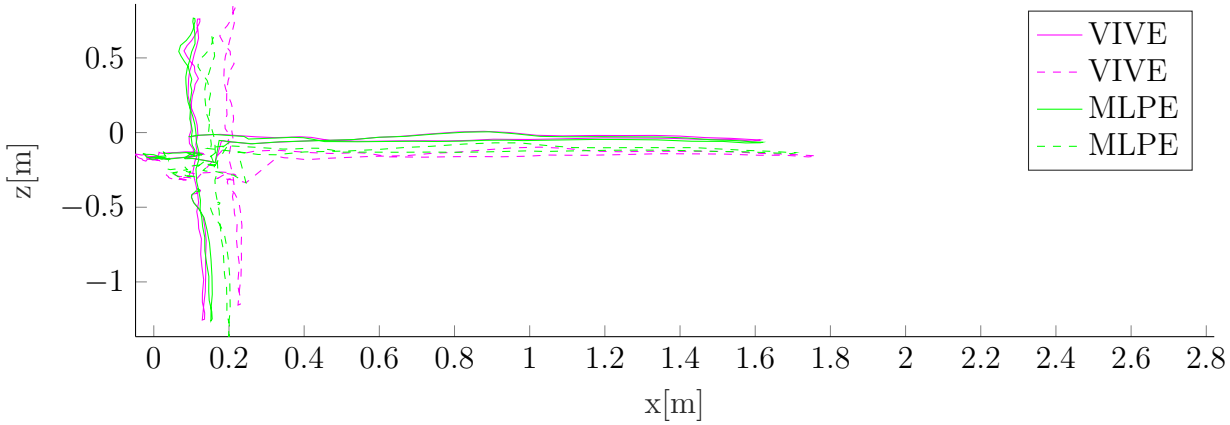


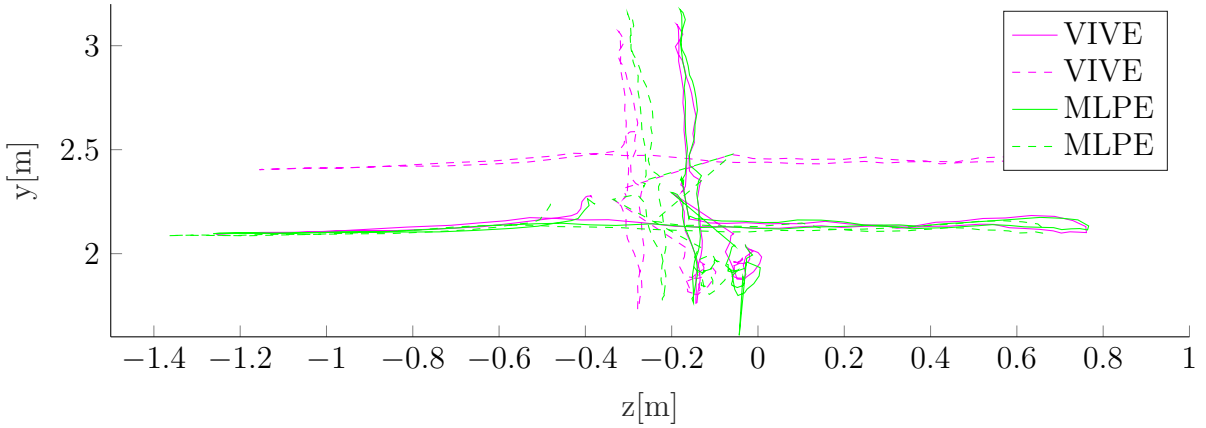
Figure 4.7: Direct comparison between HTC Vive position tracking (dashed magenta line) and our position tracking (solid line, green MLPE) with a **single calibrated** lighthouse



(a) XY-view of trajectories in Figure 4.7

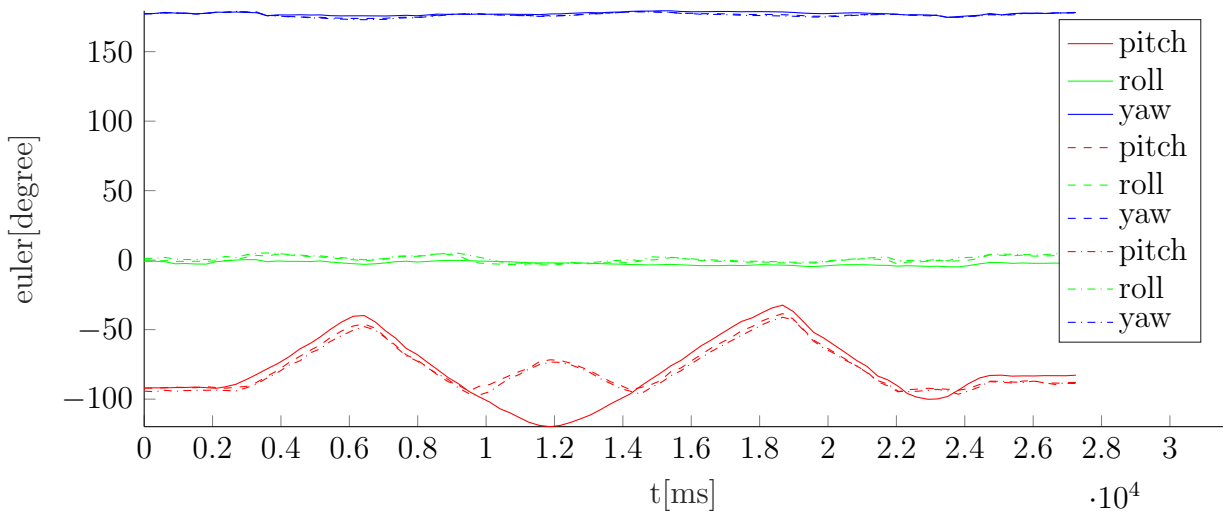


(b) XZ-view of trajectories in Figure 4.7

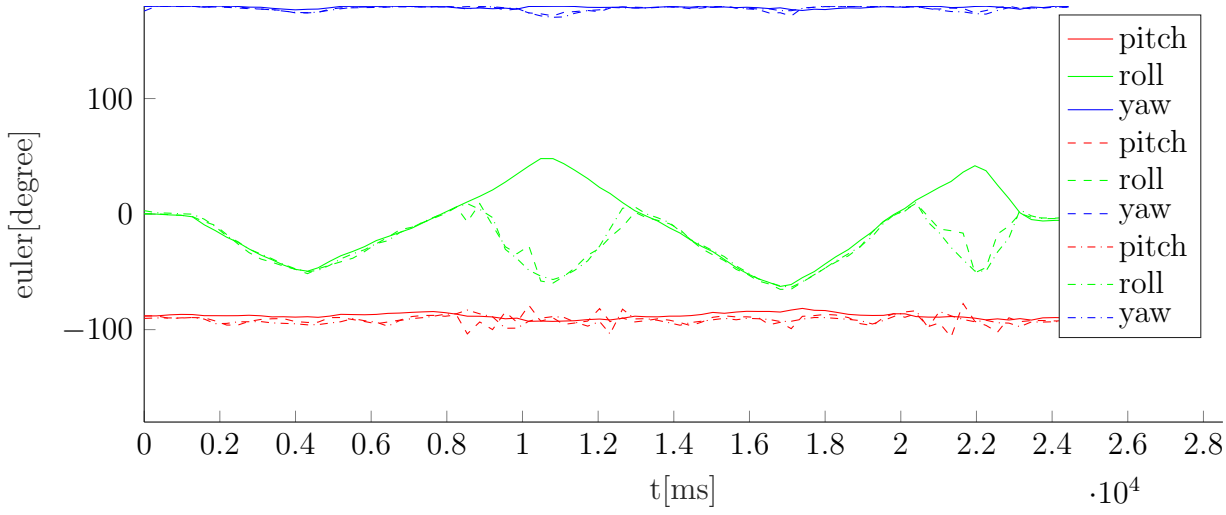


(c) ZY-view of trajectories in Figure 4.7

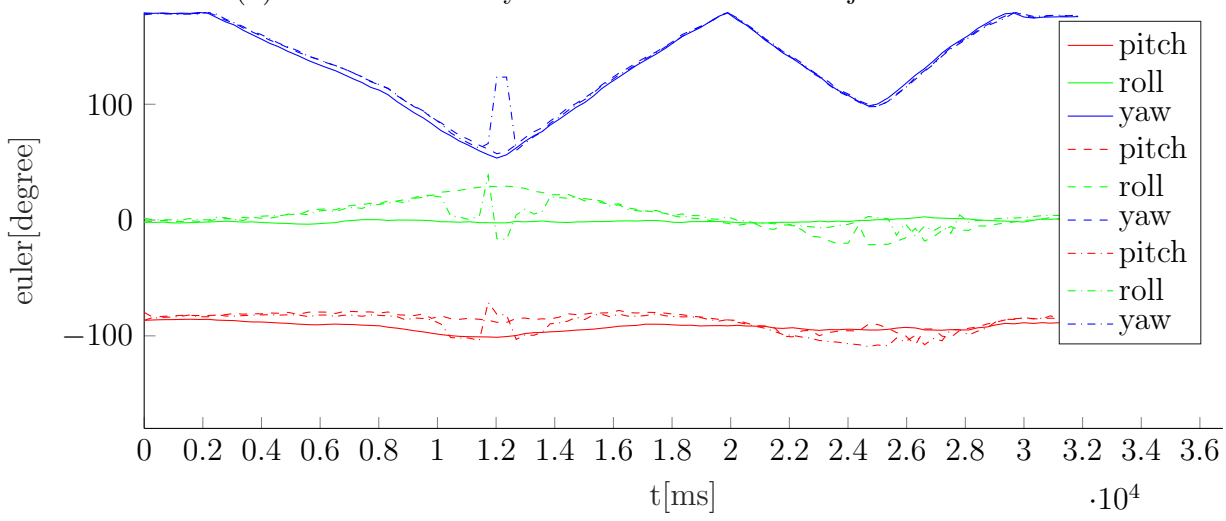
Figure 4.8: Comparison between HTC Vive position tracking (magenta lines) and our position tracking (solid line calibrated lighthouses, dashed line uncalibrated lighthouses)



(a) Rotation around x-axis of the calibration object

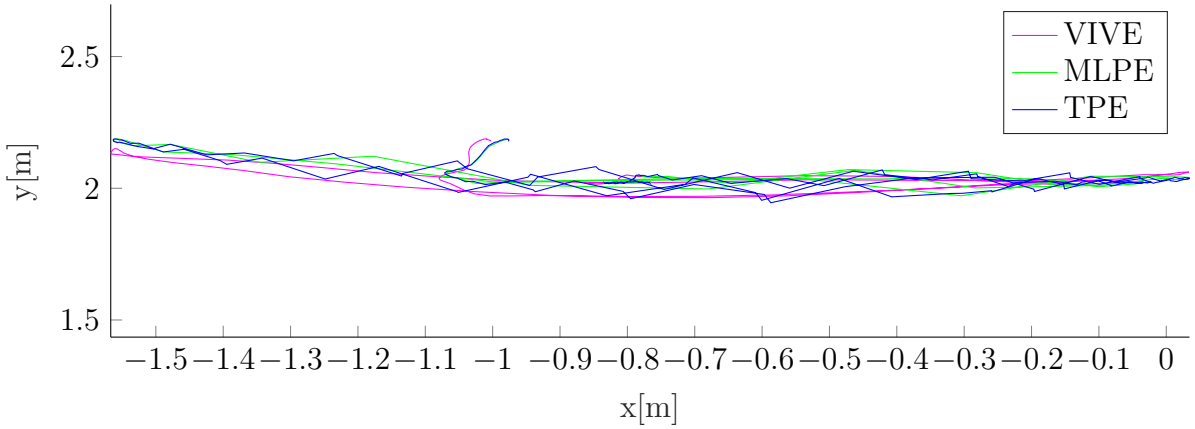


(b) Rotation around y-axis of the calibration object

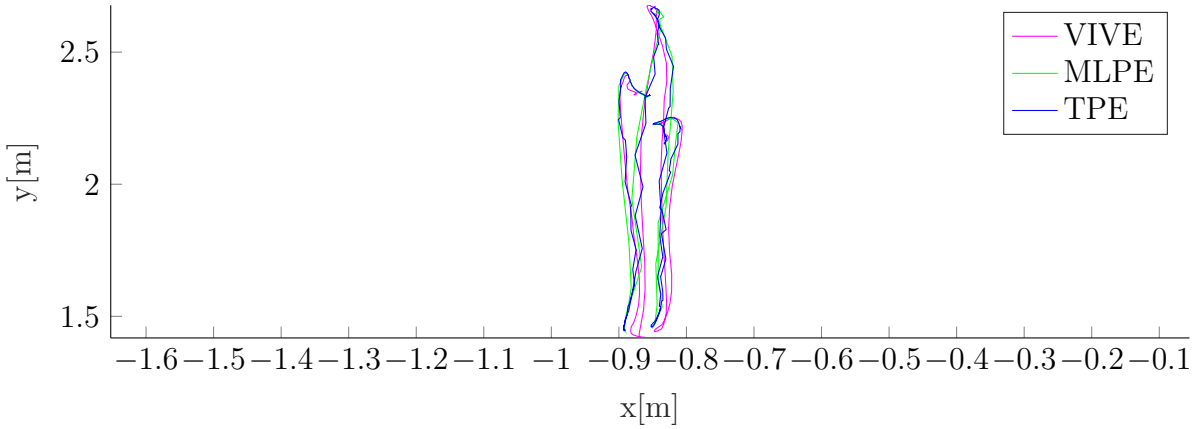


(c) Rotation around z-axis of the calibration object

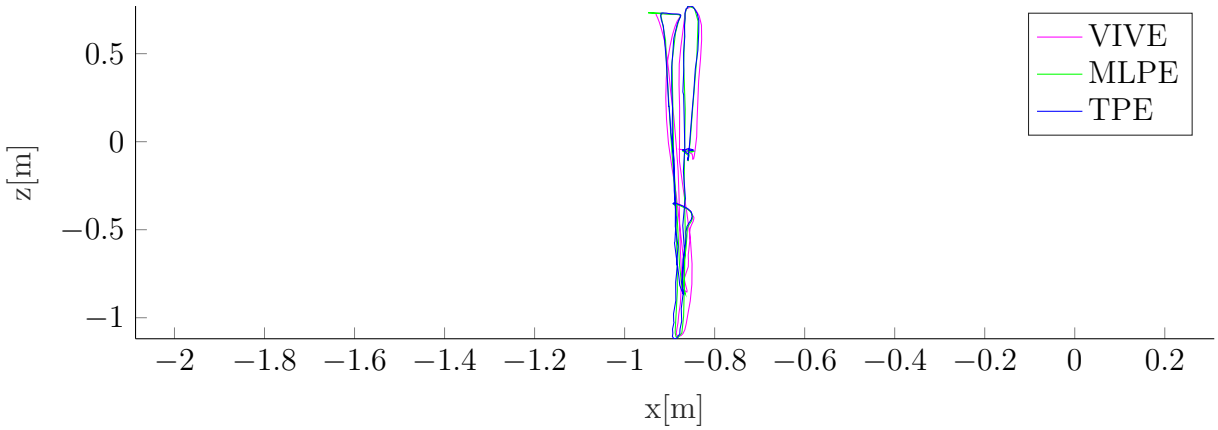
Figure 4.9: Orientation comparison between our system using MLPE (dash-dot line) using a **single calibrated** lighthouse and the Vive system (solid line)



(a) fast movements along x-axis of the calibration object

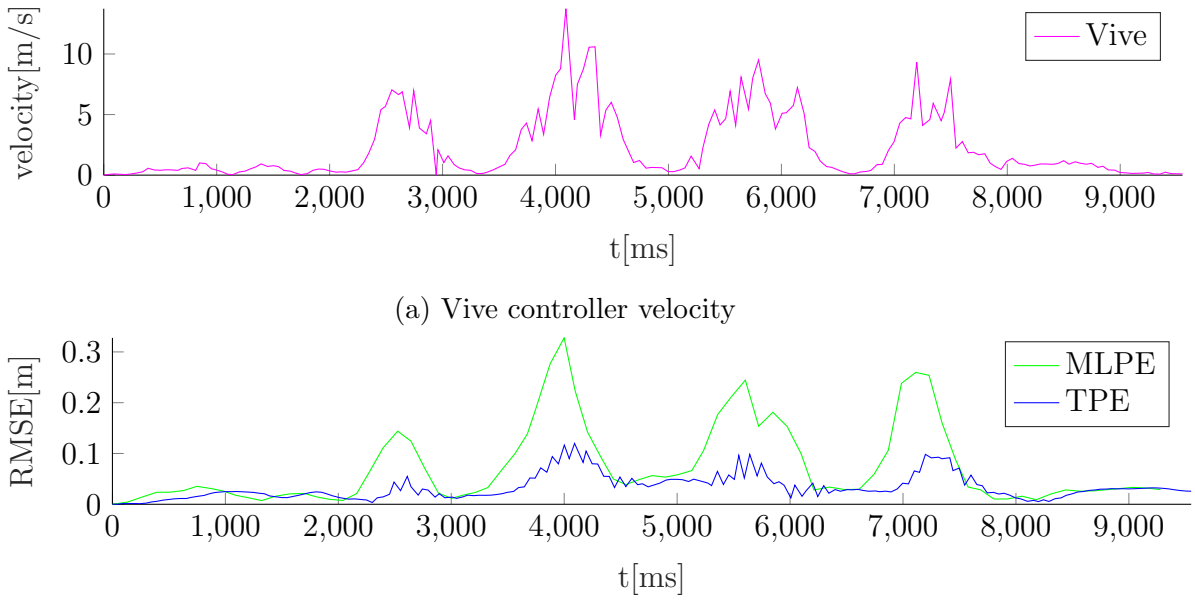


(b) fast movements along y-axis of the calibration object



(c) fast movements along z-axis of the calibration object

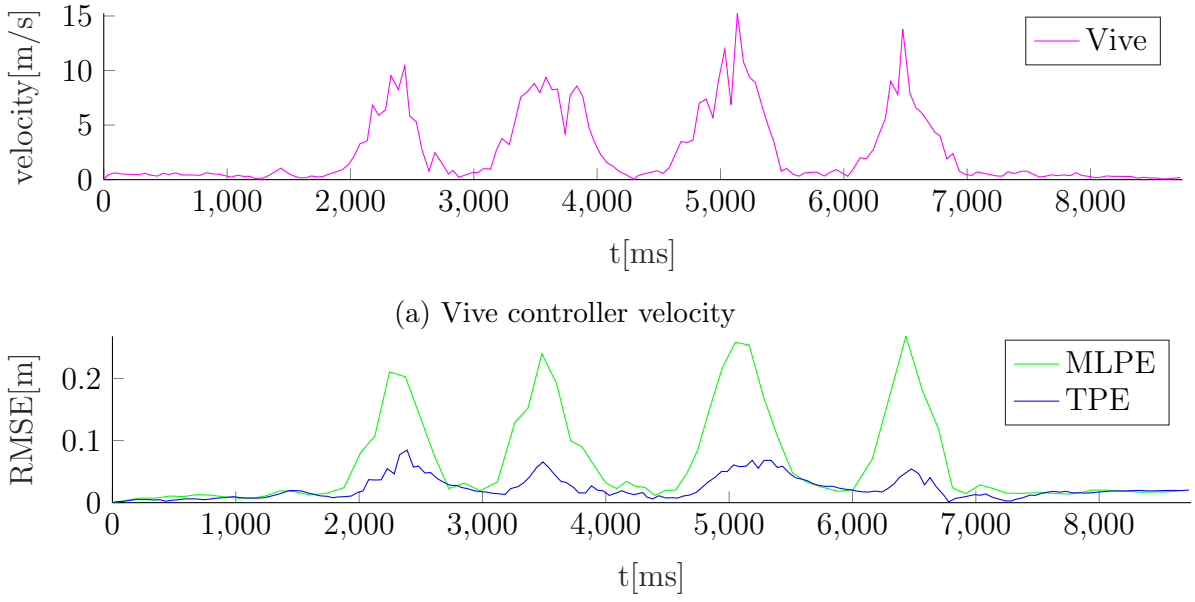
Figure 4.10: Position comparison between our system (TPE blue line, MLPE green line) using a **single calibrated** lighthouse and the Vive system (magenta line)



(a) Vive controller velocity

(b) position RMSE

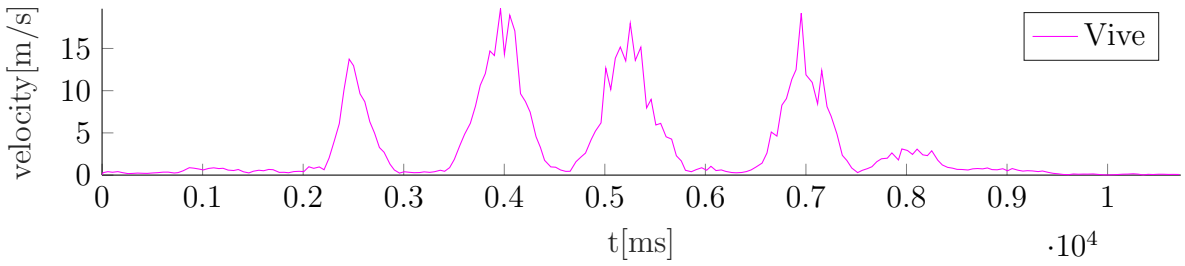
Figure 4.11: Vive controller nominal velocity (top), position RMSE (bottom) between the Vive positions and our system for x-trajectory in Figure 4.10a



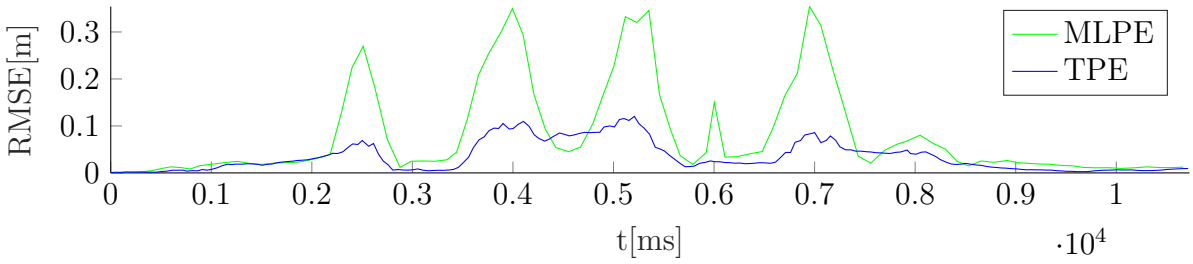
(a) Vive controller velocity

(b) position RMSE

Figure 4.12: Vive controller nominal velocity (top), position RMSE (bottom) between the Vive positions and our system for y-trajectory in Figure 4.10b



(a) Vive controller velocity



(b) position RMSE

Figure 4.13: Vive controller nominal velocity (top), position RMSE (bottom) between the Vive positions and our system for z-trajectory in Figure 4.10c

Chapter 5

Discussion

5.1 openPOWERLINK

The communication in general was functional. However, it was only possible when the CN in the openPOWERLINK network was run from a standard Linux PC. When the FPGAs where connected without the Linux PC, the CN would not respond to the initialization from the MN running on another FPGA. The network could therefor not be tested to its' full capabilities, because a slow CN running on the Linux PC was setting the lowest limit for the cycle time. The communication was stable for a cycle time of 20 ms.

5.2 Lighthouse tracking

Our custom calibration failed to find suitable calibration parameters that would be valid outside the small FOV that was covered by the calibration object. For an accurate estimation of calibration parameters that would also be valid outside the FOV covered by the calibration object, the FOV was not big enough.

As it turned out, the calibration functions as described in Section 3.4.2 were probably very close to the ones used in the Vive system. The direct comparison between our tracking system and the HTC Vive system was crucial for testing our hypothesis on the calibration functions. Although the presented functions correct the pseudo bearing angles to some extent, the HTC Vive system probably uses some functions that are close but not identical with our correction functions. One significant advantage using the Vive calibration parameters is the ability to directly use any lighthouse from HTC Vive without the necessity to calibrate, because our system decodes the OOTX frame transmitted by each lighthouse.

The results presented in Figure 4.3 clearly show how important it is to use calibrated lighthouses. All axis show significant differences between the original Vive position tracking and our system when using uncalibrated bearing angles, which can be examined in Figure 4.8a-4.8c. This is also reflected in the position errors in Table 4.1. MLPE seems a lot more sensitive towards outliers than TPE, which can be seen in the jerky position estimates in Figure 4.8a and 4.8c. The reason why TPE suffers less from this, is the fact that TPE uses triangulated sensor positions. The lighthouse rays, therefore, have to be valid for both lighthouses which implicitly filters the sensor signals.

The sensitivity of MLPE towards sensor signal outliers is even more apparent in the orientation comparisons in Figure 4.6a. At around 13 seconds the pitch estimate of MLPE jumps because of wrong measurements from some of the sensors. Wrong measurements appear for the following reasons:

1. The infrared light intensity is not sufficient, for example when the sensor is too far away from a lighthouse (for our custom sensors above 4 meters)

2. The infrared light intensity is too big, for example when the sensor is too close to a lighthouse (the minimal distance for our custom sensors was about 1.5 meters)

3. The angle of the sweep plane incidence is too steep on the photo-diode. This can be avoided if the sensors are sank into the object, such that the maximal incidence angle is cut off by the physical barrier.

The overall orientation estimates for both algorithms of our tracking system is close to the original Vive system, even when using uncalibrated lighthouses, which is also reflected in Table 4.2.

Using the calibration values, as transmitted by each lighthouse in the OOTX frame, in the calibration functions proposed in Section 3.4.2 and 3.4.2, clearly improves the position estimates for the trajectories in Figure 4.3. Especially the position errors for the y-axis is reduced, which is reflected in Table 4.1. In fact, all position and orientation estimates improve when using the proposed calibration functions together with the HTC Vive calibration values.

In a direct comparison between MLPE and the Vive system using only a single lighthouse, MLPE suffers massively from sensor outliers. In single lighthouse mode it seems even more crucial to use calibrated lighthouses, which can be seen in Table 4.3. The position errors for the calibrated lighthouses are relatively low, except for the y-axis. This is probably related to the calibration, but might also be affected by inaccuracies of the sensor placement on the calibration object. Small errors in the relative sensor placement jeopardize the distance estimate in the y-axis to a lighthouse. Analogously to a monocular camera, where the distance of an object of known size can be estimated using the focal length of the camera. Small errors in

the measured size of the object affect the distance estimate here as well. One idea to circumvent this problem would be to calibrate the relative sensor positions on the tracked object as well.

The orientation estimates of MLPE in a direct comparison with the Vive system as depicted in Figure 4.9 exhibit enormous errors. This is related to the symmetry of the calibration object for which the sensors were arranged in a plane. For the pose estimation in MLPE there are ambiguous solutions to certain orientations of the calibration object. This can be seen in Figure 4.9b from 9 to 13 and 21 to 23 seconds where the roll angle estimates suddenly jump to negated values.

Our tracking system was directly compared to the HTC Vive system for fast movements of the calibration object in its x-, y- and z-axis in separate experiments. The resulting position estimates are shown in Figure 4.10. The position estimation error was directly related to the velocity of the calibration object, which can be observed in Figure 4.11-4.13. The reason for the relationship between the position error and the nominal velocity of the tracking object lies in the sensor data processing. As described in Section 3.6 the two lighthouses scan the room sequentially with their horizontal and vertical motors. The sensor measurements as decoded by our system are therefore in general from different moments in time. The effect of the sensor measurements being misaligned in time becomes apparent when the tracked object is moved and is directly dependent on its velocity. For the x-axis the errors were particularly big because the two lighthouses were standing on the same level parallel to each other and facing the calibration object. Therefore the x-axis was aligned with the x-axis of both lighthouses and the errors introduced by the interleaved sensor measurements were maximized.

Chapter 6

Conclusion

This thesis describes the development of next level hardware control for Roboy 2.0. It was implemented for FPGAs, which makes it reliable and fast. The expensive MyoGanglion was replaced by off the shelves available FPGA development boards. An arbitrary number of motors can now be connected to one common SPI bus, as opposed to the individual SPI connections to a maximum of four motor boards connected to one MyoGanglion. The control frequency for six connected motors could be increased to approx. 3.2kHz, which means it is faster than the MyoToolKit MyoGanglion which controls four motors at 2.5kHz. The implemented MyoControl modules can be run in parallel on the FPGA, which means more SPI buses can be synthesized, and all MyoControl modules control the connected motors at the same frequency. The controllers have proven to be very reliable. The reason for this is two fold. On one hand the FPGA portion is running independent from the ARM cores, such that even on system failure of the program running in the ARM cores, the muscle units are still under control by the FPGA. The other reason is that the communication with the motor boards is independent from the next stage communication (Ethernet). This was not the case for the legacy MyoGanglion, where a muscle unit could not be hot plugged into the system without FlexRay to be restarted. Therefore, muscle units can be added and removed from the new system without having to restart the whole communication network.

This thesis describes how an affordable and off-the-shelf available FPGA development board can be used to run the high-performance hard real-time communication network openPOWERLINK. OpenPOWERLINK as a more reliable and faster communication bus for Roboy 2.0 was introduced. The functionality of the network was proven. Unfortunately, a bug in software/hardware causing the CN to refuse initialization could not be fixed by the end of this thesis. Therefore the performance evaluation of the network needs to be postponed. We are confident that the close cooperation with B&R will alleviate this cavity soon and Roboy 2.0 shall be run with openPOWERLINK. However, in the course of the development some disad-

vantages of openPOWERLINK for application on Roboy 2.0 became apparent. The first disadvantage is the fact, that the on-board RGMII Ethernet transceiver of the DE10-Nano-SoC could not be used for the communication. An external Ethernet transceiver had to be connected to the GPIOs of the FPGA. The on-board Ethernet transceiver could be used for heavy-load data of Roboy 2.0, such as audio and video streaming. Consequently, a separate Ethernet line would be necessary. But because of size constraints this would be undesirable. Recent developments in automation industries have evolved a new extension for hard real-time Ethernet called time sensitive networks. Here a specialized switch controls the transmission of prioritized Ethernet packets. The first prototypes are available now, which enable hard real-time communication up to 10 kHz using giga-bit Ethernet with parallel streaming of media content. The time sensitive network could be an alternative for a future design of the communication infrastructure.

This thesis also shows how the virtual reality tracking system from the HTC Vive can be used for pose estimation of arbitrary objects. The original HTC Vive tracking was reverse engineered and a complete pipe-line was designed with the following features:

- CAD plugins allowing the export of arbitrary objects from Fusion 360, which can be imported into the tracking system
- FPGA modules written in VHDL and Verilog decoding the HTC Vive sensor protocols
- a pose estimation system with two different pose estimation algorithms was implemented, for estimating the 6-DOF pose of an arbitrary object visible by two or a single lighthouse
- many objects can be tracked simultaneously with our system

The tracking system was compared directly to the original HTC Vive system. Position errors down to two centimeters were achieved and angle errors below one degree. A more accurate calibration of the system will reduce these errors further. The tracking system also features simulation of lighthouses and extensively visualizes the tracking in rviz. This simplifies the development of future tracking algorithms.

The implemented pose estimation algorithms show good performance for static and slow motions of a tracked object. For faster movements the sequential nature of the incoming lighthouse bearing angles needs to be dealt with in a future project. The pose estimates from our system can be fused with IMU data, which will further improve the tracking performance.

Appendices

Appendix A

DarkRoom GUI panels

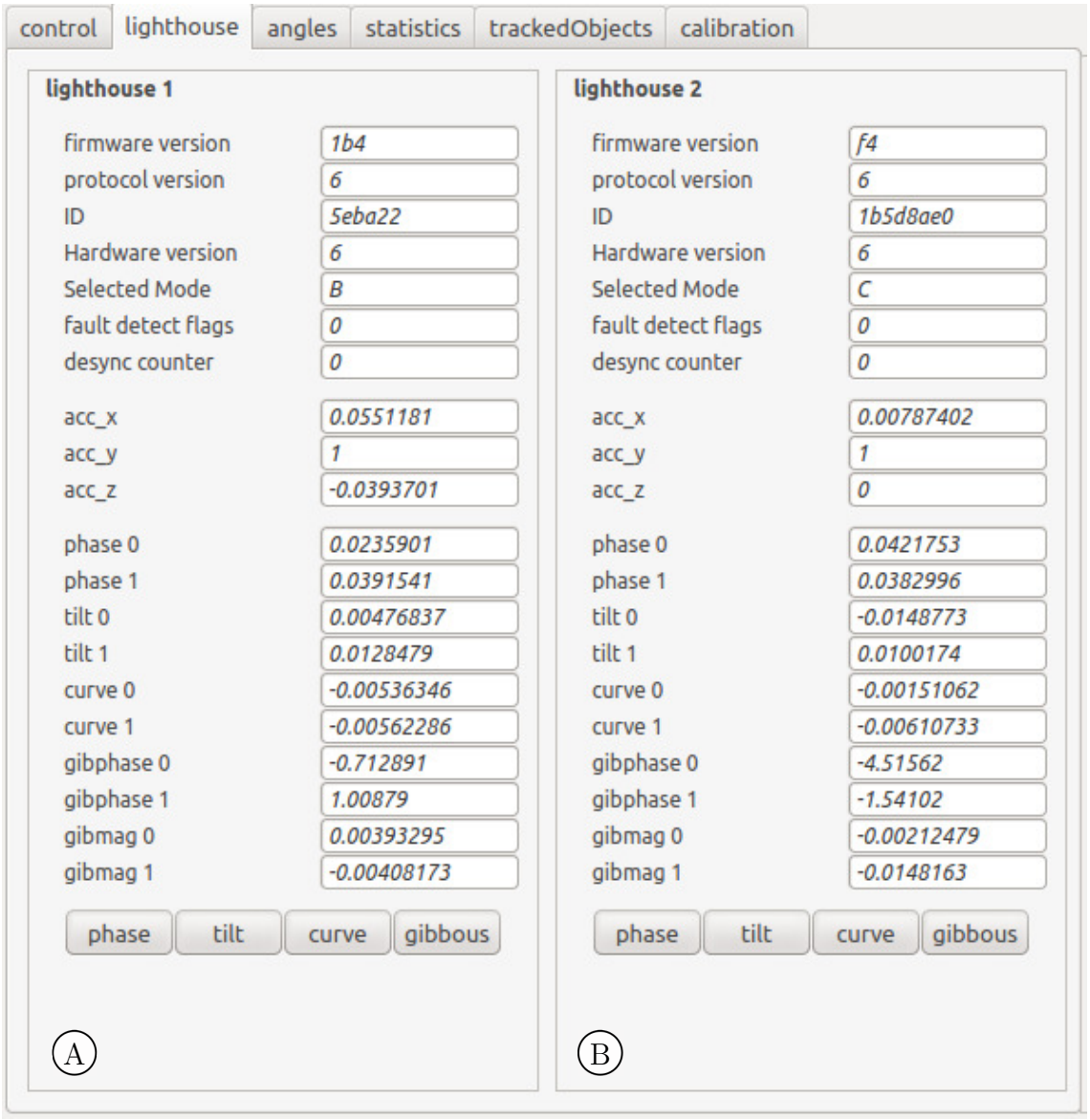


Figure A.1: Lighthouse panel of DarkRoom GUI displaying the decoded OOTX values. The application of different lighthouse calibration values can be activated/deactivated using the respective push buttons.

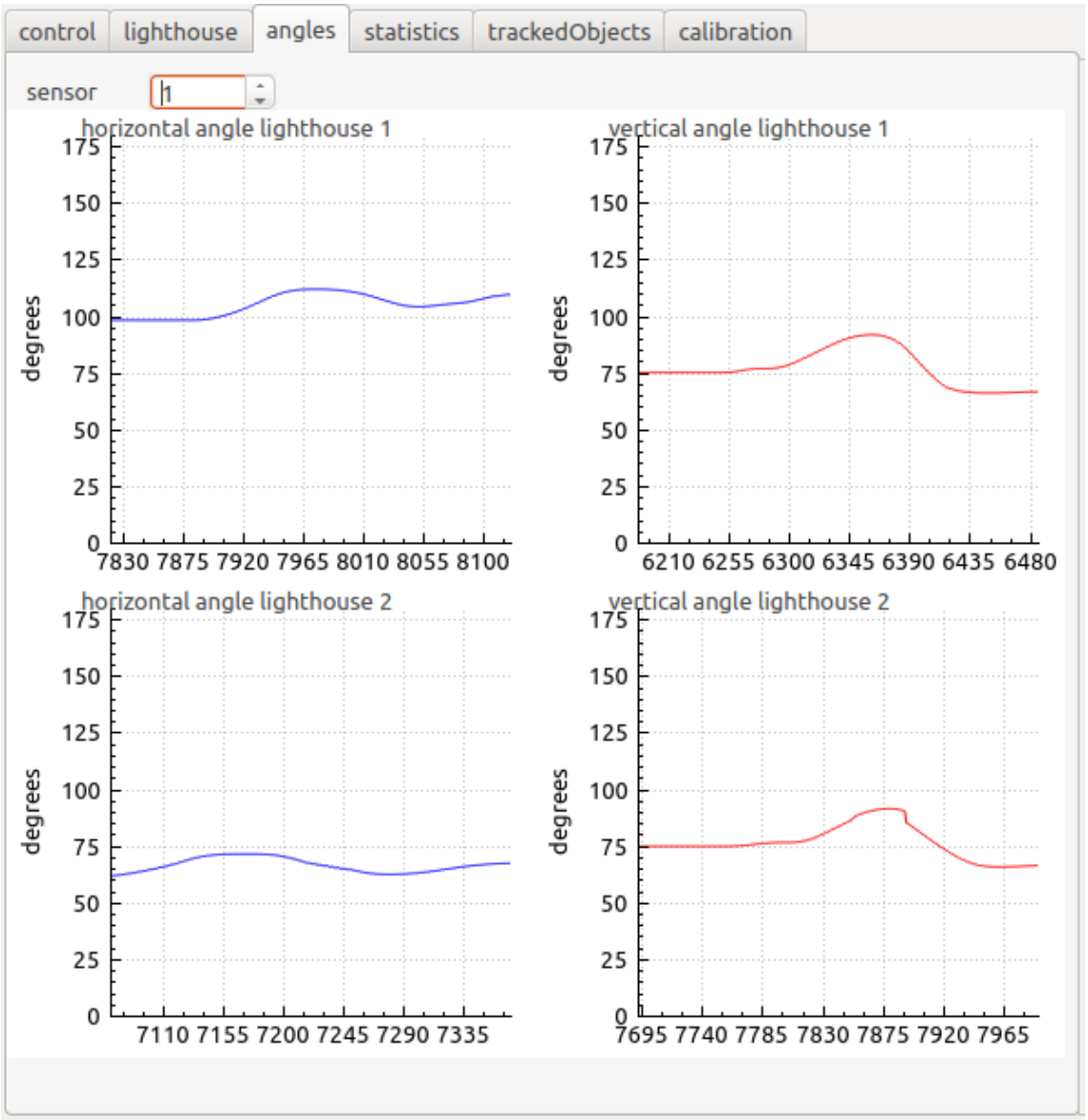


Figure A.2: Angles panel of DarkRoom GUI which plots live angle data for a chosen sensor for both lighthouses

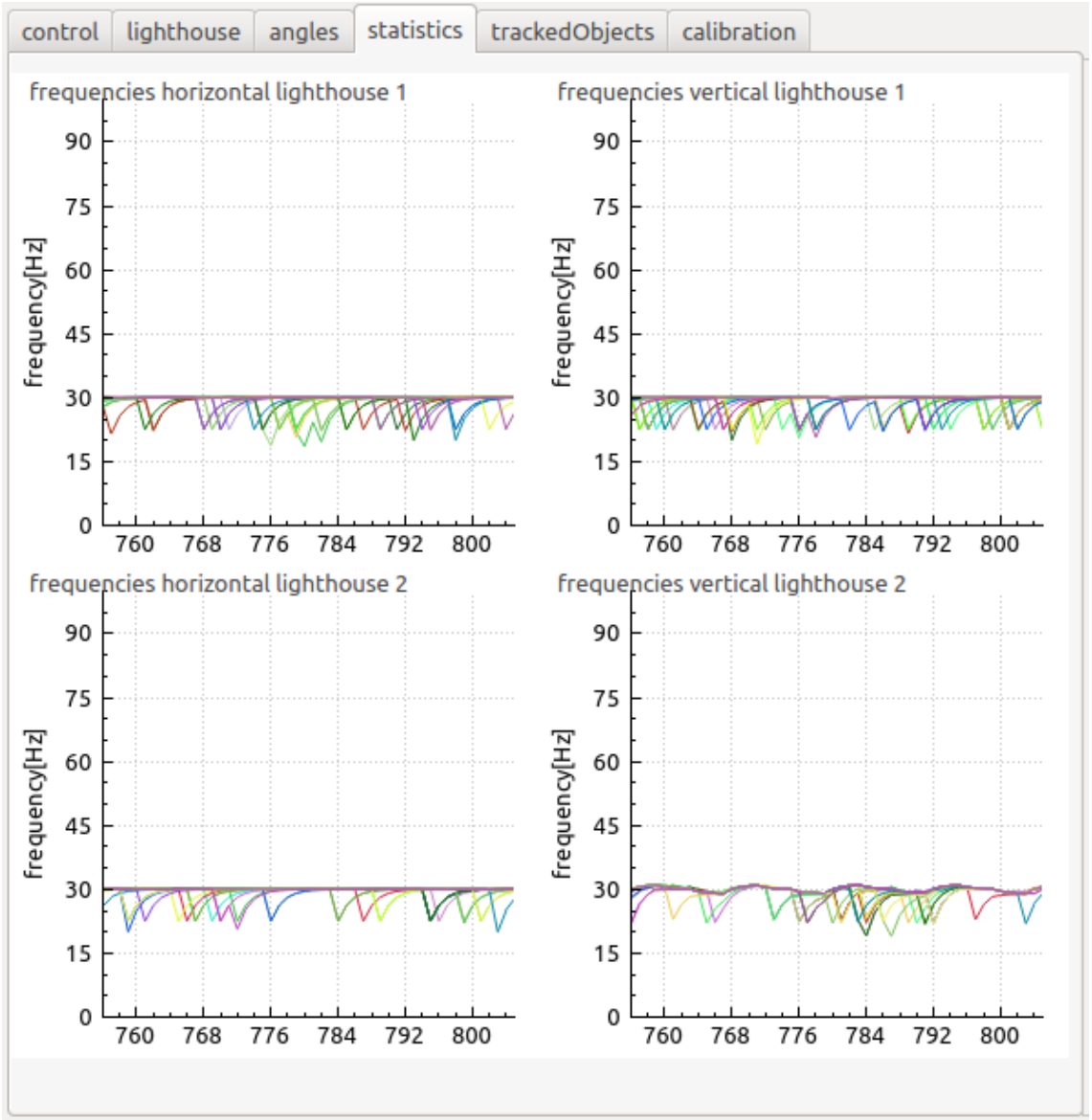


Figure A.3: Statistics panel of DarkRoom GUI, which plots the angle update frequencies for all sensors

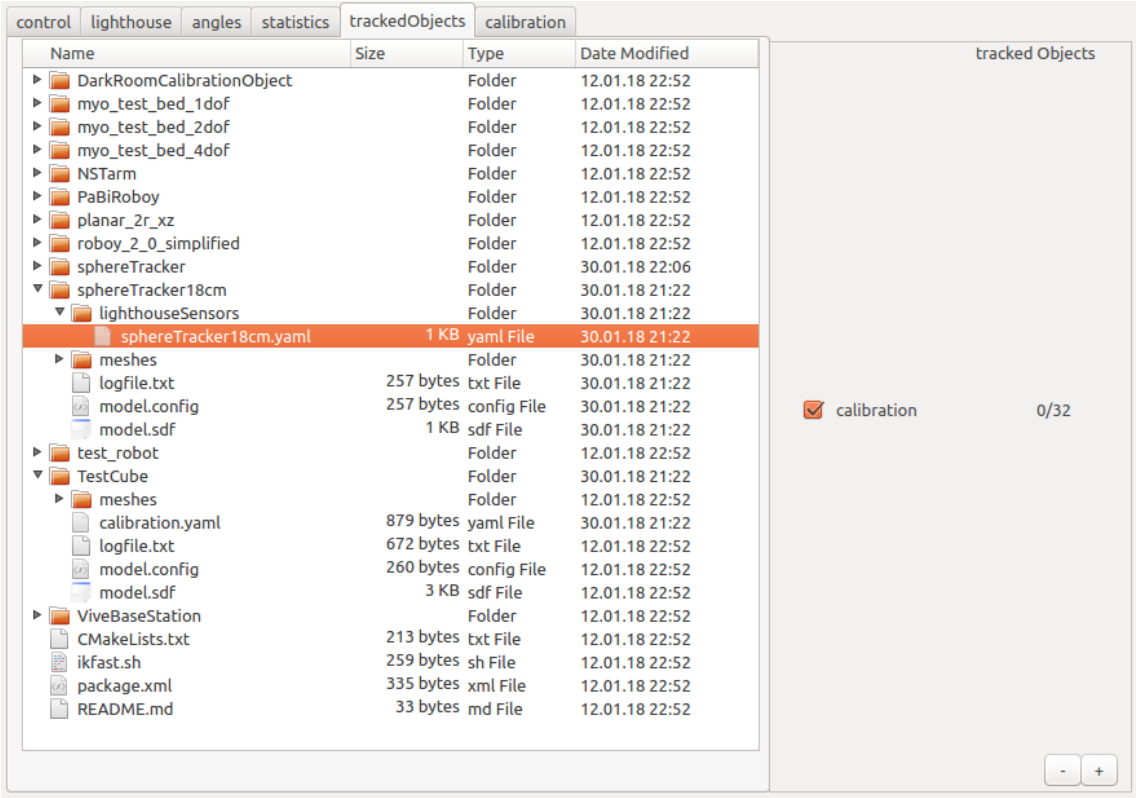


Figure A.4: Tracked objects panel of DarkRoom GUI. Here the user can select a tracked object configuration file and add the object using the plus button. Any selected object in the list on the right can be removed using the minus button.

Figure A.5: Calibration panel of DarkRoom GUI. The slider can be used to change the calibration values. The different push buttons implement the following functions: A estimates the calibration parameters based on a point estimate as described in 3.4; B resets any applied calibration values; C estimates the calibration parameters using a continuous recording and pose estimates from EPnP; D estimates the calibration parameters using a continuous recording and pose estimates from MLPE

Appendix B

Roboy 2.0 DE10-Nano-SoC Pinout

roboy de10-nano-soc PINOUT

LEGEND	
SCK[0-1]	SPI clock lines
MOSI[0-1]	SPI Master Out Slave In lines
MISO[0-1]	SPI Master In Slave Out
SS0[0-5]	SPI Slave Select lines of SPI bus 0 (the [] is the motor ID)
SS1[6-11]	SPI Slave Select lines of SPI bus 1 (the [] is the motor ID)
SCL[0-2]	I2C clock lines for 3 cores
SDA[0-2]	I2C data lines for 3 cores
GP0[0-1]	GPIO pins controlled by I2C core 0
GP1[0-1]	GPIO pins controlled by I2C core 1
GP2[0-1]	GPIO pins controlled by I2C core 2
SCK[ESP]	SPI clock line for ESP8266
MOSI[ESP]	SPI Master Out Slave In line for ESP8266
SS[ESP]	SPI Slave Select line for ESP8266
LS[0-31]	DarkRoom Lighthouse Sensor lines
LS_TX	DarkRoomOOTXDecoder uart TX

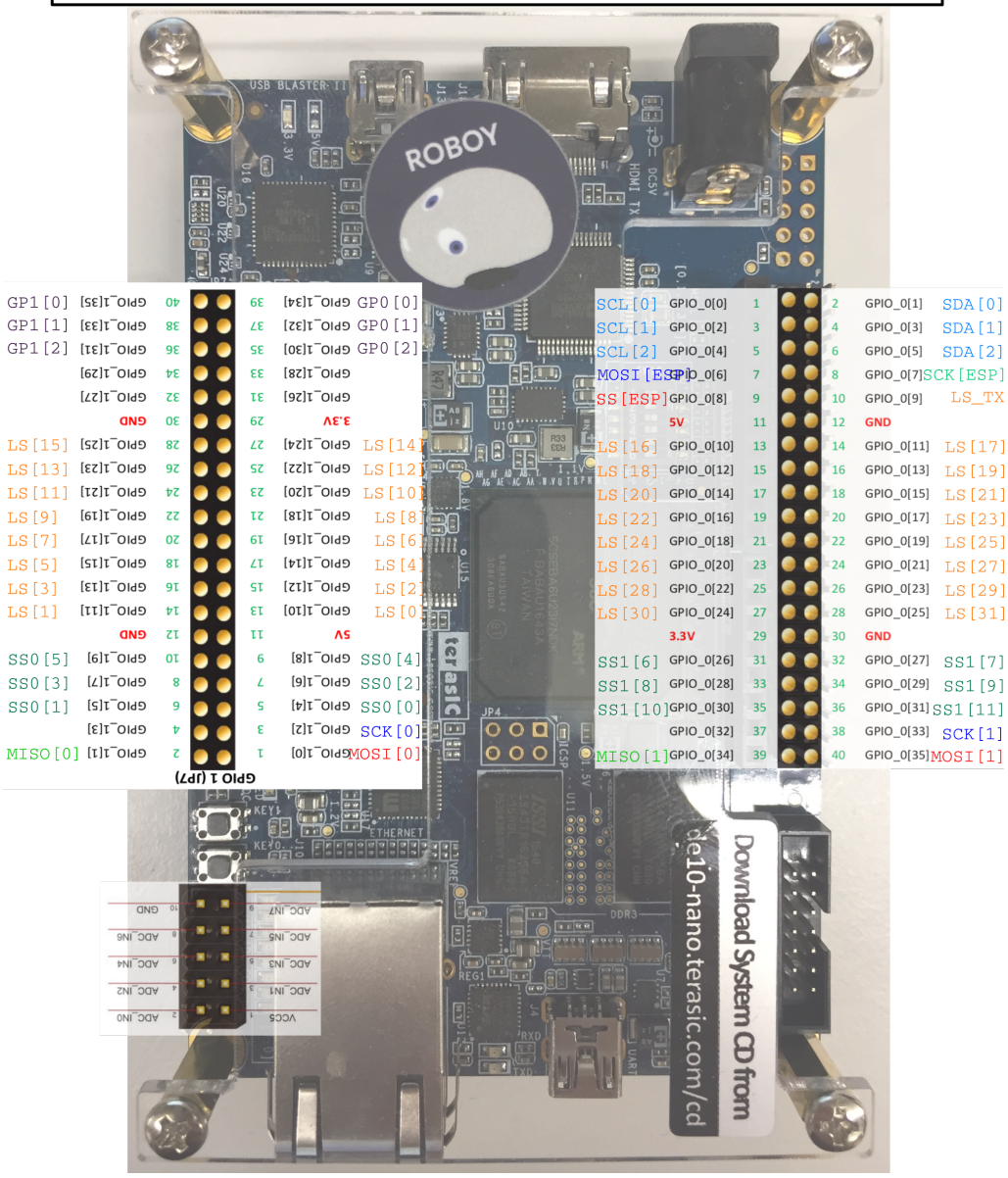


Figure B.1: Pinout of Roboy 2.0 DE10-Nano-SoC FPGA

List of Acronyms

API application programming interface 35

ARM Advanced RISC Machine 23, 24, 30, 63

AXI Advanced eXtensible Interface 24, 28, 33

BLDC Brush-less Direct Current Motor 9

CAD computer aided design 34, 64

CN Controlled Node 13–15, 30, 59, 63

CRC cyclic redundancy check 34

DART Dense Articulated Real-Time Tracking 7

DOF degrees of freedom 7, 24, 31, 34, 64

EKF extended kalman filter 41

EPnP Efficient Perspective-n-Point Camera Pose Estimation 7, 42, 72, 81

FOV field of view 17, 59

FPGA field programmable gate array 15, 23–25, 27–32, 40, 59, 63, 64, 74, 79, 81

FSM finite state machine 33

GPIO general-purpose input/output 27, 29–32, 64, 79

GUI graphical user interface 3, 41, 42, 44, 68–72, 79–81

HMD head mounted display 15

HPS hardware peripheral system 29

IDE Integrated development environment 14, 30

IMU interial measurement unit 41

IP intellectual property 28–30, 33, 34

ISR interrupt service routine 31

JTAG Joint Test Action Group 30

MDF medium-density fibreboard 38, 39

MDIO Management Data Input/Output 29

MII Media-Independent Interface 13, 29

MISO master in slave out 27

MLPE multi lighthouse pose estimation 35, 39, 40, 42, 46, 47, 49, 51, 52, 54, 55, 60, 61, 72, 80, 81

MN Managing Node 13–15, 30, 59

MOSI master out slave in 27

MyoToolKit MYOROBOTICS toolkit 9, 25–28, 63

OOTX Omnidirectional Optical Transmitter 20, 21, 33, 34, 38, 41, 59, 60, 68, 79, 80

PCB printed circuit design 18, 25

PHY physical layer 13, 15, 29

PReq poll request 15

PRes poll response 15

PWM pulse width modulation 11, 27

RGMII Reduced Gigabit Media-Independent Interface 29, 64

RISC reduced instruction set computer 30

RMII Reduced Media-Independent Interface 13, 29

RMSE root mean square error 46, 49, 56, 57, 80

ROS Robot Operating System 6, 24, 39–41

rviz ROS visualizer 41, 42

SoA start of asynchronous phase 15

SoC start of cycle 15

SPI Serial Peripheral Interface 26–28, 33, 63, 79

TCP Transmission Control Protocol 6, 13, 15

TPE triangulation pose estimation 35, 40, 42, 46, 47, 49, 50, 55, 60, 80

UART universal asynchronous receiver-transmitter 33, 34

UDP User Datagram Protocol 6, 13, 15, 33, 39, 42

VHDL Very High Speed Integrated Circuit Hardware Description Language 32, 33, 64

VR virtual reality 7, 15, 16

List of Figures

Bibliography

[1] T. BrogÃrdh, "Present and future robot control developmentâan industrial perspective," *Annual Reviews in Control*, vol. 31, no. 1, pp. 69 – 79, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1367578807000077

[2] C. Gehring, S. Coros, M. Hutler, C. D. Bellicoso, H. Heijnen, R. Diethelm, M. Bloesch, P. Fankhauser, J. Hwangbo, M. Hoepflinger, and R. Siegwart, "Practice makes perfect: An optimization-based approach to controlling agile motions for a quadruped robot," *IEEE Robotics Automation Magazine*, vol. 23, no. 1, pp. 34–43, March 2016.

[3] X. Da, R. Hartley, and J. W. Grizzle, "Supervised learning for stabilizing underactuated bipedal robot locomotion, with outdoor experiments on the wave field," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3476–3483.

[4] K. G. PEARSON, "Central programming and reflex control of walking in the cockroach," *Journal of Experimental Biology*, vol. 56, no. 1, pp. 173–193, 1972. [Online]. Available: http://jeb.biologists.org/content/56/1/173

[5] M. Hoffmann, H. Marques, A. Arieta, H. Sumioka, M. Lungarella, and R. Pfeifer, "Body schema in robotics: A review," *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 4, pp. 304–324, Dec 2010.

[6] V. Potkonjak, B. Svetozarevic, K. Jovanovic, and O. Holland, "The puller-follower control of compliant and noncompliant antagonistic tendon drives in robotic systems," *International Journal of Advanced Robotic Systems*, vol. 8, no. 5, p. 69, 2011. [Online]. Available: https://doi.org/10.5772/10690

[7] R. Pfeifer, P. Y. Tao, H. Gravato Marques, S. Weydert, D. Brum, M. Weyland, R. Hostettler, F. Volkert, V. GmÃ$\frac{1}{4}$nder, and D. Halbeisen, "Roboy anthropomimetic robot," Andreasstrasse 15, 8051 Zurich, 2013. [Online]. Available: www.roboy.org

[8] H. G. Marques, M. JÃntsch, S. Wittmeier, O. Holland, C. Alessandro, A. Diamond, M. Lungarella, and R. Knight, "Ecce1: The first of a series of an-

thropomimetic musculoskeletal upper torsos," in *2010 10th IEEE-RAS International Conference on Humanoid Robots*, Dec 2010, pp. 391–396.

[9] "Documentation page of the myorobotics myotoolkit," http://myorobotics.readthedocs.io/en/latest/, accessed: 2018-03-04.

[10] I. E. Committee, "Iec 61158-2, industrial communication networks - fieldbus specifications - part 2: Physical layer specification and service definition," 2014.

[11] V. Lepetit, F.Moreno-Noguer, and P.Fua, "Epnp: An accurate o(n) solution to the pnp problem," *International Journal Computer Vision*, vol. 81, no. 2, 2009.

[12] T. Schmidt, R. Newcombe, and D. Fox, "Dart: dense articulated real-time tracking with consumer depth cameras," vol. 39, 07 2015.

[13] "Vicon infrared retro-reflective marker tracking system," https://www.vicon.com/, accessed: 2018-03-14.

[14] "Htc vive virtual reality equipment," https://www.vive.com/eu/, accessed: 2018-03-14.

[15] H. G. Marques, M. Christophe, A. Lenz, K. Dalamagkidis, U. Culha, M. Siee, P. Bremner, and the MYOROBOTICS Project Team, "Myorobotics: a modular toolkit for legged locomotion research using musculoskeletal designs," in *Proc. 6th International Symposium on Adaptive Motion of Animals and Machines (AMAM'13)*, Darmstadt, Germany, 2013.

[16] H. Frazier, "The 802.3z gigabit ethernet standard," *Netwrk. Mag. of Global Internetwkg.*, vol. 12, no. 3, pp. 6–7, May 1998. [Online]. Available: http://dx.doi.org/10.1109/65.690946

[17] "Epsg ds 301 v1.2.0 - powerlink communication profile specification," https://www.ethernet-powerlink.org/downloads/technical-documents, accessed: 2018-03-04.

[18] "Positional tracking systems and methods patent," http://www.freepatentsonline.com/y2016/0131761.html, accessed: 2018-03-04.

[19] "Reverse engineered ootx frame," https://github.com/nairol/LighthouseRedox/blob/master/docs/Light%20Emissions.md, accessed: 2018-03-04.

[20] "Reverse engineered lighthouse info block," https://github.com/nairol/LighthouseRedox/blob/master/docs/Base%20Station.md#base-station-info-block, accessed: 2018-03-04.

[21] "De10-nano-soc user manual," https://www.altera.com/content/dam/altera-www/global/en_US/portal/dsn/42/

doc-us-dsnbk-42-4302081511597-de10-nano-user-manual.pdf, accessed: 2018-03-14.

[22] "Product page of de10-nano-soc from altera," http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1046, accessed: 2018-03-04.

[23] "Open-source verilog spi core," https://opencores.org/project,spi_master_slave, accessed: 2018-03-04.

[24] "Open-source verilog uart core," https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html, accessed: 2018-03-04.

[25] G. Terzakis, P. Culverhouse, G. Bugmann, S. Sharma, and R. Sutton, "On quaternion based parameterization of orientation in computer vision and robotics," *Journal of Engineering Science and Technology Review (JESTR)*, vol. 7, no. 1, pp. 82–93, 2014.

[26] C. Ancey, P. Coussot, and P. Evesque, "An improved method of pose estimation for lighthouse base station extension," *NCBI online*. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5677447/

License