TCP/IP facilitated flexible robotics controller

Johannes Marthinus Albertus Kotzé

Advisor: H.R. Gerber
Co-advisor: Prof. C.J. Fourie

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science Engineering at the University of Stellenbosch.

March 2007
Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work, and that I have not previously, in its entirety or in part, submitted it at any university for a degree.

Signature:

Date:
Abstract

Robot control by TCP/IP communication is investigated in this thesis for existing robots used in tertiary education. The request for newer software for robotic computer control came from the University of Stellenbosch Industrial Engineering department where existing software dating back to 1988 is still in use.

A thorough investigation into the research and technologies available is followed by a discussion on the proposed software to adhere to the requirement of compatibility with existing languages in use by the department and provide tools to assist in future research in robotic manipulators and control.

The proposed software solution uses a client/server model running over an IP-based network providing online and offline programming with visual feedback by means of video streaming and 3D simulations, developed as separate modules combined into an effective tool for future research and development.

Opsomming

Die beheer van robotte oor 'n TCP/IP kommunikasie netwerk word ondersoek vir die gebruik in tersiêre studie. Die versoek vir nuwe robotbeheersagteware kom van die Universiteit van Stellenbosch se Bedryfsingenieurswese department vir die vervanging van sagteware wat in gebruik is sedert 1988.

'n Volledige ondersoek is gedoen oor bestaande navorsing en tegnologieë, gevolg deur 'n voorgestelde sagtewarepakket wat voldoen aan die vereiste van versoenbaarheid met die bestaande programmeringstale in gebruik deur die departement, en hulpmiddels te verskaf vir verdere navorsing in robotika.

Die voorgestelde sagteware pakket is gebaseer op 'n bediener/kliënt model wat gebruik maak van 'n IP-netwerk vir kommunikasie om 'n omgewing te verskaf wat aanlyn en aflyn programmering van robotte toelaat met visuele voorstelling deur middel van video terugvoer asook 3D simulasies. Die pakket is ontwikkel in aparte modules, geïntegreer in 'n pakket geskik vir toekomstige navorsing in en ontwikkeling van robotika.
Acknowledgements

I would like to thank my mentor, H.R. Gerber, for his unremitting support, understanding and trust throughout the development of this project.

I would like to express thanks to Prof. C.J. Fourie for establishing the need for the research, as well as the assistance provided during the course of this project,

Further I want to give my gratitude to C.W. Möller for listening when I needed to voice my thoughts, helping me to decide on and understand concepts.

To M. Myburgh and my parents, my thanks for their support and understanding throughout the two years, even through the darker times where motivation lacked.

I want to thank my grandfather, J.M.A. Kotzé, for his contribution to my motivation and education, showing me that working hard is worth the trouble when done correctly.

Last, but definitely not least, I would like to thank my Creator for the abilities He blessed me with, and the opportunities He presented me with to learn and take advantage of my gifts.
Contents

LIST OF FIGURES .................................................................................................................. VII
LIST OF TABLES .................................................................................................................. VIII
NOMENCLATURE................................................................................................................ IX

CHAPTER 1 INTRODUCTION ............................................................................................ 1
  1.1 HISTORY OF ROBOTICS ........................................................................................ 1
  1.2 ROBOT PROGRAMMING ....................................................................................... 2
  1.3 ROBOT CONTROL SOLUTIONS ............................................................................ 6
  1.4 MVAL2006 ........................................................................................................... 7

CHAPTER 2 KINEMATICS ................................................................................................. 9
  2.1 INTRODUCTION ..................................................................................................... 9
  2.2 DIRECT-KINEMATICS .......................................................................................... 10
  2.3 INVERSE-KINEMATICS ....................................................................................... 14
  2.4 CONCLUSION ....................................................................................................... 19

CHAPTER 3 SERVER ....................................................................................................... 20
  3.1 INTRODUCTION ..................................................................................................... 20
  3.2 ARCHITECTURE ................................................................................................... 21
  3.3 HARDWARE CONTROLLER .................................................................................. 22
  3.4 MVALSERVER ...................................................................................................... 32

CHAPTER 4 CLIENT ......................................................................................................... 50
  4.1 INTRODUCTION ..................................................................................................... 50
  4.2 COMMUNICATIONS LAYER ................................................................................. 51
  4.3 GRAPHICAL USER INTERFACE ........................................................................... 51

CHAPTER 5 TESTING AND RESULTS ........................................................................... 54
  5.1 OVERVIEW ............................................................................................................ 54
  5.2 DRIVERS ................................................................................................................ 54
  5.3 COMPILER ............................................................................................................. 59
  5.4 CLIENT/SERVER ................................................................................................... 59

CHAPTER 6 CONCLUSION ............................................................................................... 61
List of Figures

Figure 1.1 Robot Programming grouping ................................................................. 2
Figure 1.2 Manual Programming Categorisation .................................................... 3
Figure 2.1 Denavit-Hartenberg parameter explanation ............................................. 10
Figure 3.1 Main window of MVALServer ............................................................... 20
Figure 3.2 MVALServer/MVALClient functional layout ....................................... 21
Figure 3.3 MVALServer block layout .................................................................. 22
Figure 3.4 MVALDriverWrapper internal functional flow for optional movement commands. ................................................................. 29
Figure 3.5 MVALServer's controller block layout ............................................... 32
Figure 3.6 MVALProgram structural layout ........................................................... 38
Figure 3.7 Execution thread flowchart ................................................................. 43
Figure 4.1 Main window of MVALClient ............................................................... 50
Figure 4.2 Simulation window of MVALClient ...................................................... 52
Figure 5.1 CS113 Joint-interpolated tool movement animation and time line ........ 56
Figure 5.2 CS113 Straight line tool movement animation and time line ................... 56
Figure 5.3 RTX joint-interpolated tool movement animation and time line .......... 58
Figure 5.4 RTX straight line tool movement animation and time line .................... 58
List of Tables

Table 2.1 Mathematical symbols summary ................................................................. 9
Table 2.2 Denavit-Hartenberg Parameters ................................................................. 11
Table 3.1 Driver function headers ............................................................................ 24
Table 3.2 Mathematical equations used for robot control and positioning .............. 30
Table 3.3 Program request action summary ............................................................... 35
Table 3.4 Location request action summary ............................................................... 36
Table 3.5 Driver request action summary ................................................................. 36
Table 3.6 Robot request action summary ................................................................. 36
Table 3.7 Parameter descriptions ............................................................................ 39
Table 3.8 Example of generated tokens for MVAL command “base 10,20,-5,45”. ................................................................................................................................. 41
Table 3.9 Commands not supported by MVALServer .............................................. 44
Table 3.10 Execution Status Description ................................................................. 45
Table 3.11 Functions affecting execution of programs ............................................ 45
# Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>ASE</td>
<td>ASCII Scene Export</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>KB</td>
<td>Kilobyte</td>
</tr>
<tr>
<td>MB</td>
<td>Megabyte</td>
</tr>
<tr>
<td>MPL</td>
<td>Mozilla Public License</td>
</tr>
<tr>
<td>MVAL</td>
<td>Micro Vicarm Assembly Language</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RMI</td>
<td>Robot Manipulator Interface</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VAL</td>
<td>Vicarm Assembly Language</td>
</tr>
</tbody>
</table>
Chapter 1 Introduction

1.1 History of robotics

In January 1921 playwright Karel Čapek’s R.U.R. (Rossum’s Universal Robots) opened in Prague, introducing a story about a paradise where machines brought numerous benefits, only to result in a blight of unemployment and social unrest [13][38]. The word “robota” was a murmured response from his brother, who had used the word previously in a short story in response to Karel’s own choice of “labori”, which seemed too academic [16]. With the international success of R.U.R., “robot” was gradually accepted into other languages. The original meaning of “robota” comes from the feudal ages, where it designated the three days of the week peasants had to work the nobleman’s lands for no remuneration [26].

The word “robotics” was first used by the well known writer Isaac Asimov. He became tired of robot stories depicting robots turning on its creators or mankind, and proposed three rules that robots should follow, later adding the ‘zeroth’ law [39]:

| Law Zero | A robot may not injure humanity, or, through inaction, allow humanity to come to harm. |
| Law One  | A robot may not injure a human being, or, through inaction, allow a human being to come to harm, unless this would violate a higher order law. |
| Law Two  | A robot must obey orders given it by human beings, except where such orders would conflict with a higher order law. |
| Law Three | A robot must protect its own existence as long as such protection does not conflict with a higher order law. |

Academic studies on Asimov’s robotic laws raised questions on the validity and practical implications of these laws[20].

In 1956 George C. Devol and Joseph F. Engelberger discussed the writings of Isaac Asimov. Together they developed the first working robot. Engelberger started the robot manufacturing company Unimation Inc. in 1962 and produced the first commercial robot, “Unimate” [13][32]. Thereafter he was often referred to as “the father of robotics”.

General Motors was the first company to acquire and use a Unimate robot in manufacturing. Kawasaki manufactured robots in Japan under license from Unimation from 1968 until their license expired in 1985 [6].
1.2 Robot programming

1.2.1 Overview

Industries require methods to control the robots for the specific applications they are intended. Robot programming in general encompasses different aspects, as shown in Figure 1.1. This section is to a great extent based on the works of G. Biggs et al. [1] and P. Frenger [11].

![Figure 1.1 Robot Programming grouping](image)

1.2.2 Automatic Programming

With automatic programming the user/programmer uses tools to inform the robot about the required tasks. Categories are PbD (Programming By Demonstration), Learning Systems and Instructive Systems.

PbD is the most common method of automatic programming. To this day many robots include “teach pendants” as a controller to enable the user/programmer to move the robot through the required steps to accomplish the required task. Other methods include voice commands and physical gestures.

Learning systems require the robot to learn its tasks by human induction or self exploration. As enhancements are made in robotics and the tasks robots are required to perform are becoming more complicated, robots will be required to learn to optimise their operation by experience.
With instructive systems, robots can be commanded in real-time by human instruction (i.e. voice recognition) to perform previously learned or programmed tasks. This type of programming must therefore be accompanied by other methods described in this chapter.

### 1.2.3 Manual Programming

Another method more commonly used in the industry is manual programming. This type of programming allows the programmer to give specific commands to the robot which it must follow. This can be further broken down into textual and graphical programming.

![Figure 1.2 Manual Programming Categorisation](image)

#### 1.2.3.1 Textual programming

Textual languages require the programmer to develop the programs by hand. These programs can be either interpreted or compiled. The program may be uploaded to the robot to be executed by an internal processor, or be executed by an external controller with only the necessary commands sent to the robot’s internal controller.

Controller-specific languages (Figure 1.2) are normally limited to a specific robot manipulator or controller, providing specific interfaces with very basic command
structures. Although these languages existed from the beginning of the industrial age of robotics, very little improvement has been made on languages such as VAL (Vicarm Assembly Language), even though these early systems, in many cases, still include the same richness of functions that today’s packages provide [29][2]. Another example is the Pascal-like language KAREL. It was originally developed as a programming teaching language. Due to the nature of the language, it was used for robotic simulations and with the 1995 release of KAREL, actual robots were successfully controlled in laboratories [11].

The drawback of these textual languages is that they are generally specific to a manufacturer or a specific robot, and are limited in functionality. Requests have been made to vendors to standardise robotic programming to some degree. Researchers and vendors proclaimed that the programming language is part of the research environment, and depends on the goal of the project, and can therefore not be standardised [2].

General Procedural Languages are extensions to generic programming languages like C++ and Pascal [1]. The generic language can vary, depending on the researcher and the goal of the project. The libraries normally include common methods that might be used in robot control, for example tool movement methods. Drivers or object abstractions are then used to implement robot-specific functions and hardware communication. The library would then use these functions to implement higher-level functionality without the programmer needing to work with the robot-specific I/O communication.

ARCL (Advanced Robot Control Library) is one such library [7]. ARCL uses the formal language C and the power of modularity to achieve portability. Another similar package that was developed is RCCL, however the maintainer has ceased development of the official RCCL package.

An alternative approach to procedural languages is behavioural languages. This class of languages specify conditions for the robot’s movement, and how it should react to its environment and changes to it, and not specific actions that should be completed, as is the case with procedural languages. One of the many sub-classes of behavioural languages, is FRP (Functional Reactive Programming). The Domain Specific Language FROB (Functional ROBotics), is an example of a FRP language developed at Yale University. FROB was later integrated into the Haskell package.
Yampa [28][41][14]. These FRP languages are focused on mobile robotics, rather than industrial robot arms.

1.2.3.2 Graphical programming

Graphical programming is sometimes referred to as icon-based programming. In contrast to writing programs, graphical programming allows the developer to use graphs, flowcharts and diagrammatic environments for development, but may include other representations. Textual programming features are commonly found coupled with graphical programming environments to easily and quickly create an action sequence, but still allow the programmer to integrate more complex structures into the program. The use of graphical tools to automatically create underlying code is commonly known as Rapid Application Development (RAD).

Lego uses a flowchart-based (Figure 1.2) control development environment for its Mindstorm robotic kits where the programmer can stack blocks on top of each other representing the sequence of actions the robot must perform[1].

Onika is a diagrammatic graphical programming language designed specifically for sensor-based robot control in the Chimera real-time operating system. It employs predefined blocks which can be connected to achieve the necessary functionality [12]. LEONARDO is an offline programming tool with easy-to-use graphical tools to build and simulate advanced robot control sequences [18]. M. Samaka also developed an offline graphical environment with a library of commands which can be selected interactively to assist the programmer with collision-free path creation [33].

1.2.4 Software Architectures

The last main class of Robot programming, Software Architectures, supplies the necessary underlying infrastructure for robot control (such as I/O communications). These architectures are typically developed by the manufacturer of the robot and are generally specific to the robot. Other third-party solutions have also been developed, eg. UWC’s RobotScript [19]. In section 1.3 a more in-depth discussion on this topic is presented.
1.3 Robot Control Solutions

The development of smaller, more powerful computers and embedded systems made it possible to create more advanced applications. This helps programmers to take greater advantage of the robot. Development has moved away from pure textual programming to include graphical tools. Powerful graphical tools now enable a programmer to benefit from a RAD environment. The development of faster and more robust networks has enabled the use of networked control systems, taking advantage of embedded control systems and computers in a single environment for improved production floor control.

Software solutions typically allow the programmer to use graphical tools to establish the base of the program. If the programmer requires more control, textual programming methods can be used to further extend the functionality of the program.

Due to the development of smaller and more powerful embedded control systems, the “teach pendant” has become a sophisticated tool. Teach pendants now available commercially such as those offered by DENSO Robotics, include touch screens, running complete graphical programming environments, and allowing the developer to develop programs online with great efficiency. To breach the gap between online programming by means of a teach pendant and offline programming, a number of suites simulate real teach pendants in software, simulating robot movements by means of 3D acceleration technologies such as OpenGL.

To further enhance productivity, simulators are included in most packages. This allows the developer to develop his program offline and simulate the final result without the need to disrupt or stop production. Simulators have grown from simple robot-only environments to complete production line systems. Another feature also available is real-time monitoring of active robots on a remote computer in a simulated 3D environment. Robot Studio (ABB) and Silma (MetroLogic) are examples of commercially available suites containing these features.

Another function that has become a necessity is a sensing ability. Visual and physical sensors are now commonly used in industrial production lines. Visual object recognition is a common feature currently found in a number of commercial packages. It may vary from picking out faulty products to distinguishing between different products on a single/shared production line. Physical sensing, such as touch
sensitivity, is also becoming a requirement in many autonomous industrial environments, allowing manipulators to pick up fragile objects, or protect items from being damaged.

The ultimate goal in robotics programming is an autonomous system where the programmer would only specify the task that should be completed. A group of interdependent robots would then optimise the required actions into the most effective sequence of actions, run autonomously and use sensors to produce perfect results [37][27].

1.4 MVAL2006

In 1988 H.O.W. von Petersdorff developed MVAL (Micro VAL), a variant of VAL, for the Centre for Robotics at the University of Stellenbosch (SENROB). Although this software proved to be an excellent tool for its time, the emergence of new technologies required an update to the software.

This thesis is a joint venture between the Industrial and Electronic Engineering departments of the University of Stellenbosch.

The topic of this thesis is the development of a software package for educational and research purposes. The user must be able to control the robot using MVAL, in both an offline and online environment. In the environment the user must be able to simulate the programs with a visual representation of the actual movements of the robot. In online programming the user must optionally be able to view the robot by means of video feedback. The software must also enable the user to perform all of the above actions from a remote computer.

The solution will encompass only the simulation and control of industrial robot manipulators and exclude mobile and humanoid robots.

The proposed software solution must be a network-based system, implemented on a client-server model on a TCP/IP network. A simple communication protocol must be in place to allow easy expansion of the package. MVAL programs must be uploaded to the server and stored in memory until the client requests the server to execute those programs. Furthermore, the server must be able to control a number of robots by means of interchangeable drivers that can be easily added after
deployment. The client must be able to query the server on robot position and control status.

The software should be developed on a model that would benefit future research and development around robotic manipulators.

Chapter 2 examines the mathematics surrounding robotic manipulators, and the models required to analyse and transform manipulators. Chapter 3 and Chapter 4 discuss the development of MVAL2006, with Chapter 3 focusing on the server side, examining and explaining the different abstractions used in the development of the server. The focus of Chapter 4 is the client and aspects surrounding it. Chapter 5 describes the methodology used in testing, and results observed with explanations.
Chapter 2 Kinematics

2.1 Introduction

In this chapter an overview will be given of the mathematical model used to represent robot arms. Denavit-Hartenberg (DH) is a common model used to represent robots, and was chosen as the preferred robot model for this thesis. Attention will be given to the mathematical formulas and how they are derived.

The inverse calculations for the kinematics are also of great importance in any robot control application. Special attention should be given to the two methods of solving the inverse-kinematics problem.

In the following sections a number of mathematical symbols will be used. This table summarises these symbols.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_i$</td>
<td>DH matrix of joint $i$</td>
</tr>
<tr>
<td>$A_i^n$</td>
<td>$A_1 \times A_2 \times \ldots \times A_{n-2} \times A_{n-1}$</td>
</tr>
<tr>
<td>$J_i$</td>
<td>Symbol for joint $i$</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of joints</td>
</tr>
<tr>
<td>$T_n$</td>
<td>Matrix representing final joint coordinate frame.</td>
</tr>
<tr>
<td>$q_i$</td>
<td>DH variable for joint $i$</td>
</tr>
<tr>
<td>$\ddot{q}$</td>
<td>Vector of all DH variables</td>
</tr>
<tr>
<td>$L$</td>
<td>Location of point in the world-coordinate system</td>
</tr>
<tr>
<td>$Z$</td>
<td>Base coordinate frame</td>
</tr>
<tr>
<td>$E$</td>
<td>Tool (end-effector) coordinate frame</td>
</tr>
<tr>
<td>$O_i$</td>
<td>Origin of frame $i$</td>
</tr>
</tbody>
</table>
2.2 Direct-Kinematics

Kinematics is the branch of dynamics that deals with the motion of rigid bodies without kinetics (references to its mass, energy or forces exerted on the body) [17]. The focus of this thesis is on the control of robotic manipulators as an academic tool for students to gain familiarity with robot control and simulating production lines in laboratories. For this reason, the complexity of control is limited to ideal environments, and kinetics is ignored.

The first step in creating the mathematical model is to define base and end-effector (tool) coordinate systems, and a local coordinate system for each manipulator joint. These coordinate systems are assigned in the World plane which is defined using the Cartesian coordinates. Making the assumption that each joint is rigidly linked, equations can be derived to translate from the base coordinate system to the end-effector’s coordinate system.

The DH notation stipulates a stepwise procedure for assigning a coordinate frame to each joint (see Figure 2.1). It does, however, limit the orientation of the local coordinate systems for each joint, and care has to be taken selecting each coordinate frame. It may also be necessary to return to a previous joint’s coordinate frame and change those parameters, so that the correct orientation can be defined for the current coordinate frame.

Figure 2.1 Denavit-Hartenberg parameter explanation.
The DH notation makes use of four parameters: \( a_i \), \( d_i \), \( \alpha_i \) and \( \theta_i \). Using these parameters, a homogeneous transformation \( A_i \) can be created to represent joint \( J_i \). A description for each of the parameters for the DH convention is given in Table 2.2.

### Table 2.2 Denavit-Hartenberg Parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \theta_i )</td>
<td>Anti-clockwise rotation of axis ( x_{i-1} ) around axis ( z_{i-1} ). With revolute joints, this parameter is a variable</td>
</tr>
<tr>
<td>( d_i )</td>
<td>Translation along ( z_{i-1} ). With prismatic joints, this parameter is a variable</td>
</tr>
<tr>
<td>( a_i )</td>
<td>Translation along ( x_i )</td>
</tr>
<tr>
<td>( \alpha_i )</td>
<td>Anti-clockwise rotation of axis ( z_i ) around ( x_i )</td>
</tr>
</tbody>
</table>

The definition of the DH matrix is given below. Note that for brevity, “\( \cos \)” and “\( \sin \)” are represented by “\( c \)” and “\( s \)” respectively, with the denominations denoting the parameter of the associated function.

\[
A_i = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

\[A_i = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

Given \( N \) joints in the manipulator’s configuration, a point can be moved from joint \( J_i \)’s coordinate system to the last joint \( J_N \)’s coordinate system using the following formula:

\[
T_N = \prod_{i=1}^{N} A_i(\theta_i, d_i, a_i, \alpha_i)
\]

Each joint variable is defined as follows:
The combination vector of joint variables for the robot manipulator is defined as:

$$\tilde{q} = \{q_1, q_2, \ldots, q_N\}$$

Using these definitions, $$T_N$$ may also be referenced by $$T_N(\tilde{q})$$. This describes $$T_N$$ for the specific robot manipulator configuration with the variables parameters defined by $$\tilde{q}$$.

Matrix $$T_N$$ can be divided into four sections. It is important to know which section of the matrix contains what information, and where it will be used in subsequent equations.

$$T = \begin{bmatrix} Rotation & Translation \\ Perspective & Scaling \end{bmatrix} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \end{bmatrix}$$

The 3x3 rotation sub-matrix is defined using Euler’s ZYZ notation. The translation matrix is a 3x1 vector containing the position data for this transform. The perspective 1x3 matrix and scaling scalar will remain 0 and 1 respectively in most situations. When used for 3D simulation purposes, they may differ when implementing post-processing to incorporate a field of view.

The next step in the kinematics analysis procedure is to include the base and end-effector’s coordinate systems to find a relation to the robot’s location in the world coordinate system.

$$L = Z \times T_N \times E$$

The base ($$Z$$) and end-effector’s ($$E$$) coordinate systems are commonly defined with Euler-based transforms. These transforms may be constructed with the following formula:
To retrieve the Euler angles and coordinates from the transformation matrix, the following equation is used:

\[
P(x, y, z, \phi, \theta, \psi) = \begin{bmatrix}
c_{\phi}c_{\theta}c_{\psi} - s_{\phi}s_{\psi} & -c_{\phi}c_{\theta}s_{\psi} + s_{\phi}c_{\psi} & c_{\phi}s_{\theta} & x \\
s_{\phi}c_{\theta}c_{\psi} + c_{\phi}s_{\psi} & -s_{\phi}c_{\theta}s_{\psi} + c_{\phi}c_{\psi} & s_{\phi}s_{\theta} & y \\
-s_{\theta}c_{\psi} & s_{\theta}s_{\psi} & c_{\theta} & z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]  

(7)

When rotations are performed, singularities may cause unexpected results. This normally occurs when axes line up. To overcome this problem, the \( \tan^{-1} \) function needs to be fully defined for the whole range of angles.

\[
\tan^{-1}(\frac{a}{b}) = \begin{cases} 
0 & x = 0 \land y = 0 \\
\frac{\pi}{2} - \frac{\pi}{2} \text{sign}(y) & x = 0 \land y \neq 0 \\
\text{sign}(x)\frac{\pi}{2} & x \neq 0 \land y = 0 \\
\tan^{-1}(\frac{x}{y}) & x \neq 0 \land y > 0 \\
\tan^{-1}(\frac{x}{y}) + \text{sign}(x)\pi & x \neq 0 \land y < 0
\end{cases}
\]  

(9)

In summary, the first step in creating a mathematical model for the robot manipulator is the assignment of the DH parameters. The next step is to define base and end-effector coordinate frames. Using the formula \( L = Z \times T_{N} \times E \) in conjunction with the inverse Euler formula stated above, the forward-kinematics solution is fully qualified.


Chapter 2 - Kinematics

2.3 Inverse-Kinematics

2.3.1 Overview

Inverse-kinematics (IK) is the method of calculating the necessary joint angles and displacement required to put the robot manipulator in a configuration so that a certain position and orientation in space can be attained.

The task of IK is to find $\mathbf{q}$ to solve $T_N$ so that $L = Z \times T_N \times E$ is satisfied. $L$ is the required location, and $Z$ and $E$ are fully qualified transformations.

In solving the non-linear equations required for IK, two issues arise: existence of solutions and multiple solutions. This can easily be visualised by the reader by using his/her arm. Although we have a lot of leeway in our joints, certain points are impossible to reach for most of the population, for example the middle of your back. In this case $T_N$ is unsolvable with any valid $\mathbf{q}$. Multiple solutions are best visualised by keeping your hand in front of you and moving your elbow without the need to change the position and orientation of your hand. Here no unique $\mathbf{q}$ exists to satisfy $T_N$, and may result in multiple (or an unlimited number of) solutions.

Multiple solutions can in some cases be seen as an advantage. It can be used in environments to help the robot to work around obstacles. In MVAL, four functions are implemented specifically for this situation: above, below, lefty and righty. These functions instruct the robot to keep its elbow at a relative position (described by the function name) to its end-effector. It can, however, not always be guaranteed, and the controller will have to decide whether it is more important for the robot to acquire the required end-effector position, or whether the rules of orientations should be followed, in which case the specific position may not be attained by the end-effector.

Robot manipulators generally have similar DOFs (Degrees of Freedom) as human arms, but are rigidly attached and are limited to specific axes of freedom. A 6-DOF robot manipulator is commonly used since this allows the robot great freedom to position its end-effector at required positions, assuming that the mechanical limits of the joints are not exceeded. The area the manipulator can reach (taking into consideration its mechanical limits) is referred to as its workspace.
When a robot manipulator possesses less than six DOFs, it’s over-constrained and only a subset of positions is attainable within its workspace. With more than six DOFs, the manipulator is classified as under-constrained and therefore possesses redundant DOFs. It is for this reason that it is common to find 6-DOF robot manipulators, especially generic manipulators not designed for a specific task.

Regardless of the number of solutions, the following must hold true:

\[ T_N = Z^{-1} \times L \times E^{-1} \]  

Since \( Z \) and \( E \) are homogeneous transformation matrices, the IK transformation can easily be found using the following equation with \( A \) as the transformation:

\[
A = \begin{bmatrix}
R & p \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
n_x & o_x & a_x & p_x \\
n_y & o_y & a_y & p_y \\
n_z & o_z & a_z & p_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
A^{-1} = \begin{bmatrix}
R^T & -R^T p \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
n_x & n_x & o_x & -\left( p_x n_x + p_y n_y + p_z n_z \right) \\
o_x & o_x & a_x & -\left( p_x o_x + p_y o_y + p_z o_z \right) \\
a_x & a_x & a_x & -\left( p_x a_x + p_y a_y + p_z a_z \right) \\
0 & 0 & 0 & 1
\end{bmatrix}
\]  

The next two sections discuss the two different approaches to solving these equations.

### 2.3.2 Analytical Solution

Solving IK equations is a complex task. With analytical solutions the designer or developer must, by hand, reduce the matrices to unambiguous equations. The equations can then be given to a computer to calculate.

This type of solution is very limiting for control applications, since it is a robot-specific solution. Some parameters can be left as symbols for the computer to substitute at calculation time. In the process of simplifying the equations to extract unique solutions for the variables, some parameters must, however, be substituted with their values, especially with robot manipulators with greater DOFs. Simplifying equations for 6-DOF manipulators is impossible without substituting some parameters.
Analytical solutions can be divided into two subgroups, algebraic and geometric. The geometric solutions work best in cases where \( \alpha_i \subset \{-\frac{\pi}{2}, 0, \frac{\pi}{2}\} \). Equations are derived by analysing the structure of the robot manipulator. This approach is generally faster and allows less chance of making mistakes. Mistakes can be easily avoided using computer packages such as Mathematica\textsuperscript{®} or Maple\textsuperscript{®}-based systems.

The algebraic process makes use of the definition of \( T_N \) in terms of \( A_i \). Terms are regrouped to identify unique relationships between variables and the required position. As an example, the CS113 robot uses the following combinations to ascertain unique relationships between the required location and orientation in the Cartesian plane and the joint variables (see section C.2.2):

\[
(A_6^1)^{-1}T_N = A_6^6
\]
\[
(A_2^1)^{-1}(A_1^1)^{-1}T_N = A_2^6
\]  

(12)

Using \((A_6^1)^{-1}T_N = A_6^6\), solutions for variables \( q_1 \) and \( q_5 \) are easily obtained. Due to the nature of the robot manipulators’ configuration, the task of solving the other variables is quite tedious and requires numerous mathematical manipulations and familiarity with trigonometry.

The greatest advantages of analytical solutions are stability with regards to time, and their solvability. It is a linear process, allowing the duration of the calculations to be estimated with great accuracy. Solvability is also a major factor in the analysis of the IK problem in robot control, and one where analytical solutions have great advantage over numerical solutions.

### 2.3.3 Numerical Solution

Numerical solutions allow applications to dynamically solve the IK problem using only the mathematical description of the robot. Although numerical analysis of the IK problem involves much more computation than analytical solutions, advances in computing now makes it a viable alternative that can be used for real-time control.

For this study, the focus was placed on Newton-Raphson-based algorithms, although a number of alternative solutions have been proposed for the IK solution. Depending on the specific implementation, the choice of differential (commonly the Jacobian) matrix and the function calculating the direction vector to the target
position for the end-effector, can be altered. Below the pseudo-code for the Newton-Raphson IK algorithm is given:

1. Calculate differential matrix
2. Determine $\Delta \tilde{q}$
3. Adjust variables: $\tilde{q}_{i+1} = \tilde{q}_i + \Delta \tilde{q}_i$
4. Calculate error (distance to target position)
5. If error > tolerance, return to point 1.

The Jacobian matrix is defined as follows:

$$ J(\tilde{q}) = \left[ \frac{\partial \tilde{s}_i}{\partial q_j} \right]_{ij} \quad (13) $$

Vector $\tilde{s}_i$ is the current coordinate frame position of $J_i$ with $q_j$ denoting joint $J_i$'s variable value. To determine the partial derivative, the following equations may be used:

$$ \frac{\partial \tilde{s}_i}{\partial q_j} = \begin{cases} \tilde{v}_j \times (\tilde{s}_i - \tilde{p}_j) & (J_j \text{ revolute}) \\ \tilde{v}_j & (J_j \text{ prismatic}) \end{cases} \quad (14) $$

Vector $\tilde{v}_i$ denotes the direction of the Z-axis and $\tilde{p}_i$ the position of joint $J_i$. Using the Jacobian (eq. 13), determine the change to $\tilde{q}$ for each iteration of the Newton-Raphson method:

$$ \Delta \tilde{q}_i = J(\tilde{q}_i)^{-1} \times \tilde{e}_i \quad (15) $$

Vector $\tilde{e}_i$ defines the direction and magnitude to the target position, along which this algorithm will try to move the end-effector. The typical choice for this vector with $\tilde{r}_i$ denoting the target position is:

$$ \tilde{e}_i = \tilde{r}_i - \tilde{s}_i \quad (16) $$

When examining the definition of the Jacobian, it can readily be seen that in most circumstances the matrix will not be square, and therefore not invertible. Even in the cases where it is invertible, results showed that it is a poor choice. This poses a significant problem.

For this study three cases for alternative differential matrices were investigated:
Chapter 2 - Kinematics

Transposed Jacobian coupled with scalar:
\[ \Delta \tilde{q}_i = \alpha J(\tilde{q}_i)^T \times \tilde{e}_i \]  

Pseudo-inverse (Moore-Penrose inverse):
\[ J(\tilde{q}_i)^+ = \left(J(\tilde{q}_i)^T \times J(\tilde{q}_i)\right)^{-1} \times J(\tilde{q}_i)^T \]
\[ \Delta \tilde{q}_i = J(\tilde{q}_i)^+ \times \tilde{e}_i \]

Damped Least Squares (DSR):
\[ \Delta \tilde{q}_i = J(\tilde{q}_i)^T \left(J(\tilde{q}_i)J(\tilde{q}_i)^T + \lambda^2 I\right)^{-1} \times \tilde{e}_i \]

The different differential functions based on Jacobian matrices (eq. 17, 18 and 19) were first tested with MATLAB® using a 5-DOF manipulator model. Results were compared to the work of S.R. Buss [3]. Requirements for this thesis stipulated a very high level of accuracy which can be guaranteed within a specific timeframe measured in milliseconds. Due to its speed advantage and accuracy, the DSR was selected for implementation and further testing in a formal programming language.

The next function to be examined is the error function \( \tilde{e} \) (eq. 16). Using the above definition of the error function, it commonly causes repeated overshooting with large error vectors.

The Clamping method is an alternative to the common definition of \( \tilde{e} \). Using the definition given below, the distance the algorithm will try to move the end-effector in each iteration to the target position, is limited to \( D_{max} \).

\[
ClampMagnitude(\tilde{w}, d) = \begin{cases} 
\tilde{w} & \text{if } \|\tilde{w}\| \leq d \\
 d \frac{\tilde{x}}{\|\tilde{x}\|} & \text{else}
\end{cases}
\]
\[ \tilde{e}_i = ClampMagnitude(\tilde{x} - \tilde{s}_i, D_{max}) \]

A number of combinations of the equations were tested using MATLAB (for verification) and Delphi programs (visualisation and timing) with a five 5-DOF robot manipulator arm.

Although the algorithms showed great potential, deadlocks were still a frequent problem. Without altering the algorithm as described by pseudo code previously in this section, a proposed solution is to select a new starting manipulator configuration. When a deadlock has been detected, the algorithm will select a random position from
where the IK algorithm will be applied. This may be applied a number of times, but
tests have shown that the algorithm’s success rate did not increase drastically if the
random selection retry count was larger than two.

The major drawback of numerical analysis is solvability. With analytical solutions
it can be guaranteed that a valid solution will be found if it exists, and if the analysis
fails to find a configuration, it does not exist. Numerical analysis relies on the starting
configuration and the numerical path the analysis will follow. It cannot be guaranteed
that numerical analysis will always give an accurate answer either way.

2.4 Conclusion

Robot control applications focus on task completion in industrial environments.
These tasks require the movement of tools in a coordinate plane by means of
coordinates or sensors. For this reason a stable and accurate IK algorithm is
required.

Numerical solutions allow applications to calculate IK with only knowledge of the
mathematical model (eg. DH) of the robot manipulator. This allows the developer and
his/her application much freedom. As repeatedly proven, freedom always comes at a
price. While improvements in technology and research in mathematical solutions for
robotics has made it possible to use numerical solutions for real-time applications,
stability does pose a great problem.

Deadlocks in the numerical analysis can be overcome with random selection of
starting configurations, but using random selection prevents the guaranteed repeata-
bility required in industrial environments.

Even though analytical solutions require much effort by the developer, they
permit IK solutions to be time-limited and accurate.

The application developed for this thesis is intended as an introduction to robot
control for students and laboratory use, and must therefore adhere to the require-
ments of the industrial workplace. For this reason analytical solutions were chosen,
providing the user with an environment that will duplicate industrial environments.
Chapter 3 Server

3.1 Introduction

In this chapter the layout of the client and server applications developed for this thesis are discussed. Although the system is dependant on this server to function, other client abstractions are possible. Taking this into consideration, the focus was on the development of a reliable server with the necessary interfaces to allow the project to be adapted to different environments by means of custom implementations. This greatly increases the power of this project.

The server was developed as a network-accessed system, ensuring stability and reliability so that the user should never require physical interaction with the server. In the case that the system may fail, the session would need to be reinitiated from the client. Problems related to the hardware may, however, cause complete server failure.

This chapter will commence with an overview of the system architecture allowing the reader to get a top down view of the system. It then continues with an in-depth discussion of the different system components making up the software solution.
3.2 Architecture

The system can broadly be divided into three sections:

- Robot Manipulator Interface (RMI)
- MVAL Server
- MVAL Client

![Figure 3.2 MVALServer/MVALClient functional layout](image)

Different manipulators from different manufacturers frequently employ unique interface protocols. The I/O medium connecting the robot manipulator to the computer controller may also vary between parallel, serial, USB and custom internal local bus solutions. Depending on the different manipulators, certain actions may be completed more efficiently. Each robot also possesses unique characteristics depending on its configuration which must also be taken into account. For these reasons the RMI is designed as a separate component with a common interface to the MVALServer.

The MVALServer is a TCP/IP-based server accepting commands from the client allowing a centralised, robot-independent system. Robot applications based on the MVAL language may be uploaded and executed without direct interaction from the client, assuming that the network session is kept active while execution of programs is active. Control of remote video monitoring is also handled by the server, allowing users to monitor a robot from an arbitrary location.
MVALClient is a software application that facilitates easy robot control and application development. The MVALClient can easily be replaced by other applications or components due to the open nature of the network protocol used. This allows integration with applications such as Mathworks\textsuperscript{®} MATLAB\textsuperscript{®} or National Instruments\textsuperscript{®} LabView\textsuperscript{®}.

### 3.3 Hardware Controller

#### 3.3.1 DRIVER

**3.3.1.1 Driver Model**

To date, robot interfaces have not been standardised. Depending on demands from clients, robots are modified or designed to meet specific requirements. To allow MVALServer to take full advantage of different protocols and I/O mediums, a specific driver must be developed for each robot manipulator.

Within the Microsoft\textsuperscript{®} Windows\textsuperscript{®} environment, the Dynamic Link Library (DLL) is a common tool to enable modular, expandable systems. Since a DLL is loaded by a user-mode application, the Operating System (OS) kernel will deny direct access to low level hardware access.

To develop a driver that may have full and direct access to the computer’s hardware, a true Microsoft\textsuperscript{®} Windows\textsuperscript{®} driver would be required. The development of such a driver encompasses much more time, effort and expense than a DLL-based...
driver. Stability is another problem that accompanies these drivers. With the newer Protected Mode OSs (Windows® NT and newer), the kernel is protected from faults that may occur in applications. This protects the computer from a complete system failure commonly accompanied by the Windows® “Blue Screen of Death”. With the new Windows® Vista® to be released by the end of 2006, Microsoft® has made it public that fees will be charged to certify drivers. In the case that the driver is not certified, the system will refuse to load it.

Although low-level hardware access is protected from normal programs by the kernel, a number of alternatives may still be investigated. Considering serial and parallel communications, a free library called InpOutp32 is freely available on the internet. This library grants access to the Port functions required to access these communication ports. Although there is uncertainty regarding this library in future versions of Windows®, there will most certainly be a commercial alternative available in the case that development of this library is ceased. In the case of USB, Bluetooth and local bus implementations, drivers would be provided by the manufacturer, through which the DLL will be able to gain access to the robot. For these reasons, a DLL was selected as the preferred driver implementation.

3.3.1.2 Driver Functions

Initialisation

At the start of a session the Init function is called. This will allow initialisation of the robot and any parameters that might be needed by the driver. The opposite of Init is the DeInit function. This function is called when the session is closed. An exitproc handler should be assigned to the DeInit function which should disable this handler when all necessary finalisation routines have finished. This will protect the driver from unforeseen problems which may occur within the driver, and make sure that everything has been done to ensure that the robot is placed back in the home configuration and switched off (depending on the robot’s capabilities).
<table>
<thead>
<tr>
<th>Driver Function Header</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialisation</strong></td>
<td></td>
</tr>
<tr>
<td>function InitRobot: integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function DeinitRobot: integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Manipulator Description</strong></td>
<td></td>
</tr>
<tr>
<td>function GetDHPParameter(JntNumber: integer; var aDH: TDH; var JntType: byte; var jMin, jMax: real): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetInverse(EulerPosition: TEuler; var JointPosition: RealArray): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetJointCount: integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetJointPositions(var JntPos: RealArray): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetRobotTransform: TTransform: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Setup</strong></td>
<td></td>
</tr>
<tr>
<td>function Above: integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function Below: integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function GetBase(var EulerTransform: TEuler): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetTool(var EulerTransform: TEuler): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function Lefty: integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function Righty: integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function SetBase(var EulerTransform: TEuler): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function SetTool(var EulerTransform: TEuler): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td><strong>General Hardware Related functions</strong></td>
<td></td>
</tr>
<tr>
<td>function GetHardware(var state: byte): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function SetHardware(state: byte): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetSpeed(var CurrentSpeed: real): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function SetSpeed(var NewSpeed: real): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetSignal(SignalNumber: integer; var State: byte): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function GetSignalCount: integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function SetSignal(SignalNumber: integer; State: byte): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function GetSwitch(SwitchNumber: integer; var State: byte): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function GetSwitchCount: integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function SetSwitch(SwitchNumber: integer; State: byte): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td><strong>Manipulator movement functions</strong></td>
<td></td>
</tr>
<tr>
<td>function Calibrate: integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function Ready: integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetGripper(var Gap: real): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetGripperMax(var Gap: real): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function GetGripperMin(var Gap: real): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function SetGripper(var Gap: real): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function DriveJoints(JointPos: RealArray): integer: stdcall;</td>
<td>✓</td>
</tr>
<tr>
<td>function Drive(JointNumber: integer:delta: real; Speed: real): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function Move(Location: RealArray): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function MoveS(Location: RealArray): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function Approv(var Location:RealArray;Distance: real): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function ApproS(var Location:RealArray;Distance: real): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function Depart(Distance: real): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function DepartS(Distance: real): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function MoveW(dx,dy,dz: real): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function Draw(dx,dy,dz: real): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function Rotate(Roll,Pitch,Yaw: real): integer: stdcall;</td>
<td>×</td>
</tr>
<tr>
<td>function Align: integer: stdcall;</td>
<td>×</td>
</tr>
</tbody>
</table>
Manipulator description

The next set of commands provides means to retrieve information about the specific robot manipulator regarding kinematics and inverse-kinematics. As stated previously, the DH notation is used as a basis. A program using this driver should be able to retrieve the DH parameters to enable it to build its own model of the robot. These parameters can be retrieved by using GetJointCount and GetDHParameter. This information combined with the model described in Chapter 2, will enable the host application to build a complete mathematical model of the system.

To assist the host application, GetTransform and GetJointPositions are also included in the library. GetTransform combines the necessary DH parameters into a transform matrix, and GetJointPositions returns the values of the DH parameters for the current robot configuration (as determined by the driver). The driver must allow the host application to request joint positioning data while joint manipulation functions are still in progress. Critical sections should therefore always encapsulate any code that might change the driver’s internal joint positions, or ensure that multiple requests to the I/O layer of the robot manipulator are atomic.

Although these functions supply all the necessary information to build a mathematical model of the robot manipulator by the host system, there is still the matter of IK. As discussed in section 2.4 the solution of the IK is best determined by the algebraic approach. For this reason GetInverse is also included in the API. GetInverse calculates the necessary values of the DH variables to position the last DH frame at a specific position and orientation. This is done by solving $T_N = Z^{-1} \times L \times E^{-1}$.

Setup

The first set of commands is the arm orientation related commands: Above, Below, Lefty and Righty. These functions would generally only change an internal property used by GetInverse when calculating the necessary joint angles.

As discussed in Chapter 2, a base frame and a tool frame are combined with the DH notation to complete the mathematical model. For these frames the following
commands are included: GetBase, SetBase, GetTool and SetTool. These commands are only interfaces to internal driver properties, and do not change the position of the manipulator.

**General hardware-related functions**

To allow the driver to run in a simulation mode, functions are included to change the state of the driver and retrieve the current state: SetHardware and GetHardware. When hardware communication is established, the driver should set all necessary variables and robot properties. When the hardware mode is disabled, all functions must be simulated internally in the driver, and must happen transparently to the calling application. It is the choice of the developer whether the robot should be recalibrated if the hardware mode is enabled. When the hardware is enabled, the driver’s simulated positions should be set to the current position of the robot, normally the home position. The default robot speed should also be set.

For different types of tasks executed by the manipulator, different speeds may be required. The speed is not limited to the actual speed of the manipulator, but is also a factor with the moving of heavier objects. In cases where heavier items are grasped by the tool, lower speeds would help with accuracy and ameliorate overshooting. Further it should help the robot’s drive mechanics cope with the heavier load. For these reasons GetSpeed and SetSpeed have been included. Speed is specified as a percentage of the maximum speed the manipulator is capable of moving. The speed is dependant on the robot’s capabilities, and with certain functions discussed later in this section, it may not always be possible to move the robot at the set speed.

With the multitude of manipulator products available, many manipulators have specific features that may only be useful to the user in specific applications. These functions can be accessed by turning switches on and off. To achieve this, SetSwitch, GetSwitch and GetSwitchCount can be used. These functions may also, at the discretion of the developer, be used to set internal properties of the driver.

Some manipulators are used in conjunction with other equipment, e.g. a conveyor belt, in which case sensors may detect the arrival of objects at a specific position. Some manipulators have capabilities built into their hardware to accept inputs for these purposes. In other cases it may be necessary to use the host
computer’s own I/O peripherals. These signals are driver dependant and are accessed via \texttt{GetSignalCount}, \texttt{GetSignal} and \texttt{SetSignal}.

\textbf{Manipulator movement functions}

The accuracy of a robot manipulator is of great importance. It is common to perform calibration after power-up and/or before tasks are started, and depending on the robot’s design, even in between tasks. This is done by \texttt{Calibrate}.

The \texttt{Ready} command will move the robot back to its home position when called. Where robot manipulators lack position feedback and a home function, the driver should move the robot to its home position using the simulated joint variable values.

In cases where the end-effector’s tool is a gripper, the distance between the fingers is accessible by means of \texttt{GetGripper} and \texttt{SetGripper}. To identify the limits of the specific driver’s robot gripper, \texttt{GetGripperMax} and \texttt{GetGripperMin} are used.

The movement of the manipulator is handled by \texttt{DriveJoints}. The function’s parameters contain the new values for the variables of the DH structure. The function employs a joint-interpolated algorithm to move the manipulator from its current position to the new configuration at the last speed set by \texttt{SetSpeed}.

\texttt{DriveJoints} allows movement of the manipulator based on specific joint variables. Most robot tasks require movement defined by the Cartesian plane rather than joint variables. Some of these actions may optionally be included in the driver. This may be advantageous in cases where these movements can be optimised for a specific robot. This is discussed in more detail in section 3.3.2.

The \texttt{Drive} command is similar to \texttt{DriveJoints}. This command moves the joint by the specified speed. The joint is moved by the value specified, relative to the current position.

The \texttt{Move} and \texttt{MoveS} functions allow movement to a specific position in the Cartesian plane by either a joint-interpolated move, or by trying to move the tool in a straight line (\texttt{MoveS}) from the starting position to the target position.

The \texttt{Appro} and \texttt{ApproS} commands are very similar to the move commands. These functions will approach the object, staying a specified distance along the
Z-axis of the end-effector’s coordinates from the target position. This can be quite useful in positioning the gripper, e.g. in cases where the tool has to move into an opening to grip an object. Once again there is a choice between joint-interpolated movement and straight-line movement of the tool.

In contrast to the approach commands are the depart commands, \texttt{Depart} and \texttt{DepartS}. These commands will move the tool a specified distance along the positive Z-axis of the tool’s coordinate frame.

To specify relative movement of the tool, three functions are used: \texttt{MoveW}, \texttt{Draw} and \texttt{Rotate}. \texttt{MoveW} moves the tool in the world coordinate frame by the relative distance specified. Similarly the \texttt{Draw} command moves the tool in the tool’s coordinate frame by the distance specified. The last function, \texttt{Rotate}, allows the rotation of the tool around the tool’s coordinate frame.

To setup the tool for a task, \texttt{Align} is provided. This function will align the tool’s Z-axis to the closest axis of the world-coordinate frame.

\subsection*{3.3.2 Driver Wrapper}

The driver wrapper component (\texttt{MVALDriverWrapper}) is the intermediary between the application and the driver. It provides the necessary loading routines and error checking. There are a number of reasons for choosing to implement a layer between the drivers and the application.

During the development and testing of the robot drivers, the wrapper provides a common interface that can be easily tested with external applications. Using an external component also allows the driver to be easily integrated in other projects. The wrapper model also allows easy transparent overloading of functions.

The wrapper’s interfaces can be divided into two groups:

- Configuration routines
- Driver routines

The configuration routines provide interfaces to load a driver, to check whether a driver is loaded, to retrieve the name of a loaded driver, and to unload a loaded driver. When loading a driver, the wrapper will check the DLL whether all required functions have been implemented. If this is not the case, the driver will fail to load. If
all the prerequisites are met, a list of functions available in the driver is stored, and
the driver’s `Init` method is called to initialise the robot. When the driver is unloaded,
the wrapper will call the driver’s internal `DeInit` function, after which the driver is
freed from memory.

The wrapper divides the driver methods into three groups:

- Required
- Optional
- Overload

The `Required` methods form the set that defines the minimum requirements for a
valid robot driver. These routines are listed in Table 3.1 under `Required` functions.
The wrapper exports these functions directly.

The `Optional` methods provide robot-specific actions that may or may not be
found in different systems. These methods are the signal, switch and arm orientation
methods. They are not critical functions, but allow extra functionality to be utilised. In
the case where these functions are not available in the driver, the wrapper will return
an error result `DRV_FUNC_NOT_SUPPORTED`.

Figure 3.4 MVALDriverWrapper internal functional flow for optional movement commands.
The Overload group of methods is limited to the movement operations depicted in Table 3.1 under group “Manipulator movement functions”. These functions implement mathematical equations to perform the robot’s actions, and predominantly work with coordinates and orientation, which are not directly related to the physical configuration (excluding limitations on the DOF). These functions can be implemented by the driver to overload the internal functions.

The internal functions perform the necessary calculations to satisfy the mathematical model. The action is then executed directly or indirectly by DriveJoints. Move commands directly call MoveRobot, which is capable of calculating straight-line as well as joint-interpolated movement. Approach and depart commands rely on the GeneralApproach and GeneralDepart routines to calculate the target position, which can be used by MoveRobot to complete the task. Table 3.2 lists the equations used by each movement function in the wrapper when not overloaded.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>$L = Z \times T_N(q) \times E$</td>
</tr>
<tr>
<td>Approach</td>
<td>$L \times \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; d \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} = Z \times T_N(q) \times E$</td>
</tr>
<tr>
<td>Depart</td>
<td>$T_N \times \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; d \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} = Z \times T_N(q) \times E$</td>
</tr>
<tr>
<td>MoveW</td>
<td>$Z^{-1} \times \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; dx \ 0 &amp; 1 &amp; 0 &amp; dy \ 0 &amp; 0 &amp; 1 &amp; dz \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} \times Z \times T_N = Z \times T_N(q) \times E$</td>
</tr>
<tr>
<td>Draw</td>
<td>$T_N \times \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; dx \ 0 &amp; 1 &amp; 0 &amp; dy \ 0 &amp; 0 &amp; 1 &amp; dz \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} = Z \times T_N(q) \times E$</td>
</tr>
<tr>
<td>Rotate</td>
<td>$Z^{-1} \times T_N \times R \times E^{-1} = Z \times T_N(q) \times E$ with $R =$ Euler rotation matrix</td>
</tr>
</tbody>
</table>
The straight-line movement is computed by drawing a straight line between the start and target positions, and then dividing the line into segments of a specified length. Joint interpolation is then used to move the end-effector along the segments. Although this method is very effective, there are some limitations. In cases where the coordinates produce joint variable values close to a joint’s constraints, the IK solution between two points may produce a large change in joint angles moving the robot into a different configuration.

Another common problem with the straight-line algorithm is slow, jagged movement. Most manipulators’ onboard controllers have integrated algorithms for joint-interpolated movement providing smooth operation. For straight line movement, commands have to be sent for each segment to estimate a smooth line of movement. The driver needs to wait for the robot controller to finish active operations before the next command is sent through. When the command is sent, some time is needed for the controller to read and process the command before execution starts. Many controllers integrate acceleration and deceleration into the movement profile. Thus for each segment the joints will accelerate to optimal speed, and then decelerate to the final position. Multiple movement commands sent to the robot’s internal controller may thus present jagged movement. This jagged effect may be improved with robot hardware controllers that allow buffered commands or include motion planning systems.

### 3.3.3 3D ASE ROBOT MODEL

To provide client applications with a 3D representation of the actual robot for a specific driver, ASE (ASCII Scene Export) models can be used. The model must be stored in the same directory as the robot driver. The name must also correspond with the driver’s name, and have an “.ASE” extension. Design programs such as 3D Studio Max can export ASE files.

The ASE file format has limited features but provides enough functionality for the purpose of simulations. Features include the grouping of meshes, allowing a joint to be grouped as a single entity. Materials can be applied to groups and subgroups, providing parameters for colour, ambient light, surface diffusion, specular properties and transparency.
The ASE model uses a group identifier “JointX”, with X being the number of the joint ranging from 1 to N to identify which group within the file corresponds with which DH frame. Orientation of the group in the ASE model is aligned with the DH frame’s coordinate axes. Pivot point specification is a space-delimited list appended to the group’s name. Below is an example of a group representing joint 2 with its pivot point positioned along the x axis at 102mm, along the y axis at 95mm and along the z axis at -10.5mm.

Joint2 x=102 y=95 z=-10.5

Interpretation of the contents of the model file is done by the client application, and has no effect on the working of the driver.

3.4 MVALServer

![MVALServer's controller block layout](image)

Figure 3.5 MVALServer's controller block layout

3.4.1 MESSAGE CONTROL

3.4.1.1 Network layer

Network communication is implemented using the Indy[35] set of components to interface with the Windows sockets API. A number of listening sockets are created
on specific ports to allow clients to connect to the server over a TCP/IP-based network. Multi-homed servers are supported, allowing the server machine to be connected to a number of networks, each allowing connections to the server without needing complex network routing configurations. This may be advantageous to a machine connected via a standard Ethernet connection, wireless networking or when servicing virtual networks.

Secure Socket Layer (SSL) encryption is included to allow communication to be protected from data packet interception attacks [15][24].

When a client connects to the host, the Robot Controller (section 3.4.2) and UDP (User Datagram Protocol) server for positioning (section 3.4.3) are created together with the network session thread. All incoming messages are handled by the message control unit which is part of the network session thread.

### 3.4.1.2 Message Handling

A number of communication models have been developed with *inter alia* XRCL[2] and RoboML[21]. These models were developed to be general XML-based communication models applicable to a wide variety of applications. RoboML is a very attractive model for the communication protocol, and developing a system following this model would provide easy integration with other existing software solutions. The RoboML XML specification is designed for robot control, and lacks structures for system control [31].

Another attractive technology is SOAP (Simple Object Access Protocol)[25]. SOAP is generally based on an HTTP network, and exchanges messages in XML format, which provides great portability. SOAP is based on a connectionless communication channel, which is not ideally suited to this application. The application relies on a guaranteed connection which the server can monitor to make sure that robot manipulators are correctly powered-off after use.

A custom protocol using XML encoding was deemed to be the best choice for this thesis. XML as encoding format was chosen for the following reasons:

1. A large number of tools exist for most programming languages to manipulate XML structures
2. With its powerful hierarchical design, a well-structured communication protocol interface can be designed.

The Message Control unit is responsible for communications between the client and the server. Message sequences (requests followed by responses) are handled synchronously, with update messages sent by the server to the client asynchronously. Although buffering of incoming messages allows multiple requests to the server, commands will be serviced in a FIFO (first-in-first-out) order. The server acts on request messages, and will send a response message on completion of the requested action. Only after a request has been answered with a response will the next message in the buffer be serviced.

The Message Control Unit is built using multiple level try-except blocks to prevent any failures to go by unhandled. Messages are sent to the client to notify it of any problems that have occurred.

While the server is servicing a request, messages can be sent asynchronously to the client. This obviates the need for the client to poll the server continuously. Two types of asynchronous messages can be sent to the client: \texttt{<message>} and \texttt{<update>}. The former is used to send informational text messages to the client, the latter is for updates on program execution status.

The server accepts the following request messages (see Appendix B for specification):

- login
- logout
- execute
- program
- location
- driver
- robot
- videostream

When a session is initiated, the client must first authenticate itself for the session before any actions can be performed. The \texttt{<login>} request is accompanied by a username and a password, and can optionally be accompanied by a driver to be loaded. The information is verified against the registered users, and the IP address of
the client (section 3.4.5). If the driver was omitted during authentication, the `<driver>` request must be used to specify a driver. Most requests require a driver before they can be completed. The `<logout>` request unloads all session-specific components, and closes any connections that may still be open (positioning data unit and video streaming unit).

Single MVAL commands are executed using `<execute>` requests. Commands are session-related. Transformations assigned in previous `<execute>` commands, or uploaded by a `<location>` request, are accessible for the duration of the session. To execute multiple commands, programs can be uploaded with `<program>` requests. Transformations are uploaded using `<location>` requests.

In the definition of `<execute>`, a language parameter is included. This allows the server to handle different scripting languages, and allows for future enhancements.

<table>
<thead>
<tr>
<th>Action</th>
<th><code>&lt;program&gt;</code> Request Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>Get a list of programs in the server memory</td>
</tr>
<tr>
<td>get</td>
<td>Retrieve program from server</td>
</tr>
<tr>
<td>set</td>
<td>Change existing program</td>
</tr>
<tr>
<td>upload</td>
<td>Upload a group of programs and locations</td>
</tr>
<tr>
<td>remove</td>
<td>Remove program from server's memory</td>
</tr>
<tr>
<td>clear</td>
<td>Remove all programs, locations and variables from the server’s memory</td>
</tr>
<tr>
<td>execute</td>
<td>Execute a program</td>
</tr>
<tr>
<td>status</td>
<td>Check if a program is currently running, paused or finished</td>
</tr>
<tr>
<td>pause</td>
<td>Pause execution of program by the server</td>
</tr>
<tr>
<td>step</td>
<td>Execute next line of code of a paused program</td>
</tr>
<tr>
<td>resume</td>
<td>Resume execution of paused program</td>
</tr>
<tr>
<td>stop</td>
<td>Stops execution of any active program</td>
</tr>
</tbody>
</table>

An MVAL program is a set of MVAL commands allowing users to specify a series of actions to be performed. Programs can further increase productivity and efficiency by combining a subset of tasks which can be embedded in other programs. This allows users to quickly and efficiently take advantage of previously written programs to perform common tasks. All program-related requests are accessed by a `<program>` request, the lifetime of which is limited to the session. A short description of the available actions that can be requested by a `<program>` request is listed in Table 3.3. These commands allow clients to upload, retrieve, modify and
remove programs from the server’s memory. It further provides access to execution commands which are used to control program execution.

Table 3.4 Location request action summary

<table>
<thead>
<tr>
<th>Action</th>
<th>&lt;location&gt; Request Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A list of locations in the server memory</td>
</tr>
<tr>
<td>get</td>
<td>Retrieve location from server</td>
</tr>
<tr>
<td>set</td>
<td>Upload/change location variable on server</td>
</tr>
<tr>
<td>remove</td>
<td>Remove location from server’s memory</td>
</tr>
</tbody>
</table>

An integral feature of the server is its ability to save and recall positions in the Cartesian space. These positions are stored as transformations. Transformations are also used to describe relative position and orientation. These transformations are accessed by means of a \(<location>\) request.

Actions included in the \(<location>\) request are similar to the program manipulation routines. Methods are available to retrieve a list of available locations, retrieve or set a location variable or remove the location from the server’s memory. All location variables use the Euler definition of \(\{x, y, z, \phi, \theta, \psi\}\) (section 2.2). A list of available actions is listed in Table 3.4.

Table 3.5 Driver request action summary

<table>
<thead>
<tr>
<th>Action</th>
<th>&lt;driver&gt; Request Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>Get a list of available drivers</td>
</tr>
<tr>
<td>get</td>
<td>Get the name of the currently loaded driver</td>
</tr>
<tr>
<td>load</td>
<td>Load driver</td>
</tr>
<tr>
<td>unload</td>
<td>Unload active driver</td>
</tr>
</tbody>
</table>

Robot selection is handled by means of \(<driver>\) requests. Driver files are stored on server accessible storage space, and must use the “.drv” extension. The path to the driver files is configurable by the server.

Table 3.6 Robot request action summary

<table>
<thead>
<tr>
<th>Action</th>
<th>&lt;robot&gt; Request Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getconfiguration</td>
<td>Get DH parameters and robot constraints from the loaded driver.</td>
</tr>
<tr>
<td>getposition</td>
<td>Get DH variables of the current robot position</td>
</tr>
<tr>
<td>getmodel</td>
<td>Get 3D ASE model of the robot</td>
</tr>
<tr>
<td>getmodelinfo</td>
<td>Get information about the file storing the 3D model on the server</td>
</tr>
</tbody>
</table>
Most clients would require information about the robot’s physical structure. The `<robot>` request can retrieve the DH parameters of the robot manipulator controlled by the loaded driver, and the DH variables of the robot’s current configuration.

Along with the driver, a 3D model may optionally be included to represent the structure of the robot (section 3.3.3). The ASE file’s contents can be retrieved using the `getmodel` action.

To retrieve file information, `getmodelinfo` is provided. ASE models with a reasonable amount of detail normally generate large (greater than 1MB) files. The ASE data structure employs a simple text encoding, allowing for very efficient compression. For this reason ZIP compression can be requested [8]. The file is then compressed before being sent to the client. To further optimise bandwidth usage, `getmodelinfo` provides file information such as a MD5 checksum and the file size[30]. This is used by MVALclient to check whether the file has been updated on the server and requires downloading.

The last request is `<videostream>`. This request interfaces with the video stream unit (section 3.4.4).

### 3.4.2 ROBOT CONTROLLER

#### 3.4.2.1 MVAL Program

The robot controller is designed to employ a scripting language to execute scripts. There are two aspects of management required: managing both textual language and interpreted code used to execute the program. MVALProgram is the management unit that provides these functionalities.

Multiple programs can be uploaded to the server for execution in a single session. This allows a program to call other programs. Integrated into MVALProgram is MVALProgramCode, which handles the individual programs and their properties. To provide greater freedom in programs not supported by the language, variables are not limited to a scope, and are therefore implemented within the scope of MVALProgram.
On the textual side, `MVALProgramCode` provides simple structures of lines of text to store the programs to be interpreted in a standard Delphi TStringList object. A program identifier is also associated with `MVALProgramCode` to enable unique identification of programs. `MVALProgram` provides the necessary routines to add, modify and delete programs.

After the textual representation has been parsed by the interpreter, the code structure is stored in `MVALProgramCode` in a dynamic array of `TProgCode` (see Figure 3.6).

```pascal
TProgCode = record
  InstructLabel : Integer;
  Cmd : Integer;
  Param : Array of TParam;
end;
```

The intermediate structure contains three fields which contain the command identifier, the parameter list and an optional label. Command identifiers are integers which must correspond to commands defined in unit `MVALTypes`. Parameters are stored in a dynamic array, which can store any combination of parameters. A list of valid parameter types is given in Table 3.7.
The dynamic structure `TParam` allows the use of multiple types of parameters, however memory usage increases considerably. Each record representing a parameter uses the amount of memory that the largest variable in the structure requires.

**Table 3.7 Parameter descriptions**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Data type</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>Variable name</td>
<td>Array of 255 characters</td>
<td>255 bytes</td>
</tr>
<tr>
<td>Operation</td>
<td>Arithmetic or logical operators</td>
<td>Set of operators</td>
<td>1 byte</td>
</tr>
<tr>
<td>String</td>
<td>Array of characters</td>
<td>Array of 255 characters</td>
<td>255 bytes</td>
</tr>
<tr>
<td>Integer</td>
<td>Number</td>
<td>Signed 32 bit number</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Float</td>
<td>Floating point value</td>
<td>64bit Floating point value</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Program</td>
<td>Program name</td>
<td>Array of 24 characters</td>
<td>24 bytes</td>
</tr>
<tr>
<td>Label</td>
<td>Label number</td>
<td>Signed 32 bit number</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

The type `String255` is capable of holding 255 characters, and occupies 255 bytes of memory. This can be compared to the floating point values which only occupy 8 bytes. The memory overhead is justified by the simplification of code as well as the potential for extending functionality. Large programs would rarely be longer than 1000 lines of code, which translates to 300KB of data (1000 lines times an average of 1.5 parameters/command times 255 bytes per parameter). This could be reduced to 30KB using a more efficient memory model, but may also decrease
computational speed. Both these problems are, however, insignificant. Size is negligible on any modern computer system with a minimum of 256MB RAM.

Variables used by a program are also stored in MVALProgram, and referenced by an unique identifier. A dynamic array of TVALVariable stores transformations or variables of type integer. Transformations are not limited to execution sessions as with integer variables, and may be uploaded or downloaded from MVALProgram with methods.

### 3.4.2.2 MVAL Interpreter (MVALCompiler)

MVAL as derived and implemented in 1988 at the University of Stellenbosch, is based on one of the original scripting languages used in robotics, VAL[29], which is used by, amongst others, Unimation. The language is designed around a Cartesian coordinate system using homogeneous transformations for positioning. Most commands in MVAL are derived directly from VAL. One of the largest differences between the two languages, is functionality for using locations based on a joint variable’s value being omitted in MVAL.

MVAL commands are interpreted by the MVAL interpreter, which translates MVAL commands and verifies the syntax. The commands are then stored as a data structure held in an MVALProgram object and performed by the execution unit. Standard notation of commands in MVAL adhere to the following format:

```
<COMMAND> <parameter1>,<parameter2>,...,<parameterN>
```

Commands may be abbreviated using the least amount of letters that identifies a unique command.
Interpretation of the commands is performed using a single-pass token-based system. Tokens represent a number of characters and/or symbols that have significant value. This includes keywords and separation characters such as commas and arithmetic operators. The following MVAL command example changes the base’s location to position \((x,y,z)=(10,20,-5)\) with a Z-rotation of 45 degrees.

\[
\text{base 10,20,-5,45}
\]

The tokens generated from this command are listed in conjunction with the actions performed by MVALProgram in Table 3.8.

While the system tokenises the MVAL text, syntax checking, type checking and existence of variables are checked against the definition of the specific command where possible. This prevents executing programs from being terminated before completion due to human programming errors.

<table>
<thead>
<tr>
<th>Token</th>
<th>Token type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>Command</td>
<td>Add TProgCode entry for base</td>
</tr>
<tr>
<td>10</td>
<td>Floating point value</td>
<td>Add Float parameter</td>
</tr>
<tr>
<td>,</td>
<td>Character</td>
<td>None</td>
</tr>
<tr>
<td>20</td>
<td>Floating point value</td>
<td>Add Float parameter</td>
</tr>
<tr>
<td>,</td>
<td>Character</td>
<td>None</td>
</tr>
<tr>
<td>-5</td>
<td>Floating point value</td>
<td>Add Float parameter</td>
</tr>
<tr>
<td>,</td>
<td>Character</td>
<td>None</td>
</tr>
<tr>
<td>45</td>
<td>Floating point value</td>
<td>Add Float parameter</td>
</tr>
</tbody>
</table>

Interpretation of commands is done per line. For each line the compiler starts by identifying the first token. This may be either a Command token of type string which identifies a unique MVAL command, or an Integer token which identifies a label for the line. A label is only valid in the case where the line of code being interpreted is part of a program, and not just a single console command. If the first token was an Integer token, it is stored in the TProgCode entry for this command, and the next token, which must be a Command token, is extracted and saved in the TProgCode entry.

At this stage the interpreter retrieves the first parameter token (which may be empty), and branches into subroutines which handle the parameters associated with each command, and adds it to the current TProgCode entry. This process is repeated for each line of code to be interpreted.
To allow programs to call other programs (GOSUB), flags are added to each program. These are cleared at the start of interpretation. Along with the clearing of the compiler flag, all variables except location variables are cleared. After the program to be executed has been interpreted, the interpreter sets the flag on the current program. For each GOSUB call in the current program, the flag for the called program is checked. If the program has not been compiled, the process is repeated for the sub program.

### 3.4.2.3 Program Execution

The core of the Server application is the Program Execution unit. This unit is responsible for executing commands stored in MVALProgram. These commands can be either single commands or complete programs.

An execution thread runs continuously in the background, checking if any commands are ready to be executed. After each check, the thread is forcefully rescheduled by using the `sleep` command. This allows the execution of other threads in the program, generally increasing the responsiveness of the application. It also decreases the chance that execution of a command will be interrupted by rescheduling by the OS.

At every iteration of the thread, the program checks whether there is a console command ready for execution, and if so, executes it.

Console commands are implemented as a blocking procedure called with a `<execute>` request to the Message Control Unit. The command is first interpreted to determine validity and type, and then executed directly.

Some MVAL commands, such as file operation commands for loading and storing files, are not supported by the server. These commands fail to execute and must be handled by the client (see Appendix A for commands limited to a calling scope).
Figure 3.7 Execution thread flowchart
The first step after compiling the program is to verify that the command is supported by the server, and if not the case, a message is sent to the client. A list of commands not supported by the server is given in Table 3.9. A flag is then set informing the execution thread that a command is ready. The execution of the console command is then blocked until such time that the thread has finished executing the command and has cleared the flag. The result of the execution of the command is then forwarded to the Message Control Unit.

**Table 3.9 Commands not supported by MVALServer**

<table>
<thead>
<tr>
<th>Commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDIT</td>
<td>Edit a program</td>
</tr>
<tr>
<td>LISTF</td>
<td>List programs stored on storage medium</td>
</tr>
<tr>
<td>STORE</td>
<td>Store program and location variables on storage medium</td>
</tr>
<tr>
<td>STOREP</td>
<td>Store program on storage medium</td>
</tr>
<tr>
<td>STOREL</td>
<td>Store location variables on storage medium</td>
</tr>
<tr>
<td>LOAD</td>
<td>Load program and locations variables from storage medium</td>
</tr>
<tr>
<td>LOADP</td>
<td>Load program from storage medium</td>
</tr>
<tr>
<td>LOADL</td>
<td>Load location variables from storage medium</td>
</tr>
<tr>
<td>DELETEFILE</td>
<td>Delete program or location file on storage medium</td>
</tr>
<tr>
<td>TEACH</td>
<td>Teach new positions</td>
</tr>
<tr>
<td>POINT</td>
<td>Modify a location variable. This command is handled by the server only if more than 1 parameter is specified</td>
</tr>
</tbody>
</table>

Execution of a program entails multiple calls to the MVAL console execute command. When the execute method is called, the program specified as a parameter is compiled and stored for future use. The program counter is reset and the program status set to Running.
### Table 3.10 Execution Status Description

<table>
<thead>
<tr>
<th>Execution Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>No program is currently in memory to be executed</td>
</tr>
<tr>
<td>Running</td>
<td>The program is running</td>
</tr>
<tr>
<td>Paused</td>
<td>The program is paused</td>
</tr>
<tr>
<td>Step</td>
<td>The program is performing one step after which it will be paused again</td>
</tr>
<tr>
<td>Error</td>
<td>An error occurred while executing the program</td>
</tr>
</tbody>
</table>

At each iteration of the execution thread, the program status flag is checked to see whether the program is in either Running or Step mode. If this is the case, the current command is executed. If any errors occurred during execution of the command, the program stack is cleared, the program status flag is set to error, and the client is informed with an `<update>` message. If the execution succeeded, the program status flag is set to Paused if it was Step. If the code executed was the last of the current program, the call stack is popped. If the call stack was empty, the program status is set to Idle and the client is informed.

### Table 3.11 Functions affecting execution of programs.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTE</td>
<td>Compile program and set program status flag</td>
</tr>
<tr>
<td>GOSUB</td>
<td>Call program</td>
</tr>
<tr>
<td>GOTO</td>
<td>Jump to a label in the current program</td>
</tr>
<tr>
<td>IF</td>
<td>Jump to a label in the current program if arithmetic evaluation is true</td>
</tr>
<tr>
<td>IFSIG</td>
<td>Jump to a label in the current program if signal evaluation is true</td>
</tr>
<tr>
<td>PLAY</td>
<td>Program continues execution</td>
</tr>
<tr>
<td>PAUSE</td>
<td>Program execution is paused</td>
</tr>
<tr>
<td>STEP</td>
<td>Program executes next program line</td>
</tr>
<tr>
<td>HALT</td>
<td>Program execution is halted</td>
</tr>
</tbody>
</table>

Most commands perform operations that do not affect the flow of the program and interface directly with the MVALDriverWrapper unit. The exceptions are the program status commands listed in Table 3.11. These commands manipulate the program status, call stack and counter to control program flow.
### 3.4.3 Position Data

For simulation purposes, periodic updates must be sent to the client about the position of the robotic arms. The update messages must be independent of processing on the server side.

Using the available TCP connection, asynchronous messages could be sent using `<update>` and `<message>` messages. TCP is, however, a network protocol that guarantees packet delivery and requires a lot of overhead, and includes integrated shaping to adjust to network characteristics. The UDP protocol is designed to be a passive, small and efficient protocol, but is not a guaranteed network service, tolerating the loss of packets. UDP is commonly used with video and audio streaming. Based on these reasons, a separate thread is created within the server which communicates with the client using UDP packets. This greatly increases the rate at which update messages can be sent to the client without affecting the TCP communications.

Integrated in the UDP protocol is support for anycast, broadcast and multicast. This provides an ideal infrastructure to allow for remote monitoring. This advanced functionality is however not included.

The UDP connection is created by the server when a session is initiated. The UDP port number used in the client is defined as the TCP port number used on the server, plus two. Thus, if the server is hosted on TCP port 21320, then the UDP packets are sent to the client on port 21322.

Data is sent as pure textual data strings. An example of an update packet for a 4-DOF robot:

```
data 12.024,14.000,16.000,-23.000
```

The message can be quickly decoded by the client with minimal processing. An alternative method, using a byte encoding to send the data, will require that the client be aware of the bit-order notation (such as little-endian) used by the server. This conflicts with the original objective of providing a software application that can be easily extended by any developer.

Two components that were investigated to provide communications are Delphi’s standard `UDPSocket`, and the Indy component’s `idUDPServer`. Delphi’s
UDP Socket has a known problem with two-way communication when using a single component, and does not allow easy binding to a specific network interface. The idUDPServer component does not present the same shortcoming as the UDPSocket. Furthermore, capability to bind to multiple network interfaces is integrated into the component.

3.4.4 Video Streaming

Video streaming is a simple term belying a complex structure. A number of commercial solutions are available to make the process of video grabbing and streaming less complex.

The first problem that arises with video streaming is the acquisition of data from the video source. With the development of Microsoft® Windows® and its multimedia interface library DirectX, programmatically manipulating video has been simplified greatly. It does, however, require a video source that is supported by DirectX (most web cameras are).

Normal web cameras support resolutions of 640 pixels horizontally and 480 pixels vertically using 24-bit colour. This translates to 921,600 bytes of data per frame, requiring a bandwidth of 13.2MB/s for a 15 frames per second (fps) data feed. Using compression this can be greatly reduced. Video streams compressed using encoding libraries such as DivX or XviD at 175KB/s can provide the same resolution, but at 25fps. These libraries are however not suited to real-time network streaming [9][40]. MPEG (Moving Picture Experts Group) encodings on the other hand, were designed with this in mind [22].

The VideoLAN project is a cross-platform GPL (GNU General Public License) project [36]. It provides full functionality for video capture, encoding, streaming and playback. The official software application of the VideoLAN project, VLC (VideoLAN Client), is a complete suite including playback from multiple sources (including video capture devices), encoding of data streams and recording. A VLC ActiveX component was used to provide the necessary functionality for video streaming.

Using the standard streaming settings does, however, create a large delay of up to 4 seconds in the video streaming. This can be attributed to the buffering of video to provide fluid playback. The system developed in this thesis is designed for a LAN
environment with very low latency (<5ms) and a constant data throughput of at least 500KB/s. Due to the nature of the environment, buffering can be greatly reduced.

The first buffer is the DirectX buffer capturing the video. This can usually be modified in the Windows Control Panel for the specific capture device. The next set of buffers that come into play is the capturing and streaming buffers of the VLC server. These buffers can be disabled (set to 0ms) without ill effect. At the client a video buffer of at least 20ms is recommended. With caching buffer values of 15ms and smaller, video playback easily emptied the playing buffer, resulting in sporadic video performance. Video playback was successfully tested with a real-time delay of 700-800ms.

### 3.4.5 Access Control

Providing a service controlling expensive and potentially dangerous equipment must be done with great caution. Access control is therefore a very important facet that must be included.

Users are registered on the MVAL server. A client must provide the correct authentication details before access to the feature set of the robot is unlocked. Authentication detail is a tuple consisting of a username, password, client IP and server IP.

Access details are stored in a XML file in the server installation directory with the following file structure:

```xml
<users>
  <user>
    <username>Username</username>
    <password>Password</password>
    <validclient>
      <ip>1.2.3.4</ip>
      <mask>255.255.255.0</mask>
    </validclient>
  </user>
  ...
</users>
```
Password fields in the XML file are encoded using 128-bit RSA-MD5 encryption [30]. MD5 is a destructive (one way) algorithm. Passwords cannot be retrieved from the file except through brute-force algorithms.

Each user can be associated with a specific computer or computer network. A specific computer can be specified using the IP node. To provide a more general level of access, a computer network can be opened using an IP subnet mask.

With local configurations (MVALServer and MVALClient running in the same session), password protection is unnecessary. The MVALServer must be run with administrative privileges to allow hardware access, and users with administrative rights can easily delete or modify any file on a computer. Administrative users could thus easily circumvent the protection put in place. Taking everything into consideration it was deemed sufficient to use the Windows® Session password protection. MVALServer may therefore be run in a background session on the computer with administrative permissions, allowing the client to run as a normal, unprivileged user.
Chapter 4 Client

4.1 Introduction

The MVALClient is a software application for robotic control using MVAL as scripting language. The client provides an easy-to-use, but nevertheless extensive, environment to implement, test and execute robotic control applications. It is suited to present students with an introduction to robotic control, and may well prove to be an indispensable tool for research, simulation and testing.

The client provides a clean, organised graphical interface to the server, allowing users to quickly develop scripts to be performed by the manipulator. The client is designed as a portal to take advantage of the features provided by the server.

By design, the client is divided into a communications layer and a graphical front-end. The communications layer is implemented in a thread (MVALClientThread) that is independent of the graphical interface.

The MVAL2006 User Manual can be referenced for an operational guide to MVALClient. The manual is not presented as part of this thesis document, and forms a separate document.
4.2 Communications Layer

MVALClientThread provides static methods to allow the front-end to initiate and close network sessions, send and receive messages, and gain access to the network connection’s properties to extract information such as IP bindings.

MVALClientThread checks the message buffer for messages posted by the static message methods. The message request buffer can contain only one message at a time. Eliminating the message buffer would cause message requests to block until acknowledged by the server. As a result the graphical user interface would be sluggish. Increasing the buffer to allow multiple messages to be queued would not affect the usability of the front-end, since the server uses a synchronous communication channel. Furthermore message requests would then require identifiers to match a response to a request, adding unnecessary complexity and increasing the possibility of program failure. Using a single thread with buffered requests allows asynchronous <message> and <update> messages to be filtered and handled independently.

4.3 Graphical User Interface

4.3.1 CONSOLE

The centre of the MVALClient user interface is the command console. This is the main control window allowing textual commands to be sent to the server using MVAL as language. To provide a similar feeling to various other command consoles, MPL-licensed TConsole, written by Michael Elsdörfer, was used [10].

TConsole is an event-driven component derived from TCustomControl, overriding the events as required to achieve the required functionality. Keyboard input support by TConsole is insufficient for this project. The KeyPress and KeyDown events use TConsole’s private KeyCommandHandler method, which takes only an 8-bit code to signify the key pressed. With the translation from the 16-bit scancode to the 8-bit scancode, data is lost and keys overlap. This prevents keys such as the Delete and arrow keys to be used. MVALClient uses a modified version of TConsole, correcting some of the problems regarding the scancode conversion to allow the necessary editing functionality of the console.
Application messages are used to allow the console to execute a command and release the application handle, allowing redrawing of the contents of the windows. On execution of a command by the console, the command is sent to the MVALClientThread, and all user interface components which may interfere with the messaging are disabled. The main application window then waits for a WM_HandleResponse message. The handler for this message retrieves the message from MVALClientThread. The handler checks whether the reply is an error of type NOT_SERVER_SUPPORT. Commands that involve file or program management or require user input fall in this class and are handled by the client (see Table 3.9). The decoded structure is sent by the server as part of the response to allow the client to act on the command without the need to reinterpret the command. Any other type of response from the server is echoed in the console window. The handler then enables any previously disabled components of the user interface.

4.3.2 SIMULATION

![Simulation window of MVALClient](image)

The simulation allows offline design and testing of programs. Simulation does not allow real-world objects to be represented in the 3D environment, although they can be easily included.

When the session to the server is created, the client requests file information of the ASE model for the specific driver. The information is compared to a local cache, and if the information differs, the file is downloaded from the server.
OpenGL is used to render the simulation. The 3D simulation manipulator is built using the DH parameters retrieved from the server. To increase speed and decrease memory usage, grouping was used to combine joints into OpenGL object groups called call lists. Using “ASE reader for Delphi” developed by Tom Nuydens, the ASE file is interpreted [23]. The robot manipulator model is then created and saved as a group for each joint (See section 3.3.3).

Integrated into the simulation is the UDP listener. The UDP listener sets the position of the simulation model, allowing the model to follow the movement as seen by the server.

4.3.3 Video Streaming

MVALClient allows the user to monitor a remote robot’s actual movement in near real-time. Streaming from the server is done over a UDP connection. The UDP port on the client is defined as the TCP port number used on the server, plus four. Thus, if the server is listening on port 23210 then the data is streamed to the client on port 23214.

Demuxing and playback of the video stream is handled by a VLC ActiveX component. To decrease the delay on the video stream, the client buffer should be set to the smallest cache that provides smooth video. On a local loopback connection, 30ms proved to be sufficient. The cache settings can be changed using the Preferences in the VLC application.
Chapter 5 Testing and Results

5.1 Overview

The main development of MVAL2006 was concentrated around the development of drivers and the driver interface library. Development then moved on to the client and server applications using these driver interfaces. The focus was on modularity to allow future enhancements or changes to be easily applied. Apart from modularity, the server’s development focused on developing code protected from faults on multiple structural levels.

5.2 Drivers

To be able to develop and test drivers in a real-world environment, drivers for the CS113 and Unimation Inc. RTX robots were implemented. Between these two robots, enough differences exist to allow thorough testing.

The CS113 robot is a 5-DOF robot with no positioning feedback. The CS113 provides communications by means of standard parallel (LPT) port. Commands sent to the CS113 are simple text string commands. Due to these characteristics, testing was first completed on the CS113 before continuing on to the RTX manipulator.

The RTX robot is a 6-DOF robot with positioning feedback. It uses RS-232 serial communications as I/O medium, and provides a rich set of functionality.

Aspects that can be tested are the different DOFs, feedback support (or the lack thereof) as well as different levels of control for functions such as alignment and movement commands, either with a full implementation or using overloaded functions that depend on drivejoints.

Drivers are developed using a two stage process. The first step is the development of the model and simulation properties. This is done after the specific robot and interface is studied. During the study, the focus was on what the robot is capable of, and how the driver interfaces can use the characteristics of the robot to enhance control.
Using the simulation (no hardware access) mode of the driver, it is possible to verify the kinematics and movement of the robot in an offline state.

The second step in the robot driver development is the actual hardware access. The required hardware instructions are inserted and tested where necessary, and are usually limited to only a few functions.

To facilitate the testing of the driver and the development of a driver interface library that can be used by MVALServer, an application was developed. The library used to show 3D simulations of the robot was developed before the driver test application. The simulation library was further enhanced and improved during the development of the driver tester.

Using the driver tester in conjunction with the simulation drivers allows easy testing, and facilitates the fault finding process. Methods are called directly from the driver interface library, resulting in only one abstraction level. Exceptions are with functions that can be implemented on higher levels such as the move and moves commands which can be wrappers around the drivejoints method.

The CS113 manipulator is simplistic in structure, and provides a simple communication protocol for control. Lacking support for feedback, automatic calibration is only possible by driving each joint to its limit. When user input does not stop the joints when their limits are reached, the belts used to drive the joints slip, causing damage.

Apart from the lack of calibration and precise control, the CS113 provides a reasonably fast controller which allows consecutive commands to be executed with a minimal time delay between request and execution. Controller command execution speed is extremely important where the tool is moved along a straight line. The manipulator controller’s supported commands are implemented using joint-interpolation.

To achieve straight line movement, a number of joint-interpolated movement commands are sent, producing close approximation of a straight line. With reasonably fast controllers, straight line movement can be completed within a reasonable time frame. Figure 5.1 and Figure 5.2 illustrate the difference with timestamps for joint-interpolated movements and straight-line movements.
Figure 5.1 CS113 Joint-interpolated tool movement animation and time line

Figure 5.2 CS113 Straight line tool movement animation and time line

The RTX manipulator provides feedback control, allowing accurate control of the manipulator. A complex serial communication protocol implements the necessary functionality for manipulator control. Development of the RTX manipulator’s driver was a tedious task with imprecise protocol specifications provided.

With support for feedback, the focus of the RTX manipulator’s driver was on accuracy, rather than speed. With the focus on accuracy, movement functions
request updated joint positions to calculate accurate simulation positions. Although
the manipulator supports the retrieval of current positions while driving joints, it
requires consecutive calls to retrieve details for each individual joint, quickly adding
up in time. For this reason joint position is only checked at the beginning and end of
drive commands.

This causes a considerable delay of up to two seconds between requests for
execution on the actual movement of the manipulator. These delays can easily be
decreased by removing requests to retrieve current joint positions before execution of
joint commands, and rather storing the information from previously executed
commands. In doing this, simulation timing may be affected, although actual
movement of the robot would not be affected greatly due to the absolute nature of
positions used by the controller.

The RTX manipulator’s controller has built-in support for acceleration and deceler-
ation of joints on each joint command. These properties, although configurable,
make the use of standard movement commands unsuited for implementation for
straight-line tool movements. Each consecutive call would require the robot controller
to calculate the necessary movement characteristics required for each command,
and go through the actions of accelerating and decelerating the joint with each step
of the straight-line movement.

To allow movement of joints bypassing these features, raw commands can be
sent to the controller which are sent directly to the specific joints. These commands
do, however, require constant input to provide smooth results since they only support
a maximum step length of four. When a joint is in movement, and the next step
command is received late, the joint produces jagged movement. This proved to be a
major problem with multitasking environments such as Windows® XP.

Figure 5.3 and Figure 5.4 illustrate the resulting difference between joint-
interpolation and straight-line tool movement for the RTX manipulator. This is effec-
tively the difference between manipulator and computer controlled movement.

Looking at the time lines for each action, it is quite evident that straight-line
movement with the RTX does not provide very good results, with time to completion
increasing twelve-fold. This can be attributed to scheduling and the manipulator
ccontroller communication mode used.
The RTX manipulator's controller is actually two controllers, in communication with each other. Controller switching can be done in either automatic or manual modes. With manual mode, a specific request is sent to the controller to switch to the alternate controller. With automatic switching enabled, the active controller is automatically switched after each command. Automatic switching is the preferred method
of communication. During testing it was found that manual controller switching allows safer communication. Where communication is lost, or external reset commands received, the active controller may be switched, which would affect robot actions extensively. For this reason, manual controller switching is used, guaranteeing commands to be sent to the correct controller, and if the controller is switched accidentally, the program would automatically correct the problem with the next command.

5.3 Compiler

The next step before the client and server could be developed, was to develop the compiler library. A separate application using the compiler library was developed to test the compiler. Testing was focused on interpretation of incorrect commands and the detection of inaccurate input. With testing for inaccurate input detected correctly first, testing time was dramatically decreased.

5.4 Client/Server

With the driver tester, driver interface library, 3D simulation library and compiler library completed, development of the client and server was pursued. The base libraries’ completion allowed testing to be limited, to a large degree, to abstraction levels. Attention was given to the interfacing methods and parameters passed, rather than functionality.

Development of the client and server was done as a single application, running in separate threads, communicating via the local loopback device. Taking into consideration that the server application should be able to run as a service without any graphical user interface, development focused on logs and feedback to the client through the socket connection for debugging.

The client and server were also tested, focusing on incorrect commands, incorrect formatting of XML protocol messages, as well as incorrect parameters specified for MVAL commands.

With client/server applications, an important aspect is the loss of connection which is not normally associated with stand-alone applications, and requires much
greater care in the development process. On the server side, great care needs to be taken to ensure that a robot is powered down correctly. On the client side, there are fewer important aspects to worry about. Due to the nature of the client/server model and the fact that a session is associated with a client connection, termination of connection between the client and server results in the closing of the client, and the termination of the session on the server.

With multiple buffers and threads used to communicate commands from the user to the server execution unit, commands are affected by minor delays. Threads use `sleep` commands to allow other processing units in the program to execute. These delays are of little consequence in most cases, with the teach pendant mode of the client being an exception. With minor delays of around 50ms to 100ms (depending on computer hardware, operating system, processes running on the computers, specific state in which the different threads find themselves and interconnecting network connection), teaching is not as responsive as would be liked. A further problem, as discussed in section 5.2, is individual commands to manipulators affected in reaction time (dependant on the specific manipulator and driver implementation).

Accuracy is limited mostly by the client and the decimal count used in the conversion of numbers to text. MVALClient limits the decimal accuracy to four decimal places, providing sufficient accuracy for the environment MVAL2006 was developed for.
Chapter 6 Conclusion

MVAL2006 is a software suite consisting of a client/server model using an IP-based network for communication, with extensibility through the use of drivers. Using MVAL as robot control language, the MVAL2006 suite provides sophisticated manipulator control. Robotic control functionality is, in most cases, limited by the specific driver and manipulator hardware.

Based on the previous work done by H.O.W. von Petersdorff [29], MVAL2006 provides an interactive graphical user interface for online and offline usage allowing remote control of robotic systems.

The integration of offline usage provides the ability for MVAL2006 to enable a large number of students to be introduced to robotic control. Students can also be introduced to hardware not available to them, by simply using a driver in simulation mode for the specific manipulator.

The use of network communications and video streaming allow for complete control from remote locations. This is suited to laboratory environments which may be hazardous to humans.

With the modular design of MVAL2006, and a simple protocol interface, the work done for this thesis provides great opportunity for further research, development and integration. From integration with programs such as MATLAB using the drivers directly or using a small wrapper application, to the development of alternative clients using the XML communication protocol, MVAL2006 provides tools for all areas of research and development regarding robot manipulators.
References


References


[31] RoboML Language Definition, RoboML Hardware Definition Schema v0.3.2, Last Accessed 12 November 2006, at http://www.roboml.org/roboml-hardware.xsd


## Appendix A: MVAL Language Definition

### Command Notation

<table>
<thead>
<tr>
<th>Console</th>
<th>&lt;instruction&gt; [&lt;space&gt;&lt;arguments&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>[&lt;label&gt;&lt;space&gt;] &lt;instruction&gt; [&lt;space&gt;&lt;arguments&gt;]</td>
</tr>
</tbody>
</table>

### Parameter Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;float&gt;</td>
<td>Floating point number (64 bit)</td>
</tr>
<tr>
<td>&lt;x&gt;</td>
<td>Floating point number; X component of a frame</td>
</tr>
<tr>
<td>&lt;y&gt;</td>
<td>Floating point number; Y component of a frame</td>
</tr>
<tr>
<td>&lt;z&gt;</td>
<td>Floating point number; Z component of a frame</td>
</tr>
<tr>
<td>&lt;dx&gt;</td>
<td>Floating point number; a relative X component of a frame.</td>
</tr>
<tr>
<td>&lt;dy&gt;</td>
<td>Floating point number; a relative Y component of a frame.</td>
</tr>
<tr>
<td>&lt;dz&gt;</td>
<td>Floating point number; a relative Z component of a frame.</td>
</tr>
<tr>
<td>&lt;z rotation&gt;</td>
<td>Floating point number; rotation around Z-axis</td>
</tr>
<tr>
<td>&lt;o&gt;</td>
<td>Floating point number; O component of a frame</td>
</tr>
<tr>
<td>&lt;a&gt;</td>
<td>Floating point number; A component of a frame</td>
</tr>
<tr>
<td>&lt;t&gt;</td>
<td>Floating point number; T component of a frame</td>
</tr>
<tr>
<td>&lt;transf&gt;</td>
<td>Transformation. Name must start with an alphabet character followed by alphanumeric characters. Maximum length of 255 characters.</td>
</tr>
<tr>
<td>&lt;location&gt;</td>
<td>Single or compound transformation. Name must start with an alphabet character followed by alphanumeric characters. Maximum length of 255 characters.</td>
</tr>
<tr>
<td>&lt;program&gt;</td>
<td>Name of program. Name must start with an alphabet character followed by alphanumeric characters. Maximum length of 24 characters.</td>
</tr>
<tr>
<td>&lt;file&gt;</td>
<td>Name of file including extension. In certain cases the extension will be automatically added. Path may not be included.</td>
</tr>
<tr>
<td>&lt;switch&gt;</td>
<td>Switches implemented by the driver. Used as flags inside the driver.</td>
</tr>
<tr>
<td>&lt;channel&gt;</td>
<td>An integer value representing the channel number. Positive values represent the ON state of the channel, and negative values represent the OFF state. Channels are generally used as I/O interfaces supported by the driver. Range depends on the specific driver and channels implemented by it.</td>
</tr>
<tr>
<td>&lt;time&gt;</td>
<td>Time value specified in seconds.</td>
</tr>
<tr>
<td>&lt;label&gt;</td>
<td>Label number.</td>
</tr>
</tbody>
</table>
<i.var> Integer value or temporary integer variable. The variable is accessible only while a program is executing and will be cleared when the program execution is completed or halted. The variable is defined globally, accessible by any program or sub-call to other programs.

<relation> Relational operator.
  - GT Greater
  - GE Greater or Equal
  - NE Not Equal
  - EQ Equal
  - LE Less or Equal
  - LT Less

<operation> Operation operator.
  - + Addition
  - - Subtraction
  - * Multiplication
  - / Division
  - % Remainder/Modulo

<roll> Roll angle in degrees
<pitch> Pitch angle in degrees
<yaw> Yaw angle in degrees

---

**System and Robot Configuration**

**CALIBRATE** *(console only)*

Calibrate the robot. Check specific driver’s manual for precise action performed by this command.

**SPEED <float>*

Change the speed of movement of the robot. This will affect the speed of all commands executed afterwards. The actual speed difference on the movement of the robot manipulator depends on the specific robot driver. The value must be between 0 and 100.

**SWITCH**

Display the status of all the switches available through the specific robot driver.
Appendix A: MVAL Language Definition

**V SIGNAL**
Display the state of the available I/O signals as provided by the driver.

**BASE [[<dx>], [dy], [dz], [z rotation]]**
Set a new base frame for the robot manipulator, relative to the world frame. If the command is executed with no parameters the base is displayed by the client. Default value of omitted parameters is “0.0”.

**TOOL [transf]**
Set the Tool frame equal to the transformation specified. If no parameters are supplied, the current Tool frame’s parameters are displayed by the client.

**WHERE**
The client displays the current position of the robot tool frame.

---

### Defining Programs

**EDIT <program>** *(console only)*
Client opens an editing window allowing the user to create a new, or edit an existing program.

**FORGETP <program>{","<program>** *(console only)*
Remove listed programs from memory.

---

### Program Control

**EXECUTE <program>** *(console only)*
Execute an existing program on the server.

**PAUSE** *(console only)*
Pause the currently running program.

**STEP** *(console only)*
Execute the next command of a paused program.

**RESUME** *(console only)*
Continue execution of a paused program.

**STATUS** *(console only)*
Display the execution status of the server. If a program is currently running, the program name and line number are displayed.
STOP  
(stop console only)
Start execution of a running or paused program.

DIRECTORY  
(stop console only)
Display a list of available programs in the server’s memory.

LISTL [<transf> {“,” <transf>}]  
(stop console only)
Show the components of all listed transformations. If no transformations are specified, all transformations in memory are displayed with individual components.

LISTP <program>  
(stop console only)
Display the contents of specified program loaded in memory.

DELETE <file>  
(stop console only)
Delete the filename specified.

LISTF  
(stop console only)
Display a list of files in the current work directory.

LOAD[L/P] [<file>]  
(stop console only)
Load the specified file into memory. In the case that no file is specified, an open dialog will ask the user to select a file. The program data, as well as the location data will be loaded into memory. In the case that LOAD is followed by “L” or “P”, only the location data or program data will be loaded into memory.

STORE [<file> [“=”<program>{“,”<program>}]  
(stop console only)
Save program and locations in memory to the local storage medium. If no parameters are given, the client will open a save dialog where the user may select a specific location and name. A list of programs to store may be given, separated by commas.

STOREL [<file>]  
(stop console only)
Save all locations to a storage medium.
STOREP [<file> [“=”<program>{“,,”<program>}]]  
(console only)

Save program to storage medium. If no parameters are given, the client will open a save dialog where the user may select a specific location and name. A list of programs to store may be given, separated by commas.

<table>
<thead>
<tr>
<th>Robot Orientation Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>These commands will affect all subsequent commands. Support for these commands depends on the specific robot driver.</td>
</tr>
</tbody>
</table>

**ABOVE**

Change the robot state to an “above” mode, trying to keep the elbow above the gripper.

**BELOW**

Change the robot state to a “below” mode, trying to keep the elbow below the gripper.

**LEFTY**

Change the robot state to “lefty” mode, trying to keep the elbow to the left of the gripper from the robot’s point of view.

**RIGHTY**

Change the robot state to “righty” mode, trying to keep the elbow to the right of the gripper from the robot’s point of view.

<table>
<thead>
<tr>
<th>Switches and signals</th>
</tr>
</thead>
</table>

**DISABLE <switch>**

Disable a driver switch. Support for this command depends on the specific robot driver.

**ENABLE <switch>**

Enable a driver switch. Support for this command depends on the specific robot driver.

**SIGNAL <channel>{“,,”<channel>}**

Set the state of the listed channels. Number of channels supported depends on the specific robot driver.

| Defining Locations |
FORGETL \(<\text{transf}\>\{,<\text{transf}\}>\) \hspace{1cm} (console only)
Remove the listed transformations from memory.

FRAME \(<\text{transf}\> \ "=" \ <\text{origin transf}>\","<\text{x transf}>\","<\text{y transf}>\)
Defines a frame \(<\text{transf}\>\), calculated from the frames given as parameters. The origin of \(<\text{transf}\>\) is placed at the coordinates of \(<\text{origin transf}>\). The X-axis runs through the origin of \(<\text{x transf}>\) and the Y axis such that the origin of \(<\text{y transf}>\) lies in the XY plane.

HERE \(<\text{location}>\)
Save the current robot tool position to memory.

POINT \(<\text{transf}>\ [\ "=" \ [<\text{x}>\] \","\ [<\text{y}>\] \","\ [<\text{z}>\] \","\ [<\text{o}>\] \","\ [<\text{a}>\] \","\ [<\text{t}>\]]\)
Create a new transformation. If the “=” and the following parameters are omitted, the client will provide an input dialog to acquire the necessary values for the new transformation. Parameters may only be omitted when called from the console.

TEACH \(<\text{location}>\) \hspace{1cm} (console only)
The client opens an interactive input console to allow for the interactive teaching of positions.

INVERSE \(<\text{transf}> \ "=" \ <\text{location}>\)
Compute the inverse of \(<\text{location}>\) with the result stored in \(<\text{transf}>\).

SET \(<\text{transf}> \ "=" \ <\text{location}>\)
Set \(<\text{transf}>\) equal to the frame \(<\text{location}>\).

SHIFT \(<\text{transf}> \ "BY" \ [<\text{dx}>\] \","\ [<\text{dy}>\] \","\ [<\text{dz}>]\)
Add the offsets \(<\text{dx}>\), \(<\text{dy}>\) and \(<\text{dz}>\) to the transformation \(<\text{transf}>\)’s coordinates.

---

### Motion Control

**ALIGN**
The tool is rotated so that the tool’s Z axis is aligned with the nearest axis of the world frame. This is useful for lining up the tool before actions are taken.
APPRO <location> ”,” <distance>
Move the tool to the specified distance along the negative Z axis of the tool from the location, using joint interpolation. The direction of the Z axis depends on the specific robot and may be adjusted, changing the tool transformation.

APPROS <location> ”,” <distance>
Move the tool to the specified distance along the negative Z axis of the tool from the location, along a straight line. The direction of the Z axis depends on the specific robot and may be adjusted, changing the tool transformation.

CLOSEI [<distance>]
Change the gap between the gripper’s fingers. The distance between the fingers is specified in millimetres. In the case that the distance is omitted, the gripper is closed to its minimum.

DEPART <distance>
Move the tool by the specified distance along the tool’s Z axis from the starting point. Joint interpolation is used.

DEPARTS <distance>
Move the tool by the specified distance along the tool’s Z axis from the starting point. The tool is moved along a straight line.

DRAW <dx> ”,” <dy> ”,” <dz>
Move the tool by the specified offsets in the tool’s coordinate frame.

DRIVE <jt> ”,” <change> ”,” <speed>
Drive a specific joint by the change specified. Speed is a floating point number between 0 and 100. This number specifies the speed that the joint must be moved as a percentage of the current robot speed.

MOVE <location>
Move the tool to by specified location using joint interpolation.

MOVES <location>
Move the tool to the specified location along a straight line.
MOVEW <dx> "," <dy> "," <dz>
Move the tool by the specified offsets in the world coordinate frame.

OPENI [<value>]
Change the gap between the gripper’s fingers. The distance between the fingers is specified in millimetres. In the case that the distance is omitted, the gripper is opened to its minimum.

READY
Move the robot to its home position. Specific actions and results are dependant on the robot driver.

ROTATE <roll> "," <pitch> "," <yaw>
Rotate the tool by the given angles specified in degrees.

<table>
<thead>
<tr>
<th>Program Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELAY &lt;time&gt;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>GOSUB &lt;program&gt;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>GOTO &lt;label&gt;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>HALT [&lt;text string&gt;]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>IF &lt;i.var&gt; &lt;relation&gt; &lt;i.var&gt; “THEN” &lt;label&gt;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>IFSIG &lt;channel&gt;{&quot;,&quot;&lt;channel&gt;} “THEN” &lt;label&gt;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
PAUSE [<text string>]  
(program only)
Pause execution of a program, displaying a text string on the client if specified. This function is used in programs and not with direct console input.

REMARK [<text string>]  
The program line is marked as a remark and is ignored.

RETURN [<text string>]  
(program only)
End execution of current program, displaying a text message on the client if specified. If the program is the result of a GOSUB call, program execution continues after the GOSUB command, otherwise program execution ends.

SETI <i.var> “=” <i.var> [<operation> <i.var>]  
(program only)
Set an integer variable equal to the value of the expression on the right. The first <i.var> parameter must be a variable name.

TYPE [<text string>]  
(program only)
Display a text string on the client.

TYPEI <i.var>  
(program only)
Display the value of an integer variable or a number on the client.

Hardware Control

HARDWARE “ON” / “OFF”  
Set hardware access on or off. By default, hardware access is off, effectively causing the client to work in a simulation mode. Before enabling the hardware, make sure that the robot workspace is cleared to avoid injuries.
Appendix B: MVAL2006 Server XML Interfacing Protocol

B.1 General Message Structures

<table>
<thead>
<tr>
<th>Structures</th>
</tr>
</thead>
</table>

**<response>**

**Definition**

```xml
<response result="[success|error]" code="number" desc="Description">
  <responsedata>
  </response>
</response>
```

**Description**
Format of response messages from the server to the client.

**<idlist>**

**Definition**

```xml
{idlist>
  <name>ID1</name>
  <name>ID2</name>
  ...
</idlist>
```

**Description**
Structure for holding a list of text strings, such as location ID’s.

**<programstatus>**

**Definition**

```xml
<programstatus state="[idle|running|paused|step]">
  <program>ProgramName</program>
  <line>ExecutionLine</line>
</programstatus>
```

**Description**
Structure holding the current program status.

**<programdata>**

**Definition**

```xml
<programdata name="ID">Code</programdata>
```

**Description**
Contains the program code of the program specified by attribute "name".
Appendix B: MVAL2006 Server XML Interfacing Protocol

---

**<locationdata>**

**Definition**

\[
\text{<locationdata name="ID">CSV</locationdata>}
\]

**Description**

Contains position data of a location variable determined by attribute "name". The transformation's properties are formatted as CSV data in the format "x,y,z,o,a,t".

---

**<programset>**

**Definition**

\[
\text{<programset>}
\text{[<locationdata> | <programdata>]}
\text{[<locationdata> | <programdata>]}
\text{...}
\text{</programset>}
\]

**Description**

Structure holding multiple locations as well as program structures.

---

**<dhlist>**

**Definition**

\[
\text{<dhinfo>}
\text{<base>CSV</base>}
\text{<tool>CSV</tool>}
\text{<dhlist>}
\text{<dh a="###" d="###" alpha="###" theta="###" type={revolute|prismatic} min="###" max="###"/>}
\text{<dh a="###" d="###" alpha="###" theta="###" type={revolute|prismatic} min="###" max="###"/>}
\text{...}
\text{</dhlist>}
\text{</dhinfo>}
\]

**Description**

Contains the data describing a robot manipulator. "###" are placeholders for floating point numbers.

---

**<positions>**

**Definition**

\[
\text{<positions>CSV</positions>}
\]

**Description**

List of values separated by commas. Number of values is equal to the number of joints the robot manipulator has, plus one value representing the gripper’s position.
Appendix B: MVAL2006 Server XML Interfacing Protocol

<asemodel>

**Definition**

- `<asemodel>`
  - `<name>` file name `</filename>`
  - `<date>` file date `</date>`
  - `<compression>` {none | zip} `</compression>`
  - `<mimedata>` Compressed MIME encoding `</mimedata>`

**Description**

Contain the 3D model associated with the currently loaded robot driver. The actual model data is encoded in the MIME encoding.

- Filename: Name of original file
- Filedate: File date of file in “dd/mm/yyyy hh:nn:ss” format.
- Filesize: File size in bytes

<modelinfo>

**Definition**

- `<modelinfo>`
  - `<name>` File name `</filename>`
  - `<date>` File date `</date>`
  - `<size>` File size in bytes `</size>`
  - `<md5>` MD5-128 of model file `</md5>`

**Description**

Detail on the file for the currently loaded robot driver.

- Date: “dd/mm/yyyy hh:nn:ss”

**B.2 Client to server message**

**General Commands**

<login>

**Request**

- `<login` user="username" password="password"/>
- `<login` user="username" password="password" driver="driver"/>

**Response**

Empty

**Description**

This message is used to authenticate users on the server. Authentication may not be necessary, depending on the configuration of the server. An empty response message is returned.
Appendix B: MVAL2006 Server XML Interfacing Protocol

### <logout>

**Request**

```
<logout/>
```

**Response**

Empty

**Description**

This message is used to log the user out and close the session.

### <changepassword>

**Request**

```
<changepassword old="oldpassword" new="newpassword"/>
```

**Response**

Empty

**Description**

Change the password of the currently logged in user.

---

**VAL Commands**

### <execute type="MVAL">**

**Request**

```
<execute type="MVAL">
  <command>MVAL Command String</command>
</execute>
```

**Response**

Empty

**Description**

The server will execute the specified command. A response will be sent immediately to show whether the command request was successful. With "execute" commands, execution may continue after a response message was received.

---

**Program Commands**

### <program action="list"/>

**Request**

```
<program action="list"/>
```

**Response**

```
<idlist>
```

**Description**

Returns a list of programs in the server’s memory for the current session. If the command was successful, the response message will contain a list of programs.
<program action="get">
  Request
  <program action="get">
    <name>ID</name>
  </program>
  Response
  <programdata>
  </programdata>
  Description
  If the program name was found, retrieves the program from the server’s memory. The program code will be returned in a <programdata> structure.
</program>

<program action="set">
  Request
  <program action="set">
    <programdata>
    </programdata>
  </program>
  Response
  Empty
  Description
  Upload a program’s code into the memory of the server for the current session. If the program already exists, it will be overwritten.
</program>

<program action="upload">
  Request
  <program action="upload">
    <programset>
    </programset>
  </program>
  Response
  Empty
  Description
  Uploads a complete program set into memory including location data.
</program>

<program action="remove">
  Request
  <program action="remove">
    <name>ID</name>
  </program>
  Response
  Empty
  Description
  Remove a program from the server’s memory.
</program>
### <program action="clear">

<table>
<thead>
<tr>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;program action=&quot;clear&quot;/&gt;</td>
</tr>
</tbody>
</table>

| Response | Empty |
|----------|

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove all programs and location variables from the server’s memory.</td>
</tr>
</tbody>
</table>

### <program action="execute">

<table>
<thead>
<tr>
<th>Request</th>
</tr>
</thead>
</table>
| <program action="execute">  
  <name>ID</name>  
</program> |

| Response | Empty |
|----------|

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute a program in the server’s memory. An error response will be returned if the execution of the program failed.</td>
</tr>
</tbody>
</table>

### <program action="status">

<table>
<thead>
<tr>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;program action=&quot;status&quot;/&gt;</td>
</tr>
</tbody>
</table>

| Response | <programstatus> |
|----------|

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the execution status of the server.</td>
</tr>
</tbody>
</table>

### <program action="pause">

<table>
<thead>
<tr>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;program action=&quot;pause&quot;/&gt;</td>
</tr>
</tbody>
</table>

| Response | Empty |
|----------|

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pause execution of the program currently running.</td>
</tr>
</tbody>
</table>

### <program action="step">

<table>
<thead>
<tr>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;program action=&quot;step&quot;/&gt;</td>
</tr>
</tbody>
</table>

| Response | Empty |
|----------|

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute the next program line.</td>
</tr>
</tbody>
</table>
### Appendix B: MVAL2006 Server XML Interfacing Protocol

#### <program action="resume">

**Request**

```xml
<program action="resume"/>
```

**Response**

Empty

**Description**

Resume the execution of the program currently running.

#### <program action="stop">

**Request**

```xml
<program action="stop"/>
```

**Response**

Empty

**Description**

Stop running program.

#### Location Commands

#### <location action="list">

**Request**

```xml
<location action="list">
  {<program>ProgramName</program> }
</location>
```

**Response**

```xml
<idlist>
```

**Description**

Retrieves a list of location ID’s. When the `<program>` sub-key has been specified, only the locations used by the specific program will be returned.

#### <location action="get">

**Request**

```xml
<location action="get">
  <name>ID</name>
</location>
```

**Response**

```xml
<locationdata>
```

**Description**

Get the data of a location variable.
<location action="set">
  Request
  <location action="set">
    <locationdata>
    </locationdata>
  </location>
  Response
  Empty
  Description
  Add or change the value of a location variable.
</location action="remove">
  Request:
  <location action="remove">
    <name>ID</name>
  </location>
  Response
  Empty
  Description
  Remove location variable from server's memory.

Driver Commands

<driver action="list">
  Request:
  <driver action="list"/>
  Response
  <idlist>
  Description
  Get a list of available drivers.
</driver action="get">
  Request:
  <driver action="get"/>
  Response
  <name>DriverName</name>
  Description
  Get the name of the currently loaded driver. If no driver is loaded, then return an empty name.
### Appendix B: MVAL2006 Server XML Interfacing Protocol

#### <driver action="load">

<table>
<thead>
<tr>
<th>Request:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;driver action=&quot;load&quot;&gt;</td>
</tr>
<tr>
<td>&lt;name&gt;Driver Name&lt;/name&gt;</td>
</tr>
<tr>
<td>&lt;enablehardware&gt;Boolean&lt;/enablehardware&gt; default=false</td>
</tr>
<tr>
<td>&lt;/driver&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load a robot driver. If &lt;enablehardware&gt; is not specified, the default state “false” will be assumed.</td>
</tr>
</tbody>
</table>

#### <driver action="unload">

<table>
<thead>
<tr>
<th>Request:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;driver action=&quot;unload&quot;/&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unload the current driver.</td>
</tr>
</tbody>
</table>

### Robot Commands

#### <robot action="getconfiguration">

<table>
<thead>
<tr>
<th>Request:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;robot action=&quot;getconfiguration&quot;/&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dhlist&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve a description of the robot manipulator of the currently loaded driver.</td>
</tr>
</tbody>
</table>

#### <robot action="getpositions">

<table>
<thead>
<tr>
<th>Request:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;robot action=&quot;getpositions&quot;/&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;positions&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve the current positions of the joints and gripper of the robot manipulator.</td>
</tr>
</tbody>
</table>
<robot action="getmodel">

Request:
<robot action="getmodel">
<compression>{none|zip}</compression> default=none
</robot>

Response
<asemodel>

Description
Retrieve the 3D model of the robot manipulator.

<robot action="getmodelinfo">

Request:
<robot action="getmodelinfo"/>

Response
<modelinfo>

Description
Retrieve details of the file containing the 3D data of the robot manipulator.

Videostream

<videostream action="list_inputs">

Request:
<videostream action="list_inputs"/>

Response
<idlist>

Description
List available video inputs compatible with DirectShow.

<videostream action="set_input">

Request:
<videostream action="set_input">
<id>Device Name</id>
<codec>Transcode Codec name</codec> (default WMV2)
<bitrate>Bitrate</bitrate> (default 2048)
</videostream>

Response
Empty

Description
Set details about the video input device that will be used to capture the video. For more information, refer to VideoLan’s documentation and support.
Appendix B: MVAL2006 Server XML Interfacing Protocol

<videostream action="list_targets">

Request:
<videostream action="list_targets"/>

Response
<idlist>

Description
List the targets to which the video stream will be streamed.

<videostream action="add_target">

Request:
<videostream action="add_target">
<URL>destination</URL>
</videostream>

Response
Empty

Description
Add destination of the video stream.

<videostream action="remove_target">

Request:
<videostream action="remove_target">
<URL>destination to remove</URL>
</videostream>

Response
Empty

Description
Remove destination from the list of targets for the video stream.

<videostream action="clear_targets">

Request:
<videostream action="clear_targets"/>

Response
Empty

Description
Remove all destinations from the list of targets for the video stream.
Appendix B: MVAL2006 Server XML Interfacing Protocol

<videostream action="start">
Request:
<videostream action="start"/>
Response
Empty
Description
Start video streaming. Should be preceded by call actions add_target and set_input.

<videostream action="stop">
Request:
<videostream action="stop"/>
Response
Empty
Description
Stop video stream.

B.3 Server to client messages

General Messages

<message>
Definition
<message>Message<message>
Description
General message sent to client

<update>
Definition
<update>
<program>running|step|paused|idle|error</program>
</update>
Description
General status update message sent to the client.
library SkeletonDriver;

uses
SysUtils,
RobotMaths in '..\..\Shared\RobotMaths.pas',
MValTypes in '..\..\Shared\MValTypes.pas',
SyncObjs;

{$R *.res}

(* Forward declarations *)

function SetHardware(state : byte) : integer; stdcall;forward;
var
  { CPUFreq : Int64; // Delay: CPU Frequency for timing }
  PositionCriticalSec : TCriticalSection; // Critical section protection
  OldExitMethod : Pointer;
  ModeAbove : Boolean; // For use with Above,Below,
  ModeLefty : Boolean; // For use with Lefty and Righty
  Base,Tool : TEuler;
  HardwareEnabled : Boolean;
  RobotSpeed : Real;

(**/) 

Use query performance counter to wait for specified time

(*procedure Delay(Seconds : Single);
Var
  FinalTime,CurTime : Int64;
begin
  QueryPerformanceCounter(CurTime);
  FinalTime := CurTime + round(Seconds*CPUFreq);
  QueryPerformanceCounter(CurTime);
  while curTime<FinalTime do
    begin
      Sleep(1); // Release CPU
      QueryPerformanceCounter(CurTime);
    end;
end;*)

(**/)

Initialise robot session

function InitRobot: integer;stdcall;
begin
  PositionCriticalSec.Enter;
  { QueryPerformanceFrequency(CPUFreq); // Used by delay }
  (* Code Here: 
  1. Set joint and robot gripper position 
  2. Set Base and Tool variables to default frames 
  3. Set default robot speed 
  4. Set default hardware state to simulation, HardwareEnabled:=false 
  *)
  result := DRV_FUNC_SUCCESS;
  PositionCriticalSec.Leave;
end;

(**/)

Close robot session

function DeinitRobot: integer;stdcall;
begin
  (* Code Here *)
  SetHardware(DISABLE);
  ExitProc := OldExitMethod;
  result := DRV_FUNC_SUCCESS;
end;
Appendix C: Robot Drivers

87

end;

(function GetDHParameter(JntNumber: integer; var aDH: TDH; var JntType: byte; var jMin, jMax: real): integer; stdcall;

begin
(* Code Here
If JntNumber does not exist, return
DRV_FUNC_INVALID_PARAMETER
otherwise
DRV_FUNC_SUCCESS
*)
end;

(function GetInverse(EulerPosition: TEuler; var JointPosition: RealArray): integer; stdcall;

begin
(* Code Here
1. Check if JointPosition was already allocated with
correct size, result DRV_FUNC_INVALID_PARAMETER
2. Position is in world frame, remove Base and Tool elements from the position
3. Perform IK with regard to ModeAbove and ModeLefty variables
4. Check joint constraints (is it within min and max)
If failed, return DRV_FUNC_INVKIN_FAILED otherwise DRV_FUNC_SUCCESS.
This function does not effect any internal variables
*)
end;

(function GetJointCount: integer; stdcall;

begin (* Code Here *)
end;

(function GetJointPositions(var JntPos: RealArray): integer; stdcall;

begin
PositionCriticalSec.Enter;
(* Code Here
1. Check if allocated length of JntPos is correct, result DRV_FUNC_INVALID_ARRAY_LENGTH
2. Get joint positions, result := DRV_FUNC_SUCCESS
*)
PositionCriticalSec.Leave;
end;

(function GetRobotTransform: TTransform; stdcall;

begin
PositionCriticalSec.Enter;
(* Code Here : Get position. Make sure that you do not call a
method which also require a critical section *)
PositionCriticalSec.Leave;
end;

(function SetAbove: integer; stdcall;
begin
ModeAbove := true;
result := DRV_FUNC_SUCCESS;
end;

Set the robot manipulator in an above mode where applicable

(function SetBelow: integer; stdcall;
begin
ModeBelow := true;
result := DRV_FUNC_SUCCESS;
end;

Set the robot manipulator in an below mode where applicable

(process)
Appendix C: Robot Drivers

*******************************************************************************)
{function Below: integer;stdcall;
begin
  ModeAbove := false;
  result := DRV_FUNC_SUCCESS;
end;}
*******************************************************************************

Get current robot base frame
*******************************************************************************)
function GetBase(var EulerTransform: TEuler): integer;stdcall;
begin
  EulerTransform := Base;
  result := DRV_FUNC_SUCCESS;
end;
*******************************************************************************

Get current robot tool frame
*******************************************************************************)
function GetTool(var EulerTransform: TEuler): integer;stdcall;
begin
  EulerTransform := Tool;
  result := DRV_FUNC_SUCCESS;
end;
*******************************************************************************

Set the robot manipulator in an lefty mode where applicable
*******************************************************************************)
{function Lefty: integer;stdcall;
begin
  ModeLefty := true;
  result := DRV_FUNC_SUCCESS;
end;}
*******************************************************************************

Set the robot manipulator in an lefty mode where applicable
*******************************************************************************)
{function Righty: integer;stdcall;
begin
  ModeLefty := false;
  result := DRV_FUNC_SUCCESS;
end;}
*******************************************************************************

Set a new current Base frame
*******************************************************************************)
function SetBase(var EulerTransform: TEuler): integer;stdcall;
begin
  Base := EulerTransform;
  result := DRV_FUNC_SUCCESS;
end;
*******************************************************************************

Set a new current Tool frame
*******************************************************************************)
function SetTool(var EulerTransform: TEuler): integer;stdcall;
begin
  Tool := EulerTransform;
  result := DRV_FUNC_SUCCESS;
end;
*******************************************************************************

Check if hardware mode is currently enabled
*******************************************************************************)
function GetHardware(var state: byte): integer;stdcall;
begin
  if HardwareEnabled then
    state := ENABLE
  else
    state := DISABLE;
  result := DRV_FUNC_SUCCESS;
end;
*******************************************************************************

Enable/Disable hardware connection
State: {ENABLE,DISABLE}
Appendix C: Robot Drivers

function SetHardware(state: byte): integer; stdcall;
begin
  (* Code Here
   1. HardwareEnabled:=True
   2. Move robot to home position (remember critical section)
   3. Set robot speed
  Disabling hardware
   1. HardwareEnabled := false
   3. Make sure current robot speed is not excessive, if so reduce
   2. Move robot to home position (do not change internal positions)
  *)
end;

function GetSpeed(var CurrentSpeed: real): integer;stdcall;
begin
  CurrentSpeed := RobotSpeed;
  result := DRV_FUNC_SUCCESS;
end;

function SetSpeed(var NewSpeed: real): integer; stdcall;
begin
  (* Code Here
   1. Check if 0<NewSpeed<=100, result DRV_FUNC_OUT_OF_RANGE
   3. CurrentSpeed := NewSpeed
   2. If hardware enabled, transmit message to robot manipulator
  *)
end;

function GetSignal(SignalNumber:integer; var State:byte):integer; stdcall;
begin
  (* Code Here *)
end;

function GetSignalCount:integer; stdcall;
begin
  (* Code Here *)
end;

function SetSignal(SignalNumber:integer; State:byte):integer; stdcall;
begin
  (* Code Here *)
end;

function GetSwitch(SwitchNumber:integer; var State:byte):integer; stdcall;
begin
  (* Code Here *)
end;

function GetSwitchCount:integer; stdcall;
begin
  (* Code Here *)
end;
Appendix C: Robot Drivers

(function GetSwitch Count: integer; stdcall;
begin
    (* Code Here *)
end;)

/*****************************************************************************/
Set new value of specific switch
SwitchNumber: 0..N-1
State: {ENABLE, DISABLE}
*****************************************************************************/
(function SetSwitch (SwitchNumber: integer; State:byte):integer; stdcall;
begin
    (* Code Here *)
end;)

/*****************************************************************************/
Calibrate robot and move back to home position
*****************************************************************************/
function Calibrate:Integer;stdcall;
begin
    (* Code Here *)
end;

/*****************************************************************************/
Move robot to home position
*****************************************************************************/
function Ready:Integer;stdcall;
begin
    (* Code Here *)
end;

/*****************************************************************************/
Get cuurent distance between grippers
*****************************************************************************/
function GetGripper(var Gap: real): integer; stdcall;
begin
    (* Code Here *)
end;

/*****************************************************************************/
Get the maximum distance the gripper can open
*****************************************************************************/
function GetGripperMax(var Gap: real): integer; stdcall;
begin
    (* Code Here *)
end;

/*****************************************************************************/
Get the minimum distance the gripper can open
*****************************************************************************/
function GetGripperMin(var Gap: real): integer; stdcall;
begin
    (* Code Here *)
end;

/*****************************************************************************/
Move gripper to new distance "Gap" in milimeters
*****************************************************************************/
function SetGripper(var Gap: real): integer; stdcall;
begin
    (* Code Here *)
end;

/*****************************************************************************/
Move joints to new absolute position. If the hardware is disabled it should be
simulated in regard to position/time/speed. Remember critical sections since
the joint positions may be read concurrently.
*****************************************************************************/
function DriveJoints(JointPos: RealArray): integer; stdcall;
begin
    (* Code Here *)
end;

/*****************************************************************************/
Move joint JointNumber relatively at speed Speed.
Note: Simulation similar to DriverJoints
*****************************************************************************/
(function Drive(JointNumber: integer;delta: real;Speed: real):Integer; stdcall;
begin
    (* Code Here *)
end;)

begin (* Code Here *)
end;

******************************************************************************
Move tool with joint-interpolated algorithm to new absolute position Location
Location: Dynamic Array Length 6 (x,y,z,o,a,t)
Note: Simulation similar to DriverJoints
******************************************************************************
{function Move(Location: RealArray):integer;stdcall;
begin (* Code Here *)
end;}

******************************************************************************
Move tool with straight-line algorithm to new absolute position Location
Location: Dynamic Array Length 6 (x,y,z,o,a,t)
Note: Simulation similar to DriverJoints
******************************************************************************
{function MoveS(Location: RealArray):integer;stdcall;
begin (* Code Here *)
end;}

******************************************************************************
Move tool with joint-interpolated algorithm to new position Location,
with Distance milimeters along the Z axis of the tool's frame
Location: Dynamic Array Length 6 (x,y,z,o,a,t)
Note: Simulation similar to DriverJoints
******************************************************************************
{function Appro(var Location:RealArray;Distance: real):integer;stdcall;
begin (* Code Here *)
end;}

******************************************************************************
Move tool with straight-line algorithm to new position Location,
with Distance milimeters along the Z axis of the tool's frame
Location: Dynamic Array Length 6 (x,y,z,o,a,t)
Note: Simulation similar to DriverJoints
******************************************************************************
{function ApproS(var Location:RealArray;Distance: real):integer;stdcall;
begin (* Code Here *)
end;}

******************************************************************************
Retract tool with joint-interpolated algorithm Distance milimeters along
Z axis of the tool frame from the current position
Distance: Milimeters to retract
Note: Simulation similar to DriverJoints
******************************************************************************
{function Depart(Distance: real):integer;stdcall;
begin (* Code Here *)
end;}

******************************************************************************
Retract tool with joint-interpolated algorithm Distance milimeters along
Z axis of the tool frame from the current position
Distance: Milimeters to retract
Note: Simulation similar to DriverJoints
******************************************************************************
{function DepartS(Distance: real):integer;stdcall;
begin (* Code Here *)
end;}

******************************************************************************
Move tool in the world coordinate frame with relative distances dx,dy and dz
Note: Simulation similar to DriverJoints
******************************************************************************
{function MoveW(dx,dy,dz: Real):integer;stdcall;
begin (* Code Here *)
end;}
Appendix C: Robot Drivers

{******************************************************************************
Move tool in its own frame with relative distances dx, dy and dz
Note: Simulation similar to DriverJoints
******************************************************************************}
{function Draw(dx,dy,dz: real): integer; stdcall;
begin (* Code Here *)
end;
}

{******************************************************************************
Rotate tool around its current position
Note: Simulation similar to DriverJoints
******************************************************************************}
{function Rotate(Roll,Pitch,Yaw: real): integer; stdcall;
begin (* Code Here *)
end;
}

{******************************************************************************
Align tool axis with smaller relative distance from world axis to the world
coordinate system.
Note: Simulation similar to DriverJoints
******************************************************************************}
{function Align: integer; stdcall;
begin (* Code Here *)
end;
}

exports
InitRobot,
DeinitRobot,
GetDHParameter,
GetInverse,
GetJointCount,
GetJointPositions,
GetRobotTransform,
GetBase,
GetTool,
SetBase,
SetTool,
GetHardware,
SetHardware,
GetSpeed,
SetSpeed,
Calibrate,
Ready,
GetGripper,
GetGripperMax,
GetGripperMin,
SetGripper,
DriveJoints;
{ ** Other optional functions available
Above,
Below,
Lefty,
Righty,
GetSignal,
GetSignalCount,
SetSignal,
GetSwitch,
GetSwitchCount,
SetSwitch,
Drive,
Move,
MoveS,
Appro,
ApproS,
Depart,
DepartS,
MoveN,
Draw,
Rotate,
Align;
}

begin
PositionCriticalSec := TCriticalSection.Create;
OldExitMethod := ExitProc;
ExitProc := @DeInitRobot;
end.
Appendix C: Robot Drivers

C.2 CS113

C.2.1 Denavit-Hartenberg Parameters

<table>
<thead>
<tr>
<th>Joint Number</th>
<th>d</th>
<th>a</th>
<th>( \alpha )</th>
<th>( \theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>220 mm</td>
<td>0 mm</td>
<td>90°</td>
<td>( \theta_1 )</td>
</tr>
<tr>
<td>2</td>
<td>0 mm</td>
<td>197 mm</td>
<td>0°</td>
<td>( \theta_2 )</td>
</tr>
<tr>
<td>3</td>
<td>0 mm</td>
<td>150 mm</td>
<td>0°</td>
<td>( \theta_3 )</td>
</tr>
<tr>
<td>4</td>
<td>0 mm</td>
<td>0 mm</td>
<td>-90°</td>
<td>( \theta_4 )</td>
</tr>
<tr>
<td>5</td>
<td>0 mm</td>
<td>0 mm</td>
<td>0°</td>
<td>( \theta_5 )</td>
</tr>
</tbody>
</table>

C.2.2 Inverse Kinematics

Notations

\[ c_1 = \cos(\theta_1) \quad c_2 = \cos(\theta_2) \quad c_3 = \cos(\theta_3) \quad c_4 = \cos(\theta_4) \quad c_5 = \cos(\theta_5) \]

\[ s_1 = \sin(\theta_1) \quad s_2 = \sin(\theta_2) \quad s_3 = \sin(\theta_3) \quad s_4 = \sin(\theta_4) \quad s_5 = \sin(\theta_5) \]

DH Matrices

\[
A_1^1 = \begin{bmatrix} c_1 & 0 & s_1 & 0 \\ s_1 & -c_1 & 0 & 0 \\ 0 & 1 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (A_1^1)^{-1} = \begin{bmatrix} c_1 & s_1 & 0 & 0 \\ 0 & 0 & 1 & -d_1 \\ s_1 & -c_1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

\[
A_2^2 = \begin{bmatrix} c_2 & -s_2 & 0 & a_2c_2 \\ s_2 & c_2 & 0 & a_2s_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (A_2^2)^{-1} = \begin{bmatrix} c_2 & s_2 & 0 & -a_2 \\ -s_2 & c_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

\[
A_3^3 = \begin{bmatrix} c_3 & -s_3 & 0 & a_3c_3 \\ s_3 & c_3 & 0 & a_3s_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (A_3^3)^{-1} = \begin{bmatrix} c_3 & s_3 & 0 & -a_3 \\ -s_3 & c_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

\[
A_4^4 = \begin{bmatrix} c_4 & 0 & -s_4 & 0 \\ s_4 & c_4 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (A_4^4)^{-1} = \begin{bmatrix} c_4 & s_4 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ -s_4 & c_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]
Appendix C: Robot Drivers

\[
A_5^i = \begin{bmatrix}
c_5 - s_5 & 0 & 0 \\
s_5 & c_5 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0 \\
\end{bmatrix} \quad \quad (A_5^i)^{-1} = \begin{bmatrix}
c_5 & s_5 & 0 & 0 \\
- s_5 & c_5 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Calculations

Calculate \( (A_0^i)^{-1} T_0^5 = A_i^5 \):

\[
\begin{bmatrix}
c_1 & s_1 & 0 & 0 \\
0 & 0 & 1 & - d_1 \\
s_1 & - c_1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \times \begin{bmatrix}
n_x & o_x & a_x & p_x \\
n_y & o_y & a_y & p_y \\
n_z & o_z & a_z & p_z \\
0 & 0 & 0 & 1 \\
\end{bmatrix} = \begin{bmatrix}
c_4 & 0 & - s_4 & 0 \\
0 & c_4 & 0 & s_4 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
\end{bmatrix} \begin{bmatrix}
c_5 & - s_5 & 0 & 0 \\
- s_5 & c_5 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
\Rightarrow \begin{bmatrix}
\frac{n_x + s_x}{c_1} & \frac{o_x + s_y}{c_1} & \frac{c_1 a_x + s_1 a_y}{c_1} & \frac{c_1 p_x + s_1 p_y}{c_1} \\
\frac{s_y - c_n}{s_1} & \frac{a_y - c_1}{s_1} & \frac{s_1 a_y - c_1 a_y}{s_1} & \frac{s_1 p_y - c_1 p_y}{s_1} \\
\frac{n_z}{s_1} & \frac{o_z}{c_1} & \frac{a_z}{c_1} & \frac{p_z - d_1}{c_1} \\
0 & 0 & 0 & 1 \\
\end{bmatrix} = \begin{bmatrix}
c_{234} c_5 & - c_{234} s_5 & - s_{234} & c_{23} a_3 + c_2 a_2 \\
s_{234} c_5 & - s_{234} s_5 & c_{234} & s_{23} a_3 + s_2 a_2 \\
- s_5 & - c_5 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Now solve \( \theta_1 \)

\[
s_i p_x - c_i p_y = 0 \\
\therefore \tan(\theta_1) = \frac{p_y}{p_x}
\]

Now solve \( \theta_5 \)

\[
s_i n_x - c_i n_y = - s_5 \\
\therefore s_5 = c_i n_y - s_i n_x \\
s_i o_x - c_i o_y = - c_5 \\
\therefore c_5 = c_i o_y - s_i o_x \\
\tan(\theta_5) = \frac{c_i n_y - s_i n_x}{c_i o_y - s_i o_x}
\]

Now solve \( \theta_3 \)

1. \( c_1 p_x + s_1 p_y = c_{23} a_3 + c_2 a_2 \)
2. \( p_z - d_1 = s_{23} a_3 + s_2 a_2 \)

Square (1) and (2)

\[
(c_1 p_x + s_1 p_y)^2 = (c_{23} a_3 + c_2 a_2)^2 \\
(p_z - d_1)^2 = (s_{23} a_3 + s_2 a_2)^2
\]
Appendix C: Robot Drivers

Factor

\[ k_1^2 = \left( c_{23}a_3 + c_2a_2 \right)^2 \text{ with } k_1 = c_i p_x + s_i p_y \]

\[ k_2^2 = \left( s_{23}a_3 + s_2a_2 \right)^2 \text{ with } k_2 = p_z - d_1 \]

Add

\[ k_1^2 = c_{23}a_3^2 + 2c_{23}a_3c_2a_2 + c_2^2a_2^2 \]

\[ k_2^2 = s_{23}a_3^2 + 2s_{23}a_3s_2a_2 + a_2^2 - c_2^2a_2^2 \]

\[ (1) + (2) : k_1^2 + k_2^2 = c_{23}a_3^2 + 2c_{23}a_3c_2a_2 + c_2^2a_2^2 + s_{23}a_3^2 + 2s_{23}a_3s_2a_2 + a_2^2 - c_2^2a_2^2 \]

Simplify

\[ k_1^2 + k_2^2 = a_3^2 + 2c_{23}a_3c_2a_2 + 2s_{23}a_3s_2a_2 + a_2^2 \]

\[ k_1^2 + k_2^2 = a_3^2 + 2a_4c_3 + a_2^2 \]

\[ c_3 = \left( k_1^2 + k_2^2 - a_3^2 - a_2^2 \right) / 2a_4a_3 \]

\[ s_3 = \sqrt{1 - c_3^2} \]

Now solve \( \theta_2 \)

\[ (3) \quad k_1 = c_{23}a_3 + c_2a_2 = c_i (c_3a_3 + a_2) - s_i (s_3a_3) \]

\[ (4) \quad k_1 = s_{23}a_3 + s_2a_2 = s_2 (c_3a_3 + a_2) - c_2 (s_3a_3) \]

Multiply (3) and (4) by \( s_2 \) and \( c_2 \):

\[ s_2k_1 = s_2c_2 (c_3a_3 + a_2) + s_2^2 (-s_3a_3) \]

\[ c_2k_2 = s_2c_2 (c_3a_3 + a_2) + c_2^2 (s_3a_3) \]

Subtract

\[ s_2k_1 - c_2k_2 = -s_3a_3 \]

\[ \frac{k_2}{k_1} - s_2 = \frac{s_3a_3}{k_1} \]

\[ -c_2 + s_2 \frac{k_1}{k_2} = -\frac{s_3a_3}{k_2} \]

Multiply (3) and (4) by \( c_2 \) and \( s_2 \):

\[ c_2k_1 = c_2^2 (c_3a_3 + a_2) + c_2s_2 (-s_3a_3) \]

\[ s_2k_2 = s_2^2 (c_3a_3 + a_2) + c_2s_2 (s_3a_3) \]

Add (7) to (8)

\[ c_2k_1 + s_2k_2 = c_3a_3 + a_2 \]

Regroup

\[ \frac{k_1}{k_2} + s_2 = \frac{c_3a_3 + a_2}{k_2} \]

\[ c_2 + s_2 \frac{k_2}{k_1} = \frac{c_3a_3 + a_2}{k_1} \]

Subtract (9) from (5)

\[ c_2 \left( \frac{k_2^2 + k_1^2}{k_1k_2} \right) = \frac{s_3a_3}{k_2} + \frac{c_3a_3 + a_2}{k_2} \]
Appendix C: Robot Drivers

\[ c_2 = \frac{k_1 k_2}{k_1^2 + k_2^2} \left( \frac{s_3 a_3}{k_1} + \frac{c_3 a_3 + a_z}{k_2} \right) = \frac{1}{k_1^2 + k_2^2} \left( (s_3 a_3) k_2 + (c_3 a_3 + a_z) k_1 \right) \]

Add (6) to (10)

\[ s_2 \left( \frac{k_2^2 + k_2^2}{k_1 k_2} \right) = \frac{c_3 a_3 + a_z}{k_1} - \frac{s_3 a_3}{k_2} \]

\[ s_2 = \frac{1}{k_1^2 + k_2^2} \left( (c_3 a_3 + a_z) k_2 - s_3 a_z k_1 \right) \]

\[ \tan(\theta_2) = \frac{(c_3 a_3 + a_z) k_2 - s_3 a_z k_1}{(s_3 a_3) k_2 + (c_3 a_3 + a_z) k_1} \]

Now solve \( \theta_4 \)

\[ \tan(\theta_2 + \theta_3 + \theta_4) = \frac{c_i a_x + s_i a_y}{a_z} \]

Summary

\[ \tan(\theta_i) = \frac{p_Y}{p_X} \]

\[ \tan(\theta_3) = \frac{c_i n_x - s_i n_y}{c_i o_y - s_i o_x} \]

\[ k_1 = c_i p_x + s_i p_y \]

\[ k_2 = p_z - d_z \]

\[ c_3 = \left( k_1^2 + k_2^2 - a_z^2 - a_z^2 \right)/2a_z a_3 \]

\[ s_3 = \pm \sqrt{1 - c_3^2} \]

\[ \tan(\theta_3) = \frac{s_3}{c_3} \] (below/above configuration)

\[ \tan(\theta_2) = \frac{(c_3 a_3 + a_z) k_2 - s_3 a_z k_1}{(s_3 a_3) k_2 + (c_3 a_3 + a_z) k_1} \]

\[ \tan(\theta_2 + \theta_3 + \theta_4) = \frac{c_i a_x + s_i a_y}{a_z} \]
C.2.3 DRIVER CODE (CS113.DPR)

library CS113;
uses
  Windows,
  Math,
  SysUtils,
  RobotMaths in '..\..\Shared\RobotMaths.pas',
  MVALTypes in '..\..\Shared\MVALTypes.pas',
  SyncObjs;
{SE drv}
{$R *.res}

Type
  JointDef = Record
    DH : TDH;
    jType : Byte;
    Min, Max : Real;
  end;

Const
  STRAIGHT_STEP_SIZE = 5; // millimeter
  SECONDS_PER_UPDATE = 0.01; // Interval for simulated move updates
  JOINT_COUNT = 5;
  GRIPPER_MIN = 0;
  GRIPPER_MAX = 70;
  GRIPPER_NEST = 0;
  SPEED_NEST = 50;
  LPT_PORT = 888;

  DH : Array [0..JOINT_COUNT-1] of JointDef = (
    (DH:(d: 220 ; a: 0; alpha: 90; theta: 0); jType: JOINTTYPE_ROTATION; Min:-140; Max: 140), // Body
    (DH:(d: 0 ; a: 197; alpha: 0; theta: 45); jType: JOINTTYPE_ROTATION; Min: -35; Max: 115), // Shoulder
    (DH:(d: 0 ; a: 150; alpha: 0; theta:-105); jType: JOINTTYPE_ROTATION; Min:-110; Max: -10), // Elbow
    (DH:(d: 0 ; a: 0; alpha: -90; theta: 90); jType: JOINTTYPE_ROTATION; Min: -4; Max: 176), // Wrist(Up/Down)
    (DH:(d: 0 ; a: 0; alpha: 0; theta: -90); jType: JOINTTYPE_ROTATION; Min:-180; Max: 180)) // Write(Rotate)
  );

  MotorSensitivity : Array [0..JOINT_COUNT-1] of Real = ( // Degree per Step
    0.115,
    0.115,
    0.1,
    0.089, // Motor 4 and 5 sync movement - up/down
    0.089 // Motor 4 and 5 async movement -rotate
  );

  MotorSpeed: Array [0..JOINT_COUNT-1] of Real = ( // seconds per step
    0.0021,
    0.0045,
    0.0021,
    0.0021,
    0.0005
  );

  NestPosition : Array [0..JOINT_COUNT-1] of Real = (0,45,-110,90,-90);

// External dll functions
function Out32(wAddr:word;Out:byte):byte; stdcall; external 'inpout32.dll';
function Inp32(wAddr:word):byte; stdcall; external 'inpout32.dll';

// Forward declarations for functions
function GetJointPositions(var JntPos:RealArray): Integer; stdcall; forward;
function Rotate(Roll,Pitch,Yaw : Real) : Integer; stdcall; forward;
function GetSpeed(var CurrentSpeed : Real) : Integer; stdcall; forward;
function SetSpeed(var NewSpeed : Real) : Integer; stdcall; forward;
function SetHardware(state : byte) : integer; stdcall;forward;
Appendix C: Robot Drivers

function Ready: Integer; stdcall; forward;

Var
  HardwareEnabled : boolean;
  ModeAbove : Boolean;
  Base, Tool : TEuler;
  RobotSpeed : Real;
  RobotSimSpeed : Real;
  RobotGripper : Real;
  Joint : Array [0..JOINT_COUNT-1] of Real; // Joint position in angles
  CPURfreq : Int64;
  PositionCriticalSec : TCriticalSection;
  SwitchList : Array [1..2] of Byte;
  OldExitMethod : Pointer;

(******************************************************************************
Use query performance counter to wait for specified time
*******************************************************************************)
procedure Delay(Seconds : Single);
Var
  FinalTime, CurTime : Int64;
begin
  QueryPerformanceCounter(CurTime);
  FinalTime := CurTime + round(Seconds * CPUFreq);
  QueryPerformanceCounter(CurTime);
  while curTime < FinalTime do
    begin
      Sleep(1); // Release CPU
      QueryPerformanceCounter(CurTime);
    end;
end;

(******************************************************************************
Hardware interface section
*******************************************************************************)
(* Wait for port busy to go low. Try for about Timeout miliseconds
1 - Busy
0 - Not busy
-1 - Timeout
*)
function PrinterBusy(addr : Word; TimeOut : integer) : integer;
var
  b : Byte;
begin
  b := Inp32(addr + 1);
  Timeout := Timeout div 100;
  while ((b and $80) = 0) and (timeout > 0) do
    begin
      delay(0.1);
      dec(timeout);
      b := Inp32(addr + 1);
    end;
  if (Timeout = 0) then
    result := DRV_FUNC_HW_TIMEOUT // Timed out
  else if ((b and $80) <> 0) then
    result := 0 // Not Busy
  else
    result := 1; // Busy
end;

// COMPATIBILITY MODE LPT INTERFACE
function WritePrinterByte(addr : Word; v : byte) : integer;
var
  b : byte;
begin
  // Put data on port
  Out32(addr, v);
  // Set strobe low, wait a bit and bring back up high
  b := Inp32(addr + 2);
  Out32(addr + 2, b and $FE); // Clear bit 0
  delay(0.001); // Minimum of 1us required < 1ms
  Out32(addr + 2, b or 1); // Set bit 0
  result := DRV_FUNC_SUCCESS;
end;
Appendix C: Robot Drivers

function DriverExec(CommandString : String) : integer;
Var
  i:Integer;
  PAddr : Word;
begin
  if HardwareEnabled then
    begin
      PAddr := LPT_PORT;
      // Check status register, wait for not busy
      result := PrinterBusy(PAddr,10000);
      if result <> DRV_FUNC_SUCCESS then exit;
      for i:=1 to length(CommandString) do
        begin
          result := WritePrinterByte(PAddr,Ord(CommandString[i]));
          if result <> DRV_FUNC_SUCCESS then exit;
        end;
      result := WritePrinterByte(PAddr,13);
      if result = DRV_FUNC_SUCCESS then
        result := WritePrinterByte(PAddr,10);
    end
  else
    result := DRV_FUNC_SUCCESS;
end;

Get Robot transform for specific configuration

function GetRobotTransform(J1,J2,J3,J4,J5 : Real) : TTransform;overload;
Var
  t1,t2,t3,t4,t5,a2,a3,d1 : Real;
  c1,s1,c2,s2,c5,s5,c23,c234,s234 : real;
  RobotT : TTransform;
begin
  t1 := DegToRad(J1);
  t2 := DegToRad(J2);
  t3 := DegToRad(J3);
  t4 := DegToRad(J4);
  t5 := DegToRad(J5);
  a2 := DH[1].DH.a;
  a3 := DH[2].DH.a;
  d1 := DH[0].DH.d;
  c1 := cos(t1); s1 := sin(t1);
  c2 := cos(t2); s2 := sin(t2);
  c5 := cos(t5); s5 := sin(t5);
  c23 := cos(t2+t3); s23 := sin(t2+t3);
  c234 := cos(t2+t3+t4); s234 := sin(t2+t3+t4);
  RobotT.n.x := c1*c234*c5 - s1*s5;
  RobotT.n.y := s1*c234*c5 + c1*s5;
  RobotT.n.z := s234*c5;
  RobotT.n.w := 0;
  RobotT.o.x := -c1*c23*a3 + c1*c2*a2;
  RobotT.o.y := -s1*c23*a3 + s1*c2*a2;
  RobotT.o.z := c234;
  RobotT.o.w := 0;
end;
Appendix C: Robot Drivers

RobotT.p.w := 1;
result := MultiplyTransforms(RobotT, EulerToTransf(Tool) );
result := MultiplyTransforms(EulerToTransf(Base),result);
end;

******************************************************************************
function GetRobotTransform : TTransform; overload; stdcall;
begin
PositionCriticalSec.Enter;
result := GetRobotTransform(Joint[0],
Joint[1],
Joint[2],
Joint[3],
Joint[4]);
PositionCriticalSec.Leave;
end;

******************************************************************************
// Internal functions
******************************************************************************

Inverse kinematics
function GetInverse(EulerPosition : TEuler;
var JointPosition : RealArray) : integer; stdcall;
function CheckConstraints(var RadAngle : Single; JointIdx : Integer) : boolean;
Var
aMin,aMax : Single;
begin
aMin := DegToRad(DH[JointIdx].Min);
aMax := DegToRad(DH[JointIdx].Max);
// Try to bring angle closer if out of range by full rotation
if (RadAngle<aMin) then
  RadAngle := RadAngle+2*pi;
if (RadAngle>aMax) then
  RadAngle := RadAngle-2*pi;
// Check if angle valid
result :=  (aMin <= RadAngle ) and (RadAngle <= aMax)
end;

Var
T : TTransform;
invBase,invTool : TTransform;
t1,t2,t3,t4,t5 : Single;
k1,k2 : Single;
c1,s1,c3,s3,a2,a3,d1 : single;
s2,c2 : single;
t3Sign : Single;
t3Iteration : byte;
begin
// Check if parameters sent OK
if length(JointPosition)<>JOINT_COUNT then
begin
  result := DRV_FUNC_INVALID_PARAMETER;
  exit;
end;

// Take Euler position to Base coordinate frame and then tool frame
// JointConfig = Base^-1 * EulerPosition * Tool^-1
invBase := InverseTransform(EulerToTransf(Base));
invTool := InverseTransform(EulerToTransf(Tool));
T := MultiplyTransforms(invBase, EulerToTransf(EulerPosition));
T := MultiplyTransforms(T,invTool);
d1 := DH[0].DH.d;
a2 := DH[1].DH.a;
a3 := DH[2].DH.a;

try
With T do
begin
  // Assume it solution did not work until proven so
  result := DRV_FUNC_INVKIN_FAILED;
  // theta 1
  t1 := ArcTan2Custom(T.p.y, T.p.x);
// theta 5
c1 := cos(t1);
s1 := sin(t1);
t5 := ArcTan2Custom(c1*n.y-s1*n.x, c1*o.y-s1*o.x);

// theta 3, 2 and 4
k1 := c1*T.p.x + s1*T.p.y;
k2 := T.p.z - d1;
c3 := (k1*k1 + k2*k2 - a2*a2 - a3*a3)/(2*a2*a3);
if ModeAbove then
t3Sign := -1
else
t3Sign := 1;

// Check both signs for theta 3 if first option did not give suitable solution
repeat
t3Iteration := 0;
repeat
    inc(t3Iteration);
s3 := t3Sign * sqrt(1-c3*c3);
t3 := ArcTan2Custom(s3,c3);
s2 := (c3*a3+a2)*k2 - s3*a3*k1;
c2 := (s3*a3)*k2 + (c3*a3+a2)*k1;
t2 := ArcTan2Custom(s2,c2);
t4 := -ArcTan2Custom(c1*a.x+s1*a.y,a.z)-t2-t3;
if not (CheckConstraints(t2,1) and CheckConstraints(t3,2) and CheckConstraints(t4,3)) then
    t3Sign := t3Sign * -1
else
    break;
until (t3Iteration>=2);

// Check if final joint angles in limits
if (CheckConstraints(t1,0) and CheckConstraints(t2,1) and CheckConstraints(t3,2) and CheckConstraints(t4,3) and CheckConstraints(t5,4) and (abs(T.p.z-d1-sin(t2+t3)*a3-sin(t2)*a2)<0.001))) then // Extra check for singularity problem
result := DRV_FUNC_SUCCESS;

// Save joint angles back into memory for caller
JointPosition[0] := RadToDeg(t1);
JointPosition[1] := RadToDeg(t2);
JointPosition[2] := RadToDeg(t3);
JointPosition[3] := RadToDeg(t4);
JointPosition[4] := RadToDeg(t5);
end;

end;

*******************************************************************************
Move motors to angle. Use joint angle to calculate motor movement, and modify Joint angles to reflect motor change.
*******************************************************************************

function DriveJoints(JointPos : RealArray) : integer; stdcall;
Var
i,n,count : Integer;
MotorSteps: Array [0..JOINT_COUNT-1] of Integer;
FinalJoint: Array [0..JOINT_COUNT-1] of Real;
JointInc : Array [0..JOINT_COUNT-1] of Real;
WristPitch,WristRoll : Integer;
MaxTime : Real;
begin
    // Make absolute value relative
PositionCriticalSec.Enter;
for i:=0 to JOINT_COUNT-1 do
begin
  FinalJoint[i] := JointPos[i];
  JointPos[i] := FinalJoint[i]-Joint[i];
end;
PositionCriticalSec.Leave;

MaxTime := 0;
// Calculate motor movement and change end position if valid movement
MotorSteps[0] := -Round(JointPos[0]/MotorSensitivity[0]);
  JointPos[1]/MotorSensitivity[2]/2);
WristPitch := Round(JointPos[3]/MotorSensitivity[3] +
WristRoll := Round(JointPos[4]/MotorSensitivity[4]);

// Calculate total timer required for movement
for i:=0 to JOINT_COUNT-1 do
  maxTime := max(abs(MotorSteps[i]*MotorSpeed[i]*100), MaxTime);
MaxTime := MaxTime / RobotSimSpeed;

// Calculate steps required per update
count := Round(MaxTime/SECONDS_PER_UPDATE);
if count>0 then
  for i:=0 to JOINT_COUNT-1 do
    JointInc[i] := JointPos[i]/count;

for n:=0 to count-1 do
  begin
    PositionCriticalSec.Enter;
    for i:=0 to JOINT_COUNT-1 do
      Joint[i] := Joint[i] + JointInc[i];
    PositionCriticalSec.Leave;
    Delay(MaxTime/count);
  end;
result := DRV_FUNC_SUCCESS;
end;

******************************************************************************
Move single joint
******************************************************************************
function Drive(JointNumber : Integer;delta : Real;Speed : Real):Integer;
Var
  J : RealArray;
  OldSpeed, NewSpeed : Real;
begin
  SetLength(J,JOINT_COUNT);
  GetJointPositions(J);
  if (JointNumber>=0) and (JointNumber<JOINT_COUNT) and
    (Speed>=0) and (Speed<=100) then
    begin
      J[JointNumber] := J[JointNumber]+delta;
      GetSpeed(OldSpeed);
      NewSpeed := OldSpeed*Speed/100;
      SetSpeed(NewSpeed);
      result := DriveJoints(J);
      SetSpeed(OldSpeed);
    end;
  result := DRV_FUNC_SUCCESS;
end;
else
  result := DRV_FUNC_OUT_OF_RANGE;
end;

*******************************************************************************
Required functions for driver
*******************************************************************************
function InitRobot : Integer;stdcall;
Var
  i : Integer;
begin
  PositionCriticalSec.Enter;
  HardwareEnabled := false;
  ModeAbove := true;
  result := DRV_FUNC_SUCCESS;
  for i:=0 to JOINT_COUNT-1 do
    Joint[i] := DH[i].DH.theta;
  QueryPerformanceFrequency(CPUFreq);
  Base := Euler(0,0,0,0,0,0);
  Tool := Euler(0,0,-98,0,0,0);
  for i:=0 to JOINT_COUNT-1 do
    Joint[i] := NestPosition[i];
  RobotSpeed := SPEED_NEST;
  SetSpeed(RobotSpeed);
  RobotGripper := GRIPPER_NEST;
  PositionCriticalSec.Leave;
end;
******************************************************************************
function DeinitRobot : Integer;stdcall;
begin
  SetHardware(DISABLE);
  ExitProc := OldExitMethod;
  result := DRV_FUNC_SUCCESS;
end;
******************************************************************************
function GetJointCount: Integer;stdcall;
begin
  result := JOINT_COUNT;
end;
******************************************************************************
function SetHardware(state : byte) : integer; stdcall;
Var
  J: RealArray;
  r: real;
begin
  r:=100;
  // if Hardware enable request, check if not already enabled, otherwise move
  // robot to current simulated position.
  if state = ENABLE then
    begin
      if not HardwareEnabled then
        begin
          HardwareEnabled := true;
          SetLength(J,JOINT_COUNT);
          RobotSpeed := SPEED_NEST;
          result := SetSpeed(RobotSpeed);
        end;
      end;
  end;
  if (JntNumber >= JOINT_COUNT) or (JntNumber<0) then
    result := DRV_FUNC_INVALID_PARAMETER
  else
    begin
      aDH := DH[JntNumber].DH;
      JntType := DH[JntNumber].jType;
      jMin := DH[JntNumber].Min;
      jMax := DH[JntNumber].Max;
      result := DRV_FUNC_SUCCESS;
    end;
end;
******************************************************************************
function GetDHParameter(JntNumber : Integer; Var aDH : TDH; Var JntType:Byte; Var jMin,jMax : Real): integer; stdcall;
begin
  if (JntNumber >= JOINT_COUNT) or (JntNumber<0) then
    result := DRV_FUNC_INVALID_PARAMETER
  else
    begin
      aDH := DH[JntNumber].DH;
      JntType := DH[JntNumber].jType;
      jMin := DH[JntNumber].Min;
      jMax := DH[JntNumber].Max;
      result := DRV_FUNC_SUCCESS;
    end;
end;
******************************************************************************
function SetHardware(state : byte) : integer; stdcall;
Var
  J: RealArray;
  r: real;
begin
  r:=100;
  // if Hardware enable request, check if not already enabled, otherwise move
  // robot to current simulated position.
  if state = ENABLE then
    begin
      if not HardwareEnabled then
        begin
          HardwareEnabled := true;
          SetLength(J,JOINT_COUNT);
          RobotSpeed := SPEED_NEST;
          result := SetSpeed(RobotSpeed);
        end;
if result <> DRV_FUNC_SUCCESS then exit;
  result := Ready;
end
else
  result := DRV_FUNC_SUCCESS;
end
// If hardware disable request and hardware is enabled, move robot to nest.
begin
  if HardwareEnabled then
    begin
      result := SetSpeed(r);
      if result <> DRV_FUNC_SUCCESS then exit;
      result := DriverExec('N')
    end
  else
    result := DRV_FUNC_SUCCESS;
  HardwareEnabled := false;
end;
******************************************************************************
function GetHardware(var state : byte) : integer;stdcall;
begin
  if HardwareEnabled then
    state := ENABLE
  else
    state := DISABLE;
  result := DRV_FUNC_SUCCESS;
end;
******************************************************************************
function GetBase(var EulerTransform : TEuler) : integer;stdcall;
begin
  EulerTransform := Base;
  result := DRV_FUNC_SUCCESS;
end;
******************************************************************************
function SetBase(var EulerTransform : TEuler) : integer;stdcall;
begin
  Base := EulerTransform;
  result := DRV_FUNC_SUCCESS;
end;
******************************************************************************
function GetSignalCount:integer; stdcall;
begin
  // TODO
  result := 16;
end;
******************************************************************************
function GetSignal(SignalNumber:integer; var State:byte):integer; stdcall;
begin
  // GetSignal
  // State = [ENABLE,DISABLE]
  // SignalNumber = [1..16]
  if (SignalNumber<1) or (SignalNumber>GetSignalCount) then
    result := DRV_FUNC_INVALID_PARAMETER
  else
    begin
      State := DISABLE; // Get Signal from I/O pins, must be dependent on simulation mode
      result := DRV_FUNC_SUCCESS;
    end;
end;
******************************************************************************
function SetSignal(SignalNumber:integer; State:byte):integer; stdcall;
begin
  // SetSignal
  // State = [ENABLE,DISABLE]
  // SignalNumber = [1..16]
  if (SignalNumber<1) or (SignalNumber>GetSignalCount) then
    result := DRV_FUNC_INVALID_PARAMETER
  else
    begin
      // Set state of signal I/O pins here, must be dependent on simulation mode
      result := DRV_FUNC_SUCCESS;
    end;
end;
Appendix C: Robot Drivers

function GetSwitchCount:integer; stdcall;
begin
// SwitchList[1..N]
result := Length(SwitchList);
end;

function GetSwitch(SwitchNumber:integer; var State:byte):integer; stdcall;
begin
// State = [0..255]
// SwitchList[1..N]
if (SwitchNumber>=1) and (SwitchNumber<=Length(SwitchList)) then
begin
State := SwitchList[switchNumber];
result := DRV_FUNC_SUCCESS;
end
else
Result := DRV_FUNC_INVALID_PARAMETER;
end;

function SetSwitch(SwitchNumber:integer; State:byte):integer; stdcall;
begin
// SwitchList[1..N]
if (SwitchNumber>=1) and (SwitchNumber<=Length(SwitchList)) then
begin
SwitchList[switchNumber] := State;
result := DRV_FUNC_SUCCESS;
end
else
Result := DRV_FUNC_INVALID_PARAMETER;
end;

function GetTool(var EulerTransform : TEuler) : integer;stdcall;
begin
EulerTransform := Tool;
result := DRV_FUNC_SUCCESS;
end;

function SetTool(var EulerTransform : TEuler) : integer;stdcall;
begin
Tool := EulerTransform;
result := DRV_FUNC_SUCCESS;
end;

function GetSpeed(var CurrentSpeed : Real) : Integer;stdcall;
begin
CurrentSpeed := RobotSpeed;
result := DRV_FUNC_SUCCESS;
end;

function SetSpeed(var NewSpeed : Real) : Integer; stdcall;
Var
b:byte;
b:=1;
if (0<NewSpeed) and (NewSpeed<=100) then
begin
RobotSpeed := NewSpeed;
if RobotSpeed<20 then begin;b:=1;RobotSimSpeed := 24;end
else if RobotSpeed<40 then begin;b:=2;RobotSimSpeed := 37;end
else if RobotSpeed<60 then begin;b:=3;RobotSimSpeed := 50;end
else if RobotSpeed<80 then begin;b:=4;RobotSimSpeed := 65;end
else begin;b:=5;RobotSimSpeed := 70;end;
DriverExec('S'+IntToStr(b));
result := DRV_FUNC_SUCCESS;
end
else
result := DRV_FUNC_OUT_OF_RANGE;
end;

function GetGripperMax(var Gap : Real) : Integer; stdcall;
begin
Gap := GRIPPER_MAX;
result := DRV_FUNC_SUCCESS;
end;
Appendix C: Robot Drivers

function GetGripperMin(var Gap : Real) : Integer; stdcall;
begin
  Gap := GRIPPER_MIN;
  result := DRV_FUNC_SUCCESS;
end;

function GetGripper(var Gap : Real) : Integer; stdcall;
begin
  Gap := RobotGripper;
  result := DRV_FUNC_SUCCESS;
end;

function SetGripper(var Gap : Real) : integer; stdcall;
begin
  if (GRIPPER_MIN<=Gap) and (Gap<=GRIPPER_MAX) then
  begin
    DriverExec('M0,0,0,0,0,'+IntToStr(25*Round(RobotGripper-Gap)));
    RobotGripper := Gap;
    result := DRV_FUNC_SUCCESS;
  end
  else
    result := DRV_FUNC_OUT_OF_RANGE;
end;

function GetJointPositions(var JntPos:RealArray): Integer;stdcall;
Var
  i : Integer;
begin
  if Length(JntPos)<>JOINT_COUNT then
    result := DRV_FUNC_INVALID_ARRAY_LENGTH
  else
    begin
      PositionCriticalSec.Enter;
      for i:=0 to JOINT_COUNT-1 do
        JntPos[i] := Joint[i];
      PositionCriticalSec.Leave;
      result := DRV_FUNC_SUCCESS;
    end;
end;

function Calibrate:Integer;stdcall;
begin
  result := DRV_FUNC_NOTSUPPORTED;
end;

function Ready:Integer;stdcall;
Var
  i : Integer;
begin
  PositionCriticalSec.Enter;
  DriverExec('N');
  for i:=0 to JOINT_COUNT-1 do
    Joint[i] := NestPosition[i];
  PositionCriticalSec.Leave;
  result := DRV_FUNC_SUCCESS;
end;

function Above : integer;stdcall;
begin
  ModeAbove := true;
  result := DRV_FUNC_SUCCESS;
end;

function Below: integer;stdcall;
begin
  ModeAbove := false;
  result := DRV_FUNC_SUCCESS;
end;

function Righty : integer;stdcall;
begin
  result := DRV_FUNC_NOTSUPPORTED;
end;
Appendix C: Robot Drivers

function Lefty : integer;stdcall;
begin
  result := DRV_FUNC_NOTSUPPORTED;
end;

(* Align one of the tool's axis to a world plane. The axis which is the closest
to a world plane will be aligned. *)

function Align : integer;stdcall;
function GetClosestAxis(angle: real; min_a,max_a:real) : Real;
begin
  // modify so that range will be in 0 and 90 deg, since this is all that is necessary
  while angle > 90 do angle := angle - 90;
  while angle < 0 do angle := angle + 90;
  // find angle to closest axis
  if angle<45 then result := -angle
  else if angle<90 then result := 90-angle
  else result := 0;
  // Check if within limits
  while result<min_a do result := result + 90;
  while result>max_a do result := result - 90;
  if (result<min_a) then result := 0;
end;

var
  a_cor,
  t_cor : real;
Eul : TEuler;
begin
  // Get current position
  Eul := TransfToEuler(GetRobotTransform);
  // Make sure not in movement and the find the closest axis
  PositionCriticalSec.Enter;
  t_cor := GetClosestAxis(Eul.t,DH[4].min-Joint[4],DH[4].Max-Joint[4]);
  a_cor := GetClosestAxis(Eul.a,DH[3].min-Joint[3],DH[3].Max-Joint[3]);
  PositionCriticalSec.Leave;
  // Move tool to the closest axis.
  if a_cor<t_cor then
    result := Drive(3,a_cor,100)
  else
    result := Drive(4,t_cor,100);
end;

(* Rotate robot tool around tool origin - This function ignores yaw since this
robot does not support yaw (lack of DOF) *)

function Rotate(Roll,Pitch,Yaw : Real) : Integer; stdcall;
Var
  EulT,EulCur : TEuler;
J : RealArray;
begin
  if Yaw<>0 then
    begin
      result := DRV_FUNC_NOTSUPPORTED;
      exit;
    end;
  SetLength(J,JOINT_COUNT);
  result := GetJointPositions(J);
  if result <> DRV_FUNC_SUCCESS then exit;
  EulT := TransfToEuler(GetRobotTransform(J[0],J[1],J[2],J[3],J[4]));
  EulCur := TransfToEuler(GetRobotTransform);
  // Testing : Begin
  { EulT := TransfToEuler(GetRobotTransform);
    EulT.o := EulT.o + Pitch;
    EulT.a := EulT.a + Roll; }
  // Testing : End
EulT.x := EulCur.x;
EulT.y := EulCur.y;
EulT.z := EulCur.z;

Result := GetInverse(EulT,J);

if result = DRV_FUNC_SUCCESS then
  result := DriveJoints(J);
end;

(******************************************************************************)
exports
// Init Routines
InitRobot,
DeinitRobot,
// Configuration routines
GetJointCount,
GetDHParameter,
GetJointPositions,
GetRobotTransform,
// Robot specific mathematic routines
GetInverse,
// MVAL functions
SetHardware,
GetHardware,
GetBase,
SetBase,
GetTool,
SetTool,
GetSpeed,
SetSpeed,
GetSignalCount,
GetSignal,
SetSignal,
GetSwitchCount,
GetSwitch,
SetSwitch,
GetGripperMax,
GetGripperMin,
GetGripper,
SetGripper,
Calibrate,
Ready,
Righty,
Lefty,
Above,
Below,
Align,
Rotate,
DriveJoints;
(******************************************************************************)
begin
PositionCriticalSec := TCriticalSection.Create;
OldExitMethod := ExitProc;
ExitProc := @DeInitRobot;
end.
C.3 RTX

DENAVIT-HARTENBERG PARAMETERS

<table>
<thead>
<tr>
<th>Joint Number</th>
<th>d</th>
<th>a</th>
<th>α</th>
<th>θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0 mm</td>
<td>0°</td>
<td>0°</td>
</tr>
<tr>
<td>2</td>
<td>0 mm</td>
<td>253.5 mm</td>
<td>0°</td>
<td>θ_2</td>
</tr>
<tr>
<td>3</td>
<td>0 mm</td>
<td>253.5 mm</td>
<td>0°</td>
<td>θ_3</td>
</tr>
<tr>
<td>4</td>
<td>0 mm</td>
<td>0 mm</td>
<td>-90°</td>
<td>θ_4</td>
</tr>
<tr>
<td>5</td>
<td>0 mm</td>
<td>0 mm</td>
<td>90°</td>
<td>θ_5</td>
</tr>
<tr>
<td>6</td>
<td>0 mm</td>
<td>0 mm</td>
<td>0</td>
<td>θ_6</td>
</tr>
</tbody>
</table>

C.3.1 INVERSE KINEMATICS

Notations

\[ c_i = \cos(\theta_i) \quad c_2 = \cos(\theta_2) \quad c_3 = \cos(\theta_3) \quad c_4 = \cos(\theta_4) \quad c_5 = \cos(\theta_5) \quad c_6 = \cos(\theta_6) \]

\[ s_i = \sin(\theta_i) \quad s_2 = \sin(\theta_2) \quad s_3 = \sin(\theta_3) \quad s_4 = \sin(\theta_4) \quad s_5 = \sin(\theta_5) \quad s_6 = \sin(\theta_6) \]

DH Matrices

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & d_1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
(A_0^1)^{-1} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & -d_1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
c_2 & -s_2 & 0 & a_2c_2 \\
s_2 & c_2 & 0 & a_2s_2 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
(A_1^2)^{-1} =
\begin{bmatrix}
c_2 & s_2 & 0 & -a_2 \\
-s_2 & c_2 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
c_3 & -s_3 & 0 & a_3c_3 \\
s_3 & c_3 & 0 & a_3s_3 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
(A_2^3)^{-1} =
\begin{bmatrix}
c_3 & s_3 & 0 & -a_3 \\
-s_3 & c_3 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
c_4 & 0 & -s_4 & 0 \\
s_4 & c_4 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
(A_3^4)^{-1} =
\begin{bmatrix}
c_4 & s_4 & 0 & 0 \\
-s_4 & c_4 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Appendix C: Robot Drivers

\[ A^5_2 = \begin{bmatrix} c_5 & 0 & s_5 & 0 \\ s_6 & 0 & -c_5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (A^5_2)^{-1} = \begin{bmatrix} c_5 & s_5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ s_5 & -c_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ A^6_2 = \begin{bmatrix} c_6 & -s_6 & 0 & 0 \\ s_6 & c_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (A^6_2)^{-1} = \begin{bmatrix} c_6 & s_6 & 0 & 0 \\ -s_6 & c_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

Calculations

Calculate \((A^4_1)^{-1}T = A^6_1\)

\[
\begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \end{bmatrix} = \begin{bmatrix} c_2 - s_2 & 0 & a_2 c_2 \\ s_2 & c_2 & 0 & a_2 s_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_3 - s_3 & 0 & a_3 c_3 \\ s_3 & c_3 & 0 & a_3 s_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_4 & 0 & -s_4 & 0 \\ s_4 & 0 & c_4 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_5 & 0 & s_5 & 0 \\ c_6 & -s_6 & 0 & 0 \end{bmatrix} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z - d_1 \end{bmatrix} = \begin{bmatrix} c_{2345} c_6 - s_{2345} s_6 - c_{2345} c_6 - s_{2345} s_6 \\ s_{2345} c_6 + c_{2345} s_6 - s_{2345} c_6 + c_{2345} s_6 \\ s_{2345} s_6^{\dagger} c_5 - s_5 c_6^{\dagger} s_6^{\dagger} c_5 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} c_2 & a_2 c_3 \\ s_2 & c_2 s_3 \\ 0 & 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & s_6 & c_6 \end{bmatrix} \]

Now solve \(d_1\)

\[ p_z - d_1 = 0 \]
\[ \Rightarrow d_1 = p_z \]

Calculate \((A^4_1)^{-1}(A^3_2)^{-1}(A^2_3)^{-1}(A^1_4)^{-1}T = A^6_1\)
Now solve $\theta_{234}$ (element 3,3)

$$-s_{234}a_3 + c_{234}a_y = 0$$

$$c_{234}a_y = s_{234}a_x$$

$$\Rightarrow \frac{s_{234}}{c_{234}} = \frac{a_y}{a_x}$$

$$\Rightarrow \tan(\theta_{234}) = \frac{a_y}{a_x}$$

Now solve $\theta_6$ (elements 3,1 and 3,2)

$$s_6 = -s_{234}n_x + c_{234}n_y$$

$$c_6 = -s_{234}a_x + c_{234}a_y$$

$$\Rightarrow \tan(\theta_6) = \frac{-s_{234}n_x + c_{234}n_y}{-s_{234}a_x + c_{234}a_y}$$

Now solve $\theta_5$ (elements 1,3 and 2,3)

$$s_5 = \frac{c_{234}a_x + s_{234}a_y}{a_z}$$

$$\Rightarrow \tan(\theta_5) = \frac{c_{234}a_x + s_{234}a_y}{a_z}$$

Refer back to $T = A_6^6$
Now Solve $\theta_3$

(element 1,4) \[ p_x = c_4a_2 + c_2a_3 \]
(element 2,4) \[ p_y = s_2a_2 + s_2a_3 \]

Combine

\[ p_x^2 + p_y^2 = (c_2a_2 + c_3a_3)^2 + (s_2a_2 + s_3a_3)^2 \]
\[ p_x^2 + p_y^2 = 2c_3a_2a_3 + a_2^2 + a_3^2 \]
\[ c_3 = \frac{p_x^2 + p_y^2 - a_2^2 - a_3^2}{2a_2a_3} \]
\[ s_3 = \sqrt{1 - c_3^2} \]
\[ \tan(\theta_3) = \frac{2a_2a_3\sqrt{1 - c_3^2}}{p_x^2 + p_y^2 - a_2^2 - a_3^2} \]

Now solve $\theta_2$

1. \[ p_x = c_2a_2 + c_3a_3 = c_2a_2 + c_2c_3a_3 - s_3s_2a_3 \]
2. \[ p_y = s_2a_2 + s_3a_3 = s_2a_2 + s_2c_3a_3 + s_3c_2a_3 \]

Multiply (1) with $s_2$ and (2) with $c_2$

3. \[ s_2p_x = s_2c_2a_2 + s_2c_3a_3 - s_3s_2a_3 \]
4. \[ c_2p_y = s_2c_2a_2 + s_2c_3a_3 + s_3c_2a_3 \]

Subtract (3) from (4)

5. \[ c_2p_y - s_2p_x = s_3s_2^2a_3 = s_3a_3 \]
6. \[ s_3 = \frac{c_2p_y - s_2p_x}{a_3} \]

Multiply (1) with $c_2$ and (2) with $s_2$

7. \[ c_2p_x = c_2^2a_2 + c_3^2a_3 = s_2s_2c_3a_3 \]
8. \[ s_2p_y = s_2^2a_2 + s_2^2c_3a_3 = s_2s_2c_3a_3 \]

Add (7) and (8)

9. \[ s_2p_y + c_2p_x = c_2a_3 + a_2 \]

Use (6) and (9) to solve $\theta_2$

\[ s_2 = \frac{-p_x s_3a_3 + p_y (c_3a_3 + a_2)}{p_x^2 + p_y^2} \]
\[ c_2 = \frac{p_x s_3a_3 + p_y (c_3a_3 + a_2)}{p_x^2 + p_y^2} \]
 Appendix C: Robot Drivers

\[ \tan(\theta_2) = \frac{-p_z s_3 a_3 + p_y (c_3 a_3 + a_2)}{p_z s_3 a_3 + p_y (c_3 a_3 + a_2)} \]

**Now solve** \( \theta_4 \)

\[ \theta_4 = \theta_{234} - \theta_3 - \theta_2 \]

**Summary**

\[ d_4 = p_x \]

\[ \tan(\theta_{234}) = \frac{a_x}{a_z} \]

\[ \tan(\theta_6) = \frac{s_{234} n_x + c_{234} n_y}{-s_{234} n_x + c_{234} n_y} \]

\[ \tan(\theta_5) = \frac{c_{234} a_z + s_{234} a_x}{a_z} \]

\[ \tan(\theta_4) = \frac{2a_z a_3 \sqrt{1 - c_3^2}}{p_x^2 + p_y^2 - a_z^2 - a_3^2} \]

\[ \tan(\theta_2) = \frac{-p_z s_3 a_3 + p_y (c_3 a_3 + a_2)}{p_z s_3 a_3 + p_y (c_3 a_3 + a_2)} \]

\[ \theta_4 = \theta_{234} - \theta_3 - \theta_2 \]
library RTX;

uses
  Windows,
  Math,
  SysUtils,
  RobotMaths in '..\..\..\Shared\RobotMaths.pas',
  MVALTypes in '..\..\..\Shared\MVALTypes.pas',
  SyncObjs,
  RTXinterface in 'RTXinterface.pas';

{$E drv}

type
  JointDef = Record
    DH : TDH;
    jType : Byte;
    Min,Max : Real;
  end;

Const
  STRAIGHT_STEP_SIZE = 5; // milimeter
  SECONDS_PER_UPDATE = 0.01; // Interval for simulated move updates
  JOINT_COUNT = 6;
  GRIPPER_MIN = 0;
  GRIPPER_MAX = 90;
  GRIPPER_NEST = 45;
  SPEED_NEST = 50; // Check robot comparison

DH : Array [0..JOINT_COUNT-1] of JointDef = {
  (DH:(d: 0 ; a: 0; alpha: 0; theta: 0); jType: JOINTTYPE_PRISMATIC; Min: -33; Max: 915), // Shoulder
   (DH:(d: 0 ; a: 253.5; alpha: 0; theta: 0); jType: JOINTTYPE_ROTATION ; Min: -90; Max: 90), // Upper Arm
   (DH:(d: 0 ; a: 253.5; alpha: 0; theta: 0); jType: JOINTTYPE_ROTATION ; Min: -180; Max: 151), // Lower Arm
   (DH:(d: 0 ; a: 253.5; alpha: 0; theta: 0); jType: JOINTTYPE_ROTATION ; Min: -90; Max: 188), // Wrist(Rotate)
   (DH:(d: 0 ; a: 253.5; alpha: 0; theta: 0); jType: JOINTTYPE_ROTATION ; Min: -90; Max: 181) // Wrist(Roll)
};

NestPosition : Array [0..JOINT_COUNT-1] of Real = (900,0,0,0,90,0);

RatedSpeed : Array [0..JOINT_COUNT] of Real = {
  104.2, [ mm/s ZED ]
  42.8, [ deg/s Shoulder ]
  85.6, [ deg/s Elbow ]
  128.4, [ deg/s Wrist Yaw ]
  92.7, [ deg/s Pitch ]
  92.7, [ deg/s Roll ]
  84.9, [ mm/s Gripper ]
};

function SetHardware(state : byte) : integer; stdcall;forward;
function SetSpeed(var NewSpeed : Real) : Integer;stdcall; forward;

var
  Base,Tool : TEuler;
  OldExitMethod : Pointer;
  PositionCriticalSec : TCriticalSection;
  Joint : Array [0..5] of Real;
  ModeLefty : Boolean;
  RobotSpeed : Real;
  CFUPreq : int64;
  HardwareEnabled : Boolean;
  RobotGripper : Real;

{$R *.res}

*******************************************************************************
Use query performance counter to wait for specified time
procedure Delay(Seconds : Single);
Var
   FinalTime,CurTime : Int64;
begin
   QueryPerformanceCounter(CurTime);
   FinalTime := CurTime + round(Seconds*CPUFreq);
   QueryPerformanceCounter(CurTime);
   while CurTime<FinalTime do
   begin
      Sleep(1); // Release CPU
      QueryPerformanceCounter(CurTime);
   end;
end;

procedure UpdateJointFromRTX;
begin
   // TODO: Protect for dual access
   RTX_GetJointPositions(Joint[0],Joint[1],Joint[2],Joint[3],Joint[4],Joint[5],RobotGripper);
   Joint[0] := 900+Joint[0];
end;

function InitRobot : Integer;stdcall;
Var
   i : Integer;
begin
   PositionCriticalSec.Enter;
   HardwareEnabled := false;
   ModeLefty := true;
   result := DRV_FUNC_SUCCESS;
   for i:=0 to JOINT_COUNT-1 do
   Joint[i] := DH[i].DH.theta;
   QueryPerformanceFrequency(CPUFreq);
   Base := Euler(0,0,0,0,0,0);
   Tool := Euler(0,0,177,0,0,180);
   for i:=0 to JOINT_COUNT-1 do // may remove
   Joint[i] := NestPosition[i];
   RobotSpeed := SPEED_NEST;
   SetSpeed(RobotSpeed);
   RobotGripper := GRIPPER_NEST;
   PositionCriticalSec.Leave;
end;

function DeinitRobot : Integer;stdcall;
begin
   SetHardware(DISABLE);
   ExitProc := OldExitMethod;
   result := DRV_FUNC_SUCCESS;
end;

function GetJointCount: Integer;stdcall;
begin
   result := JOINT_COUNT;
end;

function GetDHParameter(JntNumber : Integer; Var aDH : TDH; Var JntType:Byte; Var jMin,jMax : Real): integer; stdcall;
begin
   if (JntNumber >= JOINT_COUNT) or (JntNumber<0) then
   result := DRV_FUNC_INVALID_PARAMETER
   else
   begin
      aDH := DH[JntNumber].DH;
      JntType := DH[JntNumber].jType;
      jMin := DH[JntNumber].Min;
Appendix C: Robot Drivers

```plaintext
jMax := DH[JntNumber].Max;
result := DRV_FUNC_SUCCESS;
end;
end;

function SetHardware(state : byte) : integer; stdcall;
Var
  t : real;
begin
  result := DRV_FUNC_SUCCESS;
  if (State = ENABLE) then
  begin
    if not HardwareEnabled then
      begin
        HardwareEnabled := true;
        if RTX_Init('com1') then
          begin
            UpdateJointFromRTX;
          end
        else
          result := DRV_FUNC_UNKNOWN;
      end
    else
      if state = DISABLE then
      begin
        if HardwareEnabled then
          begin
            RTX_MoveAbsolute(0,0,0,0,0,100,t);
            RTX_MoveAbsoluteGripper(0,t);
            HardwareEnabled := false;
            if not RTX_DeInitComm then
              result := DRV_FUNC_UNKNOWN;
          end
        else
          state := DISABLE;
          result := DRV_FUNC_SUCCESS;
      end;
  end;
end;
end;

function GetHardware(var state : byte) : integer;stdcall;
begin
  if HardwareEnabled then
    State := ENABLE
  else
    state := DISABLE;
  result := DRV_FUNC_SUCCESS;
end;
end;

function GetBase(var EulerTransform : TEuler) : integer;stdcall;
begin
  EulerTransform := Base;
  result := DRV_FUNC_SUCCESS;
end;
end;

function SetBase(var EulerTransform : TEuler) : integer;stdcall;
begin
  Base := EulerTransform;
  result := DRV_FUNC_SUCCESS;
end;
end;

function GetTool(var EulerTransform : TEuler) : integer;stdcall;
begin
  EulerTransform := Tool;
  result := DRV_FUNC_SUCCESS;
end;
end;

function SetTool(var EulerTransform : TEuler) : integer;stdcall;
begin
  Tool := EulerTransform;
  result := DRV_FUNC_SUCCESS;
end;
end;

function GetSignalCount:integer; stdcall;
begin
  result := 0
end;
end;

function GetSignal(SignalNumber:integer; var State:byte):integer; stdcall;
begin
end;
```

result := DRV_FUNC_NOTSUPPORTED;
end;

******************************************************************************
function SetSignal(SignalNumber:integer; State:byte):integer; stdcall;
begin
result := DRV_FUNC_SUCCESS;
end;

******************************************************************************
function GetSwitchCount:integer; stdcall;
begin
result := 0;
end;

******************************************************************************
function GetSwitch(SwitchNumber:integer; var State:byte):integer; stdcall;
begin
Result := DRV_FUNC_NOTSUPPORTED;
end;

******************************************************************************
function SetSwitch(SwitchNumber:integer; State:byte):integer; stdcall;
begin
Result := DRV_FUNC_NOTSUPPORTED;
end;

******************************************************************************
function GetSpeed(var CurrentSpeed : Real) : Integer;stdcall;
begin
CurrentSpeed := RobotSpeed;
result := DRV_FUNC_SUCCESS;
end;

******************************************************************************
function SetSpeed(var NewSpeed : Real) : Integer; stdcall;
Var
Enc : RTXMotor;
begin
RobotSpeed := NewSpeed;
if HardwareEnabled then
begin
for Enc := Elbow to Wrist2 do
   RTX_MotorWrite(Enc,cp_SPEED,Round(RobotSpeed));
end;
result := DRV_FUNC_SUCCESS;
end;

******************************************************************************
function GetGripperMax(var Gap : Real) : Integer; stdcall;
begin
Gap := GRIPPER_MAX;
result := DRV_FUNC_SUCCESS;
end;

******************************************************************************
function GetGripperMin(var Gap : Real) : Integer; stdcall;
begin
Gap := GRIPPER_MIN;
result := DRV_FUNC_SUCCESS;
end;

******************************************************************************
function GetGripper(var Gap : Real) : Integer; stdcall;
begin
Gap := RobotGripper;
result := DRV_FUNC_SUCCESS;
end;

******************************************************************************
function SetGripper(var Gap : Real) : integer; stdcall;
Var
DelayTime : Real;
GapEnc : Real;
Count : Integer;
i : Integer;
begin
if (GRIPPER_MIN<=Gap) and (Gap<=GRIPPER_MAX) then
begin
   // Calculate new gripper position
   DelayTime := Abs((Gap-RobotGripper)/RatedSpeed[JOINT_COUNT]);
   if HardwareEnabled then
   begin
      RTX_MoveAbsoluteGripper(Gap,DelayTime);
// Simulate gripper movement
Count := Max(1, Round(DelayTime / SECONDS_PER_UPDATE));
GapEnc := (Gap-RobotGripper)/Count;
for i:=1 to Count do
begin
  RobotGripper := RobotGripper + GapEnc;
  Delay(DelayTime/Count);
end;
result := DRV_FUNC_SUCCESS;
end
else
result := DRV_FUNC_OUT_OF_RANGE;
end;

******************************************************************************
function GetJointPositions(var JntPos:RealArray): Integer;stdcall;
Var
  i : Integer;
begin
  if Length(JntPos)<>JOINT_COUNT then
    result := DRV_FUNC_INVALID_ARRAY_LENGTH
  else
  begin
    PositionCriticalSec.Enter;
    for i:=0 to JOINT_COUNT-1 do JntPos[i] := Joint[i];
    PositionCriticalSec.Leave;
    result := DRV_FUNC_SUCCESS;
  end;
end;

******************************************************************************
Get Robot transform for specific configuration
******************************************************************************
function GetRobotTransform(J1,J2,J3,J4,J5,J6 : Real) : TTransform;overload;
Var
  t2,t3,t4,t5,t6 : Real;
c2,s2,
c5,s5,
c6,s6,
c23,s23,
c234,s234 : real;
RobotT : TTransform;
begin
  t2 := DegToRad(J2);
t3 := DegToRad(J3);
t4 := DegToRad(J4);
t5 := DegToRad(J5);
t6 := DegToRad(J6);

  c2 := cos(t2); s2 := sin(t2);
c5 := cos(t5); s5 := sin(t5);
c6 := cos(t6); s6 := sin(t6);
c23  := cos(t2+t3);    s23  := sin(t2+t3);
c234 := cos(t2+t3+t4); s234 := sin(t2+t3+t4);

  RobotT.n.x := c234*c5*c6-s234*s6;
  RobotT.n.y := s234*c5*c6+c234*s6;
  RobotT.n.z := -s5*c6;
  RobotT.n.w := 0;
  RobotT.o.x := -c234*c5*s6 - s234*s5;
  RobotT.o.y := -s234*c5*s6 + c234*s5;
  RobotT.o.z := s5*c6;
  RobotT.o.w := 0;
end;
Appendix C: Robot Drivers

RobotT.p.z := J1;
RobotT.p.w := 1;

result := MultiplyTransforms(RobotT, EulerToTransf(Tool));
result := MultiplyTransforms(EulerToTransf(Base), result);
end;

******************************************************************************
function GetRobotTransform : TTransform; overload; stdcall;
begin
PositionCriticalSec.Enter;
result := GetRobotTransform(Joint[0],
Joint[1],
Joint[2],
Joint[3],
Joint[4],
Joint[5]);
PositionCriticalSec.Leave;
end;

******************************************************************************
Move motors to angle. Use joint angle to calculate motor movement, and
modify joint angles to reflect motor change.
******************************************************************************

function DriveJoints(JointPos : RealArray) : integer; stdcall;
Var
i, n, count : Integer;
FinalJoint: Array [0..JOINT_COUNT-1] of Real;
JointInc: Array [0..JOINT_COUNT-1] of Real;
MaxTime : Real;
begin
result := DRV_FUNC_SUCCESS;
// Make absolute value relative
PositionCriticalSec.Enter;
for i:=0 to JOINT_COUNT-1 do
begin
FinalJoint[i] := JointPos[i];
JointPos[i] := FinalJoint[i]-Joint[i];
end;
PositionCriticalSec.Leave;
for i:=0 to JOINT_COUNT -1 do
if (FinalJoint[i]<DH[i].Min) or (FinalJoint[i]>DH[i].Max) then
begin
//MessageBox(0,PChar(Format('DH J%d
%1.2f<%1.2f<%1.2f', [i+1, DH[i].Min, FinalJoint[i], DH[i].Max])),'',0);
result := DRV_FUNC_OUT_OF_RANGE;
exit;
end;
// Get motor positions
// Execute code on hardware
if HardwareEnabled then begin
RTX_MoveAbsolute(FinalJoint[0]-900,
FinalJoint[1],
FinalJoint[2],
FinalJoint[3],
90-FinalJoint[4],
FinalJoint[5],
RobotSpeed,
MaxTime);
end else begin
// Simulate time
// Calculate total timer required for movement
//MessageBox(0,PChar('Sim='+FloatToStr(RobotSpeed)),'',0);
MaxTime:=0.0000000001;
for i:=0 to JOINT_COUNT-1 do
maxTime := max(abs(JointPos[i]/RatedSpeed[i]), MaxTime);
MaxTime := MaxTime / RobotSpeed*100;
// Calculate steps required per update
end;
// Calculate steps required per update
count := Round(MaxTime/SECONDS_PER_UPDATE);
// Calculate steps required per update
for i:=0 to JOINT_COUNT-1 do
  JointInc[i] := JointPos[i]/count;
if result <> DRV_FUNC_SUCCESS then exit;
// Simulate time for robot to move
for n:=0 to count-1 do
begin
  PositionCriticalSec.Enter;
  for i:=0 to JOINT_COUNT-1 do 
    Joint[i] := Joint[i] + JointInc[i];
  PositionCriticalSec.Leave;
  Delay(MaxTime/count);
end;
// Simulate time for robot to move
if HardwareEnabled then 
begin 
  while RTX_Busy do;
  RTX_Stop(FreeStop);
  UpdateJointFromRTX; // Make sure this can not be called simulaniously
end;
result := DRV_FUNC_SUCCESS;
end;
******************************************************************************
Calculate robot configuration to acquire the designated tool position
specified by EulerPosition.
******************************************************************************
function GetInverse(EulerPosition : TEuler; var JointPosition : RealArray) : integer;stdcall;
function CheckConstraints(var DegAngle : Real; JointIdx : Integer) : boolean;
Var
  aMin,aMax : Single;
begi
  aMin := DH[JointIdx].Min;
  aMax := DH[JointIdx].Max;
  // Try to bring angle closer if out of range by full rotation
  if (DegAngle<aMin) then DegAngle := DegAngle+360;
  if (DegAngle>aMax) then DegAngle := DegAngle-360;
  // Check if angle valid
  result := (aMin <= DegAngle ) and (DegAngle <= aMax)
end;
Var
  T,invBase,invTool : TTransform;
t234,c234,s234 : real;
a2,a3,c2,s2,c3,s3 : real;
begi
  // Check if parameters sent OK
  if length(JointPosition)<>JOINT_COUNT then 
begin 
    result := DRV_FUNC_INVALID_PARAMETER;
    exit;
  end;
  try
    // Take Euler position to Base coordinate frame and then tool frame
    // JointConfig = Base^-1 * EulerPosition * Tool^-1
    invBase := InverseTransform(EulerToTransf(Base));
    invTool := InverseTransform(EulerToTransf(Tool));
    T := EulerToTransf(EulerPosition);
    T := MultiplyTransforms(invBase,T);
    T := MultiplyTransforms(T,invTool);
    a2 := DH[1].DH.a;
    a3 := DH[2].DH.a;
// Inverse Kinematics
With T do
begin
  // Joint 1
  JointPosition[0] := T.p.z;
  // Joint 6
  t234 := arctan2custom(A.y, A.x);
  c234 := cos(t234);
  s234 := sin(t234);
  JointPosition[5] := arctan2custom(c234*N.y - s234*N.x, c234*O.y - s234*O.x);
  // Joint 5
  JointPosition[4] := arctan2custom(c234*A.x + s234*A.y, A.z);
  // Joint 3
  c3 := (sqr(P.x) + sqr(P.y) - sqr(a2) - sqr(a3)) / (2*a2*a3);
  s3 := sqrt(abs(1-sqr(c3)));
  if ModeLefty then
    s3 := -s3
  else
    s3 := s3;
  JointPosition[2] := arctan2custom(s3,c3);
  // Joint 2
  s2 := (-s3*a3*P.x + (c3*a3+a2)*P.y) / (sqr(P.x)+sqr(P.y));
  c2 := ( s3*a3*P.y + (c3*a3+a2)*P.x) / (sqr(P.x)+sqr(P.y));
  JointPosition[1] := arctan2custom(s2,c2);
  // Remove {If JointPosition[1]=0 then
  Messagebox(0,PChar(Format('a2=%1.2f a3=%1.2f s2=%1.2f c2=%1.2f s3=%1.2f c3=%1.2f',
    [a2,a3,s2,c2,s3,c3])),'',0);
  // Joint 4
end;
// Convert angles from radians to degrees
JointPosition[1] := RadToDeg(JointPosition[1]);
JointPosition[2] := RadToDeg(JointPosition[2]);
JointPosition[3] := RadToDeg(JointPosition[3]);
JointPosition[4] := RadToDeg(JointPosition[4]);
JointPosition[5] := RadToDeg(JointPosition[5]);
// Check if angles is within robot arm limits
if CheckConstraints(JointPosition[0],0) and
  CheckConstraints(JointPosition[1],1) and
  CheckConstraints(JointPosition[2],2) and
  CheckConstraints(JointPosition[3],3) and
  CheckConstraints(JointPosition[4],4) and
  CheckConstraints(JointPosition[5],5) then
  result := DRV_FUNC_SUCCESS
else
  result := DRV_FUNC_INVKIN_FAILED;
end;
 exception
  result := DRV_FUNC_INVKIN_FAILED;
end;
**************************************************************
**************************************************************
(function GetLargestValue(Value : RTXValues):RTXMotor;stdcall;
Var
  InterSteps : Array of RTXInterpolateSteps;
  InterJoints : Array of Array [0..JOINT_COUNT-1] of Real;
begin
  result := Zed;
  if abs(Value[Shoulder]) > abs(Value[result]) then result := Shoulder;
}
if abs(Value[Elbow]) > abs(Value[result]) then result := Elbow;
if abs(Value[Yaw]) > abs(Value[result]) then result := Yaw;
if abs(Value[Wrist1]) > abs(Value[result]) then result := Wrist1;
if abs(Value[Wrist2]) > abs(Value[result]) then result := Wrist2;
end;

procedure CalcInter(FromPos, ToPos : RealArray);
Var
  StartEnc, DifEnc, EndEnc : RTXValues;
  Mcnt, count, n, i, StartIdx : SmallInt;
  Mdif : Real;
  SP, // Supposed Position
  DifStep : Array[0..5] of Real;
  JointStep: Array[0..5] of Real;
  AP : Array [0..5] of SmallInt;
  //InterStep : Array[0..5] of SmallInt;
begin
  // Get encoder counts for start point, end point as well as difference
  RTX_Joint_to_EncoderCounts(FromPos[0], FromPos[1], FromPos[2],
  FromPos[3], FromPos[4], FromPos[5],
  StartEnc);
  RTX_Joint_to_EncoderCounts(ToPos[0], ToPos[1], ToPos[2],
  ToPos[3], ToPos[4], ToPos[5],
  EndEnc);
  DifEnc := RTX_EncoderSubtract(EndEnc, StartEnc);
  // Calculate maximum change in encoder values
  Mcnt := Abs(DifEnc[GetLargestValue(DifEnc)]);
  // With 7 step movement and current speed, how many interpolations are required
  Mdif := 7*RobotSpeed/100;
  Count := Trunc(Mcnt/Mdif);
  if Count > 0 then
  begin
    // Calculate array with changes required for each step as well as starting point
    for n:=0 to 5 do
    begin
      SP[n] := FromPos[n];
      AP[n] := Round(SP[n]);
    end;
    DifStep[0] := DifEnc[Zed]/Count;
    DifStep[1] := DifEnc[Shoulder]/Count;
    DifStep[2] := DifEnc[Elbow]/Count;
    DifStep[3] := DifEnc[Yaw]/Count;
    DifStep[4] := DifEnc[Wrist1]/Count;
    DifStep[5] := DifEnc[Wrist2]/Count;
    // Calculate and store steps for each interpolation movement
    StartIdx := Length(InterSteps);
    SetLength(InterSteps, StartIdx + Count);
    SetLength(InterJoints, StartIdx + Count);
    for n:=0 to 5 do
    begin
      JointStep[n] := (ToPos[n] - FromPos[n])/count;
      for i:=0 to Count-1 do
      begin
        // Calculate new supposed position
        for i:=0 to 5 do SP[i] := SP[i] + DifStep[i];
        for i:=0 to 5 do InterJoints[StartIdx+n][i] := JointStep[i];
        // Calculate steps required to get to supposed position
        for i:=0 to 5 do InterSteps[StartIdx+n][i] := Round(SP[i] - AP[i]);
        for i:=0 to 5 do
        begin
          if InterSteps[StartIdx+n,i] < -8 then InterSteps[StartIdx+n,i] := -8
          else if InterSteps[StartIdx+n,i]>7 then InterSteps[StartIdx+n,i] := 7;
          AP[i] := AP[i] + InterSteps[StartIdx+n,i];
        end;
      end;
    end;
    Var
      count, i, n : Integer;
    StartJointPos, JointPos : RealArray;
    Eul : TEuler;
    CurrentPos, EndPos, IncT : TTransform;
Appendix C: Robot Drivers

```
mlen : real;
//Temp : RTXInterpolateSteps;
//Temp2 : Array [0..5] of Real;

begin
// Prepare variables
SetLength(JointPos,GetJointCount);
SetLength(StartJointPos,Length(JointPos));
CurrentPos := GetRobotTransform;
// Get final joint positions
if (Length(Location)<>6) then
begin
result := DRV_FUNC_INVALID_ARRAY_LENGTH;
exit;
end;
Eul := ArrayToEuler(Location);
result := GetInverse(Eul,JointPos);
if (result <> DRV_FUNC_SUCCESS) then exit;
EndPos := EulerToTransf(Eul);
// Perform movement
if (StepLength < 0.1) then
result := DriveJoints(JointPos)
else
begin
SetLength(InterSteps,0);
// Calculate number of points along the way
IncT := MinusTransforms(EndPos,CurrentPos);
mlen := sqrt(sqr(IncT.p.x)+sqr(IncT.p.y)+sqr(IncT.p.z));
count := Trunc(mlen/StepLength)+1;
IncT := ScaleTransform(IncT,1/Count);
// Create interpolation array.
GetJointPositions(StartJointPos);
while (count > 0) do
begin
CurrentPos := AddTransforms(CurrentPos,IncT);
Eul := TransfToEuler(CurrentPos);
if GetInverse(Eul,JointPos) = DRV_FUNC_SUCCESS then
begin
CalcInter(StartJointPos,JointPos);
for i:=0 to JOINT_COUNT-1 do
StartJointPos[i] := JointPos[i];
end;
dec(count);
end;
// Do movement
LogFile(0,'InterSteps '+IntToStr(i)+'/'+IntToStr(Length(InterSteps)))
for n:=0 to 5 do
Joint[n] := Joint[n]+InterJoints[i][n];
RTX_MoveInterP(InterSteps[i]);
//sleep(10);
(Temp[0]:=Temp[0] + InterSteps[i][0];
Temp[1]:=Temp[1] + InterSteps[i][1];
Temp[2]:=Temp[2] + InterSteps[i][2];
Temp[3]:=Temp[3] + InterSteps[i][3];
Temp[4]:=Temp[4] + InterSteps[i][4];
Temp[5]:=Temp[5] + InterSteps[i][5];
}
end;
RTX_Stop(FreeOff);
UpdateJointFromRTX;
// RTX_Stop(FreeOff);
{ MessageBox(0,PChar(Format('Motor1=%d
'Motor2=%d
'Motor3=%d
'Motor4=%d
'Motor5=%d',
Temp[1],Temp[2],Temp[3],Temp[4],Temp[5]));

```
Appendix C: Robot Drivers

'Motor5=%d'#10+
'Motor6=%d'#10,
(Temp[0],Temp[1],Temp[2],Temp[3],Temp[4],Temp[5])), ''), 0);

{SetLength(InterSteps,0);
IncT := MinusTransforms(EndPos,CurrentPos);
mlen := sqrt(sqr(IncT.p.x)+sqr(IncT.p.y)+sqr(IncT.p.z));
count := Trunc(mlen/StepLength)+1;
IncT := ScaleTransform(IncT,1/Count);
result := DRV_FUNC_SUCCESS;

// Try to move in straight line, if not possible, skip middle points. Will
// only fail if final position was not achieved
while(count > 0) do
begin
GetJointPositions(StartJointPos);
CurrentPos := AddTransforms(CurrentPos,IncT);
Eul := TransfToEuler(CurrentPos);
if GetInverse(Eul,JointPos) = DRV_FUNC_SUCCESS then
begin
CalcInter(StartJointPos,JointPos);
MessageBox(0,PChar(Format('Count=%d InterSteps=%d', [Count,Length(InterSteps)])),'',0);
for i:=0 to Length(InterSteps)-1 do
RTX_MoveInterP(InterSteps[i]);
for i:=0 to JOINT_COUNT-1 do
StartJointPos[i] := JointPos[i];
//result := DriveJoints(JointPos);
end;
dec(count);
end;}
Appendix C: Robot Drivers

begin
end;

function Below: integer;stdcall;
begin
result := DRV_FUNC_NOTSUPPORTED;
end;

function Righty : integer;stdcall;
begin
ModeLefty := False;
result := DRV_FUNC_SUCCESS;
end;

function Lefty : integer;stdcall;
begin
ModeLefty := True;
result := DRV_FUNC_SUCCESS;
end;

Align one of the tool's axis to a world plane. The axis which is the closest to a world plane will be aligned.

(*function Align : integer;stdcall;
begin
result := DRV_FUNC_NOTSUPPORTED;
end;*)

(* Rotate robot tool around tool origin - This function ignores yaw since this robot does not support yaw (lack of DOF) *)

(*function Rotate(Roll,Pitch,Yaw : Real) : Integer; stdcall;
begin
result := DRV_FUNC_NOTSUPPORTED;
end;*)

exports

// Init Routines
InitRobot,
DeInitRobot,
// Configuration routines
GetJointCount,
GetDHParameter,
GetJointPositions,
GetRobotTransform,
MoveRobot,
// Robot specific mathematic routines
GetInverse,
// MVAL functions
SetHardware,
GetHardware,
GetBase,
SetBase,
GetTool,
SetTool,
GetSpeed,
SetSpeed,
GetSignalCount,
GetSignal,
SetSignal,
GetSwitchCount,
GetSwitch,
SetSwitch,
GetGripperMax,
GetGripperMin,
GetGripper,
Calibrate,
Ready,
Righty,
Lefty,
Above,
Below,
// Align,
// Rotate,
DriveJoints;

begin
PositionCriticalSec := TCriticalSection.Create;
OldExitMethod := ExitProc;
ExitProc := @DeInitRobot;
end.
C.3.3 HARDWARE INTERFACE LIBRARY (RTXINTERFACE.PAS)

unit RTXInterface;

interface

Uses
Windows, SysUtils, Classes;

type
ByteSet = Set of Byte; // : Remove
ByteArray3 = Array [0..2] of Byte;
RTXSoakMode = {Soak_Initialise,
   Soak_Start,
   Soak_Initialise_and_start,
   Soak_stop};
IPToggleMode = {Off, Once, Auto};
IPResetMode = {Full, DefineHome, Values};

RTXMotor = {Elbow, Shoulder, Zed, Yaw, Gripper, Base1, Base2, Wrist1, Wrist2, Sensor};
RTXMotorSet = Set of RTXMotor;
RTXMotorMode = {PositionMode=0, ForceMode, AbsoluteMode, RelativeMode, UserIOInput, UserIOOutput};
//RTX_Joint = {Zed, Shoulder, Elbow, WristYaw, WristPitch, WristRotate, Gripper};
RTXStopMode = {Dead, Ramp, FreeStop, FreeOff};
RTXControlParameters = (cp_ERROR,
   cp_CURRENT_POSITION,
   cp_ERROR_LIMIT,
   cp_NEW_POSITION,
   cp_SPEED,
   cp_KP,
   cp_KI,
   cp_KD,
   cp_DEAD_BAND,
   cp_OFFSET,
   cp_MAX_FORCE,
   cp_CURRENT_FORCE,
   cp_ACCELARTION_TIME,
   cp_USER_RAM,
   cp_USER_IO,
   cp_ACTUAL_POSITION);

RTXValues = Array [RTXMotor] of Smallint;
RTXInterpolateSteps = Array [0..5] of SmallInt;
RTXInfo = record
   Ratio : real; // Encoder counts per mm/degree
   Speed : real; // Maximum Encoder count's per second
   MinEnc, MaxEnc : Smallint; // Minimum and maximum encoder counts
end;

const
ZED_SPEED_MUL = 1.6;
// Response codes from RTX robot
RSP_ACKNOWLEDGE = $0;
RSP_TASK_IN_PROGRESS = $1;
RSP_COMMAND_STORED = $2;
RSP_AXIS_BUSY = $3;
RSP_COMMAND_DECODER_BUSY = $4;
RSP_PARAMETER_OUT_OF_RANGE = $5;
RSP_FAILED = $6;
RSP_READ_ONLY = $7;
RSP_SELECTION_OUT_OF RANGE = $8;
RSP_COMMAND_OUT_OF_RANGE = $9;
RSP_COMMAND_NOT_SUPPORTED = $a;
RSP_FRAME_TIMEOUT = $b;
RSP_FRAME_OVERRUN = $c;
RSP_PARITY_TIMEOUT = $d;
RSP_IP_RESTARTED = $f;
RSP_3_BYTE_RESPONSE_FOR_STATUS = $;
RSP_STATUS_RESPONSE_FROM_CONTROLLER_0 = $10;
RSP_STATUS_RESPONSE_FROM_CONTROLLER_1 = $11;
RSP_STATUS_RESPONSE_FROM_CONTROLLER_2 = $12;
RSP_STATUS_RESPONSE_FROM_CONTROLLER_3 = $13;
RSP_STATUS_RESPONSE_FROM_CONTROLLER_4 = $14;
RSP_STATUS_RESPONSE_GENERAL = $17;
Appendix C: Robot Drivers

RSP_IDENTIFICATION_RESPONSE_IP0 = $20;
RSP_IDENTIFICATION_RESPONSE_IP1 = $21;

{  RTXGearRatio : Array [RTXMotor] of Real = (
    1,        // Base1
    1,        // Base2
    13.4862,  // Wrist1
    13.4862,  // Wrist2
    1,        // Sensor
    14.6113,  // Elbow
    29.2227,  // Shoulder
    3.74953,  // Zed
    9.73994,  // Gripper
    1         // Sensor
  );}

RTXMotorInfo : Array [RTXMotor] of RTXInfo = (
  (Ratio:14.6113; Speed:6250/60*12; MinEnc:-2630; MaxEnc:2206), //Elbow
  (Ratio:29.2227; Speed:6250/60*12; MinEnc:-2630; MaxEnc:2630), //Shoulder
  (Ratio:3.74953; Speed: 975/60*12; MinEnc:-3554; MaxEnc:   0), //Zed
  (Ratio:9.73994; Speed:6250/60*12; MinEnc: -30; MaxEnc:1200), //Yaw
  (Ratio:1; Speed: 1; MinEnc: 0; MaxEnc: 0), //Gripper
  (Ratio:1; Speed: 1; MinEnc: 0; MaxEnc: 0), //Base1
  (Ratio:1; Speed: 1; MinEnc: 0; MaxEnc: 0), //Base2
  (Ratio:13.4862; Speed:6250/60*12; MinEnc:-3101; MaxEnc:2495), //Wrist1
  (Ratio:13.4862; Speed:6250/60*12; MinEnc:-3762; MaxEnc:1834), //Wrist2
  (Ratio:1; Speed: 1; MinEnc: 0; MaxEnc: 0) //Sensor
);

procedure LogFile(Level : Integer; Msg : String);

function comInit(ComPort : String) : Boolean;
procedure comClose;
procedure comRequest(RequestCount: Byte; RequestData: ByteArray3;
  var ResponseCount: Byte; var ResponseData: ByteArray3); overload;
procedure comRequest(Command: Byte; var Response: Byte); overload;
procedure comRequest(Command: Byte; var ResponseCount : Byte; var Response: ByteArray3);
overload;

function RTX_Init(ComPort : String): Boolean;
function RTX_DeInitComm: Boolean;
function RTX_IP_GoActive : byte;
function RTX_IP_Identification : byte;
function RTX_IP_Toggle(Switch : IPToggleMode) : byte;
function RTX_IP_Select(IP : Byte) : byte;
function RTX_IP_Reset(Reset : IPResetMode) : byte;
function RTX_SoakMode(Mode : RTXSoakMode): byte;
function RTX_GetMotorStatus(Motor : RTXMotor): byte;
function RTX_GetGeneralStatus(IP : Byte): byte;
function RTX_SetMotorMode(Motor : RTXMotor; MotorMode : RTXMotorMode): boolean;
function RTX_SetMode(MotorMode : RTXMotorMode): boolean;
function RTX_Busy : boolean; overload;
function RTX_Busy(Motors : RTXMotorSet) : boolean; overload;
procedure RTX_DefineHome;
procedure RTX_Stop(StopMode : RTXStopMode);
function RTX_MotorRead(Motor: RTXMotor; Parameter: RTXControlParameters;
  var Value: SmallInt) : byte;
function RTX_MotorWrite(Motor: RTXMotor; Parameter: RTXControlParameters; Value: SmallInt) : byte;
procedure RTX_GoNumeric_All;
procedure RTX_GoNumeric(Motors : RTXMotorSet);
procedure RTX_MotorReadAll(cp_Parameter: RTXControlParameters;var Values:RTXValues);
procedure RTX_MotorWriteAll(cp_Parameter: RTXControlParameters;Values:RTXValues);
procedure RTX_Joint_to_EncoderCounts(mmZed,degShoulder,degElbow,degYaw,degPitch,degRotate : Real; var EncoderCounts:RTXValues);
function RTX_EncoderAdd(A,B : RTXValues):RTXValues;
function RTX_EncoderSubtract(A,B : RTXValues):RTXValues;
function RTX_EncoderMultiply(A: RTXValues;B:Real):RTXValues;
function RTX_EncoderDivide(A: RTXValues;B:Real):RTXValues;
procedure RTX_MoveMotor(Motor : RTXMotor; newPos : SmallInt);
procedure RTX_MoveInterP(steps:RTXInterpolateSteps);
Appendix C: Robot Drivers

procedure RTX_MoveAbsolute(mmZed, DegShoulder, degElbow, degYaw, degPitch, degRotate : Real; Speed : Real; var Eta : Real);
procedure RTX_MoveAbsoluteGripper(Gap : Real; var Eta : Real);
procedure RTX_GetJointPositions(var mmZed, degShoulder, degElbow, degYaw, degPitch, degRotate, mmGripper : Real);
procedure RTX_Calibrate;

implementation

Var
ComHandle : THandle; // Handle to serial communications port
LogLevel : SmallInt = 0;

(** : Remove this function **)  
(*procedure CheckResult(result : Byte; Values : ByteSet; Msg : String);  
begin  
// TODO : Remove  
if not result in Values then  
MessageBox(0,PChar(Msg+#13#10+IntToHex(result,2)),'',0);  
end;*)

procedure LogFile(Level : Integer;Msg : String);
{Var  
F: TextFile;  
i : Integer;}
begin  
// TODO : Remove this function for logging  
if Level < 0 then LogLevel := LogLevel + Level;  
for i:=0 to LogLevel do Msg := '  '+Msg;  
if Level > 0 then LogLevel := LogLevel + Level;  
AssignFile(F,'c:	emp\log.txt');  
{$I-}  
Append(F);  
{$I+}  
If IOResult <> 0 then ReWrite(F);  
Writeln(F,Msg);  
CloseFile(F);}

(*** Port communication commands ***)

(********************************************************************************
Open the serial communications port for example "com1" or "com2", and then  
sets its baudrate to 9600, no parity and 1 stop bit.  
******************************************************************************* *)
function comInit(ComPort : String) : Boolean;
Var  
dcb : TDCB;  
begin  
result := false;  
// Open handle to serial communications port connected to RTX robot  
ComHandle := CreateFile( PChar(ComPort), GENERIC_READ or GENERIC_WRITE,  
0,nil,OPEN_EXISTING,0,0);  
if ComHandle = INVALID_HANDLE_VALUE then  
exit  
else  
begin  
// Set parameters for communications  
SetupComm(ComHandle,0,0);  
GetCommState(ComHandle,dcb);  
dcb.BaudRate := 9600;  
dcb.Parity := NOPARITY; // different from specifications, but works better.  
dcb.StopBits := ONESTOPBIT;  
dcb.ByteSize := 8;  
SetCommState(ComHandle,dcb);  
result := true;  
end;  
end;

(*******************************************************************************
Closes the com port.  
******************************************************************************* )
procedure comClose;
begin
  CloseHandle(ComHandle);
  ComHandle := 0;
end;

(******************************************************************************
Perform a request and receives response from RTX robot over the com port.
*******************************************************************************)

procedure comRequest(RequestCount: Byte; RequestData: ByteArray3;
  var ResponseCount: Byte; var ResponseData: ByteArray3); overload;
Var
  Count : Cardinal;
  n : Integer;
  S : String; // : Remove
begin
  // Sleep(16);
  // Send data to robot
  if not WriteFile(ComHandle, RequestData, RequestCount, Count, nil) or
    (Count<>RequestCount) then exit;

  // Read response type and decode if necessary to retrieve any more bytes
  if not ReadFile(ComHandle, ResponseData[0], 1, Count, nil) or (Count <> 1) then
    begin
      ResponseData[0] := RSP_FAILED;
      exit;
    end;

  // Get response size from response type byte
  if (ResponseData[0] and $F0 = $10) or // 0001xxxx - status
      (ResponseData[0] and $F0 = $d0) or // 1101xxxx - deferred read
      (ResponseData[0] and $F0 = $80) then // 1000xxxx - immediate read
    ResponseCount := 3
  else
    ResponseCount := 1;

  // Read response data
  for n:=1 to ResponseCount-1 do
    if not ReadFile(ComHandle,ResponseData[n],1,Count,nil) or (Count <> 1) then
      begin
        ResponseData[0] := RSP_FAILED;
        break;
      end;

  // : Remove this code section
  {  if (ResponseData[0] <> 0) and (ResponseData[0] <> $20) and (ResponseData[0] <> $21) then
    begin
      if RequestCount=1 then
        begin
          if RequestData[0] = $1 then
            S:='Get IP identification'
          else if RequestData[0] = $29 then
            S:='Toggle IP'
          else
            S:=Format('Written 0x%s  ',[IntToHex(RequestData[0],2)])
        end
      else
        S:=Format('Written 0x%s 0x%s 0x%s
          ',[IntToHex(RequestData[0],2),IntToHex(RequestData[1],2),IntToHex(RequestData[2],2)]);
    end
  else
    S:=Format('Written 0x%s 0x%s 0x%s
          ',[IntToHex(RequestData[0],2),IntToHex(RequestData[1],2),IntToHex(RequestData[2],2)]);

  if ResponseCount=1 then
    begin
      if ResponseData[0] = $20 then
        S:=S+' - IP0 selected'
      else if ResponseData[0] = $21 then
        S:=S+' - IP1 selected'
      else
        S:=S+Format('  Read 0x%s  ',[IntToHex(ResponseData[0],2)])
    end
  else
    S:=S+Format('  Read 0x%s 0x%s 0x%s
          ',[IntToHex(ResponseData[0],2),IntToHex(ResponseData[1],2),IntToHex(ResponseData[2],2)]);

  if ResponseData[0] = $0f then LogFile(0,'******** IP RESTARTED **********');
LogFile(0,S);
//MessageBox(0,PChar(S),'',0);
{ end; }

end;

*******************************************************************************
Perform a request and receives response from RTX robot over the com port.
*******************************************************************************

procedure comRequest(Command: Byte; var ResponseCount : Byte; var Response: ByteArray3); overload;
Var
WriteData: ByteArray3;
begin
WriteData[0] := Command;
comRequest(1,WriteData,ResponseCount,Response);
end;

*******************************************************************************
Perform a request and receives response from RTX robot over the com port.
*******************************************************************************

procedure comRequest(Command: Byte; var Response: Byte); overload;
Var
WriteData,ReadData : ByteArray3;
ReadCount : Byte;
begin
WriteData[0] := Command;
comRequest(1,WriteData,ReadCount,ReadData);
Response := ReadData[0];
end;

*******************************************************************************
Initialise communication with the RTX robot and initializes robot.
*******************************************************************************

function RTX_Init(ComPort : String): Boolean;
Var
Retried : Boolean;
begin
Retried := true;
LogFile(1,'RTX_Init : Start');
result := comInit(ComPort);
if not result then exit;
repeat
  if not Retried then LogFile(0,'*********** RTX_Init : Retried ***********');
  // Make sure both IP's are active 
  RTX_IP_GoActive;
  if RTX_IP_Toggle(Once) = RSP_IP_RESTARTED then
    begin
      RTX_IP_GoActive;
      RTX_IP_Toggle(Once);
    end;
  if RTX_IP_Toggle(Once) = RSP_IP_RESTARTED then
    begin
      RTX_IP_GoActive;
      RTX_IP_Toggle(Once);
      end;
 LogFile(0,'RTX_Init : Restarting all IP''s');
  { Restart both IP's } 
  RTX_IP_Reset(Full);
  RTX_IP_GoActive;
  RTX_IP_Toggle(Once);
  RTX_IP_Reset(Full);
  RTX_IP_GoActive;
  RTX_IP_Toggle(Once);
LogFile(0,'RTX_Init : Make sure both IP''s are active');
  { Make sure both IP's are active }
result := true;
RTX_IP_Select(0);
if RTX_IP_Identification <> 0 then result := false;
RTX_IP_Select(1);
if RTX_IP_Identification <> 1 then result := false;
Retried := not Retried;
until Retried or Result;
LogFile(0,'RTX_Init : Set Absolute mode');
{ Set IP positioning mode and enable motors }
RTX_SetMode(PositionMode);
RTX_SetMode(AbsoluteMode);
RTX_Stop(FreeOff);
LogFile(-1,'RTX_Init : End');
end;

----------------------------------------------------------
Closes handle to com1
----------------------------------------------------------

function RTX_DeInitComm: Boolean;
begin
LogFile(1,'RTX_DeInit : Start');
comClose;
result := true;
LogFile(-1,'RTX_DeInit : End');
end;

*******************************************************************************
Closes handle to com1
*******************************************************************************

function RTX_IP_GoActive : byte;
begin
LogFile(0,'RTX_GoActive');
comRequest($00,Result);
CheckResult(result,[0,$20,$21],'GoActive Failed');
Sleep(10);
end;

*******************************************************************************
Activate the IP. Required after a IP restart.
*******************************************************************************

function RTX_IP_Identification : byte;
begin
LogFile(1,'RTX_IP_Identification : Start');
comRequest($01,Result);
CheckResult(result,[$20,$21],'IP ID Failed');
if (Result = $20) or (Result = $21) then
begin
Result := Result and $1;
Result := Result xor $1; // Wrong Specs, swap IP's around
end;
LogFile(-1,'RTX_IP_Identification : End '+IntToStr(Result));
end;

*******************************************************************************
Switch IP's
*******************************************************************************

function RTX_IP_Toggle(Switch : IPToggleMode) : byte;
begin
Case Switch of
  Auto : LogFile(0,'RTX_IP_Toggle : Auto');
  Once : LogFile(0,'RTX_IP_Toggle : Once');
   Off : LogFile(0,'RTX_IP_Toggle : Off');
end;
comRequest($28+Byte(Switch),Result);
CheckResult(result,[0],'IP Toggle failed.');
end;

*******************************************************************************
Select a specific IP
*******************************************************************************

function RTX_IP_Select(IP : Byte) : byte;
begin
LogFile(1,'RTX_IP_Select : IP '+IntToStr(IP));
if (RTX_IP_Identification <> IP) then
result := RTX_IP_Toggle(Once)
/* Reset a specific IP */

function RTX_IP_Reset(Reset : IPResetMode) : byte;
begin
LogFile(0,'RTX_IP_Reset : ResetSwitch '+IntToStr(Byte(Reset)));
comRequest($20+Byte(Reset),Result);
CheckResult(result,[0],'IP Reset failed');
Sleep(10);
end;

/* Soaking function. */

function RTX_SoakMode(Mode : RTXSoakMode): byte;
begin
LogFile(1,'RTX_Soak : Mode '+IntToStr(Byte(Mode)));
RTX_IP_Select(0);
comRequest($08+Byte(Mode),result);
RTX_IP_Select(1);
comRequest($08+Byte(Mode),result);
LogFile(-1,'RTX_Soak : End');
end;

/* Get Motor Status */

function RTX_GetMotorStatus(Motor : RTXMotor): byte;
Var
IP : Byte;
Controller : Byte;
ReadData : ByteArray3;
ReadCount : Byte;
begin
// LogFile(1,'RTX_GetMotorStatus : Motor '+IntToStr(Byte(Motor)));
Controller := Byte(Motor) mod 5;
if Controller = Byte(Motor) then IP := 0 else IP := 1;
RTX_IP_Select(IP);
comRequest($10+Controller,ReadCount,ReadData);
CheckResult(ReadData[0],[$10,$11,$12,$13,$14],'GetMotorStatus Failed');
if ReadCount <> 3 then
result := ReadData[0]
else
result := ReadData[1];
// LogFile(-1,'RTX_GetMotorStatus : End');
end;

/* Get General Status. */

function RTX_GetGeneralStatus(IP : Byte): byte;
Var
Data,ReadData : ByteArray3;
ReadCount : Byte;
begin
// LogFile(1,'RTX_GetGeneralStatus : IP '+IntToStr(IP));
RTX_IP_Select(IP);
Data[0] := $17;
comRequest(1,Data,ReadCount,ReadData);
CheckResult(ReadData[0],[$17],'GetGeneralStatus Failed');
if ReadCount <> 3 then
result := RSP_FAILED
else
result := ReadData[1];
// LogFile(-1,'RTX_GetGeneralStatus : End');
Appendix C: Robot Drivers

135

end;

(* Set operating mode specific motor
***********************************************

function RTX_SetMotorMode(Motor : RTXMotor; MotorMode : RTXMotorMode): boolean;
Var
  Mode : ByteArray3;
  res : ByteArray3;
  resCount : Byte;
begin
 LogFile(1,'RTX_SetMotorMode : Motor '+IntToStr(Byte(Motor))+' Mode '+IntToStr(Byte(Motor)));
  result:= true;
  Mode[0] := $18 + Ord(MotorMode);
  Mode[1] := 1 shl (Ord(Motor) mod 5);
  Mode[2] := 0;
  RTX_IP_Select(Ord(Motor) div 5);
  comRequest(3,Mode,resCount,res);
  CheckResult(res[0],[0],'SetMotorMode Failed');
  // b.Add(Format('Setting Mode IP%d Ctrl%d Res=%d', [Ord(Motor) div 5, Mode[1], res[0]]));
  if res[0] <> RSP_ACKNOWLEDGE then
    begin
      MessageBox(0,PChar(Format('FAILED : IP%d Ctrl%d Res=%d', [Ord(Motor) div 5, Mode[1], res[0]])),'',0);
      result := false;
    end;
  LogFile(-1,'RTX_SetMotorMode : End');
end;

(* Set operating mode of all motors
***********************************************

function RTX_SetMode(MotorMode : RTXMotorMode): boolean;
Var
  Mode : ByteArray3;
  res : ByteArray3;
  resCount : Byte;
begin
 LogFile(1,'RTX_SetMode : MotorMode '+IntToStr(Byte(MotorMode)));
  result:= true;
  Mode[0] := $18 + Ord(MotorMode);
  Mode[1] := $ff; // Select all controller an IP
  Mode[2] := 0;
  RTX_IP_Select(0);
  comRequest(3,Mode,resCount,res);
  if res[0] <> 0 then result := false;
  CheckResult(res[0],[10,11,12,13,14],'SetMode1 Failed');
  RTX_IP_Select(1);
  comRequest(3,Mode,resCount,res);
  if res[0] <> 0 then result := false;
  CheckResult(res[0],[10,11,12,13,14],'SetMode2 Failed');
  LogFile(-1,'RTX_SetMode : End');
end;

(* Check if any motors are still busy
***********************************************

function RTX_Busy : boolean;
begin
  result := ((RTX_GetGeneralStatus(0) or RTX_GetGeneralStatus(1)) and 1) = 1;
end;

(* Check if a specific set of motors are still busy
***********************************************

function RTX_Busy(Motors : RTXMotorSet) : boolean;
Var
  b : Byte;
  m : RTXMotor;
begin
  b:=0;
for m := Elbow to Sensor do
  if m in Motors then
    b := b or RTX_GetMotorStatus(m);
  result := (b and 1) = 1;
end;

procedure RTX_DefineHome;
begin
  LogFile(1,'RTX_DefineHome : Start');
  RTX_IP_Select(0);
  RTX_IP_Reset(DefineHome);
  RTX_IP_Select(1);
  RTX_IP_Reset(DefineHome);
  LogFile(-1,'RTX_DefineHome : End');
end;

procedure RTX_Stop(StopMode : RTXStopMode);
Var
  mode,res : Byte;
begin
  LogFile(1,'RTX_Stop : Start - Mode '+IntToStr(Byte(StopMode)));
  Mode := $24 + Byte(StopMode);
  RTX_IP_Select(0);
  comRequest(mode,res);
  CheckResult(res,[$10,$11,$12,$13,$14],'Stop1 Failed');
  RTX_IP_Select(1);
  comRequest(mode,res);
  CheckResult(res,[$10,$11,$12,$13,$14],'Stop2 Failed');
 LogFile(-1,'RTX_Stop : End');
end;

procedure RTX_MotorRead(Motor: RTXMotor; Parameter: RTXControlParameters;
var
  Value: SmallInt) : byte;
Var
  Controller,IP,ResponseCount : Byte;
  Command,Response : ByteArray3;
  Temp : SmallInt;
begin
  LogFile(1,'RTX_MotorRead : Start - Motor '+IntToStr(Byte(Motor)));
  if (Parameter = cp_ACTUAL_POSITION) then
    begin
      RTX_MotorRead(Motor,cp_CURRENT_POSITION,Value);
      RTX_MotorRead(Motor,cp_ERROR,Temp);
      Value := Value - Temp;
      result := RSP_ACKNOWLEDGE;
      exit;
    end;
  Controller := Byte(Motor) mod 5;
  IP := Byte(Motor) div 5;
  RTX_IP_Select(IP);
  // Immediate read
  if (Parameter = cp_ERROR) or (Parameter = cp_CURRENT_POSITION) then
    begin
      if (Parameter = cp_ERROR) then command[0] := $40
      else command[0] := $48;
      Command[0] := Command[0] + Controller;
      comRequest(1,Command,ResponseCount,Response);
      result := Response[0];
    end;
if (ResponseCount<3) or ((result and $F0)<>$80) then exit;

Value := Response[1] or (Response[2] shl 8);
end else
// Deferred read
begin
// First transaction, specify which parameter
Command[0] := $60;
Command[1] := Byte(Parameter);
comRequest(3,Command,ResponseCount,Response);
result := Response[0];
if result <> $C0 then exit;

// Seconds transaction, specify which controller to read from and get value
Command[0] := $68+Controller;
comRequest(1,Command,ResponseCount,Response);
result := Response[0];
Value := Response[1] or (Response[2] shl 8);
end;
LogFile(-1,'RTX_MotorRead : End');
end;

******************************************************************************
Write parameter
*******************************************************************************)

function RTX_MotorWrite(Motor: RTXMotor; Parameter: RTXControlParameters; Value: SmallInt) : byte;
Var
Controller,IP,ResponseCount : Byte;
Command,Response : ByteArray3;
c : Byte;
TimeOut : Byte;
begin
LogFile(1,Format('RTX_MotorWrite : Start - Motor %d Param %d Value %d',[Byte(Motor),Byte(Parameter),Value]));
if (Parameter = cp_ACTUAL_POSITION) then
begin
result := RSP_FAILED;
exit;
end;
Controller := Byte(Motor) mod 5;
IP := Byte(Motor) div 5;
RTX_IP_Select(IP);
// Immediate write
if (Parameter = cp_CURRENT_POSITION) then
begin
command[0] := $58 + Controller;
command[1] := Value and $ff;
command[2] := (Value shr 8) and $ff;
comRequest(3,Command,ResponseCount,Response);
result := Response[0];
end else
// Deferred write
begin
TimeOut := 0;
repeat
// First transaction, specify which parameter
Command[0] := $70;
Command[1] := Byte(Parameter);
comRequest(3,Command,ResponseCount,Response);
result := Response[0];
if result <> $E0 then exit;

// Seconds transaction, specify which controller to write to and set value
Command[0] := $78+Controller;
command[1] := Value and $ff;
command[2] := (Value shr 8) and $ff;
comRequest(3,Command,ResponseCount,Response);
result := Response[0];
Appendix C: Robot Drivers

```pascal
(Contd from previous page)

c := (Command[1] shr 4) xor (Command[1] and $f);
c := c xor (Command[2] shr 4) xor (Command[2] and $f);
inc(TimeOut)
until (c = (result and $f)) or (TimeOut > 5);
if TimeOut > 5 then
begin
LogFile(0, Format('TimeOUT : Motor %d Param %d NewValue %d', [Byte(Motor), Byte(Parameter), Value]));
raise Exception.Create('MotorWrite failed write checksum after 5 retries.');
end;
end;
LogFile(-1, 'RTX_MotorWrite : End');

// TODO: Remove
{ if (response[0] <> 0) and (Response[0] <> $20) and (Response[0] <> $21) then
begin
MessageBox(0, PChar(Format('Written %s (%d) Read %s (%d))', [IntToHex(Command[0], 2),
3, IntToHex(Response[0], ResponseCount), ResponseCount])),'',0);
end;}
end;

(* RTX GO Directive *)

procedure RTX_GoNumeric_All;
Var
res : Byte;
begin
LogFile(1, 'RTX_GoNumericAll : Start');
RTX_IP_Select(0);
comRequest($BF, res);
RTX_IP_Select(1);
comRequest($BF, res);
LogFile(-1, 'RTX_GoNumericAll : End');
end;

procedure RTX_GoNumeric(Motors : RTXMotorSet);
Var
res : Byte;
AA : array [0..1] of byte;
m : RTXMotor;
IP, v : Byte;
begin
LogFile(1, 'RTX_GoNumeric : Start');
AA[0] := $A0; AA[1] := $A0;
// Get Bitmask
for m := Elbow to Sensor do
begin
IP := Byte(m) div 5;
v := Byte(m) mod 5;
if m in Motors then
AA[IP] := AA[IP] or (1 shl v)
end;
// Check if IP0 has any motors to move, if so numeric go
if AA[0] <> $A0 then
begin
RTX_IP_Select(0);
comRequest(AA[0], res);
end;
// Check if IP1 has any motors to move, if so numeric go
if AA[1] <> $A0 then
begin
RTX_IP_Select(1);
comRequest(AA[1], res);
end;
LogFile(-1, 'RTX_GoNumeric : End');
end;
```
Appendix C: Robot Drivers

139

procedure RTX_MotorReadAll(cp_Parameter: RTXControlParameters; var Values: RTXValues);
Var Enc : RTXMotor;
beg
LogFile(1,'RTX_MotorReadAll : Start');
for Enc := Elbow to Wrist2 do
RTX_MotorRead(Enc,cp_Parameter,Values[Enc]);
LogFile(-1,'RTX_MotorReadAll : End');
end;

procedure RTX_MotorWriteAll(cp_Parameter: RTXControlParameters; Values: RTXValues);
Var Enc : RTXMotor;
beg
LogFile(1,'RTX_MotorWriteAll : Start');
for Enc := Elbow to Wrist2 do
RTX_MotorWrite(Enc,cp_Parameter,Values[Enc]);
LogFile(-1,'RTX_MotorWriteAll : End');
end;

procedure RTX_Joint_to_EncoderCounts(mmZed, degShoulder, degElbow, degYaw, degPitch, degRotate : Real; var EncoderCounts: RTXValues);
Var PitchCount, RotateCount : SmallInt;
begin
EncoderCounts[Base1] := 0;
EncoderCounts[Base2] := 0;
EncoderCounts[Zed] := Round(mmZed*RTXMotorInfo[Zed].Ratio);
EncoderCounts[Shoulder] := Round(degShoulder*RTXMotorInfo[Shoulder].Ratio);
EncoderCounts[Elbow] := Round(degElbow*RTXMotorInfo[Elbow].Ratio);
EncoderCounts[Yaw] := Round((degYaw+degElbow/2)*RTXMotorInfo[Yaw].Ratio);
PitchCount := Round(degPitch*RTXMotorInfo[Wrist1].Ratio);
RotateCount := Round(degRotate*RTXMotorInfo[Wrist2].Ratio);
EncoderCounts[Wrist1] := PitchCount+RotateCount; // w1+w2
EncoderCounts[Wrist2] := PitchCount-RotateCount; // w1-w2
LogFile(0,Format('************************ Deg : Pitch/Rotate : %1.2f/%1.2f', [degPitch,degRotate]));
LogFile(0,Format('************************ Enc : Pitch/Rotate : %d/%d', [PitchCount,RotateCount]));
end;

function RTX_EncoderAdd(A,B : RTXValues): RTXValues;
Var Enc : RTXMotor;
beg
end;

function RTX_EncoderSubtract(A,B : RTXValues): RTXValues;
Var Enc : RTXMotor;
beg
end;

function RTX_EncoderMultiply(A: RTXValues; B: Real): RTXValues;
Var Enc : RTXMotor;
beg
for Enc := Elbow to Wrist2 do result[enc] := Round(A[enc]*B);
end;

function RTX_EncoderDivide(A: RTXValues; B: Real): RTXValues;
Var Enc : RTXMotor;
beg
if B=0 then exit
end;
else
  for Enc := Elbow to Wrist2 do result[enc] := Round(A[enc]/B);
end;

(******************************************************************************
Move RTX robot to configuration specified. Speed is a speed between 0 and 100.
******************************************************************************)

procedure RTX_MoveMotor(Motor : RTXMotor; newPos : SmallInt);
begin
  LogFile(1,'RTX_MoveMotor : Start');
  RTX_MotorWrite(Motor,cp_NEW_POSITION,newPos);
  RTX_GoNumeric_All;
  LogFile(-1,'RTX_MoveMotor : End');
end;

(******************************************************************************
Move RTX robot with direct hardware movement
******************************************************************************)

procedure RTX_MoveInterP(steps:RTXInterpolateSteps);
Var
  Data : ByteArray3;
  Resp : ByteArray3;
  RespCount : Byte;
begin
  {0->Zed
   1->Shoulder
   2->Elbow
   3->Yaw
   4->Wrist1
   5->Wrist2}
  {MessageBox(0,PChar(Format('Zed=%d Shoulder=%d Elbow=%d Yaw=%d Wrist1=%d Wrist2=%d',
    [Steps[0],Steps[1],Steps[2],Steps[3],Steps[4],Steps[5]])),'',0);}
  LogFile(LogLevel,Format('*** Zed=%d Shoulder=%d Elbow=%d Yaw=%d Wrist1=%d Wrist2=%d',
   [Steps[0],Steps[1],Steps[2],Steps[3],Steps[4],Steps[5]]));
  if (Steps[0]<>0) or (Steps[1]<>0) or (Steps[2]<>0) or (Steps[3]<>0) then begin
    RTX_IP_Select(0);
    // Databit
    Data[0] := $c0;
    // Controller 4 - gripper
    // Data[0] := Data[0] or ( and $f)
    // Controller 1 & 0- shoulder and elbow
    Data[1] := (Steps[1] shl 4) or (Steps[2] and $f);
    // Controller 3 & 2- yaw and zed
    Data[2] := (Steps[3] shl 4) or (Steps[0] and $f);
    comRequest(3,Data,Respcount,Resp);
  end;

  if (Steps[4]<>0) or (Steps[5]<>0) then begin
    RTX_IP_Select(1);
    // Databit
    Data[0] := $c0;
    // Controller 4 - Sonic
    // Data[0] := Data[0] or ( and $f)
    // Controller 1 & 0- Base1 and Base2
    Data[1] := 0;
    // Controller 3 & 2- Wrist2 and Wrist1
    Data[2] := (Steps[5] shl 4) or (Steps[4] and $f);
    comRequest(3,Data,Respcount,Resp);
  end;
  LogFile(LogLevel,'*** MoveInterP done');
end;

(******************************************************************************
Move RTX robot to configuration specified. Speed is a speed between 0 and 100.
******************************************************************************)

procedure RTX_MoveAbsolute(mmZed,DegShoulder,degElbow,degYaw,degPitch,degRotate :Real; Speed : Real; var Eta : Real);
function GetMaxTime(Difference : RTXValues):RTXMotor;
begin
  Result := Shoulder;
  if abs(Difference[Elbow]) > abs(Difference[result]) then result := Elbow;
  if abs(Difference[Yaw]) > abs(Difference[result]) then result := Yaw;
  if abs(Difference[Wrist1]) > abs(Difference[result]) then result := Wrist1;
  if abs(Difference[Wrist2]) > abs(Difference[result]) then result := Wrist2;
  // Zed works at about a 1/3 the speed of the rest of the motors. It can however
  // increase its speed to 160%, therefor a factor of only about 2.
  if abs(Difference[Zed]*ZED_SPEED_MUL) > abs(Difference[result]) then result := Zed; // only
  // 100% speed
  // if abs(Difference[Zed]*2) > abs(Difference[result]) then result := Zed; // 160% speed
end;

Var
  CurEnc : RTXValues;
  FinEnc : RTXValues;
  DifEnc : RTXValues;
  M : RTXMotor;
  ZSpeed : SmallInt;
SpdDif : Real;

begin
  LogFile(1,'RTX_MoveAbsolute : Start '+Format('%1.1f, %1.1f, %1.1f, %1.1f, %1.1f, %1.1f Speed

%1.1f]',[mmZed,DegShoulder,degElbow,degPitch,degRotate,Speed]));
  // Get current position of arm (slow, but accurate)
  RTX_MotorRead(Zed, cp_CURRENT_POSITION,CurEnc[Zed]);
  RTX_MotorRead(Shoulder,cp_CURRENT_POSITION,CurEnc[Shoulder]);
  RTX_MotorRead(Elbow, cp_CURRENT_POSITION,CurEnc[Elbow]);
  RTX_MotorRead(Yaw, cp_CURRENT_POSITION,CurEnc[Yaw]);
  RTX_MotorRead(Wrist1, cp_CURRENT_POSITION,CurEnc[Wrist1]);
  RTX_MotorRead(Wrist2, cp_CURRENT_POSITION,CurEnc[Wrist2]);
  // Calculate final position of arm
  RTX_Joint_to_EncoderCounts(mmZed,degShoulder,degElbow,degPitch,degRotate,FinEnc);
  // Get difference, and which motor has the biggest difference
  DifEnc := RTX_EncoderSubtract(FinEnc,CurEnc);
  // Determine which motor will take the longest for scaling
  M := GetMaxTime(DifEnc);
  ETA := 0;
  if DifEnc[M] = 0 then exit; // No movement, quit
  // Set speed
  SpdDif := difEnc[M];
  if M = Zed then SpdDif := SpdDif * ZED_SPEED_MUL;
  SpdDif := Abs(Round(Speed*difEnc[Zed]/SpdDif*ZED_SPEED_MUL));
  if SpdDif>100 then SpdDif := 100;
  LogFile(0,Format('*********** Requested speed %1.1f',[Speed]));
  LogFile(0,Format('*********** Diff (Max Dif = %d)',[Byte(M)]));
  LogFile(0,Format('***********    Zed %d',[difEnc[Zed]]));
  LogFile(0,Format('***********    Shoulder %d',[difEnc[Shoulder]]));
  LogFile(0,Format('***********    Elbow %d',[difEnc[Elbow]]));
  LogFile(0,Format('***********    Yaw %d',[difEnc[Yaw]]));
  LogFile(0,Format('***********    Wrist1 %d',[difEnc[Wrist1]]));
  LogFile(0,Format('***********    Wrist2 %d',[difEnc[Wrist2]]));
  LogFile(0,Format('*********** Speed (Max Dif = %d)',[Byte(M)]));
  LogFile(0,Format('***********    Zed %d',ZSpeed));
  LogFile(0,Format('***********    Shoulder %d',Abs(Round(Speed*difEnc[Shoulder]/SpdDif))));
  LogFile(0,Format('***********    Elbow %d',Abs(Round(Speed*difEnc[Elbow]/SpdDif))));
  LogFile(0,Format('***********    Yaw %d',Abs(Round(Speed*difEnc[Yaw]/SpdDif))));
  LogFile(0,Format('***********    Wrist1 %d',Abs(Round(Speed*difEnc[Wrist1]/SpdDif))));
  LogFile(0,Format('***********    Wrist2 %d',Abs(Round(Speed*difEnc[Wrist2]/SpdDif))));
  // Slow, but required
  RTX_MotorWrite(Zed, cp_SPEED,ZSpeed);
  RTX_MotorWrite(Shoulder,cp_SPEED,Abs(Round(Speed*difEnc[Shoulder]/SpdDif)));
  RTX_MotorWrite(Elbow, cp_SPEED,Abs(Round(Speed*difEnc[Elbow]/SpdDif)));
  RTX_MotorWrite(Yaw, cp_SPEED,Abs(Round(Speed*difEnc[Yaw]/SpdDif)));
  RTX_MotorWrite(Wrist1, cp_SPEED,Abs(Round(Speed*difEnc[Wrist1]/SpdDif)));
  RTX_MotorWrite(Wrist2, cp_SPEED,Abs(Round(Speed*difEnc[Wrist2]/SpdDif)));
  // Calculate ETA for movement to finish in seconds
  ETA := Abs(DifEnc[M]/RTXMotorInfo[M].Speed*100/Speed)*1.1;
  if M = Zed then ETA := ETA / ZED_SPEED_MUL;
Appendix C: Robot Drivers

**RTX_Stop(FreeOff);**
// Get new positions
// Slow, but required
RTX_MotorWrite(Zed, cp_NEW_POSITION, FinEnc[Zed]);
RTX_MotorWrite(Shoulder, cp_NEW_POSITION, FinEnc[Shoulder]);
RTX_MotorWrite(Elbow, cp_NEW_POSITION, FinEnc[Elbow]);
RTX_MotorWrite(Yaw, cp_NEW_POSITION, FinEnc[Yaw]);
RTX_MotorWrite(Wrist1, cp_NEW_POSITION, FinEnc[Wrist1]);
RTX_MotorWrite(Wrist2, cp_NEW_POSITION, FinEnc[Wrist2]);

// Move arm

MessageBox(0, PChar(Format('%d,%d,%d,%d,%d,%d', [FinEnc[Zed], FinEnc[Shoulder], FinEnc[Elbow], FinEnc[Yaw], FinEnc[Wrist1], FinEnc[Wrist2]])), '', 0);
RTX_GoNumeric([Zed, Shoulder, Elbow, Yaw, Wrist1, Wrist2]);
LogFile(-1, 'RTX_MoveAbsolute : End');

```pascal
end;
```

*******************************************************************************
**Move robot gripper to new absolute position**
*******************************************************************************

```pascal
procedure RTX_MoveAbsoluteGripper(Gap: Real; var Eta: Real);
Var
CurEnc : RTXValues;
FinEnc : RTXValues;
DifEnc : SmallInt;
M : RTXMotor;
ZSpeed : SmallInt;
SpdDif : Real;
CurGripEnc, NewGripEnc : SmallInt;
GripEnc : SmallInt;
begin
// Get current gripper position
RTX_MotorRead(Gripper, cp_CURRENT_POSITION, GripEnc);

// Calculate encoder difference
NewGripEnc := Round((-0.0584 + sqrt(sqr(0.0584) + 0.0000428 * Gap)) / 0.0000214);
//MessageBox(0, PChar(Format('Gap=%1.2f Current=%d New=%d', [Gap, GripEnc, NewGripEnc]))), '', 0);
DifEnc := NewGripEnc - GripEnc;
if DifEnc = 0 then exit;

// Calculate ETA for movement to finish in seconds
ETA := Abs(DifEnc / (6250 / 60 * 12));
RTX_Stop(FreeOff);
// Set new positions
RTX_MotorWrite(Gripper, cp_NEW_POSITION, NewGripEnc);

// Move gripper
//MessageBox(0, PChar(Format('Enc %d->%d (%d)
ETA=%1.2f', [GripEnc, NewGripEnc, DifEnc, ETA])), '', 0);
RTX_GoNumeric([Gripper]);
end;
```

*******************************************************************************
**Read position in metric form from RTX**
*******************************************************************************

```pascal
procedure RTX_GetJointPositions(var mmZed, degShoulder, degElbow, degYaw, degPitch, degRotate, mmGripper : Real);
Var
v: Array[0..6] of SmallInt;
begin
LogFile(1, 'RTX_GetJointPositions : Start');
RTX_MotorRead(Zed, cp_CURRENT_POSITION, v[0]);
RTX_MotorRead(Shoulder, cp_CURRENT_POSITION, v[1]);
RTX_MotorRead(Elbow, cp_CURRENT_POSITION, v[2]);
RTX_MotorRead(Yaw, cp_CURRENT_POSITION, v[3]);
RTX_MotorRead(Wrist1, cp_CURRENT_POSITION, v[4]);
RTX_MotorRead(Wrist2, cp_CURRENT_POSITION, v[5]);
RTX_MotorRead(Gripper, cp_CURRENT_POSITION, v[6]);
mmZed := v[0] / RTXMotorInfo[Zed].Ratio;
```
mmGripper := 0.0584*v[6]+0.0000107*v[6]*v[6];
LogFile(-1,'RTX_GetJointPositions : End');
end
*******************************************************************************
Move RTX robot to configuration specified. Speed is a speed between 0 and 100.
*******************************************************************************
procedure RTX_Calibrate;
Var
i : Smallint;
begin
LogFile(1,'RTX_Calibrate : Start');
// Prepare for calibration
RTX_MotorWrite(Zed, cp_SPEED,80);
RTX_MotorWrite(Shoulder, cp_SPEED,80);
RTX_MotorWrite(Elbow, cp_SPEED,80);
RTX_MotorWrite(Yaw, cp_SPEED,80);
RTX_MotorWrite(Wrist1, cp_SPEED,80);
RTX_MotorWrite(Wrist2, cp_SPEED,80);
RTX_MotorWrite(Shoulder, cp_MAX_FORCE,64);
RTX_MotorWrite(Elbow, cp_MAX_FORCE,64);
RTX_MotorWrite(Yaw, cp_MAX_FORCE,64);
RTX_MotorWrite(Wrist1, cp_MAX_FORCE,99);
RTX_MotorWrite(Wrist2, cp_MAX_FORCE,20);
RTX_DefineHome;
RTX_Stop(FreeOff);
RTX_SetMode(AbsoluteMode);
// Zed
LogFile(0,'RTX_Calibrate : Zed');
RTX_MotorWrite(Zed,cp_NEW_POSITION,4000);
RTX_GoNumeric([Zed]);
while ((RTX_GetMotorStatus(Zed) and 1) = 1) do;
RTX_DefineHome;
RTX_Stop(FreeOff);
RTX_Stop(FreeOff);
// Pitch and Rotate
LogFile(0,'RTX_Calibrate : Wrist');
RTX_MotorWrite(Wrist1,cp_NEW_POSITION,-6000);
RTX_MotorWrite(Wrist2,cp_NEW_POSITION,6000);
RTX_GoNumeric([Wrist1,Wrist2]);
while ((RTX_GetMotorStatus(Wrist1) and 1) = 1) or
((RTX_GetMotorStatus(Wrist2) and 1) = 1) do;
RTX_MotorWrite(Wrist1,cp_NEW_POSITION,-4000);
RTX_MotorWrite(Wrist2,cp_NEW_POSITION,4000);
RTX_MotorWrite(Wrist2, cp.NEW.POSITION,0);
RTX_GoNumeric([Wrist1,Wrist2,Gripper]);
while (RTX_GetMotorStatus(Wrist1) and 1 and 1) or
(RTX_GetMotorStatus(Wrist2) and 1) = 1 do;
RTX_MotorWrite(Yaw,cp_NEW_POSITION,0);
RTX_GoNumeric([Yaw]);
LogFile(0,'RTX_Calibrate : Elbow Shoulder Yaw!');
RTX_MotorWrite(Elbow,cp_NEW_POSITION,6000);
RTX_MotorWrite(Shoulder,cp_NEW_POSITION,5000);
RTX_MotorWrite(Yaw,cp_NEW_POSITION,-6000);
RTX_GoNumeric([Elbow,Shoulder,Yaw]);
while RTX_Busy([Yaw,Elbow,Shoulder]) do;
RTX_MotorWrite(Yaw,cp_NEW_POSITION,-8000);
RTX_GoNumeric([Yaw]);
While RTX_Busy([Yaw]) do;
  Sleep(1000);
{
  RTX_MotorWrite(Shoulder,cp_CURRENT_POSITION,2630);// 3101 right value, compensation of 40
  RTX_MotorWrite(Elbow,cp_CURRENT_POSITION,-2630);
  RTX_MotorWrite(Yaw,cp_CURRENT_POSITION,-1947);
  RTX_MotorWrite(Shoulder,cp_ERROR,0);
  RTX_MotorWrite(Elbow,cp_ERROR,0);
  RTX_MotorWrite(Yaw,cp_ERROR,0);
}
(* Check if coordinates written correctly *)
{
  RTX_MotorRead(Wrist1,cp_ACTUAL_POSITION,i); MessageBox(0,PChar('Wrist1 : -3101 : '+IntToStr(i)),'
  RTX_MotorRead(Wrist2,cp_ACTUAL_POSITION,i); MessageBox(0,PChar('Wrist2 : 528 :
  // Return to normal
LogFile(0,'RTX_Calibrate : Go Home');
  RTX_DefineHome;
  RTX_MotorWrite(Shoulder,cp_NEW_POSITION,-2630-100);
  RTX_MotorWrite(Elbow,cp_NEW_POSITION,2630+50);
  RTX_MotorWrite(Yaw,cp_NEW_POSITION,1947);
  RTX_MotorWrite(Wrist1,cp_NEW_POSITION,3101-40);
  RTX_MotorWrite(Wrist2,cp_NEW_POSITION,-528+40);
  RTX_MotorWrite(Gripper,cp_NEW_POSITION,10);
  RTX_GoNumeric([Shoulder,Elbow,Yaw,Wrist1,Wrist2,Gripper]);
  While RTX_Busy do;
  Sleep(1000);
  RTX_DefineHome;
  RTX_Stop(FreeOff);
LogFile(-1,'RTX_Calibrate : End');
end;
}