RAM ROBOTICS AND MECHATRONICS

User-input device based command and control of the youBot using a RaMstix embedded board

T.A. (Twan) Spil

MSc Report

Committee:
Prof.dr.ir. S. Stramigioli
Dr.ir. J.F. Broenink
Dr.ir. T.J.A. de Vries
Dr.ir. J. van Dijk

December 2016

051RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

Summary

The youBot is a mobile manipulator developed by KUKA to be used in research and education. The youBot consists of a base with four omniwheels and an arm with five joints. Reusable software was developed termed the *motion stack* for use with the youBot, but (mis)use of this software during several research projects resulted in broken gearboxes in the youBot.

A new method of control has been realized termed the *Inverse Kinematics with Kinematic Constraints (IKwKC) solver*, which replaces the motion stack. The IKwKC solver utilizes an analytic closed form Inverse Kinematic solver, but when the formulas of the IK solver calculate unreachable joint positions the Kinematic Constraints solver recalculates the IK formulas with different redundancy parameters such that reachable joint positions are calculated and the desired end effector position is reached. With the IKwKC solver the range of motion is almost doubled compared with the IK solver it is based on.

A connection between the youBot and the RaMstix is realized using EtherCAT such that the youBot can be controlled with the 20-sim 4C toolchain. Advantages of the connection between the youBot and RaMstix are: the motors in the joint can be commanded at PWM level, the youBot is controlled hard real-time and safety features are built in the toolchain. This connection has been realized successfully, all inputs and outputs of the youBot are accessible. The execution time of one cycle is on average > 150 us, but peaks every 30 cycles to 300 us. The cause is likely a result of the network stack which is not real-time. Nevertheless, the youBot can be controlled with a frequency of up to 2 kHz. The implementation of a real-time network communication, like RTNet, is recommended.

To showcase the toolchain and the new method of control a demo has been developed. For this demo the toolchain has been expanded to allow for set-point generation and communication. This allows the youBot to be controlled wirelessly via wifi with either an Xbox controller or a keyboard.

Contents

1 Introduction

1.1 Context

The youBot is a mobile manipulator developed by KUKA to be used in research and education, a picture of the youBot is shown in Figure 1.1. The youBot consists of a base with four omni-wheels and an arm with five joints. The base and manipulator can be used independently. The Robotics and Mechatronics (RaM) group acquired two youBots for the BRICS (Best Practice In Robotics) project and were used by Brodskiy (2014) to test reusable software and fault-tolerant control. The reusable software, termed the *motion stack*, controls the youBot with the use of an impedance controller (Hogan, 1984).

Figure 1.1: The KUKA youBot (Bischoff et al., 2011)

The motion stack was used to demo the youBot with the Phantom omni, a 6 dof haptic device. This demo was later upgraded by Frijnts (2014), in which the control of arm and base is separated. The base was made controllable with a joystick, the arm with an Omega 6. The Omega 6 is also a 6 dof haptic device, but it is more precise and offers larger force feedback than the Phantom omni. The most important safety feature in the motion stack, the passivity layer (Franken et al., 2009), was not properly updated. Even though additional safety features were added, gearboxes in both youBots broke down during several student projects (Weijers, 2015).

The conclusion can be drawn that the motion stack is not safe enough, even with the additional safety features. Also the tool chain that is used is not real-time, does not have a proper embedded control software structure and does not allow for rapid prototyping.

1.2 Goals of the project

To solve the problems stated in the previous section this project has the following goals

1. *New method of control.* Several students in succession broke down both youBots in the lab using the motion stack. Therefore the need for an easier, more robust and safer control method is clear.

2. *Connecting the youBot with the RaMstix.* The RaMstix is a platform developed at RaM used to run embedded control software. The aim of this connection is to control the

youBot hard real-time and use the 20-sim 4C toolchain to implement the controller. If 20-sim 4C would have been used during the previous student projects, the passivity layer could have been updated more easily and the youBots would not have broken down. This is conceived because 20-sim 4C allows easy monitoring of data and tweaking of parameters.

3. *Showcase the new control method and toolchain with a new demo*

1.3 Outline of the report

In Chapter 2 background material on: the youBot, EtherCAT, the ECS structure and the RaMstix is detailed. Any of these topics can be passed over if the reader has sufficient knowledge about them. In Chapter 3 the design and realization of the connection between the RaMstix and the youBot is detailed. In Chapter 4 a new control method for the youBot, called the Inverse Kinematic with Kinematic Constraint solver, is detailed. In Chapter 5 set-point generation and communication is detailed, which is integrated in the toolchain for usage in the youBot demo.

2 Background

2.1 The youBot

The youBot, shown in Figure 1.1, is a mobile manipulator devoped by KUKA with the goal of bridging the gaps between mobile manipulation research, application development for cognitive manufacturing and education in mobile manipulation (Bischoff et al., 2011), such that research in mobile manipulation is translated faster to industrial applications and to serve as a state-of-the-art example to introduce robotics education with actual research problems.

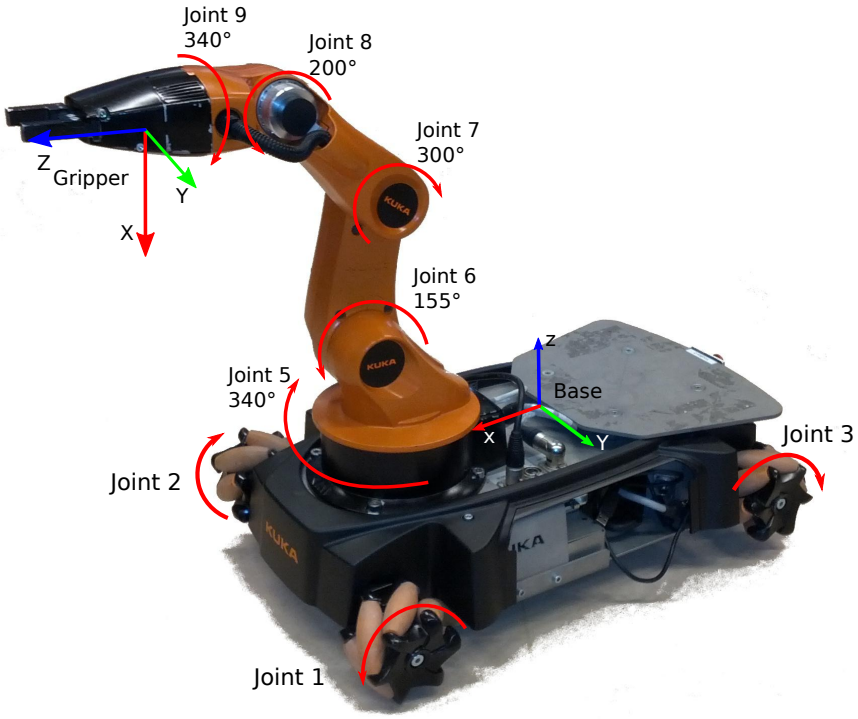

Figure 2.1: The KUKA youBot, with the positive direction of the motors indicated by the arrows (Frijnts, 2014)

2.1.1 Technical details

The youBot consists of an omnidirectional base, an arm with five Degrees Of Freedom (DOF) and a two-finger gripper. The omnidirectional base has a payload of 20 kg and is driven by four omnidirectional wheels. The wheels and arm joints are driven by brushless DC motors (also known as EC motors) with built-in gearbox, relative encoder and joint bearing. The five arm joints have mechanical end-stops. Together in the arm they can lift 0.5 kg using the gripper. Both fingers of the gripper are opened and closed with one stepper motor. All motors (4 wheels, 5 arm joints, 2 fingers) can be controlled using EtherCAT. The lowest level of command being pulse width modulation (PWM). Higher-level control commands are: current control, velocity control and position control. In Figure 2.2 these commands are visualized. In the youBot driver there is no access to pwm control. The base also contains a computer running Ubuntu. The main characteristics of the youBot are summarized in Table 2.1 and 2.2

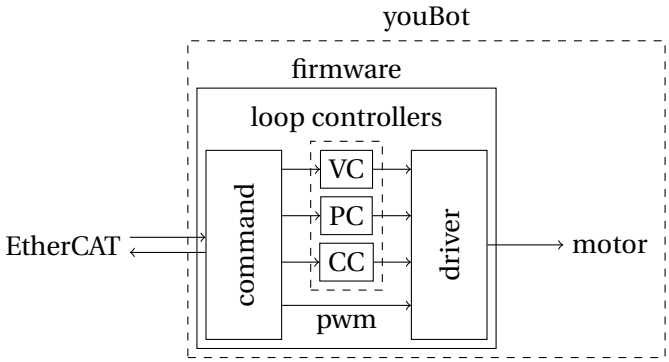

Figure 2.2: Ways of controlling a motor using ethercat. VC signifies velocity control, PC signifies position control and CC signifies current control

Serial kinematics	5 axes	
Height	655 mm	
Work envelope	0.513 m^3	
Weight	6.3 kg	
Payload	0.5 kg	
Structure	Magnesium cast	
Positioning repeatability	1 mm	
Communication	EtherCAT	
Voltage connection	24 V	
Drive train power limitable to	80 W	
Joint data	Range	Speed
Joint 5	$+/-169°$	90°/s
Joint 6	$+90°/-65°$	90°/s
Joint 7	$+146°/-151°$	90°/s
Joint 8	$+/-102°$	90°/s
Joint 9	$+/-167°$	90°/s
Gripper	Detachable, 2 fingers	
Gripper stroke	20 mm	
Gripper range	70 mm	

Table 2.1: General characteristics KUKA youBot arm (Bischoff et al., 2011)

Omnidirectional kinematics	4 KUKA omniWheels
Length	580 mm
Width	380 mm
Height	140 mm
Clearance	20 mm
Weight	20 kg
Payload	20 kg
Structure	Steel
Speed	0.8 m/s
Communication	EtherCAT
Voltage connection	24 V

Table 2.2: General characteristics KUKA youBot base (Bischoff et al., 2011)

2.1.2 youBot driver

The youBot driver is used to control the youBot, it is made up of two parts, the EtherCAT driver and the youBot Application Programming Interface (API). The EtherCAT driver utilizes an open source EtherCAT master called Simple Open EtherCAT Master (SOEM) (Karlsson, 2014).

The youBot API (KUKA, 2015) is an open source c++ library developed by the Autonomous Systems Group at Bonn-Rhein-Sieg University of Applied Sciences. The architecture of the youBot API is shown in Figure 2.3. The arrows represent flow of information. The communication to the youBot happens on a different thread called the communication thread, which is initialized when the first youBot joint is constructed. The rest of the youBot API runs on the same thread as the user application, therefore called the user thread. The idea behind the youBot API architecture is to represent the youBot as a combination of decoupled functional subsystems. After initialization of the YouBotBase or YouBotManipulator several YouBotJoints are initialized, but it is also possible to initialize a single YouBotJoint.

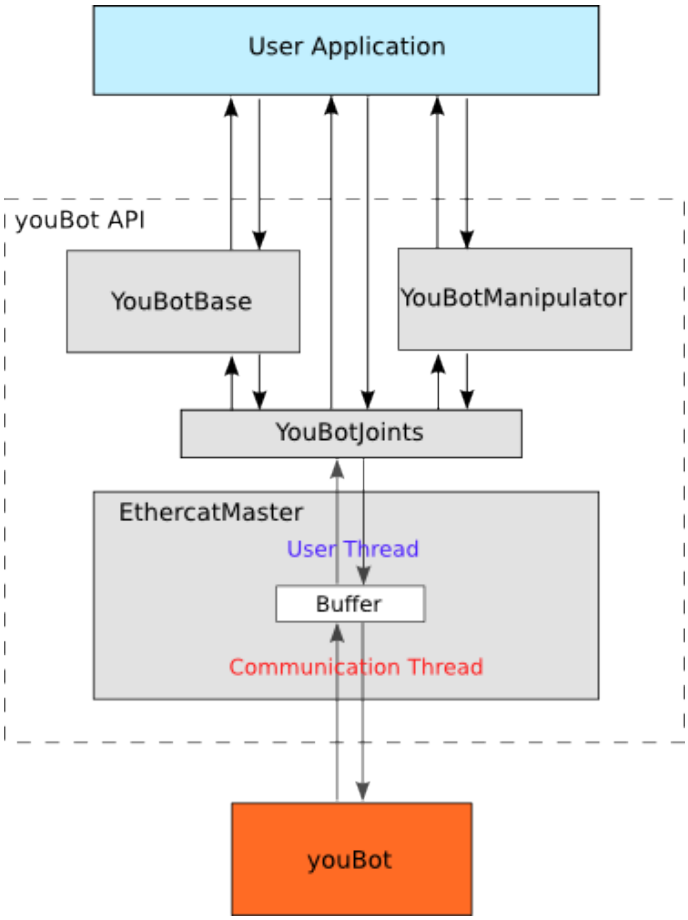

Figure 2.3: Architecture of the youBot API (KUKA, 2015)

In Figure 2.4 the state machine for the youBot Base and Manipulator class is shown. The main difference between the two is an extra state in the YouBotManipulator class which is reached by calibrating the manipulator. Calibration needs to be done before the joints can be controlled with the setData() function, because the encoders are relative. During calibration the joints are commanded to move toward their end-stop position at very low speed. By measuring the current applied by the motor it can be concluded when the joint reached its end-stop and the relative encoder is reset, such that the end-stop is the zero position of the encoder.

The doJointCommutation() function initializes the sine commutation by energizing the motor, moving the winding in the motor to the closest magnetic pole, this is called the stepper technique.

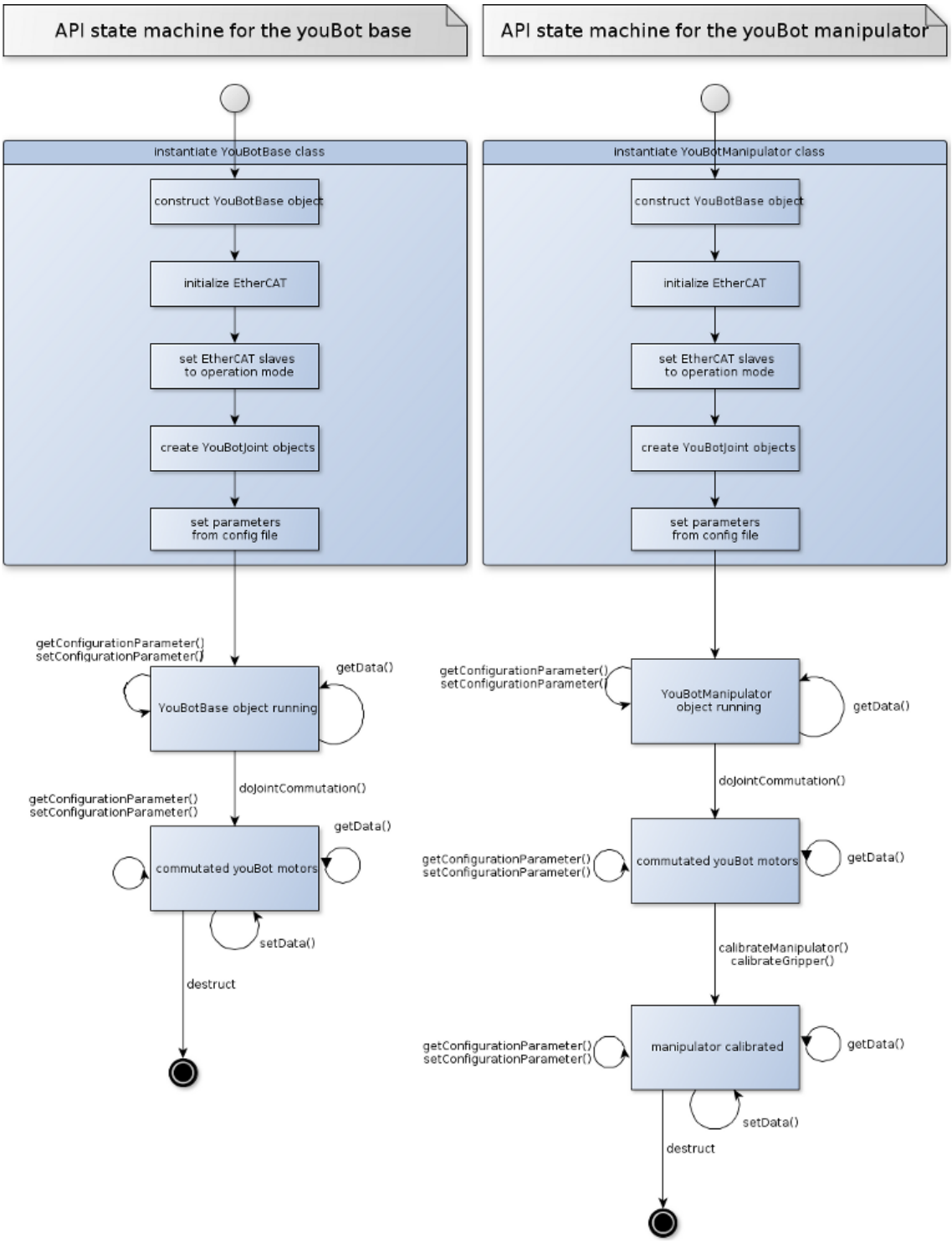

Figure 2.4: The state machine for the youBot Base and Manipulator class (KUKA, 2015)

2.1.3 Former youBot demo

In the former demo the youBot is teleoperated by an haptic device. The software architecture for this demo is developed by Brodskiy (2014) where it is called the motion stack. The motion

stack is shown in Figure 2.5 and is developed with the tool 20-sim. The *Cartesian controller* calculates the wrench to be applied to the gripper given an Cartesian set-point for the gripper. The wrench is calculated with the use of an impedance controller, which uses a virtual spring. The *youBot kinematics* component transforms the wrench on the tool tip to joint torques using forward kinematics. The *arm pose controller* component implements two safety checks, end-stop protection and network protection by use of the passivity layer. In the *base pose controller* component the three torques (two translations, one rotation) for the base are mapped to the four wheels.

The *youBot model* is developed by Dresscher et al. (2010) using bond graphs. The youBot arm is modelled as a series of links and joints. The link being a rigid body affected by gravity and the joint an actuator that has a controllable torque around one rotation. The base is modelled by its four wheels and a connection with the floor. The wheels are modelled as a series of TF-elements that transforms the energy of the actuator to the floor. The *youBot model* is used as a plant to test the motion stack, but also for gravity compensation in the *arm pose controller*.

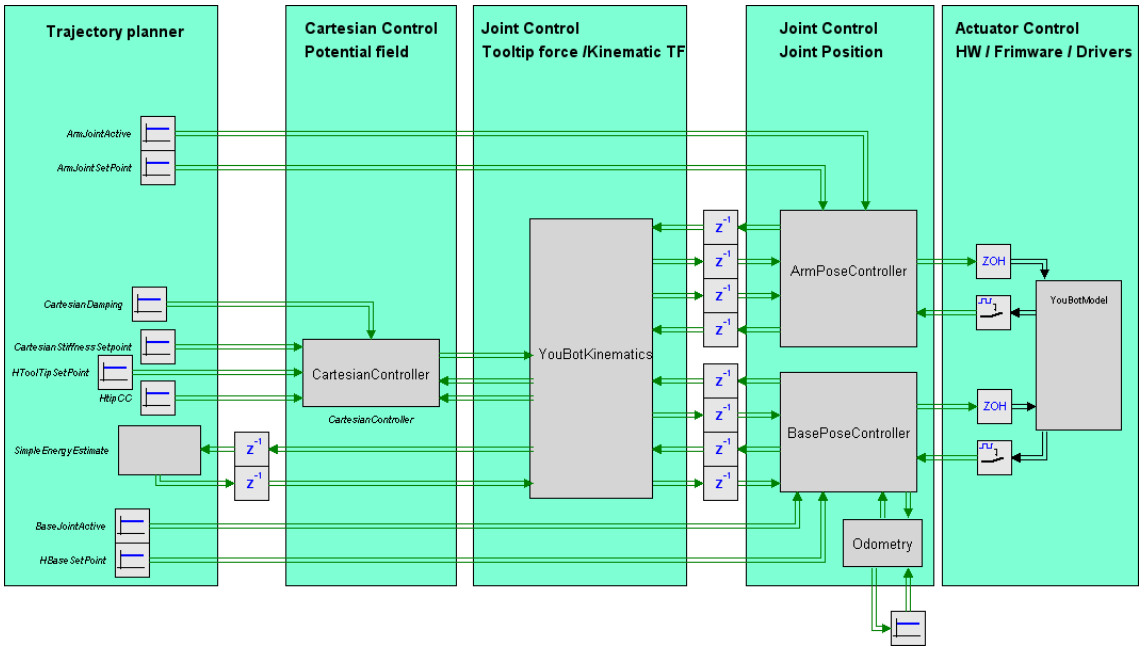

Figure 2.5: Architecture of the motion stack (Brodskiy, 2014)

Teleoperation of the youbot is achieved using multiple frameworks. Figure 2.6 shows the multiple frameworks and how they work together. The set-point generator on the remote computer generates the values that match the trajectory planning in the motion stack. This Figure shows the software framework for the updated demo developed by Frijnts (2014). In this demo the base is controlled by the joystick and the arm by the haptic device, this is done to have a greater range of motion then there was with only the haptic device.

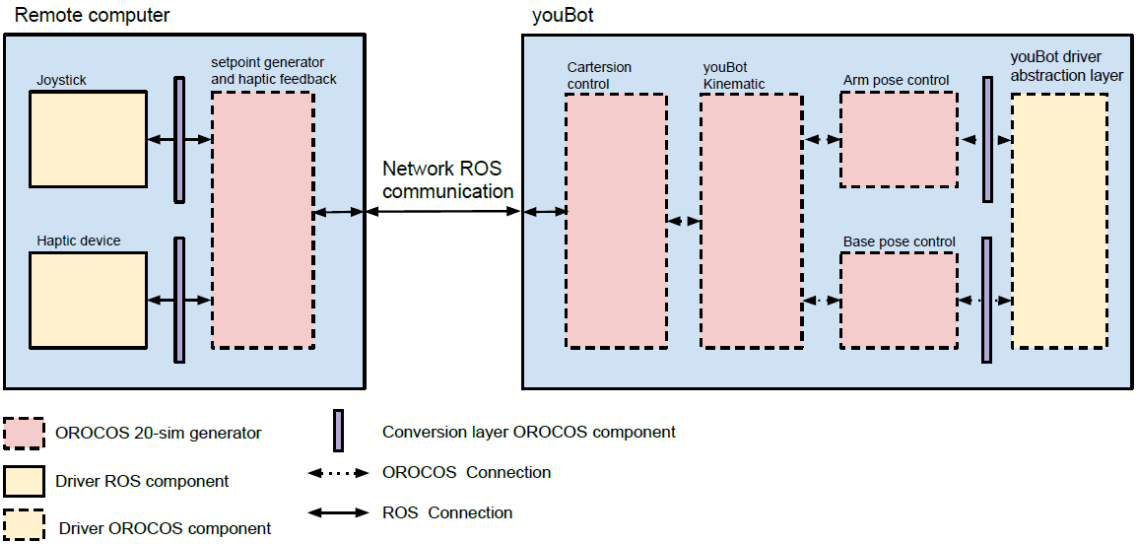

Figure 2.6: Software framework of the youBot demo (Frijnts, 2014)

2.2 EtherCAT

EtherCAT (Ethernet for Control Automation Technology) is an Ethernet-based fieldbus system that can achieve hard real-time requirements. The various goals of EtherCAT are (Jansen and Buttner, 2004)

- *broad applicability*, any pc with an Ethernet port can be an EtherCAT master.
- *full conformity with the Ethernet standard*, EtherCAT can coexist with other Ethernet devices and protocols on the same bus.
- *smallest possible device granularity without having to use a sub bus*, any type of I/O node, from 2 bit I/O to a complex node, can be used economically as an EtherCAT slave.
- *maximum efficiency*, as much of the Ethernet bandwidth as possible should be available for user data.
- *short cycle times*, cycle times significantly less than 100 μs
- *maximum deterministic properties*, even without the basis of absolute telegram transfer accuracy

2.2.1 Operating principle

From an Ethernet viewpoint, an EtherCAT bus can be seen as a single Ethernet device that sends and receives Ethernet telegrams. But instead of an Ethernet controller with downstream microprocessors, the 'device' contains EtherCAT slaves which insert and/or extract relevant I/O data and sent the telegram to the next slave along the Tx line. The last slave sends the fully processed telegram back to the master via the Rx line. Different topologies are easily realized with EtherCAT as shown in Figure 2.7. This topology is achieved due to the design of an EtherCAT slave, shown in Figure 2.8. One single EtherCAT slave can branch out to three other EtherCAT slaves further along the path.

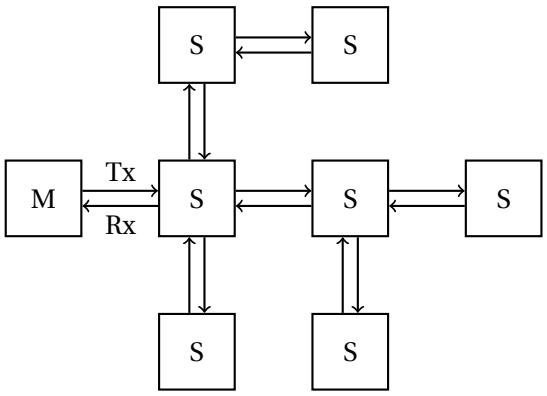

Figure 2.7: A possible EtherCAT topology. M stands for master, S stands for slave

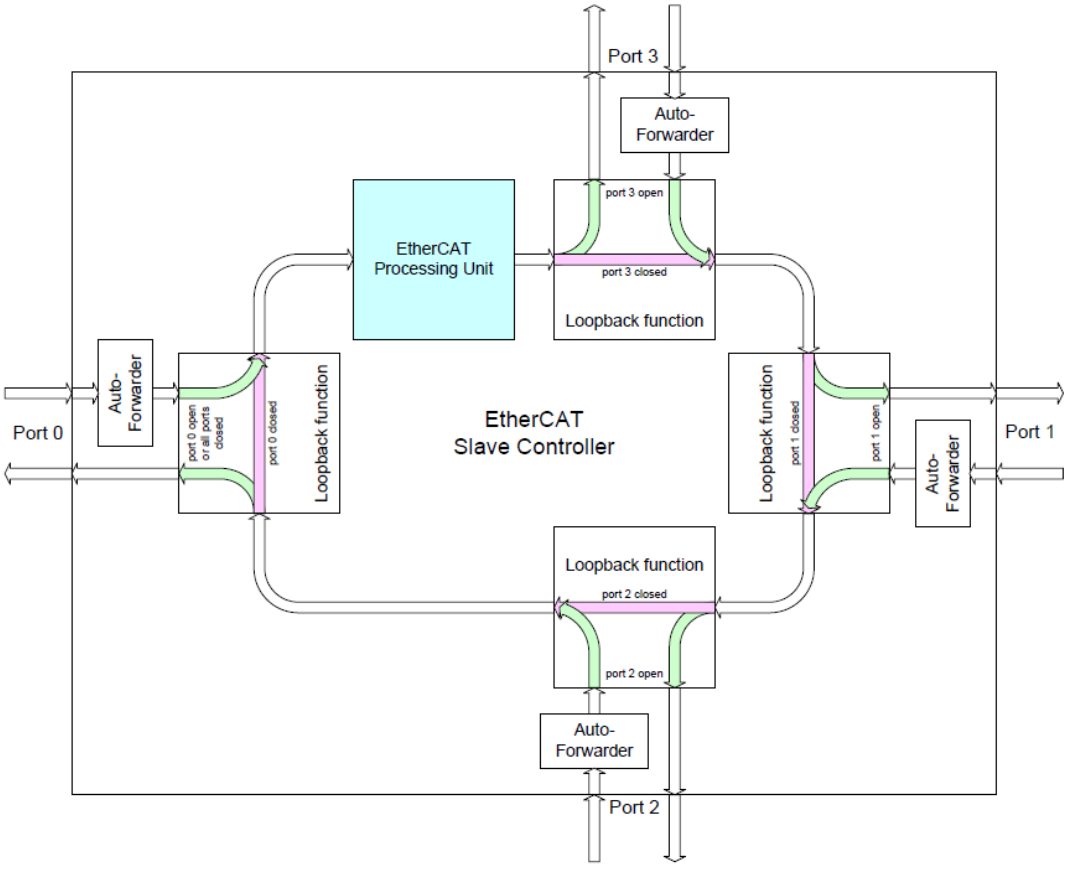

Figure 2.8: The way a telegram can travel through an EtherCAT slave (Beckhoff, 2014)

2.2.2 State machine

An EtherCAT slave operates a state machine to indicate what functionalities are available. The state machine is shown in Figure 2.9. The master is responsible for a state change, but the slave only changes state when all services required for the requested state are provided or stopped. An overview of the states is given in Table 2.3. A mailbox message transfers acyclic parameter or diagnostic data, which can be much bigger than a single process data telegram. Process data communication only transfers the required data for the process and is cyclic.

State	Functionalities
INIT	Init state. No communication on the application layer is available. The master has access only to the DL-information registers.
PREOP	Pre-Operational state. Mailbox communication on the application layer available, but no process data communication available.
SAFEOP	Safe-Operational state. Mailbox communication on the application layer, process (input) data communication available. In SafeOp only inputs are evaluated; outputs are kept in "safe" state.
OP	Operational state. Process data inputs and outputs are valid.
BOOT	Bootstrap state. Optional but recommended if firmware updates necessary. No process data communication. Communication only via mailbox on Application Layer available. Special mailbox configuration is possible, e.g. larger mailbox size. In this state usually the FoE protocol is used for firmware download.

Table 2.3: EtherCAT state descriptions (ETG, 2014)

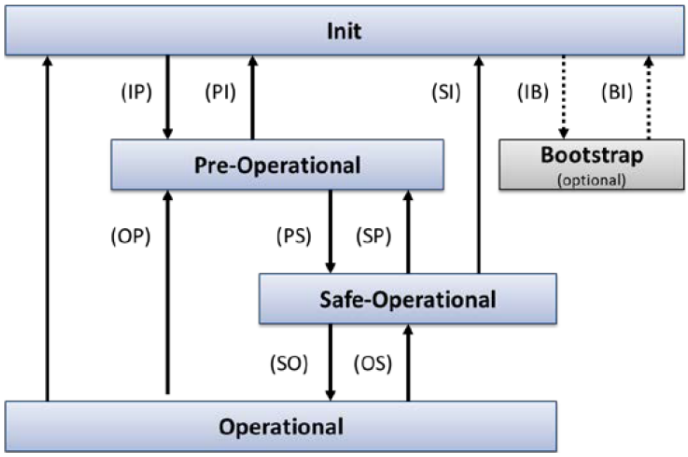

Figure 2.9: EtherCAT state machine (ETG, 2014). The arrows show the possible steps between states, arrows are named as from to state

2.2.3 Telegram processing

Telegrams are processed "on the fly", that means that a telegram passes directly to the next slave and only relevant information (i.e. commands, encoder values) is copied from and inserted into the telegram by the hardware of the slave and is therefore independent of response time of a microprocessor. The EtherCAT data is carried in standard IEEE 802.3 Ethernet fames with a reserved Ether type that distinguishes it from other Ethernet frames which makes EtherCAT able to run in parallel with other Ethernet protocols. EtherCAT data can also be encapsulated by UDP/IP if normal IP routing is needed the encapsulated protocol can be seen in the bottom half of Figure 2.10.

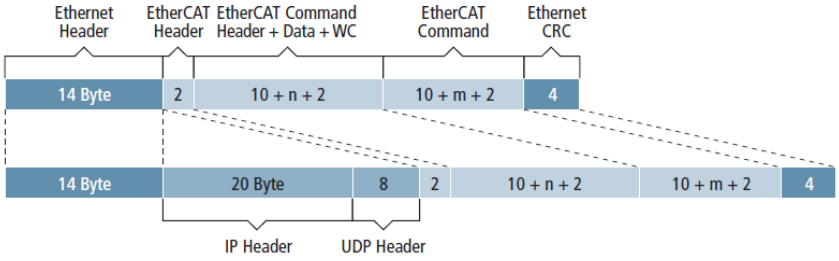

Figure 2.10: EtherCAT telegram structure (Jansen and Buttner, 2004). WC stands for working counter

2.2.4 Protection measures

EtherCAT checks whether a telegram was transmitted correctly, this is done using the standard Ethernet checksum found at the end of an Ethernet telegram. Since multiple slaves can modify the telegram during its transfer, the checksum is recalculated for every slave. If a checksum error is detected, a status bit is set in the EtherCAT slave, so that a fault can be located precisely. EtherCAT also checks whether a telegram was processed correctly using a working counter (WC). When an EtherCAT slave reads or writes data in an EtherCAT command, the working counter of that command is incremented. Since the master knows how many slaves are addressed by the telegram, it can detect form the working counter whether all slaves have exchanged their data correctly.

2.2.5 Performance

In Prytz (2008) the performance of EtherCAT and of another real-time Ethernet protocol, PROFINET IRT, are calculated and compared. EtherCAT is for every configuration (number of slaves, telegram size, link speed) faster than PROFINET. For a small number of slaves the minimum cycle time of EtherCAT is within 100μs.

2.3 Embedded control software structure

At the Robotics and Mechatronics (RaM) group, a layered architecture is used for the design of embedded control software (ECS) Broenink et al. (2010) as is shown in Figure 2.11. With layers for: *user interfacing*, *supervisory control* (Planning algorithm or image processing), *sequence control* (order of action), *loop control* (control of actuation), *safety* (safe inputs/outputs around all layers) and *measurement and actuation* (interface with hardware). The ECS is a combination of time-triggered parts (e.g. Loop control) and event-driven parts (e.g. UI) where multiple layers have different (real-)time requirements, which differ from system to system.

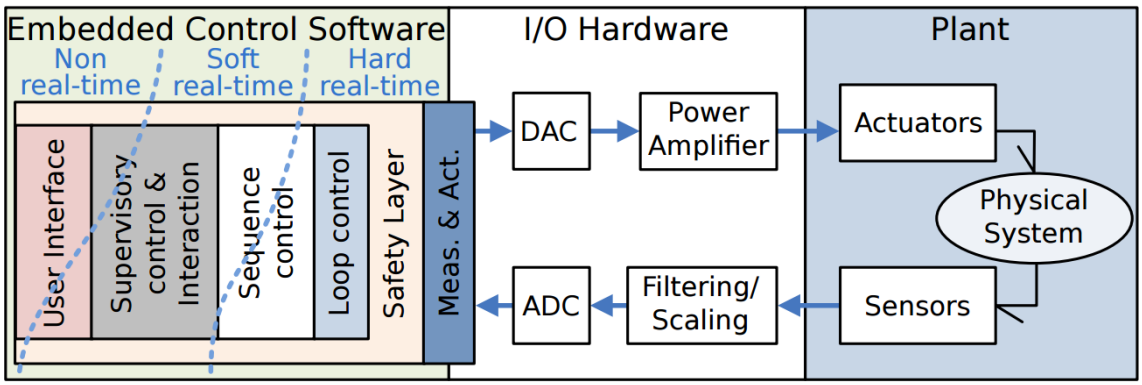

Figure 2.11: Layered architecture for embedded control software (Broenink et al., 2010)

2.4 RaMstix

The RaMstix is a platform used to run embedded control software at the RAM group, the RaMstix is shown in Figure 2.12. It is an expansion board for a processing unit called the Gumstix Overo module. The Gumstix runs on Linux Xenomai for hard real-time capabilities. The Overo module is connected to a Field-Programmable Gate Array (FPGA), the Altera Cyclone III, with a General Purpose Memory Controller (GPMC). This combination offers the following I/O possibilities.

- USB/serial debug interface
- 100 Mbit Ethernet connector
- USB master
- USB slave
- 4 dedicated Encoder inputs
- 4 dedicated PWM/Stepper outputs
- 16 digital output pins
- 16 digital input pins
- CAN bus interface
- 2 16-bit ADC
- 2 16-bit DAC

The RaMstix is also used to rapidly test developed software on the setup using the tool 20-sim 4C which allows a controller (or any other model) designed in 20-sim to be uploaded to the RaMstix in a few simple steps. While the controller is running the inputs, outputs and other variables can be monitored using 4C.

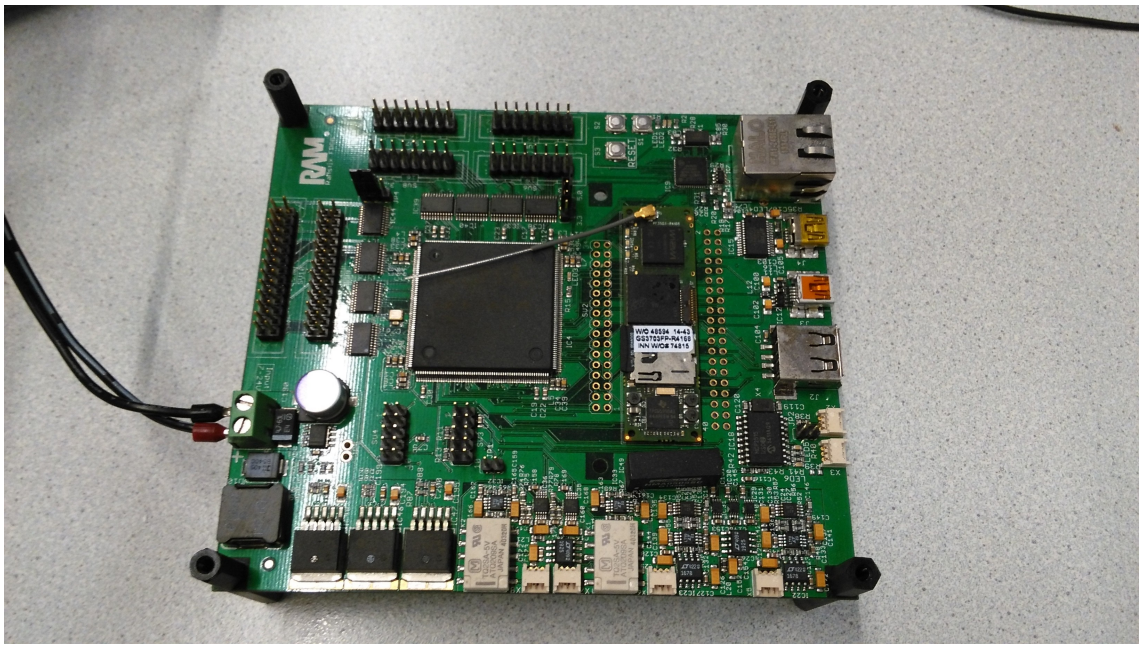

Figure 2.12: Picture of the RaMstix

3 Controlling the youBot with the RaMstix

In this chapter the development of the connection between the RaMstix and the youbot is detailed.

3.1 Design

3.1.1 Possible ways to control the youBot

Two ways of controlling the motors of the youBot using the RaMstix have been investigated:

- *Hooking up the motors directly to the RaMstix.* Every motor in the youbot has twelve pins, three to control the windings in the motor, five to read out the hall sensors and four to read out the encoder. To connect all these pins to the RaMstix the board should be expanded, the FPGA reprogrammed and the 20-sim 4C interface to the GPMC software rewritten. Another part of work would be to control the three windings in the motor such that a steady motion is achieved. A disadvantage of connecting the motors directly to the RaMstix is that there will be a lot of cables running between the RAMstix and the motors.
- *Controlling the motors with EtherCAT.* To control the youBot motors with the RaMstix an EtherCAT master needs to run on the operating system, Linux Xenomai, of the RAMstix. Also an interface between the EtherCAT master and 20-sim 4C needs to be written.

Following the discussion above, the second way is favourable. It is less time consuming to implement for almost the same results. Both ways are able to handle hard real-time constraints on high frequencies. Both ways can also control the motor directly, the first way by energizing the windings and the second way by PWM. Therefore if an EtherCAT master runs on the RAMstix the youBot will be controlled with the RAMstix using EtherCAT.

3.1.2 EtherCAT master

There are two open-source EtherCAT masters that work on Linux Xenomai:

- *Simple Open EtherCAT Master (SOEM).* SOEM is used in the youBot API and is also developed with embedded systems in mind.
- *IgH EtherCAT master.* The IgH EhterCAT master has drivers available for specific commonly used network interface cards (NIC) that communicate directly with the EtherCAT master such that it circumvents the linux network stack for better performance and guaranteed real-time performance. However the RAMstix has a NIC specially developed for embedded systems, the SMSC911x, which is not supported by the IgH Ethercat master. Therefore the IgH master uses the linux network stack to communicate with the EtherCAT slaves just like SOEM.

Both EtherCAT masters work on Xenomai, but call the linux network stack, therefore they don't have guaranteed hard real-time performance. SOEM is chosen to communicate with the youBot, because it is light weight and the youBot API can be used to understand and use SOEM in conjunction with the youBot.

3.1.3 Safety features

Two necessary safety features for the control of the youBot are identified:

- *End-stop protection*, protect the youBot arm joints from colliding with the end-stop.
- *Intrinsic safety*, protect the youBot arm from colliding with itself or its base.

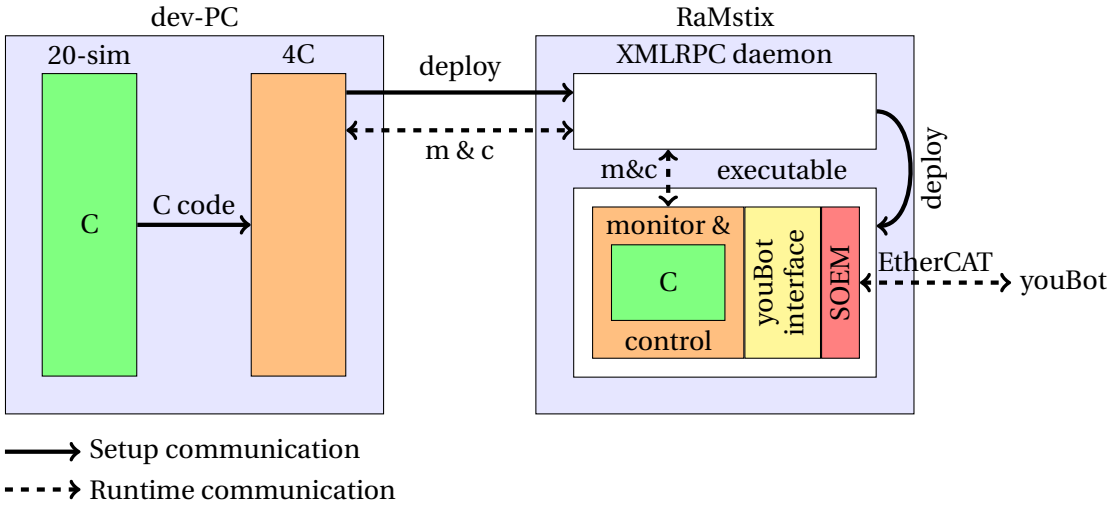

Figure 3.1: Graphical representation of the design of how the youBot is controlled using the RaMstix

Following the Embedded Control Software (ECS) structure, the safety features should reside between the loop control and measuring and actuation. Following this approach, end-stop protection is implemented in the youBot interface. Self protection can also be implemented in the youBot interface, but it is difficult to achieve on joint level. Therefore self protection is realized in chapter 4 in Cartesian space, where it is easier to realize.

It is chosen to develop end-stop protection only for the PWM control mode. The reason being that it is the most direct control of the motors and therefore should be used primarily to control the youBot. End-stop protection should be simple, such that it is fast, does not add delay and can be easily implemented in C.

3.1.4 Final design

In Figure 3.1 a graphical representation of the design is shown. On a development PC (running windows) a 20-sim model is created by the end-user that has to control the youBot. Using 20-sim's real-time toolbox this model is converted to C code and transferred to 20-sim 4C. In 4C the code of the model is encapsulated by code that manages the monitor & control, the real-time loop and the IO. Then the code is compiled to an executable and transferred to the RAMstix, through the XMLRPC daemon. This daemon is responsible for the communication between 20-sim 4C and the deployed controller. On the RAMstix the executable is deployed and the model can be started, stopped and monitored by 4C.

3.2 Realization

To realize the design an interface to the youBot using SOEM needs to be written and a RaMstix target with youBot interface needs to be set-up.

3.2.1 youBot interface

The interface to the youBot is written in C. Important functions are described in this section in pseudocode. The first function that needs to be run initializes Ethercat, this function is described in Algorithm 3.1. The functions *ec_init*() and *ec_config*() are from the SOEM library and initialize EtherCAT and configure EtherCAT respectively. More information on SOEM and the functions called in the algorithms can be found in Appendix A. In line 9 and 11 the pointers to the inputs and outputs in the IOmap are coupled to pointers to structures. This is possible because the RaMstix can handle non-aligned pointers to the IOmap. Otherwise values to be

read and written had to be done so bit wise. The structures that are coupled to the IOmap are listed in Listing 3.1.

```
struct {
        int32 value;
        uint8 controllermode;
} SlaveMessageOutput;

struct {
        int32 actualPosition;    // encoder ticks
        int32 actualCurrent;     // mA
        int32 actualVelocity;    // rpm motor axis
        uint32 errorFlags;
        int32 targetPosition;
        int32 targetCurrent;
        int32 targetVelocity;
        int32 rampGeneratorVelocity;
} SlaveMessageInput;
```

Listing 3.1: Structure of the process data

Algorithm 3.1 Initialize EtherCAT

Require: EtherCAT not initialized
if *ec_init*(IFName) **then**
  if *ec_config*(TRUE,&IOmap) **then**
    Check if all slaves reached SAFEOP state
    Request OP state for all slaves and check if they reached OP state
  end if
end if
for Each slave **do**
  if slave is TMCM-174 **then**
    Connect struct pointers of base I/O buffer to slave I/O pointers
  else if slave is TMCM-KR-841 **then**
    Connect struct pointers of manipulator I/O buffer to slave I/O pointers
  end if
end for
if four base slaves detected **then**
  BASE is **true**
end if
if five manipulator slaves detected **then**
  MANIPULATOR is **true**
end if

After running the Initialize EtherCAT function, full communication with the youBot slaves is possible. But before the motors in the joints can be controlled they need to be commutated, the function that handles this is described in Algorithm 3.2. During this process the motors move, for the joints in the base this is not a problem. But for joints in the manipulator it is because the encoder is relative and the end-stops are its zero positions. After the movement the manipulator joints are no longer in their end-stop position and need to be moved back. Therefore the initialization of the manipulator is a longer process, which is described in Algorithm 3.3.

Algorithm 3.2 Initialize Base

Require: BASE is **true**

```
1  for Each base slave do
2      Send mailbox message, clear motor controller time out flag
3  end for
4  Fill process data buffer of each base slave, with controller mode initialize
5  repeat
6      Send process data buffer
7      Receive process data buffer
8  until Each base slaves motor is commutated
9  for Each base slave do
10     Send mailbox message, reset encoder
11 end for
```

Algorithm 3.3 Initialize Manipulator

Require: MANIPULATOR is **true**

```
1  Same as Algorithm 3.2 until line 8
2  Fill process data buffer of each manipulator slave, with velocity control at two RPM
3  while Every joint is not in end-stop position do
4      Send process data buffer
5      Receive process data buffer
6      for Each manipulator slave do
7          if Exercised current of slave exceeds threshold then
8              Joint reached end-stop position
9              Fill process data buffer of slave, with current control of zero current
10         end if
11     end for
12 end while
13 for Each manipulator slave do
14     Send mailbox message, reset encoder
15 end for
```

array number	outgoing mailbox message	incoming mailbox message
0	module address	reply address
1	command number	module address
2	type number	status
3	motor number	command number
4	value (31-24 bit)	value (31-24 bit)
5	value (23-16 bit)	value (23-16 bit)
6	value (15-8 bit)	value (15-8 bit)
7	value (7-0 bit)	value (7-0 bit)

Table 3.1: Content of the mailbox message arrays from youBot slaves

In Algorithm 3.2 and 3.3 mailbox messages are sent. The process how these are sent is described in Algorithm 3.4. The structure of the mailbox messages is shown in Table 3.1. The position of the youBot gripper is controlled by use of mailbox messages, the desired position is sent in encoder counts.

Algorithm 3.4 Send mailbox message

Require: Mailbox message to send **and** number of addressed slave
1 Order mailbox message in the right bit order
2 *ec_mbxsend*(SlaveNumber,MailboxMessageSend,TimeOut)
3 **repeat**
4 *ec_mbxreceive*(SlaveNumber,MailboxMessageReceive,TimeOut)
5 Reorder received mailbox message in right bit order
6 **until** Confirmation of sent message being received

The final functions implemented in the youBot interface are:

- *Get and set functions for the I/O of the youBot.* Every input of the youBot –PWM, position set-point, velocity set-point and current set-point –can be set using their own set function. Every output of the youBot –current, position and velocity –can be accessed using the appropriate get function.
- *Send youBot data.* The I/O data is sent to the youBot using the function *ec_send_processdata*().
- *Receive youBot data.* The I/O data is received from the youBot using the function *ec_receive_processdata*().

3.2.2 20-sim 4C target

The RaMstix is already a target in 20-sim 4C. To add additional I/O the .tcf file has to be expanded. In the .tcf file all the I/O needs to be documented, one input is shown in Listing 3.2. By adding more ports the encoder data of the rest of the base can be accessed. By adding additional components with the relevant get function other inputs can be accessed. Outputs are set with the following line *setPWMBase(% VAR% ,% CHANNEL %)* in a component, by adding those components to the output section of the .tcf the I/O is linked. The executable that is compiled from the 4C target has its process described in Algorithm 3.5.

```
<COMPONENT>
        <NAME>Base encoders</NAME>
        <DESCRIPTION>
                <![CDATA[Base encoders of the youbot
                ]]>
```

```
        </DESCRIPTION>
        <VARIABLES />
        <INIT />
        <FUNCTION>
                <![CDATA[
                %VAR% = getEncoderBase(%CHANNEL%);
                ]]>
        </FUNCTION>
        <PORTS>
                <PORT>
                        <NAME>Encoder joint 1</NAME>
                        <PINS>
                                <PIN>NoPinEncJ1</PIN>
                        </PINS>
                </PORT>
        </PORTS>
</COMPONENT>
```

Listing 3.2: Piece of the .tcf file

Algorithm 3.5 20-sim 4C control loop

```
1  Initialize EtherCAT
2  Initialize base and manipulator
3  repeat
4     Receive youBot data
5     Access youBot data
6     Set youBot data
7     Send youBot data
8     Monitor and /or Log data
9     Calculate model outputs using youBot data
10    Wait for next period
11 until Time ≥ Finish time or Stop is pressed
12 Return manipulator to rest position
13 Stop motors
```

3.2.3 End-stop protection

End-stop protection is implemented with two P controllers that have their set-points as the end-stops of the joint and the process variable is the joint position in encoder counts, with the following formula.

$$PWM_{\text{min/max}} = K_p * (SP_{\text{min/max}} - PV)$$

The output of these P controllers is compared with the desired PWM. If the desired PWM exceeds the calculated minimum or maximum PWM, the desired PWM is limited to the calculated min/max.

End-stop protection is implemented in 20-sim to determine the gain (K_p) and end-stop min/max values ($SP_{\text{min/max}}$) empirically. The gain is determined by going through a range of values and selecting the value with the best response in terms of steady state error and overshoot. The SP values are determined such that the end-stop protection barely avoids hitting the end-stop when the joint is at max speed. The determined values can be found in table 3.2 and are used in the realization of the end-stop protection in the youBot interface.

Joint	K_p	$SP_{\min}$	$SP_{\max}$
5	0.002	-580000	-1000
6	0.0012	-253000	-7000
7	0.0013	-308000	-11000
8	0.0015	-155000	-1000
9	0.0025	-255000	-1000

Table 3.2: Values used in end-stop protection

3.3 Tests

Four tests are performed. The first test was designed during realization to evaluate the youBot interface. A control loop was implemented using the functions in the interface. The youBot was successfully controlled using this control loop.

The other three tests are designed to test various features of the youBot using the 20-sim 4C target implemented above.

- *I/O test*, tests the inputs and outputs of the youBot
- *Performance test*, tests the performance of the 20-sim 4C control loop with the youBot
- *End-stop protection test*, tests the end-stop protection

3.3.1 I/O test

The input/output (I/O) test is designed to assess the various inputs of the youBot and see if the outputs are working properly. During the I/O test the manipulator arm is moved to the extended arm position by controlling the encoder position using every controller mode that is possible for the youbot.

With the positioning controller mode no control loop is implemented, because the EtherCAT slave controls the encoder position of the joint in this control mode. The joints are commanded to the extended arm position with constant blocks, the position of the joints for the extended arm is given in Table 3.3.

For the other controller modes– velocity, current and PWM– a control loop is implemented. The position of each joint is controlled with a P controller. The P controller is chosen to clearly see the behaviour of each mode. The gain values of the controllers are determined empirically and are given in Table 3.3.

	Position (cnts)	K_{vel}	K_{cur}	K_{PWM}
Joint 5	-290000	0.02	0.003	0.002
Joint 6	-110000	0.03	0.003	0.0015
Joint 7	-150000	0.025	0.003	0.002
Joint 8	-75000	0.035	0.003	0.004
Joint 9	-132000	0.025	0.003	0.005

Table 3.3: Position per joint to reach extended arm position and proportional gains used in the controllers for the different modes of control: velocity, current and PWM

Results

The recorded outputs–position, velocity and current–of the arm extension test for all control modes are shown in Figure 3.2, 3.3 and 3.4 respectively. The position results match with the visual inspection performed during the experiments. The joints reached the set-point within

5 seconds using the PWM, position and velocity control modes. However using the current control mode the set-points were not reached for any joint. Using the PWM control mode the joints reached its set-point atleast a second faster than every other control type.

Discussion

There are a few reasons that using current control the joints did not reach their set-point:

- *Non-optimal controller*, a simple P controller is not sufficient to counteract the inertia once the joint is moving. Design of a better controller for the current control mode was outside the scope of this test.
- *Low gains*, the proportional gains in the controller for joint 6-8 are not high enough to overcome gravity. The gains were not increased due to the chance of overshoot and the youBot harming itself.

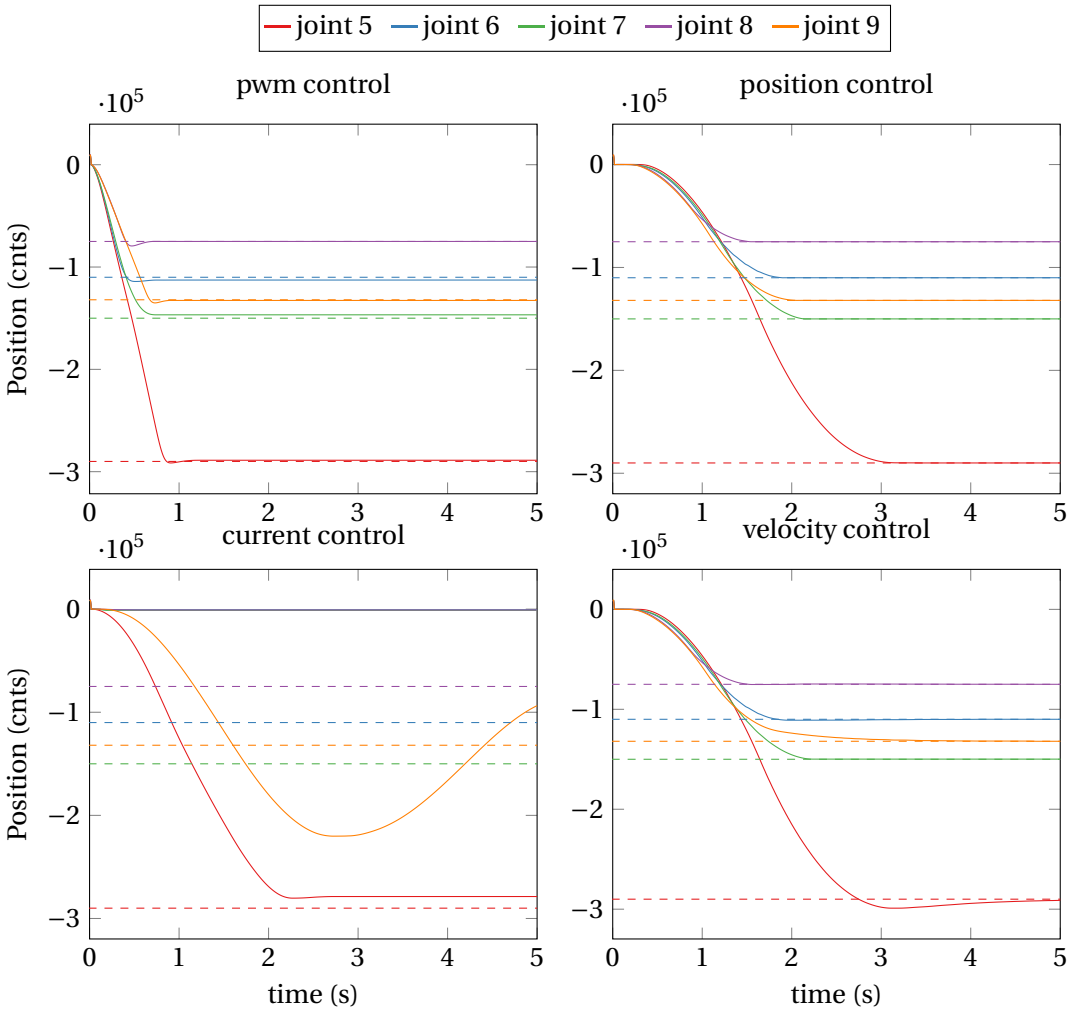

Figure 3.2: Position output of the youBot joints during arm extension for several control types

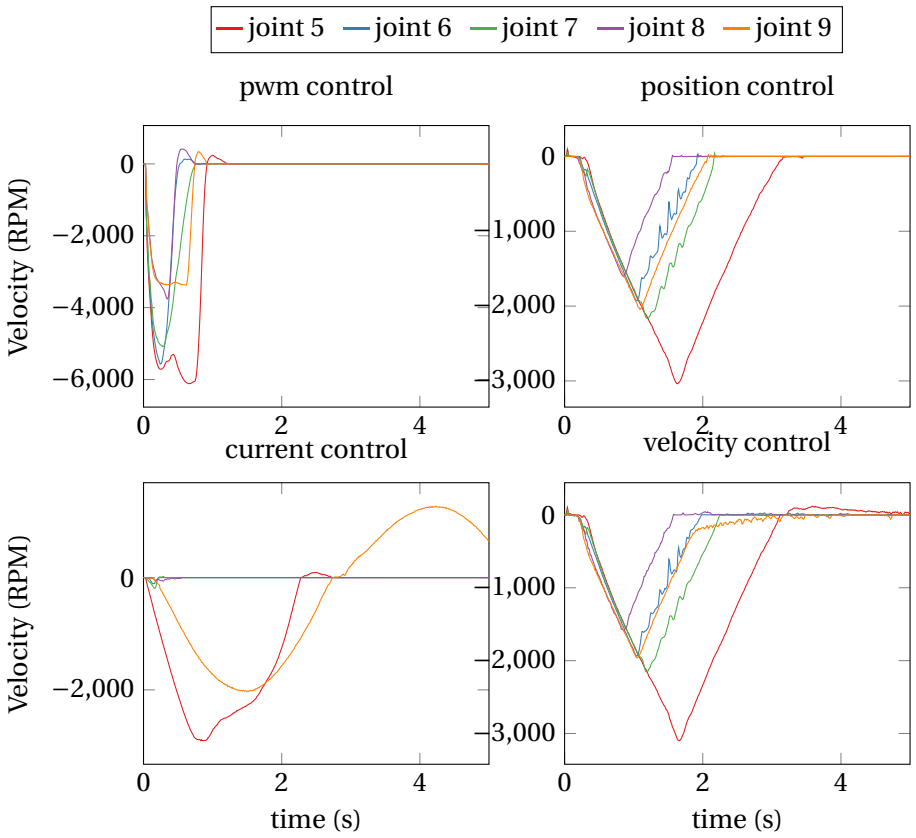

Figure 3.3: Velocity output of the youBot joints during arm extension for several control types

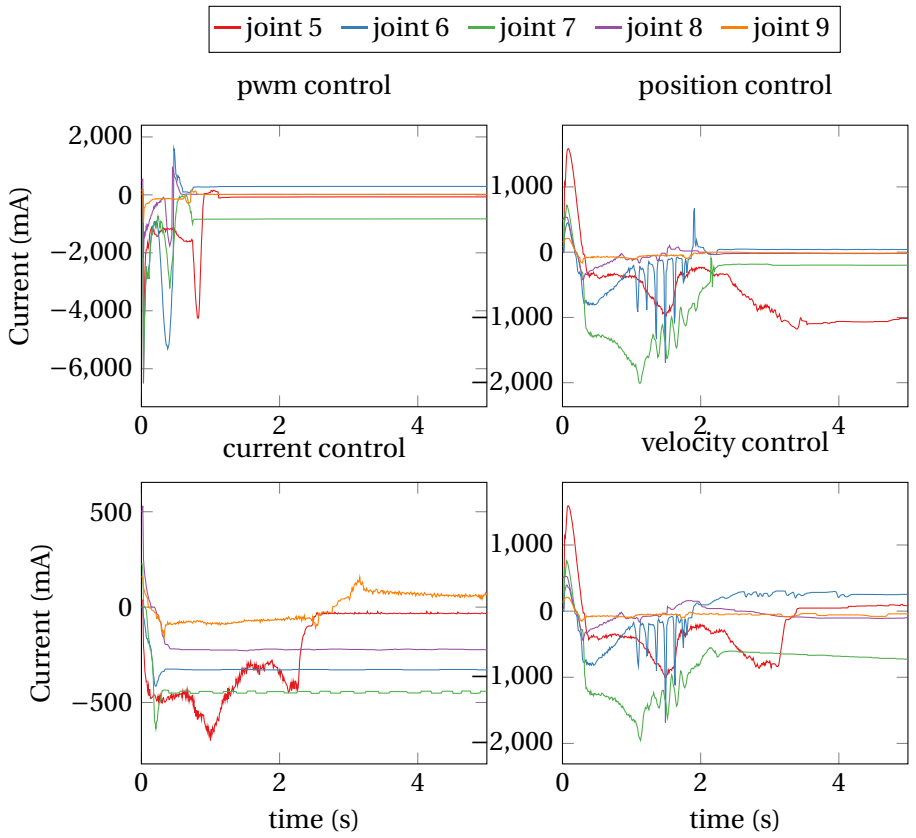

Figure 3.4: Current output of the youBot joints during arm extension for several control types

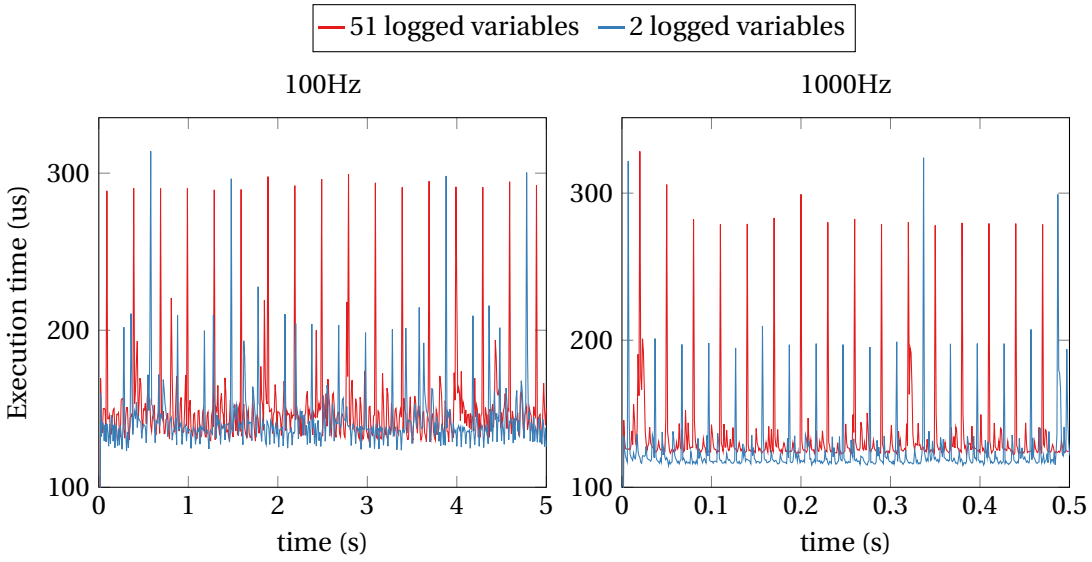

Figure 3.5: Execution time of every 20-sim 4C control loop cycle, for two different frequencies and different amount of logged variables

3.3.2 Performance test

The performance test records the amount of time it takes for the 20-sim 4C control loop to finish one cycle, called the execution time. The current time is recorded at line 10 in Algorithm 3.5, before and after the wait operation, giving the time at the end and start of a control loop respectively. The execution time is calculated by subtracting the start time from the end time. The execution time is measured for a simple model that sets the position using the position control mode such that no model calculation are required, but still has changing outputs such that the validity of the loop can be checked.

Results

The results of the performance test are shown in Figure 3.5. Every 30 cycles the execution time has an increase of $50-100\mu$s. The amount of logged variables by 20-sim 4C seems to have an influence on the height of the peaks. There is almost no difference in execution time between running the loop at 100Hz or 1000Hz, also the peaks are separated by 30 cycles on both frequencies.

Discussion

Further testing proved that the peaks are not directly caused by 20-sim 4C. By deactivating parts of the code the function that caused the peaks was found. The function *ec_send_processdata*() from SOEM causes the peaks and overhead as can be seen in Figure 3.6. But that still does not explain why the peaks vary with the amount of logged variables. The most logical explanation of the reason the peaks vary with the amount of logged variables is that SOEM has to communicate with the Linux network stack. The Linux network stack is not managed by Xenomai and therefore not real-time. Other programs can occupy the network stack, like the XMLRPC daemon that has a longer execution time when more variables need to be logged.

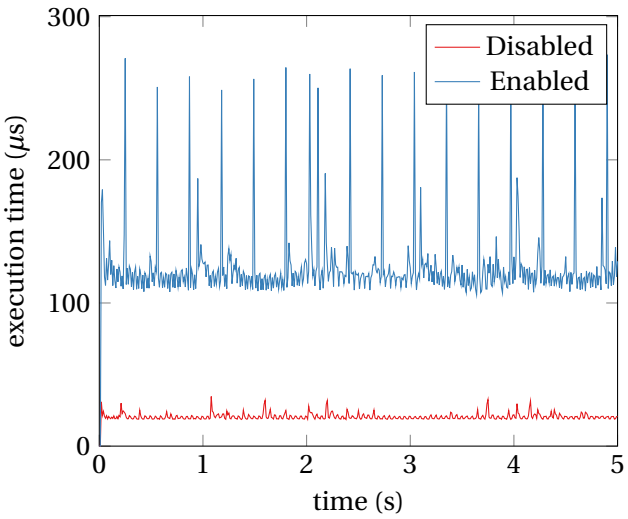

Figure 3.6: Execution time when the function *ec_send_processdata*() is enabled and disabled

3.3.3 End stop protection test

The end stop protection is tested by putting maximum PWM in both directions on the joint and see if the protection takes over close to the end stop. This is done by first controlling the joint to its extended arm position and then let one of the joint go to both end stop positions at max PWM.

Results

In Figure 3.7 the plots for all five manipulator joints is shown. The dotted line is plotted to indicate the position of the end-stops.

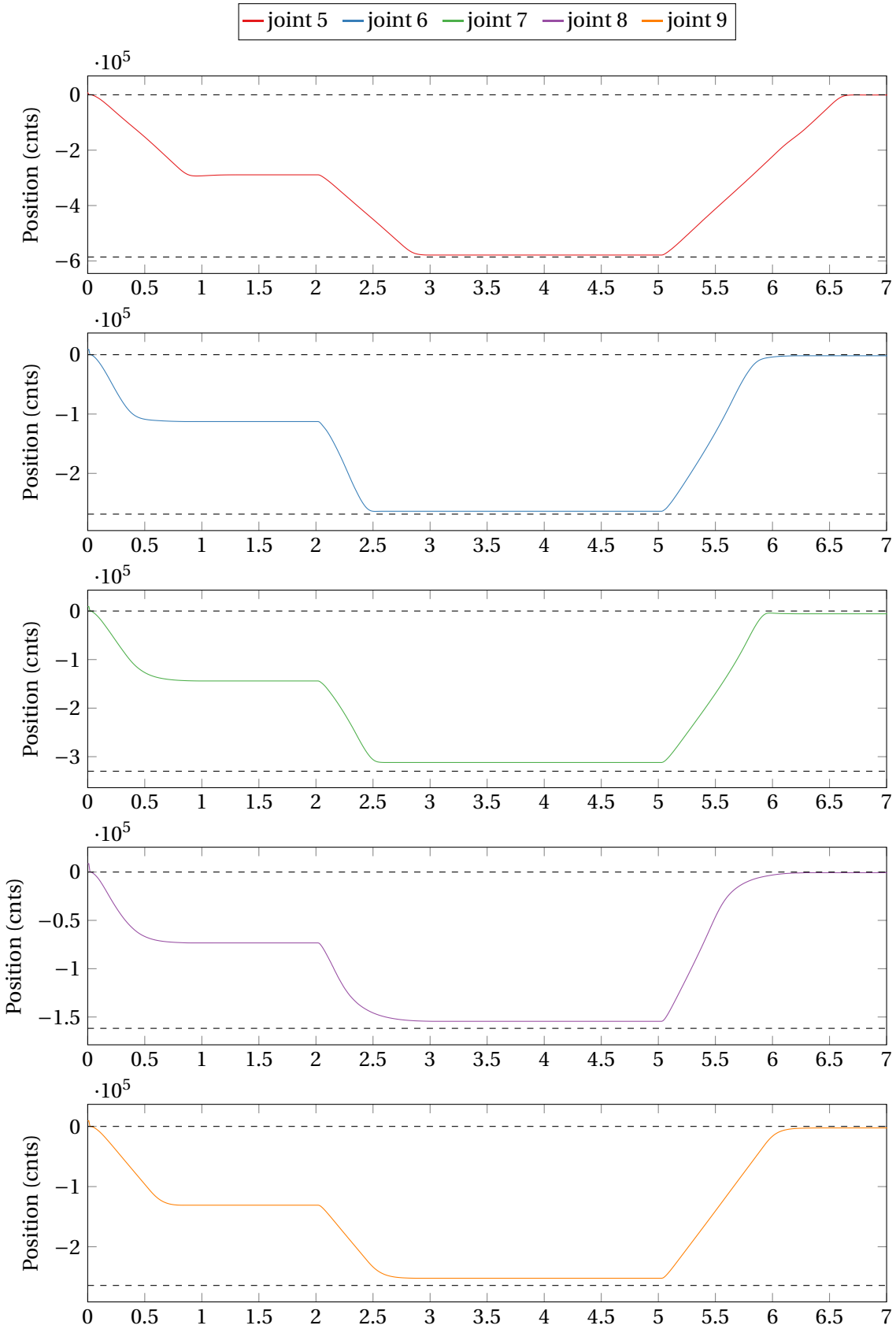

Figure 3.7: Safety layer in action for every joint

Discussion

With the end stop protection it is not possible to hit the end-stops of the joint, provided that the loop frequency is sufficient > 25Hz.

3.3.4 Conclusion

The youBot can be controlled with the RaMstix using the tool 20-sim 4C. All inputs and outputs of the youBot can be used. The youBot can be controlled with a frequency up to 2 kHz, on a higher frequency periods can be missed due to peaks in sending the process data. The end-stops can not be hit when operating the youBot with PWM.

4 Inverse Kinematics with Kinematic Constraints solver

In this chapter the design and implementation of the new control method called the Inverse Kinematic with Kinematic Constraints (IKwKC) solver is detailed.

4.1 Design

Two options for the control method have been investigated. Both are operational space controllers, the impedance controller has been used in the *motion stack* (Brodskiy, 2014).

- *Impedance control*, implemented for the youBot by Brodskiy (2014). It implements a virtual spring between the current end-effector position and the desired end-effector position resulting in a wrench. This wrench on the end-effector is transformed to joint torques with the Jacobian.

  Advantages of impedance control are that it can achieve compliant behaviour and that it has a physical interpretation. A disadvantage is that it is limited to torque (current) control.

- *Inverse Kinematics (IK) solver*, in Sharma et al. (2012) an analytic closed form inverse kinematic solver for the youBot is detailed. It calculates the joint positions from a desired position of the end-effector– 6D Cartesian coordinates of the end effector and 3 redundancy parameters– to the position and orientation of the base and arm joint angles θ_5 to θ_9.

  Because the IK solver is analytic, the joint positions can be calculated every control cycle without adding much processing time, this makes it usable as a hybrid between conventional IK solvers– which operate with some intermediate task points– and operational space control (Siciliano and Khatib, 2008, Chapter 6).

With an IK solver, the joints can still be controlled independently with any kind of joint space controllers, therefore it should still be possible to implement impedance control on joint level. Since one of the goals is to control the motors of the youBot as directly as possible (in this case using PWM), the IK solver has been chosen for end-effector control.

Sometimes the IK solver calculates joint positions that are unreachable even though a reachable end-effector position is commanded. By varying the redundancy parameters reachable joint positions can be calculated. This is termed the *Kinematic Constraint* solver.

The reachable space of z is limited and depends on the rotation of the end-effector, therefore if the youBot arm is in a position where z is minimum or maximum for that orientation– called a *corner case*– and a change of orientation is desired, then z would also need to be adjusted. Otherwise the desired end-effector position would not be reached. Since end-effector control is designed for human interaction it is chosen that when the youBot arm is in a corner case and an adjustment to the rotation is made that would lead to an unreachable end-effector position, z will be adjusted such that the desired rotation is achieved.

4.2 Realization

In Algorithm 4.1 the process of the IKwKC solver is detailed, this process is implemented in 20-sim. The closed form analytic inverse kinematic solver detailed in Sharma et al. (2012) is used with some modifications. The zero positions of the angles are altered to the extended arm position. Essential and relevant steps in the Algorithm are detailed in the following sections. In Figure 4.1 an illustration of the youBot arm and base is shown detailing joint angles, θ and

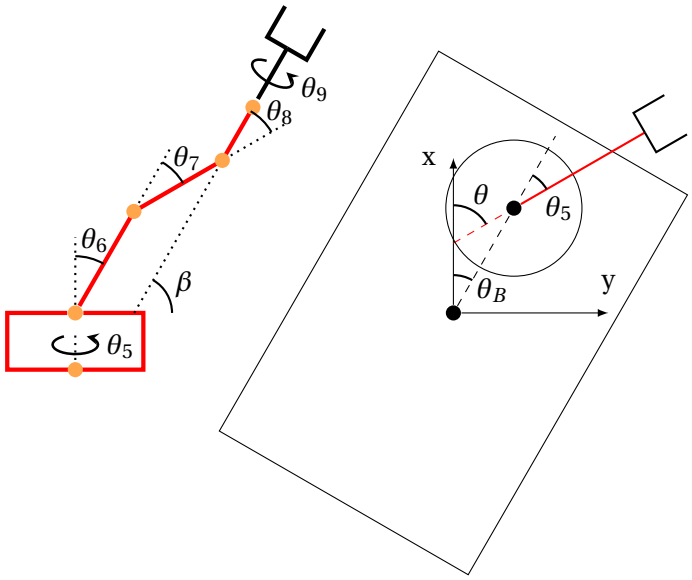

Figure 4.1: youBot arm and base representation showing all joint angles and extra variables θ and β

Figure 4.2: Rendition of the three redundancy parameters: (a) is ρ_1, (b) is ρ_2,(c) is ρ_3 (Sharma et al., 2012)

β. In Figure 4.2 renditions of several poses of the youBot are shown clarifying the redundancy parameters.

Algorithm 4.1 Inverse Kinematic with Kinematic Constraints solver

```
1  Calculate θ and β
2  Calculate base angle and joint 5 angle
3  Check goal position for a corner case
4  if Not a corner case then
5     repeat
6        Calculate angles of joint 6, 7 and 8
7        if joint angle 6, 7 or 8 is not reachable then
8           Use different value ρ2
9           if tried whole range of ρ2 values then
10             Use joint angles from previous loop
11          end if
12       end if
13    until angles are possible
14 end if
15 Calculate base position and angle joint 9
```

4.2.1 Calculation of θ and β

Values for θ and β can be extracted from the goal orientation. β is the angle the end effector makes with the X-Y plane of the base and θ is the angle the youBot turns about its center. A

problem with the Cartesian calculation of θ and β where the set-point is generated with a controller is that it is not clear at all times why it is moving the way it is. Also the range is limited since the range of β is $-\pi/2, \pi/2$. Therefore another way has also been implemented, termed *direct control*, where the buttons of the controller are directly mapped to θ and β. With *direct control* it is always clear how the youBot arm is going to move, but the Cartesian control is lost. The two possibilities for calculation of θ and β are listed in Table 4.1, where r are the indices of the goal rotation matrix calculated by the Tait-Bryon angles (θ_x, θ_y and θ_z).

	Cartesian control	*Direct control*
θ	$arctan2(r_{32}, r_{31})$	θ_y
β	$arctan2(r_{33}, \sqrt{(r_{31})^2 + (r_{32})^2})$	$-\theta_x + \pi/2$

Table 4.1: Two possibilities for calculating β and θ

4.2.2 Calculation of base angle and joint 5 angle

In Sharma et al. (2012), the base angle (θ_b) is calculated with θ and the angle of joint 5 (θ_5) with the first redundancy parameter (ρ_1), but θ_5 can be controlled much more accurately than θ_b, because the wheels of the youBot experience slip. Therefore

$$\theta_5 = \theta - \rho_1$$

and

$$\theta_b = \rho_1$$

this ensures that changing the redundancy parameter only changes the angle between the base and the arm and the end-effector remains in the same place.

4.2.3 Check corner cases

By extrapolating the physical configuration to the desired position it can be checked whether the desired position leads to a corner case. There are two distinct sets of corner cases: $z > z_{\max}$ and $z < z_{\min}$ an illustration is shown in Figure 4.3. When the youBot arm is in a corner case the joint angles for joint 6, 7 and 8 are easily calculated and only depend on β. In Table 4.2 the equations for the calculation of joint angles 6, 7 and 8 are shown for all possible corner cases.

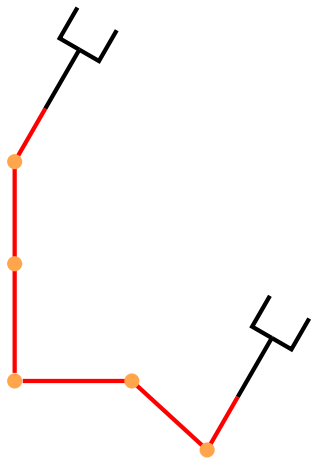

Figure 4.3: Corner cases with maximum and minimum height for a value of β

Conditions	θ_6	θ_7	θ_8
$z > z_{max}$ and $\beta < \beta_{min}$	0	$\pi/2 - \beta - \theta_{8_{max}}$	$\theta_{8_{max}}$
$z > z_{max}$ and $\beta_{min} < \beta < \beta_{max}$	0	0	$\pi/2 - \beta$
$z > z_{max}$ and $\beta > \beta_{max}$	0	$\pi/2 - \beta - \theta_{8_{min}}$	$\theta_{8_{min}}$
$z < z_{min}$ and $\beta < \beta_{min}$	$\theta_{6_{max}}$	$\pi - \theta_{6_{max}}$	$-\pi/2 - \beta$
$z < z_{min}$ and $\beta_{min} < \beta < \beta_{mid}$	$\theta_{6_{max}}$	$\pi/2 - \beta - \theta_{6_{max}} - \theta_{8_{min}}$	$\theta_{8_{min}}$
$z < z_{min}$ and $\beta_{mid} < \beta < \beta_{max}$	$\theta_{6_{min}}$	$\pi/2 - \beta - \theta_{6_{min}} - \theta_{8_{min}}$	$\theta_{8_{max}}$
$z < z_{min}$ and $\beta > \beta_{max}$	$\theta_{6_{min}}$	$-\pi - \theta_{6_{min}}$	$3\pi/2 - \beta$

Table 4.2: Equations for calculation of θ_6, θ_7 and θ_8 when the youBot arm is in a corner case. Where $\beta_{min} = \pi/2 - \theta_{8_{max}}$, $\beta_{mid} = \pi/2$ and $\beta_{max} = \pi_2 + \theta_{8_{min}}$

4.2.4 Calculate joint angles 6, 7 and 8

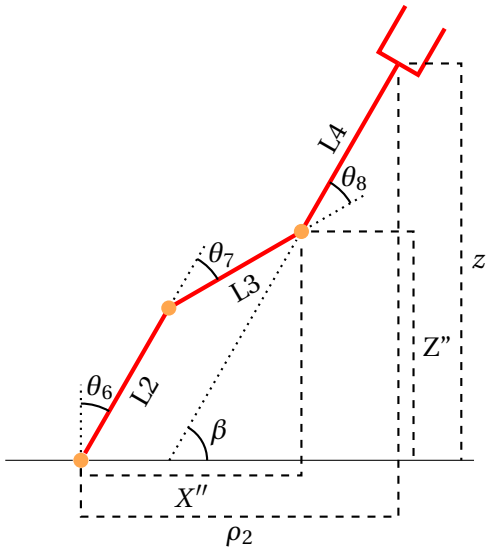

Figure 4.4: youBot arm representation for calculation of θ_6, θ_7 and θ_8

The formulas from Sharma et al. (2012) have been modified such that an extended arm is the zero position instead of a position that is not reachable by the youBot arm. Figure 4.4 gives an overview of the youBot arm and the angles that are to be calculated. Z'' and X'' can be expressed in known quantities:

$$Z'' = z - L_4 \cos(\beta) \qquad X'' = \rho_2 - L_4 \sin(\beta)$$

θ_7 can then be calculated as follows

$$\theta_7 = atan2(\sin(\theta_7), \cos(\theta_7))$$

$$\cos(\theta_7) = \frac{Z''^2 + X''^2 - L_2^2 - L_3^2}{2L_2L_3} \qquad \sin(\theta_7) = \rho 3\sqrt{1 - \cos(\theta_7)}$$

θ_6 can be calculated as follows

$$\theta_6 = atan2(X'', Z'') - atan2(L_3 \sin(\theta_7), L_2 + L_3 \cos(\theta_7))$$

and finally θ_8 can be calculated

$$\theta_8 = \pi/2 - \beta - \theta_6 - \theta_7$$

4.2.5 Kinematic constraint solver

The kinematic constraint solver is detailed in Algorithm 4.1 line 5-13. When joint angle 6, 7 or 8 is unreachable a set of possible ρ_2 values is generated. The size of this set can be chosen and the set is sorted, from smallest deviation from desired ρ_2 to biggest deviation. The first value in the set is used to recalculate θ_6, θ_7 and θ_8. If one of those angles is still unreachable the next value in the set is used to recalculate the joint angles and so on, until a value of ρ_2 is found that has reachable angles or when the set is depleted. When this is the case previous angles that are reachable are used.

4.2.6 Calculation of base position and joint 9 angle

The base position is calculated using forward kinematics which calculates the end effector position using the already calculated joint angles except for the angle of joint 9 which does not affect the distance to the end-effector. By comparing the end effector position in x and y the desired position in x an y the base position can be calculated.

Joint 9, just as θ and β, can be calculated in 2 ways. With Cartesian space control and *direct control*. In Table 4.3 the two ways are shown.

	Cartesian control	*Direct control*
θ_9	$atan2(\sin(\theta_9), \cos(\theta_9))$	θ_z

Table 4.3: Two ways to calculate angle of joint 9

$$\sin(\theta_9) = e_{11}r_{11} + e_{21}r_{21} + e_{31}r_{31}$$
$$\cos(\theta_9) = e_{12}r_{11} + e_{22}r_{21} + e_{33}r_{31}$$

Where e are the indices of the end effector rotation matrix and r the indices of the goal rotation matrix.

4.2.7 Limit set-point to reachable space

During use of the IKwKC solver, z and ρ_2 sometimes deviate from their set-point. To change the actual position of the youBot arm the set-point position first has to arrive at the actual position, or there will be weird jumps. To solve this a feature termed *dynamic set-point* was introduced an illustration of this feature is shown in Figure 4.5. With dynamic set-point the actual position can be left directly, this comes at the cost that z and ρ_2 are floating, they do not correspond to the set-point but only to the change in the set-point.

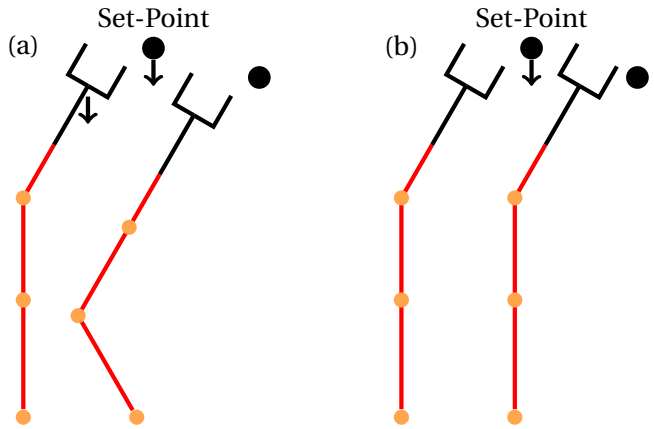

Figure 4.5: Change of youBot configuration to a change in set-point when it is outside of reachable space. (a) with *dynamic set-point*, (b) without *dynamic set-point*

4.2.8 Intrinsic safety

The youBot is able to hit itself and the base, therefore intrinsic safety is necessary. Intrinsic safety should not be included in an inverse kinematic solver, but because the set-point is dynamic the exact position is only known in the IKwKC solver, therefore intrinsic safety is included. The implemented intrinsic safety has two parts. The first part protects the arm from colliding with the base and the second part protects the arm from colliding with itself. Collision with the base is prevented by limiting the height at certain values of θ. Collision of the arm with itself is prevented by limiting ρ_2 when $\beta < 0 \wedge \beta > \pi$.

4.3 Tests

To adequately test the IKwKC during realization a real-time youBot model is necessary, similar to a plant to test controllers. For the tool 20-sim a youBot model is already developed (Dresscher et al., 2010), however this model is too computational heavy to be run in real-time, therefore a new youBot model is developed in 20-sim. This model can be run at real-time and translates the outputs of the IKwKC solver, base position and joint angles, to a stylized animation model of the youBot. This is done with forward kinematics. If the angle commanded for a joint is not reachable it is capped to its maximum value. The model assumes that the control is done perfectly and instantaneous, due to the way it is implemented.

After completion of the IKwKC solver it has been used as part of a youBot demo. The demo has been used numerous hours by numerous users without errors from the IKwKC solver. Therefore it can be concluded that the IKwKC solver works as intended. It is robust and can be used by anyone.

4.3.1 Range of motion test

The range of motion test is designed to test the range of motion the youBot arm has in the z direction depending on β. It is tested by first changing β in small increments from minimum to maximum while z is maximum and then on minimum z changing from maximum β to minimum β in small increments.

Results

In Figure 4.6 the range of motion of the IKwKC solver is shown as well as the range of motion of the IK solver presented by Sharma et al. (2012). The area the IKwKC solver covers is much bigger than the IK solver it is based on and almost fully encloses it. The IK solver only passes through the line denoted by A.

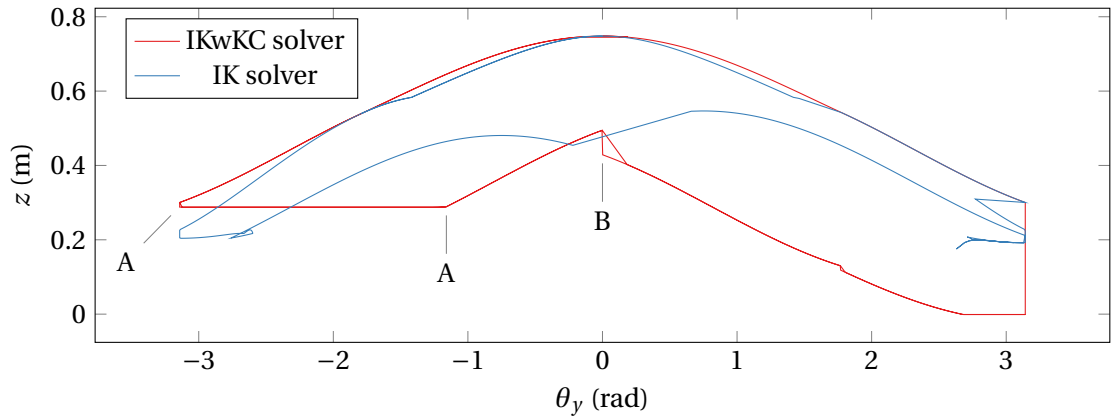

Figure 4.6: Range of motion for the inverse kinematic solver by Sharma et al. (2012) and the inverse kinematic with kinematic constraints solver

Discussion

The IKwKC solvers range of motion is greater than that of the IK solver also it almost entirely encompasses the IK solvers range of motion except for the line at A. At A the height is limited by the self protection such that the youBot arm does not hit the base. The height difference at B is caused by joint 6, which has no symmetric joint limit. The two lines at B are caused by the corner case checker. When going from right to left, the corner case checking increases the height, going from left to right z has a rate of change and therefore is a sloping line.

4.3.2 Conclusion

The IKwKC solver outputs valid joint angles for every set-point and the entire space reachable by the arm can be reached with the IKwKC solver.

5 Set-point generation and communication

Set-point generation and communication is used for user interaction in the youBot demo. In this chapter the development of a platform to interact with the youBot and integration of the platform in the toolchain is detailed.

5.1 Design

5.1.1 Set-point generation

Set-point generation will be done on the development pc, the main reason being that a lot more user input devices work on windows than linux, especially the stripped down version of the linux kernel that the RaMstix works on. 20-sim is used as a platform to generate and send set-points, since 20-sim can call functions using dynamic link library's (dll's). A few reasons for the use of 20-sim over an executable for generating and sending set-points:

- *model testing with device,* an input device can be used during real-time simulation to test a model.
- *easy switch out,* another input device can easily be used by developing a dll for it.
- *real-time comparison,* the youBot and simulated youBot can be compared during run-time with the same set-points.

5.1.2 Set-point communication

Since the RaMstix communicates via wifi with the development pc for 20-sim 4C, it is chosen to communicate the set-point unilaterally over the network as well. Information over the network can be send with sockets, there are three kinds of sockets:

- *A datagram socket,* uses the User Datagram Protocol (UDP), is also called connectionless socket and uses a minimum amount of protocol mechanism. A datagram socket is used for time sensitive applications because dropping packets is preferable over waiting.
- *A stream socket,* uses the Transmission Control Protocol (TCP) and is also called connection-oriented socket. TCP provides ordered error-checked packets, but this comes with additional overhead.
- *A raw socket,* does not use any protocol. To utilize the raw socket a protocol has to be designed.

For the communication with the RaMstix the datagram socket is chosen. It is less work to implement than the raw socket, has less overhead than the stream socket and it is commonly used for time sensitive applications.

Since the UDP packets are sent over wifi the chance for data errors is higher than sending them over TCP. The datagram socket only error checks the length of the received packet using a checksum, so if the packet length is incorrect the packet is thrown out. If some of the data in the packet is corrupted it is not detected, therefore error detection should be built in the function that receives the datagram.

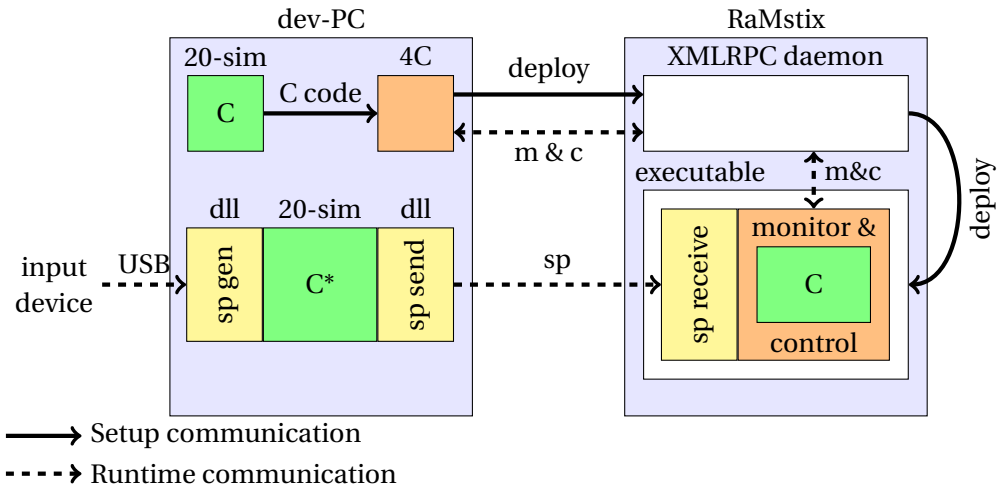

Figure 5.1: Set-point generation on development pc and communication to the RaMstix, sp stands for set-point

5.1.3 Final design

In Figure 5.1 a graphical representation of the final design is shown. The yellow coloured rectangles– set-point generator (sp gen) dll, set-point send (sp send) dll and set-point receive (sp receive) interface– have been realized. The sp receive interface should also be integrated with the RaMstix target in 20-sim 4C.

The devised operation is as follows. In 20-sim a model (C) is created that should run on the RaMstix. The code of this model is transferred to 20-sim 4C. There it is encapsulated by monitor & control software and deployed on the RaMstix. Another model (C*) in 20-sim simulates in real-time and calls a function in the sp gen dll to generate the set-point. This set-point is send to the RaMstix with the sp send dll. The sp receive interface receives the set-point and saves it such that it can be accessed by the monitor & control loop such that it can be used by the model (C).

5.2 Realization

All algorithms described in this section are written in C.

5.2.1 Set-point generation

Set-point generation for two input devices, a keyboard and an Xbox controller, is implemented. The function that generates the set-point for the Xbox controller is described in Algorithm 5.1. Set-point generation using the keyboard is done in a similar fashion, only it is implemented in 20-sim, because a dll for the keyboard is already incorporated in 20-sim. In 20-sim line 2-6 of Algorithm 5.1 are implemented for set-point generation with the keyboard.

In table 5.1 the mapping between the set-point and the two input devices is shown. The youbot has 9 DOF, 6 can be expressed in cartesian coordinates the other 3 are independent and are called ρ_1 , ρ_2 and ρ_3. These 9 dof are generated as set-points and are sent to the RaMstix.

Set-point	Xbox controller		Keyboard	
	+	-	+	-
x	D-pad up	D-pad down	W	S
y	D-pad right	D-pad left	D	A
z	Trigger right	Trigger left	E	Q
θ_x	Left stick up	Left stick down	Numpad 9	Numpad 7
θ_y	Left stick right	Left stick left	Numpad 3	Numpad 1
θ_z	Right stick right	Right stick left	Numpad 6	Numpad 5
ρ_1	Right bumper	Left bumper	Numpad .	Numpad 0
ρ_2	Right stick up	Right stick down	Numpad 8	Numpad 2
ρ_3	Stick button left	Stick button right	Space	

Table 5.1: Mapping between set-point and input devices

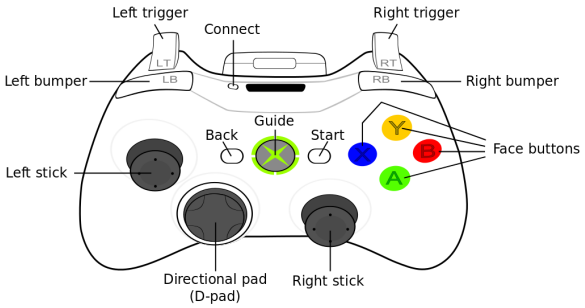

Figure 5.2: Buttons on an Xbox controller (Alphathon, 2010)

Algorithm 5.1 Generate set-point

Require: Xbox controller connected **and** rate of change (roc)

```
Get Xbox controller state
for Each button coupled to set-point do
    if Button pressed then
        Increase coupled set-point by roc
    end if
end for
for Each stick direction do
    Calculate normalized axis
    Increase coupled set-point by normalized axis times roc
end for
```

5.2.2 Send set-point

Before the set-point can be sent, the socket needs to be initialized, that process is described in Algorithm 5.3. The function that sends the set-point is described in Algorithm 5.2. The set-point is saved as an array of floats, to send these over the network via socket they have to be transformed, since the data type float is saved differently on different devices. Therefore the set-points are packed using the IEEE Standard for Floating-Point Arithmetic (Zuras et al., 2008) using functions detailed in Hall (2001).

Algorithm 5.2 Send set-point

Require: Set-point **and** Socket send initialized
1 Pack set-points values according to IEEE 754
2 Send set-points over socket

Algorithm 5.3 Initialize socket send

Require: ip-address
```
1 Read ip-address from configuration file
2 Initialize winsock dll with WSAStartup()
3 Set up structures with getaddrinfo(&servinfo)
4 for All servinfo entries do
5     Set up socket
6     if successful then
7         Break for loop
8     end if
9 end for
```

5.2.3 Receive set-point

The function that receives datagrams is a blocking function, even on Xenomai with high real-time priority the function will block until it receives a datagram on the indicated port. This is resolved by operating the receive function in a separate thread, as shown in Figure 5.3 The variable array containing the set-points is shared between the two threads, the thread with the receive function writes to it and the original thread reads from it. Therefore there is no need for mutual exclusion (mutex). The initialization of the socket and thread is described in Algorithm 5.5 and the function that receives the set-point is and error checks it is described in Algorithm 5.4.

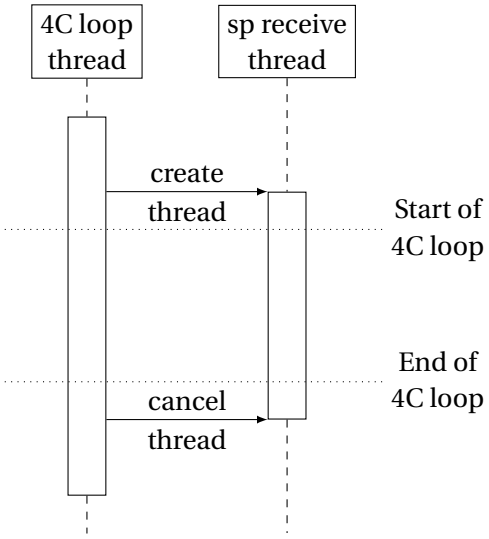

Figure 5.3: Sequence diagram to show creation of the set-point receive thread

Algorithm 5.4 Receive set-point

```
1  while thread is not cancelled do
2    Listen on socket for datagram packets containing the set-point
3    if Set-points received then
4      for every set-point do
5        Unpack set-point value according to IEEE 754
6        Calculate difference between received set-point and previous set-point
7        if difference > 1 then
8          Use previous set-point
9        else
10         Use received set-point
11       end if
12     end for
13   end if
14 end while
```

Algorithm 5.5 Initialize socket receive

```
1  Set up structures with getaddrinfo(&servinfo)
2  for All servinfo entries do
3    Set up socket
4    if successful then
5      Bind socket
6      if successful then
7        Break for loop
8      end if
9    end if
10 end for
11 Create set-point receive thread
```

5.3 Tests

Two tests are performed to test set-point generation on the development pc and set-point communication to the RaMstix.

5.3.1 Generation test

Set-point Generation is done on the development pc using 20-sim, all variables are generated with the Xbox controller and keyboard, as listed in table 5.1. The set-point values are generated by repeatedly pressing randomly on one of the two buttons or holding the thumb stick in a direction.

Results

In Figure 5.4 the results for the generation test are shown. Three variables generated with the Xbox controller are plotted: x, z and θ_x to show variables generated with the buttons, triggers and thumb stick. One variable generated by the keyboard x is plotted, to show a variable generated by the keyboard. All the other variables were also successfully generated but are not plotted.

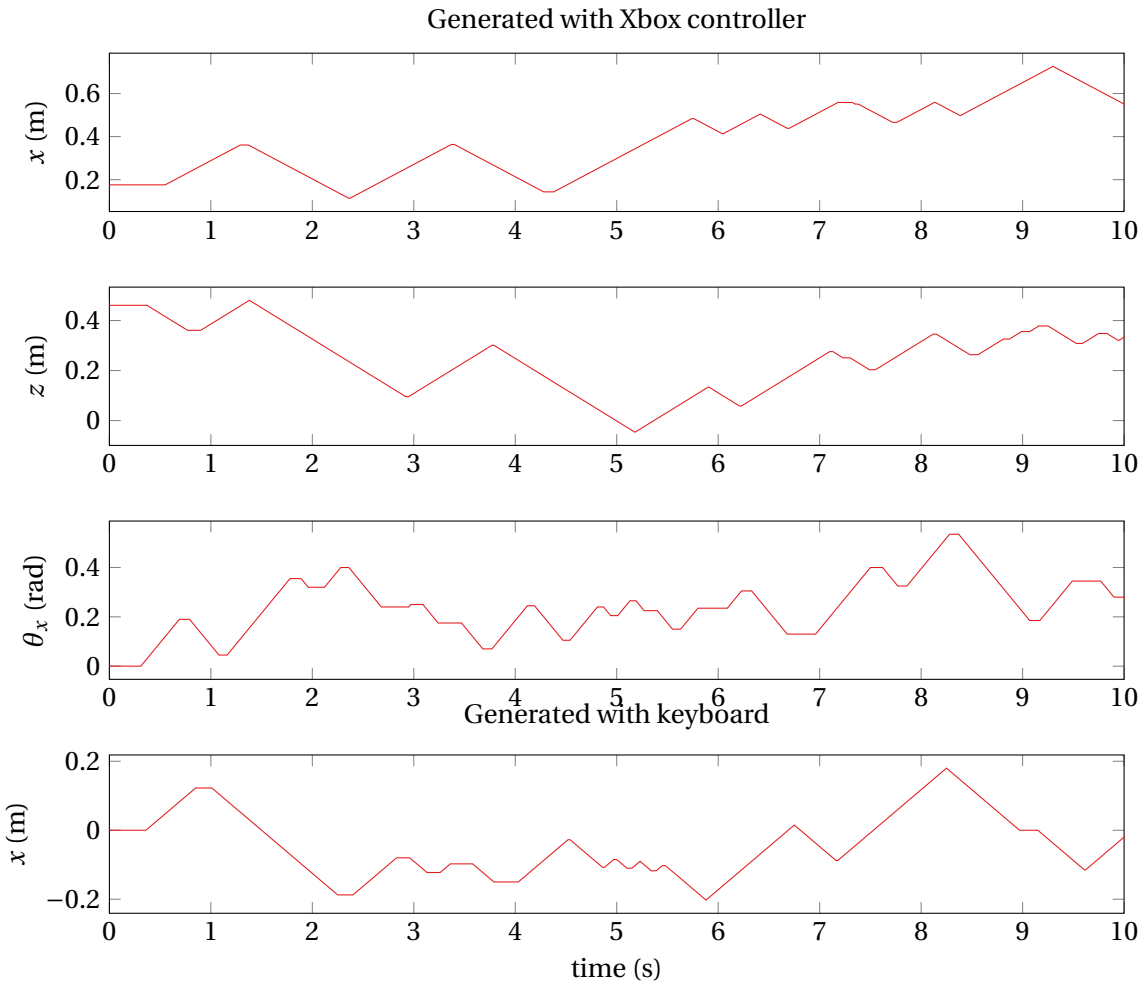

Figure 5.4: Generated set-points x, z and θ_x with the Xbox controller and x with the keyboard

Discussion

There is almost no difference in generation between the button, trigger and thumb stick. However with the thumb stick inversion of direction takes longer, because it takes time to physically move the stick.

5.3.2 Set-point communication test

In the set-point communication test three features are tested: set-point sending, set-point receiving and set-point error checking. Set-point sending is done on the development pc in 20-sim using the function described in Algorithm 5.2. With 20-sim 4C the set-points are received and error checked using the interface detailed in section 5.2.3.

A repeating signal is sent with some injected errors to test the error checking. There is a 1 in 200 chance of an error occurring every simulation step of 0.01 s. When an error occurs, the send set-point is adjusted to a unified random value between ±64. A network time out is simulated between 12 and 15 seconds, based on a real-time out. The effect of a network time-out is shown in Figure5.5. At the 12 second mark an error value is send and afterwards no values are sent until the 15 second mark.

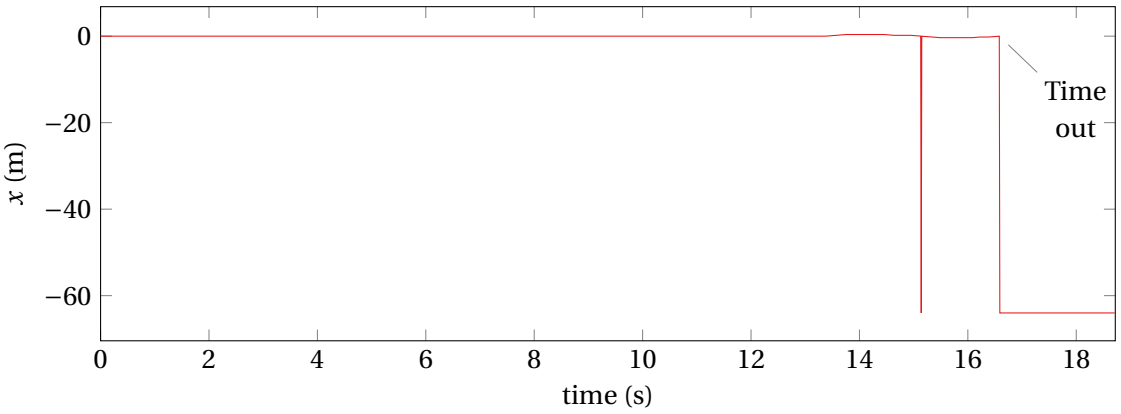

Figure 5.5: Effect of a network time out on the received set-point

Results

In Figure 5.6 the sent set-point compared with the received and error checked set-point on the RaMstix. The graph does not say anything about the delay, the received data is shifted to overlap with the send. This is done because the difference is made up of the time between the start of the 20-sim 4C control loop and the start of the 20-sim model that generates and sends the set-points.

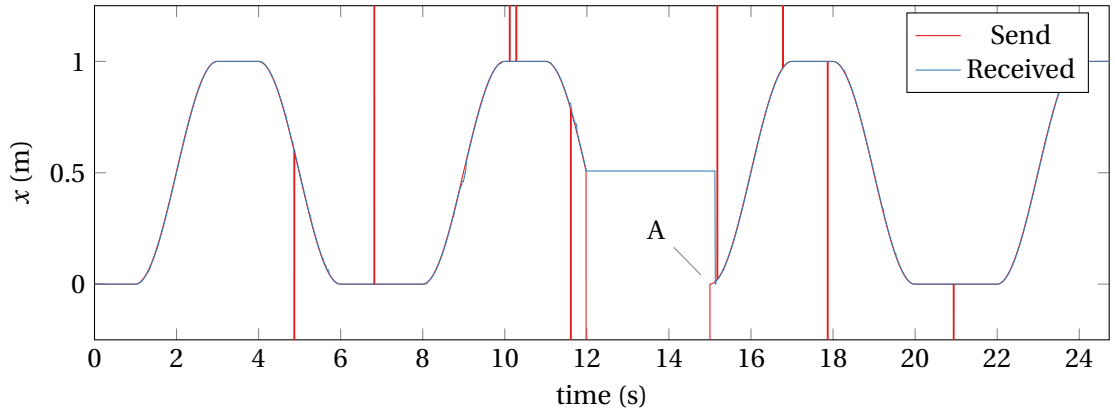

Figure 5.6: Send set-point vs. received set-point

Discussion

The communication between the development pc and the RaMstix is successful, the RaMstix received most of the sent packets. All errors are also filtered out effectively. After the simulated time out at A in Figure 5.6 it takes a moment for the RaMstix to receive the packets again.

5.3.3 Conclusion

The set-points can be generated on the development pc by both the Xbox controller and keyboard. The generated set-points can be sent from the development pc to the RaMstix. On the RaMstix the received set-points are error checked, which filters out corruption of data effectively.

6 Conclusion and Recommendations

6.1 Conclusion

The youBot can be controlled via the RaMstix using the toolchain 20-sim 4C. A new control method, the IKwKC solver, is designed and realized. With the built-in safety features in the toolchain and the inherent safer control method the youBot should be safe to be used without the youBot breaking down. A demo in which a user can control the youBot via wifi with an Xbox controller or keyboard has been realized.

6.2 Recommendations

Several recommendations can be made

- *Real-time network communication*, as noted in Chapter 3 the linux network stack is not real-time and this leads to peaks in the execution time. By implementing a real-time networking framework (like RTNet Kiszka et al. (2005)) these performance dips could be prevented.
- *Better joint controllers*, currently P-controllers are implemented at joint level. Better joint controllers can be implemented to improve steady state error and overshoot.
- *YouBot interface in LUNA*, the youBot interface that has been developed in Chapter 3 can also be used in LUNA.
- *Update Dresscher et al. (2010) youBot model*, the youBot model is developed with force inputs. But now a lower level control of the motors is possible (PWM) and the youBot model could be updated with motors to accommodate the new input.

A SOEM

SOEM is a simple EtherCAT master, an EtherCAT master is a software library that handles the communication with EtherCAT slaves, it provides the means to send and receive EtherCAT frames as well as mailbox messages. Mailbox messages can be sent with the following profiles: CAN application protocol over EtherCAT (CoE), Service drive profile according to SERCOS (SoE) and File access over EtherCAT (FoE). SOEM sends EtherCAT messages over the NIC through the use of the RAW socket. During configuration SOEM automatically maps the IO between the EtherCAT master and slave in an IOmap keeping the logical process image as small as possible. This is done by fitting bits from different slaves in a single byte. The mapped outputs are put first in the IOmap and thereafter the mapped inputs. Figure A.1 gives an example of the mapping, the colors between the physical and logical address represents the mapping of the bits, the color represents the slave.

Physical adress		Logical adress in process image				
					Inputs	Outputs
0x18cfc		0x8	Slave1	EK1100	0	0
0x18cfb		0x7	Slave2	EL4001		1x16bit
0x18cfa		0x6	Slave3	EL3061	1x32bit	
0x18cf9		0x5	Slave4	EL1008	8x1bit	
0x18cf8		0x4	Slave5	EL1008	8x1bit	
0x18cf7		0x3	Slave6	EL2622		2x1bit
0x18cf6		0x2	Slave7	EL2622		2x1bit
0x18cf5		0x1	Slave8	EL2622		2x1bit
0x18cf4		0x0	Slave9	EL2622		2x1bit
	7 6 5 4 3 2 1 0					

Figure A.1: Memory layout, mapping between the physical and logical addresses by the IOmap (Karlsson, 2014)

.

Some important functions of SOEM are

- *ec_init*(IFName), sets up the NIC to be used as EtherCAT Ethernet interface. The input argument is the name of the Ethernet device. The function returns one if it succeeds else it returns zero.
- *ec_config*(UseTable,pIOmap), is a combination of two functions. The first function initializes all the slaves to the PREOP state and sets up mailboxes for slaves that support it. The second function maps the IO in the IOmap and requests the SAFEOP state. The input argument UseTable is a boolean that is true when the slaves have a config table that can be read out. The input argument pIOmap is a pointer to the IOmap that needs to be set-up. The function returns one if it succeeds else it returns zero.
- *ec_send_processdata*(), sends the EtherCAT frame on the NIC.
- *ec_receive_processdata*(TimeOut), tries to fetch the EtherCAT frame for the duration indicated by the argument TimeOut. It returns the working counter.
- *ec_mbxsend*(SlaveNumber,MailboxMessageSend,TimeOut), sends a mailbox message to the slave indicated by the SlaveNumber. The mailbox message to be sent is an array of 8 bit unsigned integers. Due to the 8 bit restriction an integer bigger than 8 bit needs to

be split up into 8 bit portions when sending a message, starting with the most significant bit.

- *ec_mbxreceive(SlaveNumber,MailboxMessageReceive,TimeOut)*, receives a mailbox message. The received mailbox message is also an array of 8 bit unsigned integers, so due to this restriction bigger integers need to be pieced back together, starting with the most significant bit.

B Demo

The following software is necessary to run the youBot demo:

- *20-sim*, to open the models
- *20-sim 4C*, to control the youBot via the RaMstix

The following files are necessary:

- *SendIKValuesXbox.emx* or *SendIKValuesKeyboard.emx*, to send the set-point values to the RaMstix
- *XboxDemo.emx*, contains the controller for the youBot
- *xboxdll.dll*, *sendSP.dll* and *youbot.ini*, these files are necessary for the SendIKValues models to send the set-point to the RaMstix.
- *RaMstix_youBot_target.rar*, contains the 4C target of the RaMstix coupled with the files to receive the set-point and communicate with the youBot over EtherCAT.

The following steps should be taken to start the youBot demo for the first time:

- Copy the xboxdll.dll, sendSP.dll and youbot.ini files to the bin directory in the 20-sim installation directory
- Install the RaMstix 4C target following the direction on the RaM wiki
- Extract the RaMstix_youBot_target.rar file in your 4C target directory
- Generate c-code for the YouBotController submodel in the XboxDemo.emx. This is done in the simulator, selecting tools->real-time toolbox->c-code generation. Then in the target list select 20-sim 4C and generate the c-code. This should open 4C.
- in 20-sim 4C:
  - Connect to the RaMstix, the currently used RaMstix ip is: 10.0.5.114. If a different RaMstix is used alter the ip-adress in the youbot.ini file.
  - Connect the inputs and ouputs. At minimal the encoder inputs, set-point inputs and pwm outputs should be connected.
  - Compile and Run
- Begin simulating in the SendIKValuesXbox.emx or SendIKValuesKeyboard.emx model. Make sure that in the run properties the simulator attempts real-time simulation and is endless.

Now the demo should be up and running, and depending on which model you are using SendIKValuesXbox.emx or SendIKValuesKeyboard.emx you can control the youBot with an Xbox controller or with the keyboard.

Bibliography

Alphathon (2010), Wikipedia Xbox controller, accessed: 03-10-2016.
https://en.wikipedia.org/wiki/Xbox_360_controller#/media/File:360_controller.svg

Beckhoff (2014), Ethercat Slave Controller Hardware Data Sheet Section 1.

Bischoff, R., U. Huggenberger and E. Prassler (2011), Kuka youbot-a mobile manipulator for research and education, in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, IEEE, pp. 1–4.

Brodskiy, Y. (2014), *Robust autonomy for interactive robots*, Ph.D. thesis, University of Twente, Enschede.

Broenink, J. F., M. A. Groothuis, P. M. Visser and M. M. Bezemer (2010), Model-Driven Robot-Software Design Using Template-Based Target Descriptions, in *ICRA 2010 workshop on Innovative Robot Control Architectures for Demanding (Research) Applications*, IEEE, IEEE, pp. 73 – 77.
http://www.rob.cs.tu-bs.de/en/news/icra2010

Dresscher, D., Y. Brodskiy, P. C. Breedveld, J. F. Broenink and S. Stramigioli (2010), Modeling of the youBot in a serial link structure using twists and wrenches in a bond graph, in *Proceedings of SIMPAR 2010 Workshops, Darmstadt, Germany*, SIMPAR, Germany, pp. 385–400, ISBN-13 978-3-00-032863-3.

ETG (2014), Ethercat Slave Implementation Guide.

Franken, M., S. Stramigioli, R. Reilink, C. Secchi and A. Macchelli (2009), Bridging the gap between passivity and transparency, Robotics Science and Systems.

Frijnts, S. D. (2014), Upgrading the Safety Layer and Demo of the youBot Robot, Pre-msc report 007ram2014, University of Twente.

Hall, B. B. (2001), Beej's guide to network programming: using Internet Sockets.

Hogan, N. (1984), Impedance control: An approach to manipulation, in *American Control Conference, 1984*, IEEE, pp. 304–313.

Jansen, D. and H. Buttner (2004), Real-time Ethernet: the EtherCAT solution, IET, pp. 16–21.

Karlsson, A. (2014), SOEM tutorial, accessed: 16-8-2016.
http://openethercatsociety.github.io/doc/soem/tutorial_8txt.html

Kiszka, J., B. Wagner, Y. Zhang and J. Broenink (2005), RTnet-a flexible hard real-time networking framework.

KUKA (2015), youBot wiki, accessed: 22-6-2016.
http://www.youbot-store.com/wiki/index.php/API_architecture

Prytz, G. (2008), A performance analysis of EtherCAT and PROFINET IRT, in *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, IEEE, pp. 408–415.

Sharma, S., G. K. Kraetzschmar, C. Scheurer and R. Bischoff (2012), Unified closed form inverse kinematics for the kuka youbot, in *Robotics; Proceedings of ROBOTIK 2012; 7th German Conference on*, VDE, pp. 1–6.

Siciliano, B. and O. Khatib (2008), *Springer handbook of robotics*, Springer Science & Business Media.

Weijers, F. (2015), *Experimental Evaluation of a Safety Aware Impedance Controller Design*, Msc report 011ram2015, University of Twente.

Zuras, D., M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo et al. (2008), IEEE standard for floating-point arithmetic, *IEEE Std 754-2008*, pp. 1–70.