Project Butler - MDH@Home report
Project course in robotics 2015-2016

ROBIN ANDERSSON
MARCUS LARSSON
JAKOB DANIELSSON
ALBIN BARKLUND
ERIC VIEYRA
ROXANNE ANDERBERG
JONATAN TIDARE
TOBIAS KRISTRÖM
RICKARD HOLM
ANDERS OLSSON
MOBIN HOZHABRI
DENNIS EKLUND
Mälardalens University
January 20, 2016

Contents

1 Background and introduction - Robin Andersson

"Butler - MDH@Home" is a project realized at Mälardalens Högskola in collaboration with Volvo CE in Eskilstuna where the goal is to build an intelligent service robot (ISR). The project is divided into several work packages and will last for five years, where this is the initialization and the first year of the project.

Intelligent service robots is an increasing research topic and a fast growing area where researches strives to develop robots to assist people in their everyday life. Engelberger describes in his paper[1] the concept of an ISR and it's functionality and the possible usage scenarios it could have in domestic environments. Moreover he writes:
"Functionally, the home care robot would:

- *fetch and carry*
- *meal preparation*
- *clean house*
- *monitor vital signs*
- *assist ambulation*
- *manage the environment*
- *communicate by voice*
- *take emergency action (fire, intrusion)*

In domestic environments, the primarily focus of ISR have been the following:

- Assistance 'ordinary' people in their homes
- Assistance to elderly and functional disabled people

This project have focused on developing and building a robust ISR platform able to perform different tasks that are stated within the functionality framework of an ISR. The purpose have been to participate in the international contest 'Robocup@Home - German Open'

2 Goals

Since one of the main purposes of the project have been to participate in the contest Robocup@Home in Germany, the goals of the project have been specified accordingly. The aim and the goal have been to beat the world record in one of the contest branches called "Go and Grasp". In order to achieve this, the following sub-goals were formed within the project:

- Build a mechanical structure that will serve and function as the base of the robot
- Build a simple 2-DOF gripper to pick up objects with
- Vision system able to detect a predefined object

- Navigation- and localization system that will enable the movement towards the object, based on the vision- and encoder inputs.

Other goals that were not included for this year, but are included for the upcoming years are the following ones:

- Vision system able to detect multiple objects
- Speech recognition - Focus on talker and respond logically
- Follow John - Recognize John and follow him
- Participate in German Open RoboCup@Home 2016
- Build and implement two multi-DOF arms
- Complex grasping hand
- Grasp objects of different sizes with two arms and two hands
- Participate in Embedded Conference and serve coffee and bun
- Library based object detection and grasping

The work and goals that remains are planned to be divided into the Robotic course, Applied AI course and as Thesis work.

3 Requirements

Since one of the long term goals is to participate with the robot in the contest "Robocup@Home", some requirements are derived upon the regulations and the rules of the contest. The rulebook for Robocup@Home 2014 [2] have been used to extract the most useful information and form the requirements.

- The robot shall be able to navigate through dynamic environments
- Object recgoniction under natural light conditions
- The robot must be autonomus and mobile
- The system must use closed loop control
- The dimensions of the robot shall not exeed the limits of an avarge door, which is 200cm by 70cm in most countries.
- Shall be able to cope with environment and walls that are 60 cm high at minimum.
- Provide an easily accessible and visible emergency stop button, and this button has to be colored red, and when pressing this button the robot has to stop immediately
- Provide a start button to start the test if the robot isn't able to open the entrance door to the room
- The robot's internal hardware (electronics and cables) should be covered in an appealing way
- The robot may not have sharp edges or other things that could severe people.

- The robot should not permanently make loud noises or use blinding lights.
- Either the robot or some external device connected to it must have a speaker output plug. It is used to connect the robot to the sound system so that the audience and the referees can hear and follow the robot's speech output.

In addition, some requriements were formed internally without taking the contests and its rules into consideration, these were the following:

- The robot shall be able to detect a can located on a table at a distance between 0.7m to 3 m.
- The robot shall be able to drive upwards a 20 degrees incline.
- The gripper shall be able to lift and hold an object of 500 grams.
- The robot shall be able to pickup objects at different heights, including the floor.
- The onboard battery shall provide the robot and the system enough power to run the robot for two hours in normal operating mode.

The requirements that have been specified will be going thorugh more deeply within each of the sections in this report.

4 Limitations

The system running on the robot is limited to work on one Intel NUC computer. The vision system will be pre-trained with help of photos of the can that it assumes is located on a table. The robot's initial pose will start by standing still and looking at the object in such way that it doesn't need to move the base and search for the object.

The environment is clean and no obstacles will be around or in front of the robot. No other things will be located on the table more than the object itself.

The lighting condition was limited to the same through the whole project.

At the startup phase the robot will be placed in such height that it doesn't need to change it in order for the gripper to to grab the object placed on the table.

5 System description

A system model have been design according to the needed equipments that was derived upon the requirements and the functionality of the robot. In figure 1 the system model is represented. The dotted lines and boxes in the figure represents future implementation of equipment and functions that the robot will have, whereas the full lines and boxes represents how the system is constructed at this point. In the figure, there is five different colored boxes that represent the system modules that the project have been divided into.

The following five different system modules are:

- Main computer - Intel NUC
- Vision system - Kinect V2
- Wheel modules (x4)
- Torso module
- Power Supply Unit

At the top level (Orange box in figure 1) there is a personal computer called NUC, running an Intel Core i7. On this computer the core program that handles all controlling and actions of the robot is running. The software for this program is developed in Simulink and MATLAB. The Simulink model that is running on the computer is communicating with the other different system modules through USB interfaces that are available on the computer.

The vision system (Blue box in figure 1) is based on a Kinect V2, that is equipped with an RGB camera and an IR sensor, moreover it also have a microphone array located on the bottom structure. The image and depth transferring to the computer is done through the USB 3.0 cable. The image analysis and the object detection is then handled in MATLAB.

The controlling and the movement of the robot is ensured by the four different wheel modules (Red box in figure 1) . The setup is the same for each of these and each of these modules is connected via a CAN-bus up to a CAN-translator which in turns is connected through a USB serial cable to the computer and the Simulink model. Each of the wheel modules is equipped with a CAN-card that acts like the central controlling unit for each of the wheel modules. Two motor controllers is connected to this CAN card, one motor controller for the motor that drive the robot in forward and backward direction, and one motor controller for the servo motor that turns and set the wheel in a specific angle. For each of the motors, there is also an encoder connected to the CAN-card. These encoders will track the speed and the angle for each of the wheels on the robot.

The torso module (Grey box in figure 1) is equipped with a CAN card that also serves as a central controlling unit. For this module, there is one motor that controls the height of the torso, and one motor that controls the opening and closing of the gripper. Each of these motors is connected to a motor controller that in turn are connected to the CAN-card. To track the height of the spine, an encoder is used on this motor, connected to the CAN-card.

The Power Supply Unit (PSU) (Green box in figure 1) is delivering the right amount of power to the different system modules. At this point, a simpler version of the PSU is used to power only the wheel modules and the motors. The intention and the plan is to use a CAN-card on the PSU to be able to communicate with the battery system via the CAN-bus from the Simulink model.

6 Planning

The project structure that was decided to be used for this project was a pool-based structure, with four different pools, namely Software, Hardware, Commu-

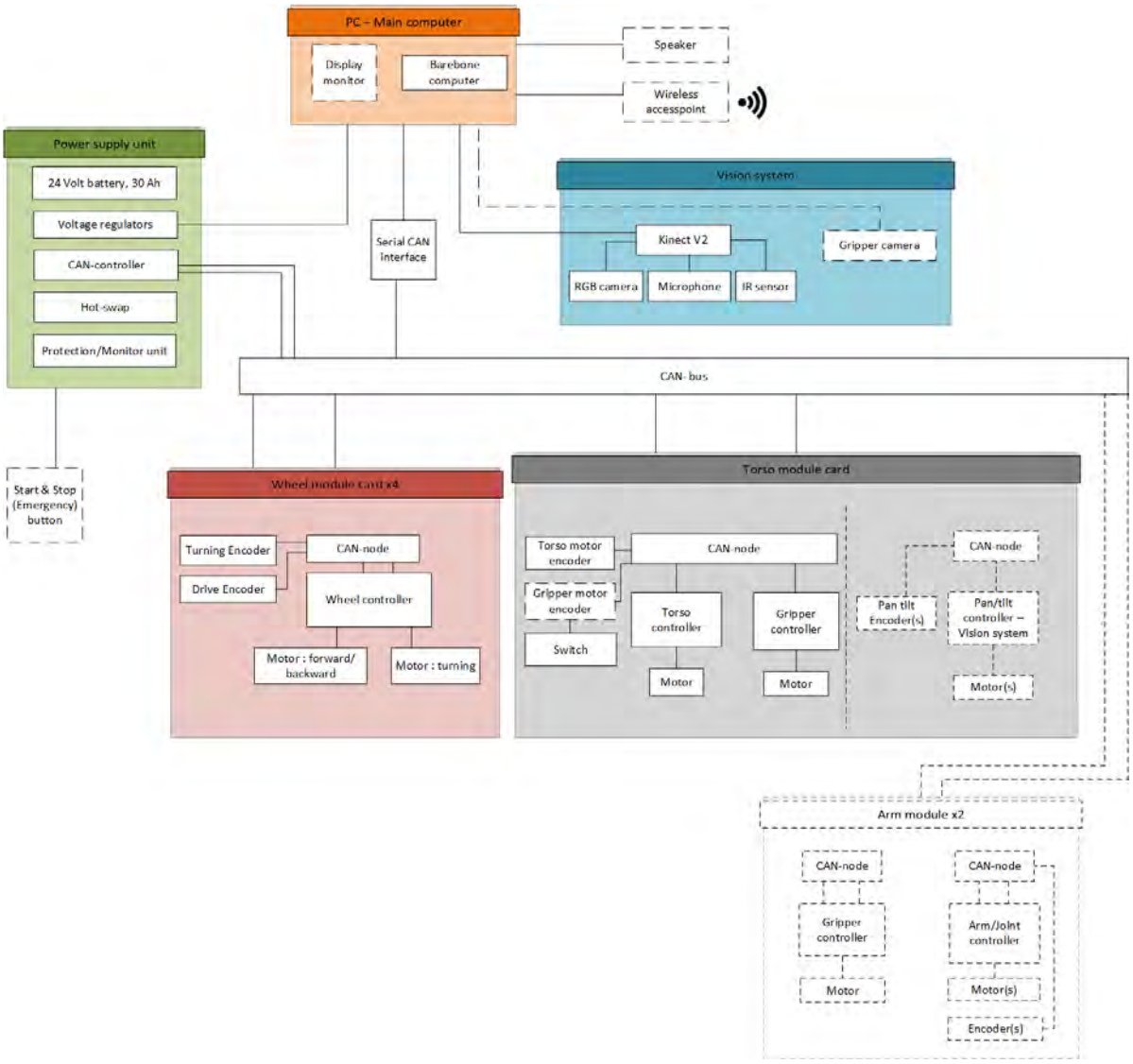


Figure 1: System overview of Butler

nication and Mechanics. The resources within these pools were shared between three other projects according to a priority list, where the Butler project was prioritized as number 2 of 4. Since no previous experience of this kind of project structure and it's organization, rough estimates of the different tasks and their deadlines were made, most based on what the student that was decided to work on the specific task felt and thought. The planning of the different tasks were made and categorized accordingly to the different pools.
A list of the different tasks can be seen in table 1, a more detailed version of the time-plan in GANTT format can be found in the appendix. As the project

evolved, different things such as soldering new CAN-cards popped up that was not planned from the beginning, and have therefore not been included in the time-plan.

Category Pool	Task	Time[h]
Hardware	Motor Controller	432
	Power Supply Unit	432
	Encoder	120
	Test and Verify hardware	96
Software	Design and define software structure	136
	Hardware spec of PC, Setup PC and environment, Kinect and the bridges	140
	Develop Kinematic model for the robot. Controller and Plant	96
	Actuator controller: Gripper-wheel and torso motors	160
	Vision software, Object detection	240
	Go and Grasp task	160
Mechanics	CAD Gripper	80
	CAD Wheel Module	160
	CAD Base	160
	CAD Torso	160
	CAD Manufacturing	392
	Mounting parts together	136
Communication	Implement SSI protocol for encoders	200
	Implement communication protocol between main computer and CAN cards	32
	Wheel controller software	56

Table 1: Time estimations of the different tasks

7 Mehanics

7.1 Base structure - Anders Olsson

7.2 Wheel module - Rickard Holm

7.3 Torso - Tobias Kriström

7.3.1 Torso and spine construction

It was early decided that the torso and spine should be height adjustable like a telescope so there is no parts sticking out on top when the torso is at its lowest height. For the spine, the possibility of using legs from height adjustable tables was early mentioned but at that point the rest of the group felt that a custom designed solution similar to the table legs is a better option.

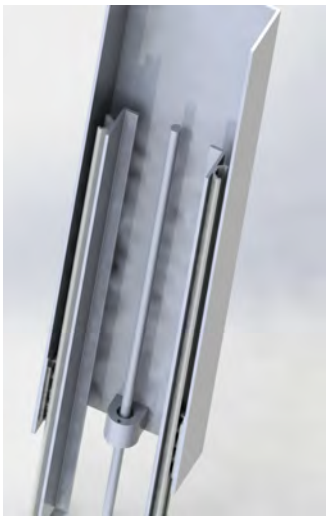


Figure 2: Spine concept - section view

Shown in figure 2 and figure 3 is the concept made for the spine, it has one inner part, a C-profile made of steel to be stable and an outer shell made by a rectangular pipe that acts as the height adjustable part. Figure 2 is a section view of the spine, showing two of three guide slides and the trapezoidal screw and nut that is needed to lift the outer shell along the C-profile. Backsides to this concept is that it is more or less impossible to find profiles for the inner and outer parts that matches each other so that guide slides can fit in, which means that either the inner or preferably the outer shell has to be custom made. That is expensive and takes a lot of time to get manufactured.

But luckily, the project leader managed to get a height adjustable table from Kinnarp as a sponsorship and the table legs, shown in figure 4, are very sturdy and seems perfect for the job. With an adjustable height in between 55-120cm, able to take a payload up to 50 kilos suits the demands in this project. So in the

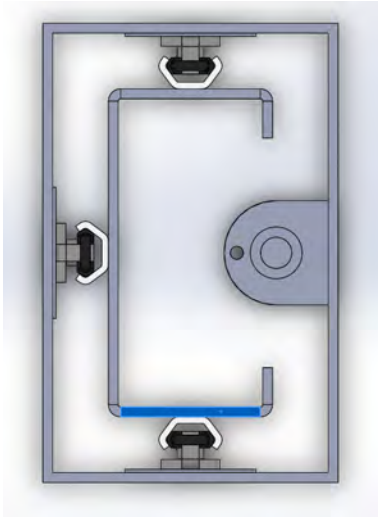


Figure 3: Spine concept - top view

end the table legs were set to be used for the spine instead so the design above will be the plan b.

In figure 5 is the existent connection for the motors that came with the table. Made in plastic, it has a very high level of play which affects the accuracy when adjusting the height. In figure 6, the plastic connection has been removed. Under the plastic part is a metal part, similar to "half a coupler"(not visible in picture) and underneath that is a six-edged rod with a standard M4-threaded hole. After examining this rod it seems to have a lot less play. The first idea was to make a new connection to this rod rather than using the old connection, however it seems like it did not provide enough support around the rod and ended up breaking the table leg. Instead, by using some of the old parts, an even newer connection was made that connects to the old parts that in turn connects to the table leg thus giving the support intended.

The first connection made, shown in figure 7, has one side with M4 standard threads and one cylindrical side. The threaded side will be attached to the rod inside the spine and locked in with Loctite glue. The cylindrical side will be connected to the shaft of a stepper motor by a coupler. This is the second design of the connection. The previous, although a lot stronger, was too complicated for companies to produce, whereas this design shown above is weaker but easier to produce.

In figure 8 is the newest connector. The old parts connects to the six-edged rod inside through a six-edged hole. This new connector is just like the six-edged rod but used on the other side and connected to the coupler via the cylindrical surface. It is a much more robust than the previous version and arguable a much more reliable solution since it uses old parts that were meant to be used with the table leg. To find the motor to use the momentum needed to lift the torso up with a weight of 30 kg has to be known. By using the formula

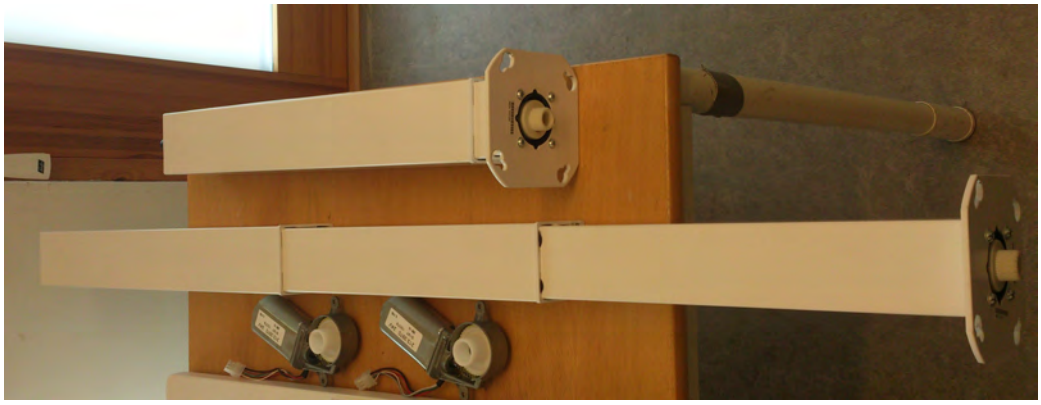


Figure 4: The table legs

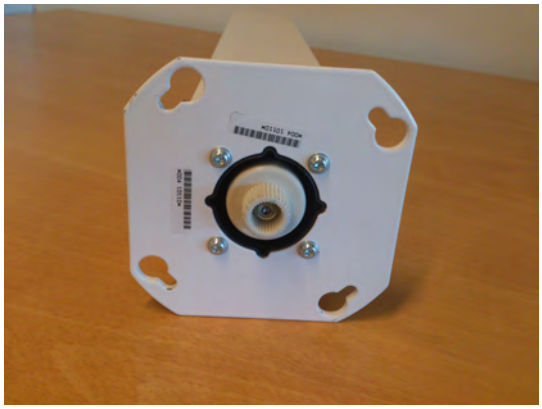


Figure 5: The outside of the old connection

from the engineering toolbox, it was calculated to be 1.19Nm. [6]

7.3.2 Simulation

Depicted in figure 9 is the simulation of the bearing construction, the torso, which sits on top of the spine made by aluminum square pipes. This simulation assumes arms of a weight of seven kilograms which is set as minimum requirement even though the arms might in the end weigh more. As can be seen in the picture some areas becomes green in the stress test and some areas, though not visible in the picture, becomes yellow.

In figure 10, the same design was tested but this time using steel square pipes. And instead of assuming minimum weight of the arms, this simulation is run with 15 kilograms instead of seven kilograms. Few, bearably visible areas becomes slightly light blue due to stress in the material but it is not deemed to be anything critical.

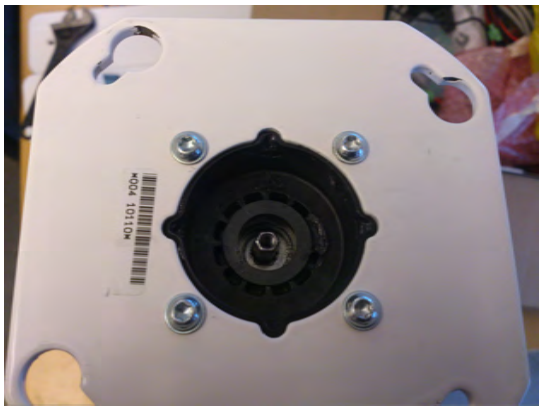


Figure 6: The inside of the old connection

Figure 7: Previous connector

Figure 11 shows the stress levels in the first made connector. Small areas are red indicating high levels of stress and larger area is green indicating “medium high”levels of stress.

In figure 12 the stress levels of the newest connector is considerable less. Where the six-edged rod meets the cylindrical face the simulation shows higher levels of stress. This is due to the fact that there is no fillet in the simulated part, however the part manufactured by Volvo has a fillet around this area so the stress is insignificant.

7.3.3 Cables

The suggested solution for getting cables up to the torso is by using cable chains. When looking at machines that uses linear actuators such as computer numerical

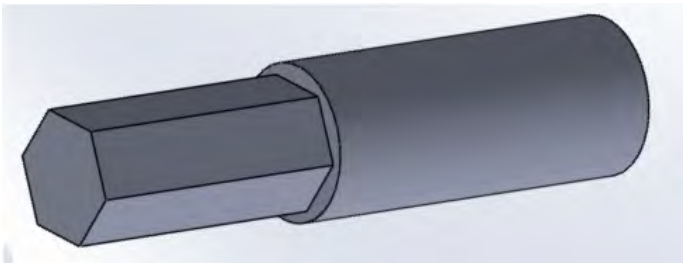


Figure 8: Connector used

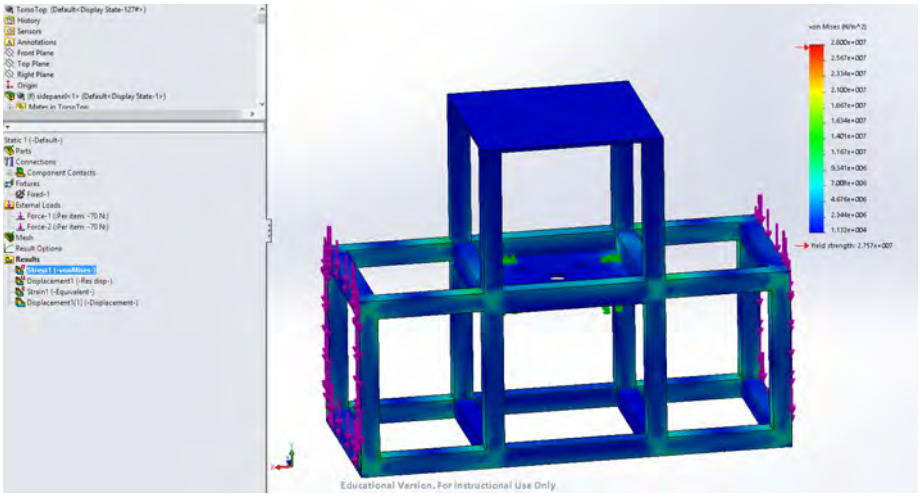


Figure 9: Aluminum torso - stress test

control systems that are available at the market, cable chains seems to be widely used. This is due to their predictable behavior, since they most often only bend in one direction.
This solution is only suggested and yet not set as to difficulty in predicting future needs. Since the arms that will be used later in the project does not currently exist there is no available information about their power consumptions or the dimension of the cables needed for the arms there is no way to certainly say what the necessary dimension of the cable chain should be.

7.3.4 Result

The bearing construction on top of the torso was first thought to be made by using aluminum square pipes since the base was already designed with the same pipes, but the simulation showed elevated signs of stress indicated by green and yellow areas. So the idea of using aluminum square pipes was dismissed and replaced by steel square pipes. The simulation of the structure when using steel square pipes was very promising and showed that it could

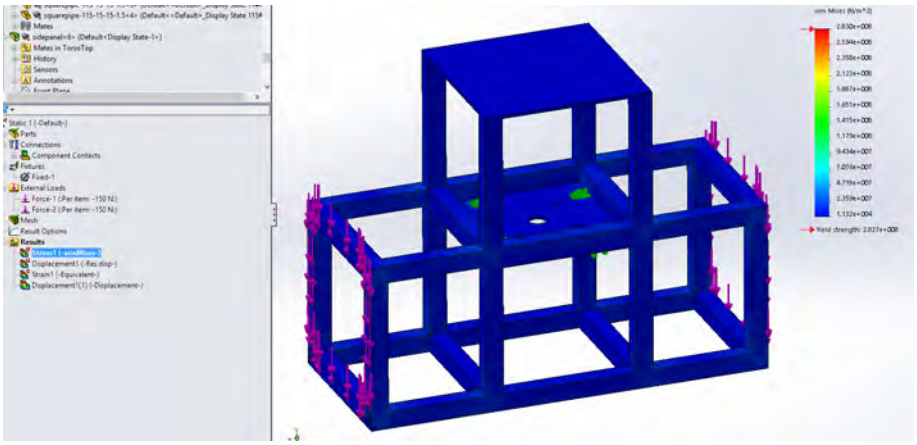


Figure 10: Steel torso - stress test

easily take double the weight of the proposed arm weight and steel has a lower cost than aluminum. So that is what will be used. Overall, the design of the torso is decided and awaits to be put in construction and the spine is already at site so it is ready for construction.

Depicted in figure 13 is the torso fitted to the spine where the green area is the boundary box for electronics. The arm in this picture is made of an aluminum profile but will be changed for a cheaper solution.

7.4 Arm and Gripper - Rickard Holm

8 Hardware

8.1 Power Supply - Eric Vieyra

8.1.1 Introduction

The design, production and user manual of the Butler's power supply is presented in the following document. This document assumes that the reader has access to the power supply's Multisim and Ultiboard files.

8.1.2 Description

As it name implies, the power source is the subsystem responsible with providing the necessary energy to the rest of the robot. By using the energy from a Nilar 24V battery, the power supply must be able to provide the power to a small Intel NUC computer, the motor actuators, a kinect device, power up a CAN bus network (with its corresponding logic devices) and supply power to the inner uC logic of the system itself.

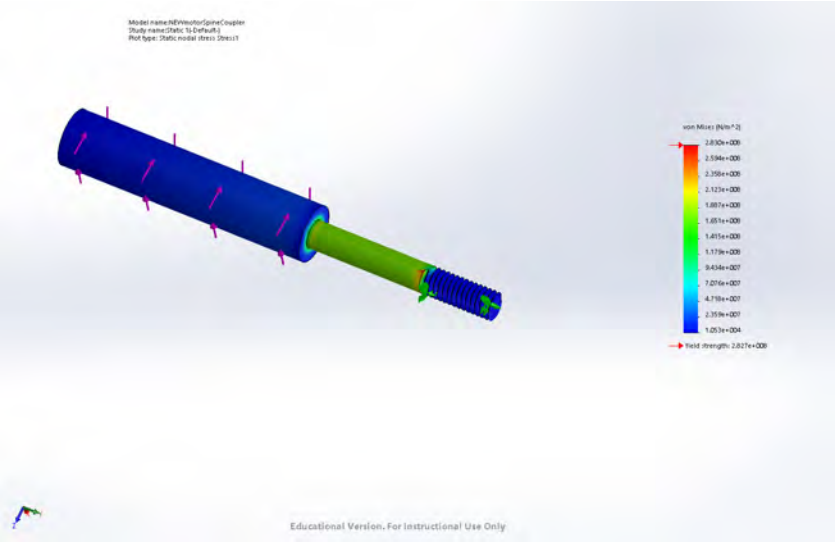


Figure 11: Previous connector - stress test

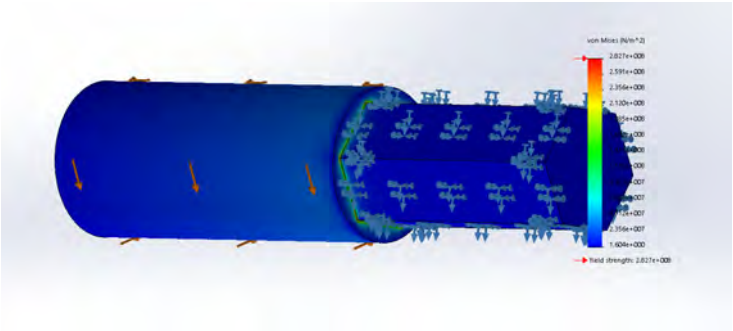


Figure 12: Connector used - stress test

8.1.3 Requirements and Limitations

The PCB was designed with the same modular design principle as its schematic. A great amount of the PCB's area is employed by thick power transmission traces since the system needs to support currents above 30Amps.

8.1.4 Design and Interface

The power source was designed with modularity in mind.Some of the sub-circuits were based in pre-existing circuits from the NAIAD's power source. The modular approach that was used during this project course ensures that the power source can be escalated in a relatively simpler way, it's testing is faster, easier and it's structure is more intuitive.

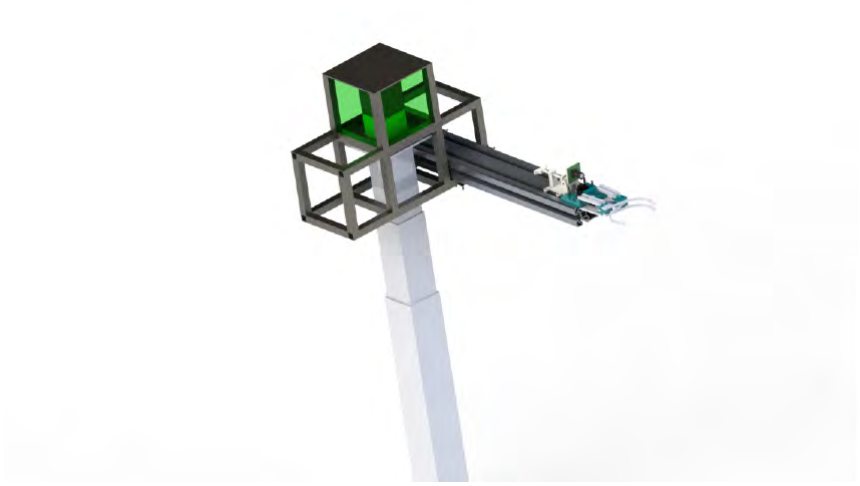


Figure 13: Torso, spine and arm assembly

Design characteristics

Inputs * :	24V @ 30A			
	24V @ 30A			
Battery:	24V @ 30A			
Outputs:	24V @ 30A	x6	Purpose:	Motors
	24V @ 30A	x1	Purpose:	Can Card
	19V @ 6A	x1	Purpose:	PC
	12V @ 6A	x1	Purpose:	Kinect
	5V @ 6A	x1	Purpose:	Logic

Protections:

Short circuit:	YES
Reverse polarity:	YES
Over-voltage:	YES
Reverse current:	YES

Measurements:

Current:	YES
Voltage:	YES
Software operated:	YES

Table 2: Design characteristics

Each one of this subsystems, as well as other secondary modules were implemented in NI Multisim as subcircuits. This approach encapsulates similar

PCB characteristics

Size:	350 x 140
Layers:	2
Max Nominal I:	30A
Inputs: Battery Inputs:	2
CAN-Card Dock:	1
RS485 battery com ports:	2
+24v Motor Outputs:	6
+24v CAN-Card Outputs:	1
+19v PC Outputs:	1
+12v Kinect Outputs:	1
+5V VCC Outputs:	1

Table 3: PCB characteristics

and related subsystems and makes the whole design easier to understand.

The power source supplies the rated current and voltage on each of its outputs. Nonetheless, the shown current is the max current that can be provided, and, since the battery cannot supply more than 30A of sustained current (without suffering any damage) the rated fuse that is used for protection has a value of 30A. The system is designed so that the specified channels do not work at their maximum capacity all the time.

The subcircuits that make up the power supply are the following: SafeFets, BatSwap, Voltage measurements, I measurements, transistor array, smartswitch single, buck 5v, buck 12v, buck 19v, module switch, mcu connections, rs485. An overview of each of this circuits now follows:

SafeFETS

Description:

This subcircuit protects the system against inverted polarity and remanent currents that may arise under certain conditions.

Requirements:

Provide reverse current protection to the system in an efficient way. This subsystem must be able to withstand strong inverted voltages with minimal power waste.

Design and Interface:

3 Mosfets are connected to each battery in a configuration that only activates the gate of the Mosfet when the correct polarity prevails in the system. For an incorrect polarity, the Mosfets will act as an open circuit.

Tests and simulation results:

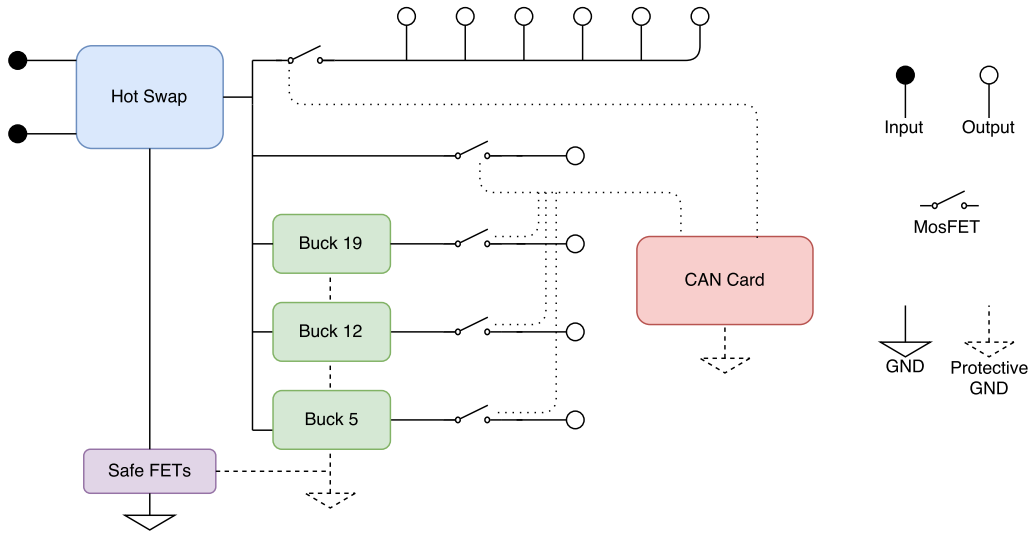


Figure 14: A simplified power supply diagram

The subcircuit was produced and tested as a separate board in conjunction with the rest of the power source. The tests results were successful and the circuit performs as expected. For inverted voltages, the system acts as an open circuit, providing the first protecting layer to the rest of the circuit. The power dissipation was lesser than 500mW.

Unit test:
When testing in the complete power source system, apply an inverse voltage to the power source. A voltage of 0 Volts should be present at the input of the selected battery swap

Using the system:
The transistors must be connected as shown in the schematic of the sub circuit. There are no extra operational instructions that must be met in order to achieve a correct performance from this sub circuit.

Notes:
-

Versions:
Version 0.6 of the power supply uses 3 transistors in each pair in order to distribute power dissipation, nonetheless, after several tests, we came to the conclusion that two transistors per pair are more than enough. Version 0.7 contains only 2 transistors per pair.

Future work
-

Hot-swap
Description:
As the battery gets discharged, the power source can connect to another previously charged battery and perform a so-called âĂIJHot SwapâĂİ without interrupting the power supply to the systems loads. This functionality breaks down in 2 tasks: provide a steady amount of current during the swapping and isolate the batteries from each other.

Requirements:
Isolate the batteries from each other and maintain an acceptable power level to the load during the Hot Swap phase. Do so by emulating the behaviour of a perfect diode. This is, the losses by power dissipation that the normal diode has are not acceptable. The integrated circuit used can withstand voltages up to 48V, this limitation must be kept in mind in future design in case that the battery voltage must be changed.

Design and Interface:
An OR-ing circuit can be easily implemented with diodes, but the voltage drop of the diodes constitutes an unacceptable loss. To fix this, the LM5050-1 integrated circuit is used in order to emulate the behaviour of a perfect diode. The circuit that is found in the Hot-swap module is adapted from an application circuit from the LM5050-1 datasheet. When this circuit detects a voltage in its input stage, it controls the MOSFET in order to supply current to the load. This ensures a steady supply to the load in a parallel way and also isolates the batteries from each other with almost no voltage drop.

Tests and simulation results:
This subcircuit was tested in various iterations as a separate board. The first iterations suffered from high power dissipations. The final iteration (used in the power source 0.6) can withstand the currents necessary for the robot at a minimum power loss.

Unit test:
Connect a battery or a power source to any of the inputs of the circuit. The power supply should work without any problem. Connect two voltage sources to the inputs and the complete system will also work accordingly. Lower the voltage of one input and the other input should continue providing power to the system without any glitch.

Using the system:
Connect a 24V 30A battery in any of the twin terminals of the power source. When one battery is running low, you can connect another battery in the remaining terminal, the system will automatically take care of guarding the supply against a power spike and will act as an ideal OR-ing circuit while the two batteries are connected. The low-voltage battery can be disconected at any time once the replacement has been connected

Notes:
The LM5050-1 used in the multisim and ultiboard files is a custom component. Two power MOSFETS were used in the final version of the HotSwap in order to increase the throughput of the system at an acceptable (very small) power loss rating.

Versions:
The first version of the circuit used only one transistor, the second version used four of the same transistor model. The final version used two automotive power MOSFETS. Only the final version was implemented in the power supply 0.6 and beyond, The rest of the versions were never implemented.

Future work
-

Voltage measurements
Description:
In order to monitor the voltage levels of the different outputs of the source, the corresponding nets are redirected to this measuring circuit through a voltage divider and a clipping zener. The measurements are then read by a uC unit.

Requirements:
The subcircuit needs to measure the voltage of 6 stages: Battery 1, Battery 2, 24V-Motor Voltage, 24V-Can Voltage, 19V-Computer Voltage and 12V-Kinect Voltage. The output are 6 analog scaled voltages.

Design and Interface:
Six voltage dividers are implemented in order to scale the voltages to a value between 0 and 5 volts. A Zener diode is used as an extra regulating protection (Zener voltage clipper).

Tests and simulation results:
The voltage divider values were calculated and simulated in multisim. The intended behaviour is to produce a voltage range from 0 to 5 depending on the value of the voltage to measure.

Unit test:
For each voltage measurement, a corresponding voltage between 0 and 5 should be present at the corresponding I/O pin of the MCU card.

Using the system:
The voltage values need to be read by the analog inputs in the MCU. This functionality needs to be implemented in the software.

Notes:
-

Versions:
-

Future work
-

Current measurement
Description:
To measure the overall current consumption of the system.

Requirements:
The system needs to measure in an accurate fashion while being robust enough to withstand the current.

Design and Interface:
An ACS756xCB sensor is used to measure the current. The capacitors in the subcircuit are recommended in the application sheet as noise removal components. The sensor produces a proportional voltage signal that is read by one of the ADC inputs in the CAN-Card.

Tests and simulation results:
While testing the system and verifying the results with an Amperemeter, the system produced the expected proportional voltage that it was designed for.

Unit test:
While operating the power source, an applied current between 0 and 100 A will produce a proportional voltage within 0 and VCC.

Using the system:
-

Notes:
The ACS756xCB is a custom component in Multisim.

Versions:
-

Future work
-

Transistor array
Description:
To turn on and off the power stages that they are connected to.

Requirements:
The module needs to control the power supply to the branches they are con-

nected to.

Design and Interface:
The BTS555 is a Power MOSFET transistor that can be controlled by an enable pin. When this pin is driven to 0 (GND), the device lets the power pass through it up to it's output. Two of this device were used in order to increase the current capability and heat distribution.

Tests and simulation results:
A separate board testing this circuit was developed in order to test the power dissipation and functionality of the system. The tests were accomplished in a successful way.

Unit test:
Activating the enable terminal on the transistors with a logic 0 should cause the array to output the voltage it has on its input.

Using the system:
The transistor array is operated via software by the MCU.

Notes:
The ACS756xCB is a custom component in Multisim.

Versions:
-

Future work
-

SmartSwitch single
Description:
To enable and disable the branch that the system is connected to.

Requirements:
The subsystem needs to power on and off the desired output of the power source. This subsystem also needs to provide protection against remanent currents that may arise from the inductive loads.

Design and Interface:
The BTS50085-1TMA is a Power MOSFET transistor that can be controlled by an enable pin. When this pin is driven to 0 (GND), the device lets the power pass through it up to it's output.

Tests and simulation results:
Just as the transistor array, the single transistor used for the motors was tested as an individual board. The test showed that the solution was viable and effective both in functionality and power conservation.

Unit test:
Activating the enable terminal on the transistors with a logic 0 should cause the array to output the voltage it has on its input. An open circuit causes the switch to deactivate.

Using the system:
The transistor array is operated via software by the MCU.

Notes:
-

Versions:
-

Future work
-

Buck5, Buck12 and Buck19
Description:
From the 24V input from the battery, a down conversion to 19, 12 and 5 volts is performed by the respective subcircuit.

Requirements:
The devices that need to be powered with a different voltage other than 24V are: The computer (19V, 6A), Kinect (12V, 1A) and uC(s) (5V, 0.5A).

Design and Interface:
The LM2678S-x is a buck converter that lowers an input voltage into a fixed, predefined voltage. Depending of the integrated circuit bought, the voltage can be lowered to 3.3, 5.0, 12 or a variable voltage. The integrated circuit is wired up with it's corresponding components in accordance to the datasheet.

Tests and simulation results:
The buck converters were not tested and or simulated. After encountering problems with the net power they could output, simulations from the TI page suggest different values in order to achieve a bigger power output. **While testing the component in the power source, the Buck converters could not provide the 5A currents that we tried to get from them.** This may be caused by sub optimal layouts and different values necessary.

Unit test:
The Buck converters can be tested by measuring the output terminal's voltage. The voltage should be 5, 12 and 19, depending on the device measured.

Using the system:
As with most of the components in the power source, the Buck converters can

be activated by software.

Notes:
NOTE THAT THE LM2678S INTEGRATED CIRCUIT HAS A FIXED VOLTAGE DEPENDING ON ITS TERMINATION: The LM2678S comes in 4 variants: LM2678S-3.3, LM2678S-5.0, LM2678S-12 and LM2678S-ADJ. The LM2678S-5.0 and LM2678S-12 are used for the Buck5v and Buck12v subcircuits, and the adjustable LM2678S-ADJ IC with a fixed value of 19V is used for the Buck19v subcircuit.

Versions:
-

Future work
Improving the layout and choosing better values should be the main priority for a new version of the power source. New buck converters with a higher power rating can also be a good choice for a revised version of the power source.

Module switch
Description:
Control with digital signals and an Emergency button the behaviour of the previously mentioned array of transistors. Indicate with leds the current state of the system.

Requirements:
This subcircuit must generate the outputs that drive the transistor arrays in response to 2 digital inputs coming from the micro controller. As a safety measure, when the power is gone the arrays should be normally open. Also, an Emergency switch with the power to override every signal must be present. Indicators (Leds) need to be driven in order to provide an insight of the current state of the power source.

Design and Interface:
The 8L41-05-011 is connected with a BJT transistor in order to drive the relays that control the corresponding outputs. The emergency Switch S1 is connected in series with the Motor enable signal in order to be able to override the signal in case of emergency. The Leds are connected to the ULN2004 inverting Led-Driver. This IC is controlled by Digital IO of the micro controller in a low-state drive fashion.

Tests and simulation results:
The connections were tested manually by connecting to Vcc or Gnd the corresponding pins. The transistor arrays showed the intended behaviour.

Unit test:
Just as the transistor array, the corresponding voltage must be seen at the output when an activating signal is issued by software.

Using the system:
-

Notes:
Note that the 8L41-05-011 and the ULN2004 are custom components (special attention to the footprint).

Versions:
-

Future work
-

MCU connections
Description:
To connect the CAN card into the power supply.

Requirements:
The circuit must connect the desired uC lines to the rest of the supply and map them correctly to the corresponding footprint.

Design and Interface:
The CAN card is implemented as a multi-section component in multisim. This means that the Analog IO, the SPI, etc is implemented as multiple components (that share one footprint). A filtering cap and a voltage hold cap is also implemented around the CAN voltage connection.

Tests and simulation results:
This component was tested by controlling the rest of the power source and manually wiring the pins to the corresponding digital signals.

Unit test:
-

Using the system:
This is a software oriented subsystem. The pinout of the system is presented in this section.

Notes:
-

Versions:
Version 0.7 of the power source has a correction where the power pin from the MCU was routed to a different pin.

Future work

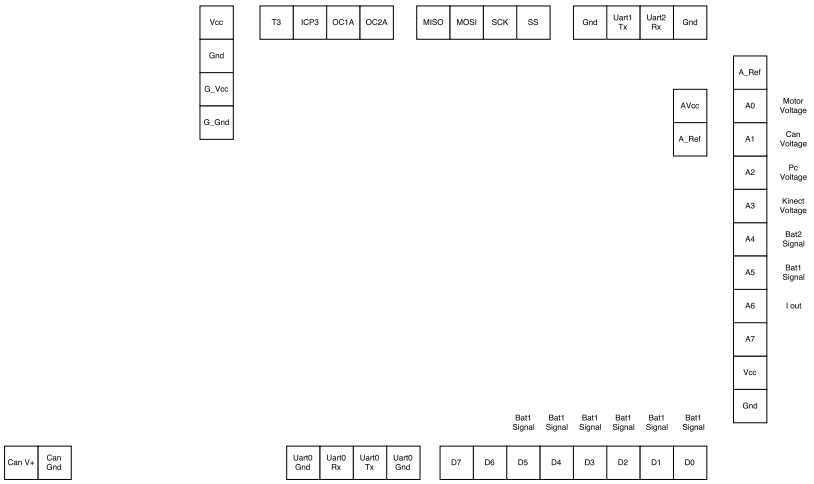


Figure 15: A simplified power supply diagram

-

RS232 RS485
Description:
To interface the uC's rs232 peripheral with the rs485 interface present in the Nilar batteries.

Requirements:
The subsystem needs to be able to convert rs232 signals into differential rs485 signals that the batteries can interpret. An integrated transceiver performs the signal conversion. The necessary wiring and signal filtering are provided.

Design and Interface:
The SN65HVD485 integrated circuit performs a conversion between rs232 signals and rs485 ones. The rest of the components are selected in accordance to the recommended values of the IC's datasheet.

Tests and simulation results:
This subcircuit was not tested

Unit test:
-

Using the system:
This is a software oriented subsystem. The pinout of the system is presented in this section.

Notes:
This circuit has not been tested and its final implementation is still not certain.

Function	Status	Notes
Hot-swap	Working	
Reverse polarity protection	Working	
24V supply (Motors)	Working	
24V supply (CAN)	Working	
19V supply	Working? (barely meets the power requirements)	max current 3.6A approx (3.4 required)
12V supply	Working (sub optimal performance)	max current 4.0 A
5V supply	Working (sub optimal performance)	max current 2A
Voltage measurements	Working	Corrections made in version 0.7

Table 4: Test and simulation results

Versions:
Version 0.7 of the power source has a correction where the power pin from the MCU was routed to a different pin.

Future work
-

8.1.5 Test and simulation results

Aside from the sub circuit tests, the power source was tested as a complete system while it was being built part by part. While the majority of the tests lead to a positive outcome, some design mistakes were discovered (discussed in the future work section). Unfortunately, while most of this mistakes could be solved using (not so elegant) hacks, the biggest design issue was the poor performance of the buck converters; These devices had a performance of around 60-70 percent which led to an inability to effectively drive the robot subsystems. The following table contains a summary from the results.

8.1.6 Using the system

Connections:

1. Connect the CAN card to the Can Connectors.
2. Connect the Motor Actuators to the Motor Outputs.
3. Connect the single 24V outlet to the Can Ring.

4. Connect the 19V outlet to the Intel NUC computer.
5. Connect the 12V outlet to the Kinect peripheral.
6. Connect one 24V battery to one of the two available inputs.

Voltage measurement pins:

A0	Motor voltage
A1	Can voltage
A2	PC voltage
A3	Kinect voltage
A4	Bat2 voltage
A5	Bat1 voltage
A6	I measure

Table 5: Voltage measurement pins

8.1.7 Future work

There are several aspects that need to be addressed, both as corrections or as improvements to the current design.

Corrections:

- Power pins for the CAN-Card are incorrectly routed
- The fuse footprint should be changed to a standard automotive fuse base.
- The Buck converter stages need to be redesigned
- The Smart Switch tab terminal is incorrectly routed to Gnd

Improvements (In order of priority):

- Correct the number of connectors (Add more connectors to the 12+ and 5+ supplies).
- Reduce size to facilitate production (make the PCB small enough to fit in the oven).

8.2 CAN card - Albin Barklund

8.2.1 Description

The so called CAN-ring is a system of cables which integrates a CAN bus and two power lines in a single housing. The purpose of the CAN-ring is to distrubute power throughout the system and provide communication between system modules. The follwing modules are connected to the CAN-ring:

- 4x Wheel modules
- Power Supply

- CAN to USB translator
- Torso and grippper module (in the future)

8.2.2 Requirements and limitations

Integrate the following into a common system of cables:

- CAN-bus capable of 1 Mbit/s data transfer.
- Power line capable of 24V @ 30A for motors.
- Power line capable of 24V @ 5A for electronics.

8.2.3 Design and interface

The CAN-ring is built up of a 3-pole shielded data cable together with several stranded copper wires wrapped up in spiral wrap tubing as can be seen in figure 16. The sheilded data cable constitutes the CAN-bus. The pair of $1.5mm^2$ stranded copper wires makes up the power distrubution lines for the CAN-controllers and misc. electronics. The pair of $10mm^2$ stranded copper wires makes up the power distrubution lines for motors and motor controllers. These two power lines are kept seperate in order ensure that noise from the motors controllers don't spreading to the electronics.

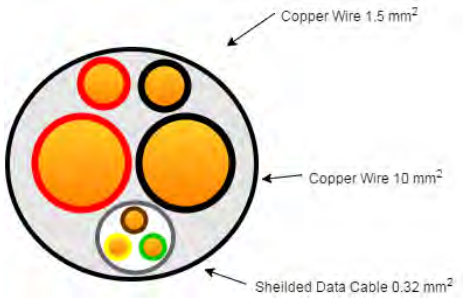


Figure 16: A cross section of the CAN-ring

Quantity	Description	Copper cross sectional area	Poles	AWG
1	Shielded data cable	$0.32mm^2$	3	22
2	Stranded copper wire	$10.0mm^2$	1	7
2	Stranded copper wire	$1.5mm^2$	1	15

The wires are cut into pieces which fits between the modules of the system according to the laoyut depicted in figure 18. The wires are then stitched together with a copper thread and soldered together with a third wire that breaks out from the CAN-ring and connects with the module as figure 17 indicates. A shrinking tube is then used to isolate the induvidual wires.

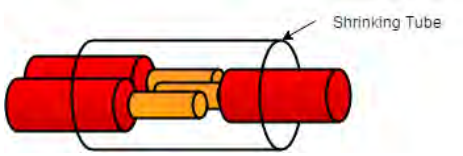


Figure 17: Solder

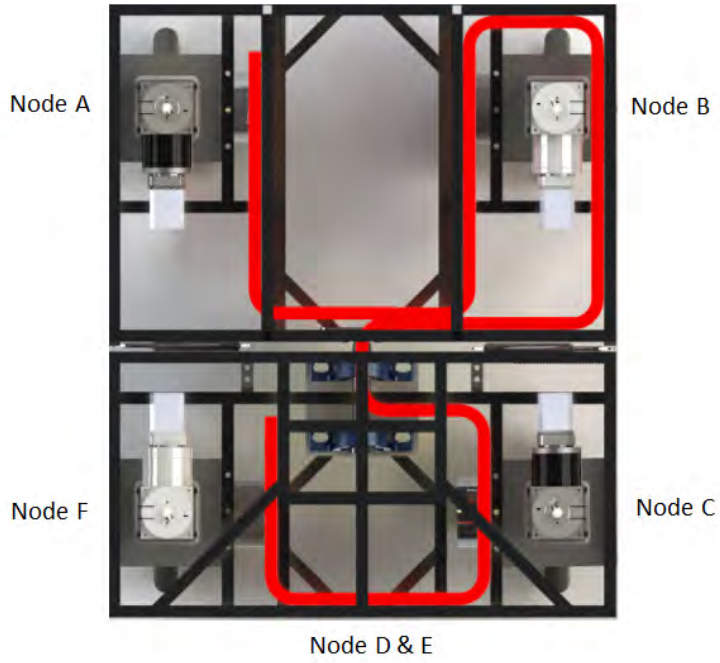


Figure 18: The layout of the CAN-ring

Modules

8.2.4 Wheel modules (Node A, B, C & F)

Amount	Type	Connector
2	Motor	Twisted and soldered
1	Electronics	Vertical connector w. flanges 3.5mm
1	CAN-bus	Female Terminal Housing 2.54 mm

8.2.5 CAN to USB translator (Node E)

Amount	Type	Connector
1	CAN-bus	Female Terminal Housing 2.54 mm

8.2.6 Power Supply (Node D)

Amount	Type	Connector
3	Motor	Twisted and soldered
1	Electronics	Vertical connector w. flanges 3.5mm
1	CAN-bus	Female Terminal Housing 2.54 mm

8.2.7 Unit test

There are two tests that can be performed to verify the functionality of CAN-ring, one test checks for short circuits and another test verifies continuity of the

cable system.

Short circuit There are 7 induvidual copper wires inside the CAN-ring. With a multimiter check that there are no conductivity between any of the different wires by trying all the 21 combinations.

Continuity Check for each wire inside the CAN-ring that there continuity at each node with a multimeter.

8.2.8 Using the system

Connect the power supply to both power lines at node D (3 connectors for the motors and 1 for the electronics) and to the CAN-bus. C

8.2.9 Future work

The CAN-ring needs to be extended to the torso and gripper module from node F.

8.2.10 Appendices

BOM

Name	Value	Vendor	Art.Nr
Shielded data cable	0.32 mm^2	Elfa	15563319
Stranded copper wire	10.0 mm^2	Elfa	15530613
Stranded copper wire	1.5 mm^2	Elfa	15533120
Spiral wrap tubing	5-20 mm	Elfa	15500545
Vertical connector w. flanges	3.5mm	Würth	691364100002
Vertical connector	5.0 mm	Würth	691352710002
Female Terminal Housing	2.54 mm	Würth	61900311621
Female Crimp Contact	2.54 mm	Würth	61900113722

8.3 Wheel module docking card - Albin Barklund

8.3.1 Description

The wheel module docking card connects the electronics of the wheel module together. The card could be seen as a cable replacement since it doesn't have any electronical components except for a few resistors. Its only job is to connect the following parts together for communication and power.

No.	Name	Value
2	Motor controller	BOOST-DRV8711
1	Stepper Motor	34HY1802-03
1	Stepper Motor	23HS2417
2	Rotary Encoder	AS5145H
1	CAN controller	AT90CAN128

8.3.2 Requirements and limitations

- The CAN-controller should to be able to control the stepper motors through the motor controllers and to read the sensor values through from the rotary encoders.
- The motor controllers, CAN controller and rotary encoders needs to be connected to power.
- The mechanical construction poses a physical constraint on the wheel module depicted in figure 19. It specifies a boundary box and holes for mounting.

Figure 19: Boundary box

Motor Controller The CAN-controller connects to the motor controllers through SPI, PWM and several I/O ports.

DRV8711	**CAN-controller**	**Comment**
3.3V	NC	
POT	AI	
nSLEEP	Logic High	
SCLK	SCK	same for both
RESET	DIO	same for both
STEP/AIN1	PWM	
DIR/AIN2	DIO	
GND	GND	
nSTALL	DIO	interupt
nFAULT	DIO	
SDI	MISO	same for both
SDO	MOSI	same for both, external pullup
BIN1	NC	
BIN2	NC	
SCS	DIO	

Rotary Encoder The CAN-controller connects to the rotary encoders through several I/O ports.

AS5145H	CAN-controller	Comment
GND	GND	
DO	DIO	
CLK	DIO	same for both
CS_n	DIO	
VDD	5V	
NC	NC	

8.3.3 Design and interface

Connections In figure 20 all the connection to the wheel module docking card are depicted, the motors and the power to the motor controllers are connected to the actual motor controllers. For actual pinout see the schematics and CAD files.

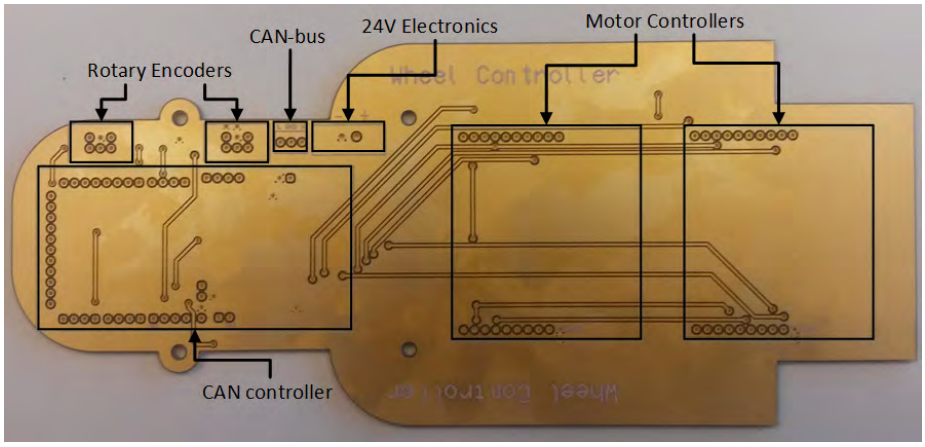


Figure 20: Connectors

8.3.4 Future work

A silk screen need to be added to both sides of the PCB in order to protect the and prevent short circuits.

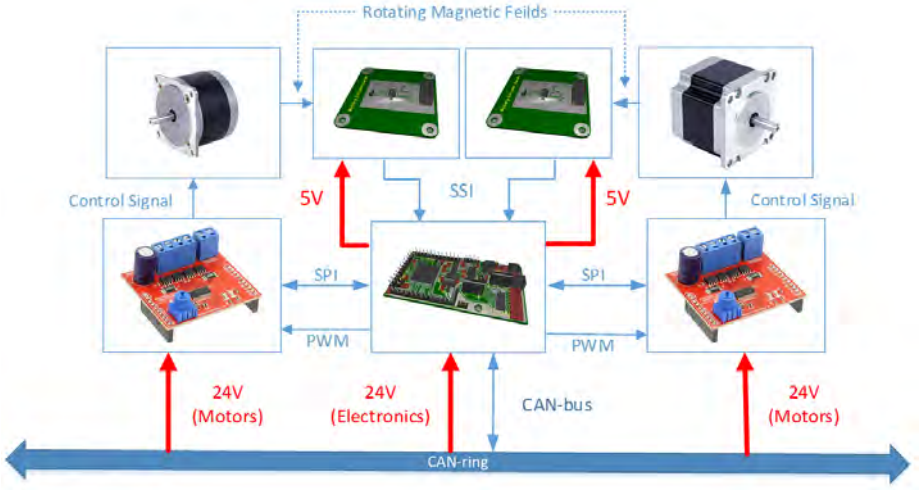


Figure 21: Subsystem overview

8.3.5 Appendices

BOM Bill of material

Name	Vendor	Ordernumber	RefDes	Value	Shape
2.54 mm Socket Header	Würth	61300911821	U1		DRV8711
2.54 mm Socket Header	Würth	61300911821	U2		DRV8711
2.54mm Male Box Header	Würth	61200621621	U4		Male Connector
2.54mm Male Box Header	Würth	61200621621	U7		Male Connector
2.54 mm Socket Header	Würth		U8		AN card with CAN pins
3.50mm Vertical with Flanges	Würth	691364100002	U5		3 pin locking header
2.54mm Male Locking Header	Würth	61900311121	U6		3 pin locking header
Resistor			R1	3.3k	R0603
Resistor			R2	3.3k	R0603
Resistor			R3	3.3k	R0603
Resistor			R4	3.3k	R0603
Resistor			R5	3.3k	R0603

8.4 Encoders - Roxanne Anderberg

8.4.1 Description and Requirements

To track the wheel motion of the Butler it is decided to use absolute encoders for the wheel systems. There are four wheel modules, each steered by two motors, one to turn the wheel, and one for rotating the wheels. There is also a motor to steer the torso. Every motor will have a rotary encoder card attached to them.

8.4.2 Design and Interface

The design was chosen according to the design in the operation manual [3] To have exact movement of the motors, the rotary encoder chip is attached in the center of the PCB. There are three capacitors connected to it fallowing the instructions from the datasheet [4] of the encoder card. The rotary encoder has 10 pins with different input/outputs but we have only decided to use five of them, ground, power, and the pins for the SSI (Digital Output, Chip Select and Clock). The PCB CAD design was made so that it would fit perfectly above the motors and the connector was placed at the edge of the PCB so that it would be easy to attach and detach and so that it would not be in the way of any mechanical parts of the wheel systems.

8.4.3 Versions

There are two different versions of the card, the only difference is the measurement for the holes since Butler uses two different motors. 37

8.4.4 Future Work

If there is need to make new encoder cards, make sure the measurement of the holes are correct. Measure the screw thread by hand.

9 Software - Jonatan Tidare

9.1 Simulink Model Overview

The purpose of this simulink model is to successfully move the robot to its goal. As seen in Figure 22, the Plant keeps track of the robots pose and goal pose. The controller reads the poses and determines how to set the motors for all wheels in order to minimize the distance to the goal. The CAN block is the interface between this simulink model and the real robot. It sends motor commands and recieves encoder data, and this encoder data is sent to the Plant. The Plant updates the position and the loop continues til the goal is reached. Every 0.5 seconds, the kinect sends a more exact pose of robot relative to the goal, and this information goes to Plant Car which updates the pose.
Together with this report is an attached file named "fromCADtoSimulink.pdf"

which describes all neccesary steps when importing a CAD model to simulink and getting Git version control for your project.

9.2 The Simulink Model

See Fig 23. This shows the highest level in the simulink model. The blocks are numbered and explained below.

1. Plant Car
2. Error introduction

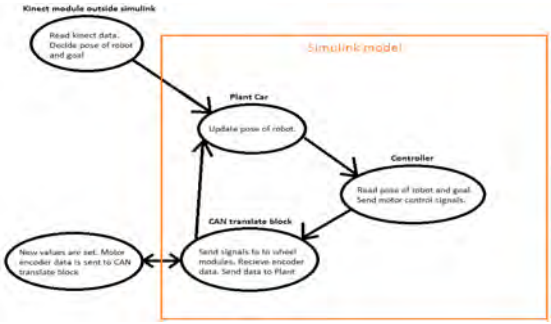


Figure 22: System full loop example.

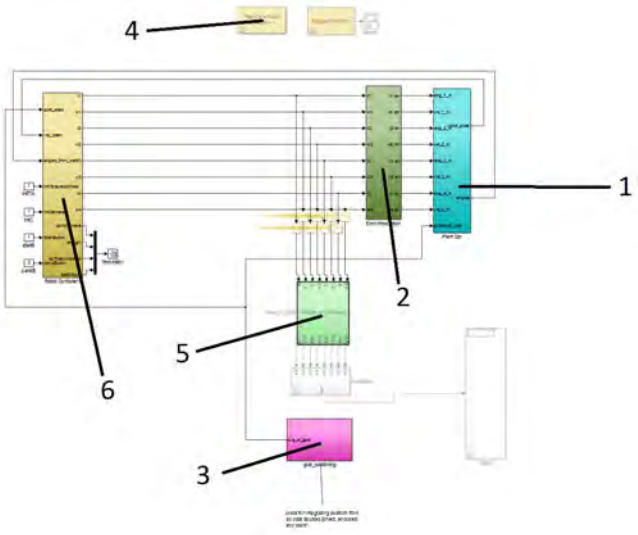


Figure 23: Simulink view of butler_base

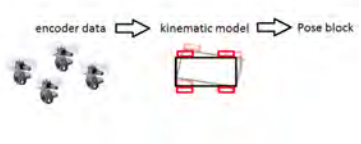


Figure 24: An example of how the kinematic block converts the encoder data to motion of the robot car.

3. Goal positioning
4. Realtime Pacer
5. CAN translate block

Note that the sixth block, the control block, is not described here. The description of the control block has its own part in the report.

9.2.1 Plant Car

The Plant Car (car was an old name for the butler base) mainly consists of a kinematic model and a pose block, where the pose block is the biggest function.

Pose Block. This block updates the robot position by recieving change in motion of the robot and adding it to the current pose. The pose (position and angle) is kept in a coordinate system where the goals pose is constant. The pose block also updates the pose from kinect data and keeps a bis buffer for better approximation of the kinect data.

Bis buffer. When kinect data arrives it is X millseconds old. In the bis buffer, the newest bis pose is compared to the bis pose from X milliseconds ago, giving a bis vector. This bisvector represents the robots movement from when the kinect picture was taken. By adding the bis vector and the kinect vector, a very good estimate of the robots pose is found!

check_excess_values This block simply warns the user if any motor appears to run too fast. This block was used during testing when control system output was directly fed to the plant.

Motor_Limitations This block limits both the absolute values and the derivatives. Any signal exceeding the limit will be reduced to the maximum value inside this block.

wheel kinematics As seen in figure 24, The kinematic block contains a model which converts the encoder data from the wheel modules into a motion of the robot base. This change in motion is then fed as input to the Pose Block.

Generate kinect data This block is used to generate kinect data. It is used to replace the actual kinect data and the conversion block from kinect coordinate frame to robot coordinate frame. These parts are mentioned in future work.

9.2.2 Error introduction

In this block, errors are introduced to the model. Delay and precision are seperated as two blocks. Delay is close to constant and can be measured, but is currently set to 0.2 seconds. Errors in precision are currently commented out but was used for testing the controller. Adding constant errors to motors is easily done and therefore this block has no variants. Currently, random errors from -0.1 to 0.1 are added to all motors when this block is uncommented. If constant errors are desired for testing, simply add them and run.

9.2.3 Goal positioning

This block generates a position for the goal. In the end product, the robot should first read a kinect frame and from that frame set the coordinate system such that the robot has pose (0,0,0) and the goal gets the pose (gx,gy,gth). Any kinect frame after this one will calculate the robots position from (gx,gy,gth) take the distance from robot to goal, then calculate the distance from the goal to the robots position.

9.2.4 Realtime Pacer

The real time pacer allows the execution to slow down to real time. The system needs real time to successfully control the butler. In addition, this Realtime pacer reduces cpu load on computer. Not all realtime packages do this. The reduced amount can be seen by running the simulation indefinitely with and without realtime pacer while watching the cpu usage on the computer.

9.2.5 CAN translate block

In short, the CAN translate block recieves motor control signals from the controller and sends them to the wheel modules through the CAN bus. This block also reads motor encoder data from the CAN bus and sends them to the "Plant Car" in the simulink model. Note that this block still experiences errors and the sending is not reliable.

type_conversion & m_to_cm. These small blocks change the data from the motor controller to the correct type. The CAN cards require angles in the span 0 to 180 degrees but the motor controller outputs angle in degrees, from -pi/2 to pi/2. The CAN cards require speed in cm/s but the motor controller outputs in m/s.

enable_on_input_change. This block senses change in any of the 8 motor control signals. If a signal change is detected, then it enables output for one sample period.

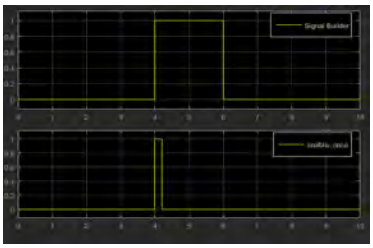


Figure 25: An example of a signal going through the block enable_once. The graph above is sent to the block and the signal below is the output from the block

Create modbus messages. This block takes 8 motor control signals and creates 4 modbus messages, one for each wheel module.

Make_one_message2. This chart sends modbus messages to all wheel modules, one at a time. This chart enables the block "Send_Message" and ensures that the correct message is sent.

enable_once. When this block gets a 1 as input, it outputs a 1 for exactly one sample period. See example in figure 25.

Send_Message. This is an enabled subsystem which can activate the Serial send block in simulink. The block must be deactivated in order not to send too many messages. When it is enabled, it will immediately send whatever message is at its input.

Serial Receive. This is the basic serial read block in simulink. It reads messages as they are recieved in the serial port.

Subsystem. Poor name for this. It decodes the serial message and reads the content. address tells which module sent the message. funcAddr tells what kind of message it is. dataByte1 and dataByte2 contains the info bytes. If the message is from the motor encoders, then the databytes contain speed and angle of the wheel. For other types of messages, see the modbus pdf.

read_outputs. This block reads info from decoded messages and sets its output accordingly. it has output for each wheel module, and the output for each motor is unchanged until a new message takes its place. This works well assuming that continuous data from the wheel modules is recieved.

After this block there are a few converter blocks which change the angle from degrees to radians.

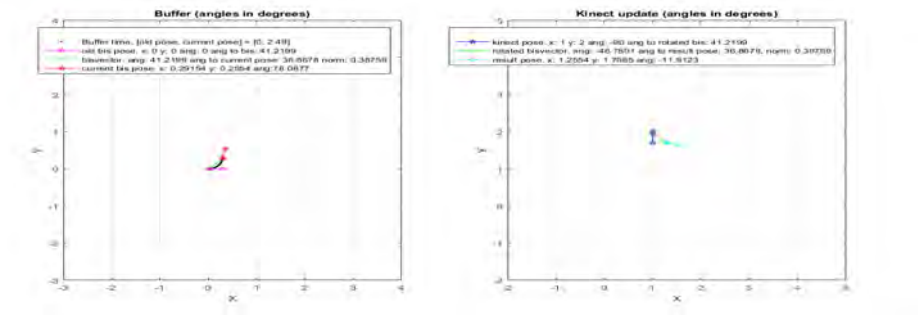


Figure 26: To the left: The bis vector (green) is calculated. To the right: The bis vector is added to kinect pose to find

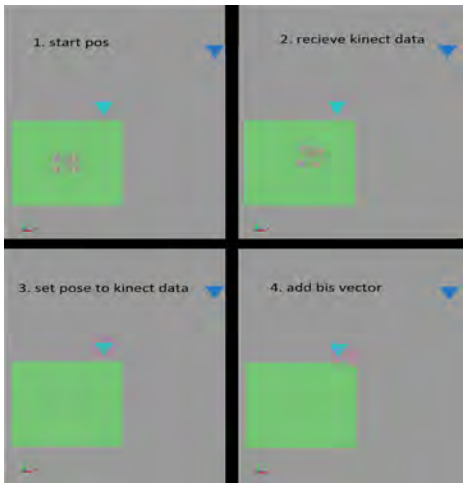


Figure 27: The kinect update phase

9.3 Examples

In this example we will use two figures, 26 and 27. These are taken from the same run and represent the same motion. The kinect data which arrives is not real kinect data, but comes from the Generate kinect data block (described in section Plant Car). The example generates a huge error in pose in order to more clearly show the update phase. All angles are in radians. Note that the goal is represented as a blue triangle and the kinect update is represented by a cyan triangle. Both these point downwards (angle is -pi/2 radians).

1. So, in this example, the robot starts in (x,y,angle)=(0,0,0). So, it is looking to the right as seen in figure 27 step 1.
2. The robot starts moving and makes a left turn. The robot is now pointing up-right as seen in figure 27 step 2.
3. The kinect data shows that the robot actually started in pose (x,y,angle)=(1,2,pi/2), and updates current pose. The robot now points downwards as seen in

figure 27 step 3.

4. The bis vector is calculated and added to the current pose. The robot now points down-right as seen in figure 27 step 4. The bis calculations can be seen in figure 26.

9.4 Future Work

These are the challenges we did not overcome and that must be solved.

9.4.1 The CAN communication

For future work, the CAN translate block must be reworked. We made several tests and work-arounds to get a reliable and steady communication but failed. We have tested several ways of sending just the modbus messages we want, and with a static, reliable delay between each message. This has not been achieved. When sending messages from a windows computer through the serial port, it may take anywhere between 10ms to 400ms for the message to be sent. However, the computer measures the times to be exact and with specified delay between. We believe that the function "Serial Send" made by Mathworks is bad. This is not the only possible way to send data through the COM ports, but it's the only solution we have tried during our project.

Another big issue is that the Serial Send block in simulink slows down the simulation a LOT! Make sure you watch the execution time and cpu usage of the model when working towards a solution.

To do: Find a way to send the modbus messages from simulink to the CAN controllers with reliable and short delay. One millisecond would be great. This is important so the motors get their updates fast and very much at the same time.

9.4.2 The kinect vector

As it is now, the model cannot recieve the kinect vector to the goal and update the robots pose. What must be done is a block which converts between the kinects coordinate system to the simulink coordinate system. There are ROS blocks in simulink that can be used to recieve data from ROS, which is how the kinect will send its data.

To do: Make a block to recieve the kinect data, convert it to the robots coordinate frame and find the pose of the robot in the Pose block.

9.5 Controller - Tobias Kriström

9.5.1 Overview

In figure 28 is the control system for the butler robot. The control system consists of several controller parts with each part marked with a number. The number will be referenced to when functionality of the respective part is explained. The

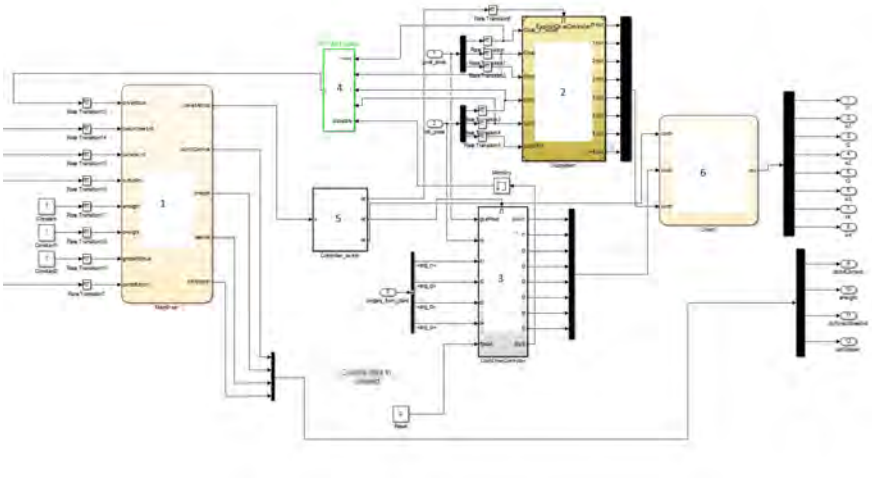


Figure 28: Control system - overview

purpose of this controller is to drive the robot from one position to another. Marked with the number 1) is the main brain. The main brain decides what to do and when it should be done. The explicit drive, number 2), is used to drive the robot over longer distances and uses a 'pose to pose'-type controller. When the explicit drive is done, the main brain starts the crab drive mode. The crab drive controller, number 3), is used to precisely adjust the robots position.
To decide when the exlpicit drive is done, a miner code block, number 4) is used. It checks when the robot is 'close enough' to its target and when the distance between the goal and the robot is within a certain margin, it sends a signal to the main brain to tell the main brain that the explicit drive is done.
The main brain has a single signal to switch between the explicit drive and the crab drive but since these two are implemented as enabled systems, logic in between is needed to turn two signals on and off from the one signal sent by the main brain. Both controllers has the same type of output, volecities and steering angles for each wheel. Just in case, a flow chart, number 6, is used. It takes the output from both the explicit drive and the crab drive and, depending on the signal from the main brain, only forwards one of this to the plant.

9.5.2 Main Brain

In figure 29 is the main brain of the control system. The main brain consists of a flow chart with five different stages. Its purpose is to control the gripper, spine and wheels depending on kinect data, but since neither the kinect, spine or gripper has been implemented it's only task is to control the drive of the robot in the simulation.
The purpose of the initial state, number 1), is to initialize and check the system, to see if the gripper, spine and wheel modules are working and responding correctly. As can be seen in the image, functionality of the initialization state

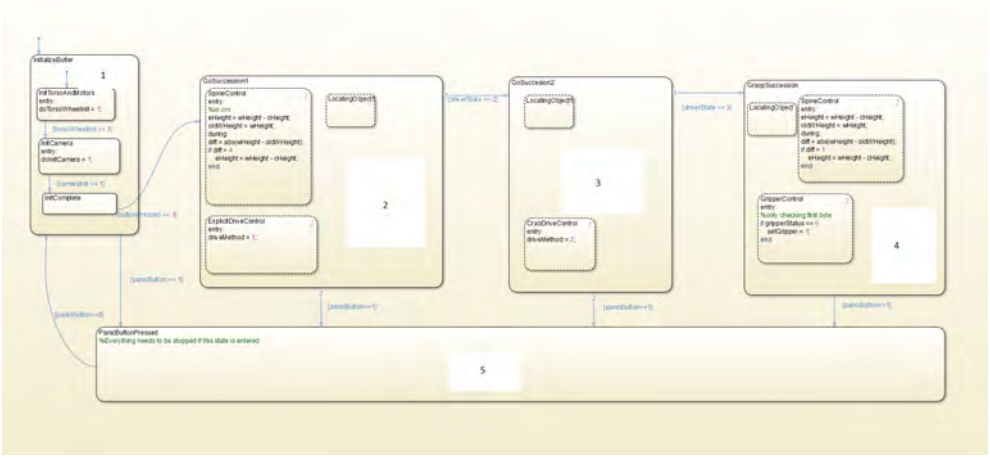


Figure 29: Main brain

has been done but at the moment it is assumed to work so the simulation can be run. Depending on input from the rest of the system, it will proceed with driving to the goal. In stage 'GoSuccession1', number 2), is where the explicit drive is activated and when the explicit drive is done the main brain proceeds to 'GoSuccession2, number 3), where the crab drive is used. After the drivers are both done comes the challange in lifting the object, which is supposed to be adressed in the 'GraspSuccession' stage, number 4). At any point in time, to robot should be able to cancel what it is doing in case of an emergency. Therefore a state, number 5, has been added called 'PanicButtonPressed'. Although it does not do anything at the time there should at least be such a state to go to.

9.5.3 Explicit Drive Controller

Shown in figure 30 is the explicit drive controller. Marked in the figure are three different areas. The actual controller is in the area marked with the red border. The controller uses polar coordinates so the error needs to be converted from cartesian coordinates, this is done in the green area. The output from the controller is a forward velocity and steering angle for a single point, i.e. the center of the robot but what is needed are steering angles and velocities for each wheel, so the output needs to be mapped from the center to the wheels. This is done in the blue area.
Basically, the controller operates by reducing three different errors. Rho, the length of the vector $\hat{x}$ between the start point and the goal. Alpha, the angle between the robots x-axis and $\hat{x}$. And Beta, the angle between $\hat{x}$ and the x-axis of the goal pose. These errors are reduced by the gainblocks: krho, kalpha and kbeta respectively. E.g. set kbeta to zero and you would have a very stable from pose to point controller. This controller is explained in further detail in the book "Autonomous Mobile Robots "by Siegwart R. and Nourbakhsh I. R., how implementation of such a controller can be done is shown in the book:

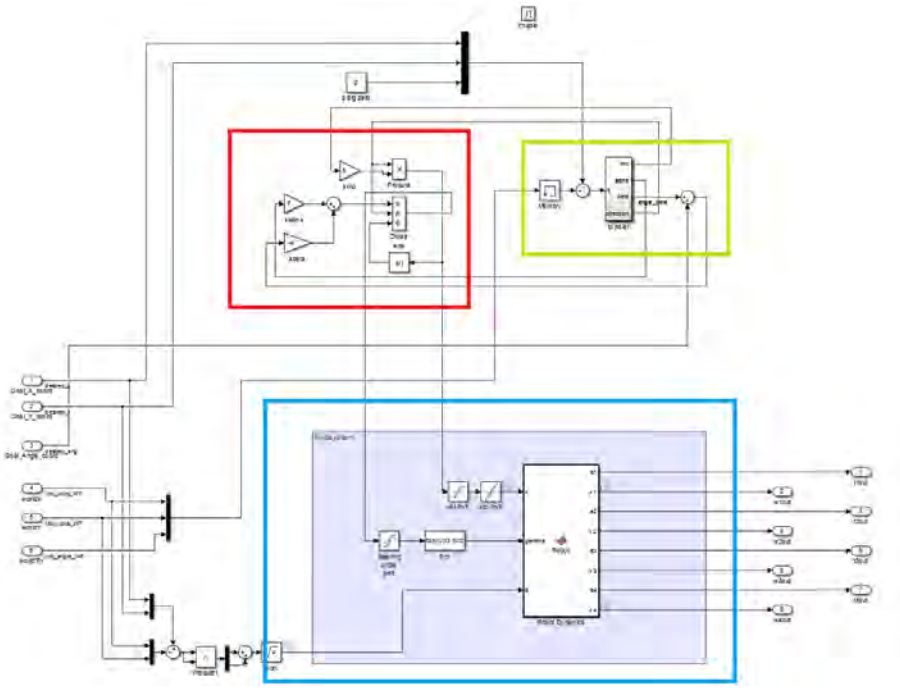


Figure 30: Explicit drive controller

"Robotics, Vision and Control "by Peter Corke.
In the green area, the function "to polar "has another functionality other then just converting from cartesian to polar coordinates. It also chooses a way to describe the error depending on the angle alpha. If alpha is between -pi/2 and pi/2 at time zero, a forward direction is chosen, otherwise a backwards direction is chosen. The direction is maintained and improves the stability of the controller.

9.5.4 Crab Drive

Figure 31 shows the entire system of the crab drive controller. It is very intuitive to use flow chart for the crab drive since it uses a very specific procedure, to first rotate the body of the robot and then to translate the body of the robot. First of, the need to rotate the robot has to be checked, this is done in state 1). If the robot needs to be rotated the wheels has to have the right steering angle, this is checked in state 2) and the wheels are turned in state 3). When the wheels has the correct steering angle, velocity needs to be applied to the wheels for the robot to rotate, this is done in state 4). The velocity is calculated by a function called "controlVelRot "that works as a PI controller. It is somewhat difficult to tune and could use more work to decrease settling time with larger errors (unfortunately auto-tune is not possible). When the robot is correctly rotated it is marked by entering state 5), and then directly goes to state 6). State 6) checks whether the robot needs to be translated or not. If it needs to be translated a

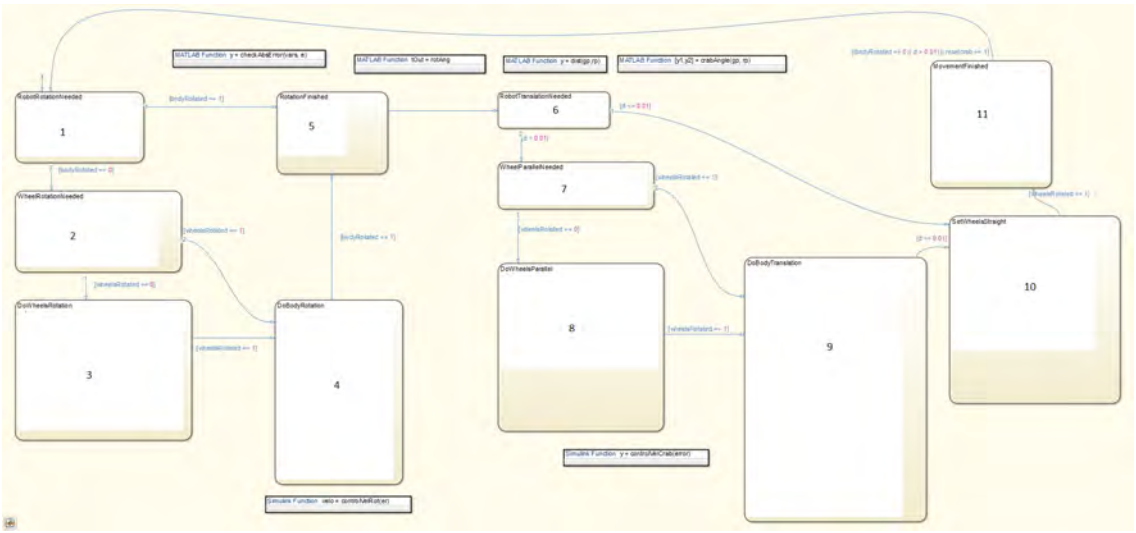


Figure 31: Explicit drive controller

procedure similar to rotating the robot is used (the difference is that the wheels are set to be parallel, not tangent). In state 9) where the robot is translated a new PI controller is used, "controlVelCrab "for setting wheel velocities, it could also be better. Another function to set the wheels parallel to perform the translation is used called "crabAngle ", this function has been heavily experimented with and could also benefit of some improvement.

9.6 How to run the Initialization of Butler and manually send speed and angle commands - Robin Andersson

9.6.1 Description

This section will walk through the Simulink model that will initialize the robot, to later be able to send different speed and angle commands to the four wheel modules. The initialization phase regards configuring the motor controller of each wheel module, and turning each of the wheels in a straight forward direction. The configuring is done by sending the specific configuring messages out on the USB cable which is received by the CAN-translator.

Program requirements: Minimum: MATLAB 2015b

Hardware equipment: USB to Serial converter cable

MATLAB Files that are necessary for this test can be seen in table 6

Figure 32: USB to Serial converter cable

Name	Description (bytes)
initializationAndRemoteControllBlockButler.slx	Contains the model that will initialize the robot by configuring all wheel modules and make it possible to manually send speed and angle commands
parametersButler.m	Contains the necessary parameters for configuring the motor controllers.

Table 6: MATLAB files

9.6.2 Design of the program

Figure 33 shows the model and the different parts of it. The program have been divided into two parts, the first part where the initialization takes place, and the second where the user is given the opportunity of sending various speed and angle commands to the robot by altering the two sliding bars. The robot will hold the current speed and angle until it receives another speed and angle message.

The **violet box** in figure 33 contains the parts that can be seen in table 7
The **green box** in figure 33 contains the parts that can be seen in table 8
The **blue box** named 'Send_Message' will send the message that are forwarded to the block, and will be sent when it gets the trigger signal from either the initialization or the speed and angle command block.
The **grey box** is a debugging tool to see how many messages that are sent out on the COM port in the model. This is determined by counting the number of activation signals that are sent to the 'Send_Message' block.

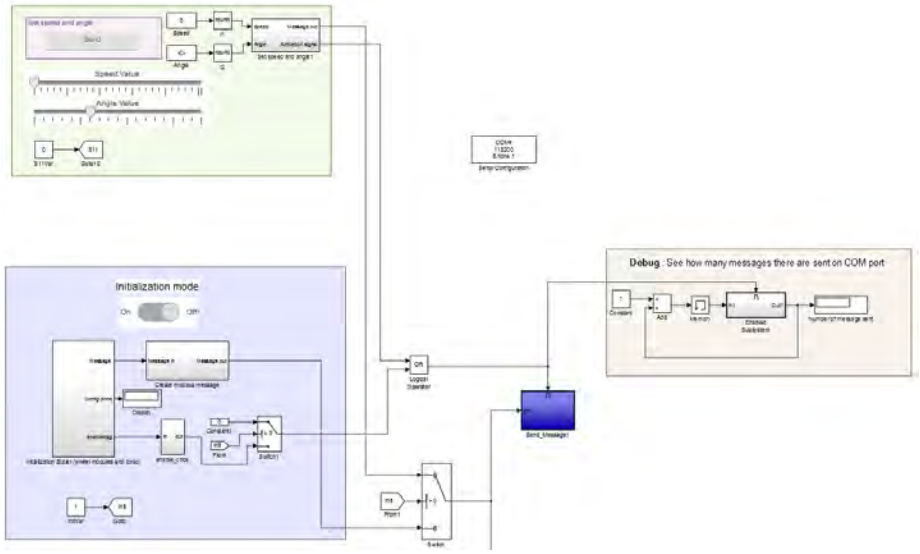


Figure 33: System overview

Name	Description
Initialization Butler	This block will create all necessary messages that are needed for the initialization phase. The output from this block is the following: **Message** –The message that are to be sent on the COM port **Config done** –This will indicate a '1' when all messages have been iterated through **enableMsg** –Activation signal that are generated each time there is a new message to send from the initialization block
enableOnce	This block will ensure that the activation signal is only triggered and enabled once.
Create Modbus message	This will create the appropriate Modbus message according to the implemented protocol.

Table 7: Violet box modules

Name	Description
Set speed and angle	This box takes as an input speed and angle from the slides.
Send button	By pressing this button, the model will send the specified speed and angle to the robot.

Table 8: Green box modules

9.6.3 How to run the program

1. Open the file named paramtersButler.m, and run this script. This will load all parameters into the workspace that are accessible from the Simulink model. In this file, there exist several of configuration messages that are to be sent to the

Encoder	Offset value
FLWMOffset	312
FRWMOffset	423
BLWMOffset	2304
BRWMOffset	757

Table 9: Encoder offset values

motor controllers. There also exist an rotary encoder offset value that is used to set each of the wheels in a straight forward direction. The encoder offset value have been visually determined and can be seen in table 9.

2. Make sure to configure the serial port. Do this by open the 'Serial Configuration' block in the Simulink model. Make sure the settings are used that can be seen in figure 34, and that the COM port corresponds to the connected 'USB to Serial' cable. After this, open the blue block that are named 'Send_Message'. Double click on the 'Serial Send' block and set the COM port that corresponds to the connected USB to Serial cable as can be seen in figure 35.

3. Make sure that the initialization slide is turned on, as can be seen in figure 36

4. It's important to set the wheels of the robot in a safe position before starting the initialization of the robot. Setting them in a wrong position and start the program will cause the cables wired to the motor to brake due to twisted cables. Figure 37 shows in what area the wheel can be put in, marked within the green arrows, and a red arrows showing the forbidden area to set the wheel in before starting the program. The line to compare is the outer line or the direction vector for each of the tires. All drive motors shall point inwards towards the base, and with the wheels facing outwards.

5. After that, press 'Run' in the Simulink model. The model will then begin to compile the model and after that execute it. To see the progress of the number of sent messages, look at the grey box, same as can be seen in figure 38. For the initialization there should be 35 messages sent. When the first initialization is done, press 'Stop' and redo the Initialization one more time in order to make the motor-controller to be properly configured.

6. After the second initialization is finished, switch off the 'Initialization mode' slide. Now it's possible to send different speed and angle commands to the wheels. For this setup, the speed will be sent to all four wheels, and the angle will only be sent to the two front wheels, this configuration enables the bicycle model mode (the two rear servo motors will always be set to static 90 degrees) Specify an angle and a speed in the sliders provided within the box, that can be seen in figure 39, then press the button 'Send' to send the command. It's possible to send angles between 0-180 degrees, and speed between 0-127 (forward), 128-255 (backward). For more detailed description about the messages, see the

'Communication manual: Simulink and CAN nodes' in appendix.

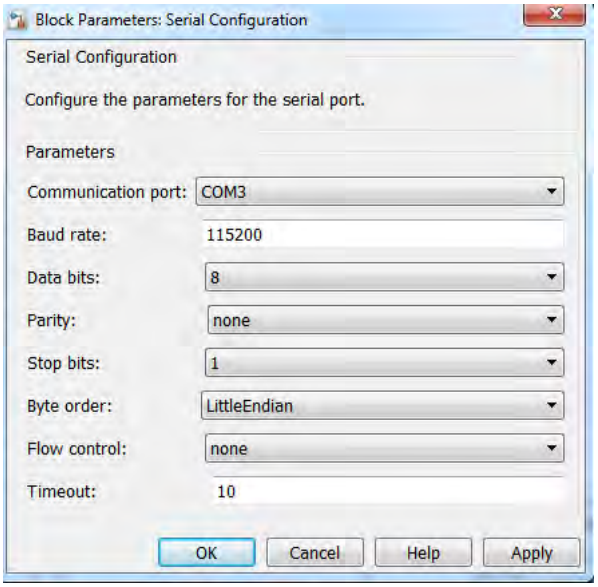


Figure 34: Serial configuration

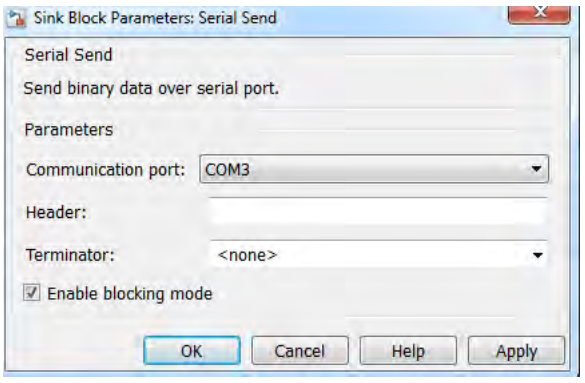


Figure 35: Serial send configuration

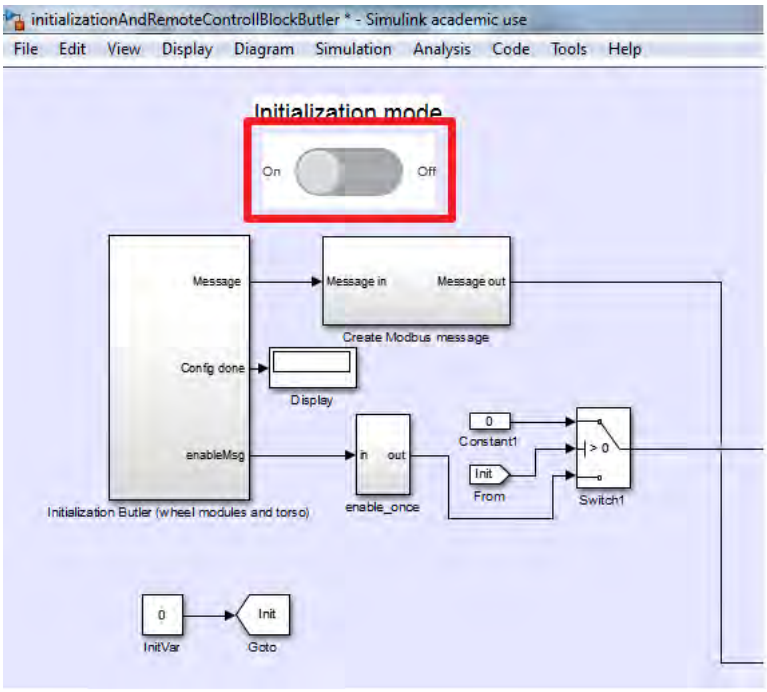


Figure 36: Initialization mode

10 Vision software - Mobin

10.1 Description

For the Butler to be able to succeed in the mission of picking up the targeted object, it has to be able to see where the object is and if there is any obstacle in the path to the object.For this to be achieved the distance is needed and the image to find the objects from. Using stereo vision the butler will be able to both detect and get the distance to the location of the object.

10.2 Requirements and limitations

The requirements are that the butler should be able to detect and locate the object being a coffee cup and give the location in of the object with maximum of 2cm error in all directions for distances 70cm to 3m. It should also be able to detect the object while moving at a pace of 0.5m/s and give an error in the objects location with a maximum of 2cm in each directions. The limitations are that the object will be in front of the butler within 3m. The environment should be clean off distractions and have good lighting. The Kinect v2 are only able to send 30 FPS in RGB image and 100 FPS in IR image. The processing power of the NUC are the bottleneck right now. It makes so rate of received frames and the detection rate has to be lowered. Since the head is not movable the field of view will be limited, the further away the object are the more to the center the

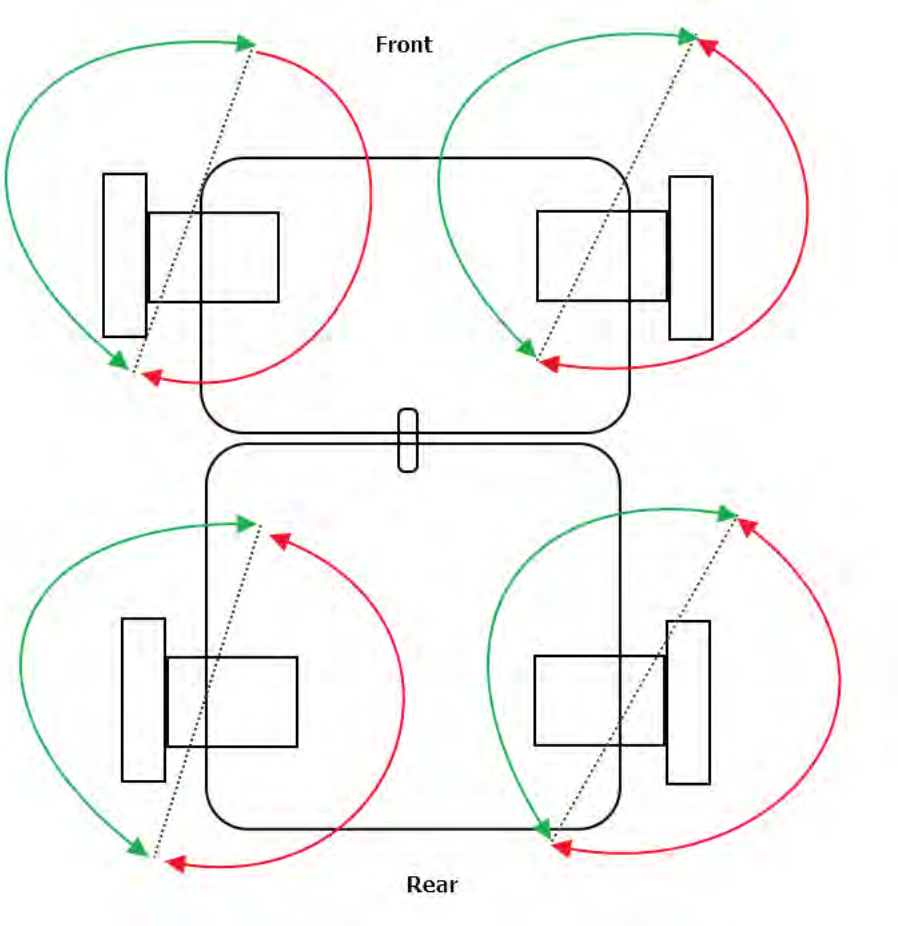


Figure 37: Wheel configuration

camera needs to be .But if the object is close it might be out of the field of view so the camera needs to be tilted down. By not knowing the orientation of the Kinect a unknown error in the location will be had.The Kinect should be set to an angle close to 0°as possible in both X and Y directions on the placement of the butler to reduce the localization error received. The positon of the object is based on Kinect's RGB eye giving it an offset to the arm's positon. The Kinect have a limitation of 70cm while the arm is only 55cm making the distance of the last 15cm unreliable. The detection always find an object even if it is not the wanted object.

10.3 Design and method

The whole algorithm is dependent on Matlab computer vision toolbox. Hence, it is necessary to purchase the licenses for it. If everything in the installation part goes well, you should be able to subscribe a topic from Kinect Bridge. To do

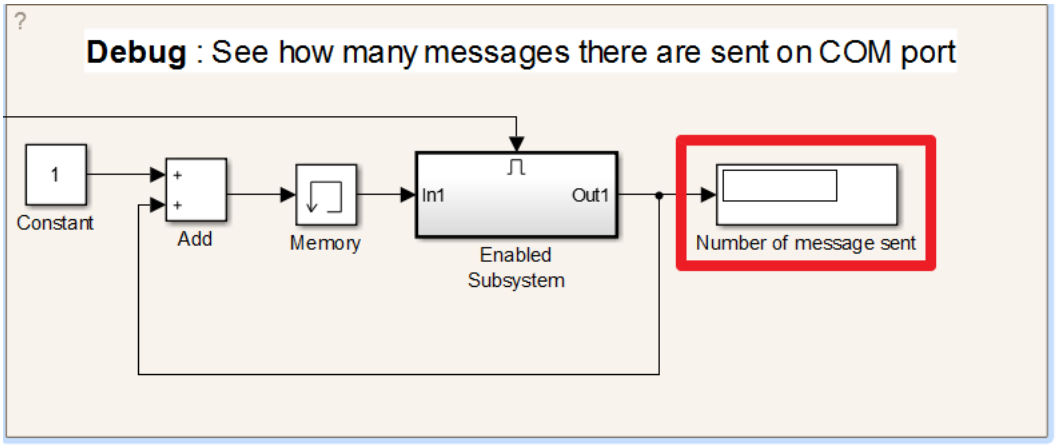


Figure 38: Number of messages sent

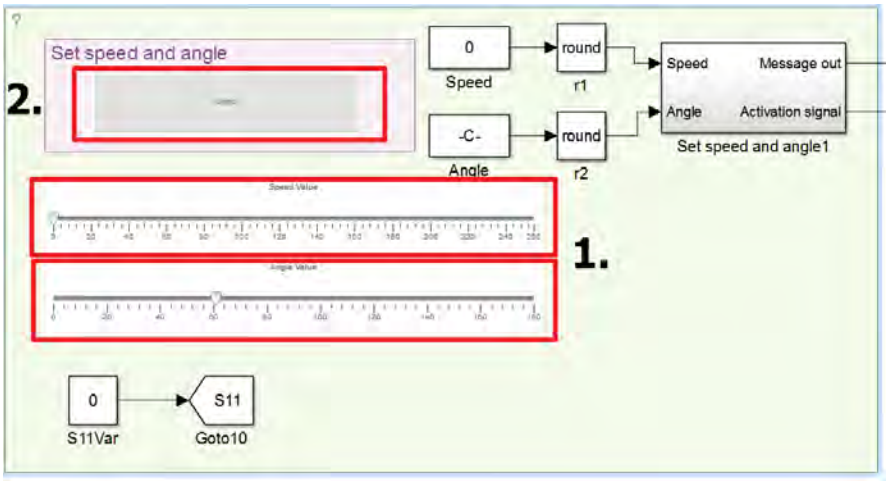
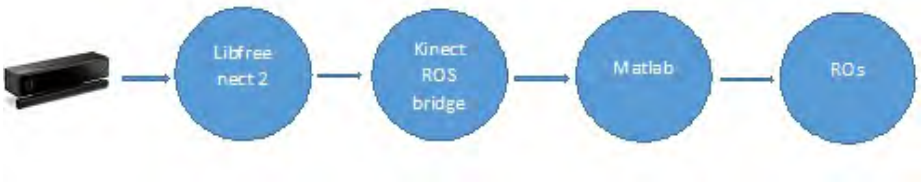


Figure 39: Speed and Angle

this you need these steps to be done: Open a new terminal in Ubuntu and type: roscore This command will start the ROS core. For more information about ROS and its functionality just visit ROS.org. Open another terminal and write this command to run the bridge: Rosrun kinect2_bridge kinect2_bridge_fps_limit:=5 This will run the grabbing from Kinect Bridge at max 5 HZ. Now you can open Matlab and run the code in order to initialize the system.
See figure 40 on page 57.

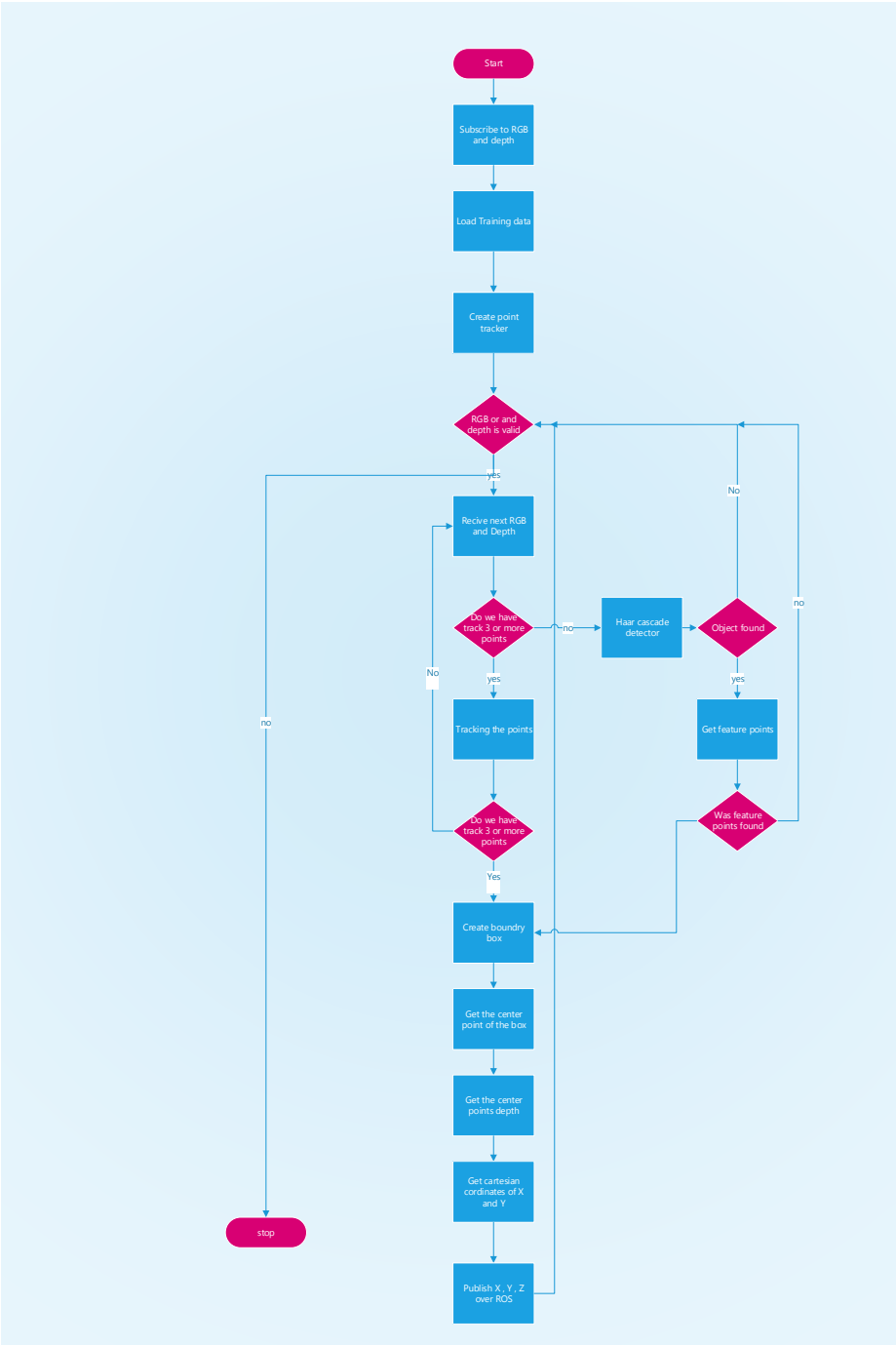


10.4 Calibration

If you are using a new Kinect or if you want to improve the sight of it, you need to do the calibration of the camera first. This part is really important since the correctness of your data is dependent to it. Fallow the orders of this link for calibrating the Kinect version 2 via Kinect bridge tool. Try not to calibrate it with other tools since bridge has its own configuration and model for doing it. If you are really an expert you can calibrate the camera in Matlab considering Depth to RGB mapping problem and also you should know how to manipulate YAML files in order to replace the calibration files of the bridge with Matlab calibration information:
https://github.com/code-iai/iai_kinect2/tree/master/kinect2_calibration#calibrating-the-kinect-one.Be sure you do everything step by step and precise because this is a delicate procedure for bridge system. You should never get re-projetion error more than 0.2 for Ir, depth and sync. Read more about it at:
https://en.wikipedia.org/wiki/Reprojection_error.The calibration information should be something like this:
Error in this picture is around 0.5 which should be lessened to 0.2 and less. The best and simply unachievable is 0. Also check your viewer after each time you calibrate the camera. You can run the viewer by simply running the bridge and after afterwards the viewer itself by this command: rosrun kinect2_viewer kinect2_viewer kinect2 both As you see in the below image, the camera is not calibrated successfully. You can see the mismatches of depth frame represented by black shadow and the RGB frame represented by colors. The importance of calibration advents when you start to extract the Pose of the object. A small error in re-projection error can lead to 20 cm Pose estimation error. You may also find it really difficult to calibrate when it comes to sync. Sync is a heavy process hence try to run it in a powerful system (At least 8 cores) to encounter less repetitive lags or screen freezes during calibration. You can of course run the cloud image of Kinect and use that to assess the calibration with the help of Rviz point cloud viewer or by just running Kinect viewer with this command: rosrun kinect2_viewer kinect2_viewer kinect2 sd cloud.

10.5 Train an object with HAAR Cascade

Here we use a built in software of Matlab vision toolbox named 'trainingImageLabeler' which trains an XML file with the information of the trained object. This simple amazing program it is not only for single object training but also



[h]

Figure 40: Matlab Flowchart

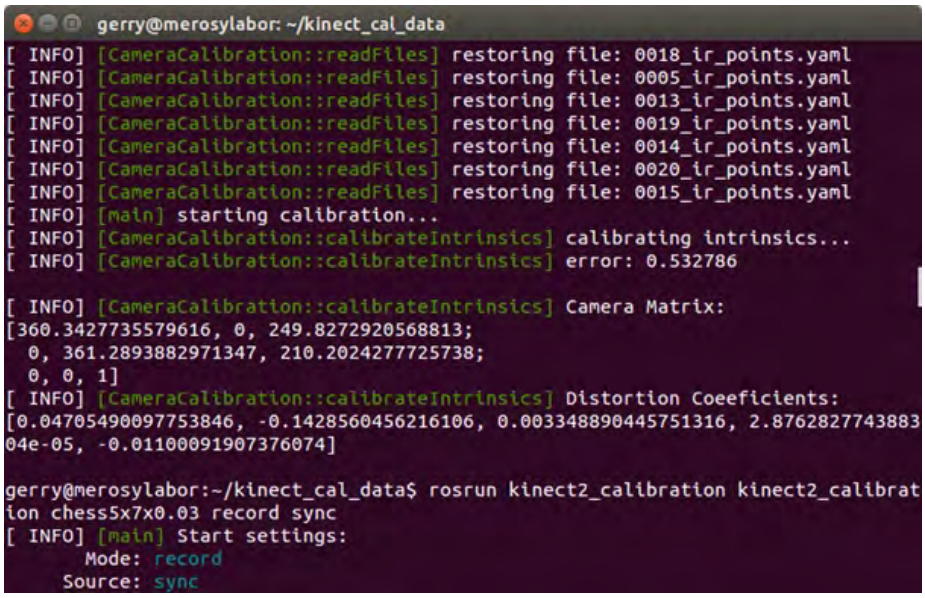


Figure 41: Re-projection error

can be used as multiple different object training. Fallow this link to get a grip of how the system works:
http://se.mathworks.com/help/vision/ref/traincascadeobjectdetector.html In this example Mathworks is using stop sign pictures to train a system. For your own purposes you need to feel up the program with your desired object positive and negative pictures. We used a simple coffee cup for our goal object and we took the pictures with Kinect its own RGB frames. It is totally recommended to do it with Kinect since it would affect the detection because other cameras they have other resolutions and different properties. We used 7 stages to train the system with 0.2 false alarm rate and we got really good results. If you want to train the system with 10 stages and low alarm rate you need both positive and negative pictures. After finishing the training and running it, Matlab will deliver a XML file which can be used in our code. If you have to suddenly change your environment you also have to add the new positive and negative pictures of it to HAAR. Nevertheless you can still use the old training pictures from the old environment. This will make a more robust detector. More pictures is equal to better detection rate.

10.6 ROS topics for Kinect V2

/kinect2/hd/camera_info
/kinect2/hd/image_color
/kinect2/hd/image_color/compressed
/kinect2/hd/image_color_rect
/kinect2/hd/image_color_rect/compressed

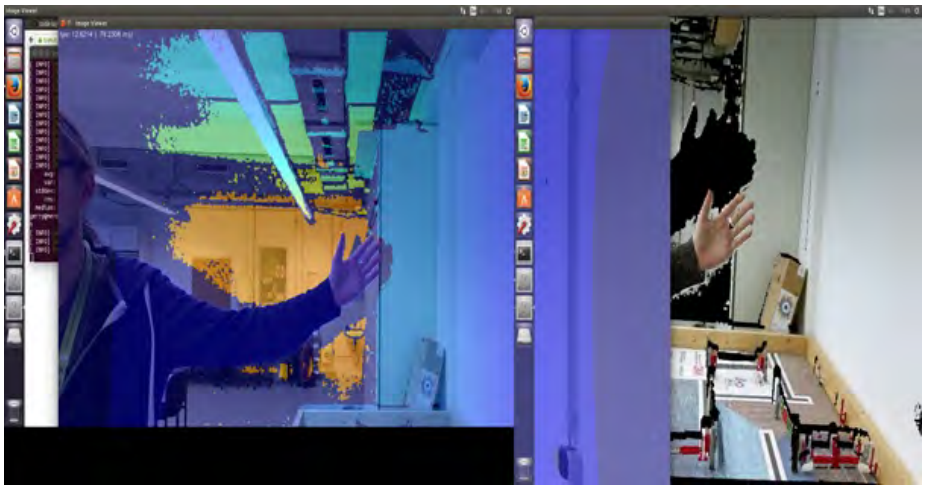


Figure 42: Image Viewer indicating calibration problems

/kinect2/hd/image_depth_rect
/kinect2/hd/image_depth_rect/compressed
/kinect2/hd/image_mono
/kinect2/hd/image_mono/compressed
/kinect2/hd/image_mono_rect
/kinect2/hd/image_mono_rect/compressed
/kinect2/hd/points

It is possible to see ROS topics by typing 'rostopic list' in a terminal. From Kinect topics we chose:
/kinect2/hd/image_color_rect (For RGB)
/kinect2/hd/image_depth_rect (For depth)
Important thing is for our algorithm is not possible to have color image and you need to choose greyscale image which is the mono topic. Also HD is necessary and topic like QHD, although they are faster and less CPU draining, in the same time they have lower detection ratio. Better resolution will always lead to better detection ratio. But about the rect, we should mention that this is the actual synchronized (Mapped) RGB and Depth frame, so they are the needed topics for our solution.

10.7 Cartesian Coordinates

So when we find the coffee cup in RGB frame, we send the center of the mass as a pixel to Depth frame and with two ways. One is Impixel function which is described as, Impixel(I) returns the value of pixels in the specified image I, where I can be a grayscale, binary, or RGB image or in our case Depth image. This function uses the nearest neighbor method to find the represented X, Y coordinates from RGB into Depth map. This represented pixel which in our

case is the center of the mass, has a certain value in Depth map which is the desired distance to the object. This function can accept double variables for X and Y like 672.800, hence, it can give a more accurate result of distance. In the same time this function may not be able to find any near neighbor accordingly to X and Y. Thus, we have used another way to find the exact represented RGB pixel in depth map:
depth = double(CLoud2(X,Y));
Cloud2 here is the depth frame. X and Y
With the estimated distance you can find the real world X, Y coordinates or in

Figure 43: Left picture represents RGB frame and found object which has a certain value in Depth map, the right image. Closer points to camera have brighter pixels.

other name the 3D coordinates. The method used for this approach is called Cartesian Coordinates.
Camera RGB intrinsic paramters

fx_rgbcam = 1075.5161696440566
fy_rgbcam = 1074.400421387934
cx_rgbcam = 990.3886054391227
cy_rgbcam = 530.3584716640612

This snippet shows the RGB intrinsic parameters of the camera which can be find by ROS topic called:
/kinect2/hd/camera_info
We are using RGB parameters in the code since depth is already mapped and synced with RGB in the bridge. It is a well-known formula how to calculate the 3D point with depth and focal length of the camera. And everything is already

explained in the code. One thing you should understand is that this 3D point has true value if the camera is not tilted or rotated in any way. If you want to calculate that accordingly to the tilt or pan of the camera, you need some feedback from servo motors. We tried to calculate it statically but it was not very easy but we achieved some successes points which are commented parts of the Cartesian coordinate section.

10.8 Using the system (Dependencies, Installation, configuration and execution)

Kinect V2 does not have any interface with Matlab, but it is said that MathWorks will support it soon. Until then, the only way to communicate with ROS is the model which is shown here on top of the script. Also we do not recommend you to work with Simulink toolbox since we tested it and was not integrating with ROS Bridge. In future, when Matlab could support Kinect V2, then just grab the frames with it, do your image processing and send out the data to your ROS node with ROS publisher.
For the start it is necessary to mention that this system is supposed to run on Ubuntu 14.3 or higher. That is because ROS support for windows is only experimental and also Libfreenect on windows 7 or overall on windows is buggy. Hence the whole system is only meant to be mounted on Linux OS.
First start with installing ROS on Ubuntu. The recommended version of ROS is indigo. You can choose different versions of ROS but try to analyze your system first, both from hardware and software aspect and then decide which version is compatible with it. The recommended version is ROS Indigo. For finding out the details about Indigo and other versions, visit ros.org.
For ROS installation visit:
http://wiki.ros.org/indigo/Installation/Ubuntu
Test your ROS installation with "rosrun rviz rviz". If Rviz is working, then pretty much the whole ROS system shall work.
Then we have to install libfreenect2:
https://github.com/OpenKinect/libfreenect2
In this Git, you willl find all the description you need for running the Protonect, which is the frame viewer of Libfreenect. This is also a good program to evaluate if the frame grabbing process is flawless and also it will provide the FPS rate of the grabbing. Here you might encounter a problem, which is Protonect windows ending up showing four black windows instead of IR, RGB, Depth Sync frames. This means your system is not supporting OpenCL or OpenGL or both. We assure you that it is not possible to work with the CPU mode grabbing both for RGB and Depth, on the current system (Intel NUC, 2 cores i7 Haswell). The reason is that this mode is quiet slow and CPU draining. (Up to 94 percent). Thus, you need to run OpenCL since OpenGL mode will only work in Depth frame grabbing and not in RGB grabbing. We recommend two fixing methods. First, to manually install Beignet from this link and run all its tests: http://www.freedesktop.org/wiki/Software/Beignet/.
And second to install manually OpenCL from:

http://wiki.tiker.net/OpenCLHowTo
Also try to run the tests which are very important for assessment of the correct installation of OpenCL on your system.
Once you could initialize Protonect on CL mode with all for frames working, then it is time to install Kinect V2 ROS Bridge.
To install KinectROSBridge visit this link:
https://github.com/code-iai/iai_kinect2
You may encounter some or even many problems during the installation mostly because of lack of prerequisites of the program or rarely some bugs which you can contribute to this Git and help to fix them as we did. Thiemo Wildermier is the responsible person who answers your questions in his Git forum, mentioned on the above link. Many of your questions and problems might be already answered or solved via the same forum by other people's threads. Once you finished installation this command should work without any problems: rosrun kinect2_bridge kinect2_bridge _depth_method:= opencl If you decided to limit the FPS of the frame grabbing use this command: rosrun kinect2_bridge kinect2_bridge _depth_method:= opencl _fps_limit:= 'Your limit' Matlab: Matlab is a well-known program which does not require any help for installation. Open Matlab and fallow the code.

10.9 Testing

This test sesion was with a tilted view of the kinect We are 3 meters away in front of the object:

yObj = 0.0032
xObj = -0.2555
zObj = 3.0570

We are 1.8510 meters away in front of the object:

yObj = 0.0218
xObj = -0.1960
zObj = 1.8510

We are 3.0390 meters away in front of the object with 2 cm error in Y:

yObj = 0.3653
xObj = -0.0392
zObj = 3.0390

We are 3.2220 meters away in front of the object with 1 cm error in X :

yObj = 0.0049
xObj = -0.5074
zObj = 3.2220

We are 3.2220 meters away in front of the object with 4 cm error in Z and 2cm error in X:

yObj = -0.0026
xObj = -0.2140
zObj = 2.0850

We are 3.2220 meters away in front of the object with No error:

yObj = -0.0074
xObj = -0.0361
zObj = 2.0570

We are 2.1070 meters away in front of the object with 2 cm in Y and X:

yObj = 0.3641
xObj = -0.1379
zObj = 2.1070

We are 2.2080 meters away in front of the object with 2 cm in X:

yObj = 0.0023
xObj = -0.5305
zObj = 2.2080

We are 1.0190 meters away in front of the object with 2 cm in Y:

yObj = 0.0205
xObj = -0.0444
zObj = 1.0190

10.9.1 Future works

Having a regular stereovision system will remove the extreme minimum distance to the object and become a lot easier to calibrate since both images will have the same resolution. having a pan/tilt to control the head with feedback to know the angle of the camera and having an algorithm to transform the angle and the current values into real world values. having a better computer or moving the system to a separate embedded system to increase the frames received. The system could be able to detect many different objects and learn the objects on the spot.

11 System communication

This section of the report will cover the design and implementation of the software for the CAN controllers that are controlling the functionality of the wheel modules and the torso module, as well as the structure of the whole communication system.

11.1 Description - Marcus Larsson, Jakob Danielsson

The communication system for the Butler robot is based upon a mix between CAN and MODBUS UART communication. There are four Wheel modules, and one Torso module which house one CAN card each (total five). At the top of the communication system there is a NUC which will run a Simulink model that sends command over UART to a sixth CAN card called the translator card. The purpose of the translator card is to receive UART messages from the Simulink model and convert them into CAN messages and send them out to the CAN bus. The translator card will also receive CAN messages and send them to the Simulink model as UART messages. There is a seventh CAN card that is going to be used for the power supply. During this project this has not been worked on and therefore no implementation has been done.

11.2 Requirements and limitations - Marcus Larsson, Jakob Danielsson

The requirement for the communication system is to provide a fast and secure way to transmit and receive messages throughout the system.

All the necessary tools and hardware are listed below.

- MATLAB 2015b and Simulink (for running Simulink models)
- Hercules from HW group [7] (program used for sending serial commands)
- GNAT GPL 2012 AVR [8] (Coding and compiling programs written in ADA)
- Atmel Studio 6.2 and up [9] (Program to download the code into the CAN cards)
- Atmel ICE JTAG [10] (The hardware needed for downloading the code into the CAN cards)
- USB to UART converter [11]
- Power supply for the CAN cards
- Kvaser Leaf Light v2 [12] (Used for monitoring the CAN bus)
- Kvaser CANKing [13] (The software for utilizing the Kvaser Leaf Light v2 hardware)

One limitation for the Butler project when it comes to developing the software has been the ability to debug the code. The code that was written for the CAN cards for a previous project were written in ADA using a program called GNAT GPL 2012 AVR. Since GNAT GPL 2012 AVR does not have a support to download code directly to the CAN cards it has been hard to debug the code at runtime. What has been used for debugging is a CAN bus monitor called Kvaser Leaf Light v2 for reading messages from the CAN bus. In the code a CAN message is sent instead of using for example the *printf* function used in C. Due to this it has been difficult to debug the code written for this project.

11.3 Design and interface - Jakob Danielsson

The communication architecture have been designed to follow a CAN topology network and the main internal communication is run on a CAN bus, which transfers command messages to the correct module. Currently, there are five different modules which controls the motors on Butler; front left wheel module, front right wheel module, rear left wheel module, rear right wheel module and torso/gripper module. All these modules are receiving control command messages that are sent from the Simulink program. However, since the Simulink model does not understand CAN messages, another card has also been added to the communication structure, which is the translator card. This card communicates via UART to the NUC and then translates the Modbus messages into CAN messages. The wheel modules also communicate with a motor controller through full duplex SPI communication in order to set configuration properties

of each motor. The motor commands are sent are sent as a PWM signal to the motor controller. Lastly, the wheel modules can also read the values of the rotary encoders by using software implemented SSI communication. In Figure 44, the entire communication architecture can be seen.

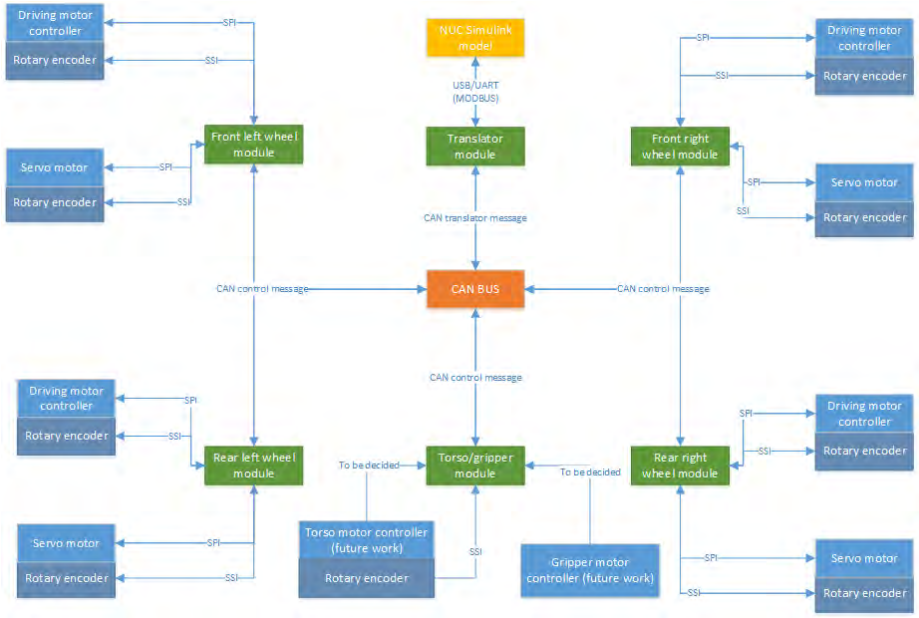


Figure 44: The communication architecture.

Table 10: Frame format for Modbus ASCII protocol.

Name	Length (bytes)	Function
Start	1	Starts with a colon sign ":" to indicate the start of the message
Address	2	The address of the specific node on the CAN-bus
Function	2	The function code specifies what the purpose of the message is
Data	4	The data field
CRC	2	Cyclic redundancy check to see that the message is correct received/transmitted
End	1	Carriage return to indicate the end of the message

11.4 Message structure - Robin Andersson

For a message to be sent out to a specific node in the network, two different stages are involved in the process. First of all, creating and sending the message

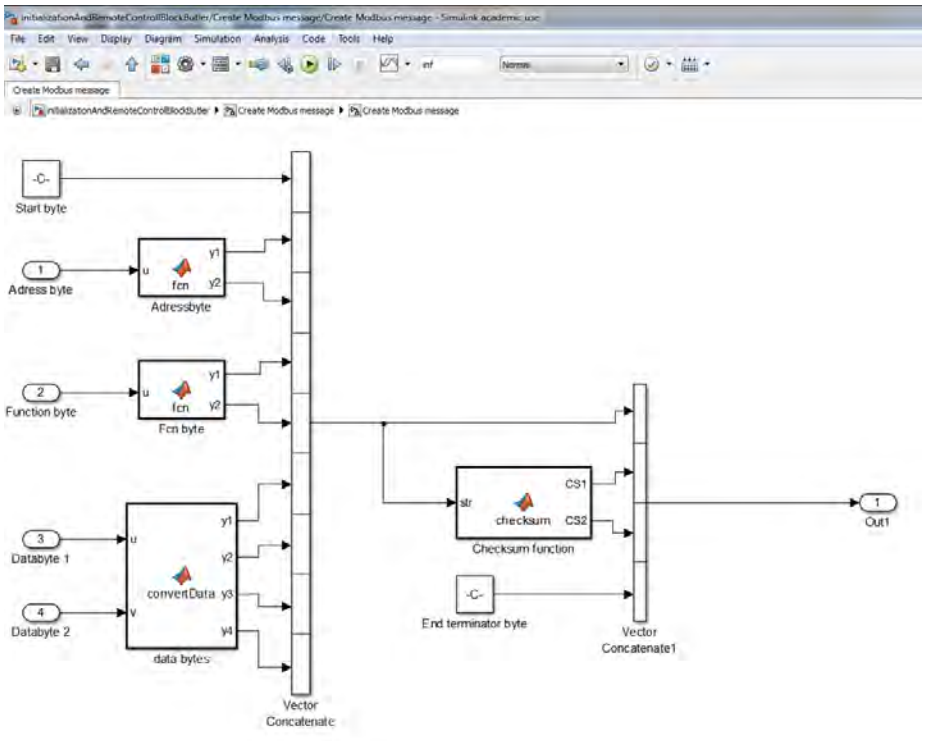


Figure 45: Modbus message creation.

according to the custom made protocol based on "Modbus ASCII protocol". The second stage is to decode this message, which happens at the CAN-translator card, and then convert this decoded message into a appropriate CAN message to last of all be sent out on the CAN and be received by a node.

11.4.1 Modbus ASCII protocol

The message structure that is used when sending commands and requests are based on the Modbus ASCII protocol, that uses the ASCII values of the hexadecimal addresses. The frame format that are used can be seen in Table 10.

11.4.2 Modbus ASCII block in Simulink

The Simulink block that applies the protocol standard and creates the messages are named 'Create Modbus message'. This can be seen in Figure 45. The different parts have been implemented accordingly to the protocol. Inside each of the blocks 'Adressbyte', 'Fcn byte','data bytes' and 'checksum function' there is a script that will convert the different part of the message into the appropriate format.

11.4.3 Transmission time Modbus message

Serial transmission time from Simulink to the CAN-translator card can be calculated according to:

$$\frac{(totalBytesInMessage) * 10bits/byte}{CommunicationBaudrate(bits/s)} = SerialTransmissionTime \quad (1)$$

With the communication setup known:

- Baudrate : 115200 bits/second
- Total bytes/message: 12

The serial transmission time for one message is then calculated to:

$$\frac{12 * 10bits/byte}{115200bits/s} = 1.041ms \quad (2)$$

The transmission time from the CAN-translator card to one CAN-node can be calculated in the same manner, but with the different setup that follows:

- Baudrate on CAN-bus: : 250000 bits/second
- Total bytes/message: 8
- Data bandwidth best case: 144.14 kpbs [5]

The transmission time for one message is then calculated to:

$$\frac{111}{144140bits/s} = 0,770ms \quad (3)$$

So for sending one message from Simulink down to a CAN-node, the theoretical time would be 1,811 ms without taking translation of messages from USART into account. In our case, there are four CAN nodes that are going to receive message continuously, more specific the wheel module nodes. The theoretical time to send four messages to all of the wheel module nodes is calculated to be :

$$1,811 * 4 = 7,244ms \quad (4)$$

The time for the response messages up to Simulink to arrive would take the same time. Total time for sending four messages down to each of the CAN nodes and then receive the response messages in Simulink would be 14,488 ms.

Total messages per second given that there is 28 CAN-nodes to reach:

$$\frac{1}{0,001811 * 28} = 19.72 \quad (5)$$

11.5 Method - Marcus Larsson

This section will cover how the software is implemented on the CAN cards, how it works and the structure of the code. The software for the CAN cards are divided up into modules, where there are four wheel modules, one torso module, and one translator card. Each of these modules serve their purpose for how the robot move and interpret commands messages received from the Simulink model. In the following sections, each module will be explained more.

11.5.1 Wheel module

For all the wheel modules, there are four separate ADA-projects for each one of the wheels. All projects contains the same file structure, which can be seen in Table 11. Each file will be explained in this section.

Table 11: The file structure for the wheel module.

module_params	Contains all module specific parameters.
main	The main file with the main loop.
wheel_module	This file contains all wheel specific functions.
step_motor	All functions related to controlling the motors are in this file.
torque_interrupt	Here are the code for managing interrupts and their routines.
create_msg	This contains one function that sends out one message to the CAN bus.

In a perfect world only one ADA-project should be needed since the code is close to identical. But since some parameters differs among the projects, this is not possible. To keep the deployment to the CAN cards as seamlessly as possible only one file need to be changed in the wheel modules. This file is called *module_params* and contains the global parameters which that specific wheel module has. A few example of the parameters are the module number, error codes, CAN IDs, and function codes that are only for the specified module. The rest of the code files are identical for every wheel module.

The main file consist of a main loop and its structure can be seen in Figure 46. Before the main loop a hardware initialization is run to set up all needed components. Once the program has entered the main loop it will for every iteration check for a CAN message. If a CAN message has been received, the procedure *runCommand* will start and then execute the appropriate function according to the received function code. There is a timer in the main file which counts up to 10ms. Once every 10ms an if-statement will execute. In this if-statement two rotary encoder values will be updated as well as the PWM frequency for the step motors, but only if there are any ongoing speed or angle changes.

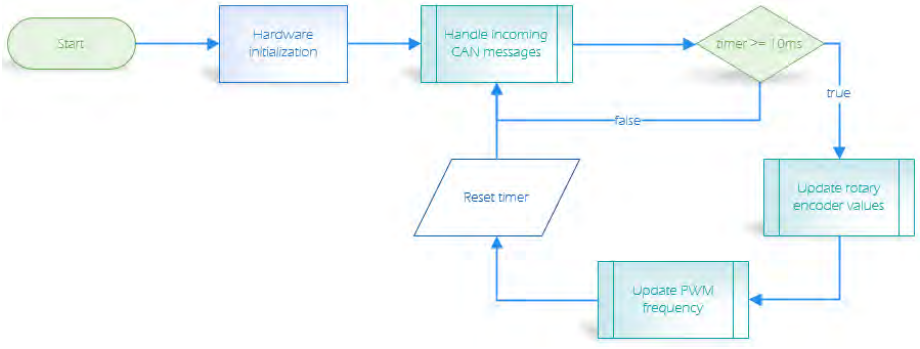


Figure 46: Main loop for the wheel module.

Next is the *wheel_module* file. This contains all functions and procedures related to the wheel module. On each wheel module there are two rotary

encoder chips. The purpose for the encoders are to achieve the speed of the wheel from the first one, and the angle position for the servo motor from the second rotary encoder. These two types of data is collected every 10ms, and the two procedures *readWheelDriveEncoder* and *readWheelAngleEncoder* are doing the job. The encoder value is a 12 bit integer, therefore the number range from 0 to 4095. During every iteration the procedure *readWheelDriveEncoder* capture the current encoder value and calculates the difference from the previous captured encoder value. This value gets stored as a sample value in an array. With several values in the array a speed can be calculated by using the following formula:

$$\frac{\sum samples[]}{encoderResolution} * \frac{updateFreq}{samples} * 2\pi r \tag{6}$$

where $\sum samples[]$ sum up all the sample values in the array, and then the total sum is divided by the encoder resolution. The *updateFreq* is a value which represent how often the encoder values get updated. Since the encoder values get updated every 10ms the frequency is 100Hz. The value *samples* is the size of the samples array i.e. how many samples that are collected during the frequency time. The last part ($2\pi r$) is the mathematical formula for the circumference of a circle, in this case the wheel.

Every procedures related to the two step motors are present in the *step_motor* file. There is one read procedure that read an address register from a specified motor controller and returns its data. There is also a write function that writes data to the specified address register. To communicate with the motor controllers, the protocol SPI is used. Other procedures are for controlling the speed for the step motors. Procedure *setMotors* apply the changes that comes from the Simulink model. Both speed and angle can be changed at the same time. If there is any change in speed the variable *enableMotorChangeDrive* will be set to true and if there is a new angle coming for the servo motor the variable *enableMotorChangeTurn* will be set to true. If any of these two variables are set to true, procedures *changeDriveMotor* and/or *changeTurnMotor* will change the PWM frequency accordingly until the speed or angle target is met.

The file *torque_interrupt* contains the interrupt routines which will trigger if any of the step motors consume more current than expected. When an interrupt is triggered there will be a torque error (see Figure 47). If one motor controller generates a torque stall both motors will be stopped and a torque error message will be sent to the translator card. The translator card will then in turn distribute stop messages to the other three wheel modules so that the Butler robot comes to a stop. Finally, the torque error will be forwarded to the Simulink model so that the model will know what is happening.

Figure 47: The process if a torque stall occurs.

In the final file, *create_msg*, only one procedure exists. This procedure simply receive an address, a function code and two data bytes that will be packed into

a CAN message and then get sent up to the translator card.

11.5.2 Torso module

The torso module contains almost the same files as for a wheel modules. The only difference is that the *wheel_module* file has been replaced with a *torso_module* file. However, this module is not as developed as the Wheel module. The reason for this is that the time and priority went into finishing the Wheel module. Most of the procedures and functions are in place, but what is missing are the functions for raising and lowering the torso and functions for opening and closing the gripper. These should be located in the *step_motor* file.

In the *torso_module* file there is a procedure which reads the rotary encoder data. This data is used to calculate the torso's height traveled after initialization. Depending on the amount of motor spins a height can be calculated. Right now, one motor spin equals two centimeters, but a more thorough measurement might be needed.

When starting the robot, the software cannot know the whereabouts of the torso and the gripper. Questions like at what height the torso is positioned and if the gripper is opened or closed is something that the software cannot answer. The solution for this is to reset both torso and gripper at startup. By doing this, a initialization command is being sent from the Simulink model which contains a height value which is where the torso should be positioned at when the initialization phase is completed. How the initialization process looks like can be seen in the flowchart in Figure 48. When the initialization loop starts the torso will get lowered and the gripper will starts to open to its maximum. In the loop there are two main conditions that needs to be fulfilled.

1. The gripper needs to be fully opened.
2. The torso needs to be at its desired height according the the height value included in the initialization command.

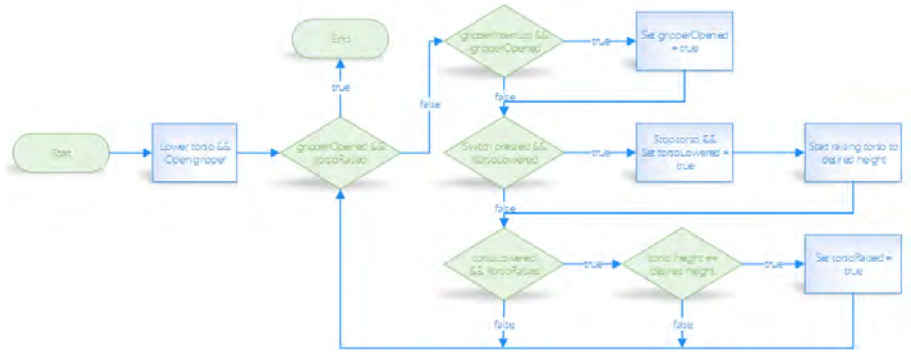


Figure 48: This shows a flowchart over how the initialization process looks like for the Torso module.

Once inside the loop there are three other conditions.

1. The first one will yield true when the gripper is fully opened. To know when the gripper is actually opened, the motor controller should have generated a torque stall which in turn trigger an interrupt. During this interrupt a global variable is set to *true* which indicate that the gripper has stalled. Since the command to open the gripper has been sent it is assumed that the gripper has been fully opened.
2. The second condition will yield true once the torso has hit the bottom. At the bottom a switch has been placed there to notify the software when the torso as reached all the way down. When this has happened the torso will start to ascend to the desired height.
3. Once the torso has started to ascend the third condition will yield true. Then it is time to check, using the rotary encoder, to see when the torso has reached its desired height. Once the desired height has been reached the torso will stop.

When both of the main conditions have yielded true then the initialization process is finished. An initiation ACK will be sent to the Simulink model saying that the initialization phase is completed and that the Torso module can receive new commands.

11.5.3 Translator card

The purpose of the translator card is to receive messages from the Simulink model via UART and forward the message as a CAN message to the CAN nodes on the CAN bus. The file structure for the translator card can be depicted in Table 12.

Table 12: The file structure for the translator card.

main	The main file containing of a main loop and its main procedures.
modbus	This file contains all specific functions related to modbus.
usart_io	All functions related to reading and writing from the UART ports.
create_msg	This contains two functions for sending UART and CAN messages.

The main file hold most of the important message handling functions and the interpretations from UART to CAN and vice versa. In the main loop there is two procedures called *Handle_ModBus_UART* and *Handle_Can_Messages*. The first one, *Handle_ModBus_UART*, receives a message via UART and parse it to see if the received message is correct. If the parse is successful, the message will be sent to the CAN bus. Procedure *Handle_Can_Messages* take care of the CAN messages sent from the other CAN nodes. For the most part, a received CAN message will be resent as a UART message back up to the Simulink model. In special cases, for example when a torque error has been received from one of the wheel modules the translator card will also distribute three stop messages to the other wheel modules. Also, when a stop message has been received from the Simulink model the translator card will send four stop messages and then wait for an acknowledge message from all of the wheel modules before sending a final acknowledge message back up to the Simulink model, indicating that all wheel modules have been stopped.

In the *modbus* file the procedures for all modbus related functions exists. The biggest procedure is the *ParseModbusMessage* which parse the incoming UART message to control if the content is correct.

usart_io consists of a read and a write procedure. The read procedure check if there is any message in a buffer to be received. If so, that message will be returned. For the write procedure, it simply sends the given message out via UART. In case there is any error in the UART buffer, the *DiscardUsart* procedure will read the buffer until it is empty.

Lastly, the *create_msg* file contains procedures for sending both CAN and UART messages. The procedure for sending CAN message looks the same as the one written for the Wheel module. For sending UART message there are two of them where the only difference lies in the arguments. There is also a procedure which converts integers to hexadecimals. This is useful when sending UART messages since the Simulink model expects hexadecimals.

11.6 Test and simulation results - Marcus Larsson

All tests and simulations will be presented in this section. For the Wheel module the tests consists of sending commands and observing the outcome. The tests for the translator card consists of transmitting and receiving times. At the last section, some graphs will be presented.

11.6.1 Wheel module

The purpose of the tests is to verify that the functionality works as intended. The tests consists of sending in commands from *Hercules* (an application from HW group) to see if the expected response is received. A list of different test cases will be listed below together with their results.

1. **Test:** Send in a motor command to set torque to the drive motor and the servo motor, and then read the torque values to check if the values has been written. **Result:** Writing the command seems to fail sometimes at the first try. The problem might be the SPI implementation. However, sending in the torque command for the second time seems always to work.
2. **Test:** Send in a motor command to enable drive motor and servo motor and then send in a disable command. **Result:** Both motors starts up as intended. You can hear a buzzing sound which indicate that there is current running through the motors. After disabling the motors the buzzing sound disappear and the motors are turned off.
3. **Test:** Turn the wheel by hand so it is pointed straight forward and send in a command to receive the rotary encoder value. **Result:** The rotary encoder value got returned and can be used to set the rotary encoder offset during the initialization.
4. **Test:** Set the wheel turn offset and see if the wheels are turning to a 90 degree position (straight forward). **Result:** When sending in the wheel turn offset, the wheel positions itself to the correct degree position. However, sometimes the wheel will turn the other way around which can be very

problematic since the wires connected to the motors might not reached a 360 degree turn.

5. **Test:** Set the forward speed to 10cm/s by sending in a speed command. Afterwards, send in the same speed but backwards. **Result:** The wheel spins up to 10cm/s and then slows down to 0cm/s before speeding up to 10cm/s again but in the other direction.
6. **Test:** Set the speed to 64cm/s and send a request command to receive rotary encoder data. **Result:** Setting the speed and receiving the data works without any problem. However, the data received is very unstable.
7. **Test:** Change the angle position of the wheel from 90 degrees, then to 0 degrees, and finally to 180 degrees. **Result:** The change between the three different angle positions works without any problems.
8. **Test:** Move the wheel (by hand) to an angle outside the boundary and see if an angle command works. If not, move the wheel within the correct range and send the same command again. **Result:** After sending an angle command nothing happens. The command works once the wheel has been moved within the correct range.

From the tests performed there are some issues left that needs to be solved. Sending in a command to a motor controller works for the most time but sometimes the commands does not reach. The guess is that there is some issue with the SPI communication. Another problem occurs when sending in the wheel offset command. If the wheel is at the wrong position the turn movement might take the longer distance instead of the closest. The reason is that the servo motor only moves within the 12 bit encoder resolution range (0-4095). The wheel will not go above 4095 or below 0. Figure 37 show how the wheels must be positioned before sending in the offset commands.

Receiving rotary encoder values after a request command works, but the values can flutter quite a lot. The main reason seems to be the position of the rotary encoder is not right above the magnet placed on the motor. The solution to fix this can be to implement a filter, or try to position the rotary encoder better.

11.6.2 Translator card

The tests for the translator card consist of checking the response time for sending messages from the Simulink model down to the translator card. The method for receiving the response time is using an oscilloscope and then measure the time between two signals. In Figure 49 the blue signal is the RX on the translator card while the yellow signal is the TX. The messages are sent from a test model created in Simulink and the purpose of this test is to see the total response time of sending a message to the translator card, which in turn sends a message to another CAN node on the bus. The CAN node sends a message back to the translator card. Finally, the message is sent back to the test Simulink model from the translator card.

According to Figure 49 the response time is around 3-4ms. All messages are sent from the test Simulink model and the messages are being sent continuously.

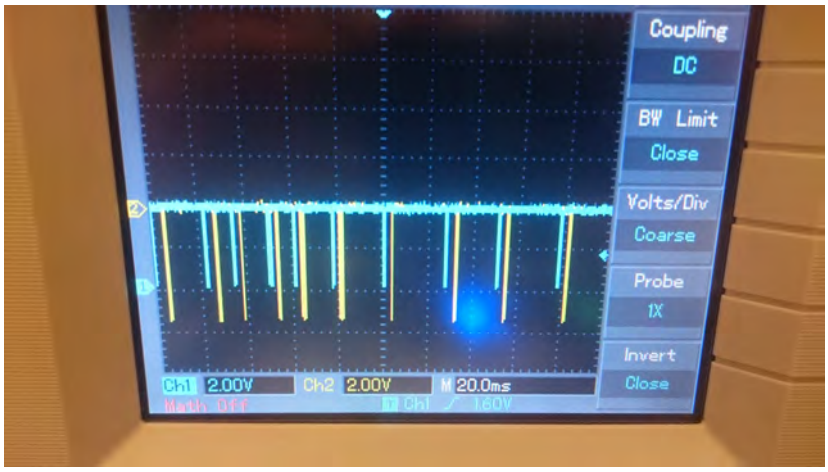


Figure 49: Response time between sending and transmitting messages.

Sometimes there is a bigger gap between two RX signals which the reason seems to be that serial block in Simulink have a hard time sending messages at a fixed interval when it comes to higher speed rate. The problem can also be seen in Figure 50.

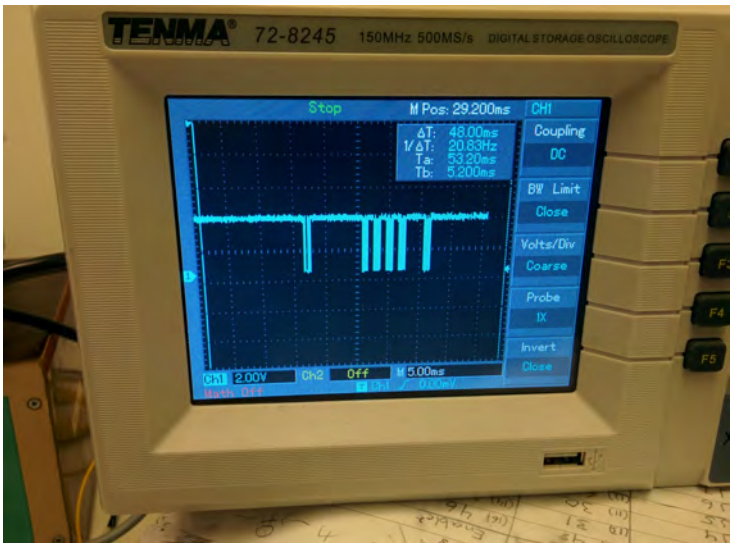


Figure 50: The spikes represent the messages received from the Simulink model by the translator card.

The messages being sent from the Simulink model are represented with blue spikes. The Simulink model sends the messages four at a time but according to Figure 50 it is not always the case. Sometimes there is a random spike that should not be there.

11.6.3 Graphs

Below there are figures showing graphs of the acceleration curves for the drive motor and the servo motor. In Figure 51 one can see the speed changes made from 0cm/s up to 60cm/s, then down to 20cm/s before going up to 60cm/s again. This is done in small steps by increasing or decreasing the speed by 20cm/s.

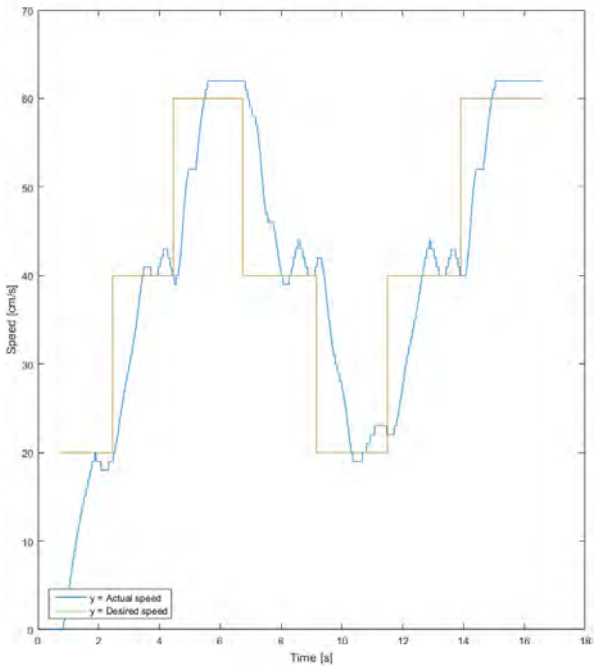


Figure 51: Speed graph over when a wheel speeds up and down.

From Figure 51 the desired speed is fairly close the the actual speed that is calculated by the rotary encoder. The exactness is off by 3-4cm/s and some spikes from the sensor reading can appear. This is due to the positioning of the rotary encoder alongside the wheel for not being in a center position above the magnet. When forcing the rotary encoder to become more in a center position above the magnet, then the speed data is smoother and the exactness is better.

Next graph that can be viewed in Figure 52 is showing how fast the wheel can turn from 0 degrees up to the maximum 180 degrees. The time is around 1.5 seconds for going from 0 to 180 degrees and then another 1.5 seconds when going back down to 0 degrees.

In Figure 53 the wheel turn time can be seen while there is a payload of 70kg being on top of the Butler. The time it takes for turning the wheel from 0 degrees up to 180 degrees remains unchanged from the previous graph.

From Figure 52 and 53 the conclusion is that the wheel turning works really well and that the servo motor can handle the extra payload on the Butler without any problem.

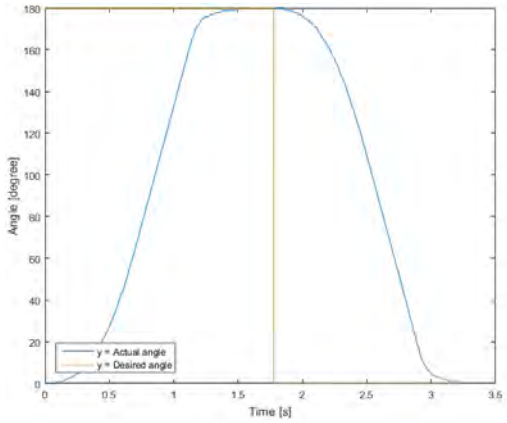


Figure 52: Wheel turning from 0 to 180, and back to 0 degrees.

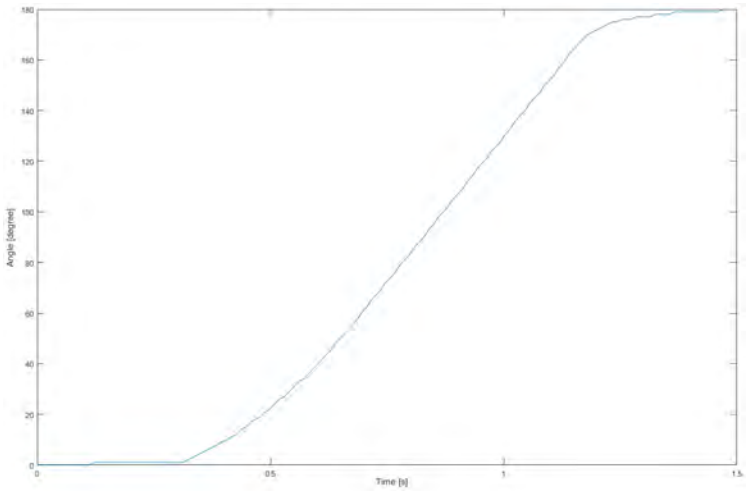


Figure 53: Turning the wheel when there is a 70kg payload on the Butler.

11.7 Using the system - Marcus Larsson

In this section there will be instructions of how to use the system and how to compile and download code into the CAN cards. Below will follow a couple of step-by-step explanation about different tasks that can be performed to be able to run the Butler system.

11.7.1 Needed software

This section will cover the needed applications that are necessary for compiling and downloading code into a CAN card. Some other tools will also be explained in the list below.

1. **GNAT GPL 2012 AVR** - This application is used for writing and compiling all the code in ADA. Check the site at [8] and choose *GNAT GPL 2012* in

the right list and then *AVR microcontroller ELF formal (hosted on Windows)* in the left list, then download and install the *.exe* application under GNAT Ada GPL 2012.

2. **Hercules SETUP utility** - This useful program is used for connecting to a serial port and write commands to the translator card via a USB to UART converter. Download the file located at [7], no installation is needed.
3. **Atmel Studio** - Atmel Studio is currently only used for downloading the compiled code into the CAN cards. To download Atmel studio go to [9].
4. **Kvaser CANKing** - This program is useful for monitoring the CAN bus for CAN messages. Go to [13] to and look for Kvaser CANKing to download the program.

11.7.2 The project structure

In this section the project structure will be explained. The main folder is called *CAN technology*. In this folder there are three folders.

1. **firmware** - All drivers are stored here in a folder called *avr*, for example PWM, SPI, etc.
2. **lib** - Here are the libraries, for example math specific functions. Not used that much in this project.
3. **software** - In this folder lies all the main projects.

The important folder is the *software* folder. Its contents will be listed below.

1. **torso_module** - Control the functionality of torso and gripper.
2. **translator_card** - Convert UART to CAN and CAN to UART.
3. **wheel_module_FL** - Control front left wheel.
4. **wheel_module_FR** - Control front right wheel.
5. **wheel_module_RL** - Control rear left wheel.
6. **wheel_module_RR** - Control rear right wheel.

Inside any of the project listed above the structure is about the same. There are three folders and one project file inside each of the projects. Below is an example how the folder structure looks inside the *torso_module* project.

1. **build** - All the compiled files are stored in this folder.
2. **src** - The source files are stored here.
3. **target** - After the compilation is completed, the target file will be created in this folder.
4. **torso_module** - To start up a project in GNAT GPL 2012 AVR, this file is the one to open. This file also contains the settings for the project.

11.7.3 How to open a project and compile the code in GNAT GPL 2012 AVR

How to open and compile a project is shown in Figure 54 and described in the list below.

1. Open a project file. A project file has the extension *.gpr*.
2. Once opened, the source files in folder *src* will be listed in the window to the left. Any of the files listed can be opened by double clicking.
3. After any modifications in a source file you press the compile button from the tool bar at the top.
4. When the compilation is finished and no errors occurred a *main* target file will be created in the *target* folder. This file is later downloaded into a CAN card using *Atmel Studio*.

If you receive any warnings when compiling, they can for the most time be ignored. Press the *Build All* button again and they should disappear.

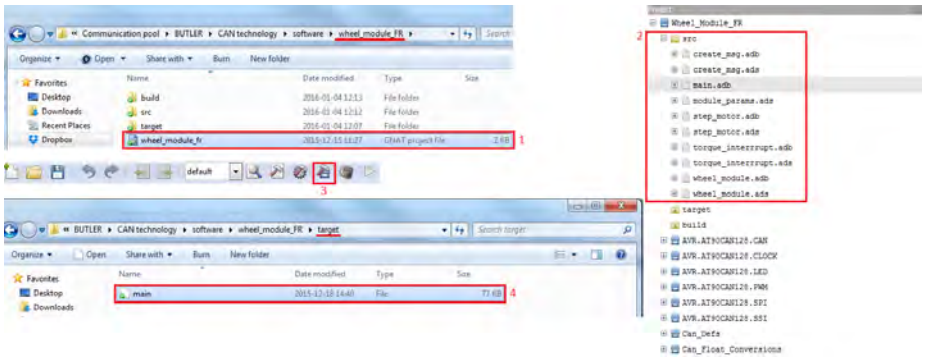


Figure 54: How to open a project and compile it.

11.7.4 Setting up the hardware environment

Before downloading the code into the CAN cards the environment first needs to be set up. All the necessary components are shown in Figure 55. What each number represent will be described in the list below.

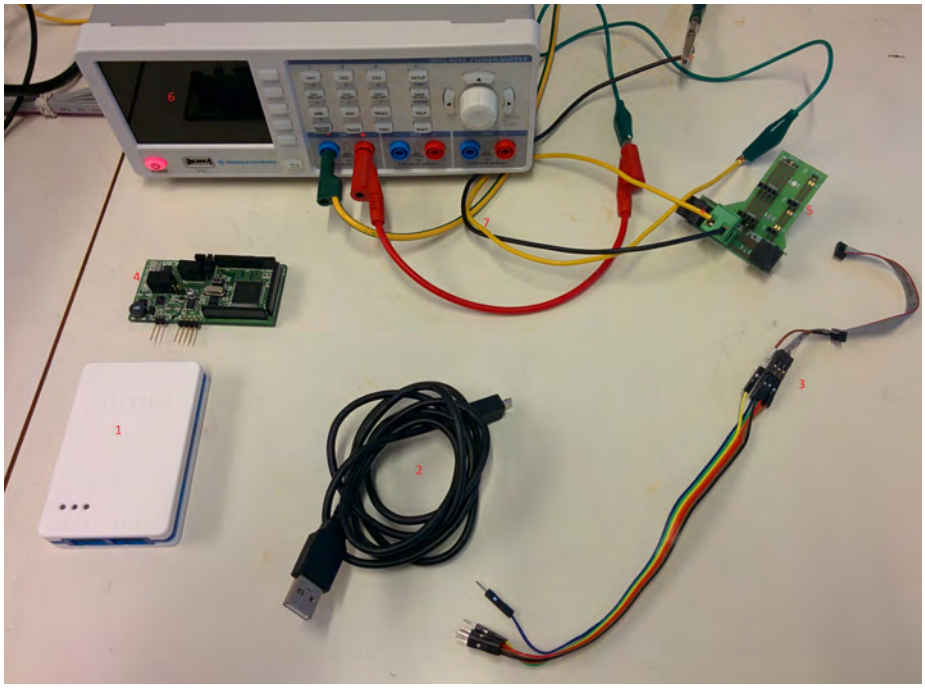


Figure 55: A picture showing all the necessary hardware needed.

1. The Atmel ICE debugger. Used for downloading the code between the computer and the CAN cards.
2. A USB cable that is connected between the computer and the Atmel ICE debugger.
3. JTAG cable. This one is connected to the right port of the Atmel ICE debugger labeled *AVR* and the other end is connected to the CAN card.
4. The CAN card which the code will be downloaded into.
5. A stack which was used to powering up the CAN card during the development.
6. A power supply is needed to give power to the stack, which in turn give power to the CAN card.
7. Some cables are needed to connect the power supply with the stack. On the stack the yellow cable is power and the black cable is ground. On the power supply the red cable is power and the green one ground.

The first step is to insert the CAN card into the stack, like shown in Figure 56. There are one four pin header and one two pin header connected to the stack

from the CAN card. The right one of the two pin header is the power pin and the one to the left is the ground pin. The four pin header is used for the CAN communication.

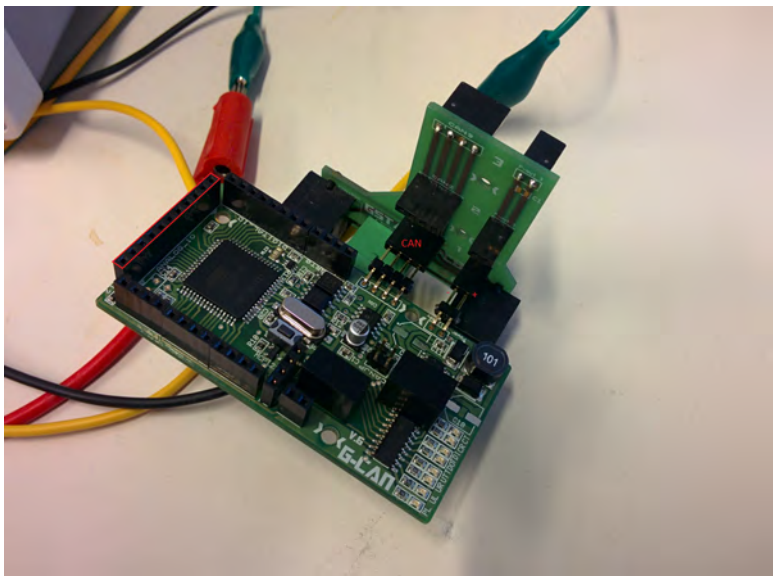


Figure 56: CAN card connected to a stack.

Next, connect one end of the USB cable to the Atmel ICE debugger and the other end to the computer. A red LED should be lit up (Figure 57). The computer should recognize that a new device has been inserted and the appropriate driver should be automatically installed.

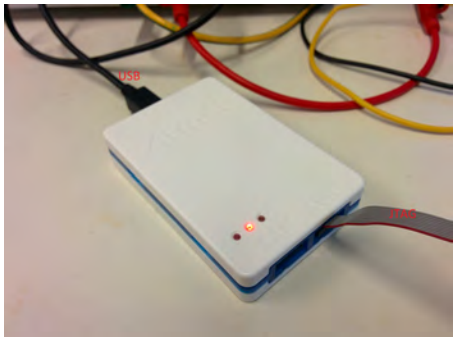


Figure 57: Atmel ICE debugger with both USB and JTAG cables connected to it.

In Figure 58 the JTAG cable has been connected into the CAN card. How the cable order is can also be seen in Figure 58 in the bottom right corner. For the

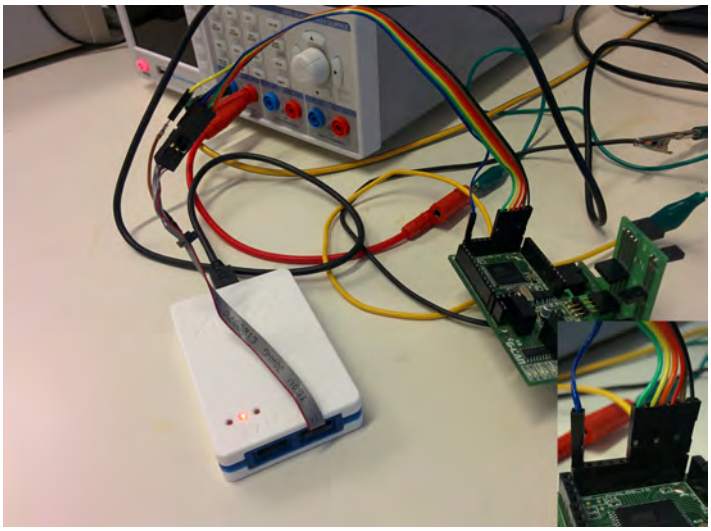


Figure 58: JTAG cable connected to the CAN card.

power supply set the voltage to 24V and the current to 100mA. Once the power supply is started it should be lighted up as in Figure 59. If the CAN cards on

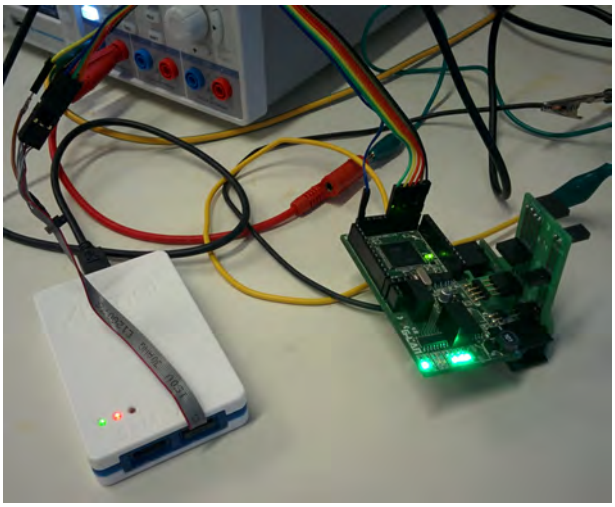


Figure 59: Everything is up and running.

the Butler robot needs to be programmed you do not need a power supply or a stack. What is only needed is the JTAG cable, Atmel ICE debugger, and the USB cable. In Figure 60 the JTAG cable is connected to the CAN card sitting on the robot. One thing to remember is to have the lonely pin on the opposite side of where the power and the CAN connectors are. When the JTAG cable is in place power on the robot and it should be possible to download a program into the CAN card.

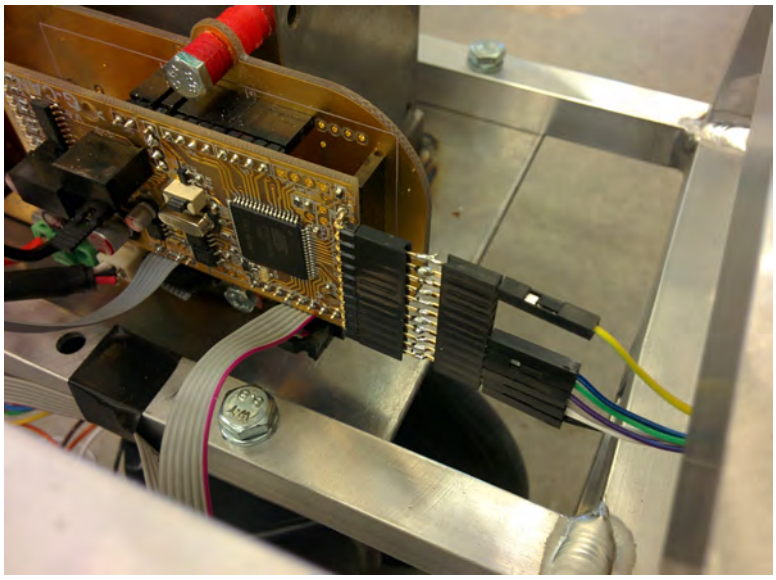


Figure 60: Connecting the JTAG cable directly to a CAN card on the Butler.

11.7.5 Download program code into the CAN card - Jakob Danielsson

When the environment is up and running it is now possible to download code into the CAN cards. The following list will go through step-by-step how to accomplish this task.

1. Open up *Atmel Studio* and press *Ctrl + Shift + P* to open up the *Device Programming* window.
2. Under *Tools* choose *Atmel-ICE*, and under *Device* choose *AT90CAN128*. For the interface choose *JTAG*. Hit apply.
3. To read the target voltage press the *read* button to the right below the lable *Target Voltage*. If the read voltage is around 5V, then everything is fine. This part is not mandatory but it is useful to see if the connection with the CAN card is fully working.
4. Choose *Memories* from the left menu.
5. As can be seen in Figure 61, first choose a file to flash. Press the button with the dots and navigate to the *target* folder. The folder might appear to be empty, so choose *All files (*.*)* from the list located at the bottom right corner. A *main* file should appear. Select the file and press *open*.
6. Once the file has been selected to be downloaded into the CAN card press the *Program* button.
7. When the programming is done, the code should now be on the CAN card.

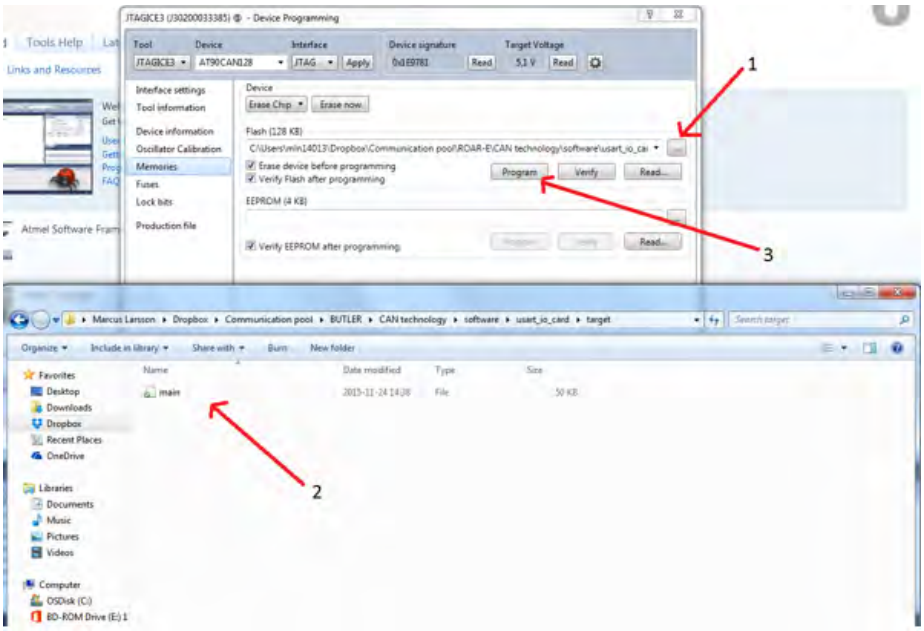


Figure 61: Select a file to be downloaded into the CAN card.

11.7.6 How to use Kvaser Leaf Light v2 and the application CANKing to monitor the CAN bus - Marcus Larsson

This section will describe how to use the hardware Kvaser Leaf Light v2 and the program CANKing to monitor the messages on the CAN bus. In Figure 62 the white cable is connected into the stack and the USB cable from the device to the computer. Once the Kvaser Leaf Light v2 has power a green LED will be turned on. A yellow LED will toggle after each time a CAN message on the bus is being monitored.

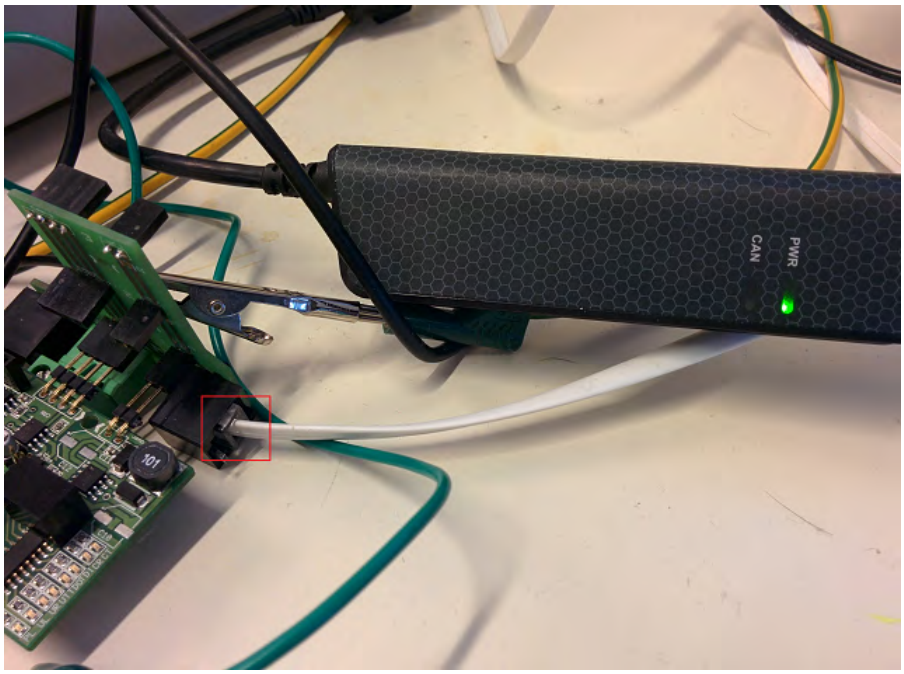


Figure 62: Kvaser Leaf Light v2 connected to the stack.

The application CANKing developed by Kvaser needs to be installed. The software can be found at [13]. Once the software is installed, follow the steps below (can also be seen in Figure 63).

1. The first window that appear after the application is started is *Kvaser CanKing*. Choose *Template* and hit *OK*.
2. In the second window *Templates* choose *CAN Kingdom (2 channels)* and then press the *OK* button.
3. Several windows should now appear. Focus on the window called *CAN 1* and press the button *Go On Bus*.
4. All the messages inside the CAN bus should now appear in the window *Output Window*.

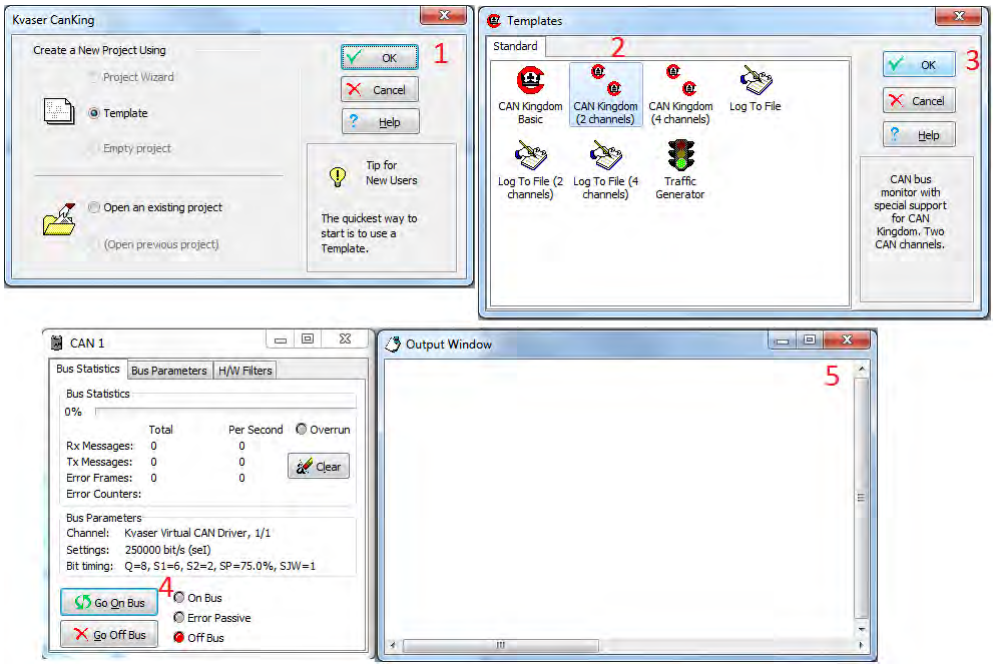


Figure 63: Steps on how to start the CAN bus monitoring.

11.7.7 Send in commands to the translator card using Hercules

When sending in commands from the computer to the translator card during development a software called *Hercules* from HW group has been used. This software allows you to send text strings out via the serial port. To actually be able to send commands one USB to serial cable is needed. The cable used in this project has been bought from [11]. The USB part of the cable goes into the computer and the serial part into the CAN cards. How the serial connection looks like can be seen in Figure 64.

After the USB to serial cable has been connected correctly follow these steps to find out which COM port the cable is connected to.

1. Open *Run (Win + R)* and write *mmc devmgmt.msc* to open the *Device Manager*.
2. Navigate down to *Ports (COM & LPT)*.
3. Look for *Prolific USB-to-Serial* or similar and take notice on the number after *COM*.

Next, start *Hercules* and select the *Serial* tab at the top.

1. To the right, choose the correct COM port which could be seen in the *Device Manager*.
2. Set baud rate to 115200.
3. Press the button *Open*. As long as no other application use the same COM port, you should now be able to start sending commands.

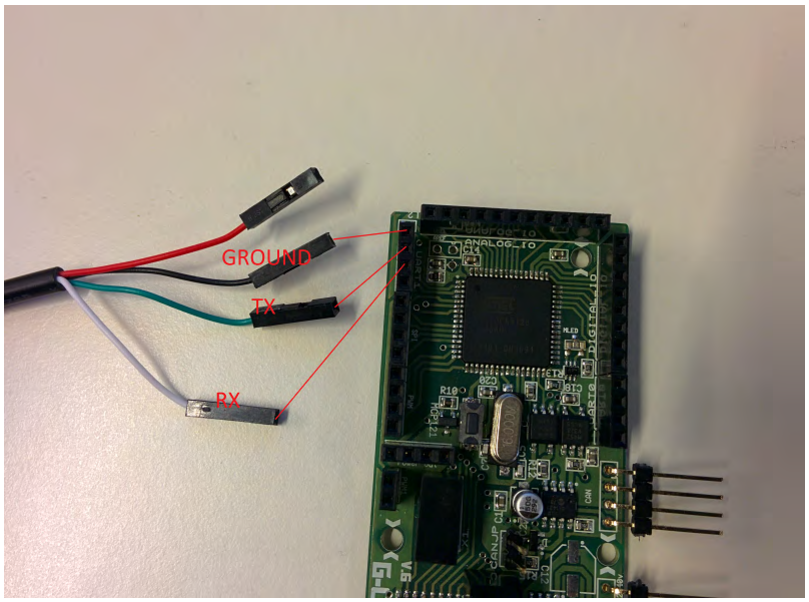


Figure 64: How to connect the serial pins from the cable to the CAN card.

Table 13: Commands for the drive motor.

Command	Front left	Front right	Back left	Back right
Set torque	:73151164FF<CR>	:7D151164FF<CR>	:87151164FF<CR>	:91151164FF<CR>
Check torque	:731F0009FF<CR>	:7D1F0009FF<CR>	:871F0009FF<CR>	:911F0009FF<CR>
Enable motor	:73154000FF<CR>	:7D154000FF<CR>	:87154000FF<CR>	:91154000FF<CR>
Disable motor	:73150C10FF<CR>	:7D150C10FF<CR>	:87150C10FF<CR>	:91150C10FF<CR>
Set 0 speed	:7301005AFF<CR>	:7D01005AFF<CR>	:8701005AFF<CR>	:9101005AFF<CR>
Move forward	:7301205AFF<CR>	:7D01205AFF<CR>	:8701205AFF<CR>	:9101205AFF<CR>
Move backward	:7301A05AFF<CR>	:7D01A05AFF<CR>	:8701A05AFF<CR>	:9101A05AFF<CR>

In Table 13 are some of the commands for the drive motor listed. The <*CR*> text is one character and translates into *carriage return*. All commands are sent in as hexadecimals. In Table 14 are some of the commands for the servo motor listed. Below are how to use *Hercules* to send commands. Figure 65 also

Table 14: Commands for the servo motor.

Command	Front left	Front right	Back left	Back right
Set torque	:731611C4FF<CR>	:7D1611C4FF<CR>	:871611C4FF<CR>	:911611C4FF<CR>
Check torque	:73200009FF<CR>	:7D200009FF<CR>	:87200009FF<CR>	:91200009FF<CR>
Enable motor	:73160E19FF<CR>	:7D160E19FF<CR>	:87160E19FF<CR>	:91160E19FF<CR>
Disable motor	:73160E18FF<CR>	:7D160E18FF<CR>	:87160E18FF<CR>	:91160E18FF<CR>
Set offset	:73020138FF<CR>	:7D0201A7FF<CR>	:87020900FF<CR>	:910202F5FF<CR>
32 degree angle	:73010020FF<CR>	:7D010020FF<CR>	:87010020FF<CR>	:91010020FF<CR>
90 degree angle	:7301005AFF<CR>	:7D01005AFF<CR>	:8701005AFF<CR>	:9101005AFF<CR>
160 degree angle	:730100A0FF<CR>	:7D0100A0FF<CR>	:870100A0FF<CR>	:910100A0FF<CR>

illustrates the process.

1. The first command in the first text box sets the torque for the drive motor for the front left wheel. Press the *Send* button to send the command. One can also use the F1 key to send the command.
2. The second command in the second text box also sets torque value for the drive motor but for all wheels. It is possible to combine several commands in one sending. Pressing the F2 key or the *Send* button to the right transmits the command.
3. In the third text box there is four commands combined that enables all drive motors. Pressing the F3 key or the *Send* button to the right sends the commands. It is important to note that the torque commands should be sent before enabling them.

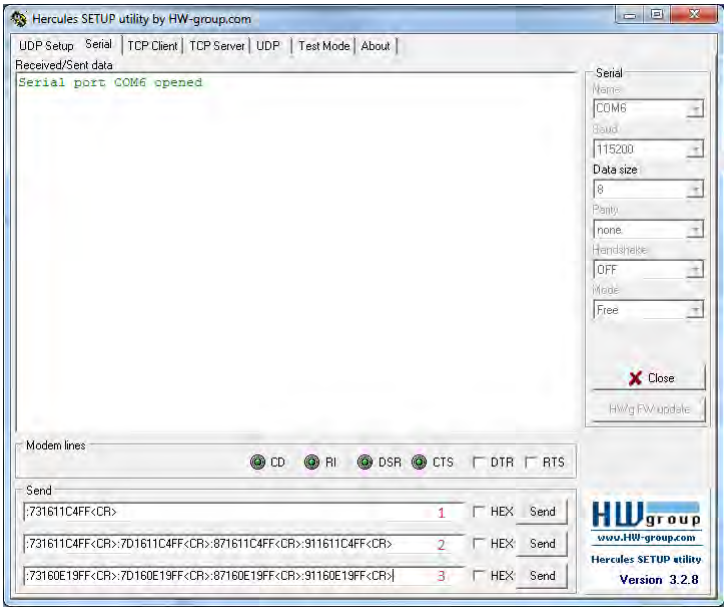


Figure 65: Sending command using *Hercules*.

When starting the Butler the following command order is recommended (the commands from Table 13 and 14 can be used).

1. Send a torque command to all motors.
2. Send a decay command to all motors.
3. Send in an enable command to all motors.
4. Send in the rotary encoder offset to the servo motors. The values for each servo motor can be found in Table 9. The commands can also be seen in Table 14. Also, remember to position the wheels correctly according to Figure 37 before sending the offset commands.

After the last command, you can now send in commands to turn the wheels or setting the speed.

11.7.8 Create a new ADA project from scratch in GNAT GPL 2012 AVR

Creating a new project can be tricky. This guide will hopefully make this process easier.

1. First, go to the *software* folder where all the main projects are located.
2. Create a new folder where the new project will be stored.
3. Inside the new folder, create three new folders:
   - **build** - all build files that will be created during compilations will get created here
   - **src** - location for the files with the source code
   - **target** - the location for the compiled file
4. Start the program GNAT GPL 2012 AVR.
5. Select *Create new project with wizard* and press *OK*.
6. In the next window select *Single Project* and press *Forward*.
7. Enter the project name and then navigate to the new folder created in step 2. Press *Forward* three times.
8. Remove the directory and press the button *Add* and navigate to the created *src* folder. Press the *Forward* button.
9. Choose the *build* folder under the *Build directory*.
10. Choose the *target* folder under the *Exec directory*.
11. Press the button *Apply*.
12. Create a new file by hitting the *Create a New File* button (Figure 66) and write a simple code that can be viewed in Figure 67.
13. Save the file as *main.ads* in the *src* folder.
14. As of now there is only one more thing to do: setting up compiler paths and flags. The easiest way to do this is to open up a previous project. Open up the project *torso_modoule* by opening the *torso_module.gpr* file.
15. In the *Project* window to the left, right click on *Torso_Module* and navigate down to *Project* and select *Edit source file*.

16. Open up the same file from your newly created project by following the same steps as above. Two GNAT GPR 2012 AVR instances should now be opened.
17. Copy the lines 1-8 from *torso_module.gpr* source file and paste those lines at the same position in your new project source file.
18. Copy lines 16-29 from *torso_module.gpr* source file and paste those lines at the same position in your new project source file.
19. Once this is done, close both GNAT GPR 2012 AVR instances and open up the newly created *.gpr* file in the folder created in step 2.
20. You should now be able to compile the code by pressing the *Build All* button. If no errors occurs then everything should be working.

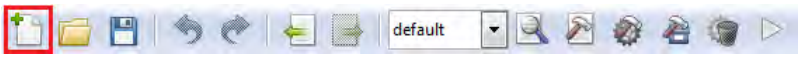


Figure 66: Picture showing where the *Create a New File* button is located.

```
procedure Main is
   pragma Suppress (All_Checks);
begin
    null;
end Main;
```

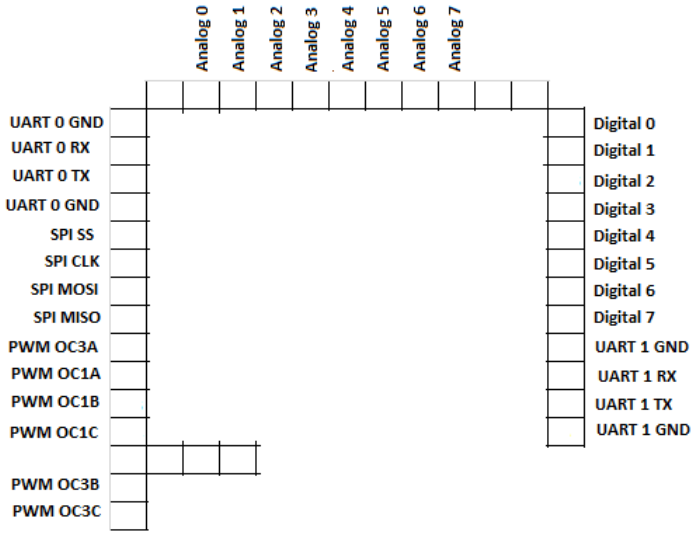
Figure 67: A simple ADA code.

11.7.9 Good things to know about - Jakob Danielsson

It is not recommended to use another compiler than GNAT GPL 2012 AVR, even when writing single functions to be transferred to the real code. This is due to restrictions of the GNAT GPL 2012 AVR. An example is when the MODBUS protocol was written. The MODBUS protocol was written in GNAT GPL 2015, and when trying to transfer that code into GNAT GPL 2012 AVR, several errors occurred which took a long time to solve.

When searching for the pins on the CAN cards from the processor legs, it may be a little difficult to find the correct mappings. Using Figure 68 below can make it a little easier to find the desired pins on the CAN cards. More information about the pin mapping can be found in the datasheet for the microprocessor at [14].

Figure 68: Processor pin mapping.



There exists two different jumper connections, one CAN jumper header and one ADC jumper header. In order to get CAN communication to work properly, one CAN jumper must be placed onto the CAN card. The same goes for the ADC, if ADC is going to be used, the ADC jumper must be in place. See Figure 69 for the location of the jumpers.

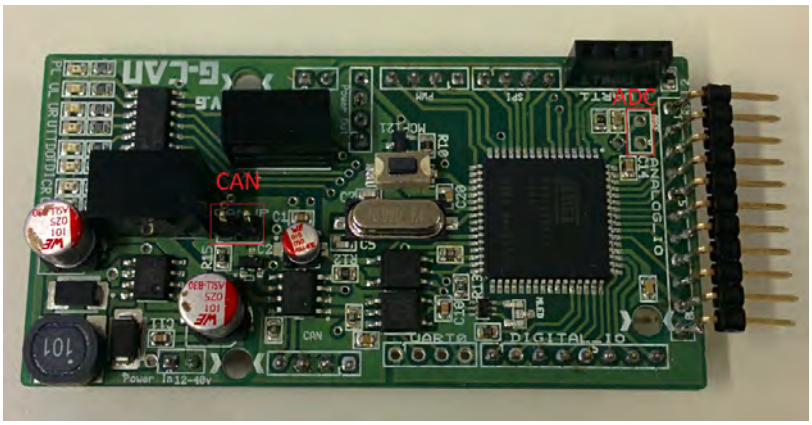


Figure 69: A picture showing where the CAN and ADC jumper should be placed if needed.

Table 15: Settings for the fuses for the CAN cards.

BODLEVEL	DISABLED
TA0SEL	[ ]
JTAGEN	[ ]
SPIEN	[X]
WDTON	[X]
EESAVE	[ ]
BOOTSZ	4096W_F000
BOOTRST	[ ]
CKDIV8	[ ]
CKOUT	[ ]
SUT_CKSEL	EXTXOSC_8MHZ_XX_16KCK_0MS
EXTENDED	0xFF (valid)
HIGH	0x99 (valid)
LOW	0xDF (valid)

When a new CAN card has been manufactured and soldered the fuses need to be programmed.

1. To do so, open up *Atmel Studio* and open up the *Device Programming* window by pressing *Ctrl* + *Shift* + *P*.
2. Select *Atmel-ICE* as the tool and *AT90CAN128* as the device. Choose *JTAG* from the interface menu and hit the *Apply* button.
3. Navigate to *Fuses* on the left menu.
4. Here you can set the settings for the fuses. See Table 15 how to configure the fuses.
5. Press the *Program* button to download the fuse settings into the CAN card. Once finished, the CAN card is ready to be used.

The current mapping of the UART pins is wrong. UART port 0 in the code is actually UART1 on the CAN card, and reverse.

Regarding the wheel modules, the code for the interrupt routines are commented in the *torque_interrupt* file, as well as the initialization part in the main file. To test the interrupts, one must uncomment the code.

11.7.10 Error messages - Marcus Larsson

When an error has occurred a CAN message will be sent with an error code. The message contents include the module number and an error value. The module number indicate from where the error has been triggered and the error value indicate the type of error. All error types are listed below.

1. Function code not handled.
2. The received ID is not correct.
3. Stop motor command failed.
4. Status bits from encoder returned a read failure (torso).
5. Status bits from encoder returned a read failure (drive).
6. Status bits from encoder returned a read failure (servo).
7. Gripper error, wrong data being sent to the gripper.
8. Error setting rotary encoder offset (value must be in 0-4095 range)
9. Write to motor controller error (wrong motor parameters).
10. Read from motor controller error (wrong motor parameters).
11. Error starting motor (wrong parameters?).
12. Error stopping motor (wrong parameters?).
13. New given angle is out or range.
14. Error shutting down motor controller.
15. Trying to request data when the broadcasting is activated.
16. Change of speed failed.
17. Normal servo motor change stop, went out of range (overshoot?).
18. Initiate servo motor change stop, went below 0 or above 4095.

11.8 Future work - Marcus Larsson, Jakob Danielsson

When it comes to future work there are some implementation that are left to do. Also, some testing of the functionality that are already implemented is needed. From the tests performed one can conclude that there are some issues that need more work to create a more stable Butler robot. Down below the issues will be listed.

1. When receiving speed data from the rotary encoders, the speed flutters a lot. There are two possible solutions. One is to implement a filter to remove the bad sensor values. The second one is to move the rotary encoder so it is right above the magnet placed on the motor. A combination of both solutions might also be a good idea.
2. Setting wheel turn position can be tricky since the wheel will only turn between 0-4095 and not above the maximum 4095 or lower than zero. It is recommended to fix this so the wheel turn always takes the shortest route, and not the longest when there is a shorter one.

3. Sending commands to the motor controller might be ignored for an unknown reason. This happens especially when sending in the torque command but it might be for other types of motor controller commands as well. It seems to be a bug in the SPI implementation, which is the same implementation used in an earlier project. A thorough investigation is needed to solve this problem.

Next, there are new implementations that are required to be able to get the torso and the gripper to work. These are pointed out in the list below.

1. Functions for controlling the step motors for torso and gripper are not implemented. The step motor functions for the Wheel module could be used to some extent with modifications.
2. The functionality for setting the height for the torso is missing as well as functions for opening and closing the gripper.
3. The initialization loop is mostly done, however, a few functionalities are missing. Most of them are commented with a *TODO* text in the code. More specific what is missing are the commands for controlling the torso and the gripper. If the functionality for setting the height for torso and handling the gripper is implemented this should be done quickly.

The height of the torso is calculated by the amount of spins the motor is taking. As of now one motor spin is set to equal two centimeters, but it might be a good idea to carefully measure that again. The first measurement was done quickly and not so thoroughly and was just done to get a quick idea of how far the torso would travel after one motor spin.

For the torque interrupt functionality there is a lot of testing that needs to be done. As of now no testing regarding interrupts has been done due to time limitations.

A possible solution of optimizing the system would be to remove the translator card and instead use a USB to CAN converter. By doing this, the transmission time between the translator card and the CAN bus will be removed as the Simulink model will send the messages straight onto the CAN bus.

References

1. Engelberger (1997). A project proposal. Service Robots, Personal Communication (Engelberger and Christensen), 1997.

2. RoboCup@Home 2014 rulebook,[Online]. Available: http://docs.google.com/viewer?a=v&pid=sites&srcid=cm9ib2N1cGF0aG9tZS5vcmd8cm9ib2N1cC1ob21l

3. Operation Manual: AS5145B-EK-AB-STM1.0. [Online]. Available: https://ams.com/eng/content/download/302565/1085449

4. AS5145 Datasheet. [Online]. Available: http://ams.com/eng/content/download/50206/533867/AS5145_Datasheet_v1-15.pdf

5. CAN calulator.[Online] Available: http://www.esacademy.com/en/library/calculators/can-best-and-worst-case-calculator.html

6. The Engineering Toolbox Screw Jacks,[Online]. Available: http://www.engineeringtoolbox.com/screw-jack-d_1308.html

7. HW group. Hercules SETUP utility. [ONLINE] Available at: http://www.hw-group.com/products/hercules/index_en.html. [Accessed 18 January 16].

8. LIBRE. Download GNAT GPL. [ONLINE] Available at: http://libre.adacore.com/download/configurations. [Accessed 18 January 16].

9. Atmel. Atmel Studio. [ONLINE] Available at: http://www.atmel.com/tools/ATMELSTUDIO.aspx. [Accessed 18 January 16].

10. Farnell. 2016. ATMEL ATJTAGICE3 DEBUGGER, JTAG, SPI, PDI, FOR AVR. [ONLINE] Available at: http://se.farnell.com/atmel/atjtagice3/debugger-jtag-spi-pdi-for-avr/dp/1972230. [Accessed 18 January 16].

11. Kjell & Company. USB till seriell-adapter för Arduino. [ONLINE] Available at: http://www.kjell.com/se/sortiment/el/elektronik/arduino/tillbehor/usb-till-seriell-adapter-for-arduino-p87898. [Accessed 18 January 16].

12. Kvaser. Kvaser Leaf Light HS v2. [ONLINE] Available at: https://www.kvaser.com/products/kvaser-leaf-light-hs-v2/. [Accessed 18 January 16].

13. Kvaser. Downloads. [ONLINE] Available at: https://www.kvaser.com/downloads/. [Accessed 18 January 16].

14. Atmel. AT90CAN128. [ONLINE] Available at: http://www.atmel.com/images/doc7679.pdf. [Accessed 19 January 16].

A Appendix A

A.1 BOM

BOM - Encoder

BOM					
Name	Vendor	Ordernr	RefDes	Value	Shape
Ceramic Capacitor	WÃijrth	885012106006	C1	10uF	CAPC1608X95N
Ceramic Capacitor	WÃijrth	885012106006	C2	10uF	CAPC1608X95N
Ceramic Capacitor	WÃijrth	885012206020	C3	100nF	CAPC1608X95N
Rotary Encoder	AMS		U1	AS5145	AS5145
2.54mm Male Box Header	WÃijrth	61200621621	U2	Male Box Header	Male connector

Table 16: BOM - Encoder

A.2 Schematics and CAD

The schematics have been made in multisim and the CAD files have been made in ultiboard.

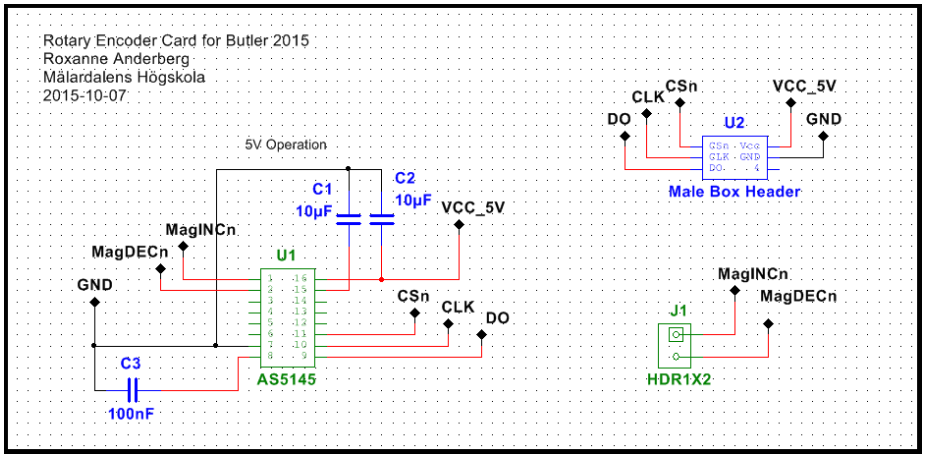


Figure 70: Schematics for the pin-outs on the cape

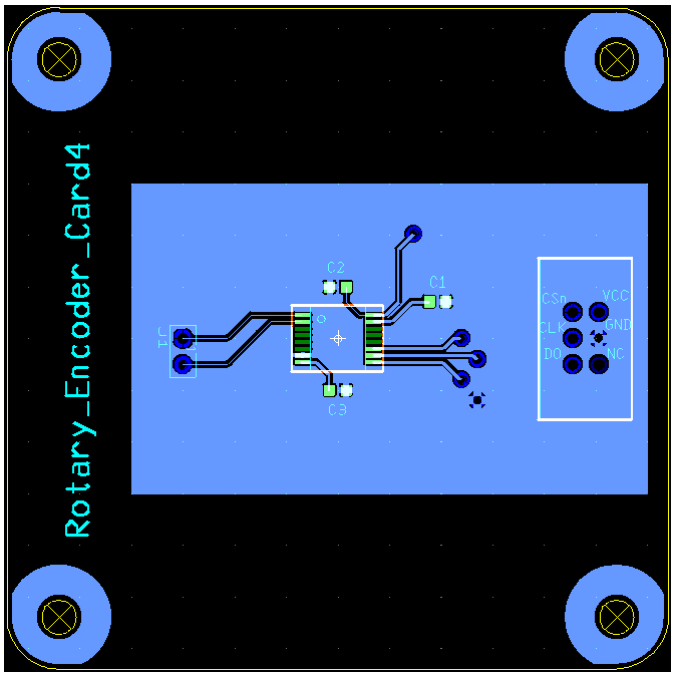
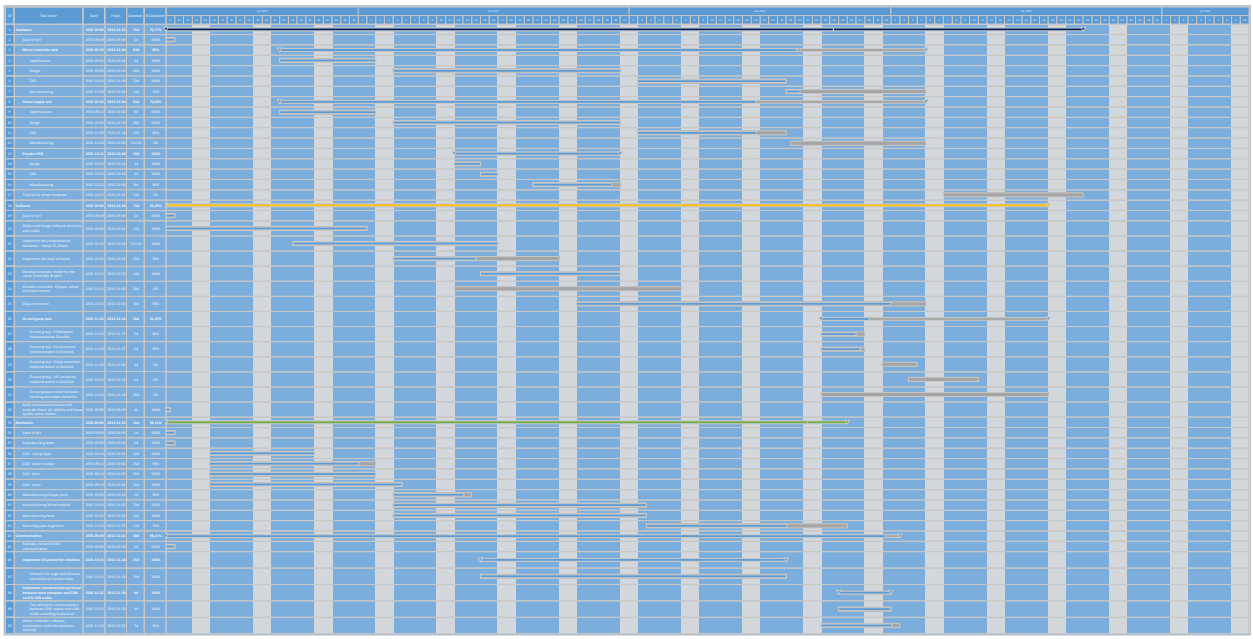


Figure 71: Top Layer

A.3 GANT Schedule



A.4 Communication manual

BUTLER – MDH@HOME

Communication manual: Simulink and CAN nodes

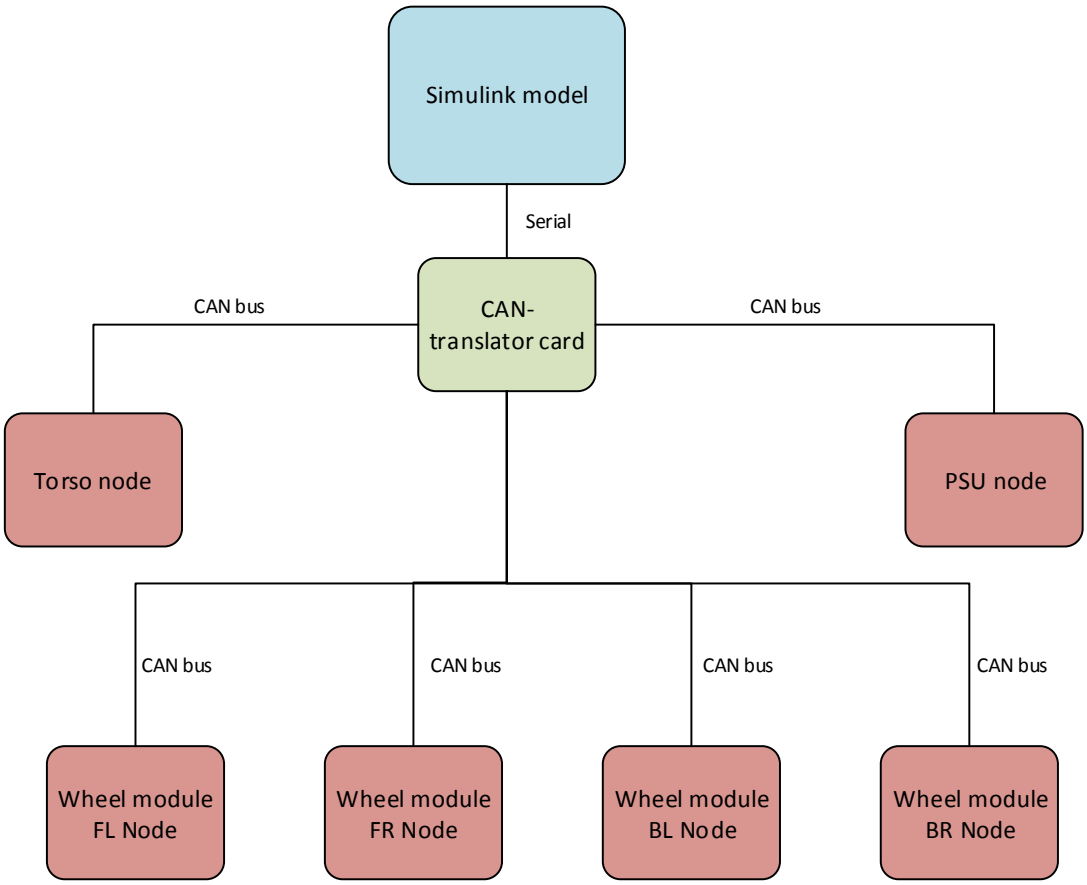
Contents

1.1 Structure and description of the communication

At the top level there is NUC computer running the Simulink model. See picture below.
From Simulink on the computer have a USB to serial connection to the CAN translator card which is connected to the CAN nodes via a CAN bus.

From the Simulink model which is the top level of the system it's possible to send different commands down to each of the CAN nodes. The communication protocol used for sending the different commands is based on the Modbus ASCII communication protocol. In order to reach a specific node a unique address is set for each of the nodes.
When a message is sent from Simulink to the CAN-translator, the message is first decoded and then translated into a valid CAN message. The message is then transmitted out on the CAN-bus with the corresponding CAN message.

Each of "wheel module nodes" is connected to two motor-controllers, and two rotary encoders. The two characters after each of the "Wheel module" indicates the location on the robot where the node is located, for instance FL stands for front left, BR stands for back right.
The Torso node is connected to two motor-controllers and one micro-switches.
The PSU node is connected to a battery system. For each of the nodes there is a specified list of available commands which can be sent from Simulink



Modbus ASCII

The communication uses a simple limited version of modus. Fields are represented using characters. This means that when using hexadecimal addresses to represent the data, the effective data is 1 byte whereas the overall data is twice as big.

The messages are sent over UART using the following configuration.

- Baudrate: 115 200
- Paritybit: none
- StopBit: none

Header

The following header is used for Modbus ascii communication.

Start	Address	Function	Data	Checksum	End
:	2 chars (2 byte)	2 chars (2 byte)	4 chars (4 byte)	2 chars (2 byte)	1 char (1 byte)

Start

The start of each message begins with the character ":"

Address

The address field is set to the address of the responding node. In order to reach a specific CAN node on the bus, the first bit in the address filed will correspond the requested CAN-node.

Function Code (FC)

The function code is used to indicate the purpose of the Modbus message.

Data

The data is limited to 4 bytes, and since the hexadress that is used is represented by two characters, each of them will have the size of 2 bytes.

The different types of data that can be sent within the data field is listed in the table below:

Description	Unit	Data-type
Speed (All motors)	0-127 [cm/s] : Reverse 128 – 255 [cm/s]: Forward	uint8 (1 byte)
Height (Torso motor)	0-255 [cm]	uint8 (1 byte)
Angle (Servo motors)	0-180 [degrees]	uint8 (1 byte)
Angular rotary encoder data	0-4095	uint16 (2 byte)
Motor Controller command	Binary array	uint16(2byte)

2 CAN Translator node

Base address, 1st byte: 104(Dec) - 0x68 (Hex)

2.1 Write commands

Address (dec)	Function code (dec)	Description	Data length [bytes]
Motor and encoder commands			
105	1	Start broadcasting data – command (All wheel modules)	0
104	2	Stop – command (All wheel modules)	2

2.2 Read commands

Address (dec)	Function code (dec)	Description	Receive data length [bytes]
Motor and Encoder			
105	3	Speed & Angle request (All wheel modules)	0
Automatic response			
115,125,135,145	42	Speed & Angle	2 X 4
96	11	Stop acknowledgement	2
20	12	General Error	2
20	13	Torque Error	2

2.1.1 Description of **write** commands

2.1.1.1 Start broadcasting data (All wheel modules)

Description:

By sending the "Start command wheel modules", all wheel modules nodes will begin to reply by continuous sending Speed and Angle information with a specified frequency first to the CAN-translator and then up to Simulink.

Message:

0x3A	0x69	0x01	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Empty [0]	Empty [0]

The data field will be empty when sending the start command

2.1.1.2 Stop (All wheel modules)

Description:

By sending a “stop command wheel modules” the specified motors that are represented in the data field will stop on each of the wheel modules by sending out four stop commands from the CAN-translator card.

Message:

0x3A	0x68	0x02	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Stop Drive motors [0-1]	Stop Servo motors [0-1]

The possible stop states are:

Stop all drive motors:

Data byte 1	Data byte 2
1	0

Stop all servo motors:

Data byte 1	Data byte 2
0	1

Stop both drive and servo motors:

Data byte 1	Data byte 2
1	1

2.2.1 Description of **read** commands

2.2.1.1 Speed & Angle request (All wheel modules)

Description:

By sending a “Speed & Angle request (All wheel modules)”, all four wheel modules will reply with the actual speed and angle to the translator card.

Message:

0x3A	0x69	0x03	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2

Empty [0]	Empty [0]

The data field will be empty when sending the start command

2.2.1.2 Speed & Angle

Description:

This message will be automatically replied up to the translator card as a result of sending the command "Speed & Angle request (All wheel modules)". The Speed and Angle will be replied for all of the wheel modules so in total four messages will be sent to the translator card. The translator card will then send each of the messages up to the Simulink model.

Message:

0x3A	0x73	0x2A	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Message:

0x3A	0x7D	0x2A	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Message:

0x3A	0x87	0x2A	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Message:

0x3A	0x91	0x2A	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Speed[0-255]	Angle[0-180]

The speed byte is split into a forward direction and a backward direction, the forward speed is specified within 0-127 (dec) and the backward speed 128-255 (dec). This means that the maximum possible speed in both directions is 0-127 cm/s. The second data byte contains information about the current angle the servo motor has. This can be within the range 0-180 degrees.

2.2.1.3 Stop acknowledgement

Description:

The stop acknowledgement is sent after the motors specified has stopped.

Message:

0x3A	0x60	0x03	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Node ID [1-2]	Motor ID [1-3]

Node ID describes which that either all of the wheel module generated a stop acknowledgement, or that the torso node generated a stop ack. The motor ID describes which of the motors that stopped.
A description of the valid values can be seen in table below:

Data byte 1 – Node ID	Data byte 2 – Motor ID
1 : Wheel modules	1: Drive motors
	2: Servo motors
	3: Both drive and servo motor
2 : Torso module	1: Torso motor
	2: Gripper motor
	3: Both torso and gripper motor

2.2.1.4 General error

Description:
The Error message is a general message that can contain various errors from the different nodes.
(The error code specified can be expanded as the project grows and if an indication of that more error codes needs to be introduced).

Message:

0x3A	0x14	0x04	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Node ID [1-5]	Error code [0-255]

The first byte in the data field will describe which of the nodes that the error comes from, and the second byte will describe which kind of error the node encountered. (The error code specification will be developed as the project goes on).

Valid values for **data byte 1:**
1: Indicates that the **FL** wheel node generated an error
2: Indicates that the **FR** wheel node generated an error
3: Indicates that the **BL** wheel node generated an error
4: Indicates that the **BR** wheel node generated an error
5: Indicates that the **Torso** node generated an error

2.2.1.3 Torque error

Description:

The torque error will be generated when a motor has stalled on a wheel module node. Which one of the two motors that generated the torque error will be specified within the data field.

Message:

0x3A	0x14	0x05	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Node address [1-5]	Motor ID for the generated error [1-2]

The first byte will describe which of the nodes that the torque error was generated on, and the second byte will describe which one of the motors that generated the torque error.

Data byte 1 – Node ID	Data byte 2 – Motor ID
1-4: Wheel modules	1: Drive motors
	2: Servo motors
5 : Torso module	1: Torso motor
	2: Gripper motor

3 Wheel module node FL (Front Left)

Base address, 1st byte: 114(Dec) – 0x72(Hex)

3.1 Write commands

Address (dec)	Function code (dec)	Description	Data length [bytes]
Motor and Encoder commands			
115	1	Speed (motor: drive), Angle (motor: servo)	2
115	2	Set rotary encoder angular offset	2
115	3	Start broadcast encoder data (Speed, Angle)	
114	4	Stop motors	2
PID controller commands			
115	11	Set 12-bit angular encoder position (motor: servo)	2
Motor controller commands			
115	21	Write register motor-controller 1	2
115	22	Write register motor-controller 2	2

3.2 Read commands

Address (dec)	Function code (dec)	Description	Send: Data length [bytes]	Receive: Data length [bytes]
Motor controller commands				
115	31	Read register motor-controller 1	1	2
115	32	Read register motor-controller 2	1	2
Motor and Encoder commands				
115	41	Read 12 bit angular encoder position (motor: servo)	0	2
115	42	Speed (motor: drive), Angle (motor: servo)	0	2
Automatic response				

115	51	Speed (motor: drive), Angle (motor: servo)	0	2
20	52	General error	0	2
20	53	Torque error	0	2
20	54	Stop acknowledge motors	0	2

4 Wheel module node FR (Front Right)

Base address, 1st byte: 124 (Dec) – 0x7C (Hex)

4.1 Write commands

Motor and Encoder commands			
125	1	Speed (motor: drive), Angle (motor: servo)	2
125	2	Set rotary encoder angular offset	2
125	3	Start broadcast encoder data (Speed, Angle)	2
124	4	Stop motors	2
PID controller commands			
125	11	Set 12-bit angular encoder position (motor: servo)	2
Motor controller commands			
125	21	Write register motor-controller 1	2
125	22	Write register motor-controller 2	2

4.2 Read commands

Address (dec)	Function code (dec)	Description	Data length [bytes]
Motor controller commands			
125	31	Read register motor-controller 1	2
125	32	Read register motor-controller 2	2
Motor and Encoder commands			
125	41	Read 12-bit angular encoder position (motor: servo)	2
125	42	Speed (motor: drive), Angle (motor: servo)	2
Automatic response			
125	51	Speed (motor: drive), Angle (motor: servo)	2
20	52	General error	2

20	53	Torque error	2
20	54	Stop acknowledge motors	2

5 Wheel module node BL (Back Left)

Base address, 1st byte: 134 (Dec) - 0x86 (Hex)

5.1 Write commands

Motor and Encoder commands			
135	1	Speed (motor: drive), Angle (motor: servo)	2
135	2	Set rotary encoder angular offset	2
135	3	Start broadcast encoder data (Speed, Angle)	2
134	4	Stop motors	2
PID controller commands			
115	11	Set 12-bit angular encoder position (motor: servo)	2
Motor controller commands			
135	21	Write register motor-controller 1	2
135	22	Write register motor-controller 2	2

5.2 Read commands

Address (dec)	Function code (dec)	Description	Send: Data length [bytes]	Receive: Data length [bytes]
Motor controller commands				
135	31	Read register motor-controller 1	1	2
135	32	Read register motor-controller 2	1	2
Motor and Encoder commands				
135	41	Read 12-bit angular encoder position (motor: servo)	0	2
135	42	Speed (motor: drive), Angle (motor: servo)	0	2
Automatic response				
135	51	Speed (motor: drive), Angle (motor: servo)	0	2
20	52	General error	0	2

20	53	Torque error	0	2
20	54	Stop acknowledge motors	0	2

6 Wheel module node BR (Back Right)

Base address, 1st byte: 144 (Dec) - 0x90 (Hex)

6.1 Write commands

Motor and Encoder commands			
145	1	Speed (motor: drive), Angle (motor: servo)	2
145	2	Set rotary encoder angular offset	2
145	3	Start broadcast encoder data (Speed, Angle)	0
144	4	Stop motors	2
PID controller commands			
145	11	Set 12-bit angular encoder position (motor: servo)	2
Motor controller commands			
145	21	Write register motor-controller 1	2
145	22	Write register motor-controller 2	2

6.2 Read commands

Address (dec)	Function code (dec)	Description	Send :Data length [bytes]	Receive : Data length [bytes]
Motor controller commands				
145	31	Read register motor-controller 1	1	2
145	32	Read register motor-controller 2	1	2
Motor and Encoder commands				
145	41	Read 12-bit angular encoder position (motor: servo)	0	2
145	42	Speed (motor: drive), Angle (motor: servo)	0	2
Automatic response				
145	51	Speed (motor: drive), Angle (motor: servo)	0	2
20	52	General error	0	2

20	53	Torque error	0	2
20	54	Stop acknowledgement motors	0	2

7.1 Description of **write** commands for wheel module nodes

7.1.1 Speed, Angle command

Description:

The speed and angle command are sent to set a specific speed at the drive motor likewise to set a specific angle at the servo motor.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Speed [0-255]	Angle [0-180]

The speed byte is split into a forward direction and a backward direction, the forward speed is specified within 0-127 (dec) and the backward speed 128-255 (dec). This means that the maximum possible speed in both directions is 0-127 cm/s. The second byte will contain the angular value that the servo motor shall be set to.

7.1.2 Set rotary encoder angular offset

Description:

The angular offset command will set the offset value required for the encoder to be correctly calibrated. The motor will turn into the specified offset value.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Angular offset byte 1	Angular offset byte 2

The angular offset value is an int16 value, which means that it has to be split up into two bytes, hence the two angular offset byte 1 and 2.The angular offset value will be within the range [0-4096].

7.1.3 Start broadcast encoder data (Speed, Angle)

Description:

The start command are sent to begin automatically send messages containing the speed and angle information from the specified wheel module up to the CAN-translator and then forwarded up to Simulink. The CAN-translator will automatically send four start commands to each of the wheel modules when a start command to the CAN-translator is sent.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Empty [0]	Empty [0]

7.1.4 Stop motor(s)

Description:

By sending a stop command, the specified motors that are represented in the data field will stop on the specified wheel module that are represented in the address.
By sending a stop command to the CAN-translator card, the CAN-translator will send four stop commands to each of the wheel modules.

Message:

0x3A	0x65	0x04	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Stop Drive motor [0-1]	Stop Servo motor [0-1]

The possible stop states are:

Stop drive motor:

Data byte 1	Data byte 2
1	0

Stop servo motor:

Data byte 1	Data byte 2
0	1

Stop both drive and servo motor:

Data byte 1	Data byte 2
1	1

7.1.5 Set 12-bit angular encoder position (motor: servo)

Description:

The set encoder command is used to set the servo motor in a specified position described by the raw encoder data in the interval [0-4095]. The encoder data that is set, corresponds to an angular value of the servo motor.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Encoder data byte 1	Encoder data byte 2

The encoder angular value is an int16 value, which means that it has to be split up into two bytes, hence the two angular offset byte 1 and 2.The angular offset value will be within the range [0-4095].

7.1.8 Write register motor-controller 1

Description:

With the “Write register motor controller 1”, each of the motor controllers that are connected to the servo motor can be configured at the top level, such as torque limit and stepping mode for the motor.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Register address and data byte 1	Register address and data byte 2

The data field will contain the register address and on the motor controller that the function and the data shall be written into. Both address and data are 16-bit, which means it is split up into 2 bytes when it is sent from Simulink down to the CAN-translator.

7.1.9 Write register motor-controller 2

Description:

With the “Write register motor controller 2”, each of the motor controllers that are connected to the driving motor can be configured at the top level, such as torque limit and stepping mode for the motor.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Register address and data byte 1	Register address and data byte 2

The data field will contain the register address and on the motor controller that the function and the data shall be written into. Both address and data are 16-bit, which means it is split up into 2 bytes when it is sent from Simulink down to the CAN-translator.

7.2 Description of **read** commands

7.2.1 Read register motor-controller 1

Description:

With the “Read register motor controller 1”, all registers on the motor controller that are connected to the motor that handles the turning can be read at the top level.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Send: Data fied

Data byte 1	Data byte 2
Empty [0]	Register address

When the read command is sent, the first data byte will be empty. The second data byte will contain the register address for the motor controller.

Receive: Data field

Data byte 1	Data byte 2

Register address and data byte 1	Register address and data byte 2

When the data is received at Simulink, the data field will contain the register address and on the motor controller that the function and the data shall be read. Both address and data are 16-bit, which means it is split up into 2 bytes when it is sent from Simulink down to the CAN-translator and vice versa.

7.2.2 Read register motor-controller 2

Description:

With the "Read register motor controller 2", all registers on the motor controller that are connected to the motor that handles the driving in forward/backward direction can be read at the top level.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Send: Data fied

Data byte 1	Data byte 2
Empty [0]	Register address

Receive: Data fied

Data byte 1	Data byte 2
Register address and data byte 1	Register address and data byte 2

The data field will contain the register address and on the motor controller that the function and the data shall be read. Both address and data are 16-bit, which means it is split up into 2 bytes when it is sent from Simulink down to the CAN-translator and vice versa.

7.2.3 Read 12-bit angular encoder position (motor: servo)

Description:

The read encoder command is used to read the raw encoder data. The encoder data that is read corresponds to the angular value of the servo motor. When sending the command, the data filed will be empty, while when receiving the data will contain the encoder data.

Message:

START	NODE ADDRESS	FUNCTION CODE	DATA	CHECKSUM	END
1	2	3	4-5	6	7

Receive Data field:

Data byte 1	Data byte 2
Encoder data byte 1	Encoder data byte 2

The angular value is an int16 value, which means that it has to be split up into two bytes, hence the two angular offset byte 1 and 2.The angular offset value will be within the range [0-4096]. The two bytes has then to be combined into an int16 in Simulink in order to read the value.

7.2.4 Speed, Angle

Description:

The Speed and Angle information will be replied when a request have been sent to the translator card.

Message:

0x3A	0x7D	0x05	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Speed [0-255]	Angle [0-180]

The speed byte is split into a forward direction and a backward direction, the forward speed is specified within 0-127 (dec) and the backward speed 128-255 (dec). This means that the maximum possible speed in both directions is 0-127 cm/s. The second data byte contains information about the current angle the servo motor has. This can be within the range 0-180 degrees.

7.2.5 General error

Description:

The Error message is a general message that can contain various errors from the node. This error is automatically sent to the CAN-translator when an error is encountered on a node.
(The error code specified can be expanded as the project grows and if an indication of that more error codes needs to be introduced).

Message:

0x3A	0x60	0x03	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Empty [0]	Error code [0-255]

The first byte in the data field is empty and the second byte will describe which kind of error the node encountered. (The error code specification will be developed as the project goes on).

7.2.6 Torque error

Description:

The torque error is sent automatically up to the CAN-translator when a torque error has been generated on a motor controller on a node. As soon as a torque error has been generated, three other stop commands will be sent from the CAN-translator to the other wheel modules nodes to stop all other motors. After this a STOP-acknowledge will be sent up to Simulink.

Message:

0x3A	0x60	0x03	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Node ID [1-4]	Motor that generated error [1-2]

The first data byte in the data field will describe which of the wheel module nodes that generated the torque error. The second byte will describe which of the motors that generated the torque error.

The valid values **data byte 1** can have:

1: Indicates that the **FL** wheel node generated the torque error
2: Indicates that the **FR** wheel node generated the torque error
3: Indicates that the **BL** wheel node generated the torque error
4: Indicates that the **BR** wheel node generated the torque error

The valid values **data byte 2** can have:

1: Indicates that a drive motor stalled
2: Indicates that the servo motor stalled

7.2.7 Stop acknowledgement

Description:

The Stop ACK is replied to Simulink when the motors have stopped as an action of a stop command.

Message:

0x3A	0x70	0x08	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Node ID [1-4]	Motor ID [1-3]

Node ID describes which of the wheel module nodes that generated the stop acknowledgement. The motor ID describes which of the motors that was stopped.
A description of the valid values can be seen in table below:

Data byte 1 – Node ID	Data byte 2 – Motor ID
1 – 4 : Wheel module	1 : Drive motors
	2 : Servo motors
	3 : Both drive and servo motor

8 Torso node

Base address, 1st byte: 154 (Dec) - 0x9A (Hex)

8.1 Write commands

Motor and Encoder commands			
155	1	Set height torso	1
155	2	Open/Close gripper	2
155	3	Init Torso	1
154	4	Stop motors	2
Motor controller commands			
155	21	Write register motor-controller torso	2
155	22	Write register motor-controller gripper	2

8.2 Read commands

Address (dec)	Function code (dec)	Description	Send :Data length [bytes]	Receive : Data length [bytes]
Motor controller commands				
155	31	Read register motor-controller torso	1	2
155	32	Read register motor-controller gripper	1	2
Motor and Encoder commands				
155	41	Read torso height	0	1
Automatic response				
155	51	Init acknowledgement	0	0
20	52	General error	0	2
20	53	Torque error	0	2
20	54	Stop acknowledge motors	0	2

8.1.1 Description of **write** commands

8.1.1.1 Set height torso

Description:

The height command are sent to set the torso at a specified height.

Message:

0x3A	0x9B	0x01	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Empty [0]	Height[0-255]

The first byte in the data field will be empty while the second byte will contain the target height for the torso. The height is limited to between 0-255 and corresponds to the unit [cm].

8.1.1.2 Open/Close Gripper

Description:

By sending the open/close command, the gripper will either close or open depending on the value in the data field. The close command will drive the motor in the corresponding direction until it reach the torque limit. The opening command will drive the motor until it reaches the torque limit.

Message:

0x3A	0x9B	0x02	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Open [0-1]	Close [0-1]

The valid combinations for the data field are:

Open command:

Data byte 1	Data byte 2
1	0

Close command:

Data byte 1	Data byte 2
0	1

8.1.1.3 Init torso

Description:

The Init command will begin to calibrate the torso height by driving it down to its lowest point it can be in, after that, it will set the torso at a specified position. The gripper will at the same time begin to calibrate itself by extend the gripper to its maximum. When the torso has reached its lowest point it will drive the torso to the target height that was specified within the data filed.

After both the gripper and the torso have been calibrated successfully it will reply with an "Init acknowledgement" message to the CAN-translator card.

Message:

0x3A	0x9B	0x03	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Empty [0]	Torso target height [0-255]

The first byte will be empty. The second byte will describe the torso target height, which will be the height that the torso will be driven to. The valid data range is between **0-255** (dec).

8.1.1.4 Stop (torso, gripper)

Description:

The stop commando will stop either the torso and/or the gripper depending on data in the data field. After a stop in either torso or gripper motor, a "stop" acknowledgement will be replied to the CAN-translator card.

Message:

0x3A	0x9A	0x04	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Stop torso [0-1]	Stop gripper [0-1]

The valid combinations for the data field are:

Stop torso command:

Data byte 1	Data byte 2
1	0

Stop gripper command:

Data byte 1	Data byte 2

0	1

Stop torso and gripper

Data byte 1	Data byte 2
1	1

8.1.1.8 Write register motor-controller torso

Description:

With the “Write register motor controller torso” command, the motor controller that are connected to the torso motor can be configured and written to.

Message:

0x3A	0x9B	0x15	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Register address and data byte 1	Register address and data byte 2

The data field will contain the register address and on the motor controller that the function and the data shall be written into. Both address and data are 16-bit, which means it is split up into 2 bytes when it is sent from Simulink down to the CAN-translator.

8.1.1.9 Write register motor-controller gripper

Description:

With the “Write register motor controller gripper” command, the motor controller that are connected to the gripper motor can be configured and written to.

Message:

0x3A	0x9B	0x16	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Register address and data byte 1	Register address and data byte 2

The data field will contain the register address and on the motor controller that the function and the data shall be written into. Both address and data are 16-bit, which means it is split up into 2 bytes when it is sent from Simulink down to the CAN-translator.

8.2.1 Description of **read** commands

8.2.1.1 Read register motor-controller torso

Description:

With the “Read register motor controller torso”, all registers on the motor controller that are connected to the torso motor can be read.

Message:

0x3A	0x9B	0x1F	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Register address and data byte 1	Register address and data byte 2

The data field will contain the register address and on the motor controller that the function and the data shall be read. Both address and data are 16-bit, which means it is split up into 2 bytes when it is sent from Simulink down to the CAN-translator and vice versa.

8.2.1.2 Read register motor-controller 2

Description:

With the “Read register motor controller gripper”, all registers on the motor controller that are connected to the gripper motor can be read.

Message:

0x3A	0x9B	0x20	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Register address and data byte 1	Register address and data byte 2

The data field will contain the register address and on the motor controller that the function and the data shall be read. Both address and data are 16-bit, which means it is split up into 2 bytes when it is sent from Simulink down to the CAN-translator and vice versa.

8.2.1.3 Read torso height

Description:

The read torso height command will read the current height of the torso.

Message:

0x3A	0x9B	0x29	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Empty [0]	Torso height [0-255]

Only the second byte in the data filed will be non-empty. The torso height will be represented in cm in the range between 0-255.

8.2.1.4 Init acknowledgement

Description:

The initialization done command will be replied when the initialization of both the gripper and the torso is done.

Message:

0x3A	0x9B	0x33	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Empty [0]	Empty [0]

Data filed will be empty when the message is replied.

8.2.1.5 General error

Description:

The Error message is a general message that can contain various errors from the torso node. This error is automatically sent to the CAN-translator when an error is encountered on a node.
(The error code specified can be expanded as the project grows and if an indication of that more error codes needs to be introduced).

Message:

0x3A	0x9A	0x34	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Node ID [5]	Error code [0-255]

The first byte in the data field describes that the error comes from the torso node. The second byte describe which kind of error the node encountered. (The error code specification will be developed as the project goes on).

8.2.1.6 Torque error

Description:

The torque error is sent automatically up to the CAN-translator when a torque error has been generated on a motor controller on the torso node.

Message:

0x3A	0x9A	0x35	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Node ID [5]	Motor that generated torque error [1-2]

The first data byte describes that the torque error comes from the torso node. The second byte will describe which of the motors that generated the torque error.

The valid values the **data byte 2** can have:

1: Indicates that the torso motor stalled

2: Indicates that the gripper motor stalled

8.2.1.7 Stop acknowledgement

Description:

The Stop ACK is replied to Simulink when the motors have stopped as an action of a stop command.

Message:

0x3A	0x9A	0x36	DATA	CHECKSUM	0x0D
1	2	3	4-5	6	7

Data field:

Data byte 1	Data byte 2
Node ID [5]	Motor that have stopped [1-3]

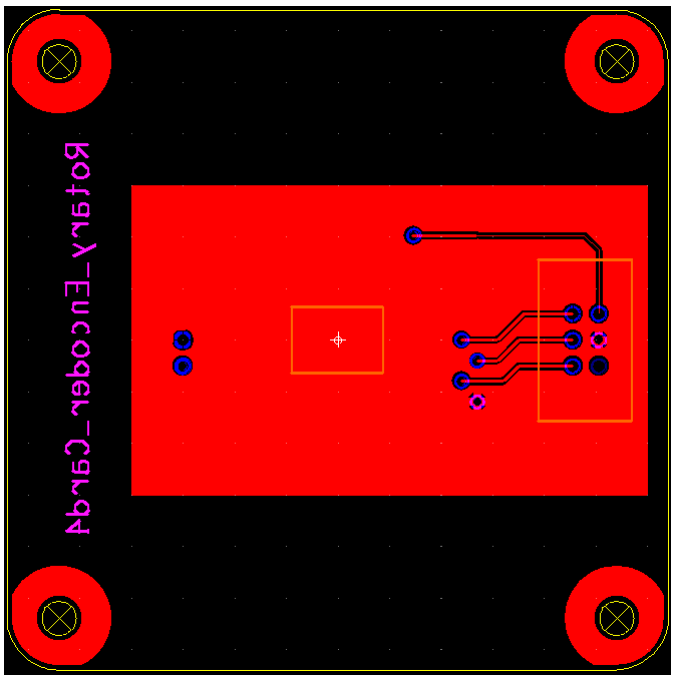


Figure 72: Bottom Layer