# Preface

As it commonly is done at the end of a thesis, we would like to express our gratitude to some people who have contributed to the realisation of our master's thesis.

First of all, we would like to thank our head thesis supervisor prof. dr. ir. Eric Demeester who has always set the highest standards for us throughout the whole thesis period. His enlightening conversations and insights kept us focused to achieve our goal. We would also like to extend our thanks to our thesis co-supervisor ir. Jeroen De Maeyer for his invaluable help and guidance throughout the master's thesis. Even when we no longer could see the wood for the trees, he provided us some useful tips to continuing the hard work. We wish you the best of luck in continuing your doctoral study.

We are also grateful to Adriaan Broere, Managing director at Valk Welding B.V. and Tom De Boom, project support engineer at Valk Welding B.V. for the support with the Panasonic software and communication. We would also like to thank ir. Gijs van der Hoorn, robotics researcher at TU Delft and ROS-Industrial Project Manager, for sharing his knowledge about the ROS environment and providing us with useful insights. Also, we would like to thank Michael Büsch for the e-mail support on the PROFIBUS software stack.

Last but not least we would like to thank our family and friends for supporting us throughout the whole year.

*David De Schepper*
*Jorn Geutjens*
*May, 2018*

# Contents


Page

List of Tables 5

List of Figures 7

List of Abbreviations 9

Abstract 11

Beknopte samenvatting 13

1 Introduction 15
- 1.1 Background . . . 15
- 1.2 Problem description . . . 16
- 1.3 Goals . . . 17
- 1.4 Approach . . . 18
- 1.5 Contributions of this work . . . 19
- 1.6 Chapter preview . . . 19

2 Literature survey 21
- 2.1 Introduction . . . 21
- 2.2 Robot Operating System (ROS) . . . 21
  - 2.2.1 ROS features . . . 22
  - 2.2.2 ROS-Industrial . . . 25
- 2.3 Specifications of a ROS-Industrial driver for industrial robots . . . 26
  - 2.3.1 Operation . . . 26
  - 2.3.2 ROS API . . . 27
- 2.4 Characteristics of available ROS drivers for industrial robots . . . 27
- 2.5 Communication between ROS and robot controller . . . 30
  - 2.5.1 General ROS communication concepts . . . 31
  - 2.5.2 Simple Message Protocol . . . 31
  - 2.5.3 Industrial Robot Client . . . 34
- 2.6 Conclusion . . . 35

3 Panasonic VR-006L robot model for ROS 37
- 3.1 Introduction . . . 37
- 3.2 Principles of a Unified Robot Description Format (URDF) . . . 37
  - 3.2.1 Description of robot links . . . 37
  - 3.2.2 Description of robot joints . . . 38
- 3.3 Panasonic VR-006L visualisation . . . 41
- 3.4 Path planning model for ROS and MoveIt! . . . 42
- 3.5 Conclusion . . . 43

**4 PROFIBUS communication** 45
4.1 Introduction . . . 45
4.2 Fieldbus . . . 45
4.3 Basics of the PROFIBUS communication protocol . . . 46
4.4 Transmission . . . 49
4.5 Applications . . . 50
4.6 Open implementations of PROFIBUS . . . 51
4.7 Conclusion . . . 53

**5 Implementation of the ROS driver** 55
5.1 Introduction . . . 55
5.2 Offline communication . . . 55
5.3 Bidirectional setup . . . 57
5.3.1 Strategy . . . 57
5.3.2 Implementation . . . 58
5.4 Experimental results and evaluation . . . 60
5.4.1 Testing the communication programme on the Panasonic robot controller . . . 60
5.4.2 Testing the communication between ROS and a Siemens ET 200S I/O system using an open PROFIBUS implementation . . . 61
5.4.3 Testing the communication between ROS and the Panasonic robot controller . . . 63
5.5 Conclusion . . . 64

**6 General conclusion** 65
6.1 Contributions of this work . . . 65
6.2 Future work . . . 66

**Bibliography** 67

**A Dutch Summary** 71

**B Complete URDF model** 83

**C Kinematic chain of the Panasonic VR-006L robot** 87

**D A Denavit-Hartenberg representation of the Panasonic VR-006L robot** 89

**E Used code** 91

# List of Tables

2.1 Comparison between the available ROS drivers for industrial manipulators 28

4.1 Overview of applications where PROFIBUS is used . . . . . . . . . . . . . . 51

5.1 Overview of the inner protocol between ROS and the Panasonic robot controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

5.2 Overview of which coloured wire corresponds to which physical wire . . . . 63

D.1 DH parameters for a Panasonic VR-006L robot. Distances are represented in mm, angles in rad. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

# List of Figures


1.1 The setup of the Panasonic VR-006L manipulator at ACRO, Diepenbeek . 15
1.2 The annual supply of industrial robots at year-end by industries worldwide 16
1.3 The relationship between the different ROS nodes used in this master's thesis. The circle displayed in red is the Panasonic driver node. . . . . . 17
1.4 A structural overview of the following chapters . . . . . . . . . . . . . . . . 19

2.1 Logo of ROS Indigo Igloo, released on July 22nd, 2014 . . . . . . . . . . . 23
2.2 Visualisation of the ROS features as explained above . . . . . . . . . . . . 24
2.3 Example of a Joint_Position Simple Message . . . . . . . . . . . . . . . . . 32
2.4 Example of a Joint_Trajectory_Point Simple Message . . . . . . . . . . . 33
2.5 Example of a Status Simple Message . . . . . . . . . . . . . . . . . . . . . 34

3.1 Schematic description of a link in a URDF file . . . . . . . . . . . . . . . . 38
3.2 Schematic description of a joint in a URDF file . . . . . . . . . . . . . . . 39
3.3 Visual aid to build the kinematic chain of a robot . . . . . . . . . . . . . . 40
3.4 Visualisation of the Panasonic VR-006L manipulator in RViz . . . . . . . . 41
3.5 Visualisation of the coordinate frames; left on the CAD model and right in RViz . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.6 The schematic representation of forward and inverse kinematics . . . . . . 42
3.7 Screenshot of the *MoveIt! Setup Assistant* . . . . . . . . . . . . . . . . . . 43

4.1 Logo of the PROFIBUS organisation . . . . . . . . . . . . . . . . . . . . . 45
4.2 The seven layers of the OSI-model applied to PROFIBUS . . . . . . . . . . 46
4.3 The master-slave protocol used by PROFIBUS . . . . . . . . . . . . . . . . 47
4.4 Principle of token passing in PROFIBUS . . . . . . . . . . . . . . . . . . . 47
4.5 Explanation of a SD1 telegram. One block contains one octet of memory storage. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.6 Explanation of a SD2 telegram. One block contains one octet of memory storage. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.7 Explanation of a Short Acknowledgement telegram, with a value of $E5_{HEX}$ 49
4.8 Schematic representation of the RS-485 transmission technique . . . . . . . 50

5.1 Premeditation of a conversion from ROS commands to a csr file . . . . . . 56
5.2 Bidirectional setup between ROS and the Panasonic robot controller . . . . 57
5.3 Pseudocode of the position streaming interface implemented on the Panasonic robot using the DTPS software . . . . . . . . . . . . . . . . . . . . . 58
5.4 Schematic overview of the three implemented robot movement types . . . . 59
5.5 Schematic representation of the communication setup between the robot controller and a Siemens S7-315 PLC . . . . . . . . . . . . . . . . . . . . . 60
5.6 Example of the data traffic send to the robot controller over the PROFIBUS network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

5.7 Schematic representation of the bidirectional communication setup between ROS and a Siemens ET 200S I/O system over PROFIBUS-DP, using a USB to RS-485 adapter . . . 62
5.8 A *Do It Yourself* (DIY) USB to RS-485 adapter . . . 62
5.9 Schematic representation of the bidirectional communication setup between ROS and the Panasonic robot controller over PROFIBUS-DP, using a USB to RS-485 adapter . . . 63

C.1 Kinematic chain of the Panasonic VR-006L robot, as described in the URDF 87

D.1 Denavit-Hartenberg frames of the Panasonic VR-006L robot . . . 90

# List of Abbreviations

**General Symbols**

| | |
|---|---|
| *ACRO* | Automation, Computervision and Robotics |
| *CAD* | Computer Aided Design |
| *OS* | Operating System |
| **PROFIBUS related Symbols** | |
| *DP* | Decentralized Peripherals |
| *ED* | End Delimiter |
| *FMS* | Fieldbus Message Specification |
| *PI* | Profibus International |
| *PLC* | Programmable Logic Controller |
| *SD* | Start Delimiter |
| *UART* | Universal Asynchronous Receiver/Transmitter |
| **Robot related Symbols** | |
| *API* | Application Programming Interface |
| *DOF* | Degrees Of Freedom |
| *DTPS* | Desktop Programming and Simulation System |
| *FK* | Forward Kinematics |
| *IK* | Inverse Kinematics |
| *rgb* | Red, Green and Blue (tristimulus) |
| *ROS* | Robot Operating System |
| *rpy* | Roll, Pitch and Yaw |
| *RViz* | ROS Visualizer |
| *SRDF* | Semantic Robot Description Format |
| *URDF* | Unified Robot Description Format |
| *XML* | Extensible Markup Language |

# Abstract

Over the last decades, a substantial part of robotics research has focused on path planning algorithms for robotic manipulators. To be generic, these novel algorithms make an abstraction of the underlying robotic hardware: they generate robot motion commands and expect robot state information in a specific format that is independent of the proprietary language in which commercial robots are typically programmed. Therefore, for each commercial robot that needs to be controlled using these algorithms, a driver is required to transform the robot-independent commands to robot-specific commands and vice versa. ROS is an example of an open-source robot software platform in which such drivers can be developed.

The main objective of this thesis is to implement a driver in ROS for a Panasonic robot. Three steps have been taken to achieve this. Firstly, a geometric and kinematic robot model was created for ROS. Secondly, a unidirectional communication between ROS and the robot controller was implemented. Finally, a bidirectional communication was implemented to make the driver robust.

A ROS driver using an open PROFIBUS implementation has been realised. Also, an offline programme for making Panasonic-specific csr files has been made. These files are used to format all the instructions in the Panasonic controller. The driver is based on Panasonic's G2 robot controller, which cannot read in joint positions. Therefore, the thesis proposes to upgrade to a G3 controller with an Ethernet connection as future work.

# Beknopte samenvatting

Afgelopen decennia is een substantieel deel binnen robotonderzoek gericht op padplanningsalgoritmes voor industriële robots. Om generiek te zijn, maken deze nieuwe algoritmes een abstractie van de onderliggende robot hardware: ze genereren robotbewegingscommando's en verwachten informatie van de robottoestand in een formaat onafhankelijk van de specieke taal waarin de commerciële robots zijn geprogrammeerd. Daarom is voor iedere commerciële robot een driver nodig om robotonafhankelijke commando's om te vormen naar robotspecifieke commando's en vice versa. ROS is een voorbeeld open-source software platform waar zulke drivers geschreven kunnen worden.

De hoofddoelstelling van de thesis is het implementeren van een driver in ROS voor een Panasonic robot. Drie stappen zijn genomen om dit te bereiken. Eerst werd een kinematisch en geometrisch model van de robot opgesteld. Ten tweede werd er unidirectionele communicatie voorzien tussen ROS en de robotcontroller. Tot slot werd er bidirectionele communicatie geïmplementeerd om de driver robuust te maken.

Een ROS-driver is gerealiseerd gebruik makende van een open PROFIBUS implementatie. Ook is een programmma geschreven dat ROS-instructies omzet in Panasonic instructies, in de vorm van een csr-bestand. Deze bestanden worden gebruikt om instructies te formatteren in de Panasonic controller. De driver is gebaseerd op de G2-controller dewelke geen joint-posities kan lezen. Daarom stelt de thesis voor om te upgraden naar een G3-controller met Ethernetconnectie.

# Chapter 1

# Introduction

## 1.1 Background

The master thesis is taking place at ACRO (Automation, Computervision and Robotics), a research facility from the university of Leuven (KU Leuven). ACRO's research focus is to use computer vision in automation and robotics applications. In addition, they also give courses on industrial automation software like PROFIBUS, PROFINET and PLC programming [1].

At ACRO, Robot Operating System (ROS) is being used for research and development of robotics. ROS is an open-source framework developed by the American software company Willow Garage in cooperation with Stanford University. ROS contains a large collection of libraries, tools and conventions to make the development of complex robot software easier. It enables higher reliability on the behaviour of the robots and does this for a wide range of robotic platforms. ROS is completely open-source, meaning that it allows *packages* (independent pieces of code) to be shared. By doing this, it promotes the development of robot software [2]. ACRO has a six-DOF Panasonic robotic manipulator. This robotic arm, depicted in figure 1.1, was used in a previous research project as an automated fruit picking robot (AFPM) [3]. The Panasonic robot has its own controller using the Panasonic programming language.

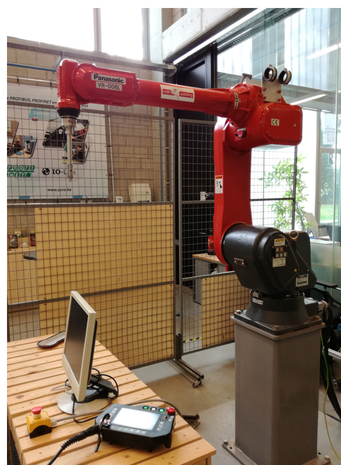



**FIGURE 1.1:** The setup of the Panasonic VR-006L manipulator at ACRO, Diepenbeek

## 1.2 Problem description

Worldwide, the number of industrial robots that is being deployed in different sectors is on the rise. This indicates the importance of the development of industrial robots. This growth and development is important because more and more industries have an acute labour shortage and a staggering cost related to interruptions of production. Figure 1.2 shows the estimated annual supply of industrial robots at year-end by industries worldwide.

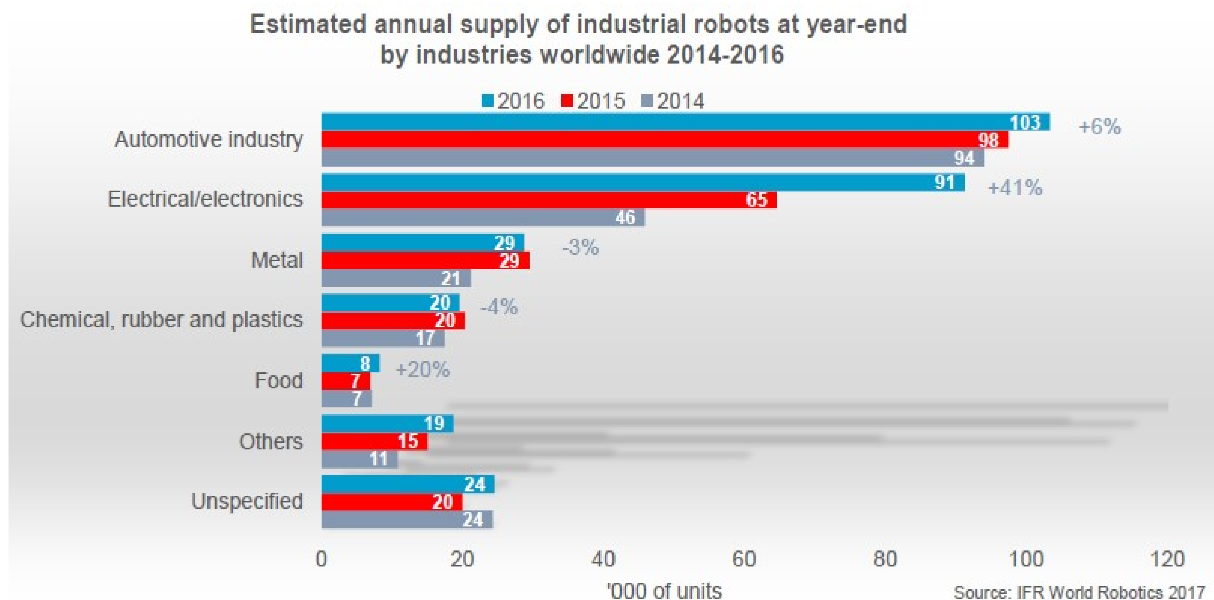



**FIGURE 1.2:** The annual supply of industrial robots at year-end by industries worldwide, adopted from [4].

In the figure above it is notable that the difference between the number of deployed robots per sector remains more or less constant. It is also notable that in sectors like food and some other applications, industrial robots are less commonly used. This can possibly be explained by a shortage of flexibility. More flexibility would mean that industrial robots become easier to install and deploy.

Because of this, sectors which now use industrial robots less often could use them easier and cheaper. Improvements in soft robotics, intuitive programming and the development of better sensors can strongly improve the flexibility of the deployment of industrial robots. Also, with the help of ROS, more flexibility can be achieved because open-source software packages for different purposes can be developed and shared [5].

At the moment, ACRO is doing research on the use of path planning tools (with the integration of ROS among other things) on industrial robotic manipulators. Such a path planning tool allows the user of the robot to make the robot move along a pre-calculated path. The path planning software will first simulate this path to test if it's even possible to execute before it is send to the robot. While the robot is moving, the path planning software requests the state of the robot to check if it's executing the path correctly. A path planning tool is very interesting to apply on industrial robotic arms because a reduction in time and operating cost can be achieved. Standard industrial robotic manipulators currently have often no software shipped with them to plan a path offline for a specific

application. There does however exist software to program robotic arms, but applying extra features and improvements to this commercial software is proven to be difficult. This is because of the fact that commercial software is often limited in use. There does however exist commercial available software to plan an offline path, but testing new research topics like the *Descartes algorithm* [6] on a industrial manipulator can not be done using commercial software. That's why the use of open-source resources is suitable for these purposes.

The six-DOF Panasonic robot has never been controlled using ROS. For this, a ROS driver is required. This driver is interesting to develop, since ROS is an open-source framework in which different libraries can be used. Examples of such libraries are, among other things, facial recognition, stereovision, and sofisticated path planning tools.

## 1.3 Goals

The central goal of this master thesis is the development of a ROS driver that makes the translation between a path planning tool, ROS and the Panasonic robot controller. Figure 1.3 shows the relations between the most important ROS nodes used in this thesis. A node is an independent programme within ROS that has a certain computational function. The translation (displayed in green) is only a part of this diagram. It makes it possible to translate commands, from a laptop with ROS and a path planning tool, to commands the computer on the robot can understand and execute.

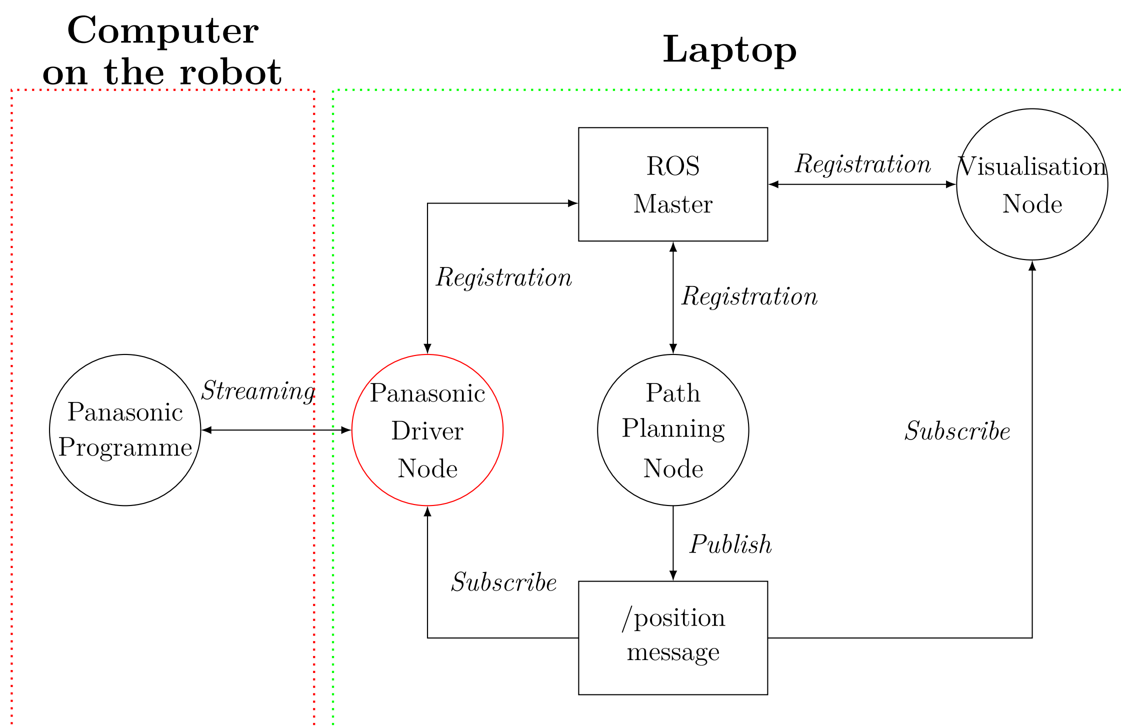



**FIGURE 1.3:** The relationship between the different ROS nodes used in this master's thesis. The circle displayed in red is the Panasonic driver node.

To accomplish the central goal (to implement a translation into a working solution) different nodes have to work together. A ROS driver can be seen as a simple node. This translation has to be implemented to be as user-friendly as possible, and it has to be in accordance to the requirements drawn up by the ROS community. First and foremost, the ROS driver has to be aware of the actual state of the six-DOF Panasonic robotic arm. Also, by knowing the state of the robot arm, the driver can know when an emergency stop is pressed. According to the ROS community, requesting the state has to happen with a minimum frequency of 1 Hz. Besides, ROS has to be informed of the geometric and kinematic model of the robot. Using these models, ROS can plan collision-free paths. Above all, the driver needs to translate the generated collision-free paths to movements which can be executed by the robot. Finally, the driver has to be able to receive error-messages and report them to the user [7].

## 1.4 Approach

To accomplish the goals described above, the following steps are taken. First of all, it is clear that a general knowledge needs to be build around the main subject. This happens in the literature study, which consists of two parts. On one side, there are tutorials to learn ROS. The ROS tutorials are important in the beginning stage of the master's thesis for the basics and later for expansion of the knowledge. On the other side, there are tutorials about Python and C++ to be learned. This is needed to know how to write most of the scripts. This will be necessary to learn in later stages.

Within the category of ROS tutorials, aside from learning general ROS workings, there has been put a lot of attention to currently published ROS drivers. ACRO has a robotic arm of the brand *Universal Robots* of which a ROS driver is currently available. This has been tested extensively to acquire more insight in the practical working and control of a robotic manipulator with ROS. Also, the driver itself has been studied. Aside from this driver, there are other ROS drivers available from other robot manufacturers. These drivers were being compared to look for how these were build and how the driver of the Panasonic robot eventually could be established.

A second part of the master's thesis is the translation between ROS and the Panasonic robotic arm. A first step to accomplish this is to make a geometric and kinematic model of the robotic arm. This model is necessary since ROS has to be aware of this to generate collision-free paths which are imposed by the path planning tool. To check if ROS is indeed aware of this model, it can be simulated in a visualisation package within ROS. A second step is to provide the communication between ROS and the robot controller of the Panasonic robotic arm. This can be checked by sending several messages from ROS to the robot. These messages can be, for example, moving from one pose to another desired pose. Next, it is necessary that this communication runs smoothly. That's why it has to comply with the requirements of the ROS community (see chapter 2). Aside from this, it is also important that the realised ROS driver will be tested and improved where needed.

## 1.5 Contributions of this work

This thesis has established a kinematic and geometric robot model of a Panasonic VR-006L manipulator in the shape of a URDF file. This model can be used in ROS for applications like path planning, navigation, collision detection and so on. Furthermore, this thesis has provided a conversion programme which translates ROS commands in a csr-file. A csr-file contains all the information of the robot programme in a Panasonic-specific format. This file can be executed on the robot using the Panasonic's DTPS software. Last but not least, an online ROS driver with a position streaming interface has been established using an open implementation of the PROFIBUS protocol as fieldbus between ROS and the robot controller. A Cartesian position and orientation can be send to the robot controller as well as a specified movement type. The robot send its Cartesian position back to the robot if the desired position is reached.

## 1.6 Chapter preview

Figure 1.4 gives a structural overview of the following chapters.

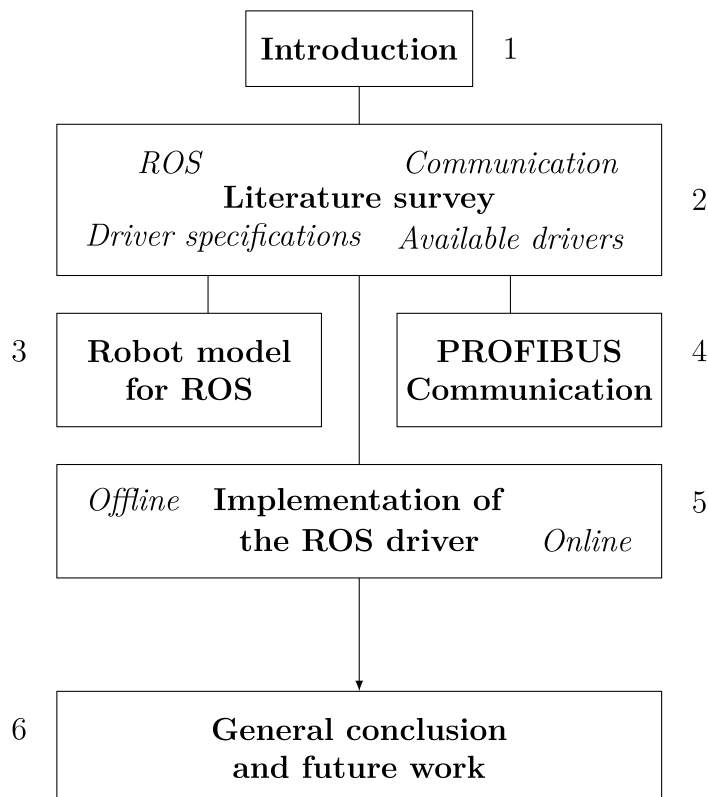



**FIGURE 1.4:** A structural overview of the following chapters

First, chapter 2 briefly surveys the range of research domains which are related to our work. These domains are the ROS environment, the driver specifications, the publicly available drivers for industrial manipulators and the communication with ROS and the robot controller. Then chapter 3 explains in more detail the robot model of the Panasonic VR-006L manipulator in ROS. Chapter 4 explains the general PROFIBUS communication.

Chapter 5 builds further on chapter 4 to implement an open-source PROFIBUS communication with Linux as operating system (OS). Finally, chapter 6 concludes this master's thesis. In addition, a Dutch article has been added in the appendix that summarizes most parts of the thesis.

# Chapter 2

# Literature survey

## 2.1 Introduction

The first step to succeed in creating a driver for a robotic manipulator, is to take a look at the existing literature about this topic. Considering that ROS is an open-source framework (see section 2.2), finding interesting information is convenient. However, there is a lack of available literature that has been published in robotic journals.

This chapter discusses mainly the software that was used in this master's thesis. First of all, the main framework that has been used during this thesis is explained. This chapter also describes the publicly available drivers that already have been developed. These drivers will be compared to see if there are some interesting features that can be used to create a ROS driver for the Panasonic manipulator. Finally, the communication will be discussed. This communication between robot controller and ROS is crucial for the efficiency of the driver and will determine the difference between a working driver and a non-working driver.

## 2.2 Robot Operating System (ROS)

In the past, writing robot software was difficult. This was partially due to a large variety of different industrial robots. These manipulators were often installed for industrial use but they were also introduced in the healthcare sector. Because of this large scope of different robotics, the variety of hardware was enormous. Robots were rarely equipped with the same common hardware features. These hardware features were specific to robot manufacturers. Therefore writing robot software was often limited to one single robot. The reuse of code was also limited [5]. To address this issue, a collaboration between Stanford University and Willow Garage[1] was engaged in 2008. Their common aim was to create a framework that can be used to create/write robot software which can be reused.

[1] Willow Garage was a robotics research institute situated in California, USA. Their aim was to develop specific hardware and open-source software for personal robotics applications.

As robotic systems increase in growth, the need of a framework, such as described above, is required. The solution to this problem was the **Robot Operating System**, often abbreviated by **ROS**. As discussed earlier, ROS is an open-source framework for creating robot software that provides a large amount of libraries, tools and conventions to make the development of complex robot software easier. ROS can be seen as an operating system that works on top of a host operating system (Host OS). This host operating system can be Linux or any other Unix-based operating system like Apple Macintosh. In practice, Linux Ubuntu is the most used host operating system to run ROS on [8].

As described by the developers in [9], ROS can be recapitulated in five main goals. First of all, the ROS framework is considered as a *peer-to-peer* network. This can be seen as a process where subprocesses are reviewed by a master process. Second, ROS is *multi-lingual*. ROS is familiar with a group of programming languages[2], for example C++ and Python. ROS also supports XML-based programming, which is for example used to visualize a robot (see further). The third goal is a *tool-based* approach. ROS can be explained as a machine with various tasks that collaborate together, instead of a central processing unit. The fourth goal is defined as *thin*. This means that code reuse can easily be done. Last but not least, the goal of ROS is to be *free and open-source*. This means that full source code is publicly available. Also any information can be found on the wiki site of ROS. Developmental issues can also be consulted online.

### 2.2.1 ROS features

This subsection discusses the features of ROS. After reading this subsection, the general working of ROS will be clear.

**Installation** The first thing to do before using ROS is to install it. But first a host operating system is necessary. Most of the entire ROS community will recommend Linux Ubuntu as host operating system. This is also because Linux is free and open-source like ROS, but also because of the fact that a Unix-based operating system is more stable for developing software. For our master's thesis, Linux Ubuntu 14.04.5 LTS (Trusty Tahr) is used, because a LTS version is the most stable version of Ubuntu.

Like Linux Ubuntu, ROS has also a few distributions. A distribution is a set of ROS packages that are particular to that version. Every year in May, ROS wil be updated with a new distribution. The first version of ROS was released back in 2010 as *ROS Box Turtle*. But this ROS distribution depends on the host operating system. To use the newer ROS distribution, an update of the host OS is needed. In our case, we will be using ROS Indigo Igloo, depicted in figure 2.1.

[2]On the moment, ROS supports four programming languages: C++, Python, Octave and LISP.

**FIGURE 2.1:** Logo of ROS Indigo Igloo, released on July 22nd, 2014. The figure is taken from [10].

The installation of the host operating system can be found online. In case of Ubuntu, this can be performed by a dual boot with for example Windows 10. When Ubuntu is installed, ROS can be installed from the terminal. This is well documented in [11].

**ROS Nodes** As discussed earlier in the beginning of section 2.2, ROS has a *peer-to-peer* goal. This means that the major functionality of ROS is divided into a number of entities that communicate with each other. Such an entity is called a ROS node. A ROS node is an independent programme within ROS that computes, calculates or processes and sends information to other entities within ROS. A programme in ROS usually consists of a few nodes working together to achieve a purpose. A ROS driver for example is such a node (see later) [12].

**ROS Master** To direct the different nodes and make them work properly together, a leader is needed. This leadership is offered by the ROS Master.

**ROS Topics** The communication between different nodes and the ROS Master is established by ROS Topics. These topics can be compared to a bus system in a programmable logic controller (PLC). Over these bus systems, information can be transferred to other entities in the ROS environment.

**ROS Messages** The information streams, that are transferred between entities in the ROS environment, are called ROS messages. Nodes communicate with each other by publishing and subscribing to topics. A message can be seen as a simple data structure. Based on the data type of the ROS message, nodes will know which information is transferred.

These four features are illustrated in figure 2.2.

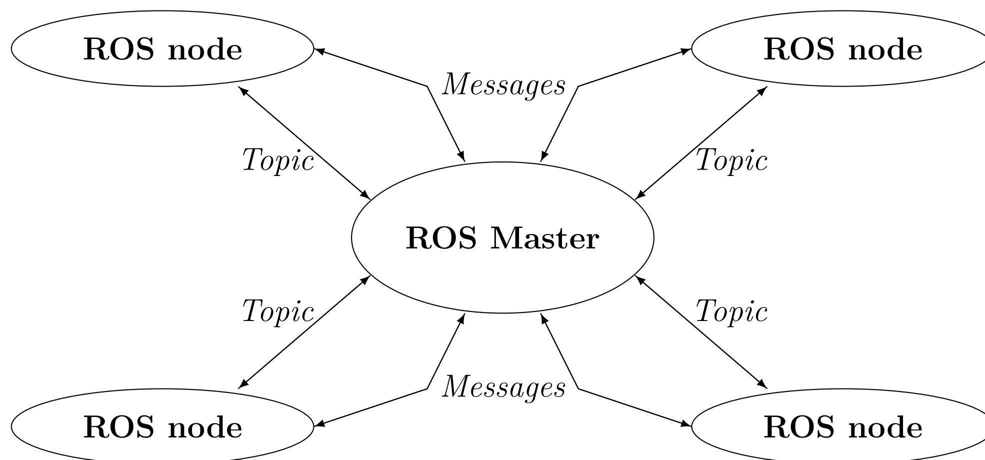



**FIGURE 2.2:** Visualisation of the ROS features as explained above

Figure 2.2 displays the functionality between ROS nodes, the ROS master and the communication via ROS messages through ROS topics as discussed earlier.

**ROSlaunch** To make a single node work, ROS has provided a simple command to make this happen. This command is called **rosrun**. With *rosrun* the developer can run an executable in a package he has created. This executable then becomes one single node. If the user/developer wants to run multiple nodes at the same time, he can use the command *ROSlaunch*. Many packages come with a so called *launch file*. This is a file, expressed in *Extensible Markup Language* (XML), which brings up the selected nodes within the launch file. The syntax of launching a launch file is:

**roslaunch** *package_name file.launch*

This command launches the file with a .launch extension in a certain package [13].

**Catkin** To compile a written piece of software, ROS has provided a build system. A build system is a kind of compiler and it is responsible for making raw source code more interpretable for an end user. Older versions of ROS (pre-Groovy) contained a build system which was called **ROSbuild**. Newer versions of ROS, from Fuerte till Melodic, have got a build system called **Catkin**. For more information about these two building systems, we refer to the wiki page of ROS [14].

**URDF** To make a robot model which can be viewed for instance on a laptop, ROS uses a tool which is called a *Unified Robot Description Format* (URDF). This is a XML format that the describes the kinematic and geometric relations of links and joints of a robot/manipulator. Another representation that is often used to model a robot, is a **SRDF** file. A *Semantic Robot Description Format* is like the URDF a XML file with information about the robot which is not used in the URDF file. In this SRDF file the semantic information about the robot structure is displayed [15]. To check if the created URDF file is correct, ROS has got a package called *liburdfdom-tools* where the examination

of the URDF file will be performed. When creating URDF files in XML language, these files can easily be a certain hundred of lines tall. To clean up and structure these files, ROS has got a tool that is called **Xacro**. With this tool, macros can be used to refer to larger parts of URDF code [16], [17].

**RViz** To make the robot model, described in a URDF, visible, ROS has provided a programme called *ROS Visualiser* or RViz. With RViz the visualisation of any robot model is possible. Given any correct URDF model in XML format, RViz turns it into a 3D-visible robot model [18].

**MoveIt!** To give a certain command/movement to a manipulator, often path planning algorithms need to find collision-free trajectories. This can be handled by MoveIt!. MoveIt! is an open-source path planning software to generate collision-free paths from one state (point in space) to another desirable state. It is provided with an easy-to-handle platform for developers of complex robot applications [19]. More information is given in section 3.4.

### 2.2.2 ROS-Industrial

ROS-Industrial, often abbreviated by ROS-I, is a community of ROS which contains drivers, libraries and packages for industrial robot hardware. This hardware can be anything like industrial manipulators, pick-and-place installations, industrial sensors, etc. The interesting thing about the ROS-I community is the fact that all these libraries and tools are open-source available on the internet. Any package created by a ROS developer can be reused because every decent and reliable package or library, which is put open-source on the internet, receives an open-source software licence[3].

The goals of the ROS-I community can be summarized in a few points:

- create a supported community;
- create robot software with a certain reliability and robustness;
- create an easy path to any industrial problem.

The ROS-I community has got a Google Groups forum[4] where developers can ask questions or support to the entire community if there is any problem for example with the development of an industrial manipulator driver [20].

[3] In the open-source world, there are many licences that are related to open-source software. The most known is a BSD license. BSD stands for *Berkeley Software Distribution*. In the past, these licenses were given to use open-source code from the Berkeley University in California. In the present, this license is given to every open-source software project which can be reused legally by a third party. On the other hand, companies for example can develop in house software based on ROS-Industrial packages without the obligation to publish the results as open-source code. BSD-licences are most commonly used, but licenses like Apache licenses and MIT licenses are also applied.

[4] In March 2018, this mailing list was moved to `discourse.ros.org` .

## 2.3 Specifications of a ROS-Industrial driver for industrial robots

Over the past few years ROS has developed a consortium where libraries, tools and drivers for industrial hardware are implemented. This shared ROS community is called ROS-Industrial, as discussed earlier in subsection 2.2.2 . The main goal is to create a community supported by robotics researchers and professionals. Besides this, ROS-Industrial is also founded to stimulate the development of robust, reliable and open-source robot software to explore new ideas with robotic hardware [20].

Together with ROS-Industrial, the development of drivers for industrial manipulators comes along. Such a driver makes the translation between commands, obtained and generated in/by ROS, and commands used by the robot controller to make the robot move, and vice versa. To make the development of the interface between ROS and the robot controller standardized, the ROS-Industrial community has come up with a few guidelines[5] when writing a driver for an industrial manipulator. These rules are summarized below [7].

### 2.3.1 Operation

The first guideline describes the reaction of the robot to high level activities. These activities are the initialisation of the robot and the communication between robot and ROS.

#### Initialisation

When connecting to a robot controller, a ROS node should initialize the connection with the controller. To make this communication work, ROS-Industrial suggest to run a service on the robot controller which automatically reacts when the controller is powered on. This service is for instance running on the background.

#### Communication

To develop a robust driver, the robot ROS connection should be prepared to handle communication loss scenarios. When any form of communication loss happens, the ROS node is obligated to reconnect with the robot. The interval for this reconnection is set at a minimum frequency of 1 Hz. Also when communication is failed, the node shall continue with publishing status messages with *connected=false*. Because of the loss of communication, the robot shall stop moving and will be powered of until new communication is set.

[5] Robot developers and researchers aren't obligated to follow the standards imposed by the ROS-Industrial community since the developed software is open-source. On the other side, each online published driver is evaluated by ROS-Industrial. The quality of the driver is then assigned with a software status. This status can be 'EXPERIMENTAL', 'DEVELOPMENTAL' or 'PRODUCTION'. So following the guidelines before making a driver publicly accessable is advisable.

### 2.3.2 ROS API

When trying to connect to the robot, ROS-Industrial has come up with a few messages which define this connection. This is done by an *Application Programming Interface* (API). In Linux Ubuntu, this API can be shown in the terminal when typing a command. The messages that will be showed are for example the IP address of the robot (when working with an Ethernet connection) and the description of the robot (URDF).

Besides this elementary information, the ROS API should also return a *feedback status* message (where feedback of the desired position and the current position is given), *joint status* messages (where feedback of the desired and current position of the robot's joints is given) and a *robot status* message (where the status of critical robot parameters is given).

A third feature which is captured in the ROS API, is the *motion control*. This node implements methods to control the robot's movements. This node subscribes two topics: the *joint path command* which executes a pre-examined joint trajectory on the robot and the *joint command* which executes dynamic motion by streaming real-time joint commands. Besides the subscription on these topics, the node also runs two services: a *stop motion* service, which stops the current robot motion, and a *joint path command* service, which executes a new trajectory on the robot. More information about the specification of industrial manipulator drivers can be found on the wiki page of ROS-Industrial [7].

## 2.4 Characteristics of available ROS drivers for industrial robots

A second part of the literature study is the study of the available ROS drivers. It is interesting to compare them with each other based on some imposed criteria. These criteria can for example be the manner in which the communication between ROS and the robot controller is set up or if there is already an available MoveIt! package to control the robot with a path planning simulator. To make a comparison, table 2.1 is used [21], [22].

**TABLE 2.1:** Comparison between the available ROS drivers for industrial manipulators

| Robot manufacturer | Controller(s) | Position Streaming | Trajectory Downloading | Trajectory Streaming | Torque Control | IO Control | Manipulator type | MoveIt! Package |
|---|---|---|---|---|---|---|---|---|
| **ABB** | IRC5 | NO | YES | NO | NO | NO | IRB-2400 | YES |
| | | | | | | | IRB5400 | NO |
| **Adept** | CX, CS | YES | NO | NO | NO | NO | Viper 650 | NO |
| **Fanuc** | R-30iA/ R-30iB | YES | NO | NO | NO | NO | LR Mate 200iC (all) | YES |
| | | | | | | | LR Mate 200iD | YES |
| | | | | | | | M-10iA | YES |
| | | | | | | | M-16iB/20 | YES |
| | | | | | | | M-20iA(/10L) | YES |
| | | | | | | | M-430iA(/2F, 2P) | YES |
| | | | | | | | M-900iA/260L | NO |
| | | | | | | | Other | NO |
| **Motoman** | DX100 | NO | NO | YES | NO | YES | SIA10D/F | NO |
| | FS100 | NO | NO | YES | NO | YES | SIA20D/F | YES |
| | DX200 | NO | NO | YES | NO | YES | MH5F | YES |
| | YRC1000 | NO | NO | YES | NO | YES | SDA10F | Yes |
| | | | | | | | Other | NO |
| **Universal Robots** | CB2/CB3 | YES | NO | NO | NO | YES | UR5 | YES |
| | | | | | | | UR10 | YES |
| **Epson**[6] | RC620 | YES | NO | NO | NO | NO | C3-A601S | YES |
| **Kuka**[7] | KR C4 | YES | NO | NO | NO | NO | KR 5,6,10,16 | YES |
| | | | | | | | KR 120, 210 | YES |
| | | | | | | | IIWA | YES |

[6] The Epson driver (developed in a previous master's thesis at ACRO) is in an experimental stage.
[7] The Kuka driver is in a developmental stage.

Table 2.1 compares the different robotic manipulators based on a few parameters. These parameters are *Position Streaming*, *Trajectory Downloading*, *Trajectory Streaming*, *Torque Control*, *IO Control* and the presence of a *MoveIt! Package*. The next paragraphs will explain the difference between these parameters.

#### Position Streaming

A ROS driver belongs to the category *Position Streaming* when commands, generated by ROS, are converted in commands that can be interpreted by the robot controller. This command is an imposed 3D point that the robot has to achieve. This point is streamed to the robot in real-time. The robot controller decides when and how quick these commands are converted to the robot itself.

#### Trajectory Downloading

When all the ROS commands are transferred to the controller and when all these commands are processed by the controller, the interface is called *Trajectory Downloading*. The generated ROS command sends a series of points that form a trajectory to the robot controller. This interface technique is often used when *Position Streaming* is not possible to implement. Since it has no real-time action, *Trajectory Downloading* is slower than *Position Streaming*.

#### Trajectory Streaming

The functionality of *Trajectory Streaming* is comparable to the *Position Streaming* interface combined with the *Trajectory Downloading* interface. ROS generates a command that streams a trajectory to the robot controller. It has basically the same function as the two combined interfaces, but the difference is that when *Trajectory Streaming* is used, the interface will also take into account the velocities and accelerations of the robot's joints. These velocities and accelerations are adhered to by the robot controller.

#### Torque Control

Some ROS drivers come up with a feature called *Torque Control*. This feature is used to control the torques acting the robot's joints. When commanding a movement, often great accelerations are involved. With large accelerations come large forces. When these forces act a long way from the current lever, large torques will act on the robot. To limit these torques on the robot, some drivers have a *Torque Control* interface implemented in their software. The ROS-Industrial community does not yet support this interface.

#### IO Control

Like the *Torque Control* feature, there also exist an interface which activates inputs and outputs on the robot through ROS. This interface is called *IO Control*. The ROS-Industrial community does not yet support this interface either.

#### MoveIt!-Package

To make path planning possible, ROS has a cooperation with MoveIt! [19]. MoveIt! is a path planning software which calculates collision-free trajectories to execute a path. To make MoveIt! and ROS work together, a MoveIt! configuration package is needed. This can be done with the *MoveIt! Setup Assistant* (see section 3.4) tool in ROS which needs the kinematic and geometric model of the robot (URDF) itself.

#### Comparison

Based on the parameters/criteria as described above, it is useful to compare each driver with each other. The thesis will take a look at the ROS drivers of the Motoman robot, the Universal Robots UR5, the Epson robot and the Kuka robot.

The drivers of the Universal Robots, Kuka and Epson robot are pretty similar. These drivers work with the *Position Streaming* interface as described in the previous paragraphs. The Universal Robots manipulator though has also got an *IO-control* feature. The Motoman driver doesn't have a *Position Streaming* interface like the other ones. This driver works with a *Trajectory Streaming* interface.

The interface between the robot controller and ROS for all four drivers happens through the TCP/IP protocol which means that the connection is established through an Ethernet connection. The Universal Robots UR5/10 works in real-time. Real-time as to be taken relatively. The robot controllers of industrial manipulators mostly have a control loop that controls the robot with a frequency of 1 kHz. This is much faster than the minimum control frequency of 1 Hz, imposed by the ROS-Industrial community. Often, the ROS driver of an industrial robot is established through a client-server connection. In this case there is a server programmed on the robot controller which sets up the communication between the robot and ROS. This communication server is written[8] in the programming language of the robot.

## 2.5 Communication between ROS and robot controller

Another crucial part in this thesis is to understand how the communication between ROS and the controller on the industrial robot is established. Communication always happens through messages. The structure of the message and the method of sending it can differ.

[8] This server is only possible if the robot allows it.

In the following subsection, the general ROS communication concepts will be explained. After that, the communication practices in different industrial robots will be looked at. Then, the commonly used simple message protocol will be treated.

### 2.5.1 General ROS communication concepts

In general, ROS can communicate via messages with the help of the concepts described below [23].

**Topics** are subjects (i.e. buses) over which nodes exchange messages through publish and subscribe relations. They should be used for continuous unidirectional data streams (sensor data, robot state, ...).

**Services** are used to send and receive messages between nodes using a request message and a response message. A client node calls the service by sending the request message and awaiting the response message. Services are used when a task should instantaneous be executed for example when doing a quick calculation such as Inverse Kinematics (IK). Services should never be used for longer running processes.

**Actions** are used when services take a long time to execute. The most important property of actions is that they can be pre-empted, i.e. cancel the request during execution. Pre-emption should always be implemented cleanly by action servers.

To implement actions, there exists a library in ROS called *actionlib* that provides tools to create servers that execute long-running goals that can be pre-empted.

### 2.5.2 Simple Message Protocol

The Simple Message Protocol defines the message structure between the ROS driver and the industrial robot controller [24]. The protocol meets the following requirements:

- the format should be simple enough that code can be shared between ROS and the controller;
- the format should allow for data streaming (ROS topic like);
- the format should allow for data reply (ROS service like);
- the protocol is not intended to encapsulate version information. It is up to individual developers to ensure that code developed for communicating platforms does not have any version conflicts.

The structure of a simple message looks as follows:

- PREFIX (not considered as part of the message)
  - LENGTH as an integer (HEADER + DATA) in bytes.

- HEADER
  - MSG_TYPE: identifies type of message (standard and robot specific values);
  - COMM_TYPE (integer): identified communication type;
  - REPLY_CODE (integer): reply code (only valid in service replies).
- BODY
  - DATA variable length data determined by message type and communication type.

The message protocol allows for an arbitrary data payload for message and communications types. However, the client-server model requires that both sides understand the data payload associated with the different message and communications types. The typed message class enforces the data payload structure. The typed message base class provides methods for creating topic, reply and request messages. If used in both, the client and server, the developer doesn't need to understand the structure of the data payload. Unfortunately, a typical robot controller cannot use C++ classes and thus the developer must understand the message protocol and payload data structure in order to parse it on the robot controller side. The documentation on message specific structures can be found in the source header files [24].

The three commonly used Simple Message types are the *Joint_Position* type, the *Status* type and the *Joint_Trajectory_Point* type. The *Joint_Position* Simple Message contains the actual joint orientation of the robot. Figure 2.3 shows an example of such a message.

| Hex | Field | Description |
|---|---|---|
| | Prefix | |
| 00000038 | length | 56 bytes |
| | Header | |
| 0000000A | msg_type | Joint Position |
| 00000001 | comm_type | Topic |
| 00000000 | reply_code | Unused / Invalid |
| | Body | |
| 00000000 | sequence | 0 (unused) |
| B81AD9FA | joint_data[0] | -0.000036919 |
| B6836312 | joint_data[1] | -0.000003916 |
| B7C043F5 | joint_data[2] | -0.000022920 |
| B8B81516 | joint_data[3] | -0.000087777 |
| B865D055 | joint_data[4] | -0.000054792 |
| B8B6365E | joint_data[5] | -0.000086886 |
| 00000000 | joint_data[6] | 0.000000000 |
| 00000000 | joint_data[7] | 0.000000000 |
| 00000000 | joint_data[8] | 0.000000000 |
| 00000000 | joint_data[9] | 0.000000000 |

**FIGURE 2.3:** Example of a Joint_Position Simple Message

As shown in figure 2.3, the message type has a value of $0000000A_{HEX}$ which is a value of ten in decimal notation. When the MSG_TYPE of the message has a value of ten, it indicates that this message is a *Joint_Position* message. From figure 2.3 it is also clear that this message type is written to ROS as a Topic (the COMM_TYPE has a value of $1_{HEX}$). Also the REPLY_CODE of this message is empty, which means that no reply is sent to the ROS environment. The BODY of this Simple Message shows us the orientation of the six joints of the robot, represented in hexadecimal value. For example, the orientation of joint 1, i.e. *joint_data*[0], shows -0.000036919 rad or -0.002115° . For industrial applications, this is accurate enough. The other four values of the array are set to zero, because a six-DOF manipulator does only have six joints.

Figure 2.4 below shows an example of a *Joint_Trajectory_Point* Simple Message which sets up a communication with the robot controller to control the robot through ROS.

| Hex | Field | Description |
|---|---|---|
| | Prefix | |
| 00000040 | length | 64 bytes |
| | Header | |
| 0000000B | msg_type | Joint Trajectory Point |
| 00000002 | comm_type | Service Request |
| 00000000 | reply_code | Unused / Invalid |
| | Body | |
| 00000001 | sequence | 1 (second TrajectoryPoint) |
| A7600000 | joint_data[0] | -0.000000000 |
| 3EA7CDE8 | joint_data[1] | 0.327742815 |
| BF5D9E57 | joint_data[2] | -0.865697324 |
| C0490FDB | joint_data[3] | -3.141592741 |
| 3F34815F | joint_data[4] | 0.705099046 |
| C0490FDB | joint_data[5] | -3.141592741 |
| 00000000 | joint_data[6] | 0.000000000 |
| 00000000 | joint_data[7] | 0.000000000 |
| 00000000 | joint_data[8] | 0.000000000 |
| 00000000 | joint_data[9] | 0.000000000 |
| 3DCCCCCD | velocity | 0.1 |
| 40A00000 | duration | 5.0 |

**FIGURE 2.4:** Example of a Joint_Trajectory_Point Simple Message

As can be seen in figure 2.4, the MSG_TYPE shows us a value of $0000000B_{HEX}$ which indicates a *Joint_Trajectory_Point* Message. The communication with ROS happens through a service. This message sends coordinates, generated in ROS, to the robot controller which powers on the motors of the robot to put the robot in the desired pose. Remarkable are the two extra forwarded data messages to the controller which are the velocity and the duration of the movement. These two control signals are very important, because with these two parameters the acceleration of the joints can be calculated by taking the derivative of the speed with respect to time. These accelerations are important because with large accelerations come along large torques. To achieve these large torques, a large current has to be send to the motors, which can often be destructive.

A final example of a Simple Message is shown in figure 2.5.

```
Hex        Field              Description

           Prefix
00000028     length           40 bytes

           Header
0000000D     msg_type         Status
00000001     comm_type        Topic
00000000     reply_code       Unused / Invalid

           Body
00000001     drives_powered   True
FFFFFFFF     e_stopped        Unknown
00000000     error_code       0
00000000     in_error         False
00000000     in_motion        False
00000002     mode             Auto
00000001     motion_possible  True
```

**FIGURE 2.5:** Example of a Status Simple Message

This type of Simple Message lets the ROS environment know in which state the robot is at the moment. Often, this message is send with a frequency ten times lower than the frequency of the *Joint_Position* Message. As the HEADER of this Simple Message says, the communication with ROS is established by a Topic. This message also shares important information of the robot with the developer. Figure 2.5 for example displays a message which tells if the drives of the robot are powered, if the robot is in an error state or sends a signal if the motion of the robot is possible [25].

### 2.5.3 Industrial Robot Client

**Overview** The *Industrial_Robot_Client* package provides a standardized interface for controlling industrial robots based on the ROS-Industrial Specifications [26]. This package contains an *Industrial_Robot_Client* library which is useful for robot-specific implementations to reuse code from this library to avoid much of the copy/paste duplication in current industrial robot driver implementations. The package also uses the *Simple_Message* protocol to communicate with a compatible server running on a standalone industrial robot controller using an Ethernet communication.

**Modification for Robot-Specific Implementations** The differences between manufacturer interfaces and robot designs will require for robot-specific control code. These differences can be found in the joint coupling, velocity scaling and communication protocols. Therefore, changes have to be made to the basic reference implementation provided in the *Industrial_Robot_Clientt* library [26].

## 2.6 Conclusion

In this chapter, the ROS basic features were reviewed. It should be clear as to why ROS is a useful framework for building robot specific software. Also, the specifications of Industrial ROS drivers have been described. Next, different existing ROS drivers have been compared by different parameters. This is useful to know what the possibilities or limitations of a new ROS driver implementation could be.

At last, general ROS communication concepts are discussed. This should give an idea on how the communication, between ROS and the robot, has to be set up. These communication principles, as explored in the literature survey, were mainly based on the TCP/IP and Ethernet principles. Since every robot and robot controller, as described in table 2.1, has an Ethernet communication, more information has been given about using this fieldbus in a ROS driver. However, the Panasonic controller has not been equipped with an Ethernet connection but with a communication card where a PROFIBUS connection can be established. Since there hasn't been a ROS driver yet where a PROFIBUS connection is used, other open-source resources need to be implemented. The principles of PROFIBUS is discussed in chapter 4.

# Chapter 3

# Panasonic VR-006L robot model for ROS

## 3.1 Introduction

This chapter discusses the visualisation of the Panasonic VR-006L manipulator in a simulation environment. To display the manipulator in a simulation, ROS has provided a tool called RViz (see chapter 2). To visualise the robot in RViz the URDF file of the robot has to be created. After that, the robot can be displayed in RViz, using a launch file which starts the RViz node.

The following chapter is structured as follows. First, the principles of making a URDF file are explained. Next, the robot will be visualized in RViz. Finally, the URDF file is converted into a SRDF file which can be used to generate trajectories for path-planning using MoveIt!.

## 3.2 Principles of a Unified Robot Description Format (URDF)

As mentioned in section 2.2.1, a URDF file describes the kinematic relations between the links and joints of the robot. These relations are important since ROS needs these when for example transferring commands to the robot controller. To create a URDF file of a robot, there are a few *rules/guidelines* for describing the visual and physical properties of the links and joints of the robot.

### 3.2.1 Description of robot links

Every serial robot has a few links and joints. In the case of the Panasonic VR-006L, which is a serial robot with six degrees of freedom (six rotations), there are six links and six joints. The description of a link is schematically depicted in figure 3.1 [27].

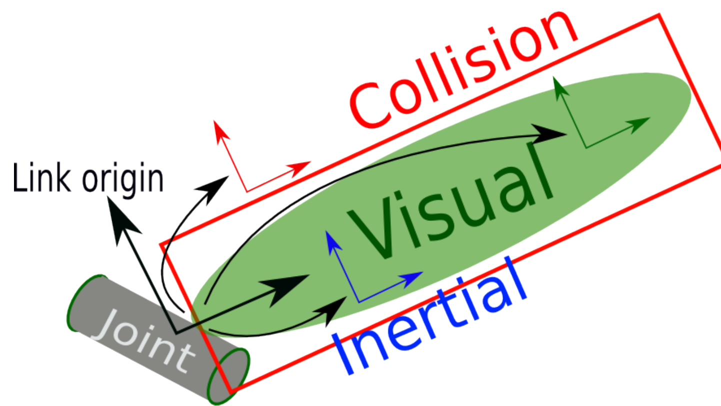



**FIGURE 3.1:** Schematic description of a link in a URDF file, taken from [27].

As can be seen in the figure above, the description can be split up into three parts:

- **Visual**. This part will determine how the link will be displayed in simulations. Often, these CAD files have a high resolution which will display a very solid visual.
- **Collision**. This part will detect the dimensions of the link and will be used for collision detection when for example planning trajectories from one point to another. To minimize the time needed for collision checking, low-resolution CAD files are used or easy 3D shapes like beams or cylinders.
- **Inertial**. This part determines the aspects of inertia of the robot's link. Properties like center of mass (CM) or moment of inertia (MOI) are often used. This part is not necessary, but adds extra information of the link.

When an exact 3D CAD file of the robot is available, it can be used to implement this in the URDF file. For the **visual** and **collision** parts of the URDF file, the origin and the geometry of the link are needed. Mostly, the origin of link $i$ is set at the origin of joint $i-1$. In addition, the color of the link can be changed using the *rgb* (red, green and blue) values and the scale of the mesh can be adjusted.

### 3.2.2 Description of robot joints

To describe a joint, a certain joint type has to be chosen. In general, six types of joints are known, which are summed up below [28].

- **Fixed**. A fixed joint is a joint where all six degrees of freedom are blocked. Because of this, this type of joint can not move or rotate in any direction.
- **Revolute**. A hinge joint that rotates along the axis and has a limited range specified by the upper and lower limits. Most robots have revolute joints. Since the Panasonic VR-006L manipulator is a robot with six rotating joints, all the joints are revolute.
- **Continuous**. A revolute joint with no upper and lower limits.

- **Prismatic**. A sliding joint that slides along the axis, and has a limited range specified by the upper and lower limits. This type of joint is used in robots where a translation is needed.
- **Floating**. A floating joint allows motion in all six degrees of freedom.
- **Planar**. A planar joint allows motion in a plane perpendicular to the axis.

When the joint type is described, a few other parameters are needed to fulfil the URDF of the joint. To illustrate these parameters, figure 3.2 is used.

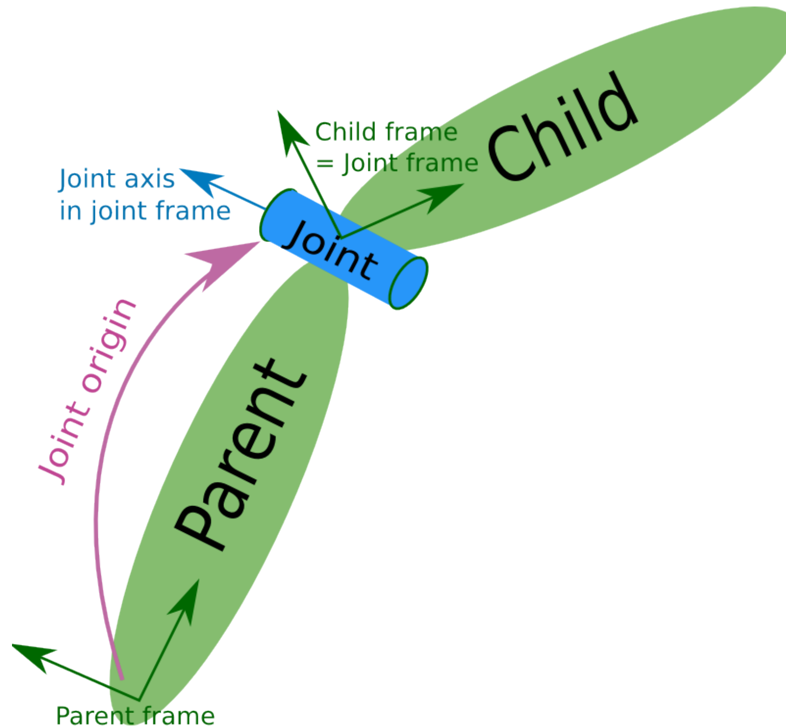



**FIGURE 3.2:** Schematic description of a joint in a URDF file, taken from [28].

The figure above describes following parameters.

- **Parent link**. This is the name of the link which is the parent of the link in the robot tree structure.
- **Child link**. The name of the link which is the child link.
- **Origin xyz**. This represents the xyz offset from the parent link to the child link. The joint $i-1$ is located at the origin of link $i$ (i.e. child link).
- **Origin rpy**. This represents the rotation around the fixed axis of a coordinate frame. R is the roll angle in radians around the X axis, p is the pitch angle in radians around the Y axis and y is the yaw angle in radians around the Z axis (cfr. Euler angles).
- **Axis xyz**. This is the joint axis specified in the joint frame. This is the axis of rotation for revolute joints, the axis of translation for prismatic joints and the surface normal for planar joints. The axis is specified in the joint frame of reference.

Fixed and floating joints do not use the axis field. It represents the xyz components of a vector. The vector should be normalized.

- **Limit lower**. This is a parameter which specifies the lower joint limit in radians for revolute joints and in meters for prismatic joints.
- **Limit upper**. This is a parameter which specifies the upper joint limit in radians for revolute joints and in meters for prismatic joints.
- **Limit effort**. This is a parameter for enforcing the maximum joint effort. This effort is expressed in $Nm$ for revolute joints and prismatic joints.
- **Limit velocity**. This is a parameter for enforcing the maximum joint velocity. This velocity is expressed in $rad/s$ for revolute joints and in $m/s$ for prismatic joints.

To build the entire kinematic chain of the robot, figure 3.3 can be used as an aid.

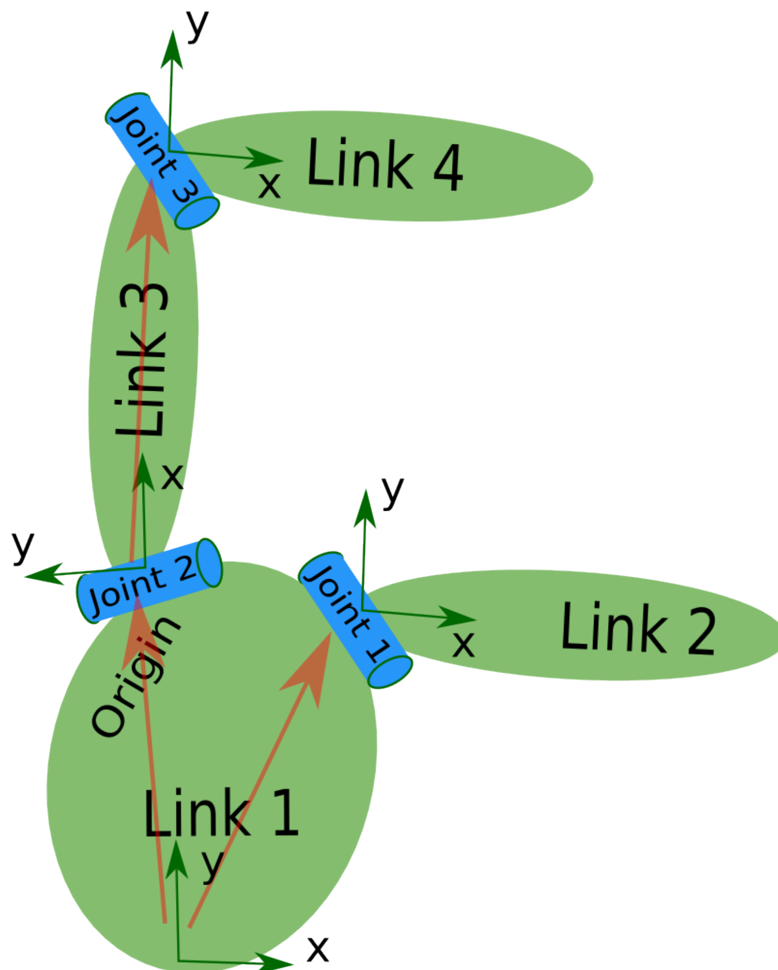



**FIGURE 3.3:** Visual aid to build the kinematic chain of a robot, taken from [16].

## 3.3 Panasonic VR-006L visualisation

When the entire kinematic chain is constructed in the URDF file, ROS has provided a tool to check if the URDF is valid or not. This tool is called *check_urdf* (within the *liburdfdom-tools* package) and can be consulted in the Linux terminal. If the URDF is validated, the kinematic chain can be viewed in a PDF file. This kinematic chain can be consulted in appendix C.

To view the robot model in RViz, a launch file has to be created which activates the RViz node, the *joint_state_publisher* and the *robot_state_publisher* node. After this is created, the robot will be visualised in RViz as in figure 3.4.

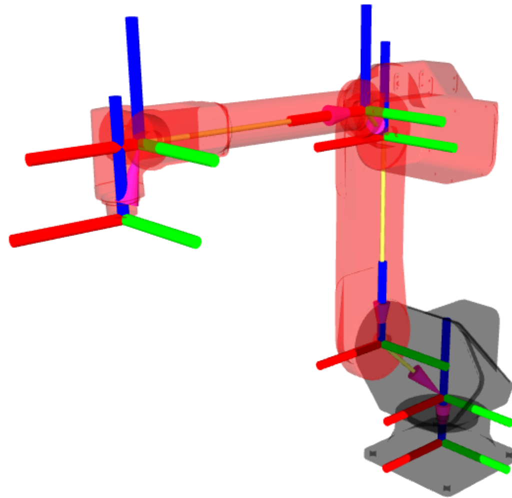
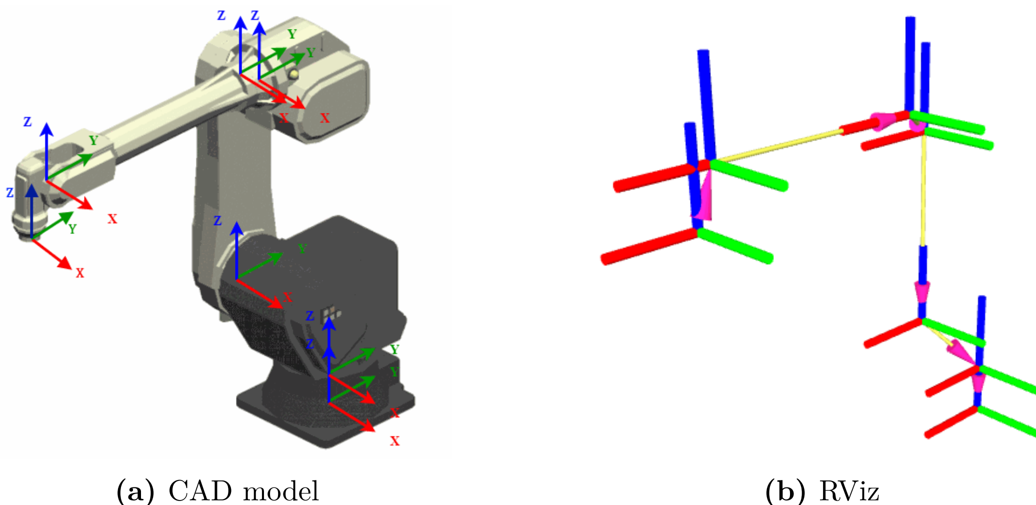


**FIGURE 3.4:** Visualisation of the Panasonic VR-006L manipulator in RViz

There are a few parameters in RViz which can be used to change the layout of the robot model. For example, the coordinate frames of each joint can be depicted on top of the model. Figure 3.5 shows the coordinate frames of the Panasonic manipulator on the CAD model (left) and without the links (right).




**(a)** CAD model **(b)** RViz

**FIGURE 3.5:** Visualisation of the coordinate frames; left on the CAD model and right in RViz

As can be seen in figure 3.5, the coordinate frames are all in the same orientation, i.e. the X, Y and Z axis of all coordinate frames are all in the same pose. ROS does not have any convention on the poses of the coordinate frames. In robotics, the Denavit-Hartenberg convention is often used, but is not used here to model the manipulator [29], [30]. Appendix D contains the Denavit-Hartenberg representation of the Panasonic VR-006L robot.

## 3.4 Path planning model for ROS and MoveIt!

To generate collision-free trajectories, a path planning tool called **MoveIt!** can be used. It will check if poses and trajectories are possible by applying inverse kinematics on the robot. Figure 3.6 gives the schematic representation of forward and inverse kinematics [31].

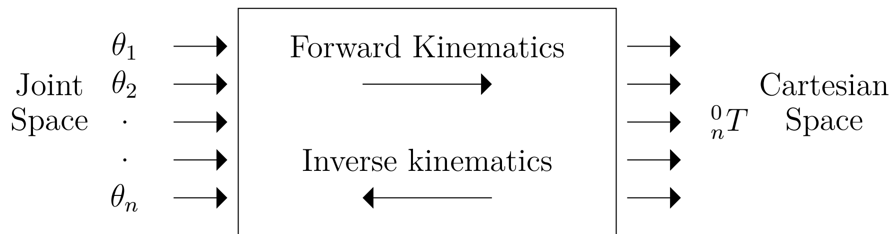



**FIGURE 3.6:** The schematic representation of forward and inverse kinematics

*Forward kinematics* (FK) calculates the pose (position and orientation) of the end effector in terms of the joint variables of a manipulator. *Inverse kinematics* (IK) calculates the joint variables given the pose of the end effector. IK is computationally more expensive because of the fact that with a given pose different joint positions can be calculated. The mathematical relationship $\boldsymbol{f}$ between the joint positions $\boldsymbol{q}= (q_1, ..., q_n)$ and the end effector poses $\boldsymbol{q}=(\vec{X}_1, ..., \vec{X}_m)$ is a nonlinear function of the joint positions and can be expressed as:

$$\boldsymbol{f} = \begin{cases} \boldsymbol{X} = \boldsymbol{g}(\boldsymbol{q}) & \text{forward kinematics} \\ \boldsymbol{q} = \boldsymbol{h}(\boldsymbol{X}) & \text{inverse kinematics} \end{cases} \tag{3.1}$$

To solve IK problems, MoveIt! uses a few IK solvers. The most popular IK solvers are IKFast, kdl (standard in MoveIt!) and TracIK. MoveIt! uses a *Semating Robot Description Format* (SRDF) from the robot model to check if trajectories on the robot will collide with objects in the environment. The SRDF file contains semantic information of the joints, links, virtual links, link pairs in collision and a possible end effector attached tp the robot. With the *MoveIt! Setup Assistant* this information can be generated in the SRDF file when the URDF file is given as an input. Figure 3.7 gives the layout of the *MoveIt! Setup Assistant*.

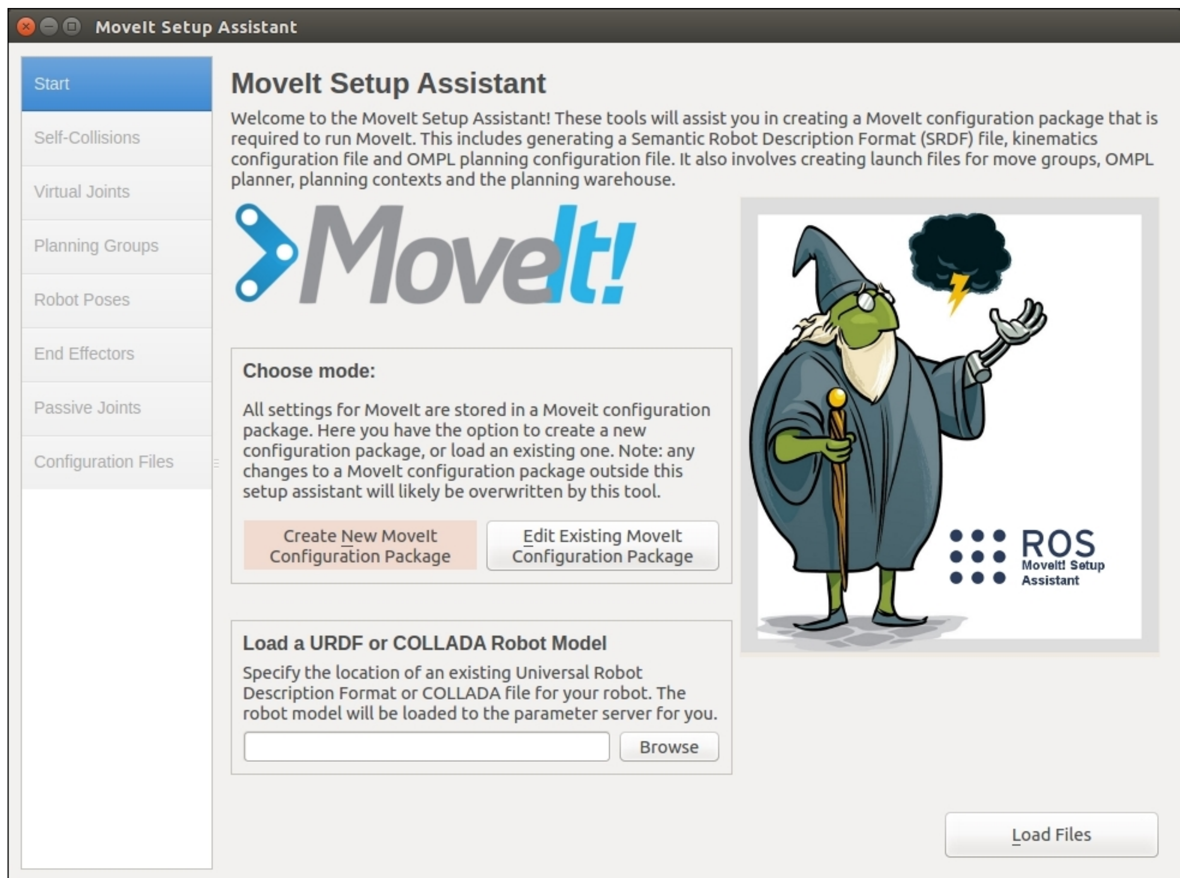



**FIGURE 3.7:** Screenshot of the *MoveIt! Setup Assistant*

The output of this setup assistant is a *MoveIt! Configuration* package where all important information of the inverse kinematics is saved. Also a few launch files are created to implement MoveIt! on the manipulator in simulation.

## 3.5 Conclusion

In this chapter the principles of a URDF file were explained. This URDF file is necessary to describe the kinematic chain of the robot since ROS needs this information. Next, the description of links and joints of the robot were specified. With a full description of the robot in a URDF file, the robot can be simulated in RViz. In addition, the explanation of creating a *MoveIt! Configuration* file has been added.

# Chapter 4

# PROFIBUS communication

## 4.1 Introduction

In this chapter a brief overview of communication using PROFIBUS will be given. This chapter will focus on the working principle of PROFIBUS, the PROFIBUS communication protocol and the applications of this fieldbus in its industrial environments.

## 4.2 Fieldbus

Back in the eighties the need was high to replace conventional wiring of in- and outputs to sensors and actuators with fieldbus technology. This change provided an enormous economisation regarding place, cable (wires) and maintenance time. Through this conventional wiring of sensors and actuators, technologies such as *smart sensors* and *smart actuators* had to wait until the first industrial fieldbus was released in the late 1980s [32].

In the late 1980s, begin 1990s a project between industrial giants like ABB, Bosch and Siemens, a pair of research centres, and a few universities was started. This project was called the *Process Field Bus* project and their aim was to standardise a field bus technology where information could be exchanged in the form of digital signals by using less cables connecting to many participants in a local network. In 1989, the first version of this project was developed and in 1991 the project became standardised under DIN 19245 under the name of *PROFIBUS*, see figure 4.1. In Europe, this standard is captured under EN 50170 and IEC 61158 [32].

**FIGURE 4.1:** Logo of the PROFIBUS organisation

## 4.3 Basics of the PROFIBUS communication protocol

### The OSI model of communication

The *Open Systems Interconnection* model (OSI model) is a conceptual model which characterises the communication principles of a network [33]. This OSI model consists of seven layers that characterise a network. PROFIBUS uses three out of these seven layers, see figure 4.2.

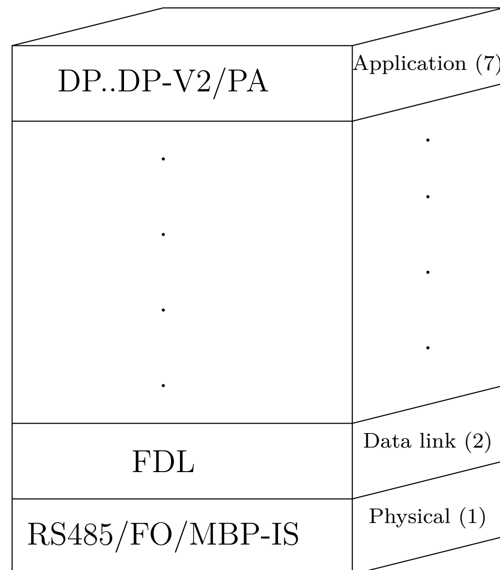



**FIGURE 4.2:** The seven layers of the OSI-model applied to PROFIBUS

Figure 4.2 displays the OSI seven layers model applied to PROFIBUS. As can be seen, PROFIBUS doesn't use all of the seven layers. PROFIBUS uses layer 1 which is the physical layer. This layer defines the technology over which the information is transferred, i.e. the transmission technology. This will be discussed later in a following section. Layer 2 describes the communication technology which is used in PROFIBUS. This will be discussed in the following subsection. Last, layer 7 is also used by PROFIBUS where the different applications of PROFIBUS are displayed. This will also be discussed later.

### Data link-Communication Protocol

In general, PROFIBUS is a serial network with two protocols running on the wire. The first one is the master-slave protocol. To create a hierarchy in the network, PROFIBUS differentiates two types of stations. These stations are either active (masters) or passive (slaves). Figure 4.3 depicts this protocol [34].

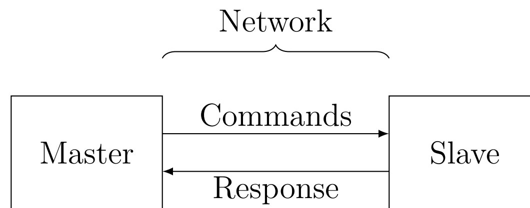



**FIGURE 4.3:** The master-slave protocol used by PROFIBUS

The *master-slave* principle, as described in figure 4.3, is a form of handshaking where a master sends information to a slave and the slave responds to the master by sending information back. A slave never receives transmission rights on the bus, unless the master determines to let the slave speak.

The second type of protocol that is used with PROFIBUS is the *token passing*. Token passing ensures that, when working with multiple masters, each master receives a certain transmission time to speak with their slaves, see figure 4.4.

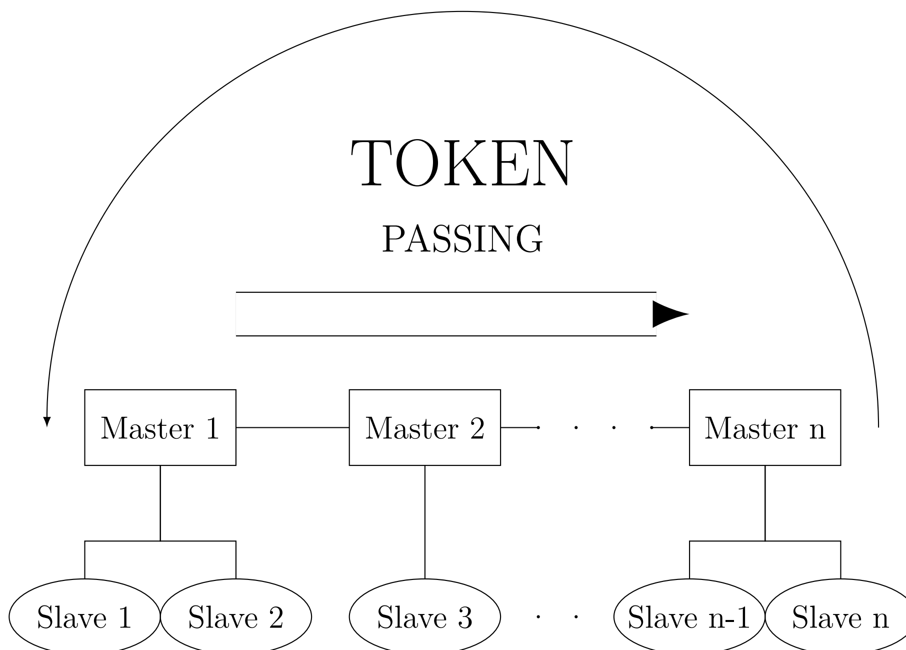



**FIGURE 4.4:** Principle of token passing in PROFIBUS

The master with the lowest PROFIBUS address may speak first to his slaves. When his speaking cycle time is over, the next master is allowed to speak. When the last master, i.e. the master with the highest PROFIBUS address, has spoken to his slaves, the first master receives the token to speak. To make the token ring as short as possible, the masters need to receive a low PROFIBUS address while the slaves receive a high address.

Communication over PROFIBUS happens in the form of messages called telegrams. These telegrams consist of *Universal Asynchronous Receiver/Transmitter* (UART) characters. An UART character is a character which is 11 bits long and is used to ensure that the receiver, by receiving the telegram, is temporarily synced with the sender and is able to receive the information correctly. Each character consists of a start bit, by which the receiver synchronizes, 8 data bits (data is send per byte), one parity bit, which controls the data on bit level, and a stop bit to indicate that the character is over. When sending telegrams, different UART characters that belong to the same telegram need to have to follow consecutively without any gaps in between.

In general, four types of telegrams can be transferred along the PROFIBUS network:

- SD1 telegram;
- SD2 telegram;
- SD3 telegram;
- SD4 telegram.

A first type of telegram is called the SD1 telegram, where SD stands for *Start Delimiter*. A SD1 telegram is a frame with a fixed length and no data field. Figure 4.5 explains this in more detail.

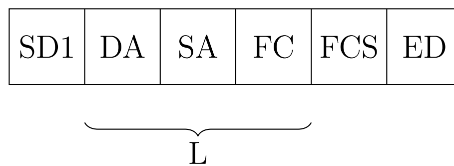



**FIGURE 4.5:** Explanation of a SD1 telegram. One block contains one octet of memory storage.

In figure 4.5 can be seen that a SD1 telegram starts with a Start Delimiter which has a value of $10_{HEX}$ and ends with an *End Delimiter* (ED) with a value of $16_{HEX}$. This kind of message is used for a state request or a transaction confirmation. The information length L is three octets long and contains the *Destination Address* (DA), the *Source Address* (SA) and the *Frame Control* (FC). The *Frame Check Sequence* (FCS) checks if the sequence of data fields is respected.

Figure 4.6 explains the structure of a SD2 telegram.

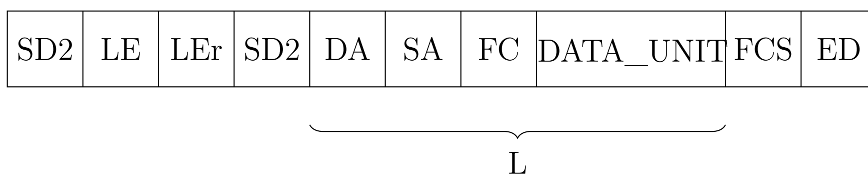



**FIGURE 4.6:** Explanation of a SD2 telegram. One block contains one octet of memory storage.

The SD2 telegram now has a value of $68_{HEX}$ which indicates a SD2 type of telegram. This type of telegram is used to transfer data of variable length over the network. This variation in length means that the *Data Unit* can contain a maximum of 246 octets. Compared to the SD1 telegram, this one has three extra blocks, namely LE, LEr and SD2. LE stands for the length of the *Data Unit*[1] (DU) which is transferred. LEr is the length of the DU repeated. When LE and LEr aren't equal, this indicates that the telegram is false and will be rejected. This is an extra control of the telegram.

A SD3 telegram is a telegram with a fixed length of 8 bytes. This kind of telegram is used when the information transferred on the network will not change in time.

A SD4 telegram is used to transfer the token from one master to the next one. In contrast to the SD1 and SD2 telegram, no control is provided to check any disturbances of the telegram. Also, no reply is given to this telegram.

A fifth type of telegram, depicted in figure 4.7, which is used by PROFIBUS, is the *Short Acknowledgement* (SC). This type of telegram, containing one byte, is send to a slave to check if the data is transmitted correctly or not.

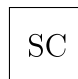



**FIGURE 4.7:** Explanation of a Short Acknowledgement telegram, with a value of $E5_{HEX}$

### Network speed

PROFIBUS contains ten transmission velocities over which data can be transferred. These velocities are depending on the transmission technique that is used (see section 4.4). Velocities from 9.6 kbit/s to 12 Mbit/s can be achieved. These velocities are a function of several criteria:

$$v_{network} = f(l_{cable}, \#masters, \#slaves, ...) \tag{4.1}$$

More information will be given in section 4.4.

## 4.4 Transmission

As mentioned in figure 4.2, PROFIBUS uses three of the seven layers of the OSI model for communication. Layer 1 defines the physical layer of the OSI model. According to PROFIBUS, three physical transmission techniques can be used. These are the RS-485 transmission technique, the MBP-IS technique and optical fibre transmission technique.

[1]The Data Unit contains the information which will be transferred over the PROFIBUS network. This DU is variable in length and is depending on the amount of data that will be transferred.

When using the RS-485 transmission technique, a serial connection is established using two or four signal lines where full duplex[2] communication is supported. Half duplex communication is also supported when two signal lines are being used. Figure 4.8 explains the wiring of this transmission technique.

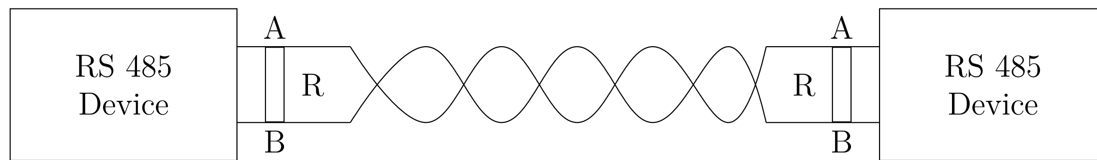



**FIGURE 4.8:** Schematic representation of the RS-485 transmission technique

The RS-485 transmission technique consists of a differential voltage principle. This means that data will be transferred when a varying voltage difference exists between both wires. In figure 4.8 the principle of RS-485 transmission is explained. As can be seen, the wires are twisted. This is to avoid disturbances caused by the environment surrounding the cable. An external disturbance will be cancelled due to an opposite current in the wire. With this transmission technique, transmission velocities of 9.6 kbit/s to 12 Mbit/s can be achieved with distances of respectively 1200 to 10 meters. Important to mention is that every segment is provided with a closing resistance. This closing resistance R (commonly around 120 Ω) is needed to keep the voltage stable at the end of the network (typically the last slave in the network).

Besides the RS-485 transmission technique, the *Manchester Bus Powered-Intrinsic Safe* (MBP-IS) technique is used in places where explosions can happen. Using this technique, a fixed transmission velocity is set at 31,25 kbit/s, distances of 1900 meters can be attained, and power supply is transferred over the bus.

A third transmission technique is to use optic fibre wire. The velocities of this technique are comparable to the RS-485 technique. However, this technique is more redundant to electrical disturbances. Because of this, distances of 100 kilometres can be achieved.

## 4.5 Applications

PROFIBUS is used in a wide variety of applications in the automation technology world. This is because of its highly favourable properties for increasing productivity and decreasing downtime in manufacturing. Because only two wires are used to transmit data, there is a real cost reduction in using this fieldbus compared to wired logic where the amount of wires grew as the number of components increase. This mainly comes from a reduction in hardware and cabling [35]. Furthermore, PROFIBUS stands out from other fieldbuses in the sense that it's a worldwide accepted standard and all leading manufacturers of automation technology offer PROFIBUS interfaces for their devices. To give an overview of the wide range of applications used by PROFIBUS, table 4.1 is shown.

[2]Half duplex communication is a form of communication where sending and receiving of data is supported but not simultaneously. Using full duplex communication sending and receiving data simultaneously can be achieved.

**TABLE 4.1:** Overview of applications where PROFIBUS is used

| Industries where PROFIBUS is used | Application |
|---|---|
| Process Automation | Purification plants<br>Chemical plants<br>Paper and textile industries |
| Power industry and power distribution | Power plants<br>Switch gears |
| Manufacturing Automation | Car manufacturing<br>Bottling systems<br>Storage systems<br>Welding systems (robots) |
| Building Automation | Traffic automation<br>Heating, ventilating and air conditioning |

## 4.6 Open implementations of PROFIBUS

As mentioned in section 4.2 PROFIBUS is standardized under DIN 19245, EN 50170 and IEC 61158. These standards are used by manufacturers of PLC software like Siemens, Beckhoff or Pilz for example. In this software a fieldbus connection can easily be set up by graphically drawing a PROFIBUS cable from one device to another. This is convenient when using standardised components like PLC's or I/O systems. However, when a PROFIBUS communication is needed for the implementation of a new research topic, this standardised software falls too short. Thus, an open-source implementation of PROFIBUS is highly recommended for this research. This section gives a brief summary of the publicly available implementations of the PROFIBUS standard.

### Serial

As mentioned in section 4.2, PROFIBUS is a serial type of communication. ROS provides a package for serial communication called *serial* [36]. This ROS package is a simple to use C++ library for using serial ports on computers. USB ports and RS-232 like devices on Linux and Windows can be interfaced using this library. RS-485 is one of the transmission techniques that can be used with PROFIBUS. Interfacing with a RS-485 device is in the first place no problem since RS-485 is RS-232 compatible. On the other hand, this package was mainly created for interfacing with Arduino Unos and not with robot controllers and so on. Also, PROFIBUS doesn't support RS-232 transmission technique.

### ROSSerial

A second library within ROS is called *ROSSerial* [37]. This package, like the serial library, is a package created to interface with devices over a serial port. Unlike the *serial* package, this library is better documented and more extensive. This package describes a protocol for wrapping standard ROS messages to multiplex various topics and services over a serial port or network socket. With this package ROS nodes can easily set up and running on various systems like an Arduino, a XBEE device or a router. Like the *serial* package, this ROS library suffers from some limitations. A first limitation is the maximum size of a message. This size is limited by 512 bytes. This is very low considering the maximum speed of 12 MB/s of PROFIBUS. Secondly, this ROS package does not support 64 bit float datatypes. The code automatically converts 64 bit floats into 32 bit floats, but communication loss can not be excluded. A third limitation of this package is that strings are stored in an unsigned character. This is not desirable, because when publishing a message, the string data has to be stored elsewhere and the string will not be copied from the deserialisation buffer. A last limitation is the use of an array. To send an array using this package, the length of the array and the pointer has to be set. These limitations are the reason that this package is currently implemented for 'simple' serial communication like interfacing with Arduino's and not with robot controllers.

### PBMaster

A third possibility of implementing an open-source version of PROFIBUS is described in [38] where a project named *PBMaster* is presented. This paper, originally a master's thesis at the Czech Technical University in Prague, presents an open software implementation of PROFIBUS Decentralised Peripherals (DP). This software implementation can run on a wide range of hardware, where the UART and RS-485 standards are used. With the PBMaster project it is possible to connect to devices using PROFIBUS on a host OS like Linux. In this project, a RS-232 to RS-485 dongle is used to plug on the serial port of a PC. Also, device drivers are included to use and a Live Linux CD to virtually run the PBMaster project on any device by simply booting to this CD. This project is probably the solution to the need of a open implementation of PROFIBUS. However this project has great potential, patents of the PROFIBUS International (PI) organisation holds the project from being distributed as open-source. Until now, this project hasn't been distributed yet online, so reuse of this code can not be done.

### PyProfibus

A final potential of an open implementation of PROFIBUS is the PyPROFIBUS software stack, developed by Michael Büsch [39]. This software stack consists of an implementation, written in Python, which covers the three OSI layers (physical, data link and application) of the PROFIBUS protocol. This implementation has been exposed to two tests by the developer. The first test was to set up a communication between a PC and a Siemens ET-200S I/O system where the PC reads the inputs of the slave and writes it to its outputs. This is similar to the Panasonic robot controller since it acts like a slave system in the network.

A second test was conducted to interface with a LinuxCNC, an open-source CNC machine controller. For this, a connection with the DP slaves was initialised and works perfectly. Unlike the DPMaster project, described in the subsection above, this software stack is publicly available and is not held down by any patents of the PI organisation. The software stack can be reused by downloading it on the website of the developer , on the PyPy website, or on its GitHub page [40].

## 4.7 Conclusion

In this chapter a brief overview of the PROFIBUS communication principle and protocol has been given. Besides the principle and protocol of PROFIBUS, the transmission techniques and the applications of PROFIBUS were discussed. Finally a series of open implementations of serial communications and PROFIBUS software stacks have been presented. In this section, the final implementation, i.e. PyProfibus, has a great opportunity to use for the communication between the Panasonic robot controller on the one hand and ROS on the other hand. The following chapter will bring together the PROFIBUS knowledge with the ROS knowledge of the previous chapter.

# Chapter 5

# Implementation of the ROS driver

## 5.1 Introduction

With the knowledge acquired in chapter 2 about the requirements of a ROS driver and in chapter 4 about the basics of PROFIBUS fieldbus communication, this chapter gives the results of the implementation of the ROS driver for the Panasonic VR-006L manipulator. It begins by examining and explaining an offline way of communicating in ROS with the robot controller. The next sections present the implementation of a bidirectional communication flow between ROS and the robot controller. These two implementations have led to several conducted experiments. Lastly, these experimental results are given and evaluated.

## 5.2 Offline communication

A ROS driver, as mentioned earlier in chapter 2, translates commands generated in ROS to commands that can be obtained by the robot controller. A first step in implementing such a driver for the Panasonic manipulator, is to make a conversion programme. This conversion programme translates commands in ROS to a csr file. A csr file is a file type specifically designed by Panasonic where commands and general information of the robot controller are stored in a Panasonic-specific format. When having such a csr file, transferring it to a programme file in the *Desktop Programming Simulation System* (DTPS) and execute it on the robot, is convenient. However, such a conversion programme is not really a driver because online communication with the controller at any time is not provided. On the other hand, it is a good step towards an online ROS driver.

Figure 5.1 below shows the premeditation of a conversion from ROS commands to a csr file.

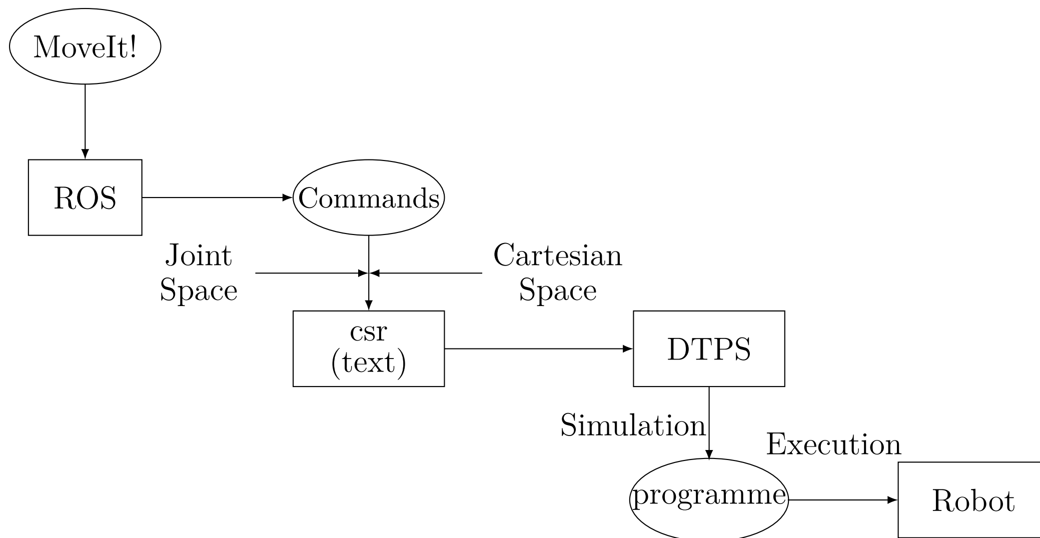



**FIGURE 5.1:** Premeditation of a conversion from ROS commands to a csr file

From figure 5.1 it can be seen that commands are generated by ROS. These commands can for example be generated by MoveIt! in cooperation with ROS. In MoveIt!, a path planner generates a point to where the end effector of the robot has to move. With the known position in Cartesian coordinates ($X, Y, Z, R_X, R_Y$ and $R_Z$), the path planner calculates the orientation of every joint (all six joint angles in degrees) to cope with the desired position.

The next step is to translate these commands to a csr file. This csr file consists of a text file where the structured information of the programme is saved, in a Panasonic-specific format. In this file for example, information about the poses of the pre-defined points are stored but you can also find which type of robot or what kind of end effector (welding torch, gripper, palpus...) is used. The poses defined within this csr file consist of joint space variables, i.e. a vector of joint angles. The commands in ROS can either be in joint or Cartesian space. If they are defined in Cartesian space, a conversion[1] to joint space variables with an IK solver needs to be done before translating it to the text format of a csr file.

When a csr file is provided, it can easily be transferred to the DTPS software on a computer. In here, the csr file is converted into a programme file. Also, it can be simulated in the DTPS software before converting it to an executable (programme file). When the csr file is converted to a programme file, it can be loaded on the robot's teach pad using a flash card. On the teach pad the desired programme can be chosen and can be executed on the robot.

[1]In the Panasonic software a pose can be defined in various ways like for example in Cartesian space, joint space, Euler angles, ... . When defining a pose in Cartesian space, the Panasonic software itself uses a IK solver to translate it to joint space variables to power up the servo motors which drive the joints of the robot.

## 5.3 Bidirectional setup

### 5.3.1 Strategy

Now the offline communication is set, a logical next step is to implement a bidirectional interface. In this bidirectional setup, ROS sends a desired Cartesian position to the Panasonic robot controller over the PROFIBUS fieldbus. The robot controller receives the desired Cartesian position and executes the desired movement. Figure 5.2 shows a schematic of the bidirectional setup between ROS and the Panasonic robot controller.

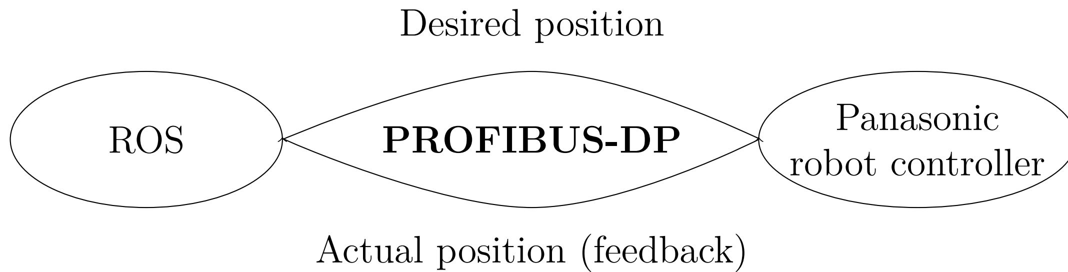



**FIGURE 5.2:** Bidirectional setup between ROS and the Panasonic robot controller

As can be noticed in figure 5.2 ROS sends the desired position in Cartesian space to the Panasonic robot controller. This data is transmitted over the PROFIBUS network. In ROS the X, Y, Z position and $R_X$, $R_Y$, $R_Z$ orientation are send by speaking with an I/O system. In this case, the I/O system in the PROFIBUS network is the robot controller. Speaking with the controller in this context means that ROS sets the configured inputs on the controller to a logic 1 or a logic 0. On the PROFIBUS card of the controller there are 14 input bytes and 14 output bytes (7 input/output words (=integers) or 112 I/O's) parametrised. Over the PROFIBUS network only data telegrams can be transmitted (see chapter 4, section 4.3). In these data telegrams the desired input and output can be send to the I/O system and received back by the master respectively. To make a robust interface, a kind of internal protocol between ROS and the Panasonic controller must be implemented. Table 5.1 gives a brief overview of this inner protocol between ROS and the Panasonic robot controller.

**TABLE 5.1:** Overview of the inner protocol between ROS and the Panasonic robot controller

| **Specified byte** | **Meaning** |
|---|---:|
| Byte 1-2 | X position |
| Byte 3-4 | Y position |
| Byte 5-6 | Z position |
| Byte 7-8 | $R_X$ orientation |
| Byte 9-10 | $R_Y$ orientation |
| Byte 11-12 | $R_Z$ orientation |
| Byte 13 | flag (1: request data, 0: no request) |
| Byte 14 | robot's movement type |
| **Total I/O's** | **112 I/O's** |

As described by table 5.1 above six bytes are reserved for the Cartesian position and six bytes are reserved for the Cartesian rotation. Byte number 13 is reserved for the *flag*. In the bidirectional communication between ROS and the robot controller this flag is treated as a request to send data to the Panasonic controller or not.

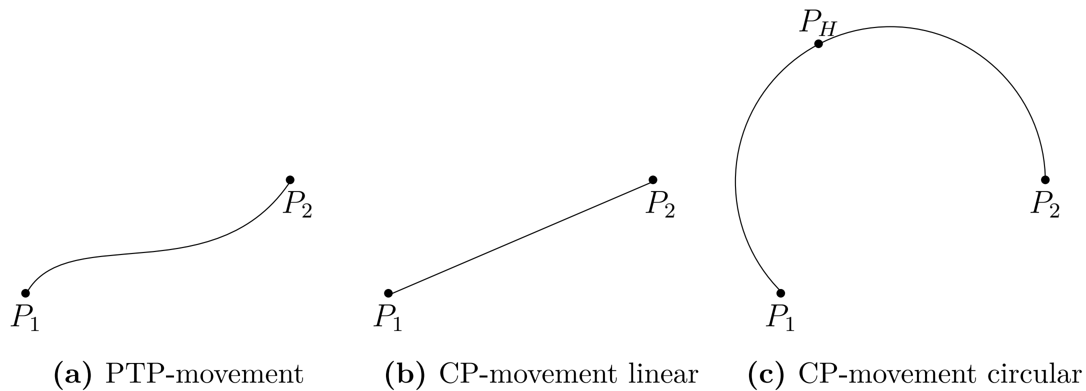
### 5.3.2 Implementation

As described in chapter 2 three types of driver interfaces can be implemented. In this bidirectional interface a position streaming interface is implemented. ROS generates commands to send a position to the robot controller. After all the data is received by the robot controller, the robot executes its imposed movement. Figure 5.3 below describes the implementation of the position streaming interface over the PROFIBUS network between ROS and the Panasonic controller in pseudo code.

```
Input: 14 bytes on the PROFIBUS inputcard from ROS
Output: 14 bytes on the PROFIBUS outputcard to ROS
Send request by setting flag to 1
while Connection == True do
    Send current position of robot to ROS
    if flag == 1 then
        Read inputs that describe X, Y and Z position
        Read inputs that describe X, Y and Z orientation
        Read inputs that specify movement method (Point-to-point (PTP), Continuous
        Path (CP) (linear or circular))
        if robot is in specified position then
            Set the flag to 0
            Send [X, Y, Z]^T to the PROFIBUS outputcard to let ROS know that the robot
            is in desired position
        end
        Wait until the flag is set to 1 for next movement
    end
end
```

**FIGURE 5.3:** Pseudocode of the position streaming interface implemented on the Panasonic robot using the DTPS software

Figure 5.3 describes the implementation of the server/programme that has been programmed on the Panasonic robot controller using the DTPS software. As can be seen, the programme expects 14 input bytes that are send over the PROFIBUS network by ROS. In these 14 bytes, byte number 13 contains the flag. The programme needs a logic 1 to execute its loop cycle. The programme will be started when the connection is established and the flag is set to 1. If this is the case, the robot sends its current position (only $[X, Y, Z]^T$) to ROS by setting the corresponding output bytes 1-6. Because of this, ROS knows what the actual starting position of the robot's movement is. Then, the programme loop will start to read the bytes on the PROFIBUS inputcard. These bytes contain the Cartesian position and orientation of a desired point. Besides this, the fourteenth input byte contains the specified robot movement. The different types of robot movement are displayed below in figure 5.4.




**(a)** PTP-movement **(b)** CP-movement linear **(c)** CP-movement circular

**FIGURE 5.4:** Schematic overview of the three implemented robot movement types

As shown in figure 5.4 three types of robot movement are implemented in the interface to let a robot move from point $P_1$ to $P_2$. A first one is a *point-to-point* (PTP) movement (figure 5.4a). This movement is specified by a starting point and an end point. The movement in between these two points is a random, interpolated function. It's typical for a PTP movement that all the joints are synchronously activated, which results in a curved trajectory [41]. A second type of robot movement that has been implemented, is a *linear continuous path* (CP linear) movement (figure 5.4b). This movement is specified by the linear trajectory between starting point and target point. The mathematical trajectory is a function of first order. Last, a third type of robot movement that has been implemented is a circular movement (figure 5.4c). This movement is specified by a auxiliary point $P_H$ which is on a circle between starting point and reaching point. The robot movement type is implemented in the DTPS programme as follows. If byte 14 equals 1 (bit 1 of byte 14 is a logic 1), the specified movement is a PTP movement. If byte 14 equals 2 (bit 2 of byte 14 is a logic 1), the specified movement is a CP linear movement. Last, if byte 14 equals 3 (bit 1 and bit 2 of byte 14 are logic 1), the specified movement is a CP circular movement.

After all the 14 input bytes are read, the robot controller will move the robot to the desired target point imposed by ROS. If the robot is in this target point, the robot controller will let this know to ROS by sending the current position (= target position) to the output bytes on the PROFIBUS output card. Therefore ROS can visualise the robot's trajectory movement in a simulation and the user can check if the imposed trajectory is the same as the simulated one. Besides this, the robot controller will also set the flag byte (byte 13) to a logic 0 when the desired position is reached to ensure that no false commands can be transmitted over the network.

## 5.4 Experimental results and evaluation

### 5.4.1 Testing the communication programme on the Panasonic robot controller

As described earlier in section 5.3 of this chapter, a server/programme is programmed on the robot controller. This server keeps controlling the PROFIBUS inputcard for any change. With the change of data a desired command corresponds. To check if the programmed server works properly, a communication can be set up between the robot controller and a PLC station. Figure 5.5 illustrates this setup schematically.

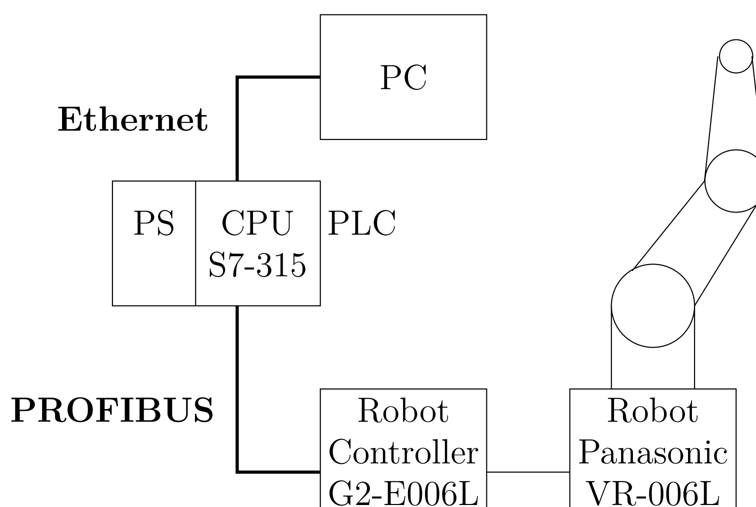



**FIGURE 5.5:** Schematic representation of the communication setup between the robot controller and a Siemens S7-315 PLC

Figure 5.5 shows that a PC (with TIA Portal Siemens software installed), is connected with a Siemens S7-315 PLC station. This communication happens over the Ethernet protocol. In the PC, a master-slave network is configured between the PLC and the robot controller where the PLC acts as a master and the robot controller as a slave. The *General Station Description* (GSD)[2] file of the slave is loaded in the TIA Portal software. Because of this file, the master knows what type of slave he is speaking to [32]. To test the proper working of the server on the robot, data is send from the Siemens PLC to the robot over the PROFIBUS network. This data corresponds to the internal data protocol between ROS and the robot controller as shown in table 5.1.

[2]A GSD file contains information about the basic capabilities of a device, displayed in a standardised XML format.

Figure 5.6 below shows an example of the data traffic send to the robot controller over the PROFIBUS network.

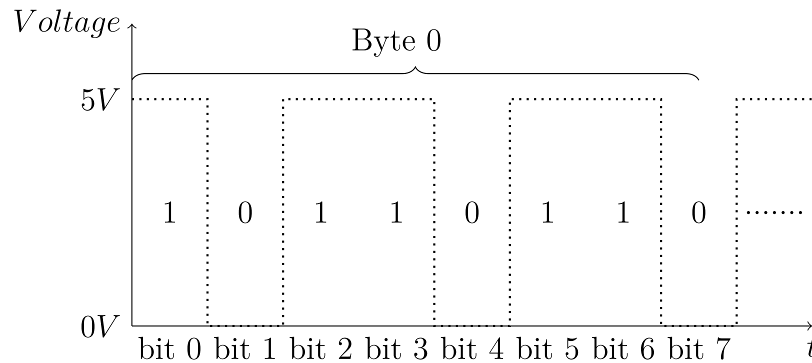



**FIGURE 5.6:** Example of the data traffic send to the robot controller over the PROFIBUS network

As explained in chapter 5 the PROFIBUS protocol corresponds to a differential voltage output transmitted over two wires. Figure 5.6 shows the sent signal, corresponding with the first byte, from the PLC station to the input card of the robot controller. On the robot controller this signal can be checked by using the robot's teach pad and go to the IO menu. When the corresponding inputs are logic true, the communication works properly. This test can also performed in a bidirectional way, i.e. by setting some output bytes of the PROFIBUS output card of the robot controller high, send it to the PLC station and check if this data can be seen.

### 5.4.2 Testing the communication between ROS and a Siemens ET 200S I/O system using an open PROFIBUS implementation

After confirming that the PROFIBUS card on the robot controller works fine, something had to be established to connect ROS with the controller. The solution for this uses an open PROFIBUS driver [39]. However, before using this to implement a solution between ROS and the controller, a setup has been made to test the connection between a PC (with Linux Ubuntu 14.04 installed on it) and a Siemens ET 200S I/O system. Figure 5.7 illustrates this setup schematically.

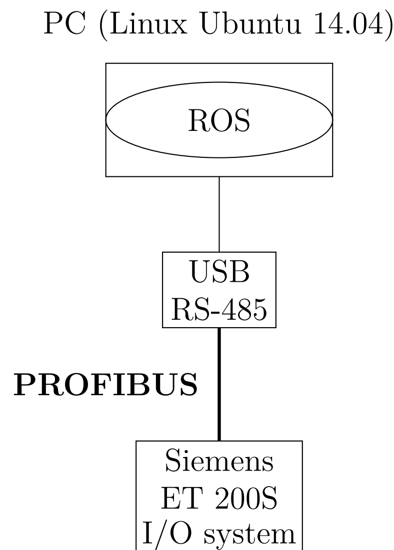



**FIGURE 5.7:** Schematic representation of the bidirectional communication setup between ROS and a Siemens ET 200S I/O system over PROFIBUS-DP, using a USB to RS-485 adapter

The connection between ROS and the Siemens ET 200S I/O system has been made possible using a USB to RS-485 adapter. This adapter is assembled with an USB to RS-232 adapter, a RS-232 to RS-485 adapter, and followed by soldering the TX-, TX+, GND, and 5V wires to a DB9 connector. The connector can than be connected to a PROFIBUS cable which is then connected with the robot controller. Figure 5.8 shows the hardware of this assembly.

**FIGURE 5.8:** A *Do It Yourself* (DIY) USB to RS-485 adapter

The setup above only needs the TX-, TX+ wires to both send and receive data over the PROFIBUS network. A test has been conducted to successfully send and receive bits to respectively from the I/O system. Table 5.2 shows which colour corresponds to which RS-485 wire.

**TABLE 5.2:** Overview of which coloured wire corresponds to which physical wire

| Colour | Corresponding wire |
|---|---|
| Green | TX- / D- |
| Yellow | TX+ / D+ |
| Blue | GND |
| Red | 5V |

### 5.4.3 Testing the communication between ROS and the Panasonic robot controller

Finally, after a successful test using the open PROFIBUS implementation, a setup has been made to connect ROS and the Panasonic robot controller. Figure 5.9 illustrates this setup schematically.

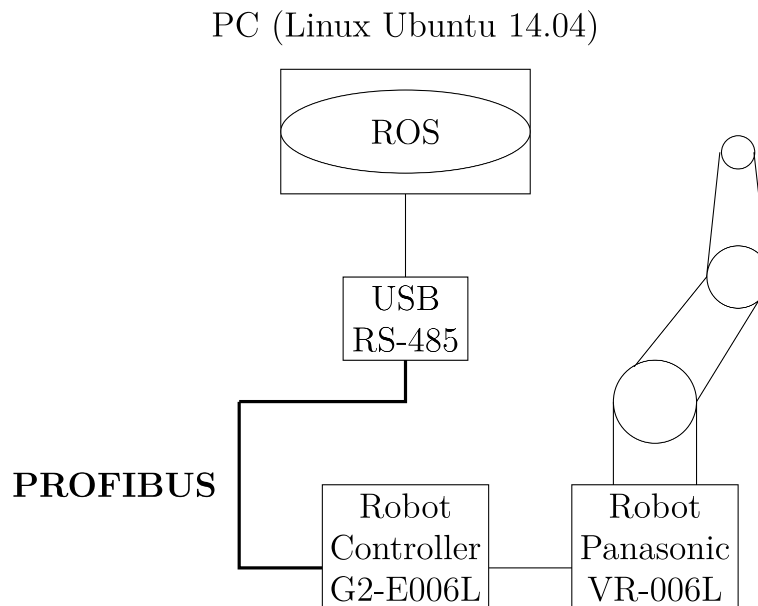



**FIGURE 5.9:** Schematic representation of the bidirectional communication setup between ROS and the Panasonic robot controller over PROFIBUS-DP, using a USB to RS-485 adapter

To make this work, a driver has been written in Python and uses the open PROFIBUS implementation to send commands from the USB port to the robot controller following the correct PROFIBUS protocol. This driver, when started, asks for the user's input, i.e. the $X$, $Y$, and $Z$ position, $R_X$, $R_Y$ and $R_Z$ rotation, and the movement type (as discussed in section 5.3). After giving this input, it is converted into 14 bytes to execute on the robot. The input is also converted to the right position and rotation coordinates for visualisation in RViz. When the user has entered the last input information (the movement type), the visualisation starts followed by execution of the commands on the robot. This visualisation is offered by the IK solver of MoveIt! where is assumed that MoveIt! gives similar solutions as the IK solver of the robot. These solutions may differ in the vicinity of the singularities of the robot.

## 5.5 Conclusion

In this chapter, an overview has been given from the steps taken to achieve the main goal of implementing a ROS driver. First, an offline conversion programme has been established. This conversion programme translates commands in ROS to a csr file which can be opened by the Panasonic DTPS software. In this software, the csr file can then be transferred to a programme file to execute it on the robot. In ROS, the commands can be represented in joint space coordinates, or in Cartesian space. When defining a pose in Cartesian space, the Panasonic software itself uses its IK solver to translate it to joint space variables.

Next, a bidirectional interface has been created. To implement this, an open PROFIBUS driver has been used. First, a test has been carried out with a Siemens S7-315 PLC station to test the programme on the robot. Then, another setup has been tested using a USB to RS-485 adapter to send commands from ROS to the PROFIBUS card on the robot controller. When ROS sends a desired Cartesian position to the Panasonic robot controller over the PROFIBUS network, the robot controller receives the desired Cartesian position and executes the desired movement. Unfortunately, joint coordinates can't be read and set by the Panasonic's G2 controller. This is a huge limitation when accurate and reliable path planning is desired. Also, the G2 robot controller isn't able to send the actual robot state at a fixed frequency. It can only send its actual state (only Cartesian position coordinates) back when the robot has executed a movement by setting outputs on the output card. Furthermore, the robot controller isn't able to send and receive floating point numbers. It is limited to accepting integer numbers. Therefore the accuracy of the imposed position and rotation is 1 mm and 1° respectively.

The way the final setup works is as follows. On the I/O system there are 14 input bytes and 14 output bytes. The first six bytes are reserved for the Cartesian position and the next six bytes are reserved for the Cartesian rotation. Byte number 13 is reserved for the flag and byte 14 for the movement type (point-to-point, continuous path linear or continuous path circular). A programme that has been made on the robot controller will read the PROFIBUS inputs and move the robot to the desired pose accordingly. After reaching this point, the controller lets this know to ROS by sending the current position (= target position) to the output bytes on the PROFIBUS output card. Therefore ROS can visualise the robot's trajectory movement in a simulation and the user can check if the the imposed trajectory is the same as the simulated one.

# Chapter 6

# General conclusion

## 6.1 Contributions of this work

First of all, this thesis describes how to create a geometric and kinematic robot model for ROS. This model has been described in a URDF file to use this for path planning applications within ROS or for collision detection. The robot model can be viewed in RViz and used in MoveIt!. Furthermore, an offline conversion programme has been written. This programme can convert commands generated in ROS to commands in a csr file. A csr file is a Panasonic-specific file containing all the information of the robot programme, which can be opened by the DTPS programme of Panasonic. In this software, a simulation can be done and commands can be executed if the robot would be connected to the computer with the DTPS software.

The central goal of this master's thesis was to develop a ROS driver that makes the translation between a path planning tool, ROS and the Panasonic robot arm controller. The implementation however, was different than first anticipated, because the robot's G2 controller used for this thesis only has a PROFIBUS connection and a few limitations. The first limitation is that it is not able to read and send joint space coordinates. A second limitation is the accuracy of the controller's network communication. The controller can only send and receive integer numbers. Floating point numbers are out of the question. Therefore the accuracy of the driver is limited to 1 mm in position and 1° in orientation. A third limitation is that the driver isn't able to send the robot's actual state at a fixed frequency (obtained by the ROS consortium). It can only shift data over the PROFIBUS network after a movement is executed, which is not a at a fixed frequency. Besides, in the driver implementation, only the $X, Y$, and $Z$ position coordinates can be send back to ROS to check if the robot has reached its desired position.

Online communication was made possible using an open PROFIBUS implentation in which the driver is written. The driver makes the translation user friendly because it asks to put in the $X, Y$, and $Z$ position coordinates and the orientations and the preferred movement type. This driver can be categorized as a *position streaming* driver since the robot controller executes a desired position of the robot that is imposed by ROS. Additional, with this driver IO control is possible since the PROFIBUS network allows it.

## 6.2 Future work

As future work, the ROS driver could be provided with an extra category, i.e. *trajectory downloading*. Since the Panasonic controller allows to give in trajectories, such an interface could be implemented.

The proposed ROS driver in this thesis is based on a Panasonic's G2 robot controller. This controller however, suffers from some major restrictions as described in section 6.1. First of all this controller cannot read in joint values over the PROFIBUS network. This isn't neat since the robot controller itself calculates the inverse kinematics to power the motors for every joint. If joint values could be read in by the controller, these inverse kinematics computation could be done by ROS. Therefore, upgrading to a newer Panasonic G3 controller would provide probably better insights to a proper ROS driver. Since this G3 controller comes with a standard Ethernet connection, a real-time client-server network could be the key to a robust ROS driver.

Most parts of the code used in this thesis is represented in appendix E and can be consulted online in the following GitHub repository: `https://github.com/davidDS96/Panasonic-VR-006L-driver`.

# Bibliography


[1] KU Leuven, "About ACRO," 2017. [Online]. Available: https://iiw.kuleuven.be/onderzoek/acro/about

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Mg, "ROS: an open-source Robot Operating System," *Icra*, vol. 3, p. 5, 2009.

[3] J. Baeten, K. Donné, S. Boedrij, W. Beckers, and E. Claesen, "Autonomous fruit picking machine: A robotic apple harvester," in *Springer Tracts in Advanced Robotics*, vol. 42, 2008, pp. 531–539.

[4] International Federation of Robotics, "Executive summary world robotics 2017 industrial robots," 2017. [Online]. Available: https://ifr.org/downloads/press/Executive_Summary_WR_2017_Industrial_Robots.pdf

[5] Willow Garage, "The Challenge: Transitioning Robotics R&D to the Factory Floor," 2013. [Online]. Available: https://rosindustrial.org/the-challenge/

[6] E. Demeester, J. De Maeyer, and B. Moyaers, "Cartesian Path Planning for Welding Robots: Evaluation of the Descartes Algorithm," Limassol, Cyprus, 2017, p. 8.

[7] "ROS-Industrial Driver Specification." [Online]. Available: https://wiki.ros.org/Industrial/Industrial_Robot_Driver_Spec

[8] "ROS Distributions," 2017, last visited on 20-09-2017. [Online]. Available: http://wiki.ros.org/Distributions

[9] Willow Garage, "About ROS," 2007. [Online]. Available: https://www.ros.org/about-ros/

[10] ——, 2014, last visited on 20-09-2017. [Online]. Available: http://wiki.ros.org/indigo

[11] "Installation of ROS Indigo Igloo," 2017, last visited on 20-09-2017. [Online]. Available: http://wiki.ros.org/indigo/Installation/Ubuntu

[12] Fleder Michael, "ROS: Robot "Operating" System," 2012.

[13] "ROSlaunch," 2017, last visited on 20-11-2017. [Online]. Available: http://wiki.ros.org/roslaunch

[14] "Catkin: a conceptual overview," 2017, last visited on 20-11-2017. [Online]. Available: http://wiki.ros.org/catkin/conceptual_overview

[15] "SRDF," 2014, last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/srdf

[16] “URDF,” 2014, last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/urdf

[17] G. Walck, “Introduction to Robot Modeling in ROS,” 2015.

[18] “RViz,” 2016, last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/rviz

[19] “MoveIt!” [Online]. Available: http://moveit.ros.org/

[20] “ROS-Industrial,” 2017, last visited on 20-10-2017. [Online]. Available: http://wiki.ros.org/Industrial

[21] “Supported Hardware,” 2015, last visited on 20-12-2017. [Online]. Available: http://wiki.ros.org/Industrial/supported_hardware

[22] YASKAWA, “Motoplus-ROS Incremental Motion interface Engineering Design Specifications,” YASKAWA. Motoman Robotics., Tech. Rep., 2017.

[23] “ROS Communication.” [Online]. Available: http://wiki.ros.org/ROS/Patterns/Communication

[24] “Simple Message Protocol,” 2015, last visited on 20-12-2017. [Online]. Available: http://wiki.ros.org/simple_message

[25] L. Castelli and W. Van der Aelst, “Ontwikkeling van een ROS-driver voor Epson C3 robots,” Master’s thesis, UHasselt, KULeuven, 2016.

[26] “Industrial Robot Client,” 2014, last visited on 20-12-2017. [Online]. Available: http://wiki.ros.org/industrial_robot_client

[27] “URDF-link,” last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/urdf/XML/link

[28] “URDF-joint,” last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/urdf/XML/joint

[29] J. Denavit and R. Hartenberg, “A kinematic notation for lower pair mechanism based on matrices,” *ASME Journal of Applied Mechanics*, pp. 215–221, 1955.

[30] P. Corke, “Denavit-Hartenberg notation for common robots,” p. 14, 2014.

[31] H. Bruyninckx, “Robot Kinematics and Dynamics,” KU Leuven, Tech. Rep., 2010.

[32] ACROMAG, “Introduction to Profibus DP,” ACROMAG, Tech. Rep., 2002.

[33] S. Paul, “The OSI Model: Understanding the Seven Layers of Computer Networks,” Global Knowledge Training LLC, Tech. Rep., 2006.

[34] PNO, *Technology and Application*. PNO, Profibus, 2002.

[35] H. Chen, X. Zbang, and X. Zbang, “Applications of Profibus to Industrial Automation,” *IFAC Proceedings Volumes*, vol. 31, no. 25, pp. 157–161, 1998.

[36] “Serial,” 2014, last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/serial

[37] “ROS Serial,” 2016, last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/rosserial

[38] P. Khanh, Tran Duy, Pisa, Pavel, Smolik, "An Open Implementation of Profibus DP," Czech Technical University in Prague, Faculty of Electrical Engineering Department of Control Engineering, Tech. Rep., 2009.

[39] M. Büsch, "PROFIBUS software stack," 2016, last visited on 20-05-2018. [Online]. Available: https://bues.ch/cms/automation/profibus.html

[40] ——, "PyProfibus," 2016. [Online]. Available: https://github.com/mbuesch/pyprofibus

[41] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*, ser. Advanced Textbooks in Control and Signal Processing. London: Springer London, 2009.

[42] P. Corke, "A Simple and Systematic Approach to Assigning Denavit-Hartenberg Parameters," *IEEE Transactions on Robotics*, vol. 23, no. 3, pp. 590–594, June 2007.

# Appendix A

# Dutch Summary

This appendix contains an article in Dutch which summarizes the most important parts of this master's thesis.

# Implementeren van een open-source PROFIBUS interface tussen ROS en een Panasonic robot

David De Schepper, Jorn Geutjens
UHasselt-KU Leuven

*Samenvatting*—**Afgelopen decennia is een substantieel deel binnen robotonderzoek gericht op padplanningsalgoritmes voor industriële robots. Deze nieuwe algoritmes controleren enkel high level beweging en moeten voor implementatie commerciële software commando's gebruiken voor iedere robot. Een open-source driver biedt hiervoor een alternatief. Drivers kunnen deze higher level robotonafhankelijke commando's omzetten naar robotcontrollerspecifieke commando's. ROS is een voorbeeld van zo'n open-source platform waar drivers kunnen worden ontwikkeld. Om de padplanningstoepassingen te testen op echte hardware is een ROS-driver essentieel.**

**De hoofddoelstelling van de thesis is het implementeren van een driver die de vertaalslag maakt tussen commando's in ROS en een Panasonic VR-robot, en vice versa. Drie stappen zijn nodig om dit te bereiken. Eerst moet een kinematisch en geometrisch model van de robot in ROS worden opgesteld. Ten tweede moet er unidirectionele communicatie worden voorzien tussen ROS en de robotcontroller. Tot slot dient er een interface te worden geïmplementeerd om de driver robuust te maken.**

**In deze thesis is een model opgesteld van de Panasonic robot in ROS. Ook is er een programma geschreven om een csr-bestand (Panasonic specifiek) te maken. Verder werd een ROS-driver gerealiseerd gebruik makende van een open PROFIBUS implentatie. Door de beperkingen van de G2-controller om jointposities te lezen en deze terug te sturen, is een logische volgende stap het upgraden naar een G3-controller met Ethernet-connectie. Daarnaast kan deze driver ook worden gebruikt om te communiceren tussen ROS en een PLC of I/O-eiland die geconfigureerd staan als slave.**

## INLEIDING

Het aantal robots dat wordt ingezet vertoont wereldwijd een stijgende tred [1]. Opvallend hierbij is dat deze robots niet enkel worden ingezet in een industriële context zoals de automotive- of metaalindustrie. Andere sectoren zoals de zorgsector vertonen een stijging van het aantal gebruikte robots. Anderzijds is ook op te merken dat de doeleinden van deze robots fel uitgebreid wordt. Deze manipulatoren worden niet enkel meer gebruikt voor het lassen of het plaatsen van werkstukken, ook werken deze samen met de mens zoals de humanoïde robots.

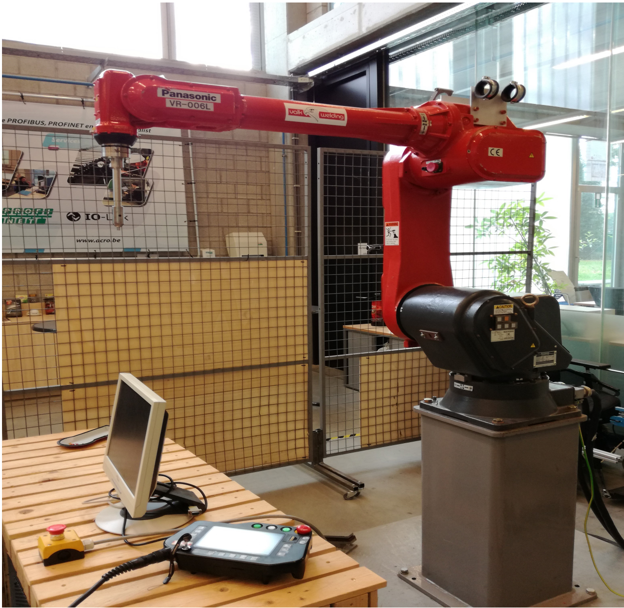
Nieuw onderzoek rond industriële robots richt zich vooral op het toepassen van padplanningstools op industriële robotica. Deze tools kunnen dan bijvoorbeeld worden toegepast op lasrobots waardoor de productie-en gebruikskosten worden gedrukt [2]. Voor het toepassen van nieuwe padplanningstools en het ontwikkelen hiervan wordt vaak gebruik gemaakt van open-source doeleinden. Commerciële software echter is vaak beperkt om nieuw onderzoek op te verrichten. Vaak kan men hier enkel maar simulaties op uitvoeren waarvoor de robot gemaakt is terwijl open-source middelen veel uitgebreider zijn.

Een voorbeeld van zo'n open-source middel is Robot Operating System (ROS) [3]. Dit is een framework van Stanford University en Willow Garage om robotsoftware te schrijven. Het bevat een grote verzameling aan bibliotheken, tools en conventies om de ontwikkeling van complexe robotsoftware toegankelijker te maken. Om nieuw ontwikkeld robotmateriaal te testen worden vaak simulaties uitgevoerd in simulatiepakketten binnen ROS. Om dit echt toe te passen in de praktijk is er een echte robot nodig. Om de robot te verbinden met ROS, is er hiervoor een zogenaamde ROS-driver nodig. Deze driver is een ROS-node[1] die commando's gegenereerd met ROS omzet naar commando's die de controller op de robot begrijpt en omzet naar een beweging van de robot zelf, en vice versa. Deze vertaalslag van ROS naar de controller is cruciaal om ontwikkelde tools in praktijk uit te testen.

Op ACRO[2] (Automatisering, Computervisie en Robotica) heeft men een Panasonic robot ter beschikking. Deze robot, afgebeeld in figuur 1, werd in het verleden nog niet aangestuurd vanuit ROS. Hiervoor is er de zogenaamde driver voor nodig.

[1]Een node is een onafhankelijk werkend programma binnen ROS met een bepaalde computationele functie.

[2]ACRO is een onderzoeksgroep van de KU Leuven.




Figuur 1: De Panasonic VR-006L robot, opgesteld in de werkplaats op ACRO, Diepenbeek

Deze Panasonic VR-006L robot is een seriële robot met zes vrijheidsgraden. In het verleden werd deze robot gebruikt als actuator van de autonome appelplukinstallatie [4]. Daarnaast wordt deze veelvuldig gebruikt als lasrobot in de industrie.

## I. Specificaties van een driver voor industriële robots

Om te voldoen aan een ROS-driver, zijn er hiervoor enkele regels/richtlijnen opgelegd door het ROS-Industrial[3] consortium. Dit is vooral belangrijk omdat deze ontwikkelde software door andere ontwikkelaars kan worden gebruikt. Hierdoor moet er een bepaalde standaardisering zijn om te kunnen spreken van een ROS-driver [5]. In het algemeen kan men spreken van twee regels om een ROS-driver te implementeren:

1) Initialisatie. Wanneer men een verbinding wilt leggen met de robotcontroller moet de node die deze verbinding monitort onmiddellijk de connectie willen aangaan. Anders gezegd mag dit niet gebeuren door een manuele service-call. Om dit zo effectief mogelijk te laten werken, stelt het ROS-Industrial consortium voor om een service op de controller van de robot die automatisch runt wanneer de spanning wordt geactiveerd. Deze service kan dan bijvoorbeeld op de achtergrond draaien.
2) Communicatie. Om een zo robuuste driver te implementeren, moet de ROS-controllerconnectie voorbereid zijn op elke vorm van communicatieverlies. Wanneer een vorm van communicatieverlies zich voordoet, moet de ROS-node herconnecteren met de robot. Het ROS-Industrial consortium raadt aan om deze communicatie te hernemen aan een minimale frequentie $f \geq 1Hz$. Ook wanneer er sprake is van communicatieverlies, moet ROS dit informeren door continu foutboodschappen te sturen ("connected=false"). Daarnaast moet de robot stoppen met bewegen totdat er sprake is van een nieuwe vorm van communicatie.

[3]Dit is een consortium waar men zich spitst op het toepassen van ROS op industriële robotica.

## II. Beschikbare ROS-drivers voor industriële manipulatoren

Alvorens een driver te implementeren, is het altijd handig om de reeds gepubliceerde drivers te bestuderen. Onderstaande tabel I geeft een kort overzicht van de mogelijkheden van vier ROS-drivers die al reeds gepubliceerd zijn [6], [7], [8].

Tabel I: Overzicht van de mogelijkheden van enkele beschikbare ROS-driver

| **Robot fabrikant** | **Position Streaming** | **Trajectory Downloading** | **Trajectory Streaming** |
|---|---|---|---|
| Universal Robots | Ja | Nee | Nee |
| Epson | Ja | Nee | Nee |
| Kuka | Ja | Nee | Nee |
| Motoman | Nee | Nee | Ja |
| Panasonic | Ja | Misschien | Nee |

Tabel I geeft drie nieuwe begrippen weer:

- Position Streaming. Een ROS-driver behoort tot deze categorie indien commando's vanuit ROS in real-time geconverteerd worden naar commando's die controller kan begrijpen. Dit commando is dan bijvoorbeeld een vector die een pose voorstelt van de robot:

$$\boldsymbol{q} = [q_1, q_2, \ldots, q_n]^T \qquad (1)$$

  Dit is vaak de makkelijkste weg om een driver te implementeren. De robotcontroller zelf beslist tegen welke snelheid de robot naar de opgegeven pose gaat.
- Trajectory Downloading. Wanneer alle ROS-commando's in één ruk naar de controller worden gestuurd en wanneer deze commando's door de controller worden verwerkt, spreekt men van Trajectory Downloading. Deze interface wordt gebruikt wanneer Position Streaming niet lukt. Deze techniek is echter trager dan de vorige aangezien deze niet in real-time wordt verwerkt.
- Trajectory Streaming. Dit is een combinatie van de vorige twee interfaces. Hierbij bepaalt ROS

tegen welke snelheid de robot beweegt en niet de controller.

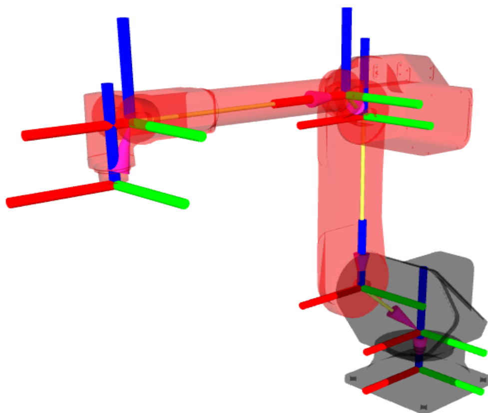
Naast deze drie types van interfaces, heeft men nog de IO-Control en de Torque-Control. Bij de IO-Control is het mogelijk om vanuit ROS ingangen te lezen en uitgangen te schrijven. Deze IO's zijn bijvoorbeeld de vacuümdetectiesensor van de perslucht of het activeren van een lasstroom bij lasrobots. Bij Torque Control kan men het koppel bepalen vanuit ROS dat nodig is om de robot op een bepaalde manier te laten versnellen of vertragen. Deze functie wordt door ROS-Industrial nog niet ondersteund.

## III. Robotmodel van de Panasonic VR-006L voor ROS

Om de robot in simulatie te zien, heeft ROS een ingebouwde plug-in om robots te simuleren en visualiseren. Deze plug-in heet RViz (ROS Visualizer). Om dit te kunnen heeft RViz een URDF-bestand nodig.

### A. URDF

Een URDF (Unified Robot Description Format) beschrijft de kinematische structuur van de robot in een XML-bestand [9], [10]. Hierin kan men ROS meedelen hoe de joint-frames ten opzichte van elkaar liggen en wat de geometrische en mechanische eigenschappen zoals traagheidsmoment en massacentrum zijn. Om deze URDF te visualiseren, moet er eerst hiervoor een launch-file geschreven worden die de RViz-node en de URDF lanceert. Figuur 2 beeldt het simulatiemodel van de Panasonic VR-006L robot af in RViz.



Figuur 2: Visualisatie van de Panasonic VR-006L met de joint-frames in RViz

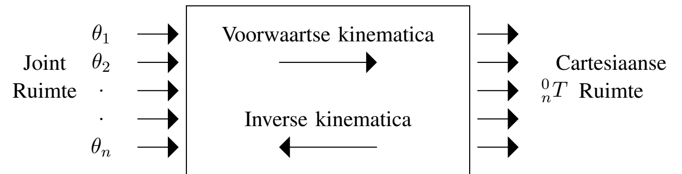
Om een joint-frame juist te leggen gerefereerd naar het vorige joint-frame kan men gebruik maken van de Denavit-Hartenberg conventie [11]:

$$[T] = [Z_1][X_1][Z_2][X_2]...[X_{n-1}][Z_n][X_n] \qquad (2)$$

waarbij $[Z]$ de jointtransformatie is en $[X]$ de linktransformatie.

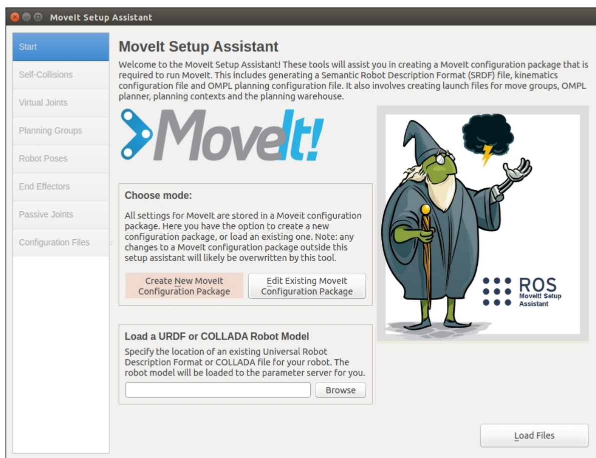
### B. Padplanning met MoveIt!

Zoals uitgelegd in de inleiding richt het onderzoek rond robotica zich momenteel op de toepassing van padplanningsalgoritmes op manipulatoren. Deze padplanningsalgoritmes genereren botsingsvrije paden om van positie X naar positie Y te gaan. Padplanningstools analyseren de beweging en proberen objecten te vermijden door gebruik te maken van de voorwaartse en inverse kinematica op een robot. Figuur 3 legt het verschil uit tussen de voorwaartse en de inverse kinematica.



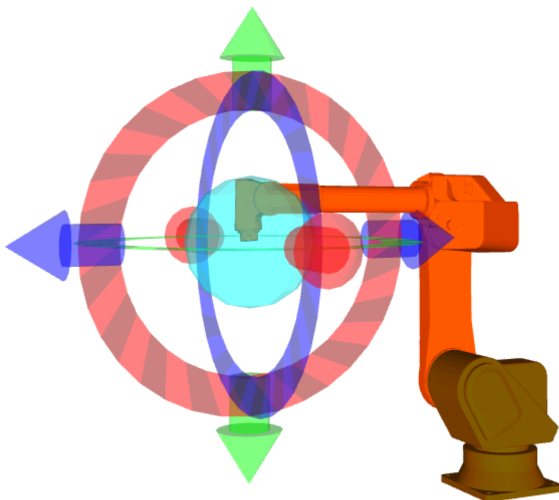

Figuur 3: Schematische voorstelling van de voorwaartse en inverse kinematica

De voorwaartse kinematica (FK) beschrijft de pose (positie en orientatie) van een eindeffector gegeven de joint variabelen van een robot. De inverse kinematica (IK) daarentegen beschrijft de joint variabelen gegeven de pose van de eindeffector. De inverse kinematica is computationeel moeilijker aangezien voor een bepaalde pose verschillende joint variabelen gevonden kunnen worden die leiden tot dezelfde oplossing. Om dit debacle wat te vergemakkelijken, is er binnen ROS een padplanningstool genaamd MoveIt!. MoveIt! maakt gebruik van verschillende IK-solvers (kdl, IKFast, TracIK, ...) om de pose te beschrijven in termen van de jointcoördinaten. Om dit te doen, heeft MoveIt! een *Semantic Robot Description Format* (SRDF)-bestand nodig. Deze SRDF bevat informatie over de gewrichten, geledereen en/of verschillende geledereen in collisie met elkaar kunnen optreden of niet. Dit SRDF-bestand kan aangemaakt worden door middel van een URDF-bestand in te geven als input in de *MoveIt! setup assistant*, zie figuur 4.




Figuur 4: Schermafbeelding van de MoveIt! setup assistant

Uit deze MoveIt! setup assistant komt een MoveIt! configuratie folder waar men de robot kan gebruiken om padplanningsimulatie uit te voeren. Onderstaande figuur geeft de Panasonic VR-006L robot weer in MoveIt!.



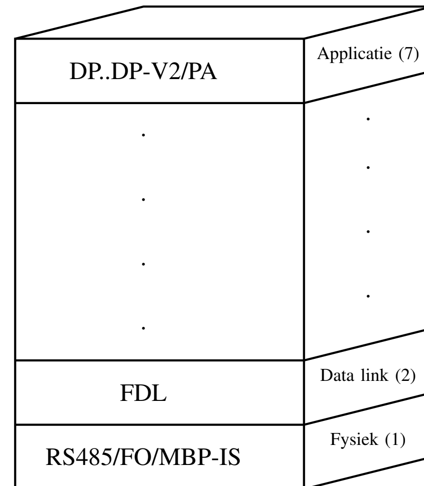
Figuur 5: Schermafbeelding van de Panasonic VR-006L robot weergegeven in MoveIt!

## IV. PROFIBUS

Wanneer naar de reeds gepubliceerde drivers wordt gekeken, valt er één ding op. Bij deze drivers wordt er altijd gebruik gemaakt van een Ethernet communicatie tussen ROS en de robotcontroller. Met behulp van Ethernet kunnen trajecten worden verzonden vanuit ROS naar de controller. Bij de Panasonic VR-006L echter, heeft men de mogelijkheid niet om gebruik te maken van deze Ethernetcommunicatie en is men genoodzaakt om communicatie over een andere veldbus te gebruiken. De Panasonic controller beschikt immers wel over een PROFIBUS netwerkkaart. In dit deel worden de belangrijkste facetten van een PROFIBUS-communicatie opgesomd.

### A. *Het PROFIBUS protocol*

PROFIBUS gebruikt drie van de zeven lagen van het OSI-model voor communicatie [12], [13]. Deze drie lagen worden weergegeven in onderstaande figuur 6.




Figuur 6: Het zevenlagen OSI-model, toegepast op PROFIBUS

Deze drie lagen zijn respectievelijk laag 1, de fysieke laag, laag 2, de data linklaag en laag 7, de applicatielaag. De fysieke laag beschrijft de transmissietechniek waar er kan gekozen worden tussen RS-485, MBP-IS of glasvezel. Laag 2 beschrijft het PROFIBUS communicatie protocol. Laag 7 ten slotte beschrijft de verschillende applicaties waar PROFIBUS kan gebruikt worden.

### B. *De data linklaag*

In het algemeen is PROFIBUS een client-server network waarbij twee protocollen over de datalijnen kunnen lopen. Deze twee protocollen zijn het master-slave protocol en het token-passing principe.

Het master-slave principe is een vorm van handshaking waar een master informatie zendt naar een slave en deze slave antwoordt door informatie terug te zenden. Een slave krijgt nooit zendrecht op de bus, tenzij de master dit expliciet vraagt. Deze communicatie kan gebeuren volgens het half-duplex of full-duplex principe. Bij half-duplex wordt er informatie verstuurd en verzonden, maar niet tegelijk, terwijl bij een full-duplex situatie tegelijkertijd kan verzonden als ontvangen worden. Het spreekt voor zich dat deze laatste vorm van communicatie sneller is dan de eerste.

Het tweede protocol dat door PROFIBUS wordt ondersteund, is het token-passing principe. Bij dit principe wordt er verondersteld dat er zich meerdere

masters en meerdere slaves zich op de bus bevinden. Token-passing zorgt ervoor dat iedere master een zekere spreektijd krijgt om met zijn slaves te communiceren. Wanneer zijn spreektijd om is, mag de volgende master zijn slaves aanspreken. Om te zorgen dat er zich geen tijdsvertragingen voordoen, worden aan de masters een zo laag mogelijk adres gegeven, terwijl aan de slaves een zo hoog mogelijk adres wordt toegekend zodat de tokenring zo snel mogelijk wordt gesloten.

Communicatie over PROFIBUS gebeurt in de vorm van telegrammen. Deze telegrammen worden opgebouwd in de vorm van UART-karakters. Dit karakter is 11 bits lang en zorgt ervoor dat de ontvanger, door een UART-telegram te ontvangen, tijdelijk gesynchroniseerd wordt met de zender zodat de overdracht van informatie correct verloopt. Elke karakter bestaat uit een startbit, 8 databits, een pariteitsbit en een stopbit. Binnen PROFIBUS onderscheidt men vier types van telegrammen:

- een SD1-telegram. Dit telegram heeft een vaste lengte en geen databits. Het wordt gebruikt om te bevestigen wanneer een boodschap correct is aangekomen;
- een SD2-telegram. Dit telegram wordt gebruikt om data te verzenden over het netwerk. Dit telegram heeft in tegenstelling tot het SD1-telegram een variabel dataveld waarin maximaal 246 octetten aan informatie kan worden verzonden;
- een SD3-telegram. Dit telegram heeft een vaste lengte van 8 bits en wordt gebruikt om informatie over het netwerk te versturen waarvan de inhoud niet verandert;
- een SD4-telegram. Dit telegram wordt gebruikt om de tokenring van master naar master te laten gaan. In tegenstelling tot de andere telegrammen is hierop geen controle. Wanneer de volgende master niet begint met zenden, wordt het SD4-telegram automatisch terug verzonden.

PROFIBUS ondersteunt tien transmissiesnelheden waarover data kan verzonden worden. Deze transmissiesnelheid is afhankelijk van de transmissietechniek en variëert van 9,6 kbit/s tot 12 Mbit/s. Deze snelheid is een functie van volgende parameters:

$$v_{netwerk} = f(l_{kabel}, \#masters, \#slaves, ...) \quad (3)$$

### *C. Transmissie*

De eerste laag uit het OSI-model is de fysieke laag. Dit omvat de verschillende transmissietechnieken gebruikt bij PROFIBUS. Binnen PROFIBUS kunnen drie technieken worden gebruikt: RS-485, MBP-IS en glasvezel.

RS-485 is een seriële transmissietechniek waarbij men data kan zenden door een differentieel potentiaal te creëren. Bij deze techniek kan men gebruik maken van twee (half-duplex) of vier (full-duplex) draden. Deze draden worden getwist over elkaar verbonden van het ene apparaat naar het andere. Door deze twisted pairs worden storingen geannuleerd en wordt de data probleemloos over de bus verzonden. Om de spanning stabiel te houden, wordt op het einde van iedere segment een afsluitweerstand geplaatst (typisch 120 Ω). Met deze techniek kunnen bussnelheden van 9,6 kbit/s tot en met 12 Mbit/s worden gehaald over respectievelijk 1200 tot 100 meter.

Daarnaast wordt de MBP-IS (Manchester Bus Powered Intrinsic Safe) vaak gebruikt in explosiegevaarlijke zones. Het gebruik van deze transmissietechniek limiteert echter de bussnelheid (31,25 kbit/s) en de transmissieafstand (1900 meter). Een groot verschil met de overige transmissietechnieken is dat er ook een voedingslijn over de bus loopt.

Een derde transmissietechniek die kan gebruikt worden bij PROFIBUSis glasvezel. Door deze optische techniek kunnen afstanden worden gehaald van 100 kilometer met een transmissiesnelheid vergelijkbaar aan die van de RS-485 transmissietechniek. Door het gebruik van glasvezel is dit systeem meer redundant tegen storingen, vandaar dat ook de transmissielengte groter is.

### *D. Toepassingen*

De PROFIBUS techniek wordt in een wijd spectrum gebruikt binnen de automatiseringswereld. Door het gebruik van deze veldbustechniek, zijn deze kosten-efficiënter dan het gebruik van bedrade logica. Tabel II geeft nog een aantal voorbeelden waar het gebruik van PROFIBUS wordt ingezet.

Tabel II: Overzicht van de verschillende toepassingen waar PROFIBUS wordt gebruikt

| **Industriën waar PROFIBUS gebruikt wordt** | **Toepassing** |
|---|---|
| Procesautomatisering | Zuiveringsinstallaties<br>Chemische industriën<br>Papier-en textielbedrijven |
| Energieopwekking | Power plants<br>Switch gears |
| Machinebouw | Car manufacturing<br>Bottling systemen<br>Stockagesystemen<br>lasinstallaties (robots) |
| Bouwautomatiseing | Verkeerskunde<br>Heat, ventilating en air conditioning (HVAC) |

### *E. Open-source doeleinden rond PROFIBUS*

PROFIBUS is gestandaardiseerd onder DIN 19245, EN 50170 en IEC 61158 [14]. Dit wil zeggen dat fabrikanten die gebruik maken van een PROFIBUS-compatibel product deze normen ontvangen en moeten respecteren. Dit is handig wanneer men met gestandaardiseerde apparaten en producenten werkt. Echter, wanneer men voor nieuwe onderzoekstopics een PROFIBUS-communicatie moet opstellen, komt men wat te kort. Een nood naar open-source PROFIBUS implementaties is zeer sterk aangeraden. Hieronder worden de voornaamste open-source implementaties rond PROFIBUS weergegegeven.

### *F. Serial*

PROFIBUS maakt gebruik van seriële communicatie. Binnen ROS heeft men een bibliotheek waar een seriële communicatie kan worden opgezet, genaamd *Serial*. Deze bibliotheek is een C++-bibliotheek waarmee men eenvoudig seriële poorten kan gebruiken op computers. USB-poorten en apparaten die RS-232 compatibel zijn, kunnen worden aangesproken op zowel Linux als Windows. PROFIBUS kan worden gebruikt bij toestellen die RS-485 compatibel zijn. Hierdoor is deze ROS-bibliotheek wat aan de beperkte kant om via PROFIBUS een communicatie op te starten [15].

### *G. ROSSerial*

Een tweede bibliotheek binnen ROS is *ROSSerial*. Zoals de Serial-bibliotheek, kan een communicatie worden opgestart door gebruik te maken van een seriële poort. In tegenstelling tot het Serial-pakket is deze bibliotheek beter gedocumenteerd en meer uitgebreid. Deze bibliotheek beschrijft een protocol om standaard ROS-berichten te multiplexen over topics en services via een seriële poort of bussysteem. Door gebruik te maken van dit systeem kunnen Arduino's geïnterfaced worden via ROS. Echter zijn er wat nadelen verbonden aan deze bibliotheek. Ten eerste is de hoeveelheid data wat beperkt. Er kunnen maximaal 512 bytes in één ruk verzonden worden. Daarnaast ondersteunt deze bibliotheek geen 64 bit float datatypes. Een 32 bit datatype kan omgezet worden naar een 64 bit datatype, maar hierdoor kan er communicatieverlies gebeuren. Een derde beperking/nadeel is dat strings worden opgeslagen in unsigned characters. Tot slot kan een array enkel maar doorgestuurd worden wanneer de lengte op voorhand wordt opgegeven en wanneer de pointer gezet wordt. Dit is niet gewenst. Deze beperkingen limiteren het gebruik van dit pakket op robots, maar worden vaak gebruikt bij Arduino's of routers [16].

### *H. PBMaster*

Een derde mogelijkheid om gebruik te maken van een open-source PROFIBUS implementatie is het *PBMaster* project. Deze implementatie is ontwikkeld tijdens een thesis aan de Tsjechische Technische Universiteit in Praag. Met behulp van de PBmaster, is het mogelijk om apparaten te connecteren via PROFIBUS op een Linux-systeem. In dit project wordt een RS-232 naar RS-485 adapter gebruikt om de seriële communicatie om te zetten van PROFIBUS naar de PC. Hoewel dit project een groot potentieel heeft, zorgen patenten van de PROFIBUS International (PI) organisatie ervoor dat deze implementatie niet gepubliceerd wordt [17], [18].

### *I. PyProfibus*

Een laatste, publiek beschikbare open source implementatie rond PROFIBUS is *PyProfibus*. Dit is een software stack, ontwikkeld door Michael Büsch waarin de drie lagen van het OSI-model, met betrekking tot PROFIBUS, in Python geïmplementeerd is. Door deze software stack te downloaden, is het mogelijk om via PROFIBUS slaves aan te spreken en hun ingangen te lezen. Dit project werd onderricht aan twee testen. De eerste test was het cyclisch inlezen van een Siemens ET-200S I/O eiland. De tweede test was het aansturen van I/O's over PROFIBUS op een LinuxCNC, een open-source CNC-controller. In tegenstelling tot het DPMaster-project, is deze bibliotheek wel open-source beschikbaar en te verkrijgen via de Python webpagina of via GitHub [19].

## V. Implementatie van de ROS-driver

Zoals eerder vermeld vertaalt een ROS-driver commando's vanuit ROS naar commando's die de robotcontroller begrijpt. Een eerste logische stap hierin is het maken van een conversieprogramma. Dit conversieprogramma vertaalt commando's vanuit ROS en zet deze in een csr-bestand. Zo'n csr-bestand is een tekstbestand dat gebruikt wordt door Panasonic® waar de verschillende commando's van het robotprogramma worden opgeslagen in een leesbaar formaat. Wanneer men in het bezit is van zo'n csr-bestand, kan deze makkelijk met behulp van de DTPS-software[4] worden overgezet naar de robotcontroller en kan men dit programma laten uitvoeren op de robot. Aan de ene kant kan men niet spreken van een ROS-driver aangezien er geen sprake is van een online communicatie tussen ROS en de robotcontroller. Aan de andere kant is dit een goede stap richting een online ROS-driver. Figuur 7 geeft een schema weer dat de overgang van ROS naar een csr-bestand beschrijft.

In figuur 7 is te zien dat commando's worden gegeneerd door ROS. Deze commando's kunnen ook worden gegeneerd door de padplanningstool MoveIt! in samenwerking met ROS. In MoveIt! genereert een padplanner een punt waar de eindeffector van de robot naartoe moet bewegen. Met de gekende positie in Cartesiaanse coördinaten ($X, Y, Z, R_X, R_Y$ en $R_Z$) berekent de padplanner de oriëntatie van ieder gewricht om de gewenste Cartesiaanse positie te bekomen.

De volgende stap is het vertalen van deze commando's en het omzetten naar een csr-bestand. Een csr bestaat uit een tekstbestand waar op een structurele manier informatie over het robotprogramma wordt opgeslagen. In dit bestand wordt bijvoorbeeld informatie opgeslagen over de verschillende poses die de robot moet bereiken tijdens het bewegen, maar het type eindeffector (lastoorts, grijper, ...) wordt hier ook vermeld. De poses hierin worden gedefinieerd in termen van joint-space variabelen, i.e. een vector van hoeken waarmee de verschillende gewrichten moeten staan. De commando's die gegeneerd worden vanuit ROS kunnen echter in joint of Cartesiaanse ruimte gedefinieerd worden. Wanneer deze gegeneerd worden in Cartesiaanse ruimte, moeten deze eerst worden omgezet naar joint-space door gebruik te maken van de inverse kinematica.

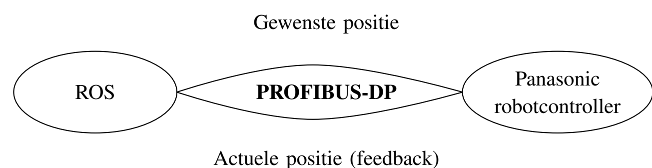
Wanneer men beschikt over zo'n csr-bestand, kan deze gemakkelijk worden overgezet naar de robot door gebruik te maken van de DTPS-software en het touchpad van de robot. In de DTPS-software kan men het geconverteerd programma nog eens simuleren om te kijken of dit klopt alvorens dit op de robot te laten uitvoeren. Het omzetten naar de robotcontroller kan door gebruik te maken van een geheugenkaart. Op de touchpad kan men het gewenste programma kiezen en uitvoeren.

[4]Desktop Programming and Simulation Software. Dit is het computerprogramma van Panasonic® waar men robotprogramma's kan maken en simuleren.

### A. *Bidirectionele communicatie*

#### *1) Strategie*

Een logische volgende stap is het opzetten van een bidirectionele interface tussen ROS en de Panasonic robotcontroller. Hierin wordt een gewenst Cartesiaans punt vanuit ROS gegenereerd en verstuurd als data over het PROFIBUS-netwerk naar de robotcontroller. De robotcontroller ontvangt deze data en zet de data om in een opgelegde beweging. Figuur 8 geeft deze interface schematisch weer.

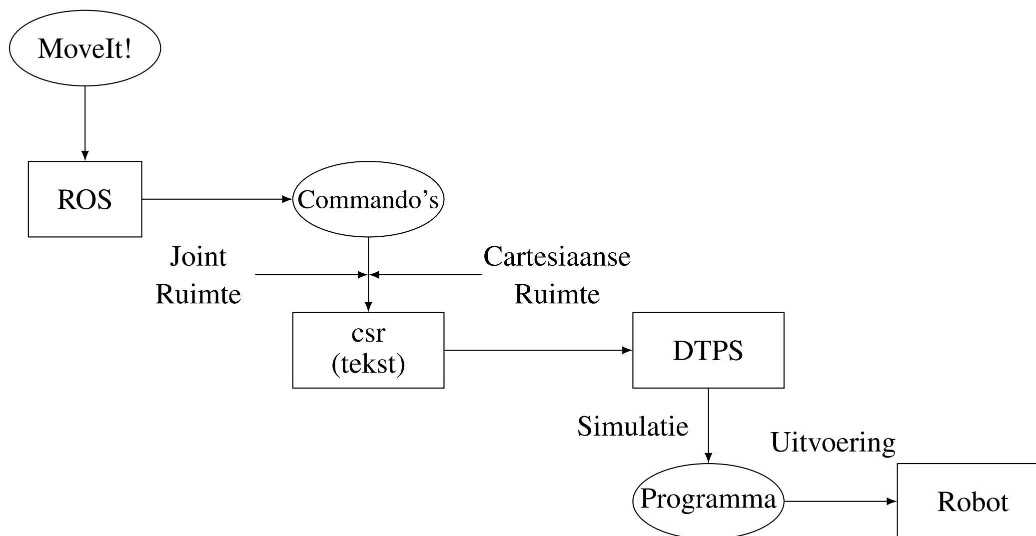



Figuur 8: Bidirectionele opstelling tussen ROS en de Panasonic robotcontroller

Zoals uit figuur 8 kan worden waargenomen wordt de Cartesiaanse positie doorgestuurd door de master (ROS) die de slave (robot) aanspreekt. Aanspreken in deze context betekent dat de master bepaalde ingangen van de slave logisch 1 of logisch 0 maakt. Om dit te verwezenlijken, wordt op de PROFIBUS-kaart van de slave 14 input- en outputbytes geconfigureerd die instaan voor het dataverkeer. Zoals eerder vermeld worden over het PROFIBUS-protocol datatelegrammen verstuurd. Deze telegrammen bevatten de nodige informatie die de master aan de slave meegeeft en omgekeerd. Om dit op een zo robuuste manier te laten verlopen, moet er hiervoor een soort van intern protocol worden voorzien. Onderstaande tabel III geeft dit overzichtelijk weer.

Tabel III: Schematische weergave van het intern protocol tussen ROS en de Panasonic robotcontroller

| **Bytenummer** | **Betekenis** |
|---|---:|
| Byte 1-2 | X-positie |
| Byte 3-4 | Y-positie |
| Byte 5-6 | Z-positie |
| Byte 7-8 | $R_X$-oriëntatie |
| Byte 9-10 | $R_Y$-oriëntatie |
| Byte 11-12 | $R_Z$-oriëntatie |
| Byte 13 | vlag (1: request, 0: geen request) |
| Byte 14 | type beweging |
| **Totaal aantal I/O's** | **112 I/O's** |




Figuur 7: Schematische weergave van de conversie van ROS naar een csr-bestand

Zoals tabel III beschrijft, worden zes bytes gereserveerd voor de Cartesiaanse positie en zes bytes voor de Cartesiaanse oriëntatie van een gewenst punt. Bytenummer 13 is voorbehouden voor de vlag. Deze vlag kan beschouwd worden als een verzoek om data te versturen of niet (zie later). Byte 14 is voorbehouden voor het type beweging dat de robot moet uitvoeren.

*2) Implementatie*

Zoals beschreven in II worden ROS-drivers opgesplitst in drie types. In deze bidirectionele interface wordt een position streaming interface geïmplementeerd. ROS genereert een positiecommando en zendt deze naar de robotcontroller. Wanneer de data is aangekomen, wordt de beweging uitgevoerd door de robot. Onderstaand algoritme beschrijft de aanpak die gevolgd is.

Algoritme 1 geeft de implementatie van het position streaming programma weer in pseudocode. Het programma verwacht 14 inputbytes dat in telegramvorm wordt verzonden over het PROFIBUS-netwerk. Bytenummer 13 is de vlag. Het programma heeft een logische 1 nodig om het programma-afloop uit te voeren. Als dit het geval is, gaat de robot naar zijn initiële homepositie (om veiligheidsredenen wordt de robot telkens naar zijn homepositie gestuurd alvorens een beweging wordt uitgevoerd). Hierdoor weet ROS wat de startpositie van de robot is alvorens de beweging wordt uitgevoerd. Daarna leest het programma de verstuurde positie- en oriëntatiedata (byte 1-6) vanuit ROS in. Hierdoor weet de robotcontroller naar welke Cartesiaanse positie de robot moet bewegen. Byte 14 zorgt ervoor dat in ROS kan gekozen worden op welke manier de robot van zijn startpositie naar zijn nieuwe positie gaat:

**Algoritme 1** Pseudocode van de position streaming interface of de robotcontroller

```
Input: 14 inputbytes op the PROFIBUS-kaart van ROS
Output: 14 outputbytes op de PROFIBUS-kaart naar ROS
Stuur verzoek door vlag hoog te maken
while Connectie == True do
    Stuur huidige positie naar ROS
    if vlag == 1 then
        Lees inputbytes die de Cartesiaanse positie voorstellen
        Lees de inputbytes die de Cartesiaanse oriëntatie voorstellen
        Lees de ingangen die het type beweging van de robot voorstellen (Punt-tot-punt (PTP), Continue Beweging (CB) (lineair of circulair))
        if robot in gewenste positie bevindt then
            zet vlag naar logisch 0
            laat ROS weten wat de actuele positie is door de uitgangen aan te sturen
        end
        Wacht tot de vlag terug hoog staat om volgende beweging uit te voeren
    end
end
```

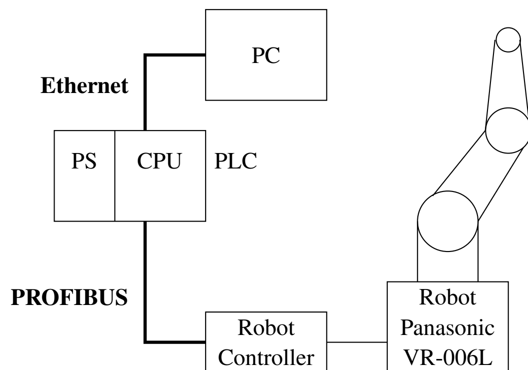
- byte 14 = 1: een punt-tot-punt beweging;
- byte 14 = 2: een continue baan beweging (lineair);
- byte 14 = 3: een continue baan beweging (circulair).

Wanneer alle inputbytes zijn ingelezen, zal de

robotcontroller de robot laten bewegen naar de gewenste positie. Eens aangekomen in het gewenst punt, zal de robot dit laten weten aan ROS door zijn huidige positie te sturen naar ROS. Hierdoor kan ROS het traject simuleren door zijn inverse kinematica toe te passen a.d.h.v. een inverse kinematica-oplosser in MoveIt!. Eens de actuele positie is doorgestuurd, wordt de vlag naar logisch 0 gezet zodat er geen valse boodschappen kunnen worden verstuurd of ontvangen.

*3) Opstelling*

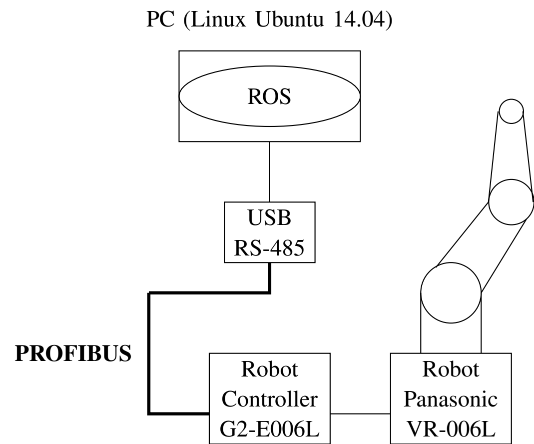
Om de bidirectionele communicatie met de Panasonic robotcontroller te testen, wordt er in eerste instantie gebruik gemaakt van een Siemens PLC-station. Onderstaande figuur 9 geeft deze opstelling schematisch weer.




Figuur 9: Schematische weergave van de communicatie-opbouw tussen een PLC en de robotcontroller

Op de PC wordt met behulp van TIA Portal V13 een PROFIBUS-netwerk opgezet tussen een Siemens S7-300 PLC en de Panasonic robotcontroller. Het GSD-bestand (Geräte Stamm Daten) van de robot wordt in de configuratie toegevoegd als PROFIBUS-slave. Op de robotcontroller wordt de position streaming interface ingeladen. Via deze weg kan het intern protocol getest worden door handmatig ingangen te sturen vanop de PLC.

Een volgende stap is het gebruik maken van de effectieve real-time opstelling tussen ROS en de robotcontroller door rechtsreeks verbinding over PROFIBUS te maken. Figuur 10 illustreert deze opstelling schematisch.




Figuur 10: Schematische weergave van de communicatie-opbouw tussen ROS en de robotcontroller

In figuur 10 zien we dat de verbinding tussen ROS de de robotcontroller tot stand wordt gebracht door een USB naar RS-485 converter te gebruiken. Deze adapter maakt de conversie van het seriële USB protocol naar het seriële RS-485 protocol dat wordt ondersteund door het PROFIBUS-protocol. Deze adapter, afgebeeld in figuur 11 zet het USB-signaal om in een RS-485-signaal waar UART-tekens worden verstuurd door middel van een differentieel spanningssignaal (zie paragraaf IV-C).

Figuur 11: De gebruikte USB naar RS-485 converter, afgebeeld samen met de fysische connectie tussen de RS-485 en de PROFIBUS-aansluiting

De snelheid van het dataverkeer van deze adapter hangt volledig af van de snelheid van het seriële protocol van USB. De adapter is USB 2.0 compatibel, waarbij een theoretische snelheid van 12 Mbps of 1.5 MB/s kan gehaald worden (zonder de conversie van de twee protocollen meegerekend).

## VI. CONCLUSIE

### A. Bijdrage van deze thesis

In deze thesis is een robotmodel opgesteld van de Pansonic VR-006L robot in ROS. Dit model kan gebruikt worden voor simulaties binnen ROS en MoveIt!. Daarnaast is een ROS-driver ontwikkeld die de vertaalslag maakt tussen ROS-commando's en commando's die door de Panasonic G2 robotcontroller kan worden begrepen, en vice versa. Hierbij werd gebruikt gemaakt van een open-source implementatie om informatie over het PROFIBUS-netwerk te verzenden vanaf ROS. Daarnaast kan deze driver ook gebruikt worden om een connectie aan te gaan met PLC's of I/O-eilanden die geconfigureerd staan als slave.

### B. Toekomstig werk

Tijdens deze thesis werden enkele moeilijkheden ondervonden die de implementatie van de driver wat bemoeilijkten. De G2-controller van de Panasonic robot is wat aan de oude kant. Deze controller bezit een communicatiekaart waarmee een PROFIBUS-connectie kan gemaakt worden met een externe master. Over dit netwerk zijn de mogelijkheden aan de beperkte kant. Er kunnen enkel Cartesiaanse poses worden verstuurd naar de robot. Hierdoor converteert de robotcontroller zelf deze pose naar een verdraaiing van ieder gewricht door gebruik te maken van zijn inverse kinematica. Uit het standpunt van ROS bekeken, is het interessanter om zelf de inverse kinematica uit te voeren en een pose door te sturen naar de robot in jointcoördinaten. Dit kan door een G3-controller aan te schaffen met een Ethernetinterface. Hierdoor kan wel zelf de inverse kinematica worden toegepast in ROS. Ook is het mogelijk om volledige programma's door te sturen naar de robot.

## REFERENTIES


[1] IFR, "Executive Summary WR 2017 Industrial Robots," IFR, Tech. Rep., 2017.

[2] E. Demeester, J. De Maeyer, and B. Moyaers, "Cartesian Path Planning for Welding Robots: Evaluation of the Descartes Algorithm," Limassol, Cyprus, 2017, p. 8.

[3] Willow Garage, "About ROS," 2007. [Online]. Available: https://www.ros.org/about-ros/

[4] J. Baeten, K. Donné, S. Boedrij, W. Beckers, and E. Claesen, "Autonomous fruit picking machine: A robotic apple harvester," in *Springer Tracts in Advanced Robotics*, vol. 42, 2008, pp. 531–539.

[5] "ROS-Industrial Driver Specification." [Online]. Available: https://wiki.ros.org/Industrial/Industrial{_}Robot{_}Driver{_}Spec

[6] YASKAWA, "Motoplus-ROS Incremental Motion interface Engineering Design Specifications," YASKAWA. Motoman Robotics., Tech. Rep., 2017.

[7] "Supported Hardware," 2015. [Online]. Available: http://wiki.ros.org/Industrial/supported{_}hardware

[8] L. Castelli and W. Van der Aelst, "Ontwikkeling van een ROS-driver voor Epson C3 robots," Ph.D. dissertation, UHasselt, KULeuven, 2016.

[9] A. Yousuf, W. Lehman, M. A. Mustafa, and M. M. Hayder, "Introducing kinematics with robot operating system (ROS)," *2015 122nd ASEE Annual Conference and Exposition*, vol. 122nd ASEE, no. 122nd ASEE Annual Conference and Exposition: Making Value for Society, 2015. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84941996020{\&}partnerID=40{\&}md5=6426430ef2e219b5ae62adf4bb08bc21

[10] G. Walck, "Introduction to Robot Modeling in ROS," 2015.

[11] P. Corke, "Denavit-Hartenberg notation for common robots," Tech. Rep., 2014.

[12] M. Popp, *The New Rapid Way to Profibus DP*, 2003.

[13] PNO, *Technology and Application*. PNO, Profibus, 2002.

[14] H. Kleines, K. Zwoll, M. Drochner, and J. Sarkadi, ",Integration of Industrial Automation Equipment in Experiment Control Systems via PROFIBUS - Developments and Experiences at Forschungszentrum Jiilich," Zentrallabor fur Elektronik, Forschungszentrum Jiilich, Germany, Tech. Rep., 1999.

[15] "Serial," 2014, last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/serial

[16] "ROS Serial," 2016, last visited on 20-03-2018. [Online]. Available: http://wiki.ros.org/rosserial

[17] P. Trnka and P. Smolik, "Profibus DP Master for PC," Czech Technical University in Prague, Tech. Rep., 2004.

[18] T. D. Khanh, "An Open Implementation of Profibus DP," Ph.D. dissertation, Czech Technical University in Prague, 2009.

[19] M. Büsch, "PROFIBUS software stack," 2016. [Online]. Available: https://bues.ch/cms/automation/profibus.html

# Appendix B

# Complete URDF model

```
<?xml version="1.0"?>
<!-- The URDF file of the Panasonic VR-006L. Here the
   kinematic and geometric model is generated for the ROS
   driver. First, all the links of the robot are specified,
   followed by the joints. -->

<robot name="panasonic">
  <!--__Description of the colours__-->

    <material name="black">
      <color rgba="0␣0␣0␣1" />
    </material>

    <material name="red">
      <color rgba="0.8␣0␣0␣1" />
    </material>




  <!--__Description of the LINKS__-->

  <link name= "base">
    <visual>
      <origin xyz="0␣0␣0" rpy="0␣0␣0" />
      <geometry>
        <mesh filename="package://panasonic_vr006l_support/
           meshes/cad/baseplatform.stl" scale=".01␣.01␣.01"/>
      </geometry>
      <material name="black" />
    </visual>
  </link>
```

```
<link name= "link1">
  <visual>
    <origin xyz="0.01␣0.014␣0.042" rpy="0␣0␣-1.57" />
    <geometry>
      <mesh filename="package://panasonic_vr006l_support/
         meshes/cad/shoulder.stl" scale=".01␣.01␣.01"/>
    </geometry>
    <material name="black" />
  </visual>
</link>


<link name= "link2">
  <visual>
    <origin xyz="0␣-0.03␣0.16" rpy="0␣0␣1.57" />
    <geometry>
      <mesh filename="package://panasonic_vr006l_support/
         meshes/cad/upperarm_rotate.stl" scale=".01␣.01␣.01
         "/>
    </geometry>
    <material name="red" />
  </visual>
</link>


<link name= "link3">
  <visual>
    <origin xyz="0.01␣0.06␣0.04" rpy="0␣0␣1.57" />
    <geometry>
      <mesh filename="package://panasonic_vr006l_support/
         meshes/cad/elbow_new.stl" scale=".01␣.01␣.01"/>
    </geometry>
    <material name="red" />
  </visual>
</link>


<link name= "link4">
  <visual>
    <origin xyz="0.197␣0␣0.004" rpy="0␣0␣1.57␣" />
    <geometry>
      <mesh filename="package://panasonic_vr006l_support/
         meshes/cad/forearm_new.stl" scale=".01␣.01␣.01"/>
    </geometry>
    <material name="red" />
  </visual>
</link>
```

```
<link name= "link5">
  <visual>
    <origin xyz="0.01␣0␣-0.008" rpy="1.55␣0␣1.57" />
    <geometry>
      <mesh filename="package://panasonic_vr006l_support/
         meshes/cad/frontend_rotate.stl"scale=".01␣.01␣.01"
         />
    </geometry>
    <material name="red" />
  </visual>
</link>


<link name= "end_effector">
  <origin xyz= "0 0 0" />
</link>


<!--__Description of the JOINTS__-->

<joint name="joint_base_link1" type="revolute">
  <parent link= "base"/>
  <child link= "link1"/>
  <origin xyz="0␣0␣0.06" rpy="0␣0␣0"/>
  <axis xyz="0␣0␣1" />
  <limit effort="100" lower="-2.705" upper="2.705" velocity
     ="2.09" />
</joint>


<joint name="joint_link1_link2" type="revolute">
  <parent link= "link1"/>
  <child link= "link2"/>
  <origin xyz="0.06␣-0.048␣0.072" rpy="0␣0␣0"/> <!-- -0.05
     -0.058 0.080-->
  <axis xyz="0␣1␣0␣" />
  <limit effort="100" lower="-1.745" upper="2.618" velocity
     ="2.09" />
</joint>


<joint name="joint_link2_link3" type="revolute">
  <parent link= "link2"/>
  <child link= "link3"/>
  <axis xyz="0␣1␣0" />
  <origin xyz="0␣-0.002␣0.243" rpy="0␣0␣0"/>
  <limit effort="100" lower="-3" upper="1.3" velocity="2.09
     " />
</joint>
```

```
  <joint name="joint_link3_link4" type="revolute">
    <parent link= "link3"/>
    <child link= "link4"/>
    <origin xyz="0.085␣0.054␣0.0425" rpy="0␣0␣0"/>
    <axis xyz="1␣0␣0" />
    <limit effort="100" lower="-3.316" upper="4.712" velocity
       ="2.09" />
  </joint>


  <joint name="joint_link4_link5" type="revolute">
    <parent link= "link4"/>
    <child link= "link5"/>
    <origin xyz="0.265␣0␣0" rpy="0␣0␣0"/>
    <axis xyz="0␣1␣0" />
    <limit effort="100" lower="-4.712" upper="4.712" velocity
       ="5.23" />
  </joint>


  <joint name="joint_link5_endeffector" type="revolute">
    <parent link="link5"/>
    <child link="end_effector"/>
    <origin xyz="0.02␣0␣-0.06" rpy= "0 0 0"/>
    <axis xyz= " 0 0 1"/>
    <limit effort="100" lower="-6.28" upper="6.28" velocity="
       5.23"/>
  </joint>

</robot>
```

# Appendix C

# Kinematic chain of the Panasonic VR-006L robot

Figure C.1 below shows the kinematic chain of the Panasonic VR-006L robot as described in the URDF model.

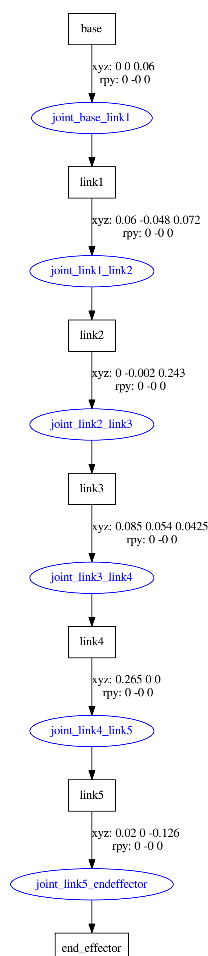



**FIGURE C.1:** Kinematic chain of the Panasonic VR-006L robot, as described in the URDF

# Appendix D

# A Denavit-Hartenberg representation of the Panasonic VR-006L robot

The Denavit-Hartenberg (DH) representation, introduced by Jacques Denavit and Richard Hartenberg in 1955, is an easy convention for attaching frames to the links of a kinematic chain, or a robotic manipulator [29], [30], [42]. Such a robotic manipulator usually possesses six links. Each of these six links are associated with a coordinate frame, going from the base frame (absolute world frame, frame 0) to the frame of the end effector (frame 6 for 6DOF robots). The transformation of frame $i-1$ to frame $i$ matches with either a translation or a rotation. The Panasonic VR-006L robot has got six rotation DOF. Applying the DH representation, the pose of the end effector is calculated using the frames of the links before the end effector. This is called the forward kinematics, as described in section 3.4.

*Formalism*: The DH formalism describes the transformation of frame $i-1$ to frame $i$ with four parameters. These four parameters are, as described in [29] and [42]:

- $\alpha_i$: the angle between the $z_{i-1}$ and the $z_i$ axis about the $x_i$ axis;
- $a_i$: the distance between the $z_{i-1}$ and the $z_i$ axis along the $x_i$ axis;
- $d_i$: the distance from the origin of frame $i-1$ to the $x_i$ axis along the $z_{i-1}$ axis;
- $\theta_i$: the angle between the $x_{i-1}$ and the $x_i$ axis about the $z_{i-1}$ axis.

Frame $i$ follows from frame $i-1$ by following four steps:

1. Rotate frame $i-1$ over angle $\theta_i$ around the $z_{i-1}$ axis;
2. Translate the frame along the $z_{i-1}$ axis by the distance $d_i$;
3. Translate the frame along the $x_i$ axis by the distance $a_i$;
4. Rotate the frame over the angle $\alpha_i$ around the $x_i$ axis.

By following these four steps, the DH representation of any kinematic chain can be formed. A small note: *the DH formalism is not a unique representation*, i.e. that two engineers can form two different correct DH representations of the same robotic manipulator. The four steps, as described above, corresponds to the homogeneous transformation ${}^{i}_{i-1}H$:

$$
{}^{i}_{i-1}H = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\cos\alpha_i & \alpha_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & \alpha_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{D.1}
$$

Figure D.1 gives an overview of the DH frames for the Panasonic VR-006L manipulator. The figure is drawn with all the joint angles in a random orientation.

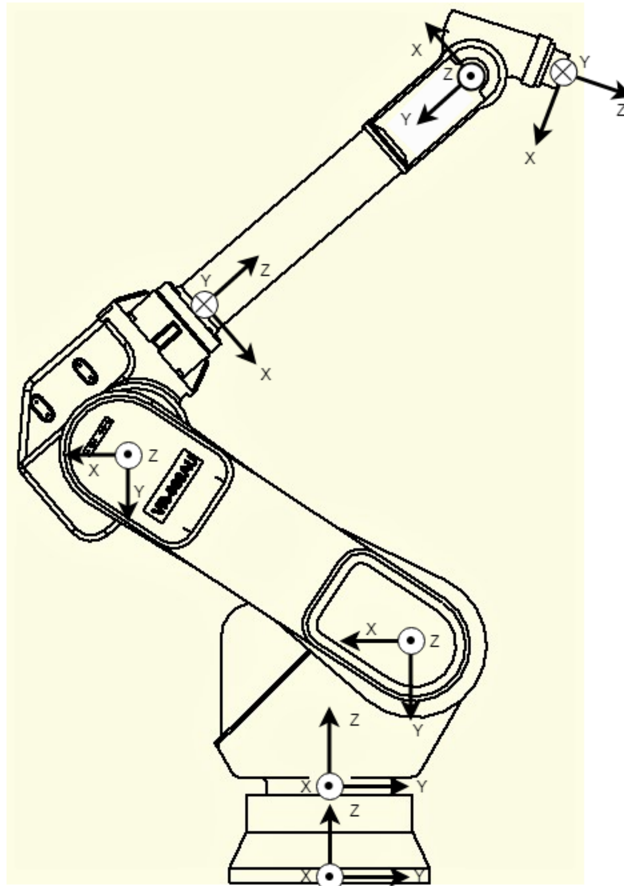



**FIGURE D.1:** Denavit-Hartenberg frames of the Panasonic VR-006L robot

Table D.1 describes the DH parameters for the Panasonic VR-006L given figure D.1.

**TABLE D.1:** DH parameters for a Panasonic VR-006L robot. Distances are represented in mm, angles in rad.

| i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $\alpha_i$ | 0 | $\pi/2$ | 0 | $\pi/2$ | $-\pi/2$ | $\pi/2$ |
| $a_i$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $d_i$ | 120 | 300 | 620 | 100 | 900 | 0 |
| $\theta_i$ | 0 | $-\pi/2$ | 0 | $\pi/2$ | $-\pi$ | $-\pi/2$ |

# Appendix E

# Used code

All the source code of the ROS driver can be consulted on the author's GitHub page (https://github.com/davidDS96/Panasonic-VR-006L-driver) or via https://github.com/JeroenDM. The source code for the offline conversion programme can be consulted via https://github.com/DavidJornThesis/Panasonic_CSR. This appendix contains some of the code that is been used to develop the ROS driver.

## Launch files

```
<launch>
  <include file="$(find␣panasonic_vr006l_moveit_config)/
     launch/demo.launch"/>
</launch>
```

```
<launch>
  <node name="interface" pkg="vr_driver" type="interface.py"
     respawn="false" output="screen">
  </node>
</launch>
```

## Driver code

```
#!/usr/bin/env python
##############################################
# This python program is a bidirectional
# interface between ROS and a Panasonic
# VR-006L controller. In this script
# the user can send cartesian poses
# to the robot through an interface in
# the terminal. The specified movement can
# be previewed in MoveIt!
# The connection is established over the
# Profibus network by using an open implmentation
# of this fieldbus protocol.
#          v--------------v-----------v
#          |   Panasonic  |    ROS    |
#          |     GII      |   Linux   |
#          |  Controller  |           |
#          v--------------v-----------v
#          |     DP       |    DP     |
#          |    Slave     |   Master  |
#          ^--------------^-----------^
#
##############################################

##############################################
#               import libraries             #
##############################################

import sys, threading, math, copy, array, time
from time import sleep
import numpy as np
import rospy
import pyprofibus
from pyprofibus import DpTelegram_SetPrm_Req, monotonic_time
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from geometry_msgs.msg import Pose
import visualization_msgs.msg
from visualization_msgs.msg import Marker


##########################################

'''
The␣function␣getInfoFromTerminal()␣gets␣the␣user␣input␣from␣
   the␣terminal
and␣checks␣if␣the␣input␣is␣in␣between␣the␣safety␣limits␣of␣
   the␣robot.␣These
```

```
limits␣can␣be␣increased␣depending␣on␣the␣available␣robot␣
   working␣space.␣The
user␣can␣specify␣a␣desired␣cartesian␣pose␣and␣can␣choose␣
   between␣a␣point-
to-point,␣a␣linear␣or␣a␣circular␣movement␣between␣start␣and␣
   goal.␣The␣function
returns␣the␣x,y,z␣position␣and␣orientation␣and␣the␣specified␣
   movement␣as␣integers.
'''

def getInfoFromTerminal():
    while True:
        try:
            x = int(input("Give␣an␣integer␣value␣for␣the␣x␣
               position␣in␣mm␣between␣100␣and␣1400\n(home␣x␣
               position␣is␣1120mm)␣:"))
        except ValueError:
            print("Sorry,␣I␣didn't␣understand␣that.")
        if not 100 <= x <= 1400:
            print("Sorry,␣x␣must␣be␣between␣100␣and␣1400.")
            continue
        else:
            break
    while True:
        try:
            y = int(input("Give␣an␣integer␣value␣for␣the␣y␣
               position␣in␣mm␣between␣-900␣and␣900\n(home␣y␣
               position␣is␣0mm):"))
        except ValueError:
            print("Sorry,␣I␣didn't␣understand␣that.")
        if not -900 <= y <= 900:
            print("Sorry,␣y␣must␣be␣between␣-900␣and␣900.")
            continue
        else:
            break
    while True:
        try:
            z = int(input("Give␣an␣integer␣value␣for␣the␣z␣
               position␣in␣mm␣between␣0␣and␣1400\n(home␣z␣
               position␣is␣725mm):"))
        except ValueError:
            print("Sorry,␣I␣didn't␣understand␣that.")
        if not 0 <= z <= 1400:
            print("Sorry,␣z␣must␣be␣between␣0␣and␣1400.")
            continue
        else:
            break
    while True:
        try:
```

```
            Rx = int(input("Give␣an␣integer␣value␣for␣the␣x␣
               orientation␣in␣degrees␣between␣-180␣and␣180\n(
               home␣Rx␣orientation␣is␣0):"))
        except ValueError:
            print("Sorry,␣I␣didn't␣understand␣that.")
        if not -180 <= Rx <= 180:
            print("Sorry,␣Rx␣must␣be␣between␣-180␣and␣180.")
            continue
        else:
            break
    while True:
        try:
            Ry = int(input("Give␣an␣integer␣value␣for␣the␣y␣
               orientation␣in␣degrees␣between␣-180␣and␣180\n(
               home␣Ry␣orientation␣is␣45):"))
        except ValueError:
            print("Sorry,␣I␣didn't␣understand␣that.")
        if not -180 <= Ry <= 180:
            print("Sorry,␣Ry␣must␣be␣between␣-180␣and␣180.")
            continue
        else:
            break
    while True:
        try:
            Rz = int(input("Give␣an␣integer␣value␣for␣the␣z␣
               orientation␣in␣degrees␣between␣-180␣and␣180\n(
               home␣Rz␣orientation␣is␣0):"))
        except ValueError:
            print("Sorry,␣I␣didn't␣understand␣that.")
        if not -180 <= Rz <= 180:
            print("Sorry,␣Rz␣must␣be␣between␣-180␣and␣180.")
            continue
        else:
            break
    while True:
        try:
            mType = int(input("Now␣give␣a␣movetype:␣Point-to-
               point␣(1),␣Continuous-Linear␣(2),␣Continuous-
               Circular␣(3)␣-->␣"))
        except ValueError:
            print("Sorry,␣I␣didn't␣understand␣that.")
            continue
        if not (mType == 1 or mType == 2 or mType == 3):
            print("Sorry,␣movetype␣must␣be␣1␣or␣2␣or␣3.")
            continue
        else:
            break

    return x, y, z, Rx, Ry, Rz, mType
```

```
'''
The␣function␣convertValueFirstByte␣has␣as␣argument␣an␣integer
   ␣value,
converts␣this␣to␣a␣binary␣representation␣and␣returns␣the␣
   first␣byte␣of
this␣integer␣value.
'''

def convertValueFirstByte(value):

    convertedValue = int('{0:016b}'.format(((1<<16) -1) &
       value)[8:],2)

    return convertedValue


'''
The␣function␣convertValueSecondByte␣has␣as␣argument␣an␣
   integer␣value,
converts␣this␣to␣a␣binary␣representation␣and␣returns␣the␣
   second␣byte␣of
this␣integer␣value.
'''

def convertValueSecondByte(value):

    convertedValue = int('{0:016b}'.format(((1<<16) -1) &
       value)[:8],2)

    return convertedValue

'''
The␣function␣convertMovementType␣has␣as␣argument␣the␣
   specified␣movement␣type
value␣that␣the␣user␣specified␣before␣and␣returns␣its␣binary␣
   representation.
'''

def convertMovementType(value):

    convertedValue = int('{0:08b}'.format(value),2)

    return convertedValue

'''
The␣function␣convertFlag␣converts␣the␣number␣'1'␣to␣a␣binary␣
   representation
and␣returns␣this␣binary␣representation.␣The␣flag␣is␣used␣in␣
   the␣program␣to
```

```
have a robust way to communicate with the robot. If the flag 
    isn't set, the
robot will not execute its imposed movement.
'''

def convertFlag():

    convertedValue = int('{0:08b}'.format(1),2)

    return convertedValue

'''
The function getValues uses the functions described above to 
    return an
array of 14 bytes to prepare it to send to the robot when in 
    data exchange
mode. The common I/O travel between ROS and robot controller 
    is specified as
14 bytes (= 112 I/O's). It also returns an array of seven 
    integer numbers that
are used to visualise the specified movement in RViz (with 
    MoveIt!).
'''

def getValues():
    data = getInfoFromTerminal()
    viz_array = data

    x1 = convertValueFirstByte(data[0])
    x2 = convertValueSecondByte(data[0])

    y1 = convertValueFirstByte(data[1])
    y2 = convertValueSecondByte(data[1])

    z1 = convertValueFirstByte(data[2])
    z2 = convertValueSecondByte(data[2])

    Rx1 = convertValueFirstByte(data[3])
    Rx2 = convertValueSecondByte(data[3])

    Ry1 = convertValueFirstByte(data[4])
    Ry2 = convertValueSecondByte(data[4])

    Rz1 = convertValueFirstByte(data[5])
    Rz2 = convertValueSecondByte(data[5])

    flag = convertFlag()
    mType = convertMovementType(data[6])
```

```
    posRotArray = [x1, x2, y1, y2, z1, z2, Rx1, Rx2, Ry1, Ry2
       , Rz1, Rz2, flag, mType]


    return posRotArray, viz_array


'''
The␣function␣setupCommunication␣sets␣up␣the␣connection␣
   between␣ROS␣and␣the␣robot
controller␣over␣the␣PROFIBUS␣network.␣This␣function␣relies␣␣
   on␣a␣config-file,␣which
can␣be␣found␣in␣the␣config␣folder.␣Here,␣the␣configuration␣of
   ␣the␣slave␣with␣the␣master
is␣implemented.␣When␣the␣configuration␣is␣finished,␣the␣slave
   ␣is␣added␣to␣the␣master-slave
network.␣If␣the␣slave␣doesn't␣react␣or␣the␣connection␣is␣lost
   ,␣a␣PROFIBUS␣error␣will␣occur
and␣the␣configuration␣has␣to␣be␣set␣again.
'''

def setupCommunication():

    try:
        config  = pyprofibus.PbConf.fromFile("/home/david/
           robot_ws/src/vr_driver/config/panaprofi.conf")

        phy = config.makePhy()

        master = pyprofibus.DPM1(phy=phy, masterAddr=config.
           dpMasterAddr, debug=False)

        for slaveConf in config.slaveConfs:
            gsd = slaveConf.gsd

            slaveDesc = pyprofibus.DpSlaveDesc(identNumber=
               gsd.getIdentNumber(), slaveAddr=slaveConf.addr
               )
            slaveDesc.setCfgDataElements(gsd.
               getCfgDataElements())
            slaveDesc.setSyncMode(slaveConf.syncMode)
            slaveDesc.setFreezeMode(slaveConf.freezeMode)
            slaveDesc.setGroupMask(slaveConf.groupMask)
            slaveDesc.setWatchdog(slaveConf.watchdogMs)

        master.addSlave(slaveDesc)

        master.initialize()
        return master, slaveDesc
```

```
    except pyprofibus.ProfibusError as e:
        print("Terminating:␣%s" % str(e))
        rospy.logerr("Communication␣is␣lost.␣Try␣to␣
           initialize␣again")

        return None, None

'''
The␣function␣sendValues␣uses␣the␣information␣of␣the␣
   setupCommunication␣function␣to
send␣the␣Cartesian␣pose␣and␣specified␣movement␣type␣to␣the␣
   robot.␣The␣function␣returns
the␣output␣that␣the␣slave␣sends␣to␣the␣master␣in␣data␣
   exchange.␣The␣output␣consists␣of
14␣bytes.␣The␣first␣6␣bytes␣of␣this␣array␣contains␣the␣
   Cartesian␣position␣of␣the␣robot
when␣it␣has␣reached␣his␣imposed␣Cartesian␣pose.
'''

def sendValues(master, slaveDesc, dataArray):
    outputArray = None
    while (outputArray is None):
        outputArray = master.runSlave(slaveDesc, dataArray)

    return outputArray



'''
The␣function␣deformatOutput()␣has␣as␣argument␣the␣output␣that
   ␣the␣slave␣send␣to␣the
master␣and␣return␣an␣integer␣number␣that␣respresents␣the␣
   Cartesian␣position␣of␣the
robot:␣[X,Y,Z]^{T}.␣This␣representation␣is␣needed␣to␣
   visualise␣the␣outcome␣in␣RViz.
'''
def deformatOutput(newArray):
    x1 = '{0:08b}'.format(newArray[0])
    x2 = '{0:08b}'.format(newArray[1])
    y1 = '{0:08b}'.format(newArray[2])
    y2 = '{0:08b}'.format(newArray[3])
    z1 = '{0:08b}'.format(newArray[4])
    z2 = '{0:08b}'.format(newArray[5])

    X = int(x1+x2,2)
    Y = int(y1+y2,2)
    Z = int(z1+z2,2)

    return [X, Y, Z]
```

```
'''
The␣function␣visualiseMovement()␣has␣as␣argument␣the␣array␣of
   ␣integer␣numbers␣that
corresponds␣to␣the␣Cartesian␣position␣and␣orientation␣of␣a␣3D
   ␣point.␣MoveIt!␣shows␣the
movement␣in␣simulation␣together␣with␣RViz.
'''

def visualiseMovement(viz_array):

    moveit_commander.roscpp_initialize(sys.argv)

    robot = moveit_commander.RobotCommander()
    scene = moveit_commander.PlanningSceneInterface()
    group_name = "manipulator"
    group = moveit_commander.MoveGroupCommander(group_name)
    display_trajectory_publisher = rospy.Publisher('/
       move_group/display_planned_path', moveit_msgs.msg.
       DisplayTrajectory, queue_size=20)

    pose_target = geometry_msgs.msg.Pose()
    pose_target.position.x = float(viz_array[0])/(1000*2.604)
            # convert to meter for ROS
    pose_target.position.y = float(viz_array[1])/1000
    pose_target.position.z = float(viz_array[2])/(1000*2.487)

    pi = math.pi
    pose_target.orientation.x = (float(viz_array[3])*pi)/180
    pose_target.orientation.y = (float(viz_array[4])*pi)/180
       - pi/4
    print pose_target.orientation.y
    pose_target.orientation.z = (float(viz_array[5])*pi)/180

    group.set_pose_target(pose_target)

    plan = group.plan()

    display_trajectory = moveit_msgs.msg.DisplayTrajectory()
    display_trajectory_publisher.publish(display_trajectory)
    group.execute(plan)

    moveit_commander.roscpp_shutdown()


'''
The␣function␣setMarker()␣uses␣the␣data␣that␣the␣robot␣sends␣
   to␣ROS␣to␣set␣a␣Marker
in␣RViz.␣This␣Marker␣specifies␣the␣goal␣position␣of␣the␣robot
   's␣movement␣when␣the
```

```
robot␣has␣executed␣his␣trajectory␣and␣reported␣it␣to␣ROS.
'''

def setMarker(viz_array):

    start = time.time()

    period = 5

    while not rospy.is_shutdown():

        topic = 'visualization_marker'
        marker_publisher = rospy.Publisher(topic, Marker,
           queue_size=10)

        sphere_marker = Marker()
        sphere_marker.header.frame_id = "/base"

        sphere_marker.type = Marker.SPHERE
        sphere_marker.action = Marker.ADD

        #sphere_marker.ns = "my_marker"
        #sphere_marker.id = 0

        sphere_marker.scale.x = 0.05
        sphere_marker.scale.y = 0.05
        sphere_marker.scale.z = 0.05

        sphere_marker.color.a = 1.0
        sphere_marker.color.r = 0.0
        sphere_marker.color.g = 0.0
        sphere_marker.color.b = 1.0

        sphere_marker.pose.position.x = float(viz_array[0])
           /(1000*2.604)
        sphere_marker.pose.position.y = float(viz_array[1])
           /1000
        sphere_marker.pose.position.z = float(viz_array[2])
           /(1000*2.487)
        sphere_marker.pose.orientation.w = 1.0

        marker_publisher.publish(sphere_marker)

        if time.time() > start + period:
            break



'''
```

```
The␣function␣main␣starts␣the␣ROS␣interface␣node␣that␣
   interacts␣with␣the␣robot␣controller.
First␣it␣starts␣the␣ROS␣node.␣Next␣it␣sets␣up␣the␣
   communication␣with␣the␣robot␣controller.
The␣user␣input␣is␣used␣to␣split␣up␣the␣amount␣of␣data.␣
   Afterwards␣communication␣with␣the
robot␣can␣be␣contracted.
'''

def main():
    try:
        rospy.init_node('interface', anonymous=True)

        rospy.loginfo("Initializing␣position␣streaming␣
           interface")

        rospy.loginfo("Setting␣up␣communication␣with␣
           Panasonic␣robot␣controller")

        master, slaveDesc = setupCommunication()


        if master is not None:

            while True:

                rospy.loginfo("Getting␣user␣information␣from␣
                   terminal")

                dataArray, viz_array = getValues()
                visualiseMovement(viz_array)             #
                   comment this function if visualisation isn
                   't necessary

                rospy.loginfo("Sending␣position␣and␣
                   orientation␣to␣robot")

                outputArray = sendValues(master, slaveDesc,
                   dataArray)


                if outputArray[1] != 0 or outputArray[2] != 0
                    or outputArray[3] != 0 or outputArray[4]
                   != 0 or outputArray[5] != 0:
                    newArray = outputArray
                    pos_robot = deformatOutput(newArray)
                    setMarker(pos_robot)                 # set
                        marker at Cartesian point in RViz
                       when robot has reached desired pose
```

```
                    rospy.loginfo("Robot␣sent␣feedback␣of␣
                       actual␣Cartesian␣position")

                rospy.spin()


    except KeyboardInterrupt:
        rospy.logerr("Keyboardinterrupt")
        rospy.signal_shutdown("KeyboardInterrupt")
        raise


if __name__ == '__main__': main()
```

# Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
**Implementation of an open-source PROFIBUS interface between ROS and a Panasonic robot**

Richting: **master in de industriële wetenschappen: energie-automatisering**
Jaar: **2018**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

**De Schepper, David** **Geutjens, Jorn**

Datum: **3/06/2018**