Development of a ROS Node for a Mitsubishi RV-2F-D

eingereichter
PROJEKTBERICHT
von

B.Sc. Michael Schreibauer

geb. am 12.09.1990
wohnhaft in:
Höckstr. 14
82549 Königsdorf
Tel.: 0171 5601575

Lehrstuhl für
STEUERUNGS- und REGELUNGSTECHNIK
Technische Universität München

Univ.-Prof. Dr.-Ing./Univ. Tokio Martin Buss
Univ.-Prof. Dr.-Ing. Dirk Wollherr

Betreuer: Stefan Friedrich, M.Sc., Volker Gabler, M.Sc.
Beginn: 27.10.2015
Abgabe: 12.01.2015

In your final hardback copy, replace this page with the signed exercise sheet.

Contents

Chapter 1

Introduction

In both industrial and scientific environments robotic manipulators are used for various purposes. For example, the industry employs them to handle heavy and/or dangerous tasks like welding or moving hazardous items in order to increase the security of the workforce. The use of manipulators also allows to automatize a lot of monotonous work processes to ease the everyday working life of the employees and to increase productivity. The use of robotic manipulators for scientific purposes is also extensive. They can be utilized in order to conduct precise measurements, to research physical human robot interaction, to test new control techniques, etc. In order to facilitate the use of a robotic manipulator, an easy to use user interface is essential. This report is about connecting an industrial, 6 degree of freedom (DOF) manipulator to the so called Robot Operating System (ROS), which is an open-source software collection designed to develop robot applications. ROS is, although its development started as recently as 2007, widely distributed, which makes it an excellent choice as the user interface for the manipulator at hand. The manipulator on the other hand allows for position and joint control only as provided by its manufacturer, Mitsubishi Electric. As it operates on a set of commands designed by Mitsubishi, the manipulator can only be connected to ROS by means of a wrapper translating the robot-specific commands into a format which can be used within the ROS framework and vice versa.

Chapter 2

Main Part

2.1 Theoretical Background

This section provides the theoretical background for the design of the interface between the Robot Operating System (ROS) and the manipulator. The first subsection presents characteristic properties and advantages of ROS, whereas the second subsection focuses on the manipulator and its technical details.

2.1.1 Robot Operating System

ROS is an open-source software collection for robot software development. It was originally developed at the *Stanford Artificial Intelligence Laboratory* under the name *Switchyard* in 2007. Since then, eight different major distributions have been released, *ROS Jade Turtle* being the last. This development was strongly supported by the robotics research lab *Willow Garage*. As of February 2013, the ROS stewardship has been taken over by the *Open Source Robotics Foundation*, a non-profit organization founded by members of the global robotics community.
ROS is geared towards the Linux distribution *Ubuntu*, but there are also experimental releases for other operating systems such as Windows, Mac OS, etc. Due to the vast collection of open-source software, ROS provides a lot of different services, such as hardware abstraction, low-level device control and two different interfaces for passing data between processes. In this sense, ROS can also be thought of as a middleware for robotics.
Within ROS, a processes is called a *node*. These nodes are coordinated in a graph-like manner, where the graph nodes represent the different running processes and the links between nodes the communication channels of the respective nodes. This architecture is one of the big advantages of ROS, as it allows to develop different functionalities independent of each other if the shared interface is specified. Also, an existing functionality can easily be replaced by a more sophisticated one without the need to touch the rest of the software, as long as the existing interface is used.
There are two major types of interfaces between processes in ROS: *messages* and

services. A message is essentially a data structure containing an arbitrary combination of fundamental data types (integer, floating point, boolean, etc.). Messages can be sent into the ROS network by a node at an arbitrary frequency and received by another one. The former process is called *publishing a message*, the latter *subscribing to a message*. There are some built-in message types provided by ROS, but it is also possible to customize messages. In order to differentiate between different messages within the ROS network, there are so-called *topics* to which messages can be broadcast and be received from. They are the communication channels between the nodes. Staying with the illustration of how ROS coordinates processes from before, the links between the graph nodes represent the topics.
Services, on the other hand, resemble traditional function calls: they can be called at any time, data can be passed to them and feedback is provided. Service calls are, in contrast to messages, *blocking*: this means, that a running process needs to wait until the service it has called is finished. For this reason, this type of interface is not used in this project and therefore not elaborated any further.

2.1.2 Mitsubishi Melfa Manipulator

The manipulator at hand is a *Melfa RV-2F-D* constructed by *Mitsubishi Electric*. Its applications range from transport of mechanical parts to the assembly of electrical parts. It is controlled by a *Mitsubishi Industrial Robot CR750/CR751 Series* Controller. It receives its commands either from a remote control, a so called *teach pad* or from a personal computer using a basic command line tool. This setup is depicted in figure 2.1.

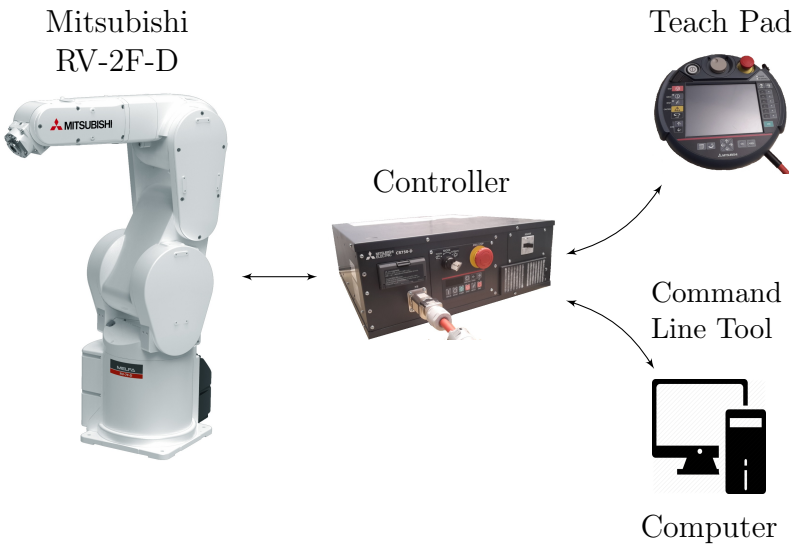


Figure 2.1: Overview of the Mitsubishi Melfa setup

The teach pad allows to either write simple point-to-point control programs or to

control single joints or the position directly. When the manipulator is controlled via a point-to-point control program, it moves with maximum operation speed, which is set to $250\frac{mm}{s}$. When controlling the manipulator directly, the operation speed can be adjusted as fractions of the maximum velocity (e.g. 50% of maximum speed, etc.). The teach pad also allows to specify the acceleration/deceleration rate of the manipulator. A more detailed explanation of the teach pad can be found in [Mit13]. The command line tool allows to move the robot using the arrow keys on the keyboard of a computer. The source code is written in C++ and provided by Mitsubishi Electric, see [Mit14].
The controller operates with a fixed motion movement control cycle of $7.1ms$ (approximately $140Hz$). That means the controller can receive a new command every $7.1ms$. The structure of the command it receives is predefined by Mitsubishi Electric. For example, it contains information about the current control type with which the manipulator is operated or about the status information the controller sends back to the teach pad or the computer. A detailed explanation of the data structure of the command can be found in [Mit14].
The manipulator operates within its own coordinate system located in its base, depicted in figure 2.2.

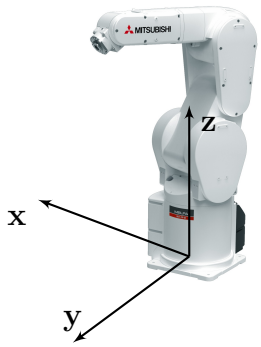


Figure 2.2: Coordinate system of the manipulator

2.2 Implementation

As mentioned in the introduction, a wrapper is needed to translate ROS commands to the controller-specific binary command and vice versa. The wrapper should be as transparent as possible, such that no other functionality is implemented inside it and the delay introduced remains low. Hence, it should only handle the communication between the controller and the ROS network. Additionally to the wrapper, a trajectory generator node and a basic status monitoring node are implemented. They are mainly needed to test the communication node, which is the interface between ROS and the controller, but they also provide an easy and intuitive access to the manipulator setup. If future applications require more sophisticated implemen-

tations, they can easily be replaced due to the modular character of ROS.
The trajectory generator node computes straight trajectories in the Euclidean space without orientation and is thus only suitable for position control. The status monitor node simply displays the current status of the robot on a terminal. These three nodes communicate via different ROS messages. As stated in subsection 2.1.1, these messages can be customized. Exploiting this fact, a `Melfa_Command` message was designed, which is needed to communicate with the communication node. Additionally, a `Position_Command` message was created, which is needed to use the trajectory generator. This setup is depicted in the following figure:

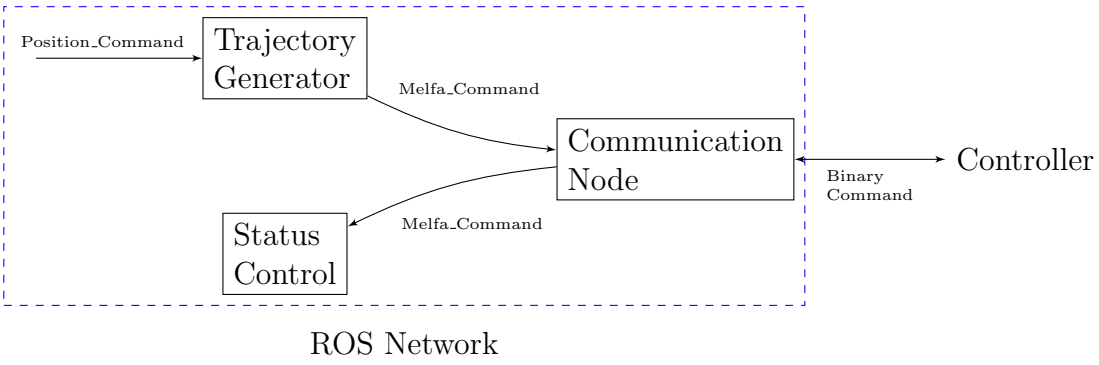


Figure 2.3: Overview of work flow with the implemented ROS nodes

In the following three subsections, the functionalities of these three nodes, as well as their corresponding ROS message types, are explained in detail.

2.2.1 Trajectory Generator Node

This subsection provides information about the trajectory generator node. The node subscribes to a topic called *mitsubishi_targets*, from which it receives a *Position_Command* ROS message, which is structured as follows:

Data type	Name
`float32`	`x`
`float32`	`y`
`float32`	`z`
`float32`	`velocity_factor`

Figure 2.4: ROS Message designed for the trajectory generator

The different components of this structure are explained in the following:

x, y, z: These three variables set a desired position in the manipulators coordinate system (see subsection 2.1.2).

velocity_factor: This variable sets the speed the manipulator moves towards the desired position. Analogously to the case of the teach pad, this variable denotes the fraction of the maximum speed the robot is able to move at (see subsection 2.1.2).

Once the generator receives a *Position_Command* message, it computes a series of sample points of the trajectory towards the goal. The number of sample points is dictated by the variable *velocity_factor* (high velocity corresponding to a low number of sample points). These sample points are then wrapped into a *Melfa_Command* ROS message, which is published onto a topic called *mitsubishi_target_sample_positions*. The communication node, which is described in the next subsection, subscribes to this topic.

2.2.2 Communication Node

This subsection describes the communication node in detail, which handles the communication between the ROS network and the controller of the manipulator. As described in the last subsection, it subscribes to the `mitsubishi_target_sample_positions` topic, from which it receives position and velocity information wrapped in a `Melfa_Command` ROS message. This is a generic message, usable for both joint and position control. Its structure is as follows:

Data type	Name
int32	Control_Type
char	Type_Monitor1
char	Type_Monitor2
char	Type_Monitor3
char	Type_Monitor4
float32	x
float32	y
float32	z
float32	rot_x
float32	rot_y
float32	rot_z
float32	j1
float32	j2
float32	j3
float32	j4
float32	j5
float32	j6

Figure 2.5: ROS Message designed for communication node

One might wonder about the choice of the data types of the different components. For example, the choice of the data type `float` for the joint/position values instead

of `double` may seem questionable, as it entails a loss of precision. The reason is the fixed data structure of the commands received by the controller. All data types predetermined by this structure are used in the implementation of the ROS nodes and messages to avoid type-casting of the variables. The different components of this message type are explained in the following:

Control_Type: This variable sets the control mode of the control box. It needs to be set to one of the following values:

- 0: None, robot does not move if the control box receives this type
- 1: Position control
- 2: Joint control

Type_Monitor: These variables allow to determine the type of feedback information the controller sends back to the computer. Only four different states out of eleven available states can be chosen and thus be monitored. This is due to the structure of the commands of the controller. By setting the `Type_Monitor` variables to values between one and eleven, the different types of feedback information are determined. The correspondence of the values of the variables and the information types are as follows:

- 1: Euclidean coordinate data
- 2: Joint coordinate data
- 3: Motor pulse coordinate data
- 4: Euclidean coordinate data (command value after filter processing)
- 5: Joint coordinate data (command value after filter processing)
- 6: Motor pulse coordinate data (after filter processing)
- 7: Euclidean coordinate data (encoder feedback value)
- 8: Joint coordinate data (encoder feedback value)
- 9: Motor pulse coordinate data (encoder feedback value)
- 10: Current command [%]
- 11: Current feedback [%]

The filter process mentioned above refers to the acceleration/deceleration of the manipulator described in subsection 2.1.2.
The order of feedback types does not matter: the first of the `Type_Monitor` variables can be set to eight, whereas the second one can be set to two and so on. But as each of the `Type_Monitor` variables corresponds to a specific part of the data structure of the controller command, one needs to remember which variable is set to which value to extract the desired information from the feedback command.

x, y, z: Position in manipulator coordinate system

rot_x, rot_y, rot_z These variables denote the offset to the orientation of the initial position

j1 - j6: Angles of the six different joints

Once the node receives a message of this type, its contents are stored in a ring buffer. This is due to the fact, that ROS nodes may operate on different frequencies, which could lead to the situation that the communication node receives commands at a rate which is slower or faster than it can operate at. This operation rate is fixed at $140Hz$, because the controller works at a rate of $140Hz$ as described in subsection 2.1.2. Receiving commands at a rate slower than $140Hz$ would lead to a slower movement as intended (this situation was encountered during the testing of the software, see section 2.3). A faster rate would only fill the buffer of the communication node, which then processes the contents at its fixed operation rate of 140 Hz. The communication node checks whether the buffer is empty or not. If it is not, the desired position in the Euclidean or joint space is translated into the controller-specific format and sent to the controller. If the sending process does not work as intended, the controller is disconnected and the manipulator is shut down for safety. If everything works according to plan, the controller sends a feedback to the communication node after its fixed execution time of $7.1ms$. This feedback is then wrapped to the generic `Melfa_Command` ROS message and published onto a topic called `mitsubishi_melfa_status`. The status monitoring node, which is explained in the next subsection, subscribes to this topic.
Besides the processing of commands, the communication node also provides the possibility to stop the manipulator during a movement. For that purpose it subscribes to a topic called `mitsubishi_anytime_stop`, from which it receives ROS standard messages of type *Bool* of the following structure:

Data type	Name
`bool`	`data`

Figure 2.6: ROS Message designed for the anytime stop

If an incoming message contains a boolean set to true (1), the ring buffer is emptied and thus stops the manipulator from moving any further. As long as no new message containing a boolean set to false (0), the manipulator will stand still. This mechanism is also used to protect the ring buffer against a buffer overflow. Once all of the available memory of the buffer is used, the buffer is emptied and the manipulator is stopped, just like the manipulator is stopped via the anytime stop functionality. The user is notified via the terminal that a buffer overflow was intercepted and that the manipulator is stopped. An incoming message via the `mitsubishi_anytime_stop` topic containing a boolean set to true (1) is required to enable the manipulator to receive new commands.

2.2.3 Status Monitor Node

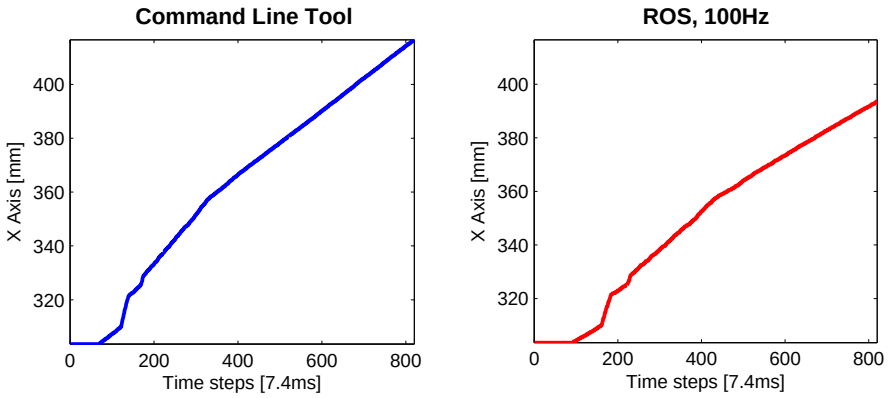
This subsection explains the node implemented to monitor the manipulators status. It subscribes to the *mitsubishi_melfa_status* topic, from which it receives a message of the *Melfa_Command* type (see subsection 2.2.2). At this moment, the node simply displays the current status of the manipulator on a terminal. It could be modified in various ways, but this modification would largely depend on the intended application of the manipulator.

2.3 Offline Test

The implemented software was first tested offline, as the manipulator can generate enormous forces while operating, which can pose a risk for the user. Thus, the implementation described in the last section is modified in order to test the functionality of the nodes: the communication node does not send any commands to the robot, but logs the generated binary commands to a text file. The idea behind this is the following: the command line tool described in subsection 2.1.2 is used to move the manipulator along an arbitrary trajectory. The binary commands generated by the command line tool are saved to a text file. After that, the trajectory generator node is used to compute the exact same trajectory as the command line tool did and the generated binary commands of the communication node are also saved to a text file. If these two text files do not differ for the period of the movement, the ROS nodes should be safe to use with the hardware.
The first tests were unsuccessful, as the two text files did not match. Analysing the files further revealed, that the commands generated by the communication node moved the manipulator somewhat slower than the command line tool, but the commands itself were valid. Further investigation showed that the trajectory generator operated on a rate slower than the execution rate of the controller. Raising the rate of the trajectory generator and adding the ring buffer described in section 2.2 solved this problem. Figure 2.7 visualizes the behaviour of the ROS nodes exemplary with a lower or higher rate than $140Hz$, respectively. The manipulator was driven along the x coordinate axis of its coordinate system. Figure 2.7(a) shows the increase of the x coordinate value generated by the command line tool, figure 2.7(b) the increase generated by the ROS nodes at slower rate as the controller operates on. It can be observed how the trend of the increase is the same, but somewhat slower. Figure 2.7(c) shows the result after tuning up the operation rate of the ROS nodes: the trajectory matches the trajectory generated by the command line tool.

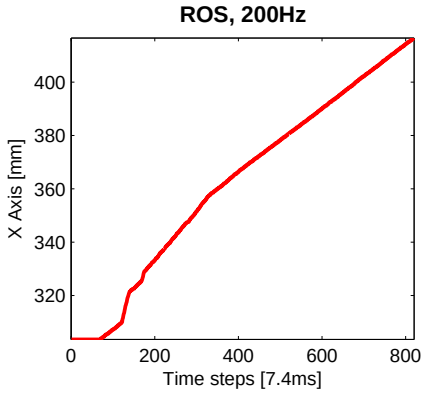
2.4 Online Test

After the successful offline tests of the ROS nodes, we were able to test the ROS nodes online with the manipulator. This is done in two steps: first, various trajec-



(a) Increase of the x coordinate value generated by the command line tool

(b) Increase of the x coordinate value generated by the ROS nodes operating at a rate of $100Hz$



(c) Increase of the x coordinate value generated by the ROS nodes operating at a rate of $200Hz$

Figure 2.7: Exemplary results of the offline testing

tories in the Euclidean space are generated using Matlab. This is due to the fact that the trajectory generator is only able to produce straight movements but also curvy trajectories need to be tested. These test trajectories are then fed into the generator, which wraps them into the necessary ROS message and sends them to the manipulator. It should be noted that these trajectories moved the manipulator at a very low speed for safety reasons. The manipulator followed each of the test trajectories as planned.

The second step of the online test is to implement a ROS node, which sends arbitrary position commands to the communication node. Four different points in space are used and the manipulator moved to point after point at various speed levels. Figure 2.8 depicts one run of that test setup. The manipulator behaved as planned for each velocity.

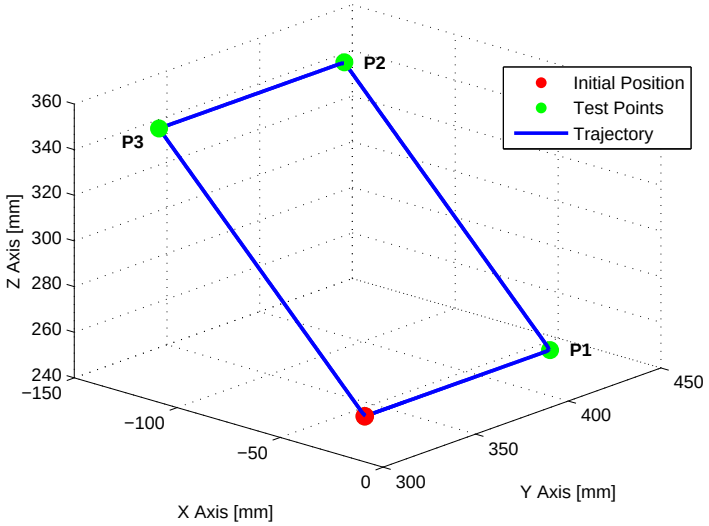


Figure 2.8: Exemplary online test of the manipulator

2.5 Operating Instructions

This section explains how to use the implemented software together with the manipulator. The first subsection explains all necessary preparatory tasks and the second subsection illustrates how to run the manipulator setup with the ROS framework.

2.5.1 Preparations

- Set the parameter of the controller via the teach pad to:
  - NETIP : 192.168.0.20
  - NETPORT : 10000
  - MXTTOUT : -1
- Create a point-to-point movement program with the help of the teach pad. This program moves the manipulator to its initial position (which can be defined arbitrarily) and then enables the controller to receive commands from the computer. The program sets the control mode of the manipulator (position or joint), which means once the manipulator is started, the control mode can not be changed. The running program needs to be terminated and to be restarted with the desired configuration. In practice, it is advisable to create two different programs, one set to joint, the other one set to position control. In the following, the structure of these programs is presented:

```
Open "ENET:192.168.0.2" As #1
Mov P1
Mxt 1,0,50 (joint) or Mxt 1,1,50 (joint)
Mov P1
Hlt
```

The program is explained line by line in the following:

Open "ENET:192.168.0.2" As #1 This line connects the manipulator to the computer via the specified IP address.

Mov P1 This line moves the manipulator to its initial position defined as P1 (either a specific joint combination or a point in the manipulators coordinate system).

Mxt 1,0/1,50 This command enables the controller to receive commands from the computer. *Mxt* denotes the *Move External* functionality of the controller:
Mxt ⟨File No.⟩, ⟨Control type⟩, ⟨Filter time constant⟩
Of interest are the second and the third argument: the second argument defines the control type and the third the acceleration/deceleration characteristic. For a more detailed explanation refer to [Mit14].

Mov P1 This line moves the manipulator back to its initial position after the external control is terminated.

Hlt This line ends the program.

- Connect the robot controller and PC with Ethernet cable

2.5.2 Operating the Manipulator

The manipulator needs always to be started before the ROS nodes, at least before the communication node. This is a safety precaution, because if the nodes are started before the manipulator, it might receive commands immediately after it is switched on. This situation might lead to unexpected movements of the manipulator which could endanger the user. For safety, one should follow the following instructions in the given order:

1. Set the controller to automatic mode.
2. Turn on the servos of the manipulator and run the point-to-point movement program via the teach pad. Be careful to run the correct control settings.

3. Start the ROS nodes. Each node advertises its successful initialization. The manipulator is now ready to receive commands.

4. To end the session close all ROS nodes. The order in which the nodes are closed does not matter.

5. Turn of the servos of the manipulator, stop the point-to-point movement program and reset it.

Chapter 3

Conclusion

This report is about the development of an interface between the Robot Operating System (ROS) and a robotic manipulator. First, the properties of both ROS and the manipulator are elaborated. After that, the design of the implementation is explained and justified. Finally, the implemented software is tested thoroughly, at first without the hardware for safety reasons. Only after successful offline testing, the software is tested online with the manipulator, which proved to be successful.

The implemented software provides an easy and intuitive way to control the manipulator via the ROS framework. Nevertheless, some modules of the software could be improved: the trajectory generator only computes trajectories in the Euclidean space of the manipulator's inherent coordinate system, but not in joint space. This is due to the lack of the inverse kinematics, which makes a reasonable computation of a trajectory in the joint space impossible. But if the inverse kinematics become available, this would be a useful enhancement. Also, the status control could be improved, for example to restrict the workspace of the manipulator: if the manipulator moved outside the allowed workspace, it would be stopped or shut down completely.

Appendices

Appendix A

Software Structure

This chapter describes the structure of the software which was implemented for the project. The first section concentrates on the ROS nodes implemented to connect the manipulator to the ROS framework. The second section describes the software used for the on- and offline testing.

A.1 Manipulator Specific Software

As ROS organizes the software in packages, a *mitsubishi_melfa* package was created, which contains all the necessary code:

A.1.1 Header Files

mitsubishi_melfa\include\mitsubishi_melfa\CommandBuffer.hpp Header file for the ring buffer class which stores unprocessed commands for the controller

mitsubishi_melfa\include\mitsubishi_melfa\UDPCommunication.hpp Header file for the communication class handling the network communication between controller and computer

mitsubishi_melfa\include\mitsubishi_melfa\UDPCommand.h Header file provided by Mitsubishi Electric containing the controller specific command

A.1.2 Source Files

mitsubishi_melfa\src\CommandBuffer.cpp Source file of the ring buffer class

mitsubishi_melfa\src\UDPCommunication.cpp Source file of the network communication class

mitsubishi_melfa\src\mitsubishi_melfa_communication.cpp Source file of the communication node

mitsubishi_melfa\src\mitsubishi_trajectory_generator.cpp Source file of the trajectory generator node

mitsubishi_melfa\src\mitsubishi_status_monitor.cpp Source file of the status monitoring node

A.1.3 Message Files

mitsubishi_melfa\msg\MelfaCommand.msg Message file containing the customized ROS message needed to communicate with the communication node

mitsubishi_melfa\msg\PositionCommand.msg Message file containing the customized ROS message needed to communicate with the trajectory generator

A.2 Test Software

mitsubishi_melfa\src\mitsubishi_anytime_stop.cpp Source file containing a simple ROS node used to test the anytime stop functionality

mitsubishi_melfa\src\mitsubishi_target_generator.cpp Source file of a simple ROS node which was used for the online testing

mitsubishi_melfa\src\Offline_Test\compareLogfiles.m Matlab script used to compare the log files generated during the offline tests

mitsubishi_melfa\src\Online_Test\compareLogfiles.m Matlab script used to evaluate the online tests

List of Figures

Bibliography

[Mit13] Mitsubishi Electric Corporation. *Mitsubishi Industrial Robot R56TB/R57TB Instruction Manual*, December 2013.

[Mit14] Mitsubishi Electric Corporation. *Mitsubishi Industrial CR750/CR751 Series Controller Instruction Manual*, August 2014.

License