Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Dynamic Systems and Control
Institut für Dynamische Systeme und Regelungstechnik

Hans Muster

FUCK DSPACE

How a company stole our money

Bachelor's Thesis

Institute for Dynamic Systems and Control
Swiss Federal Institute of Technology (ETH) Zurich

Supervision

First Supervisor
Prof. Dr. Second Supervisor

December 2016

IDSC-XX-YY-ZZ

Abstract

Abstract goes here

Keywords: Drive-By-Wire, CANOpen.

Contents

Chapter 1

Introduction

The purpose of this work was to design a drive-by-wire system for future use in the field of autonomous self driving cars. The idea of the overall project is to have a fleet of autonomous vehicles, mainly to test fleet management algorithms.
To realize this, an electric kart will be modified such that all mechanical inputs normally made by a human driving the cart, can be made by electric actuators. A kart's size and accessibility to it's components make it easy to modify. This go-kart prototype was made as a proof of concept and for early testing of the software and hardware.
Our objective was to plan and realise a solution for every critical component, i.e. throttle, brake and steering. In order to test A.I. and the fleet management algorithms, a stable and reliable low level platform needs to be in place. Requirements for the individual components in the drive-by-wire system were precision, speed, reliable communication and flexibility. Planing included steps such as finding the parts, how to build it into the kart, communicating with the part and supplying the right amount of power.

Designing a drive-by-wire system poses many challenges, which can be approached in many different ways. Most drive-by-wire vehicles are being conceptualised with safety as their highest priority. This is because those vehicles transport passengers and their well-being is of utmost importance. That is why these system contain many fallback levels, which are rather difficult to realise. Our kart does not contain such fallback levels, simply because its self-driving capabilities are only used on racing tracks and the drive-by-wire system will not be active when a human is behind the wheel.

In chapter 2 and 3 we will give an overview of the hardware and software used in this project. Chapter 4 will focus on the implementation of these parts, i.e. where to install them, challenges and solutions, while chapter 5 covers all aspects of communicating and controlling the devices.
We will sum up our results in chapter 6. Finally, we will draw our conclusion of this project in chapter 7 and propose future work that can be done on the kart.

Chapter 2

Overview Hardware

2.1 Components

In the following section, we will shortly describe the relevant parts already built into the kart, as well as all other components we installed. We will also explain their importance and why we decided for these compontents.

2.1.1 Built into Go-Kart

The go-kart used for this project was a SinusiON, an electric kart manufactured by Rimo Germany. An electric kart offers an easier implementation of a throttle-by-wire system over more common petrol-fueled kart, as the basis for such a system is already in place. It's weight prior to modifying it was roughly 170 kg, making it much heavier than a petrol-fueled kart. The dimensions (l/w/h) are 2020mm/1390mm/600mm. For any specifications or parts not listed in this chapter, we refer to the RIMO manuals.

ACD 4805 Motor controller

Each of the electric motors is controlled by an ACD 4805 motor controller, mounted on top of the motor. The ACD communicates via CANOpen and allows for easy modification of parameters and performance curves. The motor controller was likely the most important in-built part, as the whole throttle-by-wire system was based on it.

Brake

Rimo offers a dual-circuit hydraulic disc brake system. A simple lever arm connects the brake cylinder to the brake pedal. This configuration requires a precise actuator, as the way difference between a mild and hard brake was minimal.

Steering

The go-kart's steering mechanism was realized with a bell-crank linkage. This setup resulted in a non-linear steering behaviour, which needed to be accounted for when configuring the power steering. The steering shaft was short and it's surroundings offered limited space, therefore the required steering servo needed to be small as well.

Battery

The main battery consists of 16 x 3.2 V LiFeMnPO4 cells and offers 100 Ah of battery charge. The nominal on-board voltage is 48 V. Because of the battery's high capacity, it makes a separate power supply for most of the additional electric actuators redundant. Only the power steering required 12 V and 90 A peak current. Because most converters do not support such a high peak current and the idea of capacitor seemed impractical, a small 12 V battery with a capacity of 3 Ah had to be installed in the front of the car.

Motor

The kart features two "PMS 100 R" 2.8 kW double-sided synchronous motors, each controlled by one of the two ACD 4805 motor controllers. The motors also offered regenerative braking, offering longer driving time and assisting in braking the car.

2.1.2 LinMot

In order to actuate the brakes a linear motor was used. The characteristics of being fast and precised make the motor suitable for various brake maneuvers. The motor is an electromagnetic drive in tube-form.

The linear motion simulating a mechanical brake can be generated electrical without any form of mechanical interconnection between motor and motion. This results in a a very compact device.

The linear motor is composed by two parts: the stator and the runner. The runner consists of a series of neodym-magnets, placed in steel tube. In the stator, the winding as well as the bearing for the runner, position detection and supervision of the motor fit in.

The internal position sensors can dynamically transmit a position signal. Therefore position can be controlled real time. This guarantees a high level of safety and flexibility.

In order to communicate with the motor, an LinMot motor-drive is employed. Target position, maximum velocity and acceleration can steadily be adjusted.

The operating temperature of The motor ranges from -10 ° C up to 110 ° C. After reaching 130 ° C the motor sends an error and upon reaching 140 ° C the motor goes into critical error state.

2.1.3 Power steering

Thyssenkrupp Presta offered a compact and powerful solution for a power steering unit. The 3 Nm of rated torque was more than sufficient for steering the kart. The unit communicates via CANOpen, with TKP offering a Simulink blockset for easy implementation.

2.1.4 DC-DC converter

In order to provide 24 VDC for powering the Microautobox and the linear motor drive, an SD-50C-24 DC-DC converter was used.

2.1.5 Microautobox (MABX)

The power steering manufacturer implemented a blackbox model of their unit in Matlab's Simulink. Their model was programmed to only run on dSpace's Microautobox. The Microautobox is a real-time system, used for performing fast function prototyping.

If needed, the MABX can easily be connected to a more powerful computer via the in-built ethernet connector. The MABX communicates with the go-kart's components via CAN. In it's basic configuration, the system does not support CANOpen, therefore an additional Simulink blockset had to be bought. The scarce time during the project justified the purchase of MABX, otherwise a much cheaper option could have been acquired. A less expensive option obviously calls for more programming and offers less plug-and-play, which surely would have slowed down our progress significantly.

2.1.6 cases/cables/adapters

A wide variety of cables, adapters were used in order to ensure seamless integration of the components into our system. Most of the cables we prepared ourselves were used for power supply. For the CAN communication we used a preexisting solution, where no soldering was needed. A metal sheet case was bought and mounted in the back of the kart, where the DC-DC converter, the MABX and the Linmot motor drive were safely stored. A little metal box was mounted on the front of the car accomodating the battery for power steering as well as a switch to turn off the power.

To account for the peak currents of the power steering and the linear motor that sit at 90A and 25A correspondingly, additional fuses were installed.

Chapter 3

Overview Software

For this project, a variety of software was being used, namely dSpace Control Desk, LinMot Talk and Matlab Simulink. Without going into too much detail, we will elaborate on their use in this project and how they worked together.

Matlab is a numerical computing environment, with Simulink being a block diagram environment for multi domain simulation and model-based design.

dSpace Controldesk is a experiment software, used to develop and test operating ECUs. If offers data capture across different platforms and access to common bussystems, including CAN and CANOpen.

LinMot Talk is LinMot's own drive-configuration software, allowing for easy access to the drive and controlling of the linear motor. Changes to the parameters of the drive are recommended to be made with this software since it provides an intuitive user interface compared to the communication with SDO-Data via CAN.

When the system is online, the Microautobox will receive commands from a higher layer, such as a simple rc controller or an A.I. autonomously controlling the vehicle.

dSpace provided us with numerous Simulink blocksets, containing blocks of all necessary components, such as ADC, DAC and CAN. With these, a model can easily be formed and compiled. The compiler creates a C/C++ code, which then will be flashed onto the Microautobox. Controldesk can then be used to gain information about the state of the system's components and change values in real-time.

LinMot Talk was mainly used to find out the most efficient way to communicate with the linear motor and become familiar with it's characteristic. The software includes a configurable oscilloscope, where the variables and parameters of interest can be plotted. This allowed us to determine the optimal performance of the motor in the given circumstances. The software is also needed to flash the configurable firmware onto the linear motor's drive. This needed to be done once, as all of the parameters and commands can be changed afterwards via CANOpen.

To debug our programmed software and monitor the can bus, we used a PCAN USB adpater directly connected to the CAN-bus. Pcan view and Pcan stat were utilized to handle the bus setup and send messages. To monitor and record the Can-bus the software Busmaster was used.

Chapter 4

Communication

To handle the communication between all devices, a fast and easy to implement data exchange method was required. RIMO, as well as the other manufacturers of our components made use of the standardized industrial application CANOpen. Because CANOpen is based on CAN, we will first describe the CAN bus protocol.

4.1 CAN Basics

CAN stands for Control Area Network and consists of two main layers, namely the physical layer and the data link layer. CAN was developed in 1986 and is used for data exchange between different stations in a network, based on serial communication. Messages are received and transmitted via broadcasting, making every message available for all of the connected stations.

The rise in electronification in many parts of the industry called for simpler and more efficient means of communication. Extensive wiring still resulted in rather limited data exchange. A way out of this was presented by serial bit data exchange and connecting up all electronic control units to a single bus. Depending on the bus length, CAN offers up to 1 Mbit/s of data rate, while remaining robust and reliable, even in a noisy environment.

The physical layer consists of a twisted pair of wires, which can be shielded if required. The value of a bit was determined by the difference in voltage of the two wires. If the voltages are the same, the bit is recessive. If the difference is higher than 0.9 V, the bit is dominant. As both wires are affected by the same electromagnetic disturbances, their difference in voltage will not vary. The data link is therefore immune to electromagnetic disturbances. By twisting the wires, the magnetic field generated will also be reduced significantly.

To reduce reflections in the data cables with rates higher than 125kbit/s, it is recommended to terminate the ends of the communication lines with termination resistors with 120 Ohm.

Roughly speaking, a CAN message or frame is made up of an identifier(ID) and the data (up to 8 bytes). A complete CAN message looks like the following.

Name	**Length in bits**
SOF	1
ID	11
RTR	1
ID extension	1
Reserved bit	1
DLC	4
Data	0-64
CRC	15
CRC delimiter	1
ACK slot	1
ACK delimiter	1
EOF	7

SOF describes the start of the frame, followed by the identifier. The role of the identifier bit is important, as it handles bus management and represents the message priority. The remote transmit request (RTR) must be dominant for data frames and recessive for remote request frames. A remote frame is used to request data transmission from a node. The ID extension and reserved bit are dominant for base frames. The data length code (DLC) informs the receiver about the size of the data in bytes, followed by the actual data. The cyclic redundancy check is used for error detection, with the recessive CRC delimiter coming right after. The acknowledgement slot is used to confirm the message by the receiver by using a dominant bit and the transmitter sends a recessive bit. Again, a recessive delimiter ends the acknowledgement slot. The last 7 bits describe the end of the CAN frame.

4.2 CANOpen

CANOpen is a higher-layer protocol based on the aforementioned CAN. In the following, we will provide information on the project-relevant aspects of CANOpen. All real-time data is exchanged via process data objects (PDO). The adress of each individual PDO can be found in a standardized table by CAN in Automation (CiA). PDO's contain data from a single or different objects from the object dictionary. This maps each bit of the data section of a PDO to a certain object.

To change PDO mapping or customise other parameters stored in the object dictionary, service data objects (SDO) are used. These however do not transmit and receive real-time data, but are only used for service purposes. They usually contain the index, subindex and the new value of the respective object.

PDO as well as SDO communication is split into receive and transmit messages. For PDO communication the receive PDO contains the desired command for a node and the Transmit PDO is composed of the actual state or other valuable information measured on node itself.

In SDO communication a receive-SDO is either used to change a value in the object dictionary or to request the contents of an entry in the object dictionary. The receive-SDO is always answered by a corresponding transmit-PDO either acknoledging a change in the object dictionary or serving the requested data.

So for example, the mapping of the ACD's RPDO1 looks like the following.

Name	Index	Subindex	Bit
Command Word	0x2000	1	16
Command Speed	0x2000	2	16
Command Acceleration	0x2000	5	8
Command Deceleration	0x2000	6	8

Table 1, RPDO1 of the ACD 4805

So in order to set a desired speed on the right (Node 6) motor controller of around 900 rpm, the following message had to be sent.

CAN-ID	DATA					
0x206	09	00	84	03	00	00

Take note that to translate the desired target input into a bytewise message the little endian rule is most often utilized and the hex format is used to display the contents of each byte. According to the little endian rule the byte containing the higher bit values comes second in transmission. The Command speed of the message above is split into two hex numbers yielding the underlying number 384h.

Each CAN device has a personal node, ranging from 1 to 127. This ensures that every message reaches it's corresponding device.

There are various transmission types avaible in CANOpen for PDOs. Some are manufacturer specific, therefore only the types relevant for our project will shortly be described.

- Transmission type 1-240 (cyclic synchronous)
  - This transmission type relies on sync messages. After every nth sync messages, the PDO transmits it's data. So for example, if the transmission type is 6, the PDO will send after every 6th sync message.

    For the Linmot, we used the transmission type one. So in order to gain information about the motor, we had to send sync messages.
- Transmission type 254/255 (asynchronous)
  - These transmission types are event-triggered, where the event is manufacturer specific for 254 and for 255 defined in the CANOpen device profile.

    For the ACD 4805, tranmission type 254 was listed as unpack the data immediately. This was the setting used in the ACD and in the Mircoautobox.

    For the LinMot, transmission type 254 was event triggered, with an adjustable internal timer which triggers the event.

The standard table of CAN-Id's can be found below.

COB	**Function Code**	**Resulting CAN-ID**
NMT	0000_b	$0(000_h)$
CODE	0001_b	$128(080_h)$
TIME	0010_b	$256(100_h)$
EMCY	0001_b	$129(081_h)$– 255 ($0FF_h$)
PDO1(tx)	0011_b	$385(181_h)$ – 511 ($1FF_h$)
PDO1(rx)	0100_b	$513(201_h)$ – 639 ($27F_h$)
PDO2(tx)	0101_b	$641(281_h)$ – 767 ($2FF_h$)
PDO2(rx)	0110_b	$769(301_h)$ – 895 ($37F_h$)
PDO3(tx)	0111_b	$897(381_h)$ – 1023 ($3FF_h$)
PDO3(rx)	1000_b	$1025(401_h)$ – 1151 ($47F_h$)
PDO4(tx)	1001_b	$1153(481_h)$ – 1279 ($4FF_h$)
PDO4(rx)	1010_b	$1281(501_h)$ – 1407 ($57F_h$)
SDO(tx)	1011_b	$1409(581_h)$ – 1535 ($5FF_h$)
SDO(rx)	1100_b	$1537(601_h)$ – 1663 ($67F_h$)
NMT error control	1110_b	$1793(701_h)$ – 1919 ($77F_h$)

Table 2, COBs and there corresponding CAN-IDs

4.3 Matlab Model

In the following passage, some of the most important components of the matlab model will be explained. The matlab model is the underlying software basis for the C code running on the Microautobox handling the CAN bus and all communication on it.

The physical layer of CAN has been set up in the multimessage controllersetup block. The baudrate has been set to 250 Kb/s. The module together with the board number determine the pins for the CAN connection. Moreover, Various status messages of the CAN bus can be read out from this block.

The CANOpen general setup block generates a matlab block for each node on the CAN bus, sets the transmission type for each message and determines the sync interval for the network. The Sync interval has been set to 2ms and the transmission type has been defined as cyclic synchronous for all nodes. The heartbeat message and nodeguarding have been turned off in this first prototype.

The blocks generated by the general setup block allow for numerous inputs and outputs. Some of the most important functionalities are the following:

Sync Block The sync block generates the sync block according to the interval defined earlier. The input turns the sync message on or off.

Reset By feeding a rising edge into this input a node can be resetted.

NMT Timeout The time the nodes have to restart after a reset. If a node takes longer to restart than stated by this expression an error will occur.

SDO Trigger To send a SDO message a rising edge must be fed into this port.

SDO W and RXPDO Data The data for PDO respectively SDO transmission can be led into this input. The data is composed of vectors of data type uint8, each integer representing on byte.The data will be transmitted with the corrresponding identifier by default.

SDO R and TXPDO data The response to a SDO message and the cyclic transmission of the TXPDO can be read out here. The bytes of all TXPDO are encapsuled in one vector.

The following picture shows the multimessage controllersetup block and the node blocks created by the CANOpen general setup block.

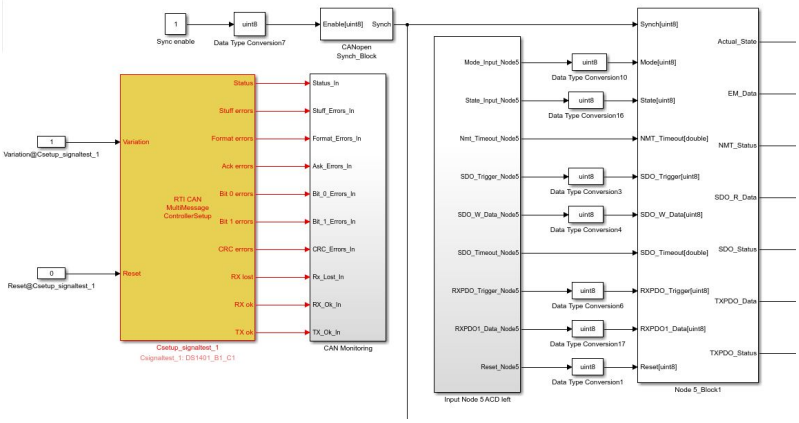

Figure 4.1

4.3.1 Analogue Digital Converter

To check the voltage of the 12 V battery mounted in the front, we used dSpace's ADC block. The module number 1 and the channel number 1 were used. Because the Microautbox's voltage input range is 0 - 5V, a voltage divider was used. The output of the ADC is between 0 and 1. The voltage input ranges from 0 to 4.3 V, therefore the maximum output of the ADC is around 0.86. So in order to get the proper scaling, the ADC output gain needs to be around 13.95.

An additional digital output and input (Microautobox) were utilized to turn on a warning lamp in case of critical error and display the state of the emergency switch.

Chapter 5

Implementation

5.1 Implementation Hardware

5.1.1 CAN Bus

The following figure shows a schematic of the implemented CAN bus.

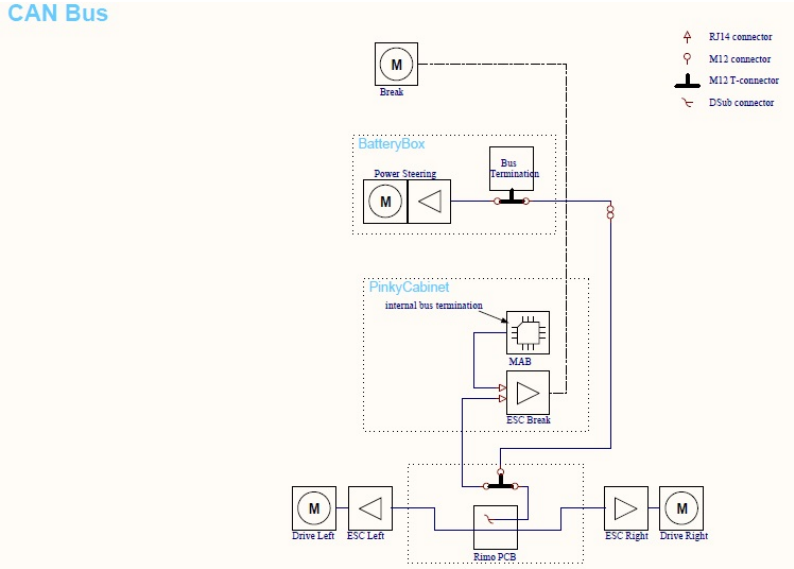

Figure 5.1

The CAN is realized by connecting all hardware nodes with a M12 cable. At both ends of the CAN bus, a 120Ω is connected to avoid unwanted reflections. The two ACDs are connected internally in Rimos circuit.

5.1.2 Braking

Braking presented itself to be the technically most difficult aspect in the drive-by-wire system. The kart offered limited space, therefore a very efficient actuator was needed to ensure the Force necessary for an emergency brake. Early testing showed that for a full brake roughly 250-350 N of

force was required. However, this testing was done with the kart jacked up, therefore we could not fully rely on these results.

Early on we set a few requirements for our brake actuator, namely speed, force and precision. After some research we decided to look further into a high end linear motor by LinMot and neglect the idea of a rotary motor. Our main reasons being that a rotary motor generally was significantly slower and usually required external sensors to be precise enough.
We also dismissed the intriguing concept of directly interfering with the hydraulic system, for example inserting an electro hydraulic pump into the system. Our main reasoning against this was that it would require a lot of time, effort and would most likely not outperfom a simpler mechanical brake actuator.

A very easy to implement and cost efficient method of braking that we proposed was plugging braking. The idea was to use the kart's electro motors to brake, by reversing the current. While it would not need any additional parts and would only require some programming, we soon realised the braking power generated by the hydraulic brake sufficed for a full brake and power consumption would rise significantly.
Because time was running low and we had to decide for a reliable solution, we chose a linear motor over the plugging braking option. If we had more time on our hands and could have done some testing of the required braking power, we might have decided otherwise.

With all other options out of our way, we were now able to focus on the linear motor. With our battery providing 48 VDC, LinMot's range of motor narrowed down quite a bit. The PS01-48x240F-C was our first idea, as it provided up to 572 N of maximum force. We quickly realized that our limited space would not allow for such a big motor. After a personal consultation with one of LinMot's employees, we decided for a PS01-37x120F-HP-C with a 300 mm runner.
Because of the smaller size, we now had various options of where to put the linear motor. Again, we strived for an easy installation. Therefore we determined the best location to be on top of the brake cylinder, just in between the two pedals. Pushing the pedal resulted in a slightly arc shaped downward motion of the braking lever. This is the point where the motor's runner has been attached to the lever. As the linear motor should not be under radial load, a design had to be realized which allowed the motor to tilt slightly. This helped to reduce the radial load, as the downward motion could be compensated with tilting the motor at an angle.

This notion was put into place by mounting the linear motor on a metal frame right above the braking cylinder. The frame is made of two sheets of metal connected by multiple steel linkages. The motor is held on the frame with two ball-bearings, allowing for a pitch motion and therefore preventing any major radial forces. The metal sheets were cut by a water jet cutter.

5.1.3 Throttle

To control the throttle, we had to access the ACD 4805 motor controller of each electric drive. After a simple start up sequence, we were able to control the velocity of each wheel via CANOpen. For throttling, no actuators had to implemented. The crucial work that went into throttle-by-wire part was mainly communication and will therefore be discussed in the following chapter.

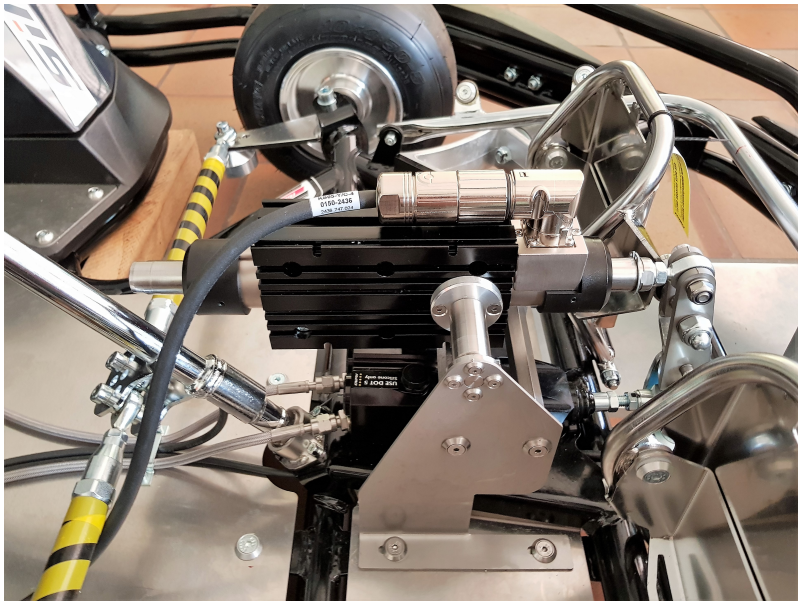

Figure 5.2

5.1.4 Steering

As already mentioned in chapter one, we used a compact power steering unit, with a rated torque of around 3 Nm. In order to install the unit, the kart's steering column was cut up, as was the power steering's drive shaft. The column and the drive shaft were then joined together by using press-fit joints. To counter the motor's torque, we fixed it to the steering wheel's metal holder and reinforced it with an additional steel linkage.

5.2 Implementation Communication

5.2.1 RPDO

The following section will take a closer look at the PDOs used to control the actuators in the CAN network.

Table 1 in chapter 4.2 shows the only RPDO in use for the two ACD motor controllers. The command word filled into the first two bytes is built up by 16 bits. Depending on the bits turned on, the state of the ACD is changed.

Figure 5.3

Bit	Description
0	Switch on (close main contactor if used)
1	Speed neutral brake ramp parameter set LSB
2	Speed neutral brake ramp parameter set MSB
3	Enable power stage
4	Speed PI controller parameter set LSB
5	Activate Torgue current boost
6	Open drain output 1
7	Open drain output 2
8	Capacitor discharge on
9	(speed) Rollback and anti clonk disable
10	Activate one quadrant
11	Speed ramp parameter set LSB
12	Speed ramp parameter set MSB
13	Foot brake on
14	Magnetisation current low on
15	BDI magnetisation current pulse enable

Table 3, Bits in RPDO1 of ACD 4805

To set the ACDs to operational mode, bit 0 and bit 3 are turned on. To set a target speed, the desired velocity is simply filled into the next two bytes. The bytes for acceleration are left empty since they are not in use for this motion command.

Linmots two RPDOs look as follows.

Name	Bit
Control Word	16
Motion Wommand Header	16
Target Position	16
Maximal Velocity	16

Table 4, RPDO1 of Linmot

Name	Bit
Acceleration	16
Deceleration	16
Empty	16
Empty	16

Table 5, RPDO2 of Linmot

Similar to The ACDs RPDO, the control word is made up of numerous bits defining the state.

Bit	Description
0	Switch On (close main contactor if used)
1	Voltage Enable
2	Quick Stop
3	Enable Operation
4	Abort
5	Freeze
6	Go To Position
7	Error Acknoledge
8	Jog Move +
9	Jog Move -
10	Special Mode
11	Home
12	Clearance Check
13	Go To Initial Position
14	Reserved
15	Phase Search

Table 6, Bits in control word of Linmot

To set the drive to operational mode, bits zero to five ar turned on. To home the drive, bit 11 is turned on additionally. To acknowledge an error, bit seven is turned on instead. To avoid errors, all bytes except the two corresponding to the command word send zeroes until the drive is homed. Following, a motion command can be transmitted by changing the subsequent bytes.

Bits	Description
0 - 3	Command Count
4 - 7	Sub ID
8 - 15	Master ID

Table 7, Bits in command header of Linmot

Master and Sub ID define the motion command used. The command in use for the brake is "VAI 16 bit go to pos". Bit 8 and 11 are turned on for this command. Each time a new command should be sent, at least one bit in the command count must change. For the ACDs and Linmot analogously, the bytes transmitting speed and likewise position and acceleration in linmots RPDO set the reference target for the internal controllers. controller parameters must be changed seperately via SDO communication. Further information about the objects used in PDO communication can be found in the object dictionaries.

5.2.2 ACD

In order to communicate with the ACD, the CAN Node-Id of each ACD had to be determined. One of Rimo's pdf files, namely "Setting up ACD Controller & Connection Diagram", serves exactly

this purpose. To find the respective node-id's, we had to check if either pin 12 (DI5) or pin 20 (DI6) was connected to any other pins on the ACD 4805 K1 connector. In our case, pin 12 of the right ACD was connected to pin 1, making it node 6. Pin 12 of the left ACD was not connected and therefore making it node 5. Afterwards our findings were confirmed when we connected the go-kart to our CAN bus. With the help of a CAN-USB adapter, we were able to receive and send messages, and after a while controlling the kart. Because the go-kart does not communicate via CAN by default but only for service and remote control purposes, we had to put all nodes into operational mode by sending a NMT start messages to all nodes, namely node 5 and 6. The ID of a NMT message is 0x000, to start the nodes the instruction code 0x01 had to be used. To reach all nodes simultaneously, the node adress needed to be 0x0.

Issues with CANOpen and dSpace.

When we tried to control the ACD via CANOpen with the Microautobox, the communication presented itself to be an issue. The wheels were not turning smoothly, but rather interrupted from time to time. We assumed that something had to be disrupting the communication. With the help of a CAN-USB adapter, we were able to monitor the CAN bus. After checking all physical layers, we came to the conclusion that it must be a software problem. Our first approach was to change synchronisation times. This however did not have any effect, neither did changing the baud rate or adjusting the step size for Matlab's solver. A smooth motion of the motors could be realized by sending the Receive PDO manually via the PCAN software. The problem turned out to arise from the second Receive PDO. While the first RPDO determines the target speed for the speed controller the second RPDO provides the possibility to control the motor with via torque control. By default, the use of one of the mentioned does not disable the function of the other. This leads to critical failure causing improper motion if both RPDOs are sent simultaneously. The motor tries to achieve both, target speed and target torgue ending up rapidly switching the motor current. The problem could be solved by completely removing the second RPDO from the Matlab model. We were not able to deactivate an individual PDO's transmission in real time with dSpace CANOpen solution.

Two's complement is convenient way to store integers, such that adding and subtracting with negative numbers becomes very easy. This was used on the ACD 4805, where certain values, such as the rotational speed of the wheels, can be negative. The basic principles of two's complement are the following.

- Zero is represented by all 0's. e.g. 0 0 0 0 = 0

- The maximum positive integer is $2^{numberofbits-1}-1$. So for 4 bits, the biggest integer is therefore 0 1 1 1 = 7, and not 1 1 1 1 = 15 as in the standard notation.

- if the integer is negative, 1's and 0's switch roles, starting from all one's, e.g. 1 1 1 1 = -1. This increases the range for negative numbers by one.

So for 4 bits it looks as follows.

0 0 0 0 = 0	0 1 0 0 = 4	1 1 1 1 = -1	1 0 1 1 = -5
0 0 0 1 = 1	0 1 0 1 = 5	1 1 1 0 = -2	1 0 1 0 = -6
0 0 1 0 = 2	0 1 1 0 = 6	1 1 0 1 = -3	1 0 0 1 = -7
0 0 1 1 = 3	0 1 1 1 = 7	1 1 0 0 = -4	1 0 0 0 = -8

So an easy way to find the negative integer of a positive integer is to convert the desired decimal number to binary, inverting all 0's and 1's and then adding 1. A more hands on approach is to again convert to binary, starting from the right to find the first 1 and inverting all bits to the left of it.

So in order to have a rotational speed of -500 revolutions per minute, the following steps have to be taken for a signed 16 bit integer.

500 = 0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0

Now invert all bits.

1 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1

And add 1

1 1 1 1 1 1 1 0 0 0 0 0 1 1 0 0 = -500

The second method would result in the following steps.

Starting from the right, find first 1.

500 = 0 0 0 0 0 0 0 1 1 1 1 1 0 **1** 0 0

Invert all consecutive bits to the left of it.

1 1 1 1 1 1 1 0 0 0 0 0 1 **1** 0 0 = -500

If a signed integer is postive or negative is easy to spot, as it's most significant bit determines if the number is negative or positive. 1 = negative, 0 = positive.

5.2.3 LinMot

important settings

For some reason, TPDO3 and TPDO4 would not transmit their data when a sync message was sent. This was tried to solve by setting the intern event timer to around 10 ms and changing the transmission type to 254. However, after a while it would stop transmitting out of a unknown reason. As time was scarce, we simply remapped the PDOs, such that the needed data would be transmitted via TPDO2.

Another issues presented itself while working on the brake. Previously we used a motion command called VAI go to position 16bit, which takes velocity, acceleration/deceleration and position as input and creates a curve for the linear motor, which results in the motion. However, a new command will only be executed if the last four bits of the motion command header, called command count, has changed. In the easiest way, bit 0 can be toggled. To avoid this, a different setting could be utilized, called PV Stream. This uses a constant stream of position and velocity inputs during a fixed streaming period, interpolates and executes the command. While this seemed very intriguing, it's implementation was not possible. For some reason an error arose, stating that our streaming was too slow. Even after checking the period time with PCAN and checking all settings, the issue could not be resolved. After that, we went back to the prior way of setting the position.

To counteract the need for change in the command count in order for a change in motion we introduced a pulse generator into the Matlab model. The pulse generators output is an alternating signal either being zero or one. With the signal from the pulse generator added to the command header, the command count changes every halve period of the pulse generator. With a sufficiently low pulse generator period a function similar to the PV stream could be implemented.

5.2.4 Power steering

As of now, the needed simulink blocks for the power steering have not yet arrived. Communication is therefore not yet possible with the power steering unit and will not covered in this section.

Chapter 6

Results and Discussion

6.1 Testing

6.1.1 Linear motor

To find the optimal parameters of the linear motor, we used a simple spring setup and the LinMot Talk software. With the software, we were able to adjust any motor parameters in real-time, have it perform different moves at various speeds and accelerations.
On the spring gauge, which was clamped in between the linear motor and the beam, we could read off the force and compare it to the calculated force in LinMot Talk. We found out that the maximum stroke was mostly limited by the maximum velocity. The maximum required stroke of around 55 mm can be reached with a maximum velocity of around 1.5 m/s and maximum acceleration.

A maximum velocity higher than that leads to the motor shutting down, which is something we certainly wanted to avoid.

The temperature of the drive rises to critically high temperatures only after maintaining a considerably high force of above 200 N. Since such a force is only required for a very hard brake it should be possible to prevent longer periods of braking with more than 200 N. The following graphs show the winding-temperature change of the motor for different forces exerted on the drive. Note that the drive overheats in the emergency brake simulation only after x seconds which is much longer than the average duration of an emergency brake.

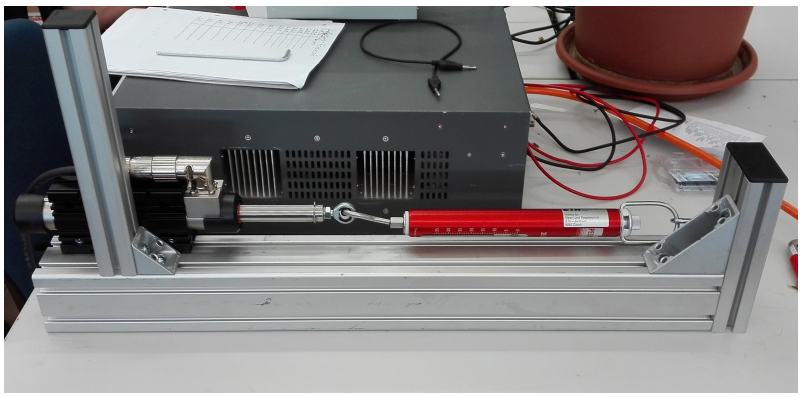

Figure 6.1

6.1.2 Brake

A first proof of concept for the brake was made with the brake mounted on the kart and a driver actuating the brake maneuver with a button. The brake was supplied with power from the kart battery.

For this test, the analog inputs of the Linmot motor driver were used to trigger a brake maneuver previously setup on the Linmot driver. A range of braking curves were tested, from a slight brake to an emergency brake. The speed of the kart during the brakes was around 50 km/h.

In a first test without actuating the brakes at all it was noted that the kart stops considerably fast only with regenerative braking and friction acting on the kart. The first, shorter braking strokes resulted in almost negligible difference to the test without braking. However, the emergency brake resulted in locked wheels.

This proofed that a maximal braking force could be achieved and showed that the linear motor can be used while on the kart. The test was performed without external power supplies and the only input into the Linmot driver being the analog input from the braking button. During the test while trying to execute an emergency brake, one of the fuses blew. The fuses in the kart were rated up to 10 A while the linear motor's peak current however is around 25 A. The fuse was subsequently replaced with a suitable new one.

It shall be noted that this test was performed with a driver in the kart at all time while the final kart will be driving without passenger and therefore be significantly lighter. This leads to a reduced requirement of force for an emergency brake because locking of the wheels will take place earlier.

In subsequent test we plotted the motors actual position, force and winding temperature with LinMot Talk's oscilloscope function. The test was done once for a hard brake and once for a soft brake. It is important to note that all data is merely calculated by the LinMot software, and thus does not reflect the actual values perfectly. However it is precise enough to analyse the motor's performance.

In **??** the oscilloscope recorded 60 seconds of data. This test was done to have a better understanding of how the motor overheats. At around 6 seconds, the motor reaches its demand position of -36 mm. At around the same time, the two windings of the motor start to heat up. The curve can more or less be assumed to be linear. Even after 54 seconds of operational time, the motor does not overheat and winding one still has around 15 ° C left until it reaches its error temperature of 150° C.

In **??** the oscilloscope was used to record when the motor overheats for a hard brake and how exactly it reacts. For this a demand position of -54 mm was given, however the position will not be reached even with an integrator. This is because the motor's maximum force of around 248 N is reached and the slider can not extend any further. In this plot it is apparent that winding one overheats much quicker than winding two. As soon as 150° C is reached, the motor shuts down and the actual position, as well as the force return to zero. In this state the motor is in an error state and no further commands can be executed. The error needs to be acknowledged and the switch on bit has to be reset. After this procedure, the motor is back in the operational state.

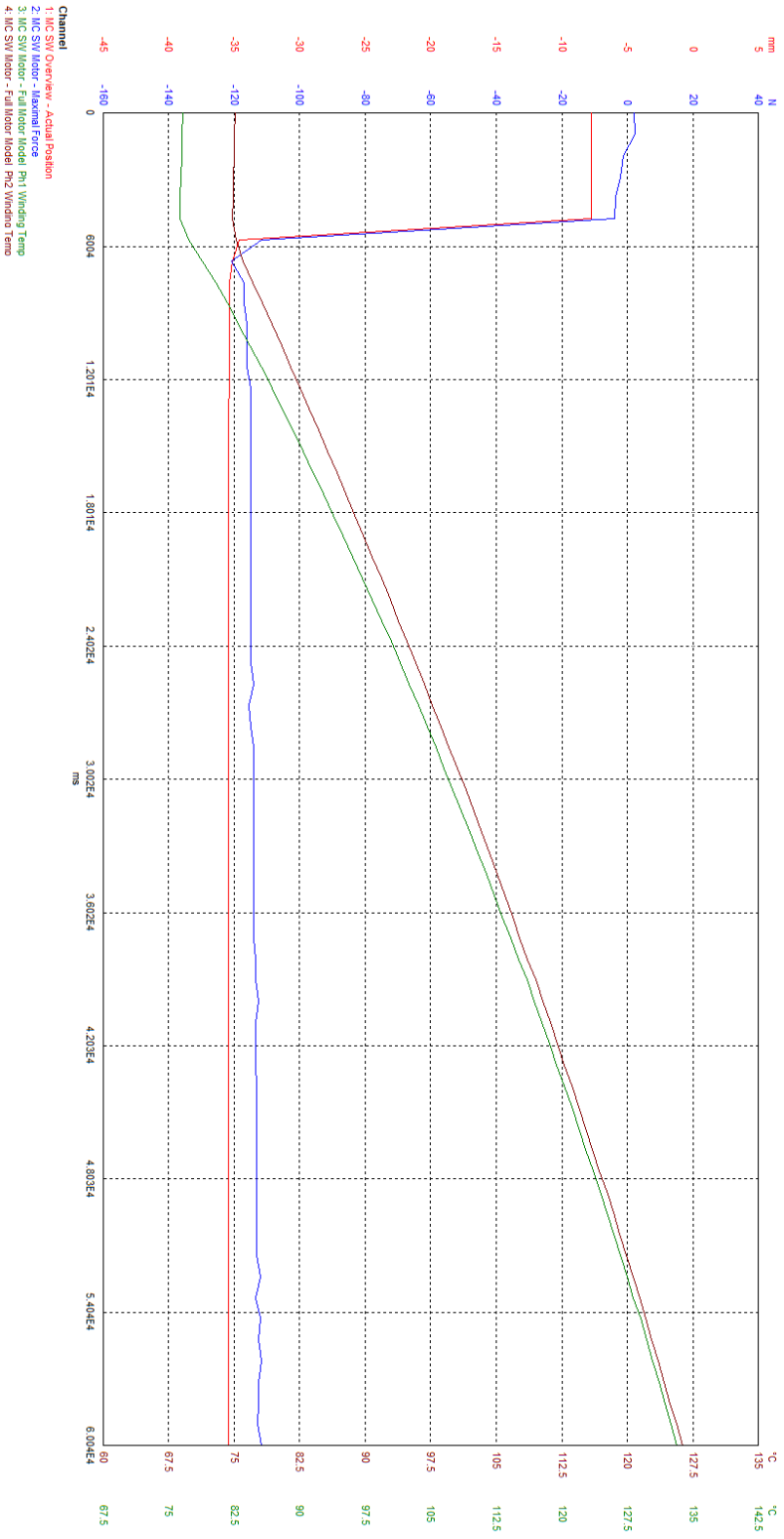
Channel
1: MC SW Overview - Actual Position
2: MC SW Motor - Maximal Force
3: MC SW Motor - Ful Motor Model Ph1 Winding Temp
4: MC SW Motor - Ful Motor Model Ph2 Windino Temp
mm
N
°C
°C
ms
0
6004
1.201E4
1.801E4
2.402E4
3.002E4
3.602E4
4.203E4
4.803E4
5.404E4
6.004E4

Figure 6.2

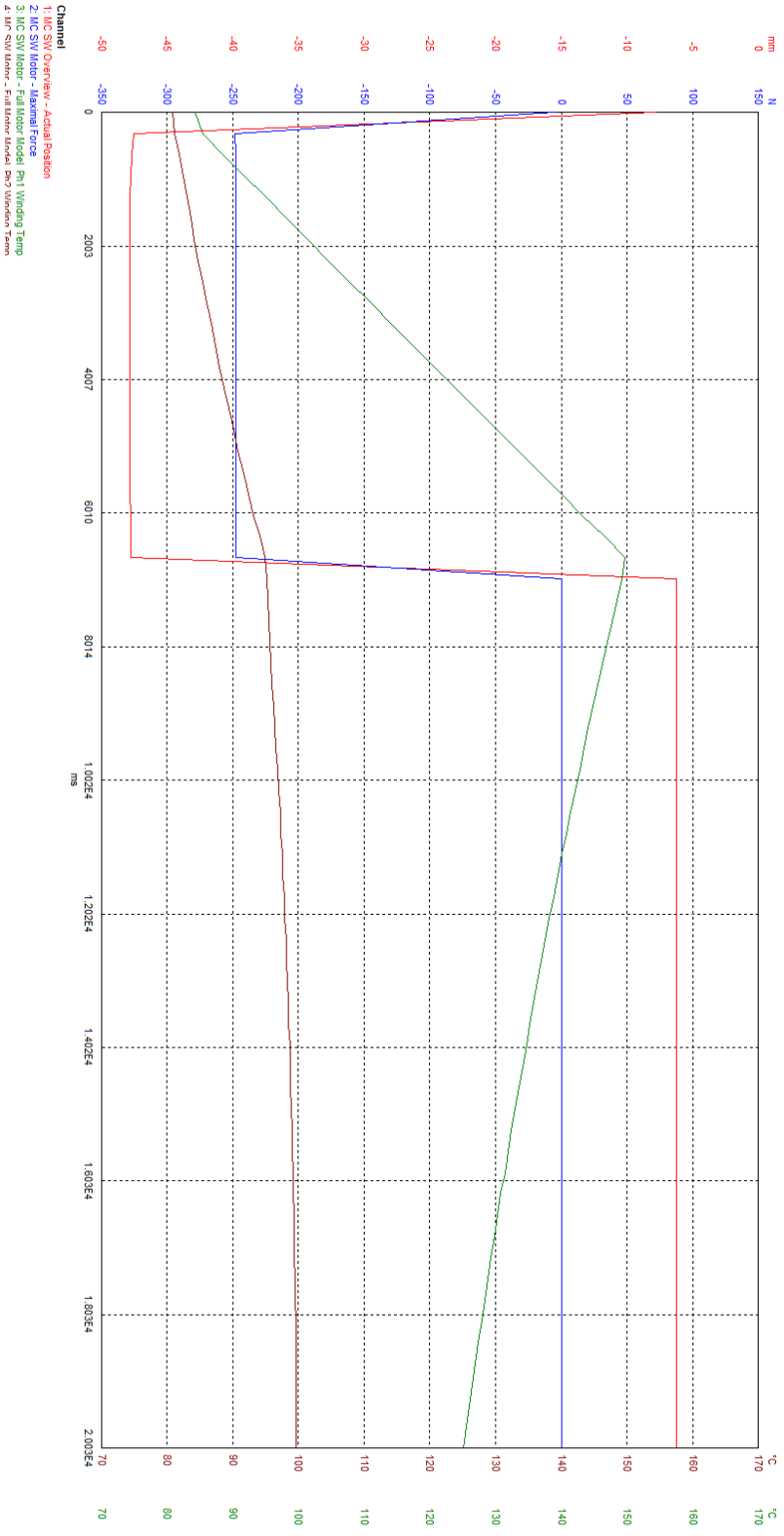
Channel
1: MC SW Overview - Actual Position
2: MC SW Motor - Maximal Force
3: MC SW Motor - Full Motor Model: Ph1 Winding Temp
4: MC SW Motor - Full Motor Model: Ph2 Winding Temp
mm
N
°C
°C
ms
0
2003
4007
6010
8014
1.002E4
1.202E4
1.402E4
1.603E4
1.803E4
2.003E4

Figure 6.3

6.1.3 Throttle

To confirm the throttle's functionality, we plotted the kart's wheels actual speed, which is transmitted in PDO1(tx).

Chapter 7

Conclusion

In conclusion, the go-kart fulfils it's requirements. With more time on our hands, we could have planned the project better. There were many unforeseen expenses, which mainly resulted from compatibility issues. Especially dSpace's CANOpen Master Solution led to a lot of issues. While it provided a somehow intuitive interface and helped in creating the needed blocks, it also lacked customisability. Many processes were somehow a black box model. This made debugging and understanding the errors very challenging or even impossible. The lack of documentation and support did not help either.

Our first proposal in regards to future work is to replace the microautobox with a cheaper microcontroller and can controller. While the microautobox provides many advantages, especially in regards to prototyping, it surely does not justify its cost.

Chapter 8

Appendix

The following code is the definition of the bibliography entry of the document class IDSCreport [1].

```
@manual{IDSCreportClass,
author = {Andreas Ritter and Philipp Elbert and Christopher Onder},
title = {How to Use the {IDSCreport} {\LaTeX{}} Class},
language = {english},
howpublished = {Version 1.4.0},
organization = {Institute for Dynamic Systems and Control ({IDSC})},
address = {ETH Z\"{u}rich, Switzerland},
month = dec,
year = 2016
}
```

Bibliography

[1] A. Ritter, P. Elbert, and C. Onder, *How to Use the IDSCreport LaTeX Class*, Version 1.4.0, Institute for Dynamic Systems and Control (IDSC), ETH Zürich, Switzerland, Dec. 2016.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Dynamic Systems and Control

Prof. Dr. R. D'Andrea, Prof. Dr. E. Frazzoli, Prof. Dr. C. Onder, Prof. Dr. M. Zeilinger

Title of work:

FUCK DSPACE

How a company stole our money

Thesis type and date:

Bachelor's Thesis, December 2016

Supervision:

First Supervisor
Prof. Dr. Second Supervisor

Student:

Name: Hans Muster
E-mail: muster@student.ethz.ch
Legi-Nr.: ??-???-???
Semester: ?

Statement regarding plagiarism:

By signing this statement, I affirm that I have read and signed the Declaration of Originality, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Declaration of Originality:

`https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/declaration-originality.pdf`

Zurich, 3.7.2017: _____