NTNU
Norwegian University of
Science and Technology

Setup and Interfacing of a KUKA Robotics Lab

Ivar Eriksen

Master of Science in Cybernetics and Robotics

Submission date:	December 2017
Supervisor:	Jan Tommy Gravdahl, ITK
Co-supervisor:	Mathias Hauan Arbo, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for teknisk kybernetikk

MSc thesis assignment

Name of the candidate:	Ivar Eriksen
Subject:	Engineering Cybernetics
Title:	Setup and interfacing of a KUKA robotics lab

Background

The department of Engineering Cybernetics at NTNU has recently acquired a former SINTEF robotics laboratory. The laboratory consists of two KUKA KR-16 robots, one mounted on the floor, and one mounted on the end of a GÜDEL gantry crane. Setup of the laboratory is challenging in both the dated software and unique setup of the hardware. The old interface had a significant communication delay, and it was difficult to establish an overview of the system. This project is an effort to revitalize the laboratory for modern robotics research. The project includes surveying the available hardware, outlining the possible choices of middleware for communication and control, and creating or extending drivers for use with the unique robot setup. Two control interfaces are of particular interest (KUKAVARPROXY and KUKA.RSI) with regards to possibility of using the laboratory for both research and laboratory assignments for control engineering students.

Key topics of the projects are:

- Survey the available hardware and summarize the information
- Assess safety and security risks in the laboratory
- Evaluate the feasibility of KUKAVARPROXY as a control interface
- Investigate the use of workspace monitoring when external axes (gantry crane) are controlled via KUKA.RSI
- Create necessary files and configurations for using the force-torque sensor (Schunk FTC-50-80) and toolchange system (Schunk SWS-020) with ROS

To be handed in by: 18/12-2017

Jan Tommy Gravdahl

Jan Tommy Gravdahl
Professor, supervisor

Abstract

To evaluate the feasibility of controlling KUKA robots via KUKAVARPROXY, the performance of two libraries for communicating with KUKAVARPROXY is investigated. It is shown that the BoostCrossCom C++ library and the jOpenShowVar Java library have comparable performance on a KUKA KRC 2. Both libraries have a mean access time of 3ms for single connections, increasing too 12ms for 5 simultaneous connections.

To interface with a gantry mounted KR 16 the existing ROS Industrial driver is expanded to control mathematically linked external axes. The raw control that RSI allows can be a safety hazard for novice users, therefore a ROS driver using BoostCrossCom is implemented.

ROS interfaces for reading data from a Schunk FTC-50-80 force-torque sensor is developed using both RSI and BoostCrossCom. In order to control a Schunk SWS-020 tool change system from ROS, a service node using BoostCrossCom is created.

Sammendrag

For å evaluere muligheten for å styre KUKA roboter via KUKAVARPROXY, er ytelsen til to biblioteker for kommunikasjon med KUKAVARPROXY blitt undersøkt. Det er vist at BoostCrossCom C ++ biblioteket og jOpenShowVar Java-biblioteket har sammenlignbar ytelse på en KUKA KRC 2. Begge biblioteker har en gjennomsnittlig tilgangstid på 3ms for enkeltforbindelser, og øker til 12ms for 5 samtidige tilkoblinger.

For å styre KR 16 montert på en traveskran, er den eksisterende ROS Industrial-driveren blitt utvidet til å kontrollere matematisk koblede eksterne akser. RSI kan være en sikkerhetsfare for nybegynnere, derfor er en ROS-driver som bruker BoostCrossCom implementert.

Et ROS-grensesnitt for lesing av data fra en Schunk FTC-50-80 kraftmoment sensor er utviklet ved hjelp av både RSI og BoostCrossCom. For å styre et Schunk SWS-020 verktøybyttesystem fra ROS, er det skrevet en service node som kommuniserer via BoostCrossCom.

Preface

The technical work for this thesis has been independently executed, including information gathering, configuration and software design. This has been an exciting project to work on, and I have enjoyed developing the laboratory and trying out my ideas on real hardware. During the time spent working on this thesis the Department of Engineering Cybernetics (ITK) have provided full access to the robotics laboratory, including 3D models and basic hardware documentation that followed the laboratory from SINTEF.

I would like to thank both my supervisor Tommy Gravdahl and my co-supervisor Mathias Hauan Arbo for their support and guidance throughout an eventful year.

I would also like to thank Filippo Sanfilippo for introducing me to jOpenShow-Var and KUKAVARPROXY, on which much of this work is based.

Contents

List of Tables

List of Figures

Abbreviations

BFL Bayesian Filtering Library. 35

DDS Data-Distribution Service for Real-Time Systems. 32

DOF Degrees-of-freedom. 2, 66, 67, 69, 70

DSE Digital Servo Electronics. 5, 7, 8

E-STOP emergency stop. 4, 5, 15, 42, 45, 47

ESC Electronic Safety Circuit. 9

FTP Focused Technical Project. 33

GUI graphical user interface. 14, 15, 28, 37

IPOC Interpolation cycle. 14, 22, 49, 51, 55, 56, 59, 60, 63–65, 69

ITK Department of Engineering Cybernetics. 1

KCP KUKA Control Panel. 5, 9, 45

KDL Kinematics and Dynamics Library. 35, 36

KRC KUKA Robot Controller. 3–10, 13–15, 20, 22, 23, 25, 26, 38, 41–43, 46, 47, 49–51, 54, 55, 59, 61, 64, 65, 69

KRL KUKA Robot Language. 15, 16, 18, 21–23, 37–39, 41, 55, 61, 63–66

KSD KUKA Servo Drive. 5, 7

KSS KUKA System Software. 3, 7, 13, 14, 21, 38, 46, 49, 54, 57

KUKA.HMI KUKA Human-Machine Interface. 14

KVGA KUKA VGA card. 4, 5

KVP KUKAVARPROXY. 2, 23, 38, 41, 47, 50, 51, 53–57, 59–61, 64–66, 69

LTS Long Term Support. 45

MFC3 multi-function card. 4, 5, 7, 13

NMI non-maskable interrupt. 13

ocl Orocos Component Library. 36

ODE Open Dynamics Engine. 34

OpenRAVE Open Robotics and Animation Virtual Environment. 37

Orocos Open RObot Control Software. 35, 36

osd Orocos State machine Description. 36

PLC Programmable Logic Controller. 15, 23, 24

PTP pose-to-pose. 17

RDC Resolver Digital Converter. 5, 7, 8, 42

RIC ROS Industrial Consortium. 33

ROS Robot Operating System. 2, 25, 26, 28, 32–37, 44, 48–51, 53, 54, 63–67, 69

ROS-I ROS Industrial. 33, 37, 38, 66, 67

RSI Robot Sensor Interface. 2, 21–23, 41, 47, 49–51, 54, 63–65, 69

RtAcc Real Time Accelerator. 13

RTD round-trip delay time. 54, 55, 57, 59, 62, 63

RTT Real-Time Toolkit. 35, 36

SPS Submit interpreter. 15, 55, 59, 65, 66

TCP Tool Center Point. 16, 19–21, 46, 49, 50, 63, 65, 66, 69

URDF Unified Robot Description Format. 28, 34, 35, 69, 70

YARP Yet Another Robot Platform. 36

Chapter 1

Introduction

In Isaac Asimov's The Caves of Steel, R. Daneel once said *"Aimless extension of knowledge, however, which is what I think you really mean by the term curiosity, is merely inefficiency"*. The lens with which curiosity is focused into research is laboratories. To this purpose the Department of Engineering Cybernetics (ITK) acquired a previously used robotics laboratory during the summer of 2016.

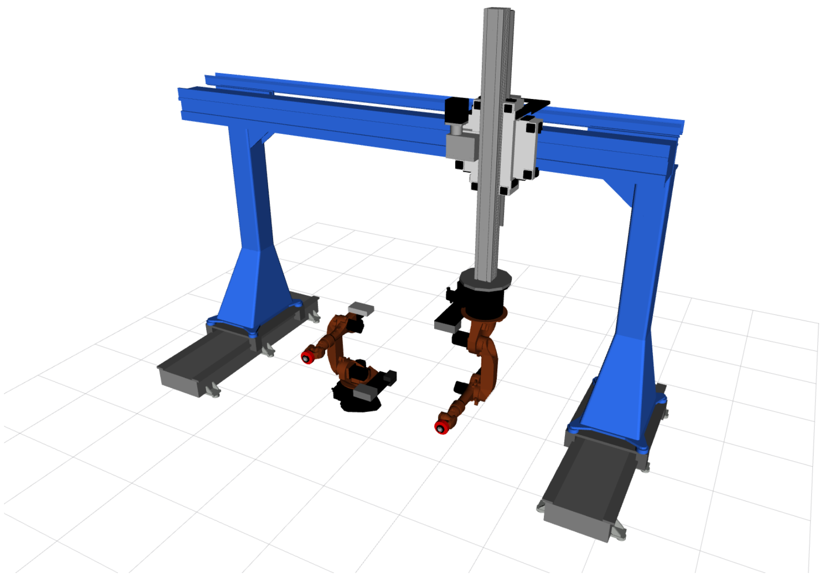

Figure 1.1: Screenshot of robot cell from rviz.

SINTEF and Hydro originally commissioned the laboratory in 2005 for research into unmanned inspection and maintenance of offshore oil platforms[1]. Some of the research conducted in the laboratory includes model based collision avoidance, teleoperation and active camera control[2], [3].

The laboratory consists of two KUKA KR 16 6 Degrees-of-freedom (DOF) robot manipulators along with a gantry crane from Güdel. One of the KR 16's is mounted on the gantry crane, giving it 9 DOF. Figure 1.1 shows a 3D model of the laboratory. Future plans for the laboratory include research into large scale additive manufacturing, and as a lab task for an introductory course during first semester in the Cybernetics and Robotics study program.

This thesis is a continuation of the work started in [4], and starts with an in depth introduction to a KUKA robot system in Chapter 2. Chapter 3 introduces middleware for use in robotics. The laboratory setup, along with safety and security concerns is outlined in Chapter 4. Timing experiments for KUKAVARPROXY (KVP) and expansion of the Robot Operating System (ROS) interfaces are explained in Chapter 5. Key contributions presented in this thesis include

- Access time and movement latency measurements for KVP
- FTC-50-80 integration with ROS
- SWS-020 integration with ROS
- Use of mathematically coupled external axes with ROS using Robot Sensor Interface (RSI)

Chapter 2

KUKA robot system

KUKA is one of the biggest robot manufacturing companies in the world[5]. The company was founded in 1898, and soon expanded into welding and welding solutions. After the rise of the Unimate, the first industrial robot, KUKA saw an opportunity in industrial manipulators. And by 1971 they delivered Europe's first welding transfer line with robots to Volkswagen AG. In 1973 KUKA made the first robot with six electromechanically driven axes. In 1996 they released the KUKA Robot Controller (KRC) 1 controller, which was the first PC based controller running both Windows and a real-time OS[6].

A typical robot system is made up from a robot or manipulator, a robot controller, software and some way to interface with the system. In this chapter we will give a description of the underlying system and their importance for safety and usability.

Several versions of the KRC and KUKA System Software (KSS) exist, and there have been some major changes between versions. Unless specified, all information regarding KRC and KSS in this text refers to version 5.4 and KRC 2 2005 edition for use with standard robots. There might be small changes in KRC and KSS for use with heavy-duty, palletizer or press-to-press robots.

2.1 KUKA Robot Controller

Most robot controllers consist of the same basic components: power supplies, servo controllers, a logic unit, and a human-machine interface of some sort. Figure 2.1 outlines the placement of these components in the KRC 2. This section is based on information found in [7].

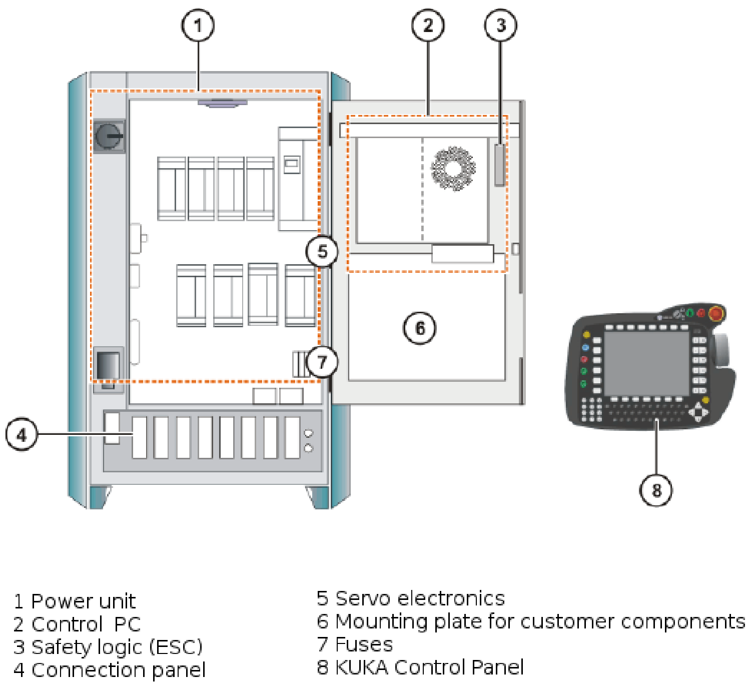

Figure 2.1: KRC 2 layout. Adapted from [7, Fig 2-2]. Courtesy of KUKA.

2.1.1 Control computer

The logic unit in the KRC is a standard x86 computer with a single core Pentium CPU. It has a few specialized PCI expansion cards for connecting to the servo drivers, field buses, analog and digital I/O as well as the emergency stop (E-STOP) system. The KUKA VGA card (KVGA) and multi-function card (MFC3) is the only non-optional expansion cards. The motherboard has several USB ports, RS232 serial ports, a parallel port, and one Ethernet port. The USB ports and Ethernet port is reserved for use with Windows.

The KVGA has a proprietary connection to the KUKA Control Panel (KCP), and offers a standard VGA plug for an extra monitor. This monitor will show the same image as is on the KCP display. MFC3 handles most of the communication between the computer and robot systems. It connects the servo controllers, E-STOP controller and DeviceNet bus.

2.1.2 KUKA Control Panel

KUKA Control Panel (KCP) is the physical part of the human-machine interface. This is a handheld programming device, commonly known as a teach pendant. Traditionally, all programming and manual control of the robot is done with the KCP. It is fitted with an 8", 640x480 px display. Along with a keypad and numpad there are several soft-keys, whose function changes depending on program context. Input from the KCP is transmitted over CAN bus. It is connected to the KRC via a 10-60 meter cable.

2.1.3 Power supply

The KRC is powered by 3x400VAC, with a rated power input of 7.3kVA. The KPS-600 is the main power supply, and is responsible for distributing both AC and DC power to the rest of the robot system. Both the main contactor, which supplies drive current to the KUKA Servo Drive (KSD), and the brake release contactor is located on the KPS 600. In addition the KPS-27 transforms 230VAC to 24VDC for use in the control electronics. Figure 2.2 outlines the placement of power modules in the KRC 2.

2.1.4 Servo control

Three main components are involved in servo control on the KRC: Resolver Digital Converter (RDC), Digital Servo Electronics (DSE) and KUKA Servo Drive (KSD).

Resolver Digital Converter (RDC)
Responsible for A/D-conversion and serialization of signals from servo encoders. The RDC is a 8 channel card, and can be used for up to two extra motors. It is placed on the robot base, and has a connection to the DSE.

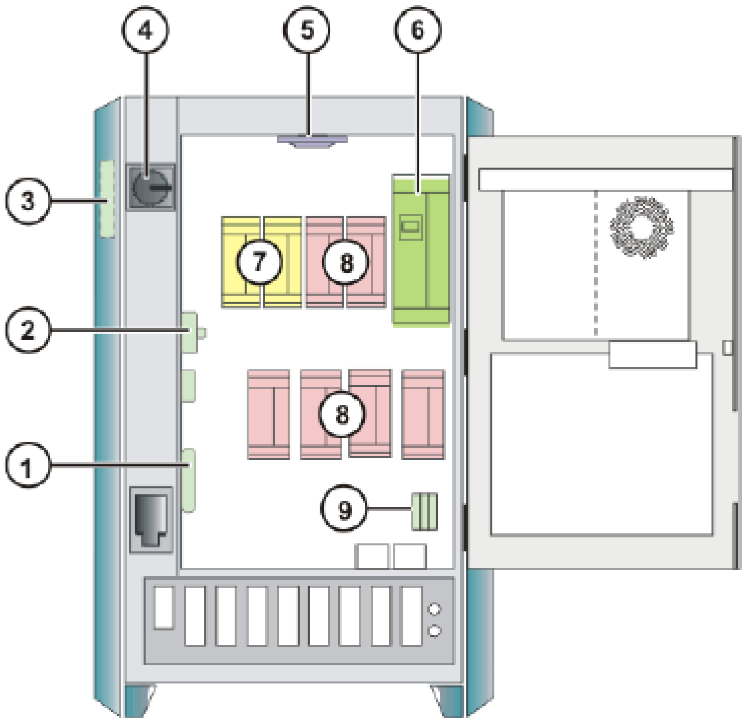

Fig. 2-20: Power unit

1	Low-voltage power supply KPS-27
2	Fuse elements (24 V without battery back-up)
3	Mains filter
4	Main switch (EU version)
5	Fan for inner cooling circuit
6	Power supply unit KPS600
7	KSDs for 2 external axes (option)
8	KSDs for 6 robot axes
9	Fuse elements (24 V with battery back-up)

Figure 2.2: Power module placement in the KRC 2. Courtesy of KUKA[7, Fig 2-20].

Digital Servo Electronics (DSE)

The brain in the servo control chain. In addition to monitoring the servo control chain it moves the servo to set points given from KSS. It is placed directly on the MFC3. In addition to the RDC connection, there is an Interbus connection from the DSE to the KSD.

KUKA Servo Drive (KSD)

Regulates and monitors servo drive current. Comes in different sizes depending on servo (robot) size. The KRC is able to operate servo motors in a master-slave configuration. This allows it to use several servos for actuating the same axis. For example, on a gantry placed on two rails, one could place one servo at each rail to ensure even traction. The standard cabinet fits a total of 8 KSD.

The RDC gives feedback on joint movement, but does not provide an absolute position of the joint. For the KRC to know which position a joint is in, a process known as mastering must be performed. During mastering, every axis is moved into a predefined position before mastering information is saved. The KRC can then use relative movement from this position to determine absolute joint position. Mastering is usually only performed during installation and after service.

A top-mounted extension cabinet is used if more than two extra servos is used. This contains extra power supplies and KSDs for controlling the external axes. One extra DSE along with an extra RDC card is required when using more than 8 servos.

2.1.5 Braking and deceleration

The robot servos are fitted with brakes. The brakes are operated by 24V DC, and needs power to release. As all brakes are controlled by the same relay in the KPS 600, they are released and engaged at the same time. To minimize response time, brakes are not used to keep the robot stationary during normal operation. If the robot is stationary for a prolonged time, or no program is running, the brakes will be applied and servo drive circuit will be disengaged.

The KRC has four different modes for stopping a robot's movement.

Ramp-down braking
Equivalent to EN 60204-1[1] STOP 2. This is used for deceleration during normal operation. Will keep the planned path during braking.

Path-maintaining braking
Equivalent to EN 60204-1 STOP 1. This uses servos to decelerate the robot. Once the robot is stationary, or after a maximum allowed time of 1 second, it applies the brakes and cuts power to the servos. Will keep the planed path during braking.

Path-oriented braking
Equivalent to EN 60204-1 STOP 0. This immediately cuts power to the servos and applies the brakes. It will attempt to follow the planned path using what remains of available energy in the drive circuit.

Short-circuit braking
Only used during controller shutdown, power failure or connection loss between the RDC and DSE. The brakes are immediately applied and power to the servos are cut.

To handle energy generated from back EMF, a ballast resistor is installed in the KRC. Energy from back EMF is dumped into this resistor. To avoid damage to the servos, temperature of the ballast resistor, brakes and servo is continuously monitored. Should any of these overheat, the controller will stop program execution and enter a fault state.

2.1.6 Backup batteries

The KRC 2 is fitted with backup batteries. During power loss, or if the KRC is turned off, these batteries provide power to the control computer. This enables the controller to save the current state to disk, before performing a graceful shutdown. The batteries are continually charged while the controller is powered on. If the

[1]Safety of machinery - Electrical equipment of machines - Part 1: General requirements

controller remains powered off for long periods, or the controllers are frequently turned on and off, the batteries might deplete or lose the ability to retain charge. If the KRC is turned off or loses power without working battery backup, mastering information will be lost.

2.1.7 Safety

The safety system consist of external and internal safeguards. The external safeguards are installation specific, like fences, gates, pressure mats, etc. Internal safeguards consists of axis monitoring, workspace monitoring, operating modes, etc. Electronic Safety Circuit (ESC) is responsible for monitoring all safety related hardware. The KRC 2 safety electronics are made in accordance with the now withdrawn EN 954-1 Category 3. This guarantees a single fault in the safety system will not interfere the function of said system. The KRC has four operating modes, different safety features are available depending on the active mode. See table 2.1 for the different modes. The KCP is outfitted with a local emergency stop. In addition to this, there is an input for external emergency stop.

External safeguards like safety gates, safety mats and laser fences are connected to the `Operator safety` input. In automatic mode, triggering this input will stop robot movement. Its intended purpose is to protect an operator entering the cell during normal operation, for example at a loading station. While there are both mechanical end stops and software limit switches, these are only intended as machine protection and should not be relied upon for operator safety. To move the robot in T1 or T2, one of the enable switches on the KCP must be pressed down.

Safety feature	T1	T2	AUT	AUT EXT
Emergency stop	STOP 0	STOP 0	STOP 1	STOP 1
Enabling switch	Active	Active	Not active	Not active
Operator safety	Not active	Not active	Active	Active
Max 250mm/s	Active	Not active	Not active	Not active
Jog mode	Active	Active	Not active	Not active

Table 2.1: Safety features dependent on operating mode[7]

2.2 KR 16

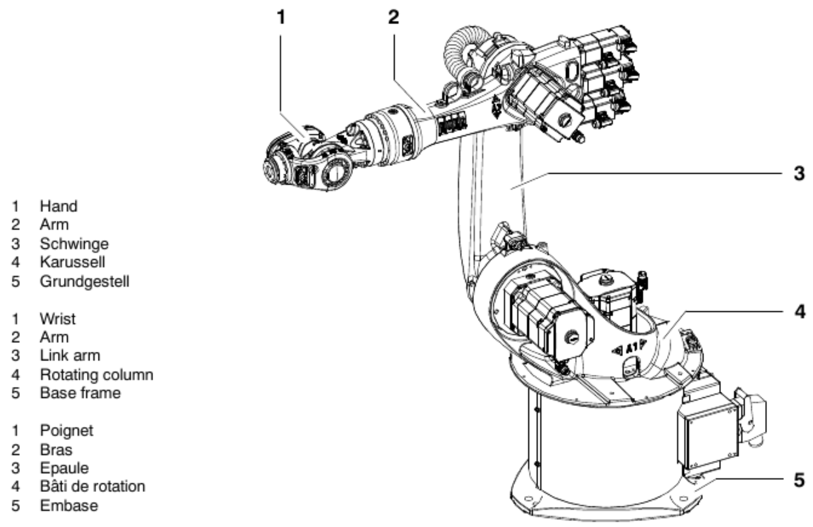

Figure 2.3: KUKA KR 16. Courtesy of KUKA[8, Fig 1-1].

The KUKA KR 16 is a 6 axis industrial robot in KUKA's "low payload" category. The robot is depicted in Figure 2.3, and major components are labeled. Several versions of the KR 16 has been made, including versions for use in foundries or clean rooms[8]. The standard version can be floor, ceiling or wall mounted. While it has a maximum payload of 16kg, there are fixtures for supplementary load on the rotating column, arm, and link arm. Refer to Table 2.2 for maximum load weights. Figure 2.5 shows the working envelope and dimensions for the KR 16. Rotational direction for all joints are specified in Figure 2.4. AC servo motors are used for all joints. The robot is not outfitted with springs or hydraulic gravity compensation, but uses brakes and servo motors to compensate for gravity.

The KR 16-2 is an updated version that was released for the KRC 2 2005ed. Apart from electrical connections, the only change is an improvement in repeatability from 0.1mm with the KRC 2, to 0.05mm with the KRC 2 2005ed[8], [9].

Load	Weight [kg]
Payload	16
Supplementary load	
Arm	10
Link arm	Variable
Rotating column	20
Robot mass	235
Total mass	281

Table 2.2: Maximum rated load for the KR 16[8]

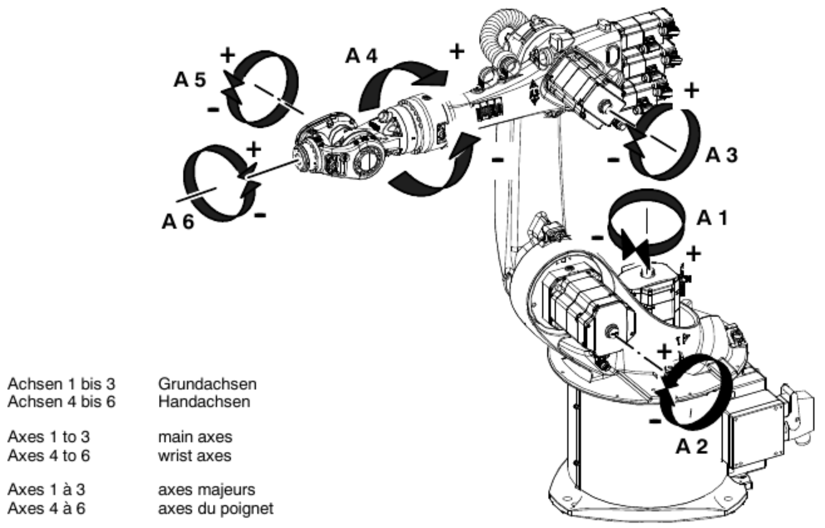

Figure 2.4: Rotational direction for robot axes. Courtesy of KUKA[8, Fig 1-2].

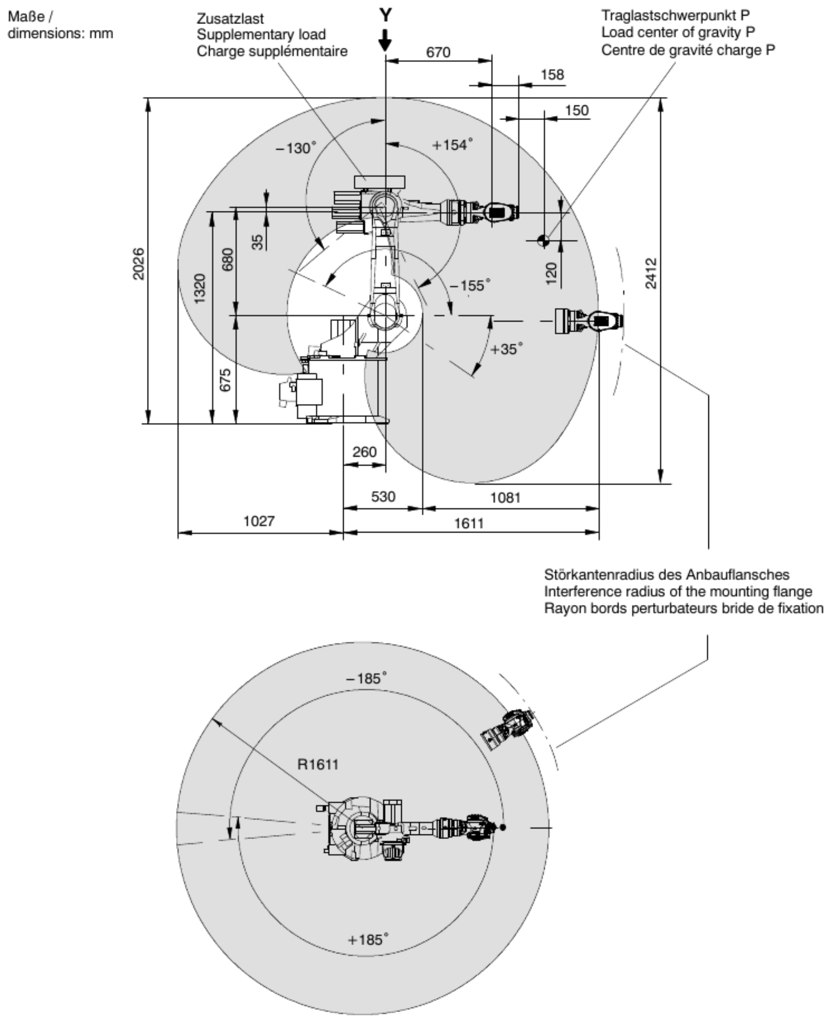

Figure 2.5: Principal dimensions and working envelope of the KR 16. Courtesy of KUKA[8, Fig 3-9].

2.3 KUKA System Software

KUKA System Software (KSS) is the software collection responsible for all robot operations. KSS v5.x is the newest version available for the KRC 2. KSS uses VxWorks and VxWin, which is included in the KSS installation. VxWorks is a real time operating system created by Wind River. VxWin was created by KUKA Controls to allow Windows and VxWorks to run on a single core processor. The original KRC 2 shipped with Windows 95, and required custom hardware to hand over control from Windows to VxWorks[10]. This consisted of a Real Time Accelerator (RtAcc) chip on the MFC3. The RtAcc triggered a non-maskable interrupt (NMI), forcing Windows to hand over the CPU to VxWorks[11]. This is illustrated in figure 2.6. With Windows XP an API providing the same functionality was introduced, removing the need for the RtAcc[12].

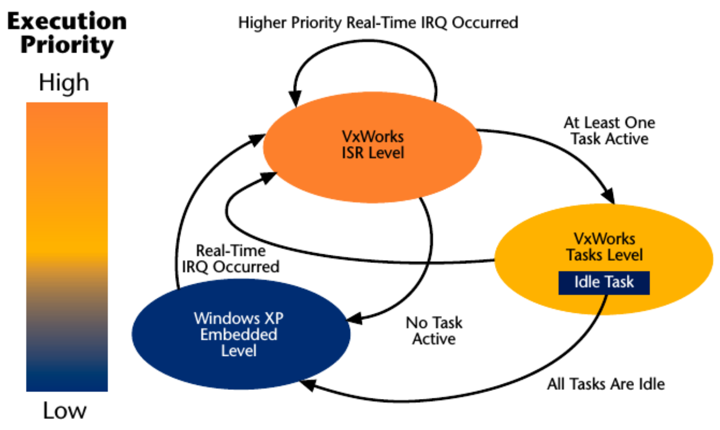

Figure 2.6: Interruptbased task scheduling. Courtesy of KUKA[13].

When starting KSS, VxWorks along with all robot programs is loaded into a RAM disk. According to [13] the x86 MMU is used to protect this RAM disk, along with VxWorks RAM space from Windows XP. This ensures VxWorks will run as expected, even if Windows XP segfaults (bluescreen). This places a limit on how much space one can use for creating programs for the robot.

Communication between VxWorks and Windows is done over a virtual network utilizing standard TCP/IP protocols. While most of the KSS runs on VxWorks, KUKA Human-Machine Interface (KUKA.HMI) runs in Windows XP. KUKA.HMI is the graphical user interface (GUI) for KSS. All robot programming and most of the configuration can be done directly from KUKA.HMI. In addition to KUKA.HMI, KUKA.Cross3, which runs in Windows, sets up and manages the connection and information flow between VxWorks and Windows. There is also a KUKA Cross-Com Visual Basic library that is used internally by KUKA. Among other functions, it allows reading and writing to global variables on the KSS.

KUKA offers several "tech packages", which expands the functionality of KSS. This includes packages for specialized operation, like KUKA.ArcTech which provides support functions for for arc welding, as well as more general packages for communication with other systems like KUKA.RSI and KRL.Ethernet.

By default three user levels are configured in KSS. These are `user`, `expert` and `administrator`. Not all functions are available for all users, and most configuration options are unavailable for the `user` level. Additional users and access levels can be set up on the KSS[14].

2.4 Interpolation Cycle

To decrease cost, there is no dedicated hardware for off-loading trajectory generation, path smoothing, etc. Instead the single core CPU on the KRC is shared between all processes. These tasks are ordered into the 12 ms long Interpolation cycle (IPOC)[15]. This cycle is again divided into four main parts:

1 KRL Interpreter and advance run
Main program execution. Motion planning and path smoothing is done in this step.

2 I/O and field bus
Fieldbus along with analog and digital I/O is updated in this step. This is one of the prioritized tasks for the controller.

3 Submit interpreter
Periodic tasks. Worth noting is that it might not complete a single run, or complete several runs depending on the execution time of code in the SPS.SUB.

4 Windows

Execution of all Windows programs and functions, including updating of GUI is in this step.

In an e-mail from KUKA Support it was clarified that part 1 and 2 runs at a higher priority than part 3. Windows has the lowest priority on the controller, and runs instead of the VxWorks idle task. Depending on complexity of part 1 and 2, run time for part 3 and 4 might be reduced.

2.5 Submit Interpreter

The Submit interpreter (SPS) acts as a software Programmable Logic Controller (PLC) in the KRC[16]. The German translation for PLC is "Speicherprogrammierbare Steuerung" and is shortened to SPS. A single KUKA Robot Language (KRL) program must be set to act as the main SPS program, by default this is `SPS.SUB`. It is started at controller boot, and runs as long as the controller is powered. Most KRL functions are available for use with the SPS, but it is not possible to move the robot or synchronous external axes. It is possible to control I/O and asynchronous external axes from SPS[16].

The SPS will run regardless of E-STOP status[17]. This means it can trigger I/O with an active E-STOP state. Any tooling connected to the robot could conceivably be started from the SPS even with an active E-STOP.

2.6 Programming

KRL is a high level programming language created by KUKA. It is based on PASCAL, and used on all KRC controllers. A KRL program consists of a `.SRC` file and an optional `.DAT` file. All program code is in the `.SRC` file. To increase readability a `.DAT` file, referred to as a datalist, can be used to initialize variables with data, and store it between program runs. KRL has support for conditional branching, loops, timers and interrupts[15]. Using interrupts it is possible to react to input or events during program execution. Table 2.4 has the most common KRL keywords. KRL has support for both subprograms and functions. Whereas a function returns a value, a subprogram does not return a value. A local subprogram or function is declared within the same file as the main program, and is only available for this program. A global subprogram or function must be declared in its own file, and is available for all programs. There is no distinction between a global subprogram or a program. Table 2.3 lists predefined datatypes. `INT S` and `INT T` in `POS` and `E6POS` stand for Status and Turn. As some robots can reach the same point with multiple joint configurations, these are used to limit axis range, giving a unique joint position for a point. The meaning of Status is dependent on robot configuration, and [15, pp. 62-65] should be consulted for the use of these.

Three main commands for motion programming is available, point-to-point (`PTP`), circular (`CIRC`) and linear motion (`LIN`)[15]. `PTP` moves the robot along the shortest path in joint space. Input can be either a joint space specification like `E6AXIS`, or a Cartesian position like `E6POS`. `LIN` and `CIRC` is used to program a continuous path for the Tool Center Point (TCP). `LIN` creates a linear movement from the current position, and takes a Cartesian position as input. In the case of `CIRC`, two Cartesian points are used as input. It creates an arc from the current position through the auxiliary point to the end point. If several motions are executed in order to create a trajectory, it is possible to define some points as approximate, allowing the controller to avoid unnecessary slowdowns due to hard turns[15]. See Figure 2.7 for an example using approximation with `PTP` motion.

2.6.1 Advance run

Advance run analyzes the KRL program, and plans movements ahead of time while waiting for the robot to complete the current move operation. If the current goal is not an end-point for the motion, advanced run is used to determine speed and acceleration through to the next goal. How many steps in advance should be calculated can be specified per program, up to a maximum of 5. It is worth noting this is the number of motion instructions, and not other logical instructions[15].

2.6.2 Coordinate systems

Cartesian coordinates is specified using X, Y, Z, A, B, C. Points in space are given by X, Y and Z, while A, B, C gives the rotation of this point. Transformation

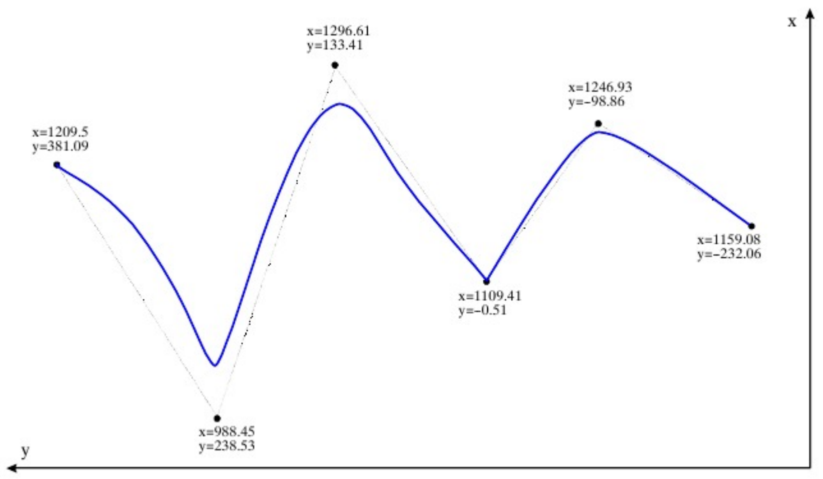

Figure 2.7: Illustration of approximation during PTP movement. Courtesy of KUKA[15, Fig. 33]

configuration

Keyword	Data type	Range
INT	Integer	$-2^{31}..2^{31}-1$
REAL	Floating point	$\pm 1.1x10^{-38}.. \pm 3.4x10^{38}$
BOOL	Bolean	TRUE, FALSE
CHAR	1 Character	ASCII character
AXIS	Struct	REAL A1,A2,A3,A4,A5,A6
E6AXIS	Struct	REAL A1,A2,A3,A4,A5,A6,E1,E2,E3,E4,E5,E6
FRAME	Struct	REAL X,Y,Z,A,B,C
POS	Struct	REAL X,Y,Z,A,B,C, INT S,T
E6POS	Struct	REAL X,Y,Z,A,B,C,E1,E2,E3,E4,E5,E6, INT S,T

Table 2.3: Predefined data types[15]

Keywords	Function
`FOLD`	Used to organize code in logical blocks. A fold can be opened or closed, hiding the code contained within the `fold`. One must have user level `expert` or above to see a `FOLD`.
`STRUC`	Create a composite data type.
`ENUM`	Create an enumeration data type.
`DECL`	Declare a variable.
`IF, ELSE, ENDIF`	Used for conditional branching.
`SWITCH`	Conditional branching. Can be used to avoid many nested IF statements
`FOR`	Counting loop
`WHILE`	Conditional loop
`REPEAT`	Conditional loop. Condition is checked at end of loop. Code is run atleast once
`LOOP`	Infinite loop. Must use `EXIT` to end loop.
`PTP`	Point-to-point motion. Moves all joints into the position supplied. This can either be supplied as joint angles or a Cartesian position.
`PTP_REL`	Same as `PTP`, but moves relative to current position.
`LIN`	Linear motion between a start point and a end point.
`CIRC`	Circular motion. Defined by three points: a start point, a end point and a auxiliary point. These are used to calculate an arc that the robot follows.
:	Geometric operator. Calculates transformation of two coordinate systems.

Table 2.4: Commonly used KRL keywords[15]

between coordinate frames can be calculated using the geometric operator ":". The four predefined Cartesian coordinate systems is listed in Table 2.5. In addition to Cartesian coordinate systems, there is a joint coordinate system. This is axis specific, and all joint values are used to define a point in space.

Coordinate system	Description
WORLD	Fixed, write-protected. Used as a reference for everything in the robot cell.
ROBROOT	Fixed but configurable. Located at the robot base. Defines position of robot in the `WORLD` coordinate system.
TOOL	Tip of the current tool (TCP). Located at the flange by default. Must be set up individually for each tool.
BASE	Used to define position of workpiece. Defaults to `WORLD`

Table 2.5: Predefined coordinate systems[15]

2.6.3 Torque mode

To allow the robot to be pushed around, for example during ejection of a workpiece from another machine, Torque mode is used. It limits the current used to hold a servo stationary, making the joint compliant[18]. While Torque mode is activated, active control of joint position is not possible. Torque mode can be activated for all axes, but it is only recommended for use with A1. For A2-A6 it is either not recommended due to axis sagging or physically not feasible due to gearing considerations[18].

2.6.4 Collision monitoring

The robot automatically adjusts the required motor torque in order to keep to the programmed path and speed. If the robot collides during movement, it will simply increase torque to keep to the programmed path. It is possible to set up a torque range the robot should stay within during movement to minimize damage during a collision[18]. KUKA refers to this range as a "monitoring-tunnel". If torque monitoring is enabled, and the robot exceeds the set torque range, the robot stops with path maintaining braking. The monitoring-tunnel must be adjusted for each movement where it is required. Collision monitoring is not available for external axes[18].

2.6.5 Workspace monitoring

Workspace monitoring can be used to detect TCP entering or leaving a defined area. This can then be used to trigger I/O or stop robot motion[18]. Workspaces can either be defined by a set of joint ranges or as a box in Cartesian space. If it is used to protect an area from collision, the brake time of the robot must be taken into consideration when creating the workspace. A total of 8 workspaces can be monitored simultaneously. Refer to [18, pp. 42-63] for configuration instructions.

2.7 External axes

An external axis is any servo controlled by the KRC 2 that is not part of the robot. The KRC can control up to six external axes. There is support for using several servo motors to control one external axis. To achieve this the motors are coupled using either torque mode or position mode depending on the stiffness of the axis.

If the motions of the external axes should be coupled with a reference frame, they can be configured as an external kinematic system. Up to three external axes can be used in one kinematic system, and 6 external kinematic systems can be defined on the KRC 2[19]. These are referenced as `#EASYS` to `#EFSYS`. If the robot itself is mounted on an external kinematic system, such as a linear positioner, it is configured as a ROBROOT kinematic system. Only one ROBROOT system is supported on the controller, and is referenced by `#ERSYS`[19]. In addition to `ROBROOT`, `BASE` and `TOOL` kinematic systems are available. A `BASE` kinematic system moves the workpiece, while the `TOOL` kinematic system moves the TCP.

The motion of an external axis can be either synchronous or asynchronous to the robot motion. For a synchronous external axis, the motion starts and ends at the same time as a programmed robot motion[19]. Synchronous motions are further divided into mathematically coupled or non-coupled motions. For mathematically coupled motions the robot must take the external kinematics into account during path planning. For non-coupled motions, like the feed servo for a welding gun, this is not required. Asynchronous motions are either coordinated or uncoordinated. Coordinated motions are controlled from the KRL program, while uncoordinated motions are controlled from a separate operating panel.

With a `ROBROOT` kinematic system, the `ERSYSROOT` coordinate system is defined. This is located at the zero point of the first axis in the kinematic system. The `WORLD` coordinate system can be offset from `ERSYSROOT`, and a transform between `ROBROOT` and `WORLD` is calculated using `ERSYSROOT` along with current axis positions for the kinematic system[19]. This allows the controller to compute the TCP position in reference to the `WORLD` coordinate system. Using a ROBROOT system, the position of the external axes must be supplied for any robot movement where it is required to move the external axes. If only a Cartesian point is specified, the controller will try to reach this point only using the robot axes. If the point is out of reach for the robot an error will be reported and the program halted.

2.8 Robot Sensor Interface

This section is based on information found in [20], [21]. RSI is a software extension package for KSS intended for sensor assisted motion. It can directly offset paths or joint values during interpolation. Maximum allowed path and axis correction is set in software, and this defaults to $\pm 5mm$ for translation and $\pm 5^\circ$ for rotational corrections.

If an extra Ethernet network card is installed and assigned to VxWorks in the KRC, RSI[1] is able to communicate over TCP/IP or UDP/IP. There is a `fast mode` and `normal mode` for Ethernet communication. As shown in figure 2.8, in `fast mode` the remote system must respond within a 2ms window, whereas it has roughly 12ms to respond in `normal mode`. While there still is only one position correction during each IPOC, with `fast mode` the correction can be made with recent data.

Transmitting KRL variables is done by linking said variables to an RSI object.

[1]RSI versions under 2.1 requires KUKA.Ethernet RSI XML, a licensed software package of KUKA

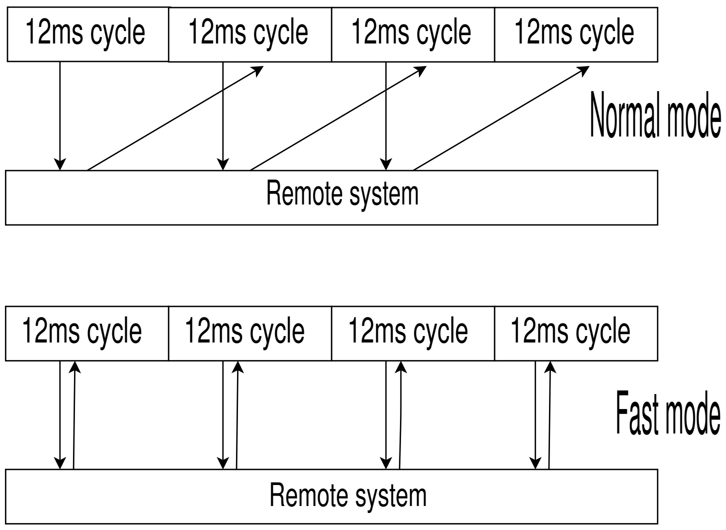

Figure 2.8: RSI timing diagram[4]

This object is then automatically updated against the KRL variable when RSI sends or receives data. Table 2.6 shows pre-existing RSI objects. Incoming data can be used for input to sensor guided motion. RSI will try to apply any position or axis corrections within the same IPOC it receives them. This allows for precise position control in 12ms steps. Large corrections will cause steep acceleration and fast motions.

RSI works on a lower level than the KRL interpreter, and some of the safety features of the controller is ignored. If a Cartesian, or joint value, correction is issued using RSI, the KRC will attempt to reach the new set point within the current 12ms IPOC. Validity of the new point is not checked before the movement is initiated. Programming errors can lead to very sudden and unexpected movements. It is recommended to limit workspace size to minimum requirements to minimize consequences of errors.

During normal KRL execution, velocity and acceleration is automatically limited to the defined maximum speed for the robot. It is possible to limit this further by setting a maximum joint speed for the current KRL program using *OV_PRO*. By limiting robot velocity to what is necessary for the planned motion, the damage caused by programming errors can be limited. These limits are enforced if the robot is being remote controlled using KVP. However, RSI does

Keyword	Description
DEF_RIst	Send the Cartesian actual position
DEF_RSol	Send the Cartesian command position
DEF_AIPos	Send the axis-specific actual position of axes A1 to A6
DEF_ASPos	Send the axis-specific command position of axes A1 to A6
DEF_EIPos	Send the axis-specific actual position of axes E1 to E6
DEF_ESPos	Send the axis-specific command position of axes E1 to E6
DEF_MACur	Send the motor currents of robot axes A1 to A6
DEF_MECur	Send the motor currents of external axes E1 to E6
DEF_Delay	Send the number of late data packets

Table 2.6: Pre-existing RSI objects[20]

not honor these restrictions. During testing in T1 mode it was found that the KRC will enter a protective STOP mode after the velocity exceeds the limit of 250 mm/s. We have not attempted to exceed maximum velocity or acceleration for the robot, but assume the controller will apply brakes and abort the program with error `1101 COMMAND ACCELERATION EXCEEDED` or `1102 COMMAND VELOCITY EXCEEDED`[22].

2.9 Fieldbus

Fieldbus is a set of standards defining networks and protocols for communication in process automation, usually between a PLC and programmable sensors and devices. It was first standardized by ISA in 1992, and received significant expansion when standardized by IEC in 2000[23]. Various transmission protocols and topologies exist within these standards, but the trend for the last years is to move to devices using Ethernet as a transmission media[23].

Profibus started out as a German national standard, to serve as a link between HMI and PLCs. When IEC 61158 was released in 2000, Profibus was included as one of the 8 fieldbus standards[23]. It uses a master-slave topology, where the master cyclically polls slaves for information. All Profibus devices must have a

GSD (equipment master data) file, which is used when configuring the master for work with a slave. The maximum transfer rate is between 9600Kbps and 12Mbps depending on cable length[23].

DeviceNet is an extension to the CAN protocol, and intended for connecting simple sensors and actuators with PLC or PCs. It can operate in either master-slave or a producer-consumer peer to peer topology, with a maximum transfer rate of 500Kbps[23]. Devices can be powered over the DeviceNet bus, eliminating the requirement for separate cabling. DeviceNet is managed by ODVA.

Interbus is divided into two section, local loop and remote bus. Devices like sensors and actuators are connected to the local loop, which is terminated in remote nodes. These nodes are connected through the remote bus, usually over Category 5, twisted-pair copper wiring. One pair is usualy used for power, and the local loop and remote bus uses 1 or 2 pairs for communication. Data is forwarded from nodes in a ring structure, with a maximum of 4096 local and remote modules. The transmission protocol has a data rate of 500Kbps and is designed to have very little overhead[23].

Chapter 3

Robot software

Software for controlling industrial robots is traditionally made by the robot manufacturer. The robot is then installed and programmed for use in a specific working environment by a systems integrator. Programming and integration costs can be more than half the up-front cost of a new robot cell[24]. In recent years robots are coming out of the big factory floors and into smaller factories and shops. This has created a demand for easier programming and smarter robots and sensors. Both manufacturer software, 3rd party proprietary software, and open source software have started to fill this gap. This chapter describes some of choices of software packages that is relevant for use with KUKA KRC 2 controllers.

3.1 Robot Operating System

Apart from paragraphs regarding rqt, rviz and diagnostics this section is taken in its entirety from [4].

ROS is an open source robotics framework. It grew out of the robotics community at Stanford University, with Willow Garage, a now defunct robotics laboratory, doing significant developments on the early versions. As of July 2016 there are over 100 supported robots, and ROS has a worldwide user base[25].

In [26] the philosophical goals of ROS are listed as:

- Peer-to-peer, with respect to intercommunication,
- Tools-based, one small program per task,

- Multi-lingual, programming language-neutral,
- Thin, all complex libraries should be reusable outside ROS,
- Free and Open-Source

In accordance with these goals, today there are over 7000 unique ROS packages[25]. These packages are loosely connected, and it is easy to change out components to suite whatever needs one might have. ROS has native bindings for Python, C++ and LISP, with experimental support for Java and Lua.

Nodes in ROS are usually programs designed for specific tasks. A complete system consists of many nodes, where each node is responsible for part of the total control of the robot. In figure 3.1, nodes are marked as ellipses. The *kuka_hardware_interface* is responsible for communication between ROS and the KRC. The *robot_state_publisher* is responsible for computing all necessary transformations for the robot.

Communication between different nodes are done using custom message formats. These messages are then sent over communication channels called topics. Topics use a publisher-subscriber communication paradigm, where nodes can communicate in a one-to-many, many-to-one, or many-to-many fashion.[27]

In figure 3.1 topics are marked as squares, with lines between the nodes. *kuka_hardware_interface* listens on the *position_trajectory_controller/command* topic, and publishes the current robot position on the *joint_states* topic.

In addition to topics, ROS has support for request based communication. This is implemented using services and the action server. Services work in a request-reply communication paradigm. A client sends a request to a service provider, and waits for a reply from the service provider. The action server operates in an asynchronous manner, and offers some more flexibility. Action servers are typically used for tasks with longer processing time, where feedback and the possibility to stop an action is required. An action client sends a goal to an action server. During computation the action server will send feedback to the action client. Upon reaching the goal a result is sent to the action client.[28]

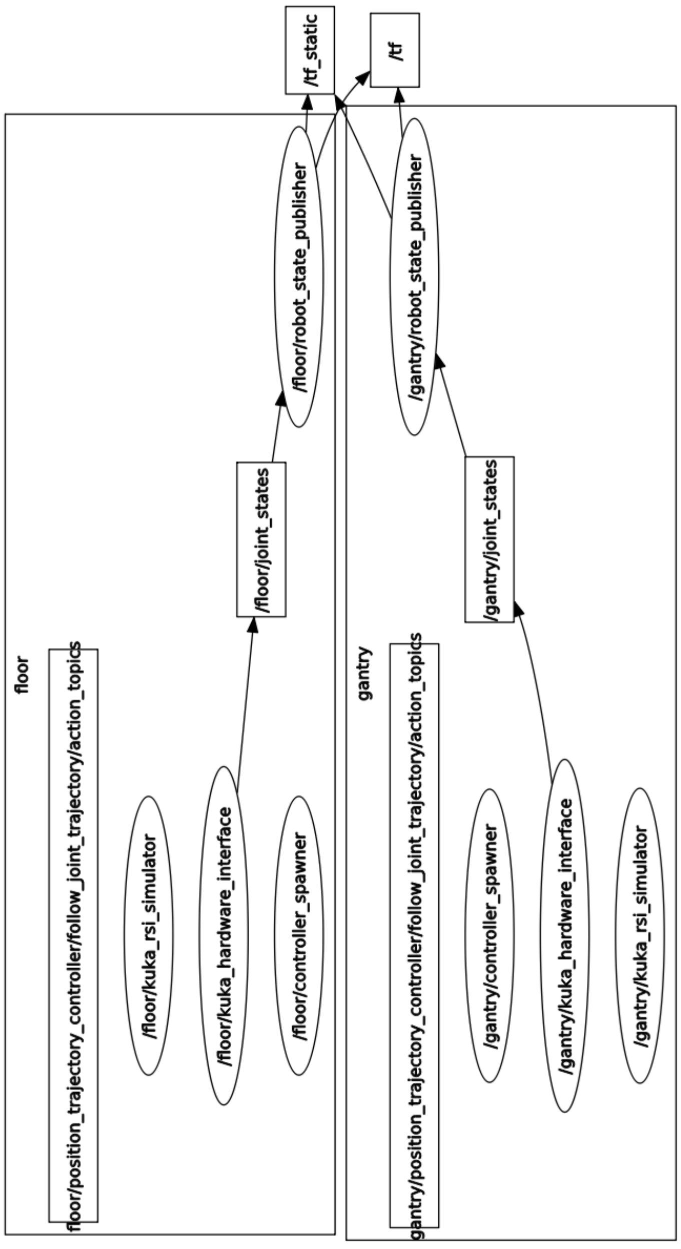
/tf_static
/tf
floor
gantry
/floor/robot_state_publisher
/gantry/robot_state_publisher
/floor/joint_states
/gantry/joint_states
floor/position_trajectory_controller/follow_joint_trajectory/action_topics
gantry/position_trajectory_controller/follow_joint_trajectory/action_topics
/floor/kuka_rsi_simulator
/floor/kuka_hardware_interface
/floor/controller_spawner
/gantry/controller_spawner
/gantry/kuka_hardware_interface
/gantry/kuka_rsi_simulator

Figure 3.1: Example of nodes and topics in ROS

Unified Robot Description Format (URDF) is an XML format used to describe robots. It specifies elements for links and joints. Links are described using 3D models or textual definition of simple geometric figures. The links are connected using joint elements, and there is support for different joint types, namely fixed, revolute, continuous, prismatic, floating and planar. It has support for defining joint speed, link inertia and link mass. Figure 3.2 shows the link and joint relationship for a KUKA KR 16 robot.

Xacro is an XML macro language. It makes it easier to work with large URDF models. It has support for parameters, simple math functions, inclusion of other files, parameters and conditional blocks. This makes it easy to use concise URDF files to build large, complex robot models.

ROS Control is a collection of ROS packages for abstraction of hardware drivers. It contains a set of controllers and hardware interfaces, and does the mapping between them. Figure 3.3 shows the data flow internally in ROS Control. If one can reuse one of the existing interfaces, creating a new robot driver for ROS is reduced to updating parameters with current robot data, and forwarding control data to the robot.

MoveIt! is a motion planner for ROS. According to [30] it is "state of the art software for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation". It has a modular design, and supports several libraries for planning, kinematics and collision checking. MoveIt! can be interfaced through C++, Python, or a GUI. Figure 3.4 illustrates the integration of MoveIt! with ROS.

rqt is a qt based GUI framework for ROS. It can be expanded by plugins, and is the preferred way to create new GUI applications for ROS. Over 20 plugins is available, including plugins for robot diagnostics and control[31].

rviz is a 3D visualization tool for ROS, that can be run through rqt or as a standalone app. In addition to render robots poses, it is used for displaying sensor

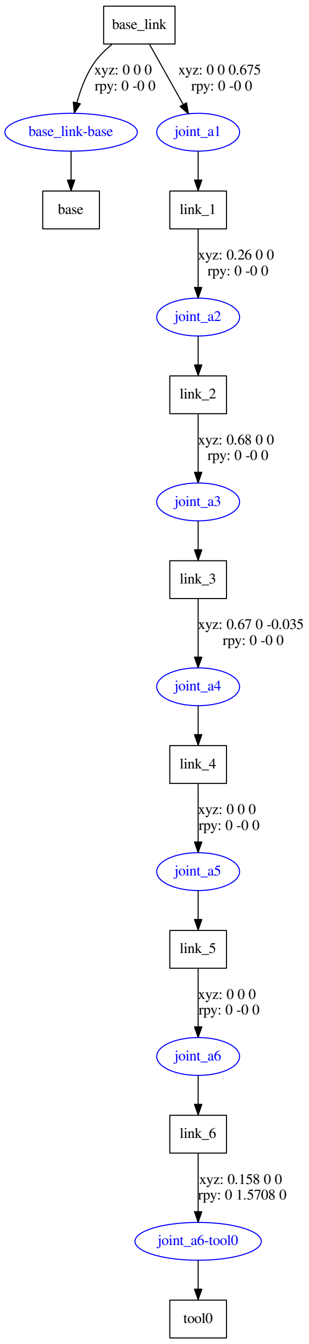

Figure 3.2: Example of URDF tree

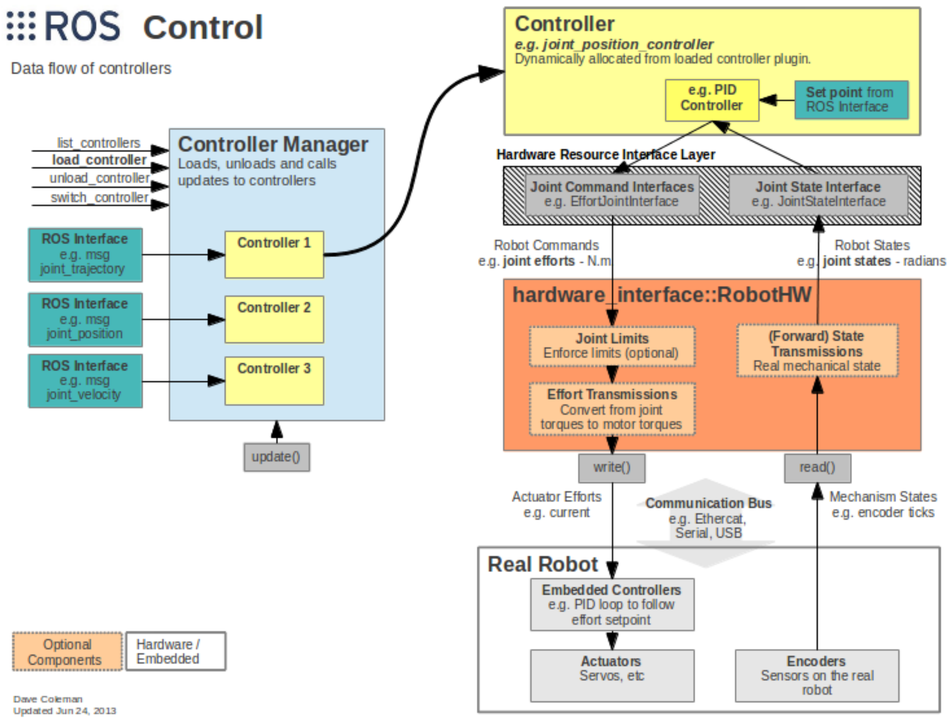
ROS Control
Data flow of controllers
list_controllers
load_controller
unload_controller
switch_controller
Controller Manager
Loads, unloads and calls updates to controllers
ROS Interface e.g. msg joint_trajectory
ROS Interface e.g. msg joint_position
ROS Interface e.g. msg joint_velocity
Controller 1
Controller 2
Controller 3
update()
Controller
e.g. joint_position_controller
Dynamically allocated from loaded controller plugin.
e.g. PID Controller
Set point from ROS Interface
Hardware Resource Interface Layer
Joint Command Interfaces e.g. EffortJointInterface
Joint State Interface e.g. JointStateInterface
Robot Commands e.g. joint efforts - N.m
Robot States e.g. joint states - radians
hardware_interface::RobotHW
Joint Limits Enforce limits (optional)
Effort Transmissions Convert from joint torques to motor torques
(Forward) State Transmissions Real mechanical state
write()
read()
Actuator Efforts e.g. current
Communication Bus e.g. Ethercat, Serial, USB
Mechanism States e.g. encoder ticks
Real Robot
Embedded Controllers e.g. PID loop to follow effort setpoint
Actuators Servos, etc
Encoders Sensors on the real robot
Optional Components
Hardware / Embedded
Dave Coleman
Updated Jun 24, 2013

Figure 3.3: Data flow in ROS Control[29]. *CC-BY-3.0 wiki.ros.org*

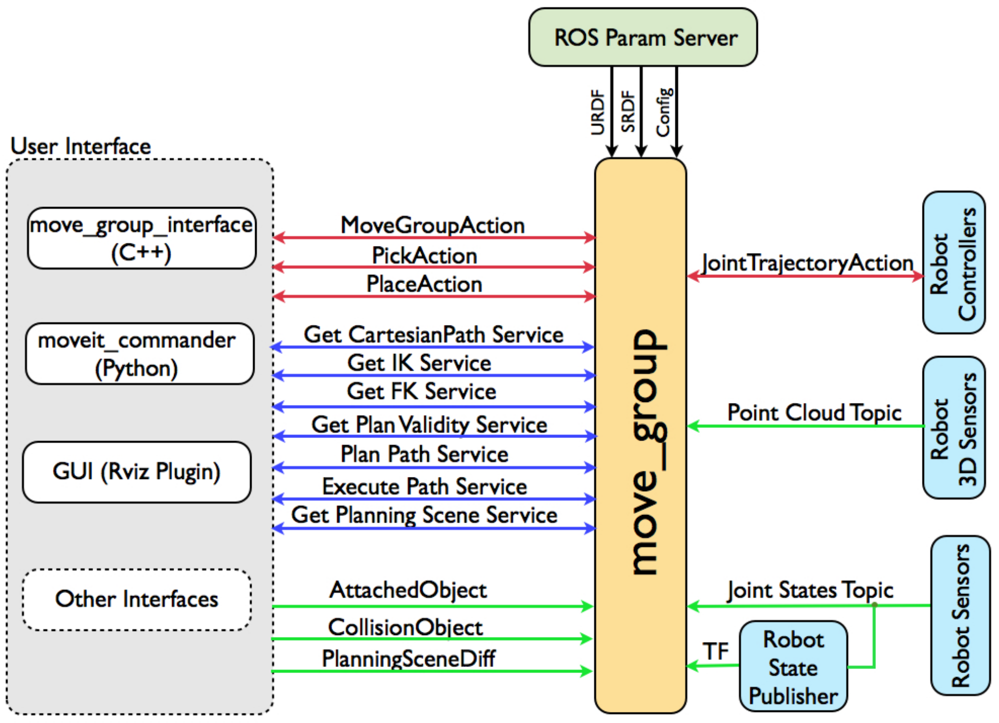

Figure 3.4: MoveIt! system architecture[30] *CC-BY-3.0 wiki.ros.org*

data, forces, path visualization (with MoveIt!), navigation data, and more[32].

Diagnostics is a helper package for collecting and analyzing diagnostics messages from a robot[33]. It is easily expanded with plugins. The rqt_robot_monitor uses diagnostics data to show color coded status information for a robot. It has tools for exporting diagnostics data to CSV files. This makes it easier to use standard plotting and analyzing tools on ROS diagnostics data.

3.2 ROS 2

Several limitations exist in the current ROS design, most notably are missing real-time support, lack of security features and support for running multiple robots at the same time. In order to address these issues, it was not deemed feasible to maintain backwards compatibility, and work was started on ROS 2[34]. Among other infrastructure changes, ROS 2 is designed to be real-time safe, and uses Data-Distribution Service for Real-Time Systems (DDS) for communication. A list of changes between ROS 1 and ROS 2 is in [35]. The default DDS implementation used in ROS 2 is eProsima Fast RTPS[36]. It offers both authentication and encryption of traffic[37]. The first non-beta release of ROS 2, Ardent Apalone, was released on December 8, 2017[38]. A communication bridge for allowing ROS 1 and ROS 2 packages to communicate is available with the ROS 2 distribution.

3.3 ROS Industrial

"ROS-Industrial is an open source project that extends the advanced capabilities of the Robot Operating System (ROS) software to manufacturing"[39]. This is done by creating hardware drivers for equipment used in manufacturing, such as industrial robots, fieldbuses, sensors, etc. In addition to hardware drivers, they create and maintain packages and plugins related to industrial use of ROS.

ROS Industrial (ROS-I) is managed by the ROS Industrial Consortium (RIC). This is a membership organization that sets the roadmap and provide support for the ROS-I community. Members interested in supporting development of new features can create a Focused Technical Project (FTP), where development is funded by interested RIC members.

3.3.1 Godel - Robotic blending

An example for a FTP is Godel. It is a project to create a automatic surface blending (sanding) application. It scans the workspace with a 3D scanner and visualizes the result, the operator then selects the surfaces that should be blended[40]. Godel automatically creates a trajectory for blending the selected surfaces. After this is done, the operator can execute the blending process. After the blending process is completed, the robot does a quality assurance scan, and additional blending is done if any defects are detected. This is intended to become a general Scan'n'Plan framework for robotic processes[40].

3.3.2 Descartes

Descartes is another example of an FTP. It is a path planning library for semi-constrained Cartesian paths. Semi-constrained in this context means that the path is specified with less degrees of freedom compared to what is available on the robot. This could for example be for drilling, where rotation around the drill bit is indifferent. As it is a Cartesian planner, it takes a list of Cartesian points, and calculates a trajectory following these points[41]. MoveIt! currently lacks a good Cartesian planner, and there is an open issue on GitHub for integrating the Descartes with MoveIt![42].

3.3.3 Robot support packages

ROS-I maintains a set of packages for using industrial manipulators from most major robot suppliers[43]. These packages are organized into a metapackage named after the manufacturer, and strive to have the same layout between manufacturers. The typical layout is listed below.

<robotname>_support

Necessary 3D models and XACRO files for creating a robot URDF. By using

XACRO macros to define the robot, it is easy to reuse these packages in larger setups.

<robotname>_moveit_config
Configuration files for using MoveIt! with robot.

<robotname>_moveit_plugins
Robot specific MoveIt! plugins, like kinematic solvers.

<robotname>_gazebo
Configuration files for using Gazebo

<manufacturer>_driver
Driver for connecting the robots to ROS. Naming is inconsistent across manufacturers

Experimental features are gathered in a separate metapackage for each manufacturer, following the same layout as the non-experimental packages. The packages for KUKA robots is still deemed experimental, and is located the `kuka_experimental`[1] repository.

3.4 Gazebo

Gazebo is a 3D physics simulation environment for robotics, which started at the Robotics Research Labs, University of Southern California[44]. By simulating a wide range of sensors, with optional noise, it provides a realistic environment for the simulated robots[44]. It has support for several physics engines, but the choice must be made at compile time. Open Dynamics Engine (ODE) is the default physics engine, and is distributed with the binary release[45]. Gazebo uses the `SDF` format to describe the simulation, with Collada or STL files for describing 3D objects.

ROS uses Gazebo as the primary physics simulation tool. The Gazebo release cycle is independent from the ROS release cycle, a specific Gazebo version is therefore locked to a ROS distribution in order to maintain lifecycle compatibility.

[1]https://github.com/ros-industrial/kuka_experimental

A `SDF` description is auto generated from an `URDF` file when using Gazebo with ROS. A `<gazebo>` element is added to URDF to provide information needed by Gazebo that not specified in the URDF standard.

3.5 Orocos

Herman Bruyninckx of Katholieke Universiteit Leuven first launched the idea of an open control framework for robotics, Open RObot Control Software (Orocos), on the EURON mailing list in December 2000. The project was made possible by an EU grant, and the initial version was released in 2002[46]. This release then turned into the Real-Time Toolkit (RTT). By 2009 both the Bayesian Filtering Library (BFL) and the Kinematics and Dynamics Library (KDL) had been added as subprojects to Orocos. They provide stand-alone functionality, allowing them to be used outside Orocos. While lack of activity on the project web site[1] might indicate this project is dead, all three sub projects have seen releases on Github over the past two years[47]–[49]. The Orocos project is focused on creating a framework for real-time control of robots[46], as such both KDL and BFL is real-time safe.

3.5.1 Bayesian Filtering Library

BFL is framework for inference in Dynamic Bayesian Networks[50]. Particle filters and Kalman filters are implemented in BFL. The documentation has not been updated since 2010, and [51] hints to undocumented features in the library.

3.5.2 Kinematics and Dynamics Library

KDL is a framework for modelling and computation of kinematic chains. According to [52] it has extensive support for geometric primitives, kinematic trees, motion trajectories and solvers for kinematics and dynamics. It is application independent, and is used by more than 20 ROS applications[53].

[1]http://orocos.org/

3.5.3 Real-Time Toolkit

RTT is a C++ framework for creating real-time applications that is modular and run-time configurable[54]. RTT programs are divided into `packages` that consists of one or more `components`. Each `component` then implements some functionality. `Components` are run by the Orocos `deployer`. Orocos State machine Description (osd) provides support for event driven `components`[55].

Communication between `components` is done through directional ports. The connection between these ports are made using transport libraries. Current implementations include MQueue for interhost communication, and CORBA for intrahost communication. Transport of all data types can be made hard real-time as long as the underlying protocol supports this[55].

OroGen is a code generator for RTT that assists in the creation of new RTT components. One specifies the component in a definition file, and oroGen creates C++ and CMake files based on this specification[56]. Orocos Component Library (ocl) contains optional RTT infrastructure components. The ROS `rtt_ros_integration` package provides RTT and ROS integration.

3.6 YARP

Yet Another Robot Platform (YARP) stems from research in humanoid robotics at the LIRA Laboratory at University of Genova, and the CSAIL and AILabs at Massachusetts Institute of Technology[57]. It started out as a library to increase maintainability and usability of software in complex systems with changing hardware. It does this by providing communication and hardware abstraction[57]. A YARP based system is organized into small `processes`, where each is responsible for a task. The underlying communication mechanism is plugin based, and can be easily changed to suit requirements and available hardware without changing code in the `processes`[58].

3.7 OpenRAVE

Open Robotics and Animation Virtual Environment (OpenRAVE) is a motion planning framework for use in robotics, developed at the Robotics Institute in Carnegie Mellon University, and first released in 2008[59]. It is not intended to be used for stand alone control of robots, but rather provide motion planning capabilities to other robotic middleware. It has an plugin based architecture, and provides abstraction layer for motion planners, hardware, and physics engines[60].

3.8 RoboDK

RoboDK is a commercial program for simulation and offline programming of robots. It is manufacturer agnostic, and has support for over 30 different manufacturers[61]. This is achieved by using post processors to generate programs in robot specific languages like KRL. Programs can be created graphically or in python. For some cases like 3D printing or milling, the program can be created automatically from CAD files, like STL or DXF. RoboDK is not free or open source, but the post processors have been released to ROS-I under an open source license.

3.9 FlexGUI

FlexGUI is a web based GUI for robot and factory management, created by PPM A/S. Since it is web based, it can run on any device with a web browser. FlexGUI 4.0 connects to ROS, and is able to control any ROS enabled robot. Custom control and monitoring screens is created from within a web browser, and allows for creating a GUI highly customized to the current user and process. With Factory Designer it is possible to create a screen containing all processes in a plant. While advanced features like Factory Designer is not free, part of FlexGUI has been open sourced, and is available via ROS-I[62].

3.10 KUKAVARPROXY

KVP is created by the IMTS S.r.L. Company, and released in binary form on sourceforge[1] together with openshowvar. It acts as a TCP/IP bridge between external systems and KSS. According to [63] a maximum of 10 simultaneous connections to KVP is supported. KVP runs on the Windows side of the KRC, and is able to communicate with the VxWorks side of the controller via KUKA CrossCom. This communication is illustrated in Figure 3.5. We have not been successful in finding any documentation on the KUKA CrossCom library. The only place KVP appears to be documented is in [63]. KVP appears to have been used successfully in numerous research projects as well as commercial software like roboDK[63]–[66].

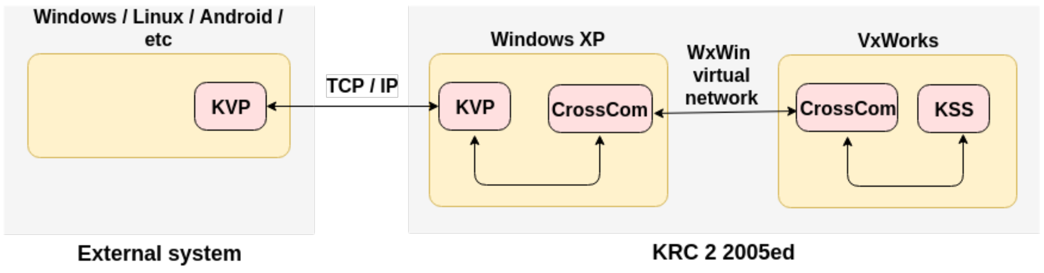

Figure 3.5: Communication between and external system and KSS using KVP.

Two libraries for communicating with KVP is available. JOpenShowVar, presented in [63], is a Java library for accessing KVP. In addition to communicating with KVP, it has implemented classes for all KRL variable types. BoostCrossCom is a C++ library inspired by JOpenShowVar, and was created for use in [64]. In an e-mail from autumn 2016, Njåstad has stated that it's going to be released on GitHub under an open licence. It is not as feature rich as JOpenShowVar, and lacks classes for KRL variables types.

IMTS S.r.L. is in the process of creating an open-source release of KUKAVARPROXY. This will help "demystify" KUKAVARPROXY and could increase the willingness to depend on it from other open-source projects like ROS-I.

[1]https://sourceforge.net/projects/openshowvar/

Chapter 4

Robot cell setup

Two of the planned research scenarios for the laboratory is additive manufacturing and testing gripping scenarios. Both of these will most likely require gathering and processing of (depth) images, which is computationally expensive. The computing power of the KRC 2 controller is limited, and outdated by today's standard. Instead of upgrading the KRC controllers, it was decided to let them only handle motion and safety, and use a external system to handle everything else.

For open-loop control, a path can be preplanned and used to generate KRL files, for example with the postprocessors from RoboDK. These files would then be uploaded to the KRC and executed. For closed-loop control, the ability to adjust the path during execution is required. In our laboratory we can use either RSI or KVP to achieve closed-loop control.

In this chapter will go through available hardware in the laboratory, before describing the choice and setup of middleware. Safety and security considerations for the laboratory is also described in this chapter. In line with the definition found in [67], throughout this text safety is considered protection against unintended harm, while security is protection against willful harm.

4.1 Robot cell hardware

The layout of the robot cell is a large fenced area and a control station directly outside the fence. As seen from Figure 4.1, a clear acrylic wall separates the control station from the robots. There are two gates in the fenced area, both monitored by gate switches connected to the E-STOP controller. The robot cell contains the following equipment:

- 1x floor mounted KUKA KR 16
- 1x 3-axis Güdel gantry crane with a ceiling mounted KUKA KR 16
- 2x KRC 2 2005 edition controllers
- 2x FTC-50-80 force-torque sensors
- 2x SWS-020 toolchangers
- 2x SMC EX250 pneumatic controllers
- 2x auxiliary DC power systems
- 1x E-STOP controller

The KRC 2 for the gantry mounted KR 16 is fitted with an expansion cabinet for controlling the gantry. The RDC for the gantry is placed on the right leg, seen from the control station. Apart from the gantry, setup for both robots are identical. The Schunk FTC-50-80 force-torque sensors are mounted on the robots, and connected to the KRC over DeviceNet. The SWS-020 toolchanger is pneumatically operated, and controlled over Profibus via the SMC EX250. The SMC EX250 is placed on 3rd axis arm of the KR 16. The auxiliary DC power system consists of two power supplies placed at the KRC, and DC-DC converters at each robot base. Together these supply 24V, 12V and 5V to equipment and tools on the robot.

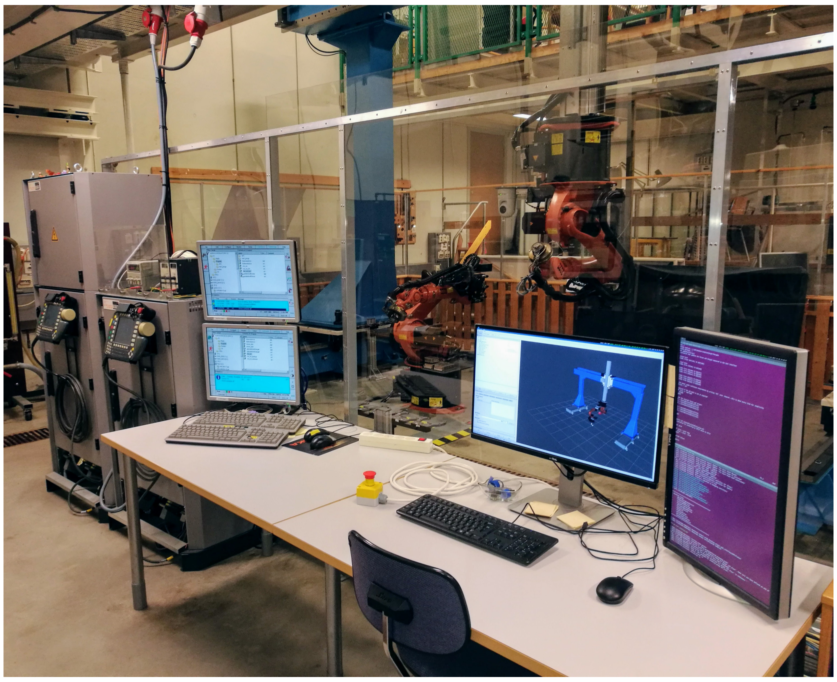

Figure 4.1: The laboratory as seen from the control station. The KRC cabinets are on the left side, with the control computer on the right side.

4.2 Selection of middleware

As seen from chapter 3 there are several available frameworks and software solutions for use in robotics. Some of the key points that were considered when selecting software solutions are listed below.

Documentation and support
: Is it well documented? Is it easy to get support?

Availability
: Free to use? Available sourcecode?

Adaptability
: How easy is it to adapt to the current task at hand?

Interoperability
: Is it already integrated into other frameworks? How easy is it to integrate with other frameworks?

Expected lifetime/Community
Is this something that will be around for years to come? How big is the userbase?

Metric	ROS	YARP	Gazebo	Orocos	RoboDK
Citations[1]	3826 [26]	470 [57]	813 [44]	573 [68]	-
Google search	263,000[2]	9,930[3]	90 200[4]	11 700[5]	228[6]
Open source	Yes	Yes	Yes	Yes	No
Licence	BSD	LGPL	Apache 2.0	LGPL, GPL	

1 Citations to referenced article. Based on data from Google scholar pr 2017-10-30.
2 Search term ros "robot operating system"
3 Search term yarp "yet another robot platform"
4 Search term gazebo robot simulator
5 Search term orocos "open robot control software"
6 Search term robodk "robot development kit"

Table 4.1: Comparison of robotic middleware. Based of table found in [69].

ROS has almost 19 000 wiki pages, along with some 18 000 users on the primary support platform[1], meaning there is a large community around ROS[70]. According to [70] ROS has over 13 million downloads of over 9000 unique packages. By utilizing existing packages in the robotics lab, we are able to spend less time on creating a support framework and more time on completing the wanted task. The major drawback of ROS is the lack of hard real-time support. It is possible to use Orocos for tasks with real-time demands, and offloading non-real-time tasks to ROS. Due to the high number of preexisting packages, and interoperabillity with other middleware, it was decided to interface the laboratory with ROS.

As ROS is primarily targeted for Ubuntu, this was a natural choice of Operating System for the control computer. In [4], the laboratory was set up using ROS Indigo and Ubuntu 14.04. MoveIt! was released for ROS Kinetic in late December 2016, and it was decided to upgrade the laboratory to Kinetic and Ubuntu 16.04 Xenial Xerus. Both Kinetic and Xenial are Long Term Support (LTS) releases, with support until April 2021[71].

[1]https://answers.ros.og

4.3 Safety

The following safety hazards have been identified in the robot cell:

- Electric shock and Electrocution
- Crunch and pinch
- Collision
- Ejection of tool during operation

4.3.1 Electric shock and Electrocution

Shock and electrocution risks can be negated by removing power before performing any maintenance or service work on the equipment. Both controllers and robots comply with Directive 73/23/EEC (Low Voltage Directive), and do not constitute shock or electrocution risk under normal operation. As the auxiliary power supply is rated under 50V DC, it does not impose a shock or electrocution risk.

4.3.2 Crunch and pinch

Crunch risks are from collisions between the robots and humans. In addition, several pinch points have been identified on both robot and gantry crane. These risks are negated by keeping humans outside the workcell during normal operation. This is done with a security fence that is placed around the robots. The gates are monitored by the E-STOP system. Any movement of the robots while the gates are open is only possible in T1 mode, and requires physically depressing the enable switch on the KCP. While in T1 mode speed is limited to 250mm/s[7].

4.3.3 Collision

Two main collision scenarios have been identified:

Robot - robot collision
: Any unwanted contact between the two robots, or any part connected to the robot.

Robot - environment collision
: Any unwanted contact between a robot and any object not part of the robot. Main contributors here are the security fence, tool holders and floor.

To minimize risk of collision, workspace monitoring in KSS should be applied. This will monitor the TCP position, and stop robot motion if the TCP enter a restricted area. For both robots, a Cartesian area where the robot can move should be defined. If the TCP leaves this area it will halt robot motion. The workspace should be defined so the following items are excluded from it:

- Floor
- Gantry rail
- Gantry leg
- Fence

While this will help in minimizing collision, it will not negate the risk completely. The robots could still collide with each other inside the defined workspace. For the gantry robot, the elbow joint and robot base could collide with the floor or gantry rails and legs even though the TCP is inside the defined workspace. As the robot needs to come in contact with the toolholders during a tool change operation, protecting against these are not possible using workspace monitoring.

The KRC has collision monitoring, but this requires tuning for each programmed path. Since this is a laboratory setup, where robot paths are expected to change often, this is not suitable for use in our case.

4.3.4 Ejection of tool during operation

It is possible to release the tool while the robot is moving, as the locking mechanism is not connected to robot motion in any way. Worst case scenario is tool release during motion, followed by the tool being thrown over the security fence. To negate this risk, one should check that the robot is stationary and positioned at a toolholder before releasing the tool lock.

4.4 Security

The security risks are identified as unauthorized access, use or modification of the robot cell. These can again be divided into physical or remote access.

4.4.1 Physical access

The laboratory itself is locked, and access is limited to NTNU staff and students with the Cybernetics department. Access the control station and the KRC cabinets should ideally be restricted to only users of the laboratory, but this is not physically feasible due to other lab equipment in the same room. To avoid unauthorized use of the robots a lockable switch is connected to the E-STOP. This must be unlocked to move the robots. In addition, the gate to the fenced area is padlocked, limiting physical access to the robots to only authorized personnel. Access to the control computer and KRC is restricted by username and password.

4.4.2 Remote access

Neither KVP or RSI has any form of authentication or authorization. This, in addition to any potential security holes in Windows XP, means we should have a separate physical network for the KRCs, and other lab equipment that operates over Ethernet. With a separate physical network for lab equipment, these can be interconnected without security risks of remote access. The Ethernet switch for this network is placed inside the fence, minimizing the chance of unauthorized access to this network. The control computer has two Ethernet ports, meaning we can connect it to the lab network and Internet at the same time. The risk vs benefit of connecting the control computer to the Internet deserves some attention.

With this being a laboratory setup, the required software stack on the control computer will evolve with current research projects. The ability to download new software directly, rather than transfer it to the system via other media simplifies this work. Should the computer not have an Internet connection, the process of installing the latest security and bug fixes will become cumbersome, and likely something that is not done regularly. Moreover, it will ease the software development process, as changes made during laboratory testing can easily be synchronized to a centralized repository.

As this is a small laboratory, where it is relatively easy to gain physical access, we consider the risk of a targeted, Internet based attack to be minimal. Untargeted attacks usually exploit known security holes in software, and patches to fix security issues are frequently released. Ubuntu has support for automatically installing security updates, limiting this attack vector[72].

ROS is not designed with security in mind, and has no mechanisms for authentication or authorization of incoming traffic. Running ROS on a system with Internet connection and no firewall would allow anyone to take control of the system. Security must therefore be solved outside of ROS itself before the control computer is connected to the Internet.

To highlight this problem, scans for open ROS installations in NTNU's network was performed on 8 occasions during February and March of 2017. These revealed an average of 6 accessible installations, spread over 13 unique IP addresses. While this is a low number of installations, each of them could expose control of real hardware to anyone on the Internet.

While Ubuntu by default comes without an enabled firewall, it has support for a kernel level firewall through Netfilter[73]. This can easily be used to only allow incoming traffic that is related to current outgoing traffic. With all unwanted incoming traffic blocked, the risk from security holes in software that opens outgoing connections still remain. Automatically installing security updates will reduce this risk, but not entirely remove it. In addition, there are plans for moving the control computer from a public routable network, to a internal NTNU network. This network uses a NAT to allow outgoing Internet traffic while limiting the exposure of connected devices.

With proper security measures, like automatic updates and a firewall, we believe the benefits far outweighs the risks of connecting the control computer to the Internet.

4.5 Control of mathematically coupled external axes with RSI

If the robot is mounted on an external axis, the axis should be mathematically coupled. This enables KSS to factor in external axes when determining joint movements with regard to maximum TCP momentum. It also enables KSS to compute the Cartesian TCP position in the `WORLD` frame, allowing us to use Cartesian workspace monitoring with external axes. This is done by calculating `ROBROOT` w.r.t. `WORLD` frame, and TCP w.r.t. `ROBROOT`, giving TCP positions in the `WORLD` frame.

RSI can be used to change the robot position, either as a offset in TCP Cartesian position, or as a change in axis joint values. To be able to control axis position, the `ST_AXISCORR` object is used. This sets a correction for the current axis joint value, and KSS attempts to reach this new axis configuration within the current IPOC. When RSI is used to move mathematically coupled external axes, it will lead to a change in `ROBROOT` w.r.t. `WORLD` frame. As KSS tries to keep the TCP constant w.r.t. `WORLD` frame, it will compensate for this change in `ROBROOT` by moving the robot axes. The same behaviour is seen using `PTP_REL` with underspecified joint values. As the ROS driver expects full control of all joints, this is unwanted behaviour that must be corrected for. KUKA support has confirmed in an e-mail that it is not possible to disable this behaviour for mathematically coupled axes. They proposed a solution where `ST_PATHCORR` is used to offset TCP position by the same amount as the axis correction for the external axes. They did not guarantee that this solution would work, and recommended through testing if it was applied. As a correction for both Cartesian position and joint offsets must be set, this solution will lead to added complexity.

Two alternative solutions to this problem exists, but both are suboptimal. The first is to remove the mathematical coupling. Movement of the external axes would then move the `WORLD` reference frame. Use of Cartesian workspace monitoring would become impossible. In addition, we must make sure the dynamic load is within acceptable limits as the KRC is unable to monitor this. The second solution is to issue corrections to robot axes to counteract KSS. This is a reactive solution, where we first must observe what corrections the controller makes, before counteracting these corrections. This is not easily implemented into the current `kuka_rsi_hw_interface`, and would require restructuring the driver.

The ability to use workspace monitoring makes up for the added complexity of the controller code. In addition to workspace monitoring, it will make it possible to check the Cartesian TCP position before releasing the tool lock on the toolchanger. It is not possible to set the mathematical coupling of axes at runtime. If it is removed, it will not be possible to use Cartesian workspace monitoring with the KVP driver. The KVP driver is unaffected by this problem as we only use fully specified joint values.

4.6 Path planning and robot motion

We are able to control the robots from an external system, like our control computer, using either KVP or RSI. In [4] it was concluded that both these interfaces could be used, and that both should be used, as they serve different needs. While KVP provides better safety, for example by being able to limit robot speed on the KRC, the RSI interface provides a real-time control loop running at 12ms. RSI however, does not forgive programmings errors. With the largest gantry axis being 4.7m long, and a top speed of 0.83 m/s, it will take just under 6 seconds to travel the entire length of the axis. As this has the potential to cause damage, the use of the RSI interface should be limited to applications needing the real-time control capabilities it offers.

ROS Control uses hardware interfaces to communicate with hardware, and controllers to communicate with ROS. This allows for some flexibility between controllers and hardware interfaces as they do some internal conversion. For example, a Joint Command interface, which takes a set-point and updates the hardware can be used with both the Joint Trajectory and Joint Position controllers. The Joint Position controller simply forwards a set-point to the hardware interface, while the Joint Trajectory controller will take a trajectory as an input, and interpolate in order to provide set-points to the Joint Command interface at every control loop update[74].

While communication with the KRC is different, both RSI and KVP hardware interfaces uses the same ROS Control interfaces. The Joint State Interface, along with the Joint State Controller is used to publish the current joint values on the `joint_states` topic. As the KRC does not provide joint velocity or acceleration values, only joint position is published. The Joint Command interface is used to receive a joint value from ROS, which is forwarded to the KRC. Update frequency for the RSI interface is controlled from the KRC and follows the IPOC. The KVP interface attempts to follow the IPOC 12ms update rate, but is dependent on read and write speed from KVP.

It is possible to use ROS Control as a Orocos Component, enabling hard-realtime control of the robot. A example of how to create this integration is available on GitHub[1].

MoveIt! is the best supported path planner for ROS. It uses a concept of move groups to plan motions. A move group is a collection of joints that should be controlled simultaneously. When given a start and goal position, it plans a collision free path in joint space between the points. Using iterative parabolic time parameterization MoveIt! can add optional time parameters to this path, generating a trajectory. This parameterization is based on velocity and acceleration limits[75]. The generated path or trajectory is handed over to a joint trajectory controller. This controller is responsible for executing the trajectory on actual hardware. An example for a trajectory controller is the ROS Control Joint Trajectory controller.

[1]https://github.com/skohlbr/rtt_ros_control_example

Chapter 5

Development and experimental work

Based in part on work done in [4], the following areas have been identified for improvements.

Workspace monitoring
: In order to enable workspace monitoring, we need the ability to control mathematically coupled axes from ROS.

Investigate KVP timing information
: To determine the viability for using KVP, more data on access time is required.

Improve the KVP driver
: Restructure the current KVP driver to increase reliability and maintainability.

ROS support for F/T sensor
: The robots have Schunk FTC-50-80 sensors mounted on them. They should be integrated with ROS

ROS support for tool changers
: The robots have Schunk SWS-020 tool changers mounted on them. A ROS node for safely controlling them should be developed.

ROS Kinetic upgrade

As MoveIt! had not been released into Kinetic yet, the lab was set up for Indigo in [4]. MoveIt! has since received a Kinetic release. Required changes to use Kinetic in the laboratory should be performed.

Export robot status information from KSS to ROS

Currently, only joint state information is exported from the KRC to ROS.

Create example code for the laboratory

5.1 Access time for KUKAVARPROXY

The only documentation of KVP performance appears to be [63]. While it shows promising results, all tests are done on a KRC 4. In order to compare the proficiency of KVP with other control methods, like RSI, performance data for the KRC 2 must be gathered.

There are three main factors that affect the performance of the library used.

1. Processing delay on control computer
2. Network round-trip delay time (RTD)
3. Processing delay on KRC

Processing delay on the control computer is expected to be negligible. As there is only one switch and less than 30m of cable between the KRC and the control computer, RTD is expected to be under 1ms. Processing delay on the KRC is a bit more complex, and can be divided into the following parts

1. KVP receives TCP/IP packet.
2. KVP executes CrossCom call
3. VxWorks responds to CrossCom call
4. KVP processes the data and transmits it over TCP/IP

Since Windows XP is running as the VxWorks' idle task, the CPU time available for Windows will depend on controller load. As the exact task planning for VxWorks on the KRC is unknown to us, we have to assume non-deterministic behaviour for windows runtime. In Figure 5.1 it is assumed all VxWorks tasks finish before Windows XP uses the remaining IPOC. This is most likely an oversimplification, but illustrates some wait-delays that may occur during KVP processing.

A number of ICMP ping tests should be done to find an average RTD. Timing a number of read and write tests with different variables, will provide data on KVP latency. By testing with two different libraries, and all other parts being equal, any major difference in test results should be due to performance of the libraries on the control computer.

In [63] it is concluded that access time is the same regardless of data type. To confirm this on a KRC2, tests will be run with a single data type (*INT*) and a composite data type (*E6AXIS*).

To see if multiple connections affects access time, tests should be run using both a single connection and multiple connections.

While it is nice to know how long it takes to write to a variable, there might be an unknown delay from KVP reports the variable as written until it is accessible from a KRL program. To get an idea of the order of magnitude of this delay, a write-copy-read strategy, which is illustrated in Figure 5.2, is used.

1. Use KUKAVARPROXY to write to a VAR1
2. SPS copies VAR1 to VAR2
3. Read VAR2 using KUKAVARPROXY

5.1.1 Results

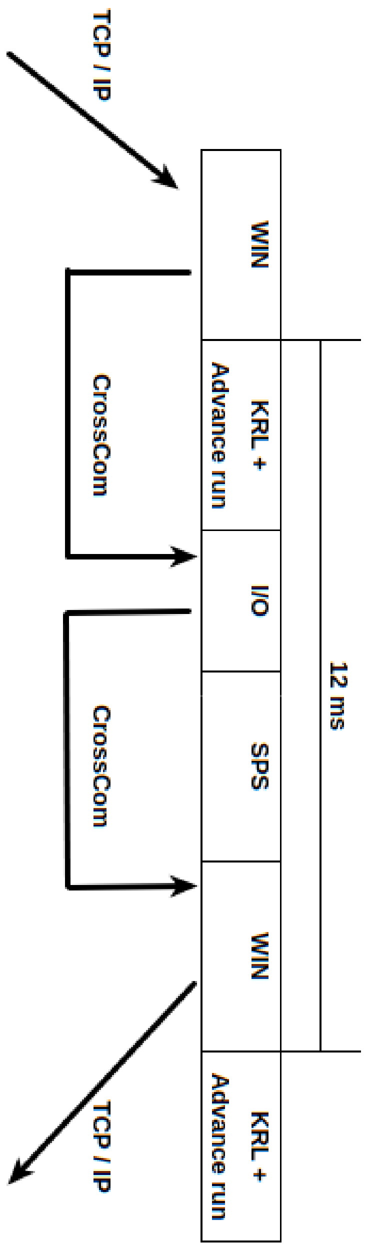

Figure 5.1: KVP processing during IPOC

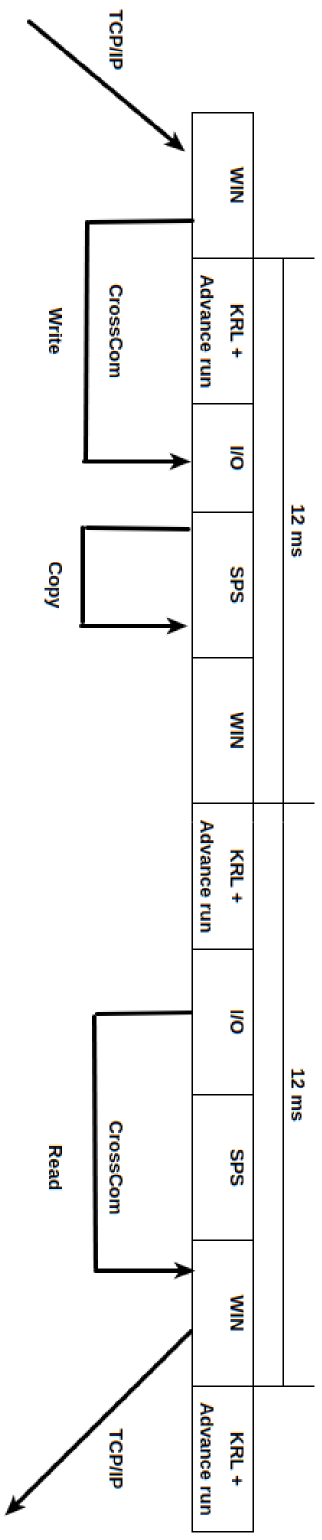

Figure 5.2: Illustration of write-copy-read strategy

All tests are conducted with the robot stationary and no program selected on the controller. Access time is measured from a command is issued until it is completed. This includes formatting of KVP packages and network delay. Each test is run 1000 times. In the 5 thread WRITE-COPY-READ test, only one of the threads is used for measurement. The remainder of the threads run read and write operations in order to add load to the network and controller.

Max	Min	Median	Mean	Std dev
3.13	0.28	0.34	0.59	0.63

Table 5.1: Round-trip delay time (RTD) for ICMP ping requests in ms. n=1000

Language	Type	Threads	Max	Min	Median	Mean	Std. Dev
Java	INT	1	23.60	1.71	2.14	2.48	0.89
C++	INT	1	23.78	1.77	2.15	2.48	0.91
Java	E6AXIS	1	13.46	1.84	2.29	2.64	0.86
C++	E6AXIS	1	9.37	1.86	2.29	2.64	0.86
Java	INT	5	30.03	6.04	11.36	11.73	1.30
C++	INT	5	74.49	2.45	11.62	11.78	1.42
Java	E6AXIS	5	36.02	1.81	12.16	11.82	2.61
C++	E6AXIS	5	36.52	2.36	12.77	12.38	1.32

Table 5.2: Access times for READ operations in ms. n=5000

5.1.2 Discussion

During tests in the lab, it has not been possible to use more than 5 simultaneous connections. This might be from some underlying differences in KUKA CrossCom between KSS 5.x and KSS 8.x, which is propagated to KUKAVARPROXY.

Results from the ICMP ping test in Table 5.1 places the network RTD in the expected range of below 1ms.

In [63] access time is reported as 4.27ms average using JOpenShowVar on a KRC4. This is higher than the 2.64ms - 3.64ms average access time on the KRC 2

Language	Type	Threads	Max	Min	Median	Mean	Std. dev.
Java	INT	1	10.55	1.64	2.06	2.38	0.82
C++	INT	1	16.05	1.67	2.08	2.39	0.87
Java	E6AXIS	1	13.00	2.64	3.33	3.64	0.96
C++	E6AXIS	1	14.44	2.60	3.28	3.60	0.96
Java	INT	5	24.14	8.45	10.82	11.35	1.28
C++	INT	5	72.86	2.14	10.97	11.41	1.54
Java	E6AXIS	5	29.74	10.30	16.15	15.38	1.88
C++	E6AXIS	5	30.48	3.44	16.55	16.08	1.33

Table 5.3: Access times for WRITE operations in ms. n=5000

Language	Type	Threads	Max	Min	Median	Mean	Std. dev.
Java	INT	1	36.50	12.79	24.07	23.97	0.80
C++	INT	1	47.61	6.97	24.03	23.92	1.14
Java	E6AXIS	1	57.71	11.55	23.79	23.97	0.98
C++	E6AXIS	1	59.03	6.89	23.69	23.91	1.35
Java	INT	5	51.99	12.83	33.57	31.35	5.81
C++	INT	5	97.55	11.40	23.42	24.11	3.12
Java	E6AXIS	5	59.29	11.37	37.08	35.80	9.55
C++	E6AXIS	5	60.27	11.86	25.03	32.58	10.83

Table 5.4: Access times for WRITE-COPY-READ operation in ms. n=5000

Test	Threads	Max	Min	Median	Mean	Std. dev	# > 30ms
Read	1	61.66	1.64	2.05	2.40	0.96	5
Write	1	58.76	1.57	2.00	2.32	0.95	4
Read	5	622.96	4.95	10.68	11.34	2.38	114
Write	5	78.78	4.05	10.17	10.92	1.52	273

Table 5.5: Additional C++ read and write test in ms. n=500 000

using JOpenShowVar. This could be due to differences in hardware and software on the KRC or differences in network setup. Without more knowledge about average RTD, configuration of the KRC 4 and other hardware used in [63] it is hard to come to any conclusions as to why there is a small difference between the tests.

As can be seen from Table 5.2 and Table 5.3, timing differences between JOpenShowVar and BoostCrossCom is negligible for single thread operations. However, for INT read and write operations using 5 threads, BoostCrossCom does have a significant higher maximum time compared to JOpenShowVar. For both read and write operations, only four of the 5000 samples are above 30ms. To determine the frequency of these spikes, additional read and write tests was performed. Results from this test is in Table 5.5. For 5 thread read and write operations, this test yielded respectively 114 and 273 samples above 30ms. The root cause for these spikes are unknown, but the occurrences are fairly rare.

While both read and write times for E6AXIS are higher than for INT, this is faster than two sequential reads. Variables could therefore be bundled into a composite data type to lower access time. This is in line with the findings from [63].

As the average access times are much lower than the 12ms IPOC, the model used in Figure 5.1 is too simplistic. In Figure 5.3 the IPOC is illustrated with interrupts and periodic start of tasks. This is likely closer to the actual structure of the IPOC.

From Table 5.4 it can be seen that average WRITE-COPY-READ time is around twice the cyclic rate of the KRC2. The only change made on the KRC was in the SPS loop. The contents of the SPS loop does not affect Windows run-time, as it will loop for a constant time regardless of contents. This delay could be from an IPOC more or less consistent with Figure 5.2, or a design weakness in the test. The test runs in a single thread using one connection to KVP. After executing the write command, it will issue consecutive reads until it gets the correct value back. This means a read call might be underway as the SPS is run, adding extra time to the response. See Figure 5.4 for an illustration of this.

For each series of tests, the operations are run back-to-back. This could potentially affect the results, as the controller is in the same place of the IPOC

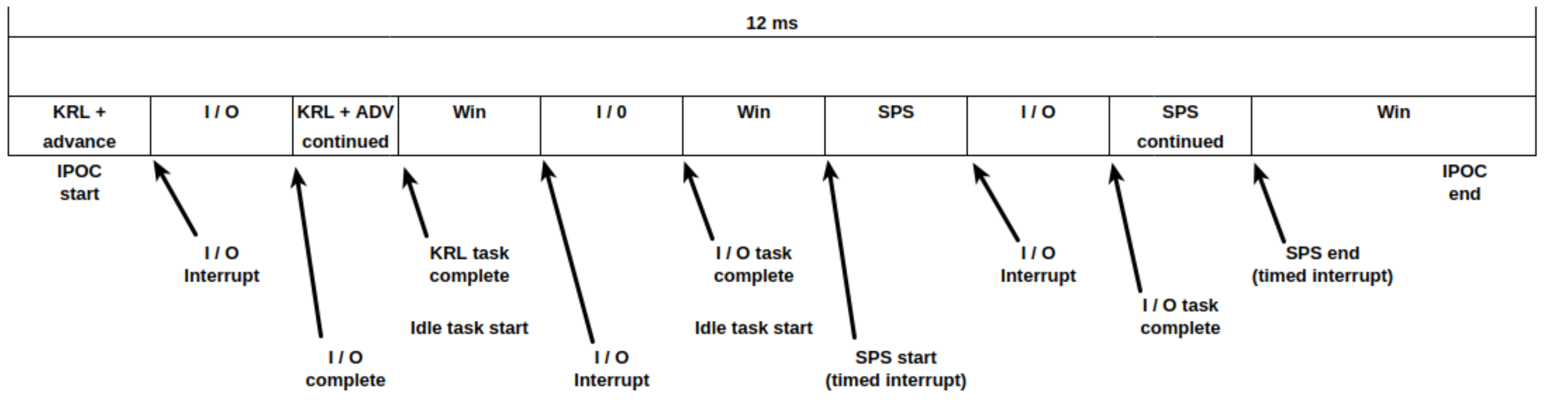

Figure 5.3: Possible IPOC layout with timers and interrupts

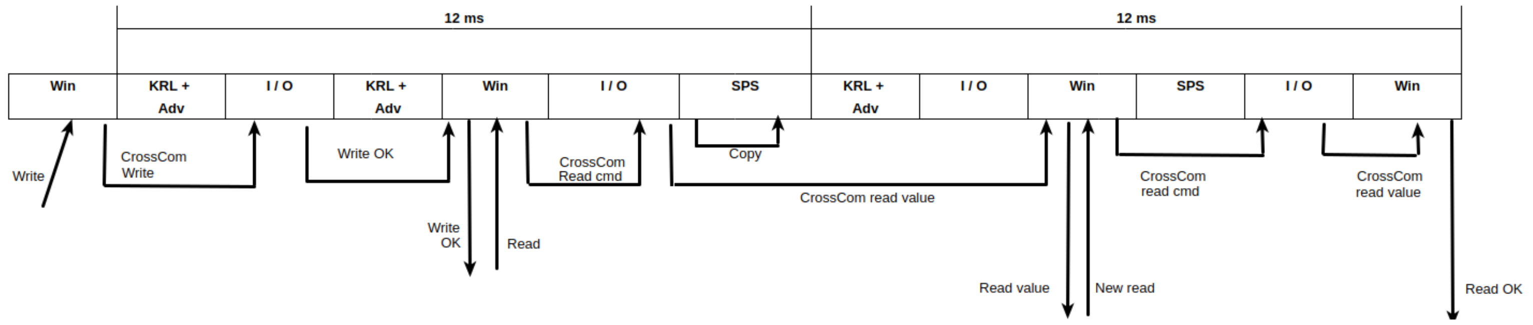

Figure 5.4: Extra time to WRITE-COPY-READ due to extra KVP read

every time a request is sent. Further studies could asses this by introducing a random wait [0-12ms] between each request.

5.2 Movement latency using KUKAVARPROXY

The time it takes for the robot to move after a write command is issued will be referred to as the movement latency. We have investigated this by placing an external accelerometer on the robot that is connected to the control computer. The movement latency is then the time from a command is sent until it is registered by the accelerometer. Since more computing is involved it is expected that this time is around half of the average WRITE-COPY-READ time from section 5.1.

The simplest way of moving the robot from KVP is with a KRL program that loops over the `PTP` command. Input to the `PTP` command is given as a `AXIS` or `E6AXIS` variable, that is updated using KVP. Listing 5.1 contains an excerpt of the code running on the KRC for these tests. `AXIS_ACT` is an global `E6AXIS` variable.

The external accelerometer consists of an Arduino micro and a MPU-6050 triple axis accelerometer. This is connected to the external computer using USB, and programmed to write to a serial terminal when movement in any direction is detected. As show in Figure 5.5, a breadboard with the Arduino and MPU was jury-rigged to a cable support on axis 6.

In order to determine the communication latency between the control computer and the Arduino, an echo test was performed. For this test the Arduino was set up to send back any data it received over USB. A program on the control computer is set up to send a byte to the Arduino, and measure the time it takes before it is returned. A static delay was added on the Arduino to simulate a processing delay. This has been subtracted from the results to get the actual transmission time.

```
LOOP
      PTP  AXIS_ACT
ENDLOOP
```

Listing 5.1: KRL code to move robot using KVP

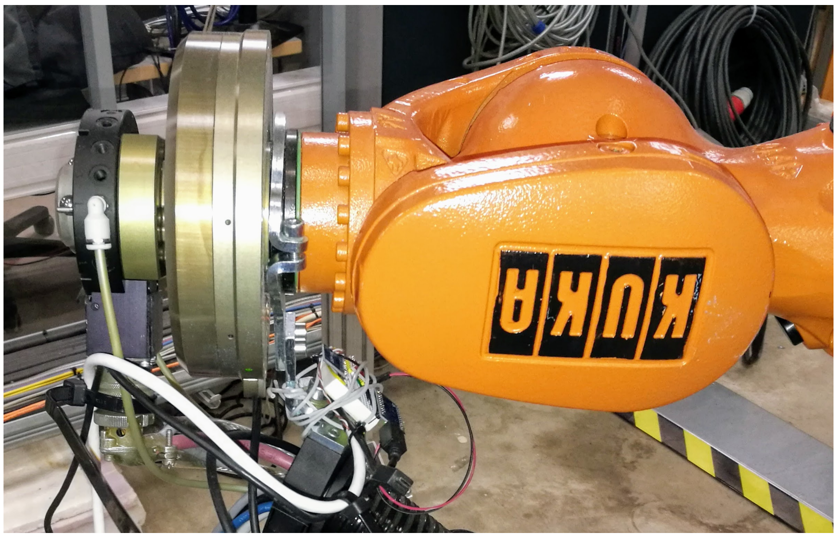

Figure 5.5: Arduino and IMU fastened to A6 on the KR 16

The testing the movement latency, a program on the control computer is set up to measure time from writing a new position to the robot until the external accelerometer detects movement.

5.2.1 Results

Language	Max	Min	Median	Mean	Std. dev.
Java	6.66	0.23	0.88	0.89	0.35
C++	0.92	0.66	0.85	0.85	0.03

Table 5.6: RTD between the Arduino and the control computer in ms. n=200

Language	Max	Min	Median	Mean	Std. dev.
Java	19.93	2.01	5.59	6.71	4.61
C++	29.59	2.01	5.04	5.74	4.07

Table 5.7: Movement latency in ms. n=298

5.2.2 Discussion

RTD between the Arduino and control computer is within the expected range. For C++ this test was done using Boost's serial port library. For the actual movement test, this library produced too high readings due to resetting of the serial port after a timeout. It was therefore dropped in favor of using the POSIX `termios.h` library. This is used by Boost on Linux, and removing the extra layer provided by boost should not affect performance in a negative way.

The first result from both the C++ and Java datasets was abnormally large, and has been discarded. The extra time for these results is most likely from the Arduino opening the serial port.

The mean movement latency of 5.74 ms for jOpenShowVar and 6.71 ms for BoostCrossCom is roughly half of the 12 ms IPOC. With the expected results around 12 ms, both libraries perform better than expected. The only major difference between jOpenShowVar and BoostCrossCom is in maximum time, where BoostCrossCom has a 1/3 higher maximum compared to jOpenShowVar.

On average one should be able to update the robot position once every IPOC. This is comparable with the performance of RSI. However, unlike RSI, the simple KRL program we used for this test is not able to abort or affect an ongoing motion.

5.3 ROS with RSI and mathematically coupled external axis

Support for controlling external axes was added to the RSI-based ROS driver in [4]. This implementation does joint position corrections by using the `ST_AXISCORR` object. It is linked to an XML formatted input which is sent over UDP/IP from an external source. While it has performed well, it does not address the problems described in Section 4.5, and requires uncoupled external axes to work correctly. To enable use of mathematically coupled external axes, the the solution suggested by KUKA support by e-mail has been implemented.

A `ST_PATHCORR` object was added to the KRL code, which is used to offset current TCP position in relation to the KUKA `#WORLD` coordinate frame. It was not possible to link both `ST_AXISCORR` and `ST_PATHCORR` objects to the same input, and extra input has to be sent from the external system. Necessary changes was made for `kuka_rsi_hw_interface` to send changes in X,Y,Z for the `ST_PATHCORR` object. Currently, the link between external axes and the `#WORLD` frame is hard-coded to align with our setup. This solution has performed well during testing, and no additional work is needed for it to work with our hardware setup. Should this code be considered for inclusion in the official repository, some more work is needed to make the coupling between axes and directions configurable at run time.

5.4 BoostCrossCom ROS Control interface

The proof of concept driver using KVP from [4] has been rewritten. To achieve a smooth path execution, two different approaches for a KVP based driver have been attempted. The first approach is largely structured after `ros_control_boilerplate`[1] example code. It uses ROS Control to handle the ROS side of the controller. The second approach implements a `Follow Joint Trajectory Action` server in order to execute trajectories. It is not possible to adjust movement speed with these controllers, but maximum speed can be limited by setting the program override speed (`$OV_PRO`) on the KRC.

The BoostClientCross library from [64] is still used for communication with KVP. A few functions for easier handling of KRL variables have been added to the library. Based on results from section 5.1 it was decided to use two threads, one for reading joint state, and one for writing joint commands. By doing parallel reads and writes, joint states can be read once every IPOC. In addition it will increase the responsiveness of the controller.

5.4.1 Joint Position Command

Like the RSI driver, the first approach uses ROS Control, and implements a Joint Position Command interface. With the Joint Position Controller it will forward the target pose to the KRC, and the KRC will create and execute a motion profile between the current position and target pose.

[1]https://github.com/davetcoleman/ros_control_boilerplate

By updating the controller with the next goal pose after the motion has started, path interpolation during the KRL advance run can update the motion profile to avoid a full stop at the current goal. For this to work, there must be enough time left of the ongoing motion for the KRC to update the motion profile. If this is not the case, the robot will slow down near to a full stop at the current pose, before starting the next motion. The result of this is a jerky motion.

The RSI driver uses the Joint Trajectory Controller from ROS Control to handle path interpolation. The interpolated points will be sent to the KRC every 12 ms, allowing for control of velocity and acceleration from ROS. Testing have shown that this update rate is too fast for KVP, producing both jerky motion and skipping some trajectory points altogether.

5.4.2 Follow Joint Trajectory

In an attempt to achieve smooth trajectory execution, a `FollowJointTrajectory` action server has been created. It has an actionlib interface for receiving a trajectory, which is then forwarded to the KRC pose for pose. During movement, KSS sets two KRL variables, indicating the start pose and goal pose for the current movement. The KVP trajectory controller uses this goal pose variable to synchronize with the KRC. As soon as a motion is started, the next trajectory point is uploaded to the KRC. This guarantees all trajectory points are executed, and the correct order of execution. If the poses are sufficiently far away in joint space, the KRC is able use the newly uploaded pose for path interpolation during the KRL advance run. This allows the KRC to create a motion profile where it avoids a full stop at every trajectory point.

Similar to the Joint Position Command driver, it uses ROS Control's Joint State interface for reading state from the KRC. To increase readability of the code, it has been separated into its own ROS node.

Unfortunately, this controller also shows jerky movements if the trajectory points are too close. Future trials should assess the impact of changing the KRL code to use a ringbuffer instead of a single variable. By continuously keeping this buffer updated with the remainder of the trajectory, the KRC will have a larger horizon during interpolation, and should provide a smoother motion profile.

5.5 Force-Torque sensor

Even though the FTC sensor has a measuring frequency of 1kHz, we are limited to an update frequency of 83Hz due to the 12ms IPOC. It is also worth noting that the sensor has a movement area of $\pm 1.4mm$[76]. This is quite large compared to the repeatability of 0.1mm for the robot, and should be taken into consideration for any tasks requiring high positional accuracy. It is possible to read the position offset from the sensor, which could then be used as a TCP correction offset for RSI.

The previous owner had set up both KRCs to read data from the FTC during the SPS run. The FTC is connected over DeviceNet, and configured to cyclically output force and torque data. This data is then converted from INT to FLOAT in accordance with [76], and stored into a global composite variable, `FTC`. This consists of 6 floats, storing forces and torques for X, Y and Z direction.

For use with the KVP based ROS interface, a separate ROS node has been created. It reads the `FTC` variable from the KRC and uses ROS Control's Force-Torque sensor interface to publish it as a ROS WrenchStamped message.

For use with RSI, the `FTC` variable is linked to the RSI XML output. The ROS `kuka_rsi_hw_interface` has been expanded to publish this data using the Force-Torque sensor interface from ROS Control.

5.6 Tool changer

Release and locking of the tool is not considered time-critical, and the use of RSI to control it not deemed necessary. A ROS node using KVP has been created to control the tool changer. It is able to read the status of the tool changer, and can write to a flag for locking or unlocking the toolholder. This flag is read by a KRL program that is created on the controller. To ensure the tool is only released when it is safe to do so, workspaces surrounding the toolholders are created. The KRL program checks that the TCP is inside the workspace before it toggles the lock on the tool changer. The ROS node disconnects from KVP after each request, as locking and unlocking the tool changer is not a frequent event. The KRL program is run from the SPS.

5.7 Updates to ROS support packages

The support packages containing URDF and launch files have received some small updates to work correctly with ROS Kinetic. Visual and collision objects have been added for auxiliary equipment placed on the KR 16s. Both toolchanger and force-torque sensor have been added to the xacro files. For the toolchanger a 3D model from Schunk is used for both visual and collision models. Schunk did not have any 3D models of the FTC available. Physically the sensor appears as a cylinder with rounded edges. Using measurements from [76] both visual and collision objects have been added by means of a URDF cylinder.

The MoveIt! configuration package defines 4 move groups. There are one each for the two KR16, one for the gantry crane and attached KR16, and one for both KR16's and gantry crane. IKFast from OpenRAVE was used to generate analytic kinematic solvers for both KR 16 move groups. These solvers can be used by frameworks like MoveIt! and Descartes during path planning. As the MoveIt! IKFast generation step currently only supports 7 DOF, KDL is still used for the other move groups[77]. Further inquiries into an analytic solver for all 9 DOF on the gantry robot should be made.

5.8 Centralized code and documentation repository

All official ROS and ROS-I packages is hosted on GitHub. To ease integration of this code with necessary changes to use it in our laboratory, we need to maintain forked versions of some repositories, like the `kuka_rsi_hw_interface`. Rather than spread these repositories on numerous private accounts, a GitHub organization was created to host them. KRL files and custom configuration for the robots are placed in a separate git repository in this organization. This will allow us to track and document changes to the configuration over time. To make this organization a "one-stop service", the documentation for the laboratory was moved into a GitHub wiki.

To represent the 9 DOF available to the gantry robot, the laboratory was named after Þrívaldi, a 9 headed jötunn from Norse mythology. The GitHub organization uses the anglicized version of the name, and is located at `https://github.com/itk-thrivaldi`

5.9 Example packages

The `thrivaldi_examples`[1] git repository is intended to collect various example packages for using the laboratory with ROS. It currently houses some simple MoveIt! examples, and work is started on a package using Descartes for motion planning. These packages are intended to serve as a starting point that new users can turn to after completing the official ROS and ROS-I tutorials.

[1]`https://github.com/itk-thrivaldi/thrivaldi_examples`

Chapter 6

Conclusion and future work

Access time and movement latency for KVP with a C++ and Java library have been investigated. The findings from this investigation show a mean access time that is lower than the 12ms IPOC. This makes KVP a viable control method for the robots, and shows that a KRC 2 2005ed has comparable performance with the results on a KRC 4 from [63].

Using `ST_PATHCORR` to adjust TCP position along with external axes position, we have enabled the use of mathematically coupled external axes with RSI. This has allowed us to set up workspace monitoring, significantly improving lab safety.

With KVP as a control interface to the robot, we are able to increase safety by utilizing functions from the KRC, like velocity limits. That gives us the opportunity to allow bachelor and master students use the laboratory for learning and research where precise trajectory control is not needed. At the same time, we have the possibility to use RSI for research that requires precise trajectory control, or hard-realtime robot control.

Both the force-torque sensor and the tool change have been integrated with ROS, with visual and collision models added to the URDF model. By only allowing release of the tool when the toolchanger is near the tool holder we have prevented accidental release of the tool.

Areas identified for further work are

- Expand the KVP driver to use a ringbuffer on the KRC
- Analytic solver for 9 DOF

- Cartesian path planner with 9 DOF
- Expand documentation based on usage experiences
- Add visual and collision objects of toolholders to URDF
- Add force torque sensor to Gazebo model
- Create Gazebo world representing the laboratory. Could possibly be based on data from a 3D camera

With the rare setup of a 9 DOF robot hovering over the workspace of a 6 DOF robot, and in the hopes that it meets a better fate than Þrívaldi, who was slain by Thor, the laboratory is ready to tackle unique challenges.

References

[1] A. A. Transet, H. Schumann-Olsen, A. Røyrøy, and M. Galassi, in. Society of Petroleum Engineers, 2013, ch. Robotics for the Petroleum Industry - Challenges and Opportunities.

[2] E. Kyrkjebø, P. Liljebäck, and A. A. Transeth, "A robotic concept for remote inspection and maintenance on oil platforms," in *ASME 2009 28th International Conference on Ocean, Offshore and Arctic Engineering*, American Society of Mechanical Engineers, 2009, pp. 667–674.

[3] M. Bjerkeng, A. A. Transeth, K. Y. Pettersen, E. Kyrkjebø, and S. A. Fjerdingen, "Active camera control with obstacle avoidance for remote operations with industrial manipulators: Implementation and experimental results," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2011, pp. 247–254.

[4] I. Eriksen, "Setup and interfacing a kuka robotics lab with ros," graduate project, Department of Engineering Cybernetics, Norwegian University of Science and Technology, 2016.

[5] A. Montaqim. (2015). Top 14 industrial robot companies and how many robots they have around the world, roboticsandautomationnews.com, [Online]. Available: https://roboticsandautomationnews.com/2015/07/21/top-8-industrial-robot-companies-and-how-many-robots-they-have-around-the-world/812/ (visited on 12/07/2017).

[6] History | kuka ag, KUKA Aktiengesellschaft, [Online]. Available: https://www.kuka.com/en-us/about-kuka/history (visited on 10/19/2017).

[7] *Kr c2 edition 05, operating instructions*, version V3.3 11.07.2007 KRC-AD-KRC2ed05-BA en, KUKA Robot Group, Jul. 2007.

[8] *Spez kr 6, kr 16, kr 16 l6 de/en/fr*, version 06.2003.09, KUKA Roboter GmBH.

[9] *Technical data kr 16*, KUKA Roboter GmbH, Mar. 2004.

[10] *Kr c..., operator control, kuka system software (kss)*, version 03, KUKA Robot GmbH, Jul. 2003.

[11] J. Heilala, "Open real-time robotics control-pc hardware, windows/vxworks operating systems and communication," 2001.

[12] "Real-time windows - 30,000 robots can't be wrong!" In *Industrial Networking and Open Control*, vol. 9, Magpye Publishing Ltd., 2003.

[13] *Vxwin sales brochure*, KUKA Controls GmbH, 2005.

[14] *Kuka system software 5.2, 5.3, 5.4*, version 0.3, KUKA Robot Group, Feb. 2007.

[15] *KR C2 / KR C3 Expert Programming*, version ProgExperteBHR5.2 09.03.00 en, KUKA Roboter GmbH, Sep. 2003.

[16] *Kr c submit interpreter operation and programming*, version 01, KUKA Roboter GmbH, May 2005.

[17] *Kr c2 edition2005 specification*, version Spez KR C2 ed05 V5 en, KUKA Roboter GmBH, Oct. 2010.

[18] *Kr c2 / kr c3 configuration kuka system software (kss) release 5.2*, version 02, KUKA Roboter GmbH, Aug. 2005.

[19] *External Axes For KUKA System Software 5.5*, version KSS 5.5 Zusatzachsen V1 en, KUKA Roboter GmbH, Dec. 2008.

[20] *KUKA.RobotSensorInterface 2.3*, version KST RSI 2.3 V1 en, KUKA Roboter GmbH, May 2009.

[21] *KUKA.Ethernet RSI XML 1.1*, version KST Ethernet RSI XML 1.1 V1 en, KUKA Robot Group, Dec. 2007.

[22] *Kr c... error messages / troubleshooting*, version 01, KUKA Roboter GmbH, Oct. 2004.

[23] D. Caro, *Automation Network Selection: A Reference Manual.* International Society of Automation, 2009.

[24] *The new hire: How a new generation of robots is transforming manufacturing*, PricewaterhouseCoopers LLP, 2014.

[25] T. Foote, *Ros community metrics report*, `http://download.ros.org/downloads/metrics/metrics-report-2016-07.pdf`, Jul. 2016.

[26] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: An open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, 2009, p. 5.

[27] Topics - ROS Wiki, [Online]. Available: `http://wiki.ros.org/Topics` (visited on 12/15/2016).

[28] actionlib - ROS Wiki, [Online]. Available: `http://wiki.ros.org/actionlib` (visited on 12/15/2016).

[29] ros_control - ROS Wiki, [Online]. Available: `http://wiki.ros.org/ros_control` (visited on 12/15/2016).

[30] I. A. Sucan and S. Chitta. Moveit! [Online]. Available: `http://moveit.ros.org` (visited on 12/15/2016).

[31] Rqt - ros wiki, [Online]. Available: `http://wiki.ros.org/rqt` (visited on 10/23/2017).

[32] Rviz/displaytypes - ros wiki, [Online]. Available: `http://wiki.ros.org/rviz/DisplayTypes` (visited on 10/23/2017).

[33] Diagnostics - ros wiki, [Online]. Available: `http://wiki.ros.org/diagnostics` (visited on 10/23/2017).

[34] Why ros 2.0? [Online]. Available: `http://design.ros2.org/articles/why_ros2.html` (visited on 11/06/2017).

[35] Changes between ros 1 and ros 2, [Online]. Available: `http://design.ros2.org/articles/changes.html` (visited on 11/06/2017).

[36] Dds and ros middleware implementations, [Online]. Available: `https://github.com/ros2/ros2/wiki/DDS-and-ROS-middleware-implementations` (visited on 11/06/2017).

[37] Security - fast rtps 1.5.0 documentation, [Online]. Available: `http://docs.eprosima.com/en/latest/security.html` (visited on 11/06/2017).

[38] Releases · ros2/ros2 wiki, [Online]. Available: `https://github.com/ros2/ros2/wiki/Releases` (visited on 12/12/2017).

[39] Description - ROS-Industrial, [Online]. Available: `http://rosindustrial.org/about/description/` (visited on 12/15/2016).

[40] Robotic blending milestone 4 technology demonstration at wolf robotics — ros-industrial, [Online]. Available: `http://rosindustrial.org/news/2017/9/7/robotic-blending-milestone-4-technology-demonstration-at-wolf-robotics` (visited on 10/24/2017).

[41] Descartes - ros wiki, [Online]. Available: `http://wiki.ros.org/descartes` (visited on 12/02/2017).

[42] Integration of descartes for cartesian path planning · issue #467 · ros-planning/moveit, [Online]. Available: `https://github.com/ros-planning/moveit/issues/467` (visited on 12/02/2017).

[43] Industrial/supported_hardware - ros wiki, [Online]. Available: `http://wiki.ros.org/Industrial/supported_hardware` (visited on 12/02/2017).

[44] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, IEEE, vol. 3, pp. 2149–2154.

[45] Gazebo, [Online]. Available: `http://gazebosim.org` (visited on 11/07/2017).

[46] History | the orocos project, [Online]. Available: `http://orocos.org/content/history` (visited on 10/25/2017).

[47] Releases · orocos/orocos_kinematics_dynamics, [Online]. Available: `https://github.com/orocos/orocos_kinematics_dynamics/releases` (visited on 10/25/2017).

[48] Releases · orocos-toolchain/rtt, [Online]. Available: `https://github.com/orocos-toolchain/rtt/releases` (visited on 10/25/2017).

[49] Bfl moved to github | the orocos project, [Online]. Available: `http://orocos.org/orocos/bfl-moved-github` (visited on 10/25/2017).

[50] K. Gadeyne. (2001). Bfl: Bayesian Filtering Library, [Online]. Available: `http://www.orocos.org/bfl` (visited on 10/25/2017).

[51] Add documentation for eigen matrix library · issue #6 · toeklk/orocos-bayesian-filtering, [Online]. Available: `https://github.com/toeklk/orocos-bayesian-filtering/issues/6` (visited on 10/25/2017).

[52] User manual | the orocos project, [Online]. Available: `http://orocos.org/kdl/user-manual` (visited on 10/25/2017).

[53] Orocos_kdl - ros wiki, [Online]. Available: `http://wiki.ros.org/orocos_kdl` (visited on 10/25/2017).

[54] The orocos real-time toolkit | the orocos project, [Online]. Available: `http://orocos.org/rtt` (visited on 10/25/2017).

[55] The orocos component builderś manual, [Online]. Available: `http://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/orocos-components-manual.html` (visited on 10/25/2017).

[56] Code generator for components and type handling in rock - the robot construction kit - and the orocos toolchain, [Online]. Available: `https://github.com/orocos-toolchain/orogen` (visited on 10/25/2017).

[57] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: Yet another robot platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006.

[58] P. Fitzpatrick, E. Ceseracciu, D. E. Domenichelli, A. Paikan, G. Metta, and L. Natale, "A middle way for robotics middleware," *Journal of Software Engineering for Robotics*, vol. 5, no. 2, pp. 42–49, 2014.

[59] R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, vol. 79, 2008.

[60] R. Diankov, "Automated construction of robotic manipulation programs," PhD thesis, Carnegie Mellon University, Robotics Institute, Aug. 2010. [Online]. Available: http://www.programmingvision.com/rosen_diankov_thesis.pdf.

[61] Simulator for industrial robots and offline programming - robodk, [Online]. Available: https://robodk.com/ (visited on 10/24/2017).

[62] Flexgui 4.0 - welcome, [Online]. Available: https://www.ppm.no/flexgui4-Home (visited on 10/24/2017).

[63] F. Sanfilippo, L. I. Hatledal, H. Zhang, M. Fago, and K. Pettersen, "Jopenshowvar: An open-source cross-platform communication interface to kuka robots," in *Proc. of the IEEE International Conference on Information and Automation (ICIA), Hailar, China*, 2014, pp. 1154–1159.

[64] E. B. Njåstad, "Robotsveising med korreksjon fra 3d-kamera," Master's thesis, Department of Production and Quality Engineering, Norwegian University of Science and Technology, 2015.

[65] S. H. Bredvold, "Robotic welding of tubes with correction from 3d vision and force control," Master's thesis, Department of Production and Quality Engineering, Norwegian University of Science and Technology, 2016.

[66] Kuka robots - robodk documentation, [Online]. Available: https://robodk.com/doc/en/Robots-KUKA.html (visited on 12/07/2017).

[67] E. Idsø and Ø. M. Jakobsen, "Objekt-og informasjonssikkerhet, metode for risiko og sårbarhetsanalyse," *Institutt for produksjonsteknikk og kvalitetsteknikk. Norges teknisk-naturvitenskapelige universitet*, 2000.

[68] H. Bruyninckx, "Open robot control software: The orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, IEEE, vol. 3, 2001, pp. 2523–2528.

[69] The need for robotics standards | the construct, [Online]. Available: http://www.theconstructsim.com/need-robotics-standards/ (visited on 11/06/2017).

[70] T. Foote, *Ros community metrics report*, http://download.ros.org/downloads/metrics/metrics-report-2017-07.pdf, Jul. 2017.

[71] Distributions - ros wiki, [Online]. Available: `http://wiki.ros.org/Distributions` (visited on 11/10/2017).

[72] Automaticsecurityupdates - community help wiki, [Online]. Available: `https://help.ubuntu.com/community/AutomaticSecurityUpdates` (visited on 11/06/2017).

[73] Ubuntu 16.04 firewall, [Online]. Available: `https://help.ubuntu.com/lts/serverguide/firewall.html` (visited on 11/06/2017).

[74] Joint_trajectory_controller - ros wiki, [Online]. Available: `http://wiki.ros.org/joint_trajectory_controller` (visited on 12/02/2017).

[75] Concepts | moveit! [Online]. Available: `http://moveit.ros.org/documentation/concepts/` (visited on 12/02/2017).

[76] *Force-torque sensor type FTC / FTCL assembly and operating manual*, version 02/FTC/en/2009-12-28/SW, Schunk, 2008.

[77] Generate ikfast plugin tutorial - moveit_tutorials indigo documentation, [Online]. Available: `http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/ikfast_tutorial.html` (visited on 12/13/2017).