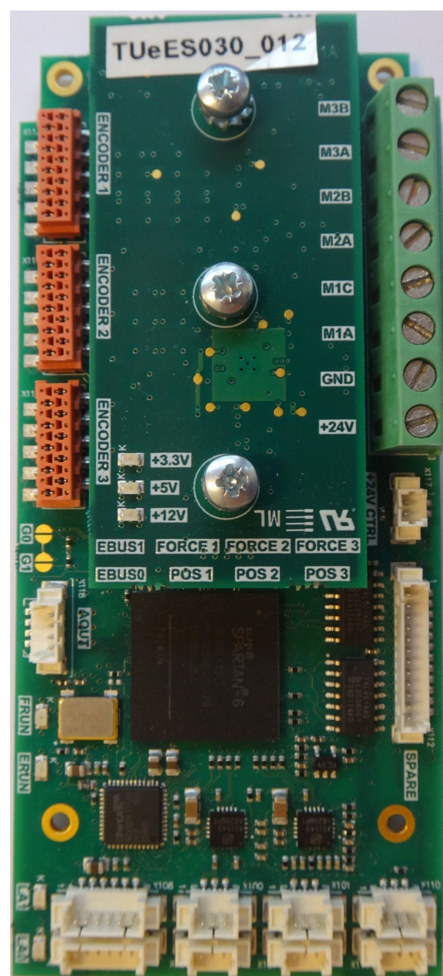
TUeES030 Manual

CST 2016.141

Alex Andriën
Ruud van den Bogaert

May 24, 2016
Control Systems Technology



With software by Arthur Ketels of Speciaal Machinefabriek Ketels

Robotics Lab
Department of Mechanical Engineering
Eindhoven University of Technology
Building Gemini-North, Groene Loper
5600 MB Eindhoven, The Netherlands

Version	**Date**	**Remarks**
1.0	17-05-2016	First Version

Table 1: Version History

Contents

1 Introduction **3**

2 System Description **5**
2.1 Power Supply . . . 5
2.2 Encoder inputs . . . 5
2.3 H-bridges . . . 5
2.4 Analog Outputs . . . 6
2.5 Analog Inputs . . . 6
2.6 Digital Inputs/Outputs . . . 6
2.7 eBus connector . . . 6
2.8 Clock . . . 6
2.9 SPI Flash & LEDs . . . 6
2.10 Current Measurement . . . 6
2.11 FPGA . . . 7

3 Field Programmable Gate Array **9**
3.1 Constraint File . . . 10
3.2 OpenBus System Document . . . 11
3.3 Schematic . . . 12
3.4 Verilog Code . . . 13
3.4.1 Basics . . . 13
3.4.2 `ECATtimer.V` . . . 15
3.4.3 `PWM2ph3.V` . . . 16
3.4.4 `AS1542_adc.V` . . . 19
3.4.5 `AD5722_dac.V` . . . 19
3.4.6 `quaddecoder.V` . . . 19
3.4.7 `swap32.V` . . . 19

4 Micro Processor **21**
4.1 Main Code . . . 21
4.1.1 Initialisation . . . 22
4.1.2 Update . . . 23
4.2 Interrupts . . . 25
4.2.1 PWM handler . . . 26
4.2.2 Encoder Handler . . . 27
4.3 Circular Buffer . . . 29
4.4 Data Structures . . . 30
4.4.1 Normal Mode . . . 30
4.4.2 FRF Mode . . . 31
4.4.3 Parameters . . . 32
4.5 Miscellaneous . . . 33
4.5.1 Swap . . . 33
4.5.2 Fixed Point Multiplication . . . 33

4.5.3 Clip . . . 33

5 Simulink Implementation **35**
5.1 Normal Mode . . . 35
5.1.1 Simulink Block . . . 35
5.1.2 S-function . . . 35
5.1.3 Read outputs . . . 38
5.1.4 Write inputs . . . 40
5.2 FRF Mode . . . 41
5.2.1 Simulink Block . . . 41
5.2.2 S-function . . . 42
5.2.3 Read outputs . . . 45
5.2.4 Write inputs . . . 46
5.3 Set mode . . . 47
5.4 Set parameters . . . 49

6 Experimental Results **51**
6.1 Modelling . . . 51
6.2 Frequency domain analysis . . . 52
6.2.1 Measurements . . . 53
6.3 Time domain analysis . . . 56
6.3.1 Feedforward parameters . . . 58

7 Manuals **61**
7.1 SOEM for Windows . . . 61
7.2 TwinCAT . . . 65

A Schematic of the electronic devices on the TUeES030 board **69**

Listings

3.1 ECATtimer.V 13
3.2 Blocking vs non-blocking example, source: https://nl.wikipedia.org/wiki/Verilog 14
4.1 int32_t main(void) 21
4.2 Initialisations 22
4.3 handle_RXPDO 23
4.4 update_actuators 24
4.5 handle_TXPDO 25
4.6 i_handler_pwm 26
4.7 currentPI 27
4.8 i_handler_ec 27
4.9 encoder_calculations 28
4.10 Circular Buffer implementation 29
4.11 Normal mode transmit PDO mapping 30
4.12 Normal mode receive PDO mapping 31
4.13 FRF mode transmit PDO mapping 31
4.14 FRF mode receive PDO mapping 32
4.15 Parameter structure 32
5.1 ec_TU_ES_030.c 37
5.2 ec_TU_ES_030_read_chan.c 38
5.3 ec_TU_ES_030_write_chan.c 40
5.4 ec_TU_ES_030FRF.c 42
5.5 ec_TU_ES_030FRF_read_chan.c 45
5.6 ec_TU_ES_030FRF_write_chan.c 46
5.7 ec_Set_TUES030 47
5.8 ec_Set_TUeES030params 49

Chapter 1

Introduction

In this report we present a description of the TUeES030 printed circuit board (PCB), shown on the front page of this report. It was developed by Neways Electronics International N.V. and (with the software described in this report) can be used to control up to three brushed DC motors. Apart from controlling the motors, the board contains six analog input and two analog output channels, digital input and output ports and eBus connectors for communication.

The target audience for this report is anyone who intends to use the board and who wants to know more about its inner workings. It is assumed that the reader has (at least some) knowledge of electronics, mechanics, programming and has access to a TUeES030 board and all the software.

In Figure 1.1 the hierarchy of the different parts of the TUeES030 is schematically represented. It starts with the user that uses Simulink to communicate with, which then communicates with the software on the micro-processor, this in turn communicates with the FPGA layer that finally controls the actual electronic hardware. The report starts at the lowest level with a hardware description in Chapter 2 and moves up through the FPGA and Micro-Processor layers in Chapters 3 and 4, respectively. The last layer is the Simulink software described in Chapter 5, after which some experimental results are discussed in Chapter 6.

Figure 1.1: Workflow of the TUeES030 software

Chapter 2

System Description

In Appendix A a block diagram of the TUeES030 board is shown, where the main electronic components are displayed but the smaller details are left out to conserve clarity. In the rest of this chapter these components will be discussed in more detail. For more information please contact Ruud van den Bogaert.

2.1 Power Supply

The board is powered using a 24V power source, connected via a PTR AK500 connector. This 24V power is filtered and converted to 12V via a low-dropout regulator (LDO), which is used to power the H-bridges. The filtered 24V signal is also converted to 5V using a DC/DC converter rated at 0.5A, which is subsequently converted to 2.5V, 1.2V and 3.3V of which the first two are used to power the FPGA and the last is used to power the external sensors and as a reference voltage for the analog-to-digital converters (ADCs).

There is no overvoltage protection implemented on the board itself. This needs to be taken into account when controlling motors, since back EMF caused by braking can damage the electronics. With the use of batteries as a source this is less of a problem, because they have the ability to sink the excess energy from braking. With most power supplies this is not the case. There are plans for improvement, placing zener diodes to improve robustness, making them safe to use on both net and battery power.

2.2 Encoder inputs

There are three differential incremental decoders placed on the board which are used to read the encoder signals coming from the motors.

2.3 H-bridges

H-bridges are used to power the three motors, so three H-bridges are present on the board. They are built using two DRV8412DDW dual full-bridge PWM motor drivers by Texas Instruments, of which one is operated in parallel mode to supply motor M1 with 6A continuous current and the other in dual mode to deliver 3A continuous current to motors M2 en M3. It is possible to drive M2 and M3 in parallel mode using jumpers, so two motors can be driven at 6A.

2.4 Analog Outputs

The board has two analog outputs, which are produced by an AD5722R dual 12-bit, serial input, voltage output, digital-to-analog converter (DAC) by Analog Devices. Currently they are not used in the SERGIO arm, but they are used in the base and torso to drive motors via Elmo Violins, since the TUeES030 board itself cannot supply enough current for these motors.

2.5 Analog Inputs

There are two 8-channel, 12-bits ADCs present on the board, namely the AS1543 by Austria Micro Systems. They accept analog input voltages in the range of 0-3.3V. One of them is used for the three absolute encoder signals (Hall sensors), three force sensors and two spare analog inputs, whereas the other one is used for the three current measurement signals and verification of the 5V reference and 24V supply voltage.

2.6 Digital Inputs/Outputs

The seven digital in- and outputs can be used for a wide range of applications, but currently only two digital outputs are used in the torso to control the callipers via a conversion to the RS-232 standard.

2.7 eBus connector

The communication with the EtherCAT master is realised using an ET1200 EtherCAT chip, that allows communication via eBus connectors. There are two eBus connectors, one for connection to the previous slave/master and the other for connection to the following slave. The chip runs on a 25 MHz clock coming from the FPGA.

2.8 Clock

On board there is a 50 MHz crystal oscillator, that is divided in the FPGA to handle the timing of the different components.

2.9 SPI Flash & LEDs

In order to reconfigure the FPGA at startup, its pin configuration can be stored on a flash memory chip that is connected via a serial peripheral interface (SPI) to the FPGA.

The LEDs can be used to indicate operating modes and other messages.

2.10 Current Measurement

The board is able to measure the currents that are applied to the motors, which are necessary in order to implement current control loops. In Figure 2.1 the part of the electric schematic that contains the current control measurement for motor M1 is shown. These currents are measured with a voltage over a resistor in series with the motor and amplified with a high speed precision current sense amplifier (LT1999), designed to monitor bidirectional currents over a wide common mode range. his amplifies

the differential voltage by a factor 20, after which the PWM switching frequency and noise are filtered by an RC-filter. The resistance of 120Ω and capacitance of 10nF result in a low pass filter with a cut off frequency of about 133 kHz.

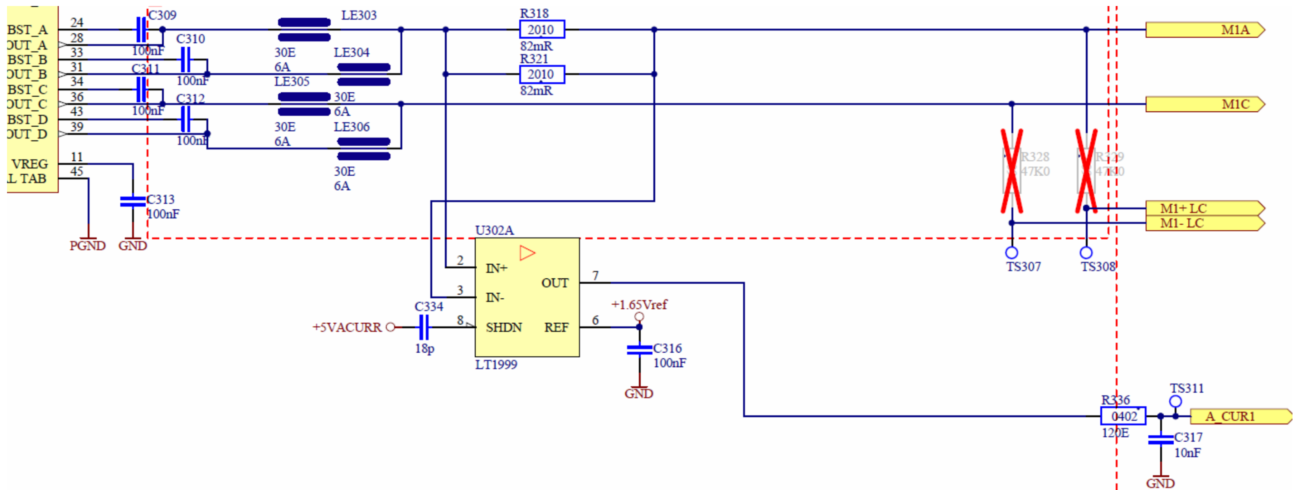


Figure 2.1: Current measurement schematic

Since the analog input to which the voltage from the LT1999 is connected has a maximum voltage of 3.3V, the maximum current that can be measured is only $I = \frac{V_{LT1999}}{20 \cdot R} = \frac{3.3}{20 \cdot 42 \cdot 10^{-3}} \approx 4$A. This is pretty low, considering that the largest motor in the arm has a stall current of around 39 A and for motors M2 and M3 this is only 2A (see the full schematic), so the choice is made to change the current sensing resistors to 10mΩ, resulting in maximum current measurements of 33A for motor M1 and 16.5A for motors M2 and M3.

The RC-filter is also adapted, because the PWM frequency is put at 80kHz and we want to filter the switching frequency out of the signal. The new filter has an adapted resistance of 1kHz, resulting in a cut-off frequency of around 16kHz.

Since some of the components were changed, it is *crucial* to verify which hardware is implemented in the current measurement when using the boards. The changes made are tracked in this document:
`https://docs.google.com/spreadsheets/d/1uwgKZpDQyB88insf7BYhXKnPYnASr0eK2cZU_hEENB4/pubhtml`

2.11 FPGA

The most important component on the board is the Field Programmable Gate Array (FPGA), which controls and manages all signals on the boad. It is discussed extensively in the next chapter.

Chapter 3

Field Programmable Gate Array

An FPGA is a reconfigurable integrated circuit consisting of (amongst others) programmable logic blocks, configurable interconnects that allow these blocks to be connected, input/output blocks (I/O blocks) and a configuration block[1]. Programmable logic blocks, or configurable logic blocks (CLBs) as they are also known, contain logic gates, flip-flops, lookup tables and other digital logic. A typical FPGA chip contains tens or hundred of thousands of these CLBs if not more. Input/output blocks or pins connect the chip to other devices and contain some logic themselves. The interconnects, as the name suggests, connect CLBs with each other and with I/O blocks and are configurable as well. The configuration block loads the information on how the CLBs, I/O blocks and interconnects are configured from an external flash memory where the configuration information is stored. A very simplified, schematic representation of an FPGA is shown in Figure 3.1, where the yellow, blue and white blocks represent I/O, configuration and CLB blocks, respectively. The red lines represent the interconnects.

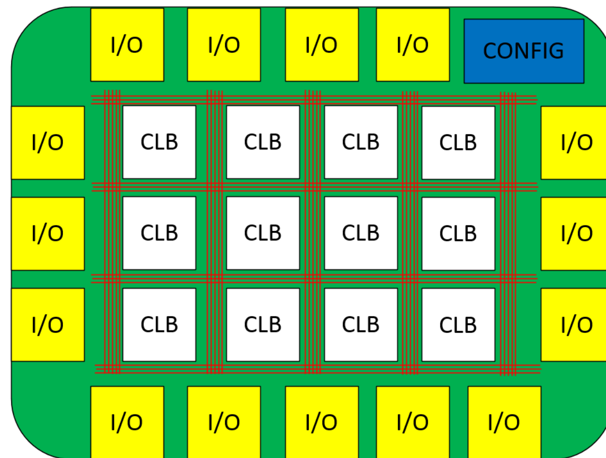


Figure 3.1: Conceptual representation of an FPGA

The advantages of an FPGA that we will make most use of are their speed (they are extremely fast) and their parallel nature. By parallel nature we mean the fact that several operations can be executed truly parallel, in contrast to micro-processors (MPs) which only perform operations sequentially.

[1]Most of the information in this introduction was derived from this YouTube video: `https://www.youtube.com/watch?v=gUsHwi4M4xE` by EEVblog

Figure 3.2: Xilinx Spartan®-6

The FPGA chip used in the TUeES030 board is the Spartan®-6 by Xilinx as displayed in Figure 3.2 and it is programmed using Altium Designer 14.3 (or higher). In the rest of this chapter the steps and files needed to program the FGPA will be presented. Note that the introduction above is only a very, very short introduction into FPGAs, for further reading the book 'FPGA for Dummies' by Andrew Moore is suggested. For more information on the code and the FPGA itself, please contact Ruud van den Bogaert.

3.1 Constraint File

Programming the FPGA chip starts with the constraint file, where the mapping from the device independent HDL circuit nets to the physical in- and output pins of the FPGA chip is defined. This means that Verilog is a device independent language, so only the constraint file needs to be changed if ports or the device itself is changed! It is *crucial* that this is done correctly, since they represent actual connection pins on the FPGA and if done incorrectly, the FPGA could be damaged.

In Figure 3.3 part of the constraint file `Neways_Spartan6.Constraint` for the TUeES030 boards is shown. Here, the third encoder and the first PWM H-bridge pins are being defined. All ports follow a similar structure, they have the `Record` type `Constraint`, the `TargetKind` of `Port`, a `Target_Id` which can be chosen by the user (we advise to keep this the same as in the electronic schematics, making it a lot easier to track pins) and an FPGA pin number `FPGA_PINNUM` that represents the physical pin of the FPGA to which the target port is connected. In other words, one line of code from Figure 3.3 says that a port in the Altium schematic be connected to the corresponding FPGA pin number, for instance that port `M1_PWM_A` should be connected to pin `R1`.

```
;Encoder 3
Record=Constraint | TargetKind=Port | TargetId=ENC_3_XP     | FPGA_PINNUM=H4
Record=Constraint | TargetKind=Port | TargetId=ENC_3_YP     | FPGA_PINNUM=H3
Record=Constraint | TargetKind=Port | TargetId=ENC_3_IP     | FPGA_PINNUM=L4

;PWM H-Bridge 1
Record=Constraint | TargetKind=Port | TargetId=M1_PWM_A     | FPGA_PINNUM=R1
Record=Constraint | TargetKind=Port | TargetId=M1_PWM_B     | FPGA_PINNUM=P2
Record=Constraint | TargetKind=Port | TargetId=M1_PWM_C     | FPGA_PINNUM=P1
Record=Constraint | TargetKind=Port | TargetId=M1_PWM_D     | FPGA_PINNUM=N3
Record=Constraint | TargetKind=Port | TargetId=M1_RESET_AB  | FPGA_PINNUM=N4
Record=Constraint | TargetKind=Port | TargetId=M1_RESET_CD  | FPGA_PINNUM=R2
Record=Constraint | TargetKind=Port | TargetId=M1_MODE1     | FPGA_PINNUM=M1
Record=Constraint | TargetKind=Port | TargetId=M1_MODE2     | FPGA_PINNUM=L3
Record=Constraint | TargetKind=Port | TargetId=M1_FAULT     | FPGA_PINNUM=N1
Record=Constraint | TargetKind=Port | TargetId=M1_OTW       | FPGA_PINNUM=M2
```

Figure 3.3: Part of the Constraint File for the TUeES030 board

3.2 OpenBus System Document

The next step is to create an OpenBus System Document, where the connections between the different peripherals and the micro-processor are defined. Note that the peripherals and micro-processor are not physical devices connected to the FPGA, instead they are parts of the FPGA that are configured to work as a MP or as a peripheral, so they can be chosen by the user to provide an insightful subdivision. This part of the FPGA program uses the *wishbone* standard for communication.

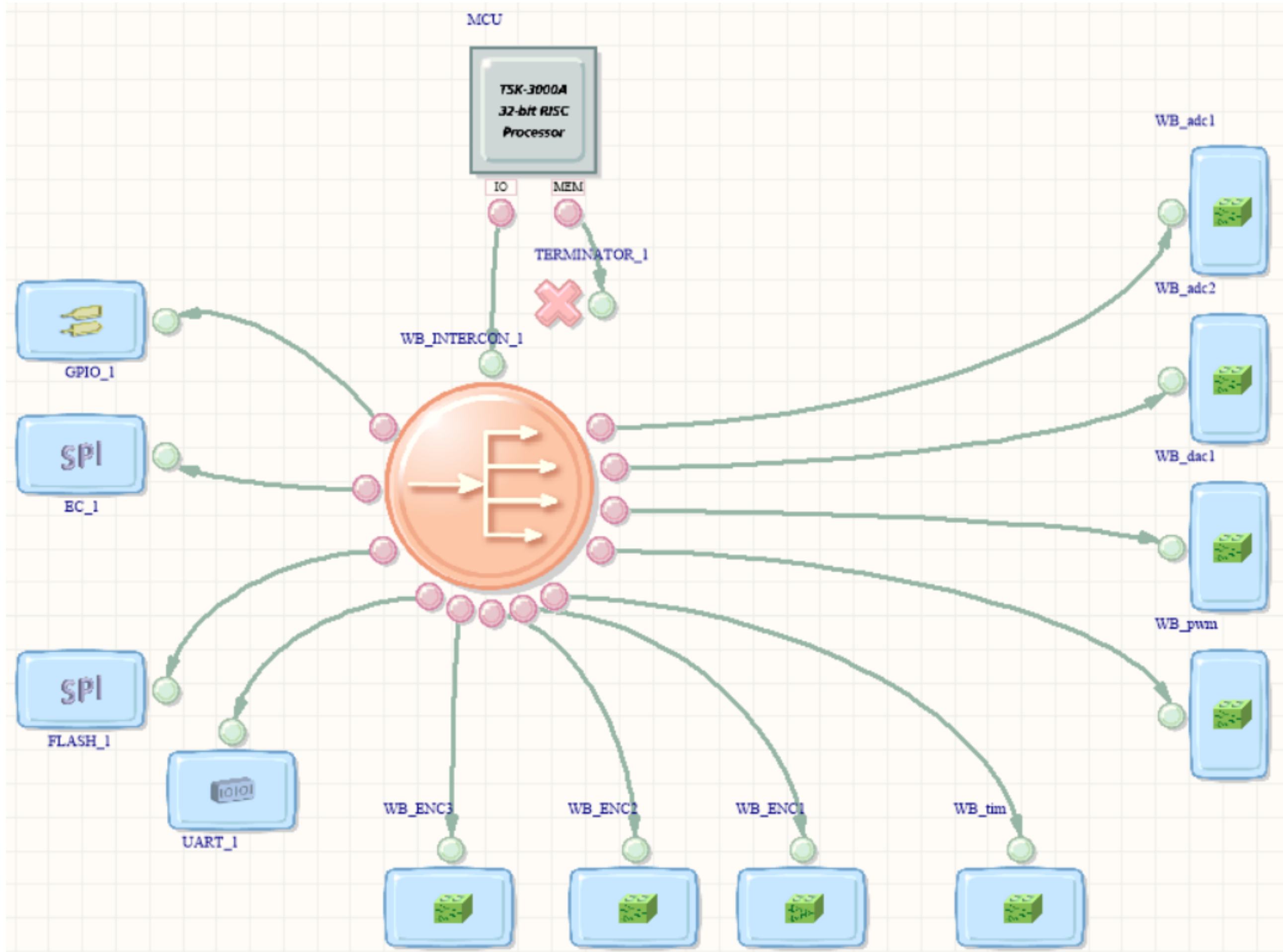


Figure 3.4: OpenBus System Document for the TUeES030 FPGA

In Figure 3.4 the OpenBus System Document for the TUeES030 boards is shown. Here we can see a micro-processor at the top (see Chapter 4), twelve peripherals and one interconnect device in the center, all of which can be found in the OpenBus palette. The peripherals are, in clockwise order starting to the right of the MP, two analog-to-digital converters (ADCs), one digital-to-analog converter (DAC), PWM creation, EtherCAT timer, three encoders, universal asynchronous receiver/transmitter (UART), two serial peripheral interfaces (SPIs) and general purpose input/output (GPIO).

The peripherals with the green blocks in them are `custom wishbone interfaces` and will be discussed in Section 3.4. The others are already programmed and available from the OpenBus palette and only need to be configured. We will not discuss the configurations in detail, but only mention their purpose. The GPIO is used for the eight digital in- and outputs, the EC_1 SPI port is used for communication over EtherCAT, the other SPI port is connected to the flash memory for storing the FPGA configuration and the UART peripheral is used for controlling the callipers.

3.3 Schematic

Now that we connected the MP with all the peripherals in the OpenBus System Document, all the internal connections are defined. What remains is the connection with the external pins and the completion of the custom wishbone interfaces. This is done using the top-level schematic, which can be added to the Altium FPGA Project by right clicking on the project and choosing `Add New to Project -> Schematic`. The result is an empty schematic, where the ports defined in the Constraint File and the connections with the OpenBus System Document need to be placed. Adding the ports is done by right clicking on the schematic and choosing `Place -> Port`, after which the port can be configured by double clicking on it and selecting the correct variable from the drop down menu `Name`. The connections with the OpenBus are most easily made by adding a sheet symbol (`Place -> Sheet Symbol`), configuring it with the OpenBus System Document (double click and select the OpenBus document as Filename) and letting Altium synchronise the ports (right click on the Sheet Symbol and select `Sheet Symbol Actions -> Synchronise Sheet Entries and Ports`). The ports can be connected using wires and busses, where wires are used for ports with single variables and busses are used for ports with arrays of variables.

The custom wishbone interfaces can be created by placing a Sheet Symbol on the schematic and choosing in the right click menu `Sheet Symbol Actions -> Create Verilog File From Sheet Symbol`. In this Verilog File the interfaces can be programmed, which will be discussed in Section 3.4.

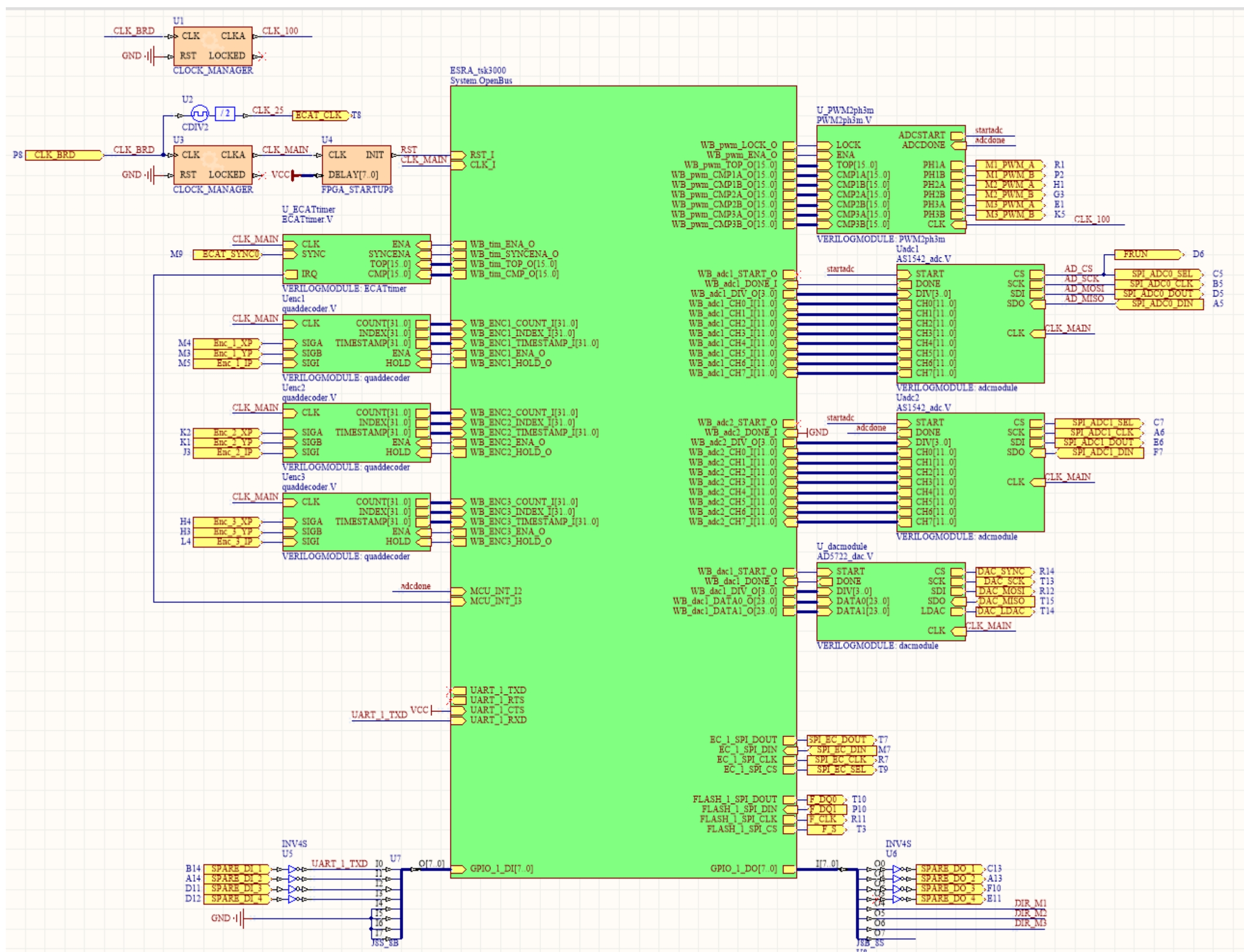


Figure 3.5: Top-level schematic of the TUeES030 FPGA code

In Figure 3.5 the top-level for the TUeES030 FPGA code is shown, where the main block in the center is the Sheet Symbol coupled to the OpenBus System Document and the custom wishbone interfaces

are the blocks to the sides. The other peripherals are represented by ports on the main block. The pink blocks in the top left corner are two Clock Managers and an FPGA Startup block. The clock managers are used to create clocks with different frequencies than the physical on-board clock and the FPGA_STARTUPS is used to delay the program in the beginning, so that there is some time for initialisation.

3.4 Verilog Code

As mentioned, the code for the custom wishbone interfaces is programmed using Verilog. Verilog is a hardware description language (HDL) that has very similar syntax to C. The name HDL is logical, since we are actually programming how the hardware should be configured, so we are *describing the hardware*. The crucial difference between C an Verilog is that it has the ability to execute code in parallel due to the fact that we are programming hardware, in contrary to C, where commands are executed sequentially.[2]

3.4.1 Basics

We will now discuss some of the Verilog basics using the simplest Verilog code that is used in the TUeES030 board: `ECATtimer.V`, as shown in Listing 3.1 in Section 3.4.2. All Verilog codes start the same, with the declaration of the module name and its input and outputs as in lines 1-8. The `[15:0]` denotes that the 0th through to the 15th bit of the TOP and CMP signals are inputs, which in this case are the complete signals since they are both 16-bits variables. `output reg` means that `IRQ` is a register, which is the same as a variable, and is an output. The reason that we need to declare the type of the output and not of the inputs is because the inputs are already defined externally whereas the output is first computed in this function.

```
1  module ECATtimer(CLK, ENA, SYNC, SYNCENA, TOP, CMP, IRQ);
2
3  input          CLK;
4  input          ENA;
5  input          SYNC;
6  input          SYNCENA;
7  input  [15:0]  TOP, CMP;
8  output reg     IRQ;
9
10 reg            IRQa;
11 reg            IRQb;
12 reg            synca, syncb;
13 reg    [15:0]  countera;
14
15 wire           clear;
16 assign         clear = ~ENA;
17
18 wire           cntatop;
19 assign         cntatop = (countera == TOP);
20
21 wire           syncpulse;
22 assign         syncpulse = (synca & ~syncb);
23
24 always @ (posedge CLK)
25 begin
26   if (clear) begin
27     countera <= 0;
28     synca <= 0; syncb <= 0;
29     IRQ <= 0;
30     IRQa <= 0;
```

[2] `https://nl.wikipedia.org/wiki/Verilog`

```
    IRQb <= 0;
  end
  else begin
    synca <= SYNC & SYNCENA;
    syncb <= synca;
    if(cntatop | syncpulse) begin
      countera <= 0;
    end
    else begin
      countera <= countera + 1'b1;
    end
    if(countera == CMP) begin
      IRQ <= 1;
      IRQa <= 1;
    end
    IRQb <= IRQa;
    if(IRQ & IRQb) begin
      IRQ <= 0;
    end
  end
end

endmodule
```

Listing 3.1: `ECATtimer.V`

Next, the internal variables are declared as registers, where `countera` has 16-bits. Following these registers are some more internal 'variables' known as *wires*. These are not really variables, but more actual wires, meaning that they change instantaneously if one of the variables they depend on changes. For instance the wire `clear` is assigned the value of not `ENA` (short for enable), so if at any moment `ENA` changes, `clear` changes immediately to the opposite of `ENA`. Similarly `cnatop` turns to a logical true if the value of `countera` equals that of `TOP` and `syncpulse` is true when both `synca` and not `syncb` are true.

Following this the main loop of the program starts with `always @ (posedge CLK)`, meaning that the code between the `begin` and its associated `end` is ran every time the input variable `CLK` (clock) has a positive edge. In Verilog this is known as a *sequential block*, which are all the blocks that start with `begin` and end with `end`. As the name suggests, inside these sequential blocks the code runs sequentially and *not* in parallel. The `if else` statements are similar to their C equivalent. A final comment on Verilog programming before we explain the workings of this piece of code is that the operator `<=` is *not* the comparator 'less than or equal to' which we are used to from C. In Verilog the operator `<=` is known as a *non-blocking assignment*, because contrary to the *blocking* assignment, denoted by the operator `=`, it does not block the program from executing.

```
module toplevel(clock,reset);
  input clock;
  input reset;

  reg flop1;
  reg flop2;

  always @ (posedge reset or posedge clock)
    if (reset)
      begin
        flop1 <= 0;
        flop2 <= 1;
      end
    else
      begin
        flop1 <= flop2;
        flop2 <= flop1;
      end
endmodule
```

Listing 3.2: Blocking vs non-blocking example, source: `https://nl.wikipedia.org/wiki/Verilog`

Blocking the program means that the next line of code cannot be executed as long as the current line with the blocking operator `=` is not executed, thus preventing, or blocking, the program from continuing. The non-blocking operator prevents this by only updating all the variables at the beginning of the next clock cycle, all at exactly the same time. The difference between the two assignment methods is best shown in an example as in Listing 3.2. In this code, the registers `flop1` and `flop2` are defined as 0 and 1 when the `reset` is given, after which they switch values each time the `clock` signal has a positive edge. If we would change the last two non-blocking assignments to blocking assignments, the code would have a completely different result, namely that `flop1` would be immediately set to the value of `flop2`, after which `flop2` would be given the value of `flop1`, thus they both end up being the same value. With the non-blocking assignment the right hand side of the operators is checked and at the next cycle the left hand side registers are updated *simultaneously*, thus resulting in the values switching.[3]

Now that the basics of Verilog are clear, the workings of the five different Verilog codes used are explained.

3.4.2 `ECATtimer.V`

As could be seen in the schematic of Figure 3.5, the input `SYNC` comes from the physical port ECAT_SYNC0, which is a signal that can be used to sync with the etherCAT master, but for now it just runs at 1 kHz. The input `CLK` is connected to the main clock of the FPGA program, running at 40 MHz. The rest of the inputs are all defined in the Openbus block and are thus coming from the softcore.

The main loop clears all registers as long as `clear` is true, so as long as the block is not enabled. If the block is enabled, `synca` goes to 1 if both `SYNC` and `SYNCENA` are true, so if syncing is enabled and the clock gives a syncing signal. Since `syncb` is defined using a non-blocking constraint, it stays at zero, only going to 1 one time step later. This means that `syncpulse` goes to 1, as soon as the `SYNC` command is given and goes to zero the time step after, thus creating a syncing pulse, as shown graphically in Figure 3.6.

[3] `https://nl.wikipedia.org/wiki/Verilog`

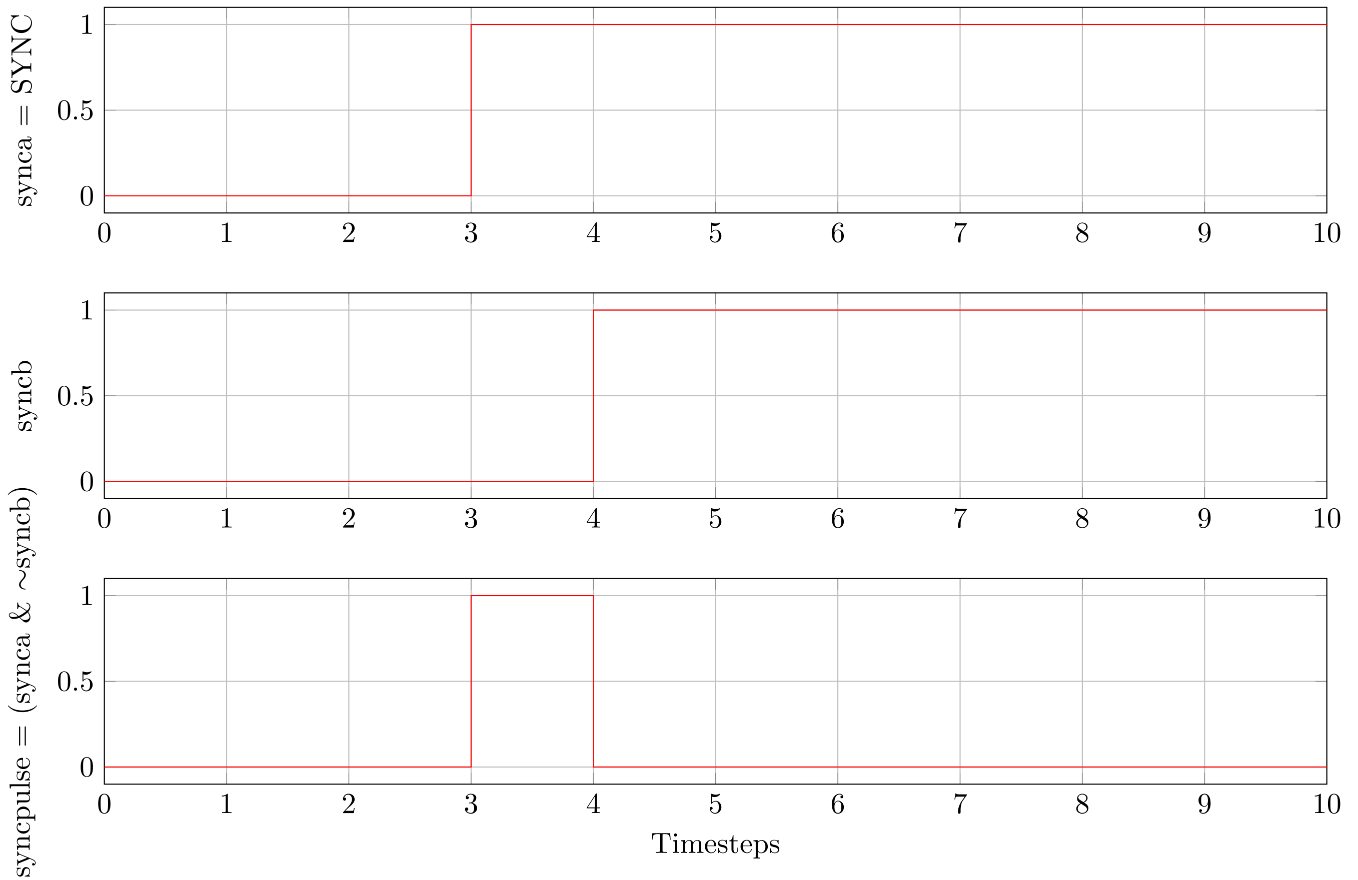


Figure 3.6: Pulse creation

The following `if else` statement in lines 36-41 resets `countera` if either it equals the value defined in top or the syncing command is given. If not being reset, the value increases with 1 every time step, thus creating a saw tooth signal. The notation `1'b1` stands for a value that has a size of 1 bit, is in binary format and has the value 1, similarly `16'h0000` is a value with a size of 16 bits, is in hexadecimal format and has the value 0.

The final part of the code in lines 42-49 creates an instantaneous interrupt (`IRQ`) pulse as soon as `countera` equals the comparator value `CMP`. Thus this Verilog file creates a synced interrupt signal using a sawtooth and comparator. It is used to activate an interrupt that takes care of the encoder calculations at a lower update rate of 1 kHz, as shown in Section 4.2.2.

3.4.3 `PWM2ph3.V`

The next Verilog code discussed is the one used to create the PWM signals for driving the H-bridges of the motors. We will no longer discuss the code line by line, but just explain its function and leave it to the reader to verify that this is indeed the result the code produces.

In order to create the PWM signals, the code uses a single counter that forms a triangle, where the top of the triangle can be adjusted in order to adjust the frequency. For each motor there are two comparators, one for phase A and the other for phase B, with their levels calculated in the higher level software of the micro-processor, see Section 4.2.1.

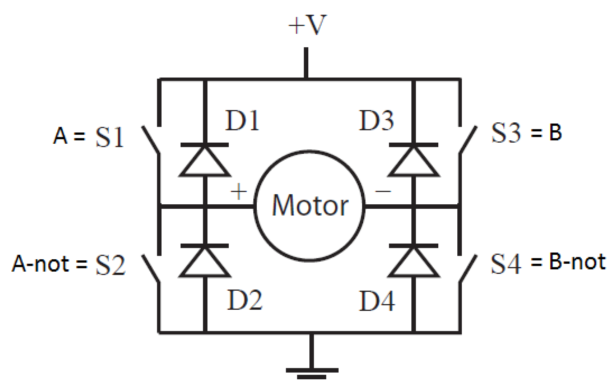


Figure 3.7: H-bridge with flyback diodes

In Figure 3.7 an H-bridge as used in the TUeES030 board is shown. The use of an H-bridge to drive a brushed DC motor is very common practice and has the advantage of being able to drive a motor using PWM signals as opposed to analog signals, which are often not available. Another advantage is that the motor can be driven in both directions and can be powered from another source than the (micro-)controller used to control the switches, which cannot supply sufficient power. In Table 3.1 the possible switch positions and their results are shown. From this we can conclude that S1 and S2 should always be in opposite position or both open, as should S3 and S4.

Closed Switches	Result
S1, S4	Positive voltage across motor
S2, S3	Negative voltage across motor
S1, S3	Braking
S2, S4	Braking
S1, S2	Short
S3, S4	Short
Three or more	Short
1 or none	Tristate (the axis can rotate freely without inducing back-emf)

Table 3.1: Switching combinations for H-bridge

The PWM signals created in this Verilog code are referred to as phase A and phase B, where phase A is the S1 control, so A-not (or $\bar{A}$) is the control for S2 and phase B is the control for S3 and $\bar{B}$ is the control for S4. Also, phase A and B are both at 50% for no input, so if we want the motor to stay still and they change equal and opposite from the center, so if we put phase A at 75%, it means that phase B has to be at 25%. Examples showing the triangular counter waveform in red, the comparator values with a black dashed line and the resulting PWM signals and end result on the motor for the 50% and 75% cases are shown in Figures 3.8 and 3.9, respectively.

The advantage of having the PWM signals at 50% for zero motor current is that it removes the problem of having very small PWM peaks at very low power, which result in high frequency noise. The advantage of using the triangular comparator signal together with PWM varying around the center can be seen in the 75% case, where if we look at the result across the motor we can see that the frequency the motor experiences is twice as large as the PWM frequency, so we doubled our PWM frequency by a good choice of signal!

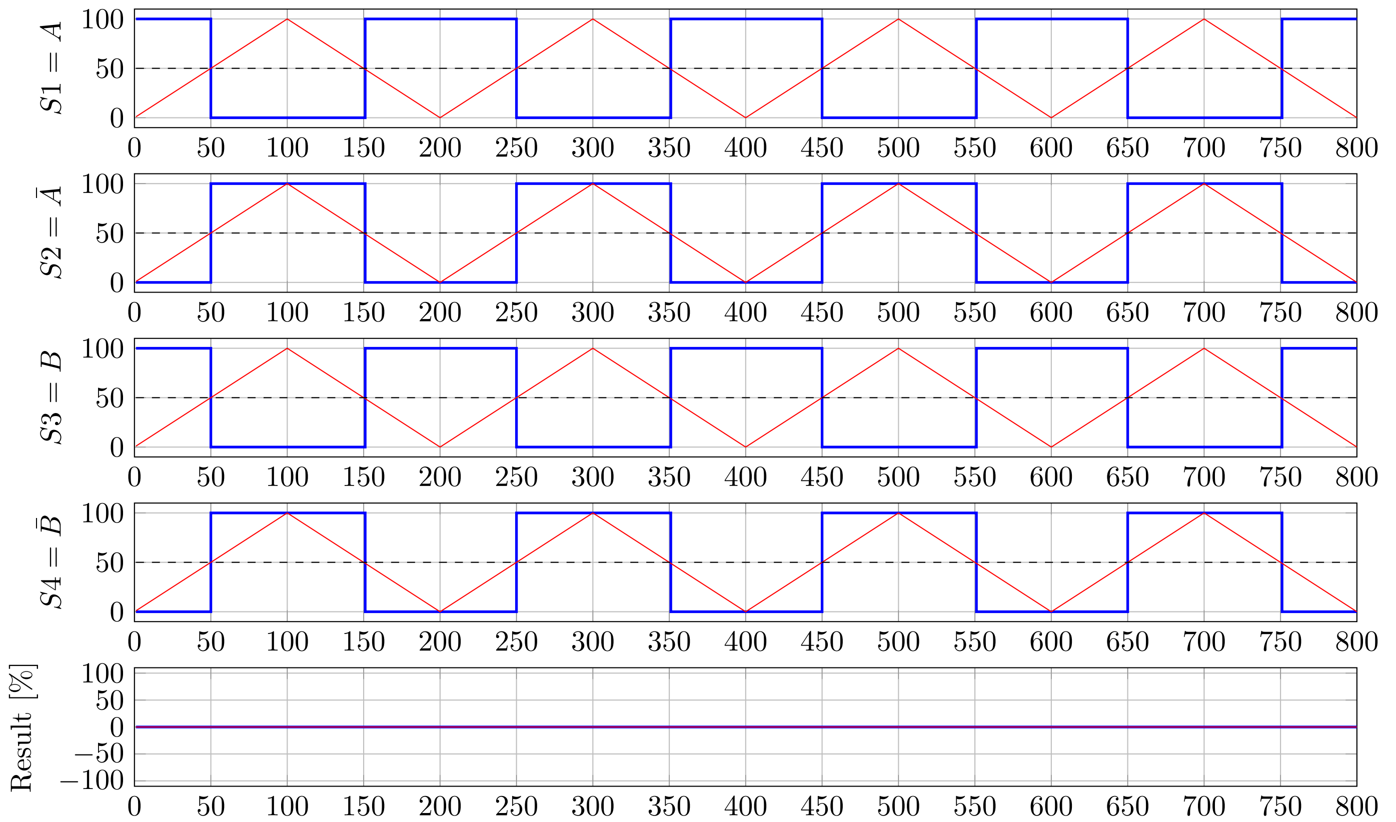


Figure 3.8: PWM waveforms at 50%

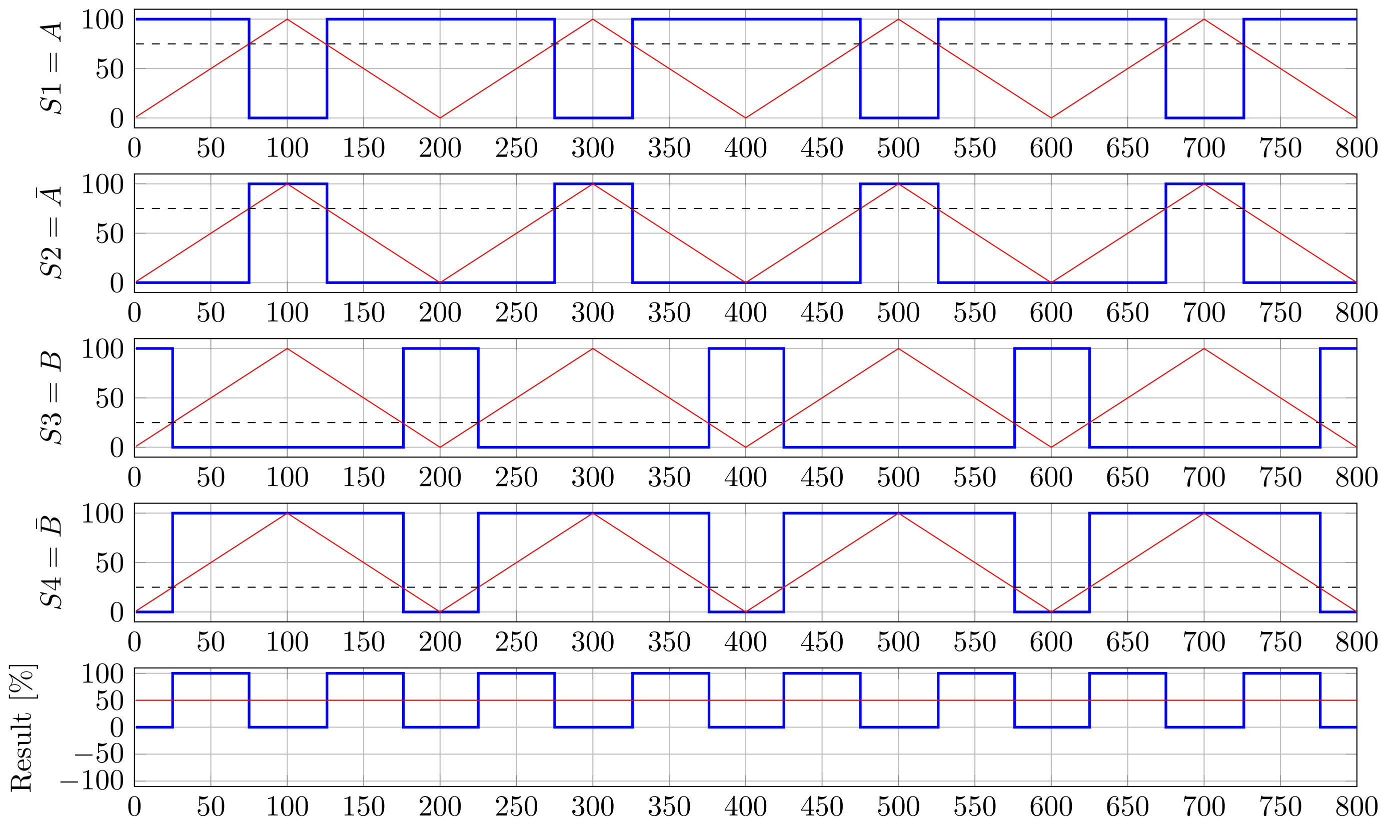


Figure 3.9: PWM waveforms at 75%

Lastly the code ensures that the top (for adjusting the PWM frequency) and comparator parameters

are only adjusted when the counter is at the bottom, thus ensuring smooth transitions. Also, the ADC is only allowed to be read when the counter has reached the top, resulting in perfect synchronisation.

3.4.4 `AS1542_adc.V`

The 16 analog inputs are read via two 8-channel multiplex ADCs, which work *sequentially*. The two ADCs are read *in parallel* in the FPGA.

3.4.5 `AD5722_dac.V`

In this block the code for the two channel DAC is implemented.

3.4.6 `quaddecoder.V`

The quadrature signal coming from the encoder receivers (shown in Appendix A), are converted to counts in this module. The signals originate from the encoders on the motor shafts, which supply two output signals A and B and an index signal I. A and B are pulse signals that are 90 degrees phase shifted from each other, which as known as being *in quadrature*. Because they are in quadrature, both the position and direction of rotation can be determined from them. The index signal gives a pulse ones every motor rotation, which can be used for absolute positioning. However, in most cases where there is a transmission attached to the motor shaft, the index cannot be used for absolute positioning.

3.4.7 `swap32.V`

This module is used to swap from the host endianness to the ethercat endianness in the FPGA, thus being faster than the micro-processor implementation explained in Section 4.5.1, where the swapping process is also explained in more detail.

Chapter 4

Micro Processor

In an FPGA, one or more microprocessors can be placed. Our FPGA project includes a 32-bit open-source freeware soft-core processor by Tasking. The code will not be explained line by line, since there are over a thousand of them. Instead, the most important are explained, for more information please view the code itself, which has comments throughout. All the code is written in the programming language C.

The board is programmed such that it has two operating modes: normal and frequency response function (FRF) mode. In normal mode all the inputs and outputs are available, whereas in FRF mode the inputs and outputs are limited to a single motor, in order to achieve enough data throughput to be able to sample with 20 kHz. More on this mode can be found in Chapter 5.

4.1 Main Code

The main file of the micro-processor code is `ecat_actuator.c` and it's where most of the code is located. As with all C-code, the program is defined in the main function, as displayed in Listing 4.1, where several sub-functions are defined.

```
912 int32_t main(void)
913 {
914   swplatform_init_stacks();
915   // Setup post config hooks
916   static esc_cfg_t config =
917   {
918      .pre_state_change_hook = NULL,
919      .post_state_change_hook = post_state_change_hook
920   };
921   ESC_config ((esc_cfg_t *)&config);
922   RXPDOsize = SM2_sml = sizeRXPDO();
923   TXPDOsize = SM3_sml = sizeTXPDO();
924   foe_init();
925   delay_ms(100);
926   waitforESCready();
927   ESC_ALstatus(ESCinit);
928   ESC_ALerror(ALERR_NONE);
929   ESC_stopmbx();
930   ESC_stopinput();
931   ESC_stopoutput();
932   init_param();
933   init_actuators();
934   init_sensors();
935   reset_wd();
936   init_interrupts();
937
938   // cyclic application loop
```

```
939   while(1)
940   {
941     ESC_read(ESCREG_LOCALTIME,(uint8_t*)&ESCvar.Time,sizeof(ESCvar.Time),(uint8_t*)&
          ESCvar.ALevent);
942     ESCvar.Time=etohl(ESCvar.Time);
943     ESC_state();
944     if(ESC_mbxprocess())
945     {
946       ESC_coeprocess();
947       ESC_foeprocess();
948     }
949     handle_RXPDO();
950     handle_TXPDO();
951   }
952 }
```

Listing 4.1: `int32_t main(void)`

4.1.1 Initialisation

The main loop starts with the initialization of the Software Stacks, by running the function `swplatform_init_stack` which initializes the wishbone devices. This is followed by several EtherCAT initialisation and configuration steps in lines 916-931, which we will not discuss.

Next in lines 932-936 are more initialisations, shown in Listing 4.2. In `init_param` the parameters (see Section 4.4.3) are first set to their default values (in SI units) in `default_param` after which they are converted to internal values in `param2int`. Then the actuator are initialised in `init_actuators`, where the digital outputs are set to zero in `set_digout`, the digital to analog converts are initialised in `init_DAC` and the PWM signals are started in `init_PWM`. In `init_sensors` the encoders are switched on and in `reset_wd` the watchdog is reset. Finally, the interrupts are initialised in `init_interrupts`, more on this in Section 4.2.

```
static void init_param(void)
{
  default_param();
  param2int();
}

static void init_actuators(void)
{
  set_digout(0);
  init_DAC();
  init_PWM();
}

static void init_sensors(void)
{
  ENC1_ENA = 0;
  ENC2_ENA = 0;
  ENC3_ENA = 0;
  ENC1_ENA = 1;
  ENC2_ENA = 1;
  ENC3_ENA = 1;
}

static inline void reset_wd(void)
{
  wd_cnt = wd_resetvalue;
  wd_trigger = 0;
}
```

```
void init_interrupts(void)
{
  // configure PWM/ADC interrupt
  interrupt_register_native( INTPWM, (void*)NULL, i_handler_pwm );
  interrupt_configure( INTPWM, EDGE_RISING );
  interrupt_acknowledge(INTPWM);
  interrupt_enable(INTPWM);

  // configure service timer and interrupt
  TIM_TOP = SERVICEPER - 1;
  TIM_ENA = 1;
  interrupt_register_native(INTEC, (void*)NULL, i_handler_ec );
  interrupt_configure(INTEC, EDGE_RISING );
  interrupt_acknowledge(INTEC);
  interrupt_enable(INTEC);
  interrupts_enable();
}
```

Listing 4.2: Initialisations

4.1.2 Update

Following the initialisation is a while loop in lines 939-951 of Listing 4.1, where the majority of the functions are called. It starts with several EtherCAT communication steps, which take care of writing and reading data to and from the master, these will not be discussed in further detail.

After the communication there are two functions that handle receiving and transmitting of process data: `handle_RXPDO` and `handle_TXPDO`. The function names stand for receive and transmit PDO (process data object) mapping, respectively.

```
535 static void handle_RXPDO(void)
536 {
537   int count, i;
538   uint8_t tmpu8;
539   if(App.state & APPSTATE_OUTPUT)
540   {
541     if(ESCvar.ALevent & ESCREG_ALEVENT_SM2)
542     {
543       if(operationmode == OPERATION_STD)
544       {
545         ESC_read(SM2_sma,(uint8_t*)&rxpdo1,RXPDOsize,(uint8_t*)&ESCvar.ALevent);
546         update_actuators();
547       }
548       // operation mode FRF
549       else
550       {
551         ESC_read(SM2_sma, (uint8_t*)&rxpdo2,RXPDOsize,(uint8_t*)&ESCvar.ALevent);
552         // use intermediate varable to prevent race condition with i_handler_pwm
553         // mcom : bit 0..1 = motor channel , bit 2 = enable , bit 3 = tristate active
554         tmpu8 = rxpdo2.mcom & MCOM_FRF_MMASK;
555         if(tmpu8 > 2) tmpu8 = 2;
556         frfchannel = tmpu8;
557         if(frfchannel != 0) motor[0].enabled = 0;
558         if(frfchannel != 1) motor[1].enabled = 0;
559         if(frfchannel != 2) motor[2].enabled = 0;
560         if(rxpdo2.mcom & MCOM_FRF_ENABLE)
561           motor[frfchannel].enabled  = MCOM_ENABLE;
562         else
563           motor[frfchannel].enabled  = 0;
564         if(rxpdo2.mcom & MCOM_FRF_TRISTATE)
565           motor[frfchannel].tristate = MCOM_TRISTATE;
566         else
```

```
567          motor[frfchannel].tristate = 0;
568        count = rxpdo2.entries;
569        if(count > FRFBUFFERSIZE) count = FRFBUFFERSIZE;
570        for(i=0; i < count ; i++)
571        {
572          if(rxfree())
573            rx_buffer_write(htoes2(rxpdo2.setpoint[i]));
574        }
575        txbufferentries = 0;
576        txpdo2.entries = 0;
577      }
578      reset_wd();
579    }
580    // watchdog time has elapsed?
581    if(!wd_cnt)
582    {
583      ESC_stopoutput();
584      // watchdog, invalid outputs
585      ESC_ALerror(ALERR_WATCHDOG);
586      // goto safe-op with error bit set
587      ESC_ALstatus(ESCsafeop | ESCerror);
588      wd_trigger = 1;
589    }
590  }
591  else
592  {
593    reset_wd();
594  }
595 }
```

Listing 4.3: `handle_RXPDO`

The receiving part is displayed in Listing 4.3, where it becomes clear that there are two operating modes, normal and FRF (frequency response function) mode. More on these two modes in Chapter 5. In normal mode the settings are read from the EtherCAT master and subsequently the actuators are updated in `update_actuators`, as shown in Listing 4.4. Here the current setpoint, feedforward signal, enabled and tristate setting are updated for each of the three motors. Also, the feedforward signal is clipped between the maximum negative and positive voltages.

```
501 static inline void update_actuators(void)
502 {
503   motor[0].current_setpoint = htoes2(rxpdo1.setpoint1);
504   motor[0].setpoint_ff      = clip16(htoes2(rxpdo1.ff1), -MAXVOLTAGE, MAXVOLTAGE);
505   motor[0].enabled          = rxpdo1.mcom1 & MCOM_ENABLE;
506   motor[0].tristate         = rxpdo1.mcom1 & MCOM_TRISTATE;
507   motor[1].current_setpoint = htoes2(rxpdo1.setpoint2);
508   motor[1].setpoint_ff      = clip16(htoes2(rxpdo1.ff2), -MAXVOLTAGE, MAXVOLTAGE);
509   motor[1].enabled          = rxpdo1.mcom2 & MCOM_ENABLE;
510   motor[1].tristate         = rxpdo1.mcom2 & MCOM_TRISTATE;
511   motor[2].current_setpoint = htoes2(rxpdo1.setpoint3);
512   motor[2].setpoint_ff      = clip16(htoes2(rxpdo1.ff3), -MAXVOLTAGE, MAXVOLTAGE);
513   motor[2].enabled          = rxpdo1.mcom3 & MCOM_ENABLE;
514   motor[2].tristate         = rxpdo1.mcom3 & MCOM_TRISTATE;
515   cpy_digout((rxpdo1.digital & 0x0f) | (dostate & 0xf0));
516   update_DAC();
517 }
```

Listing 4.4: `update_actuators`

In the FRF mode again the settings are read from the EtherCAT master and updated, but instead of having a single setpoint value that is updated, there are up to 22 setpoints that are put into a receiving buffer using `rx_buffer_write`. More on this buffer in Subsection 4.3.

```
597 static void handle_TXPDO(void)
598 {
599   int16_t temp;
600   uint8_t di;
601   if(App.state & APPSTATE_INPUT){
602     if(operationmode == OPERATION_STD)
603     {
604       read_encoders();
605       read_calipers();
606       read_ADC();
607       di = read_digin();
608       // bit 0 : enabled
609       // bit 1 : DRV fault
610       // bit 2 : DRV over temp warning
611       txpdo1.mstate1 = (uint8_t)motor[0].enabled | ((di & (M1_FAULT | M1_OTW)) >> 3);
612       // motor 2 and 3 have common driver and therefore common faults
613       txpdo1.mstate2 = (uint8_t)motor[1].enabled | ((di & (M2_M3_FAULT | M2_M3_OTW))
             >> 5);
614       txpdo1.mstate3 = (uint8_t)motor[2].enabled | ((di & (M2_M3_FAULT | M2_M3_OTW))
             >> 5);
615       txpdo1.linevoltage = htoes2(linevoltage_i);
616       txpdo1.digital = di & DIGINMASK;
617       ESC_write(SM3_sma,(uint8_t*)&txpdo1,TXPDOsize,(uint8_t*)&ESCvar.ALevent);
618     } else if(!txbufferentries)
619     {
620       while(tx_counter && (txbufferentries < FRFBUFFERSIZE))
621       {
622         temp = tx_buffer_read();
623         txpdo2.current[txbufferentries] = htoes2(temp);
624         txpdo2.entries = (uint8_t)txbufferentries;
625         txbufferentries++;
626       }
627       di = read_digin();
628       txpdo2.mstate = (uint8_t)(motor[0].enabled | motor[1].enabled | motor[2].
             enabled) |
629                      ((di & (M1_FAULT | M1_OTW)) >> 3) |
630                      ((di & (M2_M3_FAULT | M2_M3_OTW)) >> 5);
631       txpdo2.buffer = (uint8_t)rx_counter;
632       ESC_write(SM3_sma,(uint8_t*)&txpdo2,TXPDOsize,(uint8_t*)&ESCvar.ALevent);
633     }
634   }
635 }
```

Listing 4.5: `handle_TXPDO`

The transmitting of the PDO mapping as displayed in Listing 4.5 works in much the same way as the receiving part, as it also has a normal and FRF operating mode. In the normal mode the encoders, callipers, ADCs, DIs, motor states and line voltages are read and updated. In the FRF mode most of these values are not read, but instead only the motor state together with up to 22 current measurement values in the transmit buffer are sent over to the master.

4.2 Interrupts

Apart from the main application loop, there are also two so called *interrupts* defined in `ecat_actuator.c`. Interrupt functions are pieces of code that interrupt the normal sequential flow of the C-code to be ran immediately. They are used for code that has a very high timing priority, so code that needs to be executed immediately.

4.2.1 PWM handler

The first interrupt `i_handler_pwm` is shown in Listing 4.6 and handles the calculation of the PWM signals, running at a frequency of 20 kHz. It starts by subtracting the common mode voltage from the current measurement and subsequently converting the current from internal to 1 mA units. Then, if the FRF mode is active, current setpoints are retrieved from the receive buffer or set to zero if there are none (in normal mode the current setpoints are read from the master in `update_actuators`).

```
658 __INTERRUPT_NATIVE void i_handler_pwm(void)
659 {
660   int16_t adc_current1, adc_current2, adc_current3;
661   adc_current1 = ADC2_CH3 - zero_current1;
662   adc_current2 = ADC2_CH4 - zero_current2;
663   adc_current3 = ADC2_CH5 - zero_current3;
664   // convert ADC current units to 1mA units
665   motor[0].current = q8mul(adc_current1, (int16_t)(R2IGAINCH1 * Q8f));
666   motor[1].current = q8mul(adc_current2, (int16_t)(R2IGAIN * Q8f));
667   motor[2].current = q8mul(adc_current3, (int16_t)(R2IGAIN * Q8f));
668   if(operationmode == OPERATION_FRF)
669   {
670     if(rx_counter)
671     {
672       motor[frfchannel].current_setpoint = rx_buffer_read();
673       tx_buffer_write(motor[frfchannel].current);
674     }
675     else
676     {
677       motor[frfchannel].current_setpoint = 0;
678     }
679   }
680   // compute PI controller based on current setpoint, actual measured current and
        feed forward
681   currentPI(&motor[0]);
682   currentPI(&motor[1]);
683   currentPI(&motor[2]);
684   if(ADC2_CH2 > KILLVOLTAGE)
685   {
686     motor[0].enabled = motor[1].enabled = motor[2].enabled = 0;
687   }
688   // if not in operational state brake or tristate motors
689   if(!motor[0].enabled)
690   {
691     motor[0].cc_integral = 0;
692     motor[0].cc_pwm = 0;
693     if(motor[0].tristate) clear_drvena(DRVENA1);
694   } else set_drvena(DRVENA1);
695   if(!motor[1].enabled)
696   {
697     motor[1].cc_integral = 0;
698     motor[1].cc_pwm = 0;
699     if(motor[1].tristate) clear_drvena(DRVENA2);
700   } else set_drvena(DRVENA2);
701   if(!motor[2].enabled)
702   {
703     motor[2].cc_integral = 0;
704     motor[2].cc_pwm = 0;
705     if(motor[2].tristate) clear_drvena(DRVENA3);
706   } else set_drvena(DRVENA3);
707   if(App.state & APPSTATE_OUTPUT)
708   {
709     update_PWM();
710   }
711   else
712   {
713     clear_PWM();
```

```
714   }
715   interrupt_acknowledge(INTPWM);
716 }
```

Listing 4.6: `i_handler_pwm`

After the current setpoints are retrieved the control efforts are calculated using the `currentPI` function, shown in Listing 4.7. It is the implementation of a proportional-integral (PI) controller with integrator anti-windup together with back-EMF and motor resistance feedforward. First the error is determined as the difference between the setpoint and the measured current, then the integral of the error is calculated as a summation of the error and the previous integral value, after which the value is clipped if the integral exceeds the limit that is set by `ilimit` (anti-windup). Subsequently the control effort is calculated as the error times the proportional gain plus the integral of the error times the integral gain. The PWM signal is then set as the control effort plus the feedforward signal.

```
637 static void currentPI(motor_t *motor)
638 {
639     // current control error
640     motor->cc_error = motor->current_setpoint - motor->current;
641     // integrator, wind up limited if PWM saturates
642     if(!motor->cc_isclipped)
643     {
644       motor->cc_integral = clip32(motor->cc_integral + motor->cc_error, -motor->
          cc_ilimit, motor->cc_ilimit);
645     }
646     // PI controller : control = ((PGAIN * i_err) + (IGAIN * i_i))
647     // clipped at maxvoltage
648     motor->cc_control = (int16_t)clip32(q8mul32(motor->cc_error, motor->cc_pgain)
649                                      + q8mul32((motor->cc_integral >> 6), motor->
                                             cc_igain)
650                                    , -MAXVOLTAGE, MAXVOLTAGE);
651     // transform to pwm duty cycle clipped at MAXCONTROL
652     motor->cc_pwm = (int16_t)clip16(q8mul(motor->cc_control +
653                                          motor->backemf_ff +
654                                          motor->setpoint_ff +
655                                          q8mul(motor->current_setpoint, motor->rm2u)
656                                          , u2pwm), -MAXCONTROL, MAXCONTROL);
657     // clip16 isclipped result is copied in motor struct to prevent integral wind-up
658     motor->cc_isclipped = isclipped;
659 }
```

Listing 4.7: `currentPI`

The last part of the interrupt brakes the motors if the line voltage exceeds the kill voltage value and if the motors are not in enabled mode. The motors are put in tristate if the user enables this.

4.2.2 Encoder Handler

The second interrupt, `i_handler_ec`, runs at a lower frequency of 1 kHz and is shown in Listing 4.8. First the encoder positions and associated timestamps are stored, after which the motor velocities are calculated in the `encoder_calculations` function, shown in Listing 4.9. Also the voltage to PWM conversion is calculated depending on the linevoltage, with the calculation using a minimal value of 10V, since otherwise the PWM pulses will become very long. The reason that this interrupt runs at a lower frequency is that calculating the mechanical quantities at a faster rate is not very useful, since the mechanical time constant of the system is much slower than the electrical one. This also saves calculation time.

```
740 __INTERRUPT_NATIVE void i_handler_ec(void)
741 {
```

```
742     static int ledcnt;
743     icnt++;
744     txpdo1.ectime = htoes(icnt);
745     ENC1_HOLD = 1;
746     motor[0].position   = ENC1_COUNT;
747     motor[0].poscaptime = ENC1_TIMESTAMP;
748     ENC1_HOLD = 0;
749     ENC2_HOLD = 1;
750     motor[1].position   = ENC2_COUNT;
751     motor[1].poscaptime = ENC2_TIMESTAMP;
752     ENC2_HOLD = 0;
753     ENC3_HOLD = 1;
754     motor[2].position   = ENC3_COUNT;
755     motor[2].poscaptime = ENC3_TIMESTAMP;
756     ENC3_HOLD = 0;
757
758     if(wd_cnt) wd_cnt--;
759     encoder_calculations(&motor[0]);
760     encoder_calculations(&motor[1]);
761     encoder_calculations(&motor[2]);
762     // compute feed forward for back emf
763     // voltage = motor_velocity * motor_KV
764     motor[0].backemf_ff = q8mul(motor[0].velocity, motor[0].kvmotor);
765     motor[1].backemf_ff = q8mul(motor[1].velocity, motor[1].kvmotor);
766     motor[2].backemf_ff = q8mul(motor[2].velocity, motor[2].kvmotor);
767     // internal line voltage as integer value in 10mV units
768     linevoltage_i = q8mul(ADC2_CH2, R2U_FP);
769     // only caluculate when linevoltage > 10V
770     if(linevoltage_i > LINEVOLTAGELIMIT)
771     {
772         // u2pwm in q8, u control in 10mV units
773         u2pwm = (Q8f * PWMPER) / linevoltage_i;
774     }
775     else
776     {
777         u2pwm = (Q8f * PWMPER) / LINEVOLTAGELIMIT;
778     }
779     if(ledcnt++ & 0x100) toggle_digout(FRUN);
780     interrupt_acknowledge(INTEC);
781 }
```

Listing 4.8: i_handler_ec

```
637 static inline void encoder_calculations(motor_t *motor)
638 {
639     int32_t delta_pos, delta_time;
640     int32_t velo, part;
641
642     delta_pos = (motor->position - motor->prev_position) * motor->encoder_dir;
643     delta_time = motor->poscaptime - motor->prev_poscaptime;
644     if(delta_time > VELTIMELIMIT)
645     {
646       // velocity calcultation in two parts to keep temporary values in 32bit
647       // !! changing to 64bits or float will engage lib functions that disable
            interrupts
648       // !! thus leading to large jitter in i_handler_pwm
649       part = (int32_t)(CPU_CLK * 2.0f * PI_FLOAT) / motor->encoderincrements;
650       velo = (delta_pos * part * 10) / delta_time;
651     } else velo = 0;
652     // velocity in 0.1rad*s-1 units
653     motor->velocity = (int32_t)velo;
654     motor->prev_position = motor->position;
655     motor->prev_poscaptime = motor->poscaptime;
656 }
```

Listing 4.9: `encoder_calculations`

4.3 Circular Buffer

The firmware can operate in two modes, a normal and an FRF mode. In the latter, setpoints and current measurements need to be buffered to enable a high enough throughput to the master, more on this mode in Section 4.3. These buffers, `rx_buffer` for the setpoints and `tx_buffer` for the current measurements, are implemented as ring buffers in `circbuffer.c`, shown in Listing 4.10.

```
#include <stdint.h>
#include <swplatform.h>
#include "boardconst.h"

#define TX_BUFFER_SIZE  100
#define RX_BUFFER_SIZE  100

int16_t                rx_buffer[RX_BUFFER_SIZE];
int                    rx_wr_index, rx_rd_index;
volatile int           rx_counter;
int                    rx_buffer_overflow;
int16_t                tx_buffer[TX_BUFFER_SIZE];
int                    tx_wr_index, tx_rd_index;
volatile int           tx_counter;

// rx_buffer is ring buffer EtherCAT to PWM interrupt
void rx_buffer_write(int16_t value)
{
  int tmpi;
  rx_buffer[rx_wr_index++] = value;
  if (rx_wr_index == RX_BUFFER_SIZE) rx_wr_index = 0;
  interrupt_disable(INTPWM);
  tmpi = ++rx_counter;
  interrupt_enable(INTPWM);
  if (tmpi == RX_BUFFER_SIZE)
  {
    interrupt_disable(INTPWM);
    rx_counter = 0;
    interrupt_enable(INTPWM);
    rx_buffer_overflow = 1;
  };
}

int16_t rx_buffer_read(void)
{
    int16_t data;
    data = rx_buffer[rx_rd_index];
    rx_buffer[rx_rd_index++] = 0x00;
    if (rx_rd_index == RX_BUFFER_SIZE) rx_rd_index = 0;
    --rx_counter;
    return data;
}

int rxfree(void)
{
    return RX_BUFFER_SIZE - rx_counter;
}

// tx_buffer is ring buffer PWM interrupt generated data to EtherCAT
int16_t tx_buffer_read(void)
{
```

```
    int16_t data;
    data = tx_buffer[tx_rd_index++];
    if (tx_rd_index == TX_BUFFER_SIZE) tx_rd_index = 0;
    interrupt_disable(INTPWM);
    --tx_counter;
    interrupt_enable(INTPWM);
    return data;
}

void tx_buffer_write(int16_t value)
{
    tx_buffer[tx_wr_index++] = value;
    if (tx_wr_index == TX_BUFFER_SIZE) tx_wr_index = 0;
    ++tx_counter;
    if(tx_counter == TX_BUFFER_SIZE)
    {
      tx_counter = 0;
    }
}
```

Listing 4.10: Circular Buffer implementation

4.4 Data Structures

In order to communicate with the master, input and output data structures are defined. As mentioned before, the firmware is implemented such that the TUeES030 slave has two operating modes, normal and FRF, each with their own transmit and receive data structure mapping.

4.4.1 Normal Mode

When the normal mode is active, a large number of variables is transmitted to the master. For each motor is transmitted: the state in which it's in, its encoder count, associated timestamp of the encoder count, velocity and measured current. Apart from the motor related variables, also the 8 bit digital signal, two caliper readings, six analog outputs in the form of the force and position variables, two extra analog readings, the line voltage and encoder timer are sent to the master.

Transmit PDO mapping

```
typedef struct PACKED{
  uint8_t       mstate1;
  uint32_t      count1;
  uint32_t      timestamp1;
  int16_t       velocity1;
  int16_t       current1;
  uint8_t       mstate2;
  uint32_t      count2;
  uint32_t      timestamp2;
  int16_t       velocity2;
  int16_t       current2;
  uint8_t       mstate3;
  uint32_t      count3;
  uint32_t      timestamp3;
  int16_t       velocity3;
  int16_t       current3;
  uint8_t       digital;
  uint16_t      caliper1;
  uint16_t      caliper2;
```

```
  uint16_t      force1;
  uint16_t      force2;
  uint16_t      force3;
  uint16_t      pos1;
  uint16_t      pos2;
  uint16_t      pos3;
  uint16_t      analog1;
  uint16_t      analog2;
  uint16_t      linevoltage;
  uint16_t      ectime;
}txpdo1_t;
```

Listing 4.11: Normal mode transmit PDO mapping

The TUeES030 slave receives a total of 12 variables in the normal mode: a motor command, current setpoint and feedforward signal for each motor as well as a digital input and two analog inputs.

Receive PDO mapping

```
typedef struct PACKED{
  uint8_t       mcom1;
  int16_t       setpoint1;
  int16_t       ff1;
  uint8_t       mcom2;
  int16_t       setpoint2;
  int16_t       ff2;
  uint8_t       mcom3;
  int16_t       setpoint3;
  int16_t       ff3;
  uint8_t       digital;
  int16_t       aout1;
  int16_t       aout2;
}rxpdo1_t;
```

Listing 4.12: Normal mode receive PDO mapping

4.4.2 FRF Mode

The FRF mode needs to sample data at very high frequencies ($\approx$ 20kHz), which is too high for the master to keep up. The TUeES030 slave can run at such high frequencies, but since the master cannot the data that is sampled by the slave needs to be buffered and transmitted as packages to the master. Since this buffer data package containing current setpoints and measurements is large, all non-essential outputs, such as force/positions and measurements from motors other than the one on which a FRF measurement is performed, are not transmitted.

Transmitted are the motor state, amount of measurements stored in the buffer, number of useful entries in the current variable, `ectime` (currently not used) and the current measurements (22).

Transmit PDO mapping

```
typedef struct PACKED{
  uint8_t       mstate;
  uint8_t       buffer;
  uint8_t       entries;
  uint16_t      ectime;
  int16_t       current[FRFBUFFERSIZE];
}txpdo2_t;
```

Listing 4.13: FRF mode transmit PDO mapping

Received from the master are the 8 bit motor command, where bits 0 and 1 are for the selection of motor 1, 2 or 3, bit 2 is for selecting brake (0) or controlled (1) and bit 3 is for enabling (1) or disabling (0) tristate. Also the number of useful entries in the setpoint variable and the current setpoints [22] are received.

Receive PDO mapping

```
typedef struct PACKED{
  uint8_t       mcom;
  uint8_t       entries;
  int16_t       setpoint[FRFBUFFERSIZE];
}rxpdo2_t;
```

Listing 4.14: FRF mode receive PDO mapping

4.4.3 Parameters

Apart from in- and outputs that are constantly sent to and received from the slave, the firmware also incorporates *parameters* that can be set for each board. These parameters are set at the start of running the application in the current Simulink implementation in Section 5.4. The parameters consist of the resistance, speed constant, P gain, I gain, integrator limit, encoder direction, encoder resolution and current zero offset, for each of the maximum of three motors that can be controlled using the board.

```
typedef struct PACKED{
  float         m1r;
  float         m1kv;
  float         m1pgain;
  float         m1igain;
  float         m1ilimit;
  int8_t        m1encdir;
  uint16_t      m1encres;
  int16_t       m1czero;
  float         m2r;
  float         m2kv;
  float         m2pgain;
  float         m2igain;
  float         m2ilimit;
  int8_t        m2encdir;
  uint16_t      m2encres;
  int16_t       m2czero;
  float         m3r;
  float         m3kv;
  float         m3pgain;
  float         m3igain;
  float         m3ilimit;
  int8_t        m3encdir;
  uint16_t      m3encres;
  int16_t       m3czero;
}param_t;
```

Listing 4.15: Parameter structure

4.5 Miscellaneous

In this section several functions that are used throughout the micro-processor code are discussed, since they are important and might not be understood otherwise.

4.5.1 Swap

The functions `htoes`, `htoel`, `htoell`, `htoes2`, `htoel2` and `htoef2` are all used to swap the bits in variables, since the FPGA is big-endian, while the micro-processor is little-endian. The master is usually little-endian as well. The abbreviations stand for Host to Ethercat Short, Long, Long Long and Float, respectively. Whereas the number 2 indicates that the bit swapping is implemented in Verilog code on the FPGA, while the rest are implemented in the C code.

4.5.2 Fixed Point Multiplication

There are two fixed point multiplication functions: `q8mul` and `q8mul32`. The first returns the 16-bit integer result of a 16-bit integer variable multiplication with a 16 integer bit, 8 fractional bits (q16.8) fixed point variable. The second returns a 32-bit integer result of the same multiplication.

4.5.3 Clip

The functions `clip16` and `clip32` saturate a, 16 or 32 bits, variable between specified low and high values. The 32-bits version also returns a boolean stating if the value was clipped or not.

Chapter 5

Simulink Implementation

In this chapter the implementation on the master side is discussed. The master communicates with the slave (TUeES030 board) using the EtherCAT protocol and is currently implemented in MATLAB® Simulink®.

Two Simulink blocks are created to communicate with the TUeES030 board, one for the normal and the other for the FRF measurement mode. First the normal mode block will be discussed extensively by starting from the level the user sees and going deeper into its workings, after which the FRF mode block is discussed in a similar manner but much less detailed, since most of its workings are the same.

5.1 Normal Mode

5.1.1 Simulink Block

In Figure 5.1 the Simulink block for the normal mode is shown in the black border on the left. This is a so called *masked* Simulink block, which allows the user to adapt certain parameters, which are then used by underlying blocks to perform tasks. Masks are used to give the user ability to adapt a block, while on the other hand abstracting the more complicated code away. The inputs and outputs to this block are the receive and transmit PDO mappings from Subsection 4.4.1, respectively. Double-clicking on the block reveals the block parameters (which are set in the mask configuration) as shown in Figure 5.2. Apart from the first two (of which the uses are explained later) the parameters match those in Subsection 4.4.3.

If we look under this mask, we end up 1 level deeper, which is the middle black border. Here an S-function block is connected to a mux and a demux block, which are connected to the inputs and outputs. The S-function block is again a mask, with the level beneath it shown in the right black border. Here the parameters for the S-function block can be specified, which consist of the name of the S-function and the names of its parameters.

5.1.2 S-function

The S-function code is shown in Listing 5.1 and is written in C. It starts with the definition of the S-function name and level, which is standard for an S-function and obligatory. Next, `simstruc.h` is included and the number of states, inputs, outputs and parameters are defined, also standard practice. As can be seen, the S-function (and therefore also the Simulink block) has 12 inputs, 28 outputs and 10 parameters. The first two parameters are then read and are put into defines and a pointer to the input port is created. Following this, two more header files are included, `math.h` for mathematical operations and `ec.h` for EtherCAT functions.

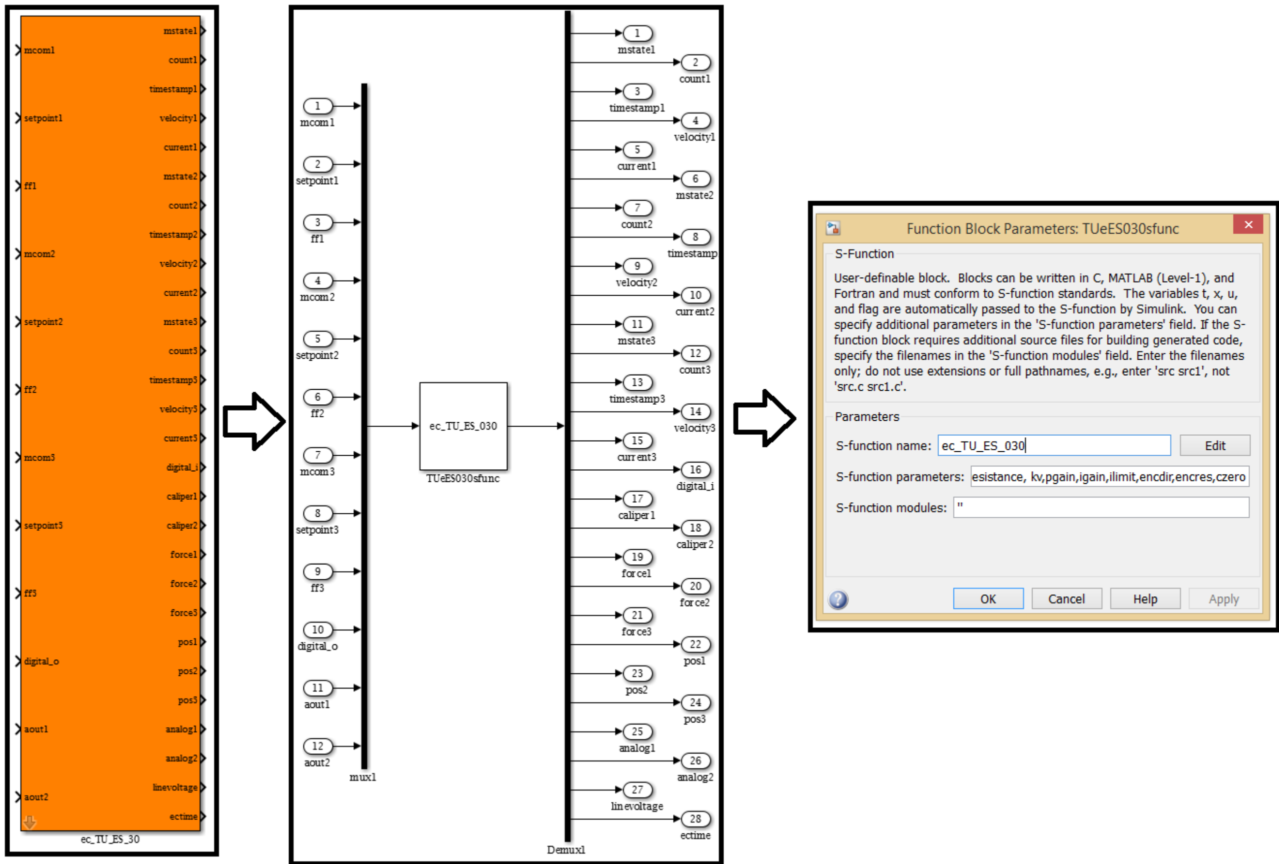


Figure 5.1: Simulink block for the normal mode

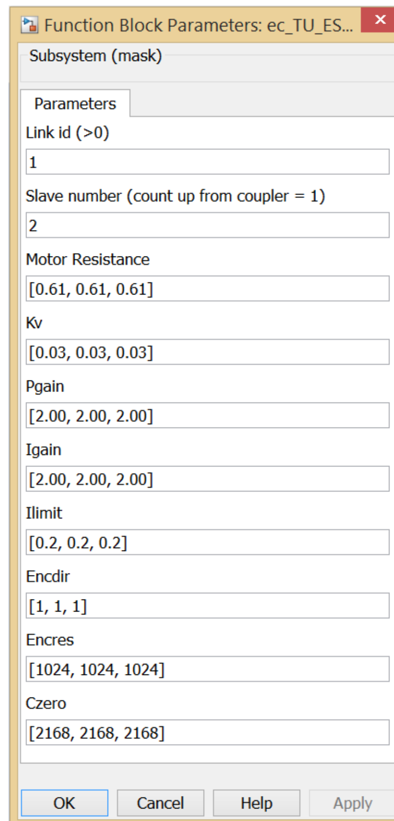


Figure 5.2: Parameters for the normal mode

After these definitions and includes, the code consists of four functions: mdlInitializeSizes, mdlInitializeSampleTimes, mdlOutputs and mdlTerminate, that take care of initialising the sizes of the input/output ports (amongst others), initialising the sample times, calculating the outputs and statements that need to be done upon termination, respectively. These functions are commonly found in S-functions and can best be referred to in the Matlab help library. We will focus on mdlOutputs, since this is where most of the custom code is placed.

It starts with retrieving pointers for the in- and outputs, followed by the retrieval of the LINK_ID and SLAVE_NUM parameters. The first parameter is used to define for which of the TUeES030 boards in the slavestack this S-function block is used. The second parameter is to define the position of the board in the entire stack. Next is an if-loop that only runs on the first time using the firstrun variable, which takes care of setting the board to the correct mode (normal or FRF) using ec_Set_TUeES030mode and writing the parameters to the board using ec_Set_TUeES030params. These functions are discussed in Section 5.3 and 5.4, respectively. Following the initialisation on the first run are the functions that read and write the outputs and inputs in a for-loop each run, ec_TU_ES_030_read_chan and ec_TU_ES_030_write_chan, respectively.

```
1  #define S_FUNCTION_NAME ec_TU_ES_030
2  #define S_FUNCTION_LEVEL 2
3
4  #include "simstruc.h"
5
6  #define NSTATES          0
7  #define NINPUTS          12
8  #define NOUTPUTS         28
9  #define NPARAMS          10
10
11 #define LINK_ID          ssGetSFcnParam(S,0)
12 #define SLAVE_NUM        ssGetSFcnParam(S,1)
13
14 #define U(element) (*uPtrs[element])  /* Pointer to Input Port0 */
15
16 #include <math.h>
17 #include "ec.h"
18
19 int firstrun = 0;
20 /*====================*
21  * S-function methods *
22  *====================*/
23 static void mdlInitializeSizes(SimStruct *S)
24 {
25     ssSetNumSFcnParams(S,NPARAMS);
26
27     ssSetNumContStates(S,NSTATES);
28     ssSetNumDiscStates(S,0);
29
30     if (!ssSetNumInputPorts(S,1)) return;
31     ssSetInputPortWidth(S,0,NINPUTS);
32     ssSetInputPortDirectFeedThrough(S,0,0);
33
34     if (!ssSetNumOutputPorts(S,1)) return;
35     ssSetOutputPortWidth(S,0,NOUTPUTS);
36
37     ssSetNumSampleTimes(S,1);
38     ssSetNumRWork(S,0);
39     ssSetNumIWork(S,0);
40     ssSetNumPWork(S,0);
41     ssSetNumModes(S,0);
42     ssSetNumNonsampledZCs(S,0);
43 }
44
45 static void mdlInitializeSampleTimes(SimStruct *S)
46 {
47     ssSetSampleTime(S,0,CONTINUOUS_SAMPLE_TIME);
```

```c
    ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T *y=ssGetOutputPortRealSignal(S,0);
    InputRealPtrsType uPtrs=ssGetInputPortRealSignalPtrs(S,0);

#ifndef MATLAB_MEX_FILE

    int_T ilink, slavenum, ireadchan, iwritechan;

    ilink= (int_T) (*(mxGetPr(LINK_ID)));
    slavenum= (int_T) (*(mxGetPr(SLAVE_NUM)));

    /*      Write Parameters on first run*/
    if (firstrun == 0) {
        if(ec_Set_TUeES030mode(1,slavenum) != 0) {
            printf("An error ocurred during switching mode of slave %d \n",slavenum);
        }
        int iwriteparam;
        for (iwriteparam=2; iwriteparam<NPARAMS; iwriteparam++) {
            double *params = (double *) (mxGetPr(ssGetSFcnParam(S,iwriteparam)));
            ec_Set_TUeES030params(params, (iwriteparam-1), slavenum);
        }
        firstrun = 1;
    }

    /*      read channels */
    for (ireadchan=0; ireadchan<NOUTPUTS; ireadchan++) {
        ec_TU_ES_030_read_chan(&y[ireadchan], ireadchan, ilink);
    }

    /*      write channels */
    for (iwritechan=0; iwritechan<NINPUTS; iwritechan++) {
        ec_TU_ES_030_write_chan(U(iwritechan), iwritechan, ilink);
    }
#endif
}

static void mdlTerminate(SimStruct *S)
{
}

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

Listing 5.1: `ec_TU_ES_030.c`

5.1.3 Read outputs

In the function `ec_TU_ES_030_read_chan.c`, which can be found in the file `ec.h`, the outputs of the TUeES030 boards are cast from a struct that contains these outputs into a pointer.

```c
int ec_TU_ES_030_read_chan(double *pvalue, int ichan, int ilink)
{
    if ( (ilink<=0) || (ilink>nTU_ES_030) ) {
        printf("ERROR: %s:%d, incorrect link ID, ...
        could not find slave with link ID : %d\n",__FILE__,__LINE__,ilink);
        return -1;
    }
```

```
8
9     switch (ichan){
10        case 0:
11            *pvalue=(double) in_TU_ES_030_[ilink]->mstate1;
12            break;
13        case 1:
14            *pvalue=(double) in_TU_ES_030_[ilink]->count1;
15            break;
16        case 2:
17            *pvalue=(double) in_TU_ES_030_[ilink]->timestamp1;
18            break;
19        case 3:
20            *pvalue=(double) in_TU_ES_030_[ilink]->velocity1;
21            break;
22        case 4:
23            *pvalue=(double) in_TU_ES_030_[ilink]->current1;
24            break;
25        case 5:
26            *pvalue=(double) in_TU_ES_030_[ilink]->mstate2;
27            break;
28        case 6:
29            *pvalue=(double) in_TU_ES_030_[ilink]->count2;
30            break;
31        case 7:
32            *pvalue=(double) in_TU_ES_030_[ilink]->timestamp2;
33            break;
34        case 8:
35            *pvalue=(double) in_TU_ES_030_[ilink]->velocity2;
36            break;
37        case 9:
38            *pvalue=(double) in_TU_ES_030_[ilink]->current2;
39            break;
40        case 10:
41            *pvalue=(double) in_TU_ES_030_[ilink]->mstate3;
42            break;
43        case 11:
44            *pvalue=(double) in_TU_ES_030_[ilink]->count3;
45            break;
46        case 12:
47            *pvalue=(double) in_TU_ES_030_[ilink]->timestamp3;
48            break;
49        case 13:
50            *pvalue=(double) in_TU_ES_030_[ilink]->velocity3;
51            break;
52        case 14:
53            *pvalue=(double) in_TU_ES_030_[ilink]->current3;
54            break;
55        case 15:
56            *pvalue=(double) in_TU_ES_030_[ilink]->digital;
57            break;
58        case 16:
59            *pvalue=(double) in_TU_ES_030_[ilink]->caliper1;
60            break;
61        case 17:
62            *pvalue=(double) in_TU_ES_030_[ilink]->caliper2;
63            break;
64        case 18:
65            *pvalue=(double) in_TU_ES_030_[ilink]->force1;
66            break;
67        case 19:
68            *pvalue=(double) in_TU_ES_030_[ilink]->force2;
69            break;
70        case 20:
71            *pvalue=(double) in_TU_ES_030_[ilink]->force3;
72            break;
73        case 21:
```

```
74                *pvalue=(double) in_TU_ES_030_[ilink]->pos1;
75                break;
76            case 22:
77                *pvalue=(double) in_TU_ES_030_[ilink]->pos2;
78                break;
79            case 23:
80                *pvalue=(double) in_TU_ES_030_[ilink]->pos3;
81                break;
82            case 24:
83                *pvalue=(double) in_TU_ES_030_[ilink]->analog1;
84                break;
85            case 25:
86                *pvalue=(double) in_TU_ES_030_[ilink]->analog2;
87                break;
88            case 26:
89                *pvalue=(double) in_TU_ES_030_[ilink]->linevoltage;
90                break;
91            case 27:
92                *pvalue=(double) in_TU_ES_030_[ilink]->ectime;
93                break;
94        }
95
96        return 0;
97    }
```

Listing 5.2: `ec_TU_ES_030_read_chan.c`

5.1.4 Write inputs

In the function `ec_TU_ES_030_write_chan.c`, which can be found in the file `ec.h`, the outputs of the master are cast from the double values in the Simulink diagram unto a member of a struct that contains these outputs.

```
1   int ec_TU_ES_030_write_chan(double outputvalue, int ichan, int ilink)
2   {
3       if ( (ilink<=0) || (ilink>nTU_ES_030) ) {
4           printf("ERROR: %s:%d, incorrect link ID, ...
5           could not find slave with link ID : %d\n",__FILE__,__LINE__,ilink);
6           return -1;
7       }
8       switch (ichan){
9           case 0:
10              out_TU_ES_030_[ilink]->mcom1 = (uint8) outputvalue;
11              break;
12          case 1:
13              out_TU_ES_030_[ilink]->setpoint1 = (int16) outputvalue;
14              break;
15          case 2:
16              out_TU_ES_030_[ilink]->ff1 = (int16) outputvalue;
17              break;
18          case 3:
19              out_TU_ES_030_[ilink]->mcom2 = (uint8) outputvalue;
20              break;
21          case 4:
22              out_TU_ES_030_[ilink]->setpoint2 = (int16) outputvalue;
23              break;
24          case 5:
25              out_TU_ES_030_[ilink]->ff2 = (int16) outputvalue;
26              break;
27          case 6:
28              out_TU_ES_030_[ilink]->mcom3 = (uint8) outputvalue;
29              break;
30          case 7:
31              out_TU_ES_030_[ilink]->setpoint3 = (int16) outputvalue;
```

```c
                break;
            case 8:
                out_TU_ES_030_[ilink]->ff3 = (int16) outputvalue;
                break;
            case 9:
                out_TU_ES_030_[ilink]->digital = (uint8) outputvalue;
                break;
            case 10:
                out_TU_ES_030_[ilink]->aout1 = (int16) outputvalue;
                break;
            case 11:
                out_TU_ES_030_[ilink]->aout2 = (int16) outputvalue;
                break;

        }

        return 0;
}
```

Listing 5.3: `ec_TU_ES_030_write_chan.c`

5.2 FRF Mode

In the FRF mode the focus is on measuring at a fast rate, so only a single motor is active and only the inputs and outputs to this motor are of importance and the other inputs and outputs are disabled. In this mode the slave runs at 20 kHz, while the master runs at 1 kHz, meaning that at each time step the master must send a packet of 20 setpoints and receive a packet of 20 current measurements from the slave. However, this is only true if the master and slave would be synchronised perfectly, which in practice is almost impossible to achieve. Instead, the slave has two ring-buffers; one for receiving setpoints and one for transmitting current measurements, each containing up to 100 values, see Section 4.3. The slave can receive and send up to a maximum of 22 setpoints and current measurements at each time step, thus allowing for a 10% overcapacity. The job of the master is to keep the buffer full but not overflowing, by changing how many setpoints are sent at each time step, which is indicated by the `entries` variable. Current measurements are sent back at the fastest rate possible, and the `entries_o` variable indicates how many measurements are meaningful at each time step.

5.2.1 Simulink Block

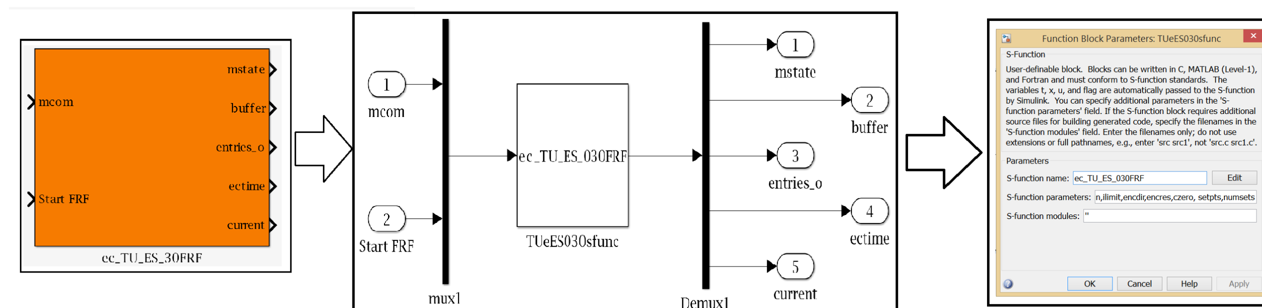


Figure 5.3: Simulink block for the FRF mode

The Simulink block as shown in Figure 5.3 has as inputs the motor command for selecting, enabling and tristating the motor (`mcom`), together with a signal that specifies that the measurement should start (`Start FRF`). The outputs are the state of the selected motor (`mstate`), entries in the buffer (`buffer`), valuable entries in the current array (`entries_o`), time (`ectime`, currently not used) and

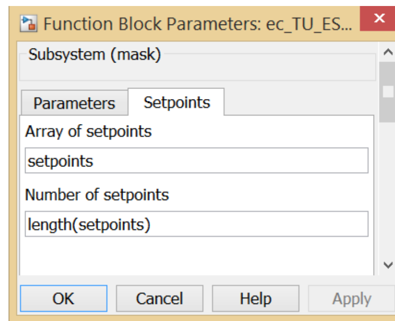


Figure 5.4: Extra parameters for the FRF mode

an array of measured current values (`current`). The parameters that can be specified are the same as those in the normal mode, except that the FRF mode has a second tab that allows the specification of an array of setpoints and the number of total setpoints (easier to specify here than to calculate in C), as shown in Figure 5.4.

5.2.2 S-function

The S-function for the FRF mode, `ec_TU_ES_030FRF.c`, has many similarities with the normal mode, except for the declaration of some more variables in the beginning and extra functionalities in the `mdlOutputs` function. In the `firstrun` if-loop, apart from setting the mode and parameters of the board, the setpoints are retrieved together with the number of setpoints. Also a time estimate for the duration of the measurement is calculated and the setpoints are cast from doubles to integers, saving time later on (since this casting is only done in the beginning, instead of each iteration).

Next the signal to start the FRF measurement, `go`, is read as well as the output channels using the `ec_TU_ES_030FRF_read_chan` function. Then the number of useful current values and number of entries in the buffer are stored. If the start signal is given, the number of setpoints (`entries`) is calculated depending on the number of entries in the buffer. As can be seen, this is done in the manner of a very simple control algorithm to keep the number of entries in the buffer around 50. When all setpoints have been sent to the slave, the number of setpoints is set to zero again. After this the signals are sent using the `ec_TU_ES_030FRF_write_chan` function, explained in Subsection 5.2.4, where the `setcount` makes sure that the correct setpoints are being sent.

The function ends with some calculations on the total current values that have been received (`entries_o_tot`), the current number of setpoints sent (`setcount`) and the number of setpoints that still need to be sent (`setleft`). When all setpoints have been sent, the `done` variable goes to 1 and some statements are printed to the screen so the user can check if everything went correctly. If the number of current values received do not match the number of setpoints sent, then something went wrong (probably a delay occurred, resulting in a lost sample), and the measurement needs to be redone.

```
1  #define S_FUNCTION_NAME ec_TU_ES_030FRF
2  #define S_FUNCTION_LEVEL 2
3
4  #include "simstruc.h"
5
6  #define NSTATES        0
7  #define NINPUTS        2
8  #define NOUTPUTS       26
9  #define NPARAMS        12
10
11 #define LINK_ID        ssGetSFcnParam(S,0)
12 #define SLAVE_NUM      ssGetSFcnParam(S,1)
13 #define SETPOINTS      ssGetSFcnParam(S,10)
14 #define NSETPOINTS     ssGetSFcnParam(S,11)
15
16 #define U(element) (*uPtrs[element])  /* Pointer to Input Port0 */
```

```
17
18  #include <math.h>
19  #include "ec.h"
20
21  int done = 0;
22  int entries = 0;
23  int entries_o_tot = 0;
24  int est_t = 0;
25  int firstrun = 0;
26  int once = 1;
27  int nsets = 0;
28  int setcount = 0;
29  int setleft = 0;
30  int time_once = 0;
31
32  typedef struct {
33      int *isets;
34  } SfunctionGlobalData, *pSfunctionGlobalData;
35
36  /*====================*
37   * S-function methods *
38   *====================*/
39  static void mdlInitializeSizes(SimStruct *S)
40  {
41      ssSetNumSFcnParams(S,NPARAMS);
42
43      ssSetNumContStates(S,NSTATES);
44      ssSetNumDiscStates(S,0);
45
46      if (!ssSetNumInputPorts(S,1)) return;
47      ssSetInputPortWidth(S,0,NINPUTS);
48      ssSetInputPortDirectFeedThrough(S,0,0);
49
50      if (!ssSetNumOutputPorts(S,1)) return;
51      ssSetOutputPortWidth(S,0,NOUTPUTS);
52
53      ssSetNumSampleTimes(S,1);
54      ssSetNumRWork(S, sizeof(SfunctionGlobalData)/sizeof(real_T)+1);
55      ssSetNumIWork(S,0);
56      ssSetNumPWork(S,0);
57      ssSetNumModes(S,0);
58      ssSetNumNonsampledZCs(S,0);
59  }
60
61  static void mdlInitializeSampleTimes(SimStruct *S)
62  {
63      ssSetSampleTime(S,0,CONTINUOUS_SAMPLE_TIME);
64      ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
65  }
66
67  static void mdlOutputs(SimStruct *S, int_T tid)
68  {
69      pSfunctionGlobalData psfgd = (pSfunctionGlobalData) ssGetRWork(S);
70
71      real_T *y=ssGetOutputPortRealSignal(S,0);
72      InputRealPtrsType uPtrs=ssGetInputPortRealSignalPtrs(S,0);
73
74  #ifndef MATLAB_MEX_FILE
75
76      int_T ilink, slavenum, ireadchan, iwritechan;
77
78      ilink= (int_T) (*(mxGetPr(LINK_ID)));
79      slavenum= (int_T) (*(mxGetPr(SLAVE_NUM)));
80
81      /*      Write Parameters on first run*/
82      if (firstrun == 0) {
```

```
83        if(ec_Set_TUeES030mode(2,slavenum) != 0) {
84            printf("An error ocurred during switching mode of slave %d \n",slavenum);
85        }
86        int iwriteparam;
87        for (iwriteparam=2; iwriteparam<10; iwriteparam++) {
88            double *params = (double *) (mxGetPr(ssGetSFcnParam(S,iwriteparam)));
89            ec_Set_TUeES030params(params, (iwriteparam-1), slavenum);
90        }
91        double *sets = (double *) (mxGetPr(SETPOINTS)); //Retrieve array of setpoints
92        nsets = (*(mxGetPr(NSETPOINTS))); //Retrieve total number of setpoints
93        est_t = nsets/20000;
94        psfgd->isets = malloc(nsets*sizeof(int));
95        int ii;
96        for (ii = 0; ii<nsets; ii++) {
97            psfgd->isets[ii] = (int) sets[ii];
98        }
99        setleft = nsets;
100       printf("Number of setpoints: %d \n",nsets);
101       firstrun = 1;
102   }
103   int go = U(1); //Retrieve go signal
104
105   /*      read channel */
106   for (ireadchan=0; ireadchan<NOUTPUTS; ireadchan++) {
107       ec_TU_ES_030FRF_read_chan(&y[ireadchan], ireadchan, ilink);
108   }
109
110   /*      Retrieve number of useful current values */
111   int entries_o = (int) (y[2]);
112
113   /*      Calculate number of entries to motor depending on buffer fill */
114   int buf = (y[1]);
115   if (go) {
116       if(time_once == 0) {
117           printf("Measurement started, estimated time (@20kHz) = %d [s]\n",est_t);
118           time_once = 1;
119       }
120       if (buf < 50) {
121           entries = 22;
122       }
123       else if (buf == 50) {
124           entries = 20;
125       }
126       else if (buf > 50) {
127           entries = 18;
128       }
129       if (setleft<entries) {
130           entries = setleft;
131       }
132   }
133   else {
134       entries = 0;
135   }
136
137   /*      write channel */
138   for (iwritechan=0; iwritechan<24; iwritechan++) {
139       if (iwritechan == 0) {
140           ec_TU_ES_030FRF_write_chan(U(iwritechan), iwritechan, ilink); //write mode
141       }
142       else if (iwritechan == 1) {
143           ec_TU_ES_030FRF_write_chan(entries, iwritechan, ilink); //write entries
144       }
145       else if (((iwritechan-2) < entries) && (go) && (!done)){
146           ec_TU_ES_030FRF_write_chan(psfgd->isets[iwritechan-2+setcount], iwritechan, ilink);
147           //write setpoints
148       }
```

```
149             else {
150                 ec_TU_ES_030FRF_write_chan(0, iwritechan, ilink); //write 0 after setpoints
151             }
152         }
153 
154         entries_o_tot = entries_o_tot+entries_o;
155 
156         if (setcount >= nsets) {
157             done = 1;
158         }
159         setcount = setcount + entries;
160         setleft = nsets-setcount;
161 
162         if((entries_o_tot >= nsets) && (once)){
163             printf("Done!\n");
164             printf("Current count = %d \n",entries_o_tot);
165             printf("Setcount = %d \n",setcount);
166             once = 0;
167         }
168 #endif
169 }
170 
171 static void mdlTerminate(SimStruct *S)
172 {
173     pSfunctionGlobalData psfgd = (pSfunctionGlobalData) ssGetRWork(S);
174 
175     free(psfgd->isets);
176 }
177 
178 #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
179 #include "simulink.c"      /* MEX-file interface mechanism */
180 #else
181 #include "cg_sfun.h"       /* Code generation registration function */
182 #endif
```

Listing 5.4: ec_TU_ES_030FRF.c

5.2.3 Read outputs

The functioning is similar to that of the normal mode, see Subsection 5.1.3.

```
1  int ec_TU_ES_030FRF_read_chan(double *pvalue, int ichan, int ilink)
2  {
3      if ( (ilink <=0) || (ilink >nTU_ES_030FRF) ) {
4          printf("ERROR: %s:%d, incorrect link ID, could not find slave with link ID :...
5           %d\n",__FILE__,__LINE__,ilink);
6          return -1;
7      }
8  
9      switch (ichan){
10         case 0:
11             *pvalue=(double) in_TU_ES_030FRF_[ilink]->mstate;
12             break;
13         case 1:
14             *pvalue=(double) in_TU_ES_030FRF_[ilink]->buffer;
15             break;
16         case 2:
17             *pvalue=(double) in_TU_ES_030FRF_[ilink]->entries;
18             break;
19         case 3:
20             *pvalue=(double) in_TU_ES_030FRF_[ilink]->ectime;
21             break;
22         case 4:
23         case 5:
```

```
24          case 6:
25          case 7:
26          case 8:
27          case 9:
28          case 10:
29          case 11:
30          case 12:
31          case 13:
32          case 14:
33          case 15:
34          case 16:
35          case 17:
36          case 18:
37          case 19:
38          case 20:
39          case 21:
40          case 22:
41          case 23:
42          case 24:
43          case 25:
44              *pvalue=(double) in_TU_ES_030FRF_[ilink]->current[ichan-4];
45              break;
46      }
47      return 0;
48  }
```

Listing 5.5: `ec_TU_ES_030FRF_read_chan.c`

5.2.4 Write inputs

The functioning is similar to that of the normal mode, see Subsection 5.1.4.

```
 1  int ec_TU_ES_030FRF_write_chan(int outputvalue, int ichan, int ilink)
 2  {
 3      if ( (ilink<=0) || (ilink>nTU_ES_030FRF) ) {
 4          printf("ERROR: %s:%d, incorrect link ID, could not find slave with link ID :...
 5           %d\n",__FILE__,__LINE__,ilink);
 6          return -1;
 7      }
 8      switch (ichan){
 9          case 0:
10              out_TU_ES_030FRF_[ilink]->mcom = (uint8) outputvalue;
11              break;
12          case 1:
13              out_TU_ES_030FRF_[ilink]->entries = (uint8) outputvalue;
14              break;
15          case 2:
16          case 3:
17          case 4:
18          case 5:
19          case 6:
20          case 7:
21          case 8:
22          case 9:
23          case 10:
24          case 11:
25          case 12:
26          case 13:
27          case 14:
28          case 15:
29          case 16:
30          case 17:
31          case 18:
32          case 19:
```

```
33          case 20:
34          case 21:
35          case 22:
36          case 23:
37              out_TU_ES_030FRF_[ilink]->setpoint[ichan-2] = (int16) outputvalue;
38              break;
39      }
40      return 0;
41  }
```

Listing 5.6: `ec_TU_ES_030FRF_write_chan.c`

5.3 Set mode

As mentioned previously, the firmware on the TUeES030 boards is programmed in such a way that it can operate in two modes, the normal operating mode and the FRF measurement mode. These two modes have different input and output data structures, as mentioned in Section 4.4. Switching between the modes is done by setting the addresses 0x1C12:01 and 0x1C13:01 to 0x1600 and 0x1A00 for normal mode and 0x1601 and 0x1A01 for FRF mode, respectively. The EtherCAT protocol specifies that these addresses can only be changed in pre-operational mode and only when addresses 0x1C12:00 and 0x1C13:00 are set to zero.

All this is done in the function `ec_Set_TUeES030mode`, which can be found in the file `ec.c` and is shown in Listing 5.7. The function receives the inputs `mode` and `slavenum`, where the former specifies which mode is required (1 for normal or 2 for FRF) and the latter specifies which slave in the stack needs to be configured. It then does everything as described above using the functions `ec_SDOread` and `ec_SDOwrite` to read and write to the addresses, together with some other function to write and check the state of the slave, which will not be discussed further.

```
1   int ec_Set_TUeES030mode(int mode, int slavenum)
2   {
3       uint16 index1, index2;
4       uint8  off = 0, on = 1;
5       int os_write_ind = sizeof(index1), os_write = sizeof(off);
6
7       int data1 = 0, data2 = 0;
8       int os_read = sizeof(data1);
9
10      if (mode == 1){
11          index1 = 0x1600;
12          index2 = 0x1A00;
13      }
14      else if (mode == 2){
15          index1 = 0x1601;
16          index2 = 0x1A01;
17      }
18
19      /* Check if slave already in correct mode */
20      ec_SDOread(slavenum,0x1C12,1,FALSE,&os_read,&data1,EC_TIMEOUTRXM);
21      ec_SDOread(slavenum,0x1C13,1,FALSE,&os_read,&data2,EC_TIMEOUTRXM);
22      if ((data1 == index1) && (data2 == index2)) {
23          printf("Slave %d is already in the correct mode \n",slavenum);
24          return 0;
25      } else {
26          if (mode == 1){
27              printf("Switching to normal mode \n");
28          }
29          if (mode == 2){
30              printf("Switching to FRF mode \n");
31          }
32
```

```
33          /*Switch to PRE-OP state */
34          ec_slave[0].state = EC_STATE_PRE_OP;
35          /* request PRE-OP state for slave */
36          ec_writestate(0);
37          /* wait for slave to reach OP state */
38          ec_statecheck(0, EC_STATE_PRE_OP,  EC_TIMEOUTSTATE);
39          if (ec_slave[0].state != EC_STATE_PRE_OP )
40          {
41              printf("Slave %d did not reach pre-operational state for switching PDO...
42               mapping \n",slavenum);
43              return -1;
44          }
45
46          /*Switch PDO mapping according to EtherCAT protocol*/
47          ec_SDOwrite(slavenum,0x1C12,0,FALSE,os_write,&off,EC_TIMEOUTRXM);
48          ec_SDOwrite(slavenum,0x1C12,1,FALSE,os_write_ind,&index1,EC_TIMEOUTRXM);
49          ec_SDOwrite(slavenum,0x1C12,0,FALSE,os_write,&on,EC_TIMEOUTRXM);
50
51          ec_SDOwrite(slavenum,0x1C13,0,FALSE,os_write,&off,EC_TIMEOUTRXM);
52          ec_SDOwrite(slavenum,0x1C13,1,FALSE,os_write_ind,&index2,EC_TIMEOUTRXM);
53          ec_SDOwrite(slavenum,0x1C13,0,FALSE,os_write,&on,EC_TIMEOUTRXM);
54
55          /*Remap and initialise slave */
56          ec_config(TRUE, &IOmap);
57
58          /*Switch to SAFE-OP state */
59          ec_slave[0].state = EC_STATE_SAFE_OP;
60          /* request SAFE-OP state for slave */
61          ec_writestate(0);
62          /* wait for slave to reach OP state */
63          ec_statecheck(0, EC_STATE_SAFE_OP,  EC_TIMEOUTSTATE);
64          if (ec_slave[0].state != EC_STATE_SAFE_OP )
65          {
66              printf("Slave %d did not reach safe-operational state after switching PDO...
67               mapping \n",slavenum);
68              return -1;
69          }
70
71          /*Switch to OP state */
72          ec_slave[0].state = EC_STATE_OPERATIONAL;
73          /* request OP state for slave */
74          ec_writestate(0);
75          /* wait for slave to reach OP state */
76          ec_statecheck(0, EC_STATE_OPERATIONAL,  EC_TIMEOUTSTATE);
77          if (ec_slave[0].state != EC_STATE_OPERATIONAL )
78          {
79              printf("Slave %d did not reach operational state after switching PDO...
80               mapping \n",slavenum);
81              return -1;
82          }
83
84          /* Verify if slave is in correct mode */
85          ec_SDOread(slavenum,0x1C12,1,FALSE,&os_read,&data1,EC_TIMEOUTRXM);
86          ec_SDOread(slavenum,0x1C13,1,FALSE,&os_read,&data2,EC_TIMEOUTRXM);
87          if ((data1 == index1) && (data2 == index2)) {
88              if (mode == 1) {
89                  printf("Slave %d switched to normal mode \n",slavenum);
90              }
91              else if (mode == 2) {
92                  printf("Slave %d switched to FRF mode \n",slavenum);
93              }
94              return 0;
95          } else {
96              printf("Slave %d did not switch mode \n",slavenum);
97              return -1;
98          }
```

```
99        }
100  }
```

Listing 5.7: ec_Set_TUES030

5.4 Set parameters

Each TUeES030 board can control up to three motors at once, with each motor having its own characteristics. For this reason the parameters can be adjusted separately for each motor. The parameters consist of the motor resistance `r`, motor speed constant `kv`, proportional (P) gain `pgain`, integral (I) gain `igain`, integral limit `ilimit`, encoder direction `encdir`, encoder resolution `encres` and current zero offset `czero`.

The function `ec_Set_TUeES030params` receives as inputs: a pointer to a double array with the parameters of a certain type (for instance an array of resistances), an integer that specifies which of eight types of parameters is specified and an integer that indicates the slavenumber of the TUeES030 board in the stack of which the parameters need to be adjusted. The function casts the parameter to the correct type and passes is to the slave using `ec_SDOwrite` and is read back and printed to be checked by the user to ensure that the parameters are indeed adjusted.

```
1   int ec_Set_TUeES030params(double *m1_param, int iwriteparam, int slavenum)
2   {
3       int os_read, os_write;
4
5       float       r;
6       float       kv;
7       float       pgain;
8       float       igain;
9       float       ilimit;
10      int8        encdir;
11      uint16      encres;
12      int16       czero;
13
14      float       data;
15      int8        data_i8;
16      uint16      data_ui16;
17      int16       data_i16;
18
19      int motors[3] = {0x8000, 0x8001, 0x8002};
20      int i;
21      for (i = 0; i <3; i++) {
22          switch (iwriteparam){
23          case 1:
24              r = (float) m1_param[i];
25
26              os_read = sizeof(data);
27              os_write = sizeof(r);
28              ec_SDOwrite(slavenum, motors[i], iwriteparam, FALSE, os_write, &r, EC_TIMEOUTRXM);
29              ec_SDOread(slavenum, motors[i], iwriteparam, FALSE, &os_read, &data, EC_TIMEOUTRXM);
30              printf("R motor(%d) = %f \t", i, data);
31              break;
32          case 2:
33              kv = (float) m1_param[i];
34
35              os_read = sizeof(data);
36              os_write = sizeof(kv);
37              ec_SDOwrite(slavenum, motors[i], iwriteparam, FALSE, os_write, &kv, EC_TIMEOUTRXM);
38              ec_SDOread(slavenum, motors[i], iwriteparam, FALSE, &os_read, &data, EC_TIMEOUTRXM);
39              printf("Kv motor(%d) = %f \t", i, data);
40              break;
41          case 3:
```

```
42                pgain = (float) m1_param[i];
43
44                os_read = sizeof(data);
45                os_write = sizeof(pgain);
46                ec_SDOwrite(slavenum,motors[i],iwriteparam,FALSE,os_write,&pgain,EC_TIMEOUTRXM);
47                ec_SDOread(slavenum,motors[i],iwriteparam,FALSE,&os_read,&data,EC_TIMEOUTRXM);
48                printf("pgain motor(%d) = %f \t",i,data);
49                break;
50            case 4:
51                igain = (float) m1_param[i];
52
53                os_read = sizeof(data);
54                os_write = sizeof(igain);
55                ec_SDOwrite(slavenum,motors[i],iwriteparam,FALSE,os_write,&igain,EC_TIMEOUTRXM);
56                ec_SDOread(slavenum,motors[i],iwriteparam,FALSE,&os_read,&data,EC_TIMEOUTRXM);
57                printf("igain motor(%d) = %f \t",i,data);
58                break;
59            case 5:
60                ilimit = (float) m1_param[i];
61
62                os_read = sizeof(data);
63                os_write = sizeof(ilimit);
64                ec_SDOwrite(slavenum,motors[i],iwriteparam,FALSE,os_write,&ilimit,EC_TIMEOUTRXM);
65                ec_SDOread(slavenum,motors[i],iwriteparam,FALSE,&os_read,&data,EC_TIMEOUTRXM);
66                printf("ilimit motor(%d) = %f \t",i,data);
67                break;
68            case 6:
69                encdir = (int8) m1_param[i];
70
71                os_read = sizeof(data_i8);
72                os_write = sizeof(encdir);
73                ec_SDOwrite(slavenum,motors[i],iwriteparam,FALSE,os_write,&encdir,EC_TIMEOUTRXM);
74                ec_SDOread(slavenum,motors[i],iwriteparam,FALSE,&os_read,&data_i8,EC_TIMEOUTRXM);
75                printf("Encdir motor(%d) = %d \t",i,data_i8);
76                break;
77            case 7:
78                encres = (uint16) m1_param[i];
79
80                os_read = sizeof(data_ui16);
81                os_write = sizeof(encres);
82                ec_SDOwrite(slavenum,motors[i],iwriteparam,FALSE,os_write,&encres,EC_TIMEOUTRXM);
83                ec_SDOread(slavenum,motors[i],iwriteparam,FALSE,&os_read,&data_ui16,EC_TIMEOUTRXM);
84                printf("Encres motor(%d) = %d \t",i,data_ui16);
85                break;
86            case 8:
87                czero = (int16) m1_param[i];
88
89                os_read = sizeof(data_i16);
90                os_write = sizeof(czero);
91                ec_SDOwrite(slavenum,motors[i],iwriteparam,FALSE,os_write,&czero,EC_TIMEOUTRXM);
92                ec_SDOread(slavenum,motors[i],iwriteparam,FALSE,&os_read,&data_i16,EC_TIMEOUTRXM);
93                printf("Czero motor(%d) = %d \t",i,data_i16);
94                break;
95        }
96    }
97    printf("\n \n");
98    return 0;
99 }
```

Listing 5.8: ec_Set_TUeES030params

Chapter 6

Experimental Results

In order to determine if the new firmware performs as desired, several experiments are performed. First the system is identified using frequency response measurements and the functioning of the feedback loop is determined. In the second part of the chapter the influence of the feedforward signal is investigated.

6.1 Modelling

The firmware for the TUeES030 is designed to operate up to three brushed direct current (DC) motors in current control mode, using a combination of feedback and feedforward. For all the experiments a Maxon RE30 310007 motor in combination with a Maxon HEDL 5540 110512 encoder was used.

In Figure 6.1 a common representation of a control scheme using feedback and feedforward is shown, where r is the reference, e is the error, u is the control output, d is the control disturbance, y is the plant output, η is the measurement noise, C is the feedback controller, H is the plant and FF is the feedforward controller. Since we implemented the feedback and feedforward controllers, we assume for the moment that these are known and we will check later if they behave as expected. We want to identify the plant, which has as input the PWM signal generated in the FPGA and has as output the current measurement going into the analog input, so after the low pass filter. This means the plant consists of the H-bridge, motor and low-pass current measurement filter.

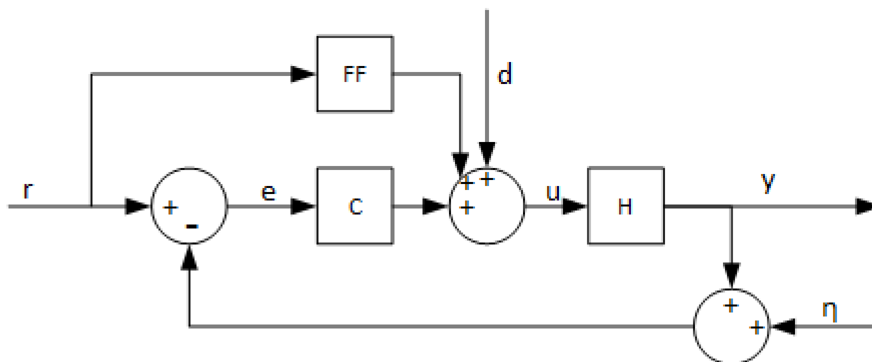


Figure 6.1: Feedback/feedforward control scheme

In order to derive a simple model of the plant, we assume that the PWM signal results in an average voltage coming from the H-bridge being supplied to the motor and that the PWM frequency, currently 80 kHz, is high enough to neglect the dynamics of the H-bridge. For the motor a model can be made

for the electronic part:

$$u = iR + L\frac{\mathrm{d}i}{\mathrm{d}t} + K_v\omega \tag{6.1}$$

where u is the voltage across the motor, i is the current, R is the terminal resistance, L is the terminal inductance, K_v is the speed constant and ω is the shaft rotation speed. Since we are interested in the transfer function from u to i, we need an expression for ω. Luckily it is known that the mechanical torque can be approximated well by:

$$\tau = J\dot{\omega} + b\omega \tag{6.2}$$

where τ is the torque, J is the rotor inertia and b is the damping. Also the torque is directly linked to the current via

$$\tau = K_v i \tag{6.3}$$

If we transform (6.2) to the Laplace domain, rewrite and substitute (6.3)we get the following expression for Ω:

$$\Omega = \frac{T}{Js+b} = \frac{K_v I}{Js+b} \tag{6.4}$$

This expression can be substituted in (6.1) to arrive at an expression for the motor:

$$\begin{aligned} U &= IR + LsI + K_v\Omega = IR + LsI + K_v\frac{K_v I}{Js+b} \\ &= I\left(Ls + R + \frac{K_v^2}{Js+b}\right) \\ M &= \frac{I}{U} = \frac{1}{\left(Ls + R + \frac{K_v^2}{Js+b}\right)} = \frac{Js+b}{LJs^2 + (Lb+RJ)s + Rb + K_v^2} \end{aligned} \tag{6.5}$$

Now that we have a model for the motor, the low-pass filter needs to be modelled as well. Similar to most electric low pass filter, it can be modelled as:

$$LP = \frac{1}{R_f C_f s + 1} \tag{6.6}$$

where $R_f = 1\text{k}\Omega$ is the filter resistance and $C_f = 10\text{nF}$ is the filter capacitance. The components of the plant are now modelled and combining them results in an expression for the plant:

$$H = M \cdot LP = \frac{Js+b}{LJs^2 + (Lb+RJ)s + Rb + K_v^2}\frac{1}{R_f C_f s + 1} \tag{6.7}$$

6.2 Frequency domain analysis

Now that we have a model of the plant, experiments need to be performed in order to check whether the system performs as expected. Looking at Figure 6.1 again, we can only set the reference, the feedforward and feedback blocks and measure the output. This makes open loop measurements possible, by switching off the feedback control and using the feedforward block to directly feedthrough the reference to the plant. Unfortunately, this resulted in poor measurements with a lot of noise. For this reason we want to do closed loop measurements, in the hope that this yields better results. Normally a closed loop measurement is done using either the two or three point method, but since we cannot inject the noises d or η nor measure e and u, these are note possible. By following the lines in the block and converting all signals to the frequency domain using the Fast Fourier Transform (FFT) we can find out what the relation between y and r gives us:

$$Y(f) = H(f)U(f) = H(f)(D(f) + FF(f)R(f) + C(f)E(f)) \tag{6.8}$$

$$= H(f)(D(f) + FF(f)R(f) + C(f)(R(f) - Y(f) - N(f)) \tag{6.9}$$

$$= D(f)H(f) + FF(f)H(f)R(f) + C(f)H(f)R(f) - C(f)H(f)Y(f) - C(f)H(f)N(f) \tag{6.10}$$

to simplify, we switch of the feedforward block. Also, we multiply the whole equation with the complex conjugate of the FFT of the reference, denoted as $R^*(f)$.

$$\begin{aligned} Y(f)R^*(f) = H(f)D(f)R^*(f) + C(f)H(f)R(f)R^*(f)... \\ - C(f)H(f)Y(f)R^*(f) - C(f)H(f)N(f)R^*(f) \end{aligned} \tag{6.11}$$

here we see expressions for the auto power spectral density (APSD) of the input signal, $S_{RR}(f) = R(f)R^*(f)$, and the cross power spectral densities (CPSDs) between several signals, which we be denoted as $S_{YR}(f) = Y(f)R^*(f)$ for the CPSD between the output and the input, for instance. Now we can rewrite the above expression further into

$$S_{YR}(f) = H(f)S_{DR}(f) + C(f)H(f)S_{RR}(f) - C(f)H(f)S_{YR}(f) - C(f)H(f)S_{NR}(f) \tag{6.12}$$

$$= \frac{C(f)H(f)}{1 + C(f)H(f)}S_{RR}(f) + \frac{H(f)}{1 + C(f)H(f)}S_{DR}(f) - \frac{C(f)H(f)}{1 + C(f)H(f)}S_{NR}(f)) \tag{6.13}$$

Now, this expression still is not off any immediate use. However, the PSD between the two signals gives a measure of how much power two signals have in common. Also, if two signals $A(f)$ and $B(f)$ are uncorrelated, then $S_{AB}(f) \approx 0$. This means that if we choose are input signal $r(t)$ as white noise, we have that

$$S_{DR}(f) \approx 0, \qquad S_{NR}(f) \approx 0 \tag{6.14}$$

and so

$$T(f) = \frac{C(f)H(f)}{1 + C(f)H(f)} \approx \frac{S_{YR}(f)}{S_{RR}(f)} \tag{6.15}$$

where $T(f)$ is known as the *complementary sensitivity.*

Now that we know which signals to apply to and measure from the set-up, we can start experimenting. Before moving on, it is mentioned that the *coherence function* between two signals gives a measure of how correlated these two signals are. In short, a coherence of (close to) 1 means that two signals are related linearly and there is little noise present. Why this is the case will not be explained, for this and further reference about this section it is suggested to review the slides of the Motion Control course.

6.2.1 Measurements

To measure the complementary sensitivity as mentioned above, a controller has to be implemented on the TUeES030 board. As mentioned in Subsection 4.2.1 a PI controller is programmed on the micro-processor of the FPGA. For all measurements in this section, the P gain is set to 6.0, the I gain to 4.0 and the integrator limit to 1.0 and the motor axis is allowed to rotate freely. Five measurements are performed with different input signals: a swept sine with low power (chirp low), swept sine with high power (chirp low) and three white Gaussian noise (WGN) measurements with increasing power (1, 2, 3). The resulting complementary sensitivities for the five input signals are shown in Figure 6.2. Here we can see that the WGN measurements are more reliable for higher frequencies. Also, there is very little noise present over a broad frequency range in all measurements, which is an indication that the hardware and software work very well.

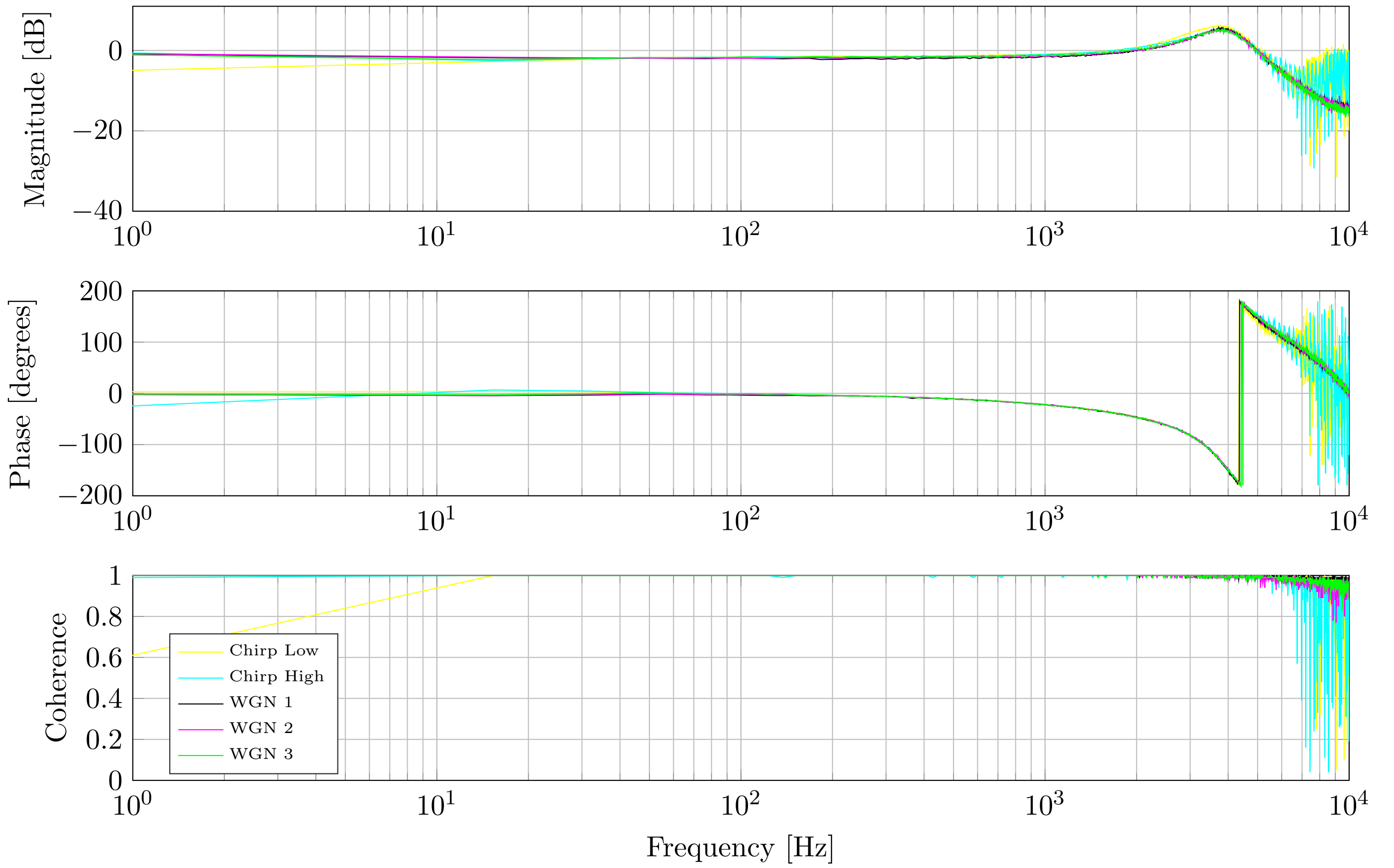


Figure 6.2: Complementary Sensitivity with several test signals

Now that we have several good measurements, we are more interested in the loop return ratio (LRR), to see if the controller resulted in a stable closed loop system and which bandwidth was achieved. For this, the LRR can be calculated directly from the complementary sensitivity:

$$LRR(f) = C(f)H(f) = \frac{T(f)}{1 - T(f)} \tag{6.16}$$

The LRR is shown in Figure 6.3, where it can be seen that, with the controller settings as mentioned above, a bandwidth of about 2.5 kHz with a phase margin of around 60 degrees is achieved. Perhaps the controller can be adjusted such that a higher bandwidth can be achieved, this will be investigated later on.

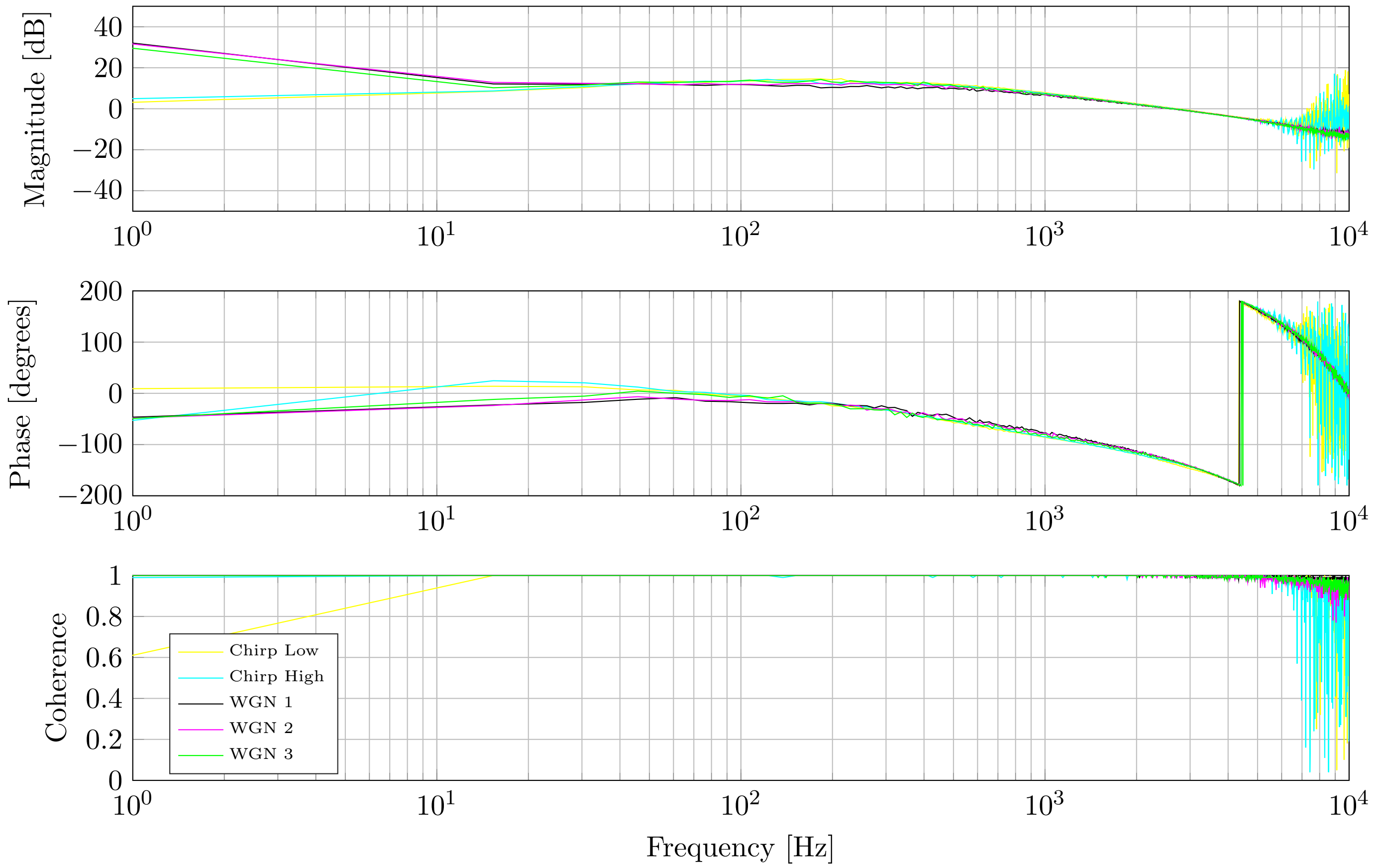


Figure 6.3: Loop return ratio with several test signals

Now that we have the LRR and since the controller that was applied is known, the plant is derived as

$$H(f) = \frac{LRR(f)}{C(f)} \tag{6.17}$$

This is shown in Figure 6.4, together with the plant model derived in Section 6.1. Note that the phase for the plant model is shown with the phase delay of sampling at 20 kHz taken into account.

From the figure it becomes clear that the model of the motor and board matches the measurement very well. This means that the assumption that the H-bridge dynamics could be neglected is valid. Only for very low frequencies (< 50 Hz) the measurements do not match the model, which is not a problem since the interest in current control lies at much higher frequencies and the plant is stable at low frequencies.

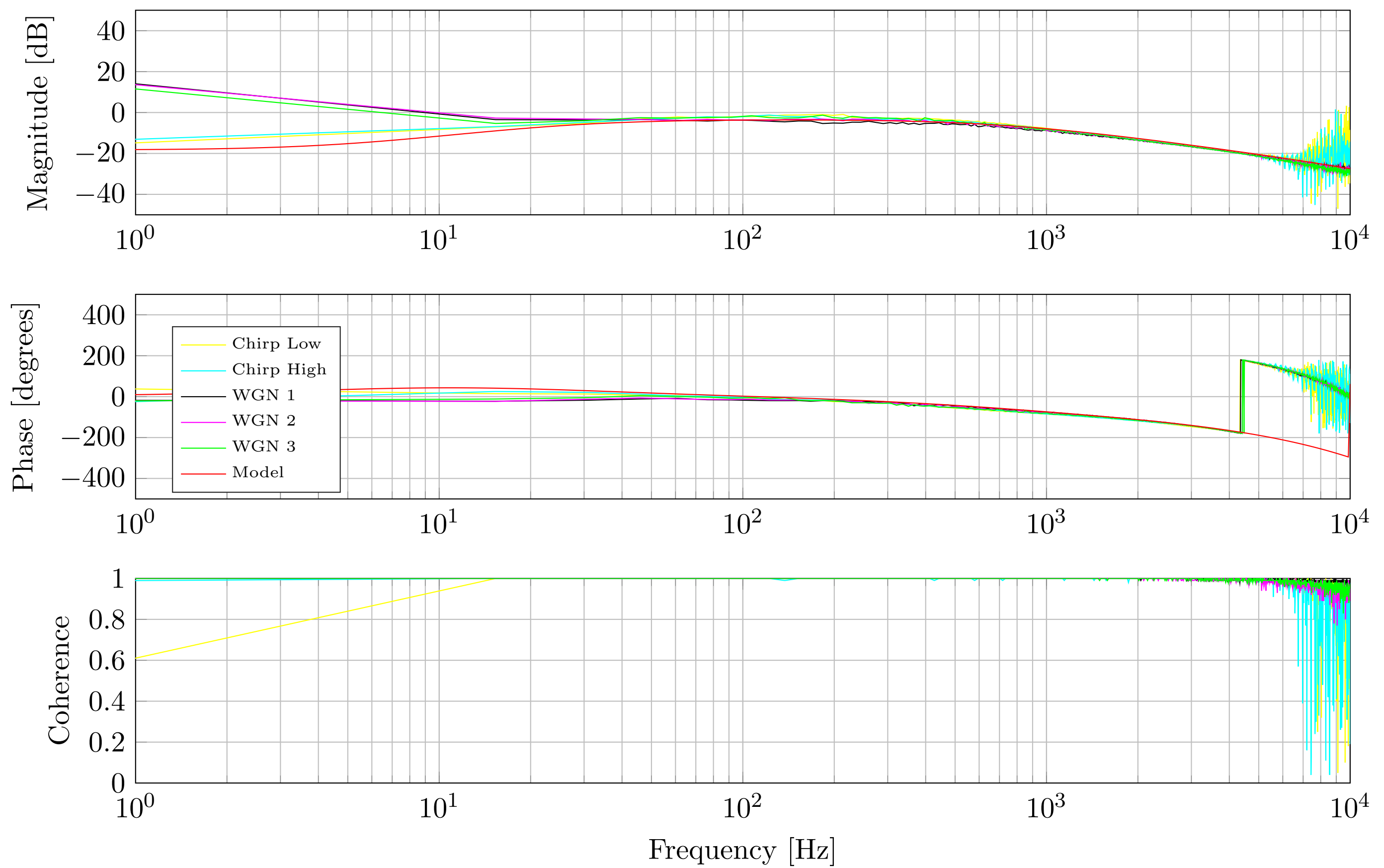


Figure 6.4: Plant with several test signals

6.3 Time domain analysis

In this section the influence of each of the controller parameters on the response of the plant in the time domain is investigated. A pulse signal is applied to the motor, with the motor axis fixated. Each of the following figures shows a single part of this pulse signal, so it can be seen as a step response of the system. Starting with Figure 6.5, where the influence of the proportional gain can be seen to act as expected, a higher gain results in better steady-state tracking, but also in a higher overshoot. A proportional gain of 10 already resulted in an unstable system, so care should be taken to not set it to high.

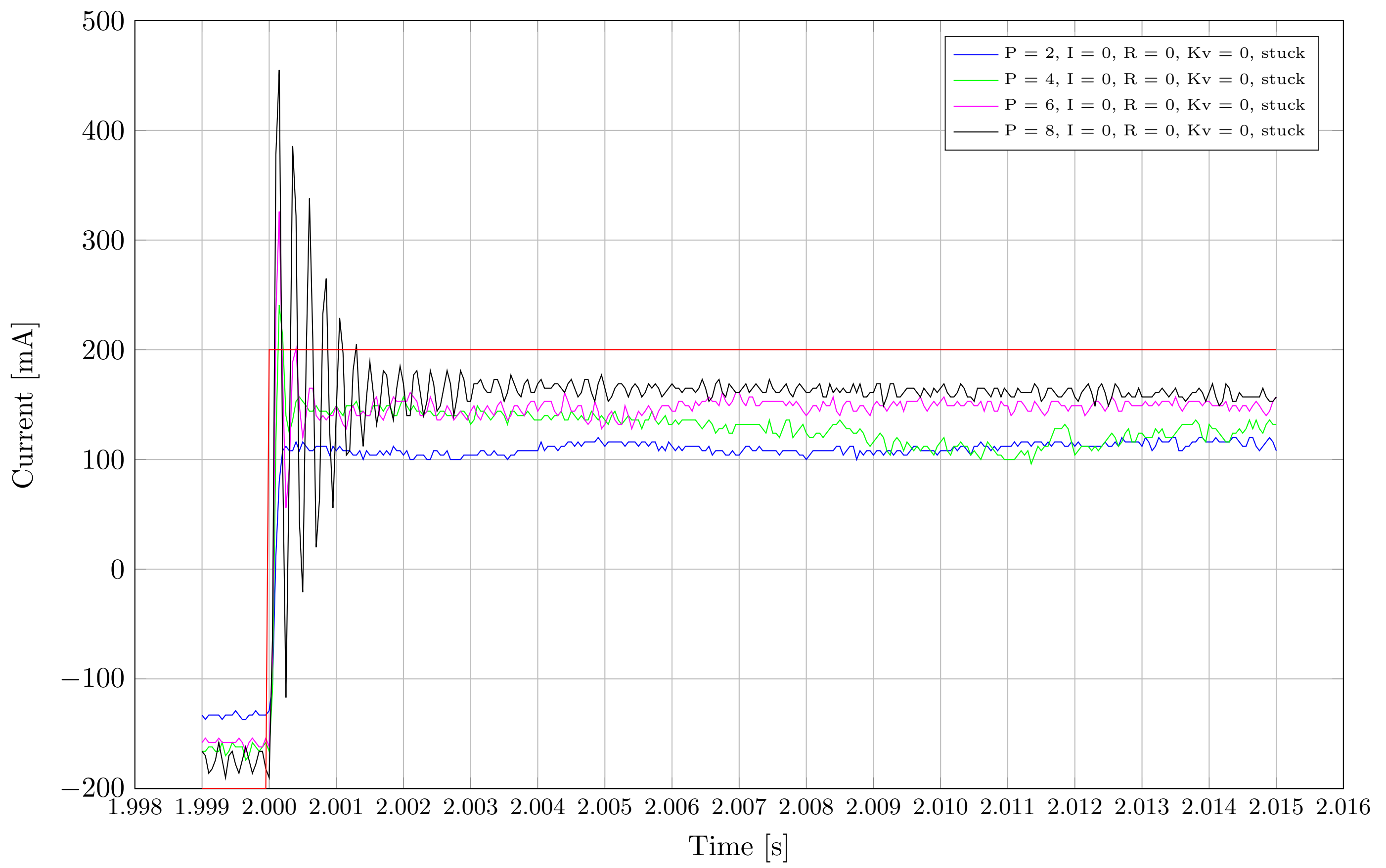


Figure 6.5: Step response for different proportional gains

Moving on the to the integral gain influence shown in Figure 6.6, it can be seen that a higher gain results in a faster response. The effect is similar to that of the proportional gain, expect that the integral effect of removing the steady state error is evident.

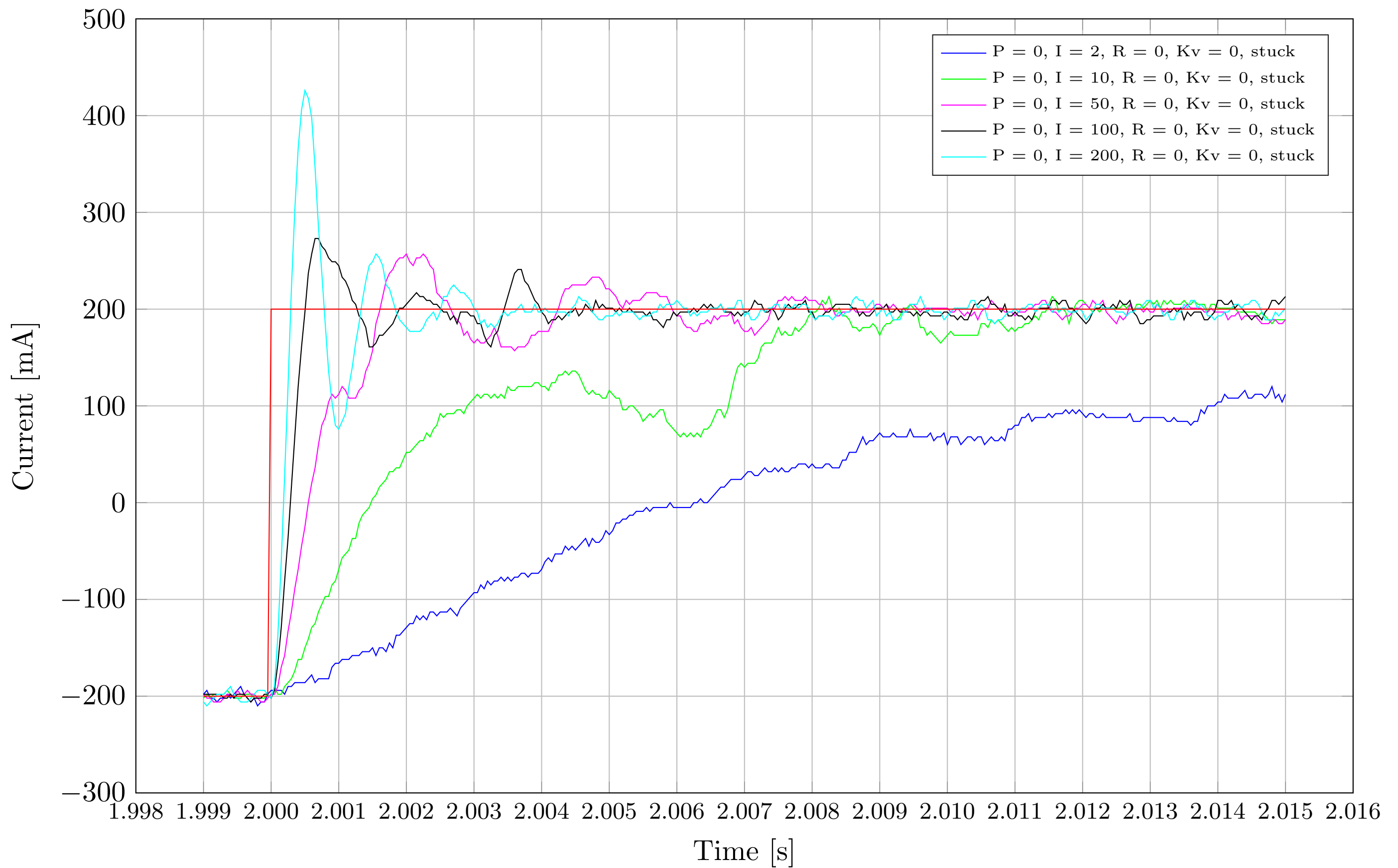


Figure 6.6: Step response for different integral gains

6.3.1 Feedforward parameters

Apart from a feedback loop consisting of a PI controller a feedforward compensation is also implemented on the firmware, see Sections 4.2.1 and 4.2.2. These feedforward signals compensate for part of Equation 6.1, stated again here for convenience:

$$u = iR + L\frac{\mathrm{d}i}{\mathrm{d}t} + K_v\omega$$

The motor resistance voltage drop, iR, and back electromechanical force (back EMF), $K_v\omega$, are compensated, with the resistance, R, and speed constant, K_v, being adjustable to handle different motors. The voltage drop due to the inductance of the motor coils, $L\frac{\mathrm{d}i}{\mathrm{d}t}$, is not compensated, since calculating the derivative of the current requires it to be exactly known beforehand, which is difficult. Also, the motors that can be controlled with the TUeES030 board are small and have a small inductance, so the voltage drop due to the inductance is assumed negligible and the feedback controller can compensate for this.

In Figure 6.7 the effect of the motor resistance feedforward compensation is shown with a constant low proportional action of 0.2 and an increasing resistance. As expected, increasing the resistance value results in a higher current, with the light blue line with a value of $R = 1.5$ being the expected resistance of the motor used in this experiment.

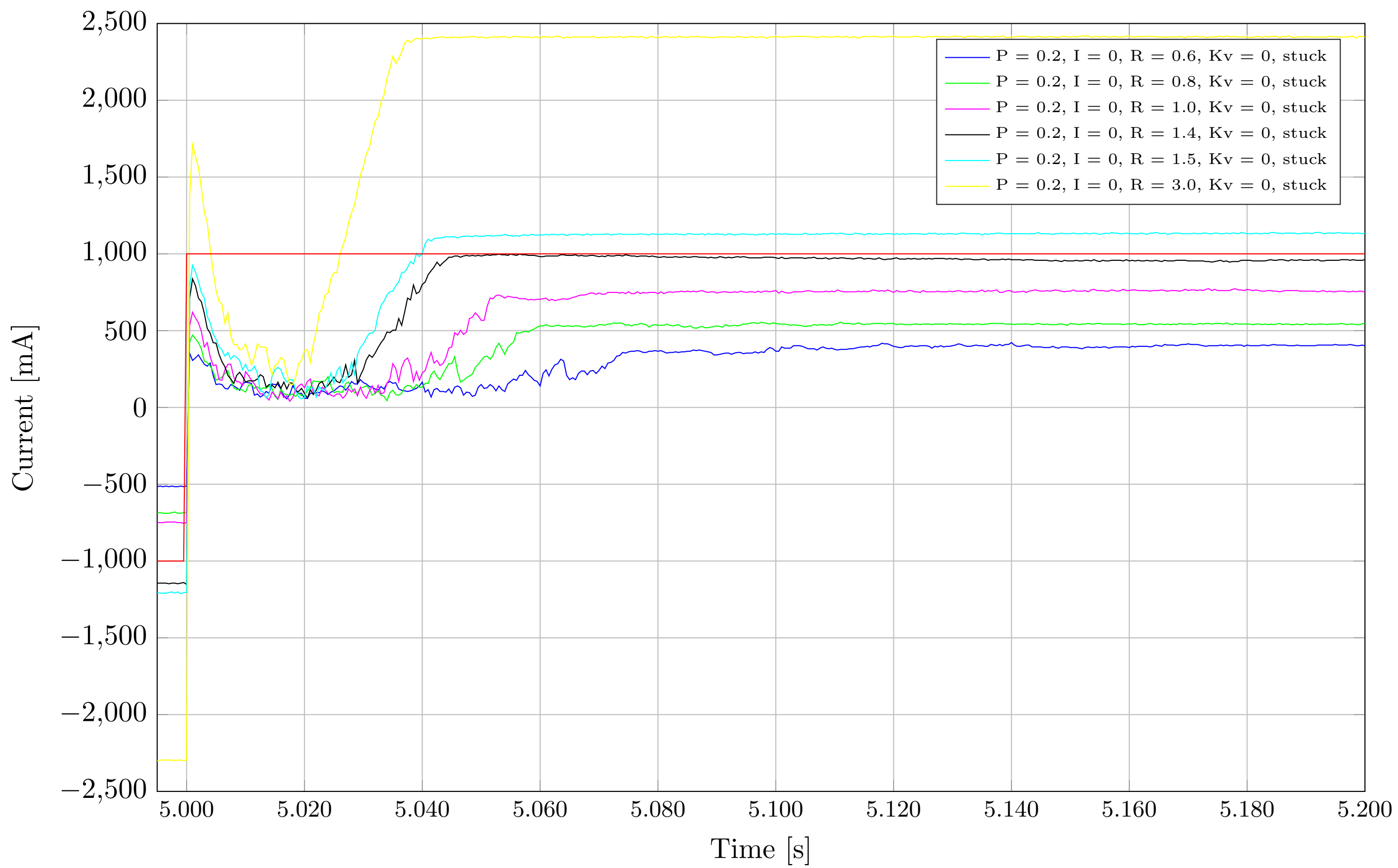


Figure 6.7: Step response for different resistance feedforward settings

The back EMF compensation is updated at 1 kHz, since the speed of the motor axis needs to be calculated. Updating the motor speed at 20 kHz like the motor resistance feedforward would result in a speed signal with a lot of noise which is propagated to the motor current through the feedforward. Updating the speed at 1 kHz means that the feedback PI controller running at 20 kHz already compensates the back EMF voltage drop before the feedforward signal gets the chance. Therefore looking at the effect on the current as was done with the motor resistance feedforward previously is not that useful. Instead the influence of the back EMF feedforward compensation is viewed on a higher level, with a low gain proportional ($P_v = 1.0$) velocity controller following a set trajectory and varying the motor speed constant to view the influence of the back EMF feedforward. This shown in Figure 6.8, where the P, I and R gain of the current controller were set to 1, 0 and 0, respectively. Note that only one in ten of the actual data points are plotted, in order to reduce this report's pdf size.

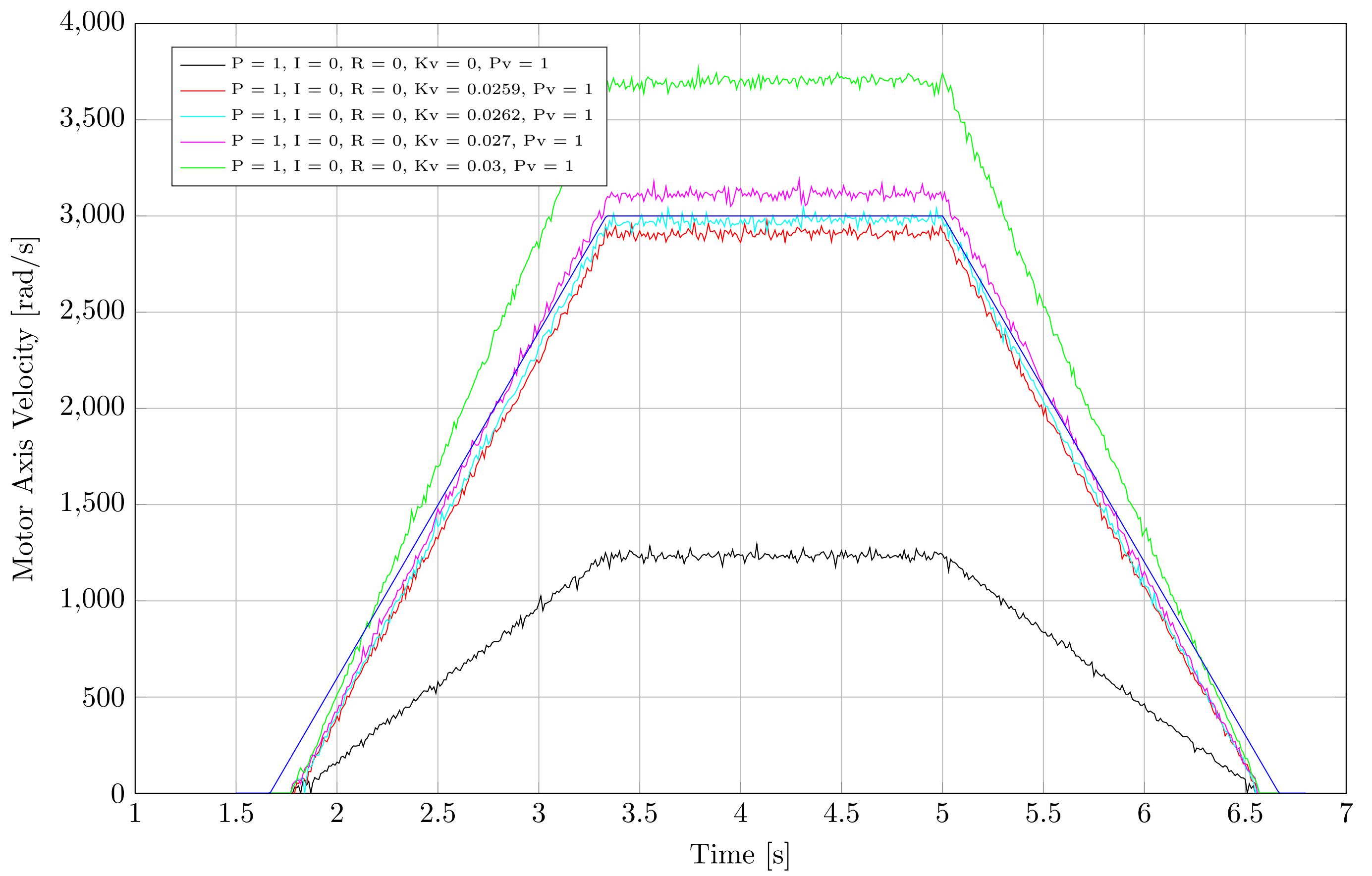


Figure 6.8: Plant with several test signals

From the figure the influence of the back EMF feedforward becomes apparent, with the red line with a value of $K_v = 0.0259$ the value as specified by the manufacturer.

Chapter 7

Manuals

7.1 SOEM for Windows

Installation
With SOEM (Simple Open EtherCAT Master) for Windows by Speciaal Machinefabriek Ketels it is possible to read and write to and test one or several EtherCAT slaves on a machine running Windows. In order for SOEMw to work correctly, WinPcap or Wireshark need to be installed in order to gain access to all the network layers. Furthermore, some anti-virus software might need to be disabled as they can prevent SOEMw from working (correctly). If TwinCAT is used in the same Windows environment, the system needs to be restarted.

Connecting to Slave
In order to connect with the in- and outputs of a slave, first the *Beckhoff* adapter needs to be selected from the dropdown menu, designated by the red box in Figure 7.1. Then the *config EtherCAT* button, designated by a blue box in the same figure, needs to be selected. In the text box *EtherCAT Configuration status* there should now be some text stating how many slaves are found. In the *Slave tree* text box the desired slave can be selected and its *IOmapping* (shown by the green box) can be expanded by clicking on it. In the figure it can be seen that we are connected with a slave that has 160 output bits (0-159) and 512 input bits (160-671), making for a total of 672 bits or 84 bytes.

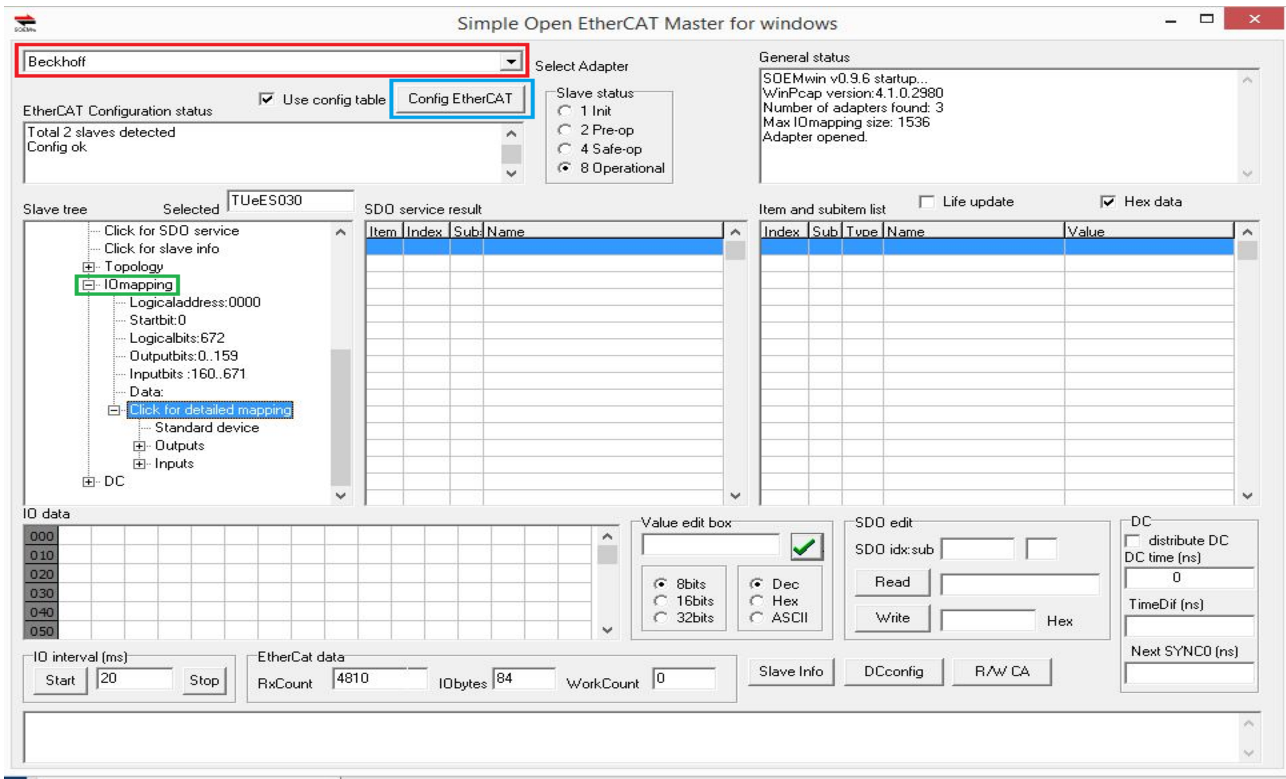


Figure 7.1: Connecting to EtherCAT slave using SOEMw

By clicking on *Click for detailed mapping* the in- and output mapping can be seen, as shown in Figure 7.2.

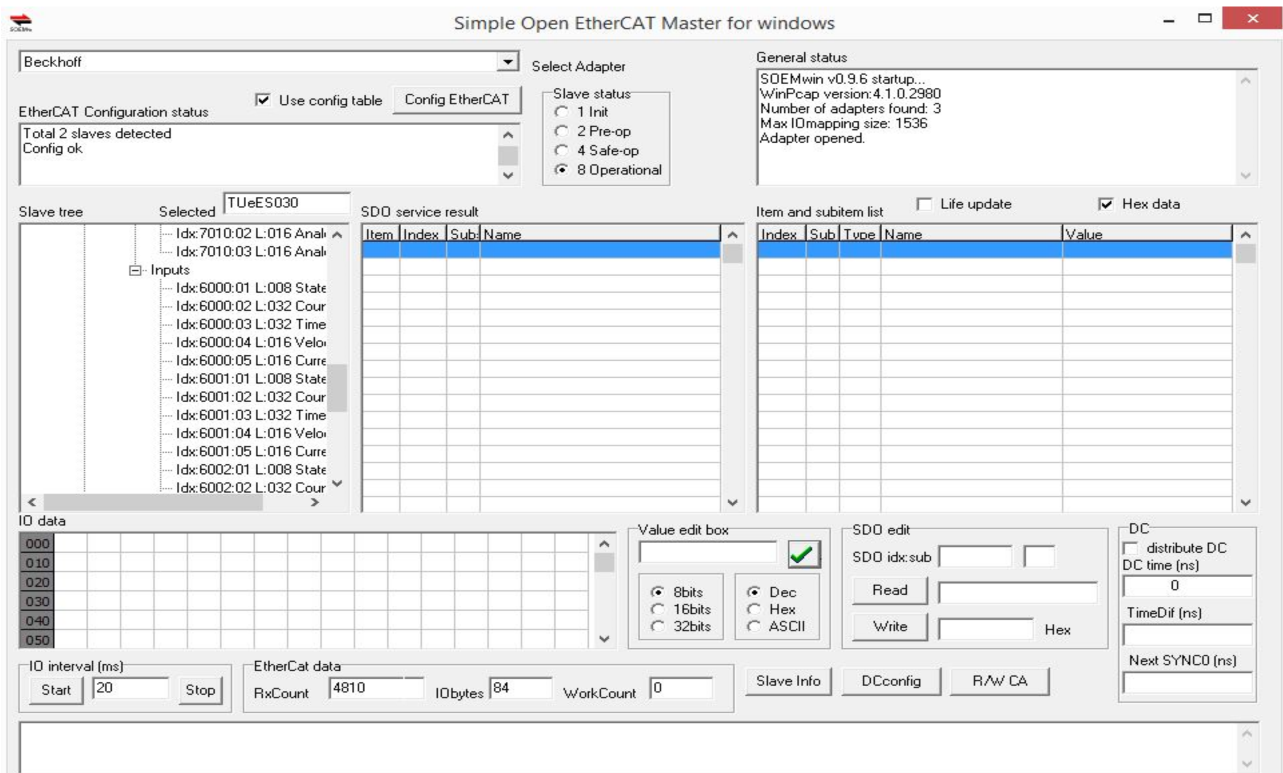


Figure 7.2: Detailed input/output mapping

The actual values of the variables can be seen by clicking on *Data*, as shown in Figure 7.3. The values

in blue are the outputs that can be sent to the slave and in green are the inputs coming from the slave. All values are represented in hexadecimal format. To edit an output value the *Value edit box* is used, where the size and representation of the variable can be specified using the radio buttons below the box.

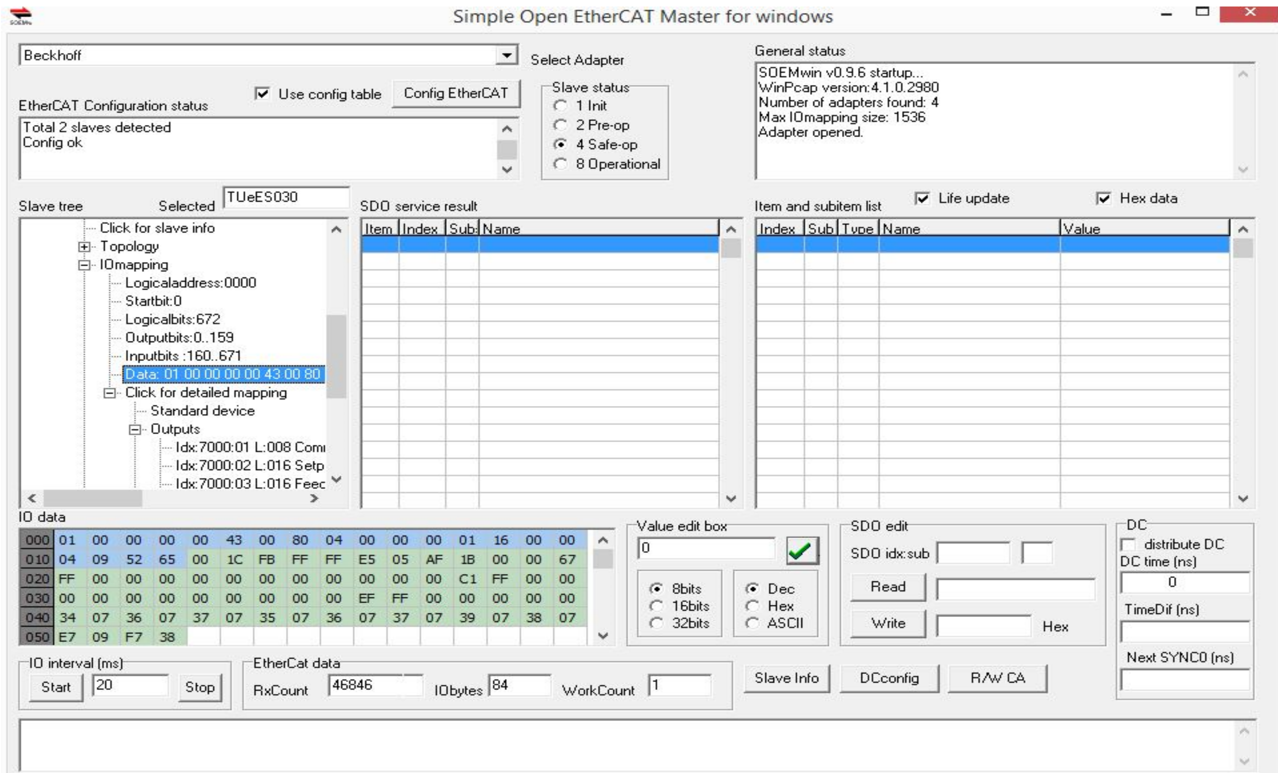


Figure 7.3: Input/output data

Changing parameters

By clicking *SDO services*, things such as the motor parameters in the case of the current firmware can be adjusted, as shown in Figure 7.4.

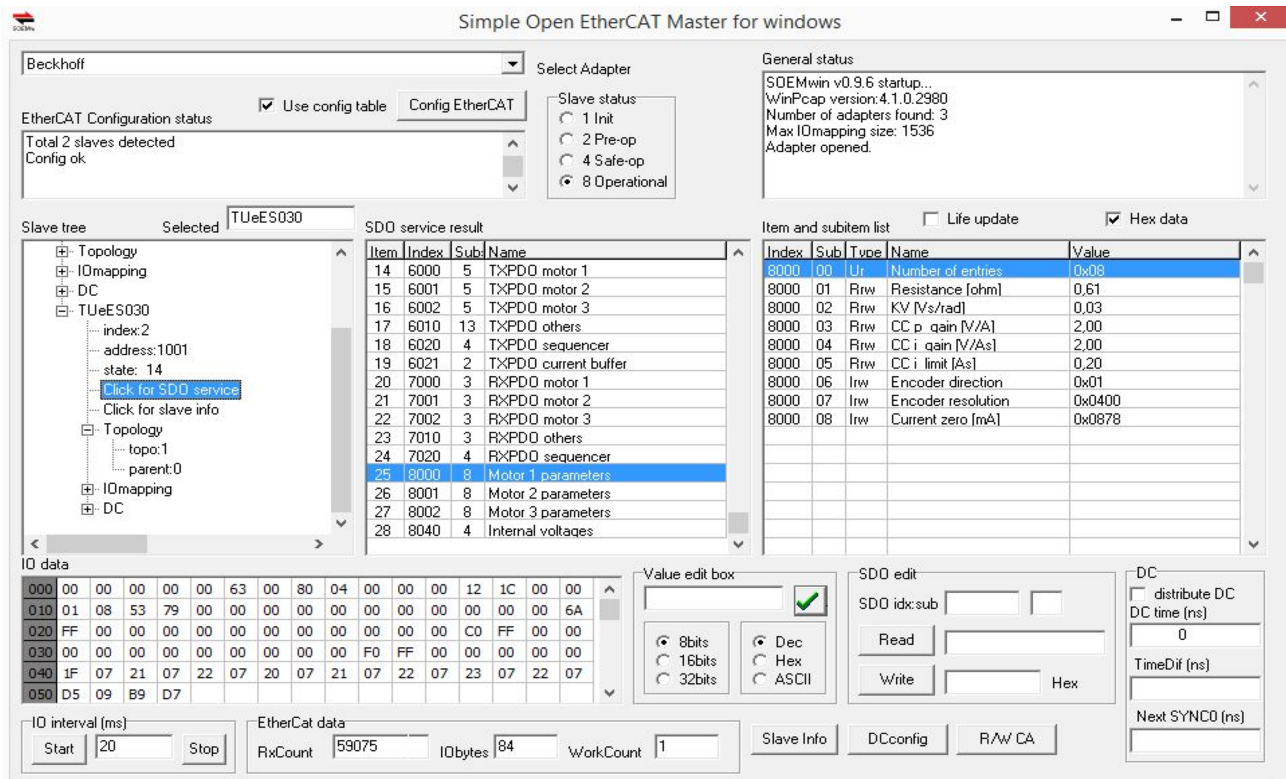


Figure 7.4: Changing parameters

Flashing the EEPROM file

By clicking on *Click for slave info* (see Figure 7.5) a new window opens, with all the information on the slave, as shown in Figure 7.6. A new EEPROM (Electrically Erasable Programmable Read-Only Memory) file can be flashed unto the slave here as well, by clicking *File* > *Eeprom*.

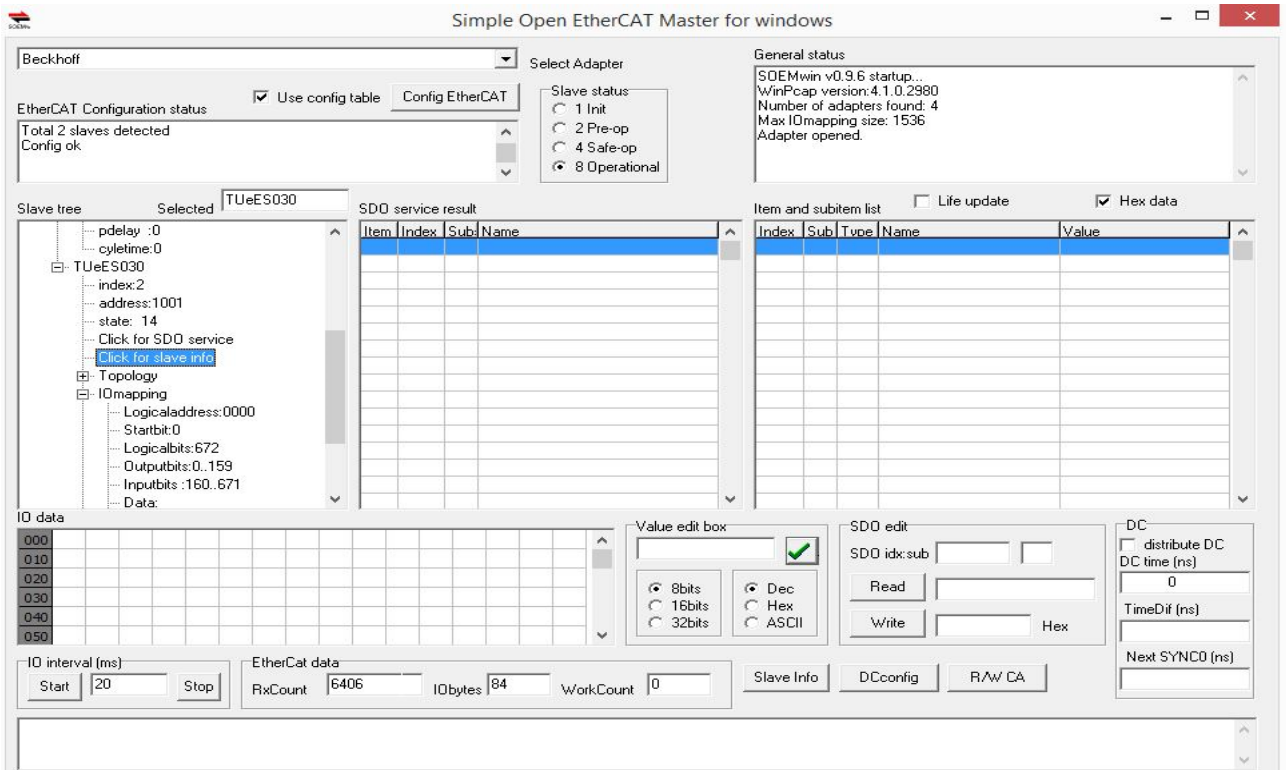


Figure 7.5: Slave info

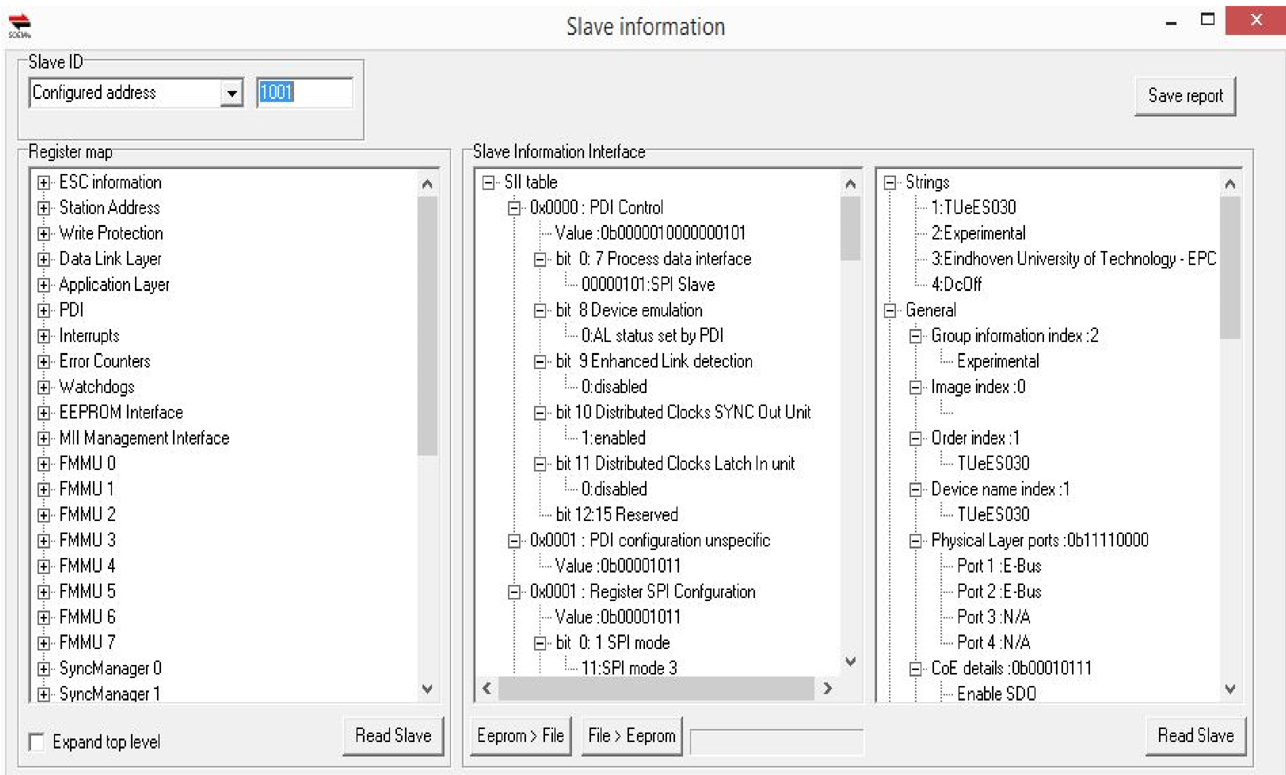


Figure 7.6: Slave information

7.2 TwinCAT

Appendices

Appendix A

Schematic of the electronic devices on the TUeES030 board

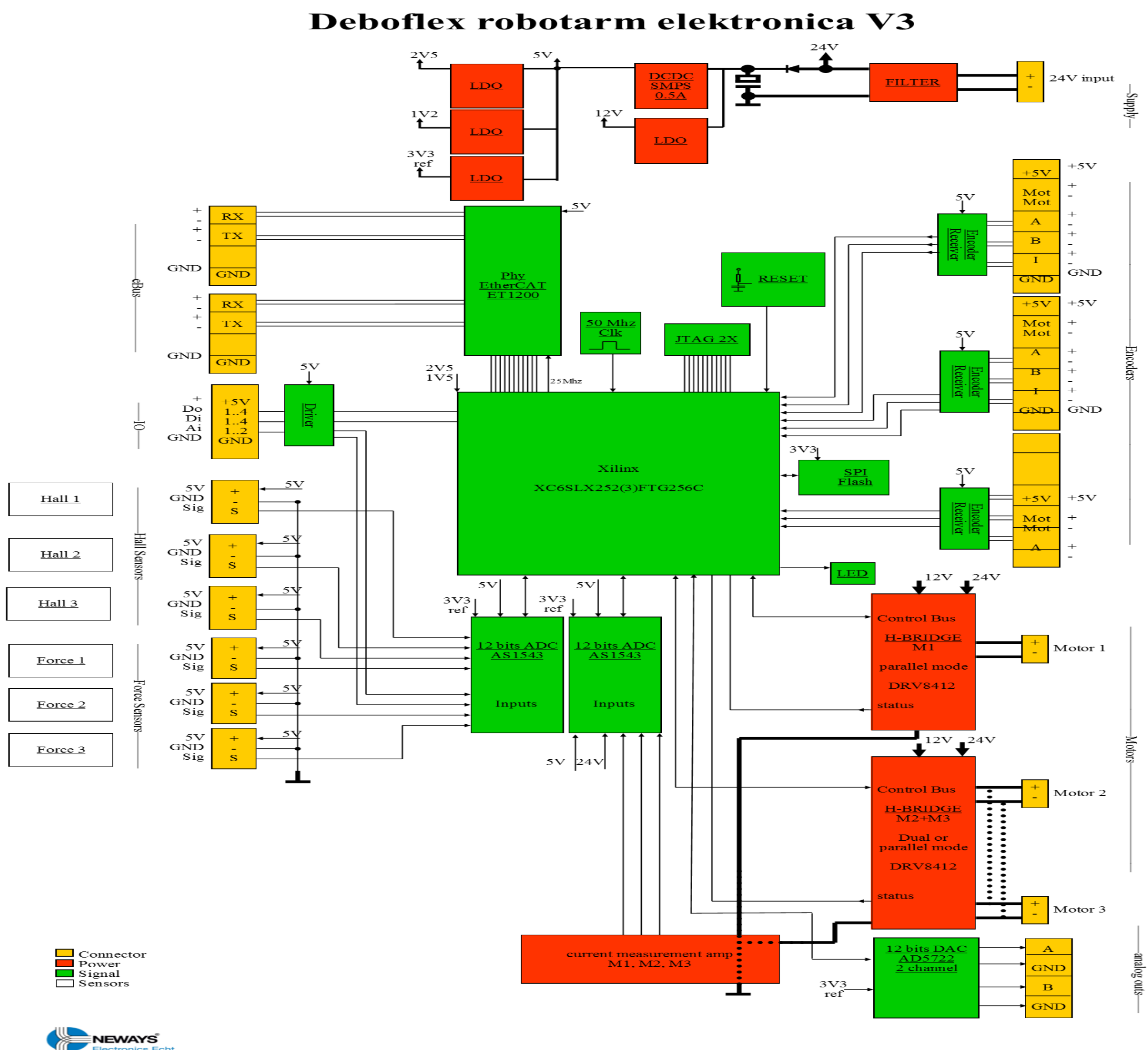


Figure A.1: Schematic of the electronic devices on the TUeES030 board by Ruud van den Bogaert