INSTITUT FÜR INFORMATIK

An FPGA-based Demonstrator for Dynamic Ticks

Andreas Boysen, Alexander Schulz-Rosengarten
and Reinhard von Hanxleden

Bericht Nr. 2001
July 2020
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Department of Computer Science
Kiel University
Olshausenstr. 40
24098 Kiel, Germany

An FPGA-based Demonstrator for Dynamic Ticks

Andreas Boysen, Alexander Schulz-Rosengarten and
Reinhard von Hanxleden

Report No. 2001
July 2020
ISSN 2192-6247

E-mail: {abo,als,rvh}@informatik.uni-kiel.de

Technical Report

Abstract

Synchronous programming languages often view time as just another input. However, handling different time inputs may result in unexpected timing behavior. This motivated making time a first class citizen to improve the performance and reduce the complexity of handling real time. Timed SCCharts is an extension of SCCharts that provides clocks to handle time. It is based on dynamic ticks, an environment that adjusts the execution of reaction dynamically and allows access to real time. As a result this environment only reforms a tick, if an input or timing event occurs. This reduces the calculation to a minimum to save power and to improve the average reaction time.

The goal of this report is to provide an example application for Timed SCCharts and dynamic tick environments. This application is used to provide data about the real world advantage of dynamic ticks and the performance of SCCharts. For this purpose a demonstrator, based on stepper motors, is built and the motor control software is written in Timed SCCharts. An FPGA is used as motor controller hardware. This allows to monitor the behavior of the motor controller and capturing data without interfering with the controller. The analysis of the data, collected this way, confirms the hypothesized advantage of dynamic ticks and further supports the usage of Timed SCCharts in critical applications.

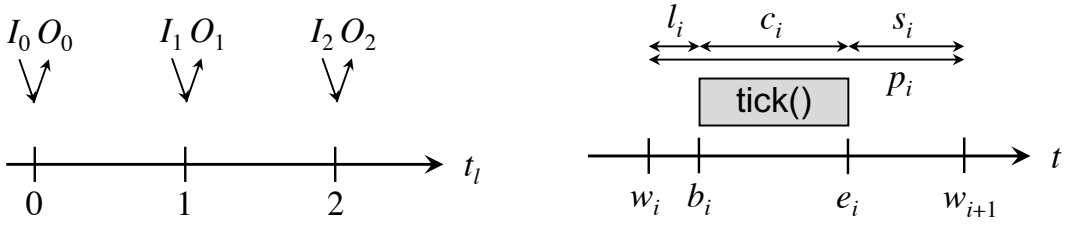
This report is an adjusted version of the Master's thesis of Andreas Boysen.

Contents

1. Introduction

Synchronous languages are well-established for the modeling and programming of reactive systems. In particular for safety-critical applications, such as flight control, automotive applications or in the medical sector, the deterministic semantics and formal grounding of synchronous languages have proven their practical value [3]. The synchronous paradigm, which states that outputs of a system are "synchronous" with their inputs, divides computations into discrete "ticks" that conceptually take zero time. Figure 1.1a visualizes this view. The timeline is separated into discrete (numbered) steps and input are read at the same time the outputs are produced. This is an abstraction from reality, since the computation of one tick, or one reaction, does of course take time as Figure 1.1b illustrates. The tick calculation is triggered at w_i. The loading of the input takes, in contrast to the theoretical model, some time. The next step is the calculation, beginning at b_i and is finished at e_i, then the outputs are written back. However, the synchronous abstraction is the basis for defining a concurrent semantics without race conditions, the Synchronous Model of Computing (SMoC). Just like boolean logic gives a well-founded, deterministic semantics to circuits that abstracts from their physical implementation and actual stabilization delays. Classical synchronous languages include Esterel, Lustre and Signal [3]; more recent languages include the hybrid modeling language Zélus [5] and the statechart dialect SCCharts [18], which now is used in the railway domain; commercially most successful at this point is probably SCADE [7], with its qualified compiler that is routinely used by Airbus and other industrial players.

Clearly, synchronous languages have been developed for real-time applications. However, unlike other languages developed for that domain, such as Ada, traditional synchronous languages do not include language features that explicitly address physical time. Instead, time is typically modeled by counting occurrences of input signals that denote the passage of a certain amount of time, or by simply counting ticks if ticks are known to occur at a certain, fixed frequency. This mechanism is rather crude and



(a) Logical time (b) Detailed physical time

Figure 1.1.: Time abstraction in SMoC [17]

has practical disadvantages, as observed by Bourke and Sowmya [6]. For example, if some input signal msec1 denotes the passage of 1 millisecond and another input signal msec10 denotes the passage of 10 milliseconds, a timeout waiting for 10 occurrences of msec1 does not necessarily take the same amount as another timeout that waits for one occurrence of msec10, as the actual waiting time depends on how the the timeouts are aligned with the timing input signals.

As von Hanxleden, Bourke and Girault have argued [17], one limitation of the traditional synchronous setting is how reactions are triggered. Specifically, it is traditionally the environment that decides on when reactions are computed. Typically, one of three options is used: 1) a *time-triggered* execution, where a reaction is computed for example once per millisecond; 2) an *event-triggered* execution, where reactions are computed whenever some input event occurs, such as for example the press of a button; or 3) an *ASAP* execution, where the next reaction is triggered as soon as the previous reaction is finished. Each of these options has its merits and is fairly easy to implement, but neither of them is particularly suitable for handling precise, fine-grained real-time requirements. However, it turns out that the synchronous paradigm can be seamlessly extended with a fourth option that is more amenable for real-time requirements. Specifically, the recently proposed *dynamic ticks* [17] give the synchronous program control not only about how it reacts to current and past inputs, but also *when* the next reaction should occur. Their proposal included a prototypical realization in Esterel, and the theoretical advantages of that approach seem rather clear. Subsequently, the concept of dynamic ticks was incorporated in Timed SCCharts [14], which basically augment SCCharts with clocks as used in Timed automata [1].

1.1. Time in SCCharts

Sequentially Constructive Statecharts (SCCharts) [19] is a Synchronous Programming Languages (SPL) designed for safety-critical applications. SCCharts is a dialect of Harel's statecharts but offers both state-based and dataflow-based programming in a hybrid notion. The underlying model is state-based and all dataflow is converted to state machines.

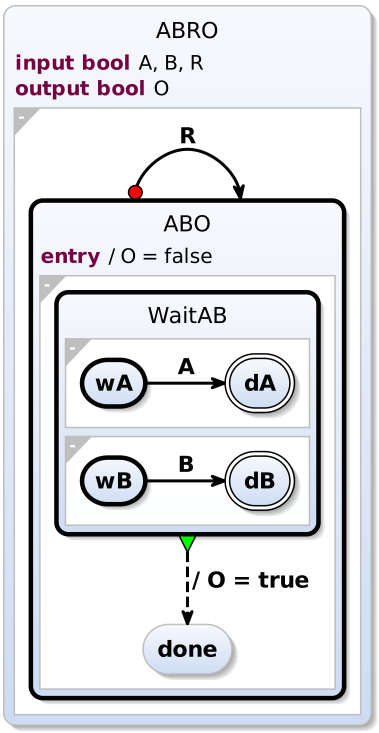
The Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) tool is the primary Integrated Development Environment (IDE) for SCCharts. It is eclipse-based and offers a compiler, simulation, and automatic visualization of SCCharts (see Figure 1.2b).

The hallo world program of SPLs is ABRO. Figure 1.2 contains the program code (a) and the visualization (b) of the program ABRO. The program output is false until both input signals, A and B, have been true for at least one tick. The SCCharts can be reset by setting the value of the reset signal R to true.

SCCharts do not interface with an operating system or any component directly. All IO is done by variables. This clear interface keeps SCCharts platform-independent. It is necessary to create a program that interfaces between the SCChart and other components like sensors, the operating system or timers. This program is from now on called the

```
scchart ABRO {
  input bool A, B, R
  output bool O

  initial state ABO {
    entry do O = false

    initial state WaitAB {
      region:

      initial state wA
      if A go to dA

      final state dA

      region:

      initial state wB
      if B go to dB

      final state dB
    }
    do O = true join to done

    state done
  }
  if R abort to ABO
}
```

(a) SCChart code



(b) SCChart visualized in KIELER

Figure 1.2.: ABRO in SCCharts

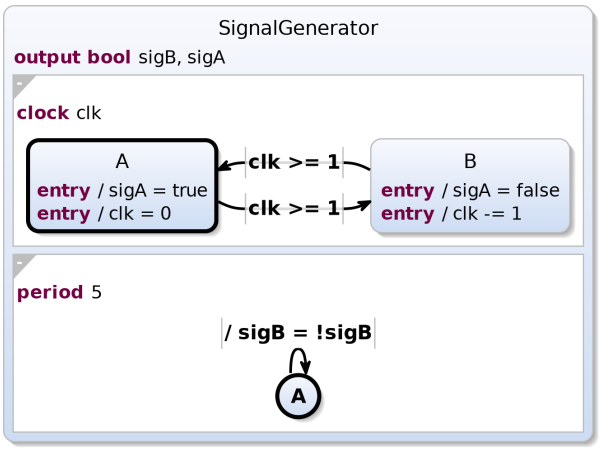
environment. This interface design keeps SCCharts system-independent.

The environment contains all system-dependent operations. It fetches the input for the tick function, runs the tick function and writes the output. The environment can work in different operation modes. In *periodical* mode the environment calls the tick function in a fixed time interval. The *ASAP* mode runs the tick function as soon as possible and the *event-based* mode triggers reactions only when inputs change. With the introduction of time in SCCharts [15] a new operation mode, the *dynamic* mode, is available. The interface between the SCChart and the environment is extended. A new signal *deltaT* is added. This signal is an input of the SCChart and contains the time passed since the last tick. This way the real-time can be used inside the SCChart. The *deltaT* signal is not accessed directly. Instead SCCharts are extended by the new signal type *clock*. Clocks can used in the same way as any other numeric variable. The difference to a number signal is that the clock value is increased by the time passed since the last tick. This allows for a natural use of clocks. Another extension is the keyword "period". In a region with a period of n transitions can only be taken every n seconds. This allows to declare regions that behave as if they are run by a periodic environment regardless of the used environment.

In Timed SCCharts it is possible to create periodical signals and delay reactions. Figure 1.3 contains a signal generator. The outputs sigA and sigB create square wave signals.

```
scchart SignalGenerator {
   output bool sigB, sigA
  region {
    clock clk
    initial state A{
    entry do sigA = true
    entry do clk= 0}
    if clk >= 1 go to B

    state B{
      entry do sigA = false
      entry do clk -= 1
    }
    if clk >= 1 go to A
  }
  region{
    period 5
    initial state A
    do sigB= !sigB go to A
  }
}
```

(a) Signal generator in Timed SCCharts



(b) Signal generator visualized in KIELER

Figure 1.3.: Signal generator

SigA has a period of two and sigB one of ten.

To produce the expected behavior, the SCChart should be run in *dynamic* mode, such that the environment not only waits for input events, but also for timing events. Additionally, the SCChart can calculate the time that has to passed until the next Timed transition is taken. This time is sent to the environment by the new output signal sleepT. The environment can then compare sleepT with the time passed since the last tick. As soon as deltaT is equal or bigger sleepT, a tick is triggered. Figure 1.4 illustrates the dynamic tick environment. The components, highlighted in red, are the addition to the event-based environment.

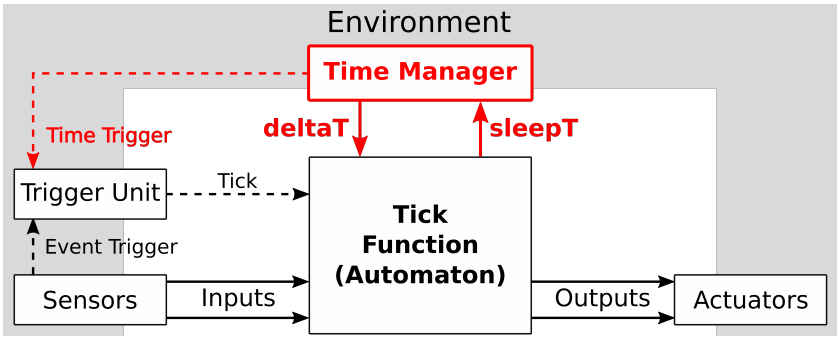


Figure 1.4.: Dynamic tick environment [16]

1.2. A Demonstrator for Timed SCCharts

With time in SCCharts and dynamic ticks, SCCharts offer capabilities for efficiently handling real-time applications. However, what was lacking so far was a practical evaluation, with very tight (i. e., microsecond scale) timing constraints, and a demonstrator with a hard real-time application, which is where this report comes in.

We describe the design and implementation of a physical demonstrator that is reasonably cheap and easy to implement but embodies a hard real-time problem with scalable timing constraints. The main platform chosen for the controller is an Field-Programmable Gate Array (FPGA) due to its real-time capability, common use in modern controllers [12] and the possibility to run real-time analysis. To demonstrate the platform independence of SCCharts and to compare the performance on different platforms, the controller hardware is replaceable.

We develop a Timed SCChart model to control the demonstrator that illustrates the usage of dynamic ticks and clocks. Then we evaluate the different controller platforms, comparing hardware (FPGA) and software alternatives, and evaluate the effect of dynamic ticks on reaction time, jitter and computational effort.

Outline

This report first presents a brief overview of technologies used for the demonstrator in Section 2. Then, Section 3 describes the basic demonstrator concept and adjustments in the SCChart compiler to enable a Very high speed integrated circuit Hardware Description Language (VHDL) code synthesis with dynamic ticks. The actual implementation of the demonstrator is presented in Section 4. Section 5 discusses the results of the performed tests and evaluates the performance of dynamic ticks. We conclude in Section 6 and address topics and improvements for future work.

2. Used Technology

The design and construction of a demonstrator for dynamic ticks includes various technologies. This section provides a brief overview of the available and used technologies. The most important is the FPGA that provides a high-performance real-time hardware controller for the demonstrator. Yvonne Lin describes in the application note *Using FPGAs to solve challenges in industrial applications* [12] the potential of FPGAs in industrial automation and industrial internet of things (IIOT). The low cost FPGAs of the Spartan-6 and Artix-7 families are recommended for motor controller applications, networking and other IO focused task.

The Artix-7 family is used for the demonstrator due to its high compute power that allows to run a network-based logic sniffer with data compression. Another advantage of Artix-7 is the better upgrade ability to Kintex and other FPGA families with more compute power.

2.1. FPGA Technology

An FPGA is a programmable logic Integrated Circuit (IC). The logic can be written to an FPGA, as if it is a program. The basic building blocks of an FPGA are LookUp Tables (LUTs), FlipFlops (FFs), and routing resources. Modern FPGAs also have hardened cores. These are not as flexible as LUTs, but they can solve their task faster, while consuming less power and space.

LUTs

LUTs are small memory blocks. Their output is only one or two bit wide and they have four to six bit wide addressing input. These blocks are used to generate simple logic function. Multiple LUT are routed together to create more complex functions

FFs

FlipFlops are used to clock the logic. The basic ports of an FF are the Data in (D), data out (Q) and Clock in (Clk). Clock Enable (CE) is an input signal. If CE is not present, the FF wouldn't store new data independent of the clock. Many FFs have an additional set or reset input. If the reset signal is present, the value of the FF is changed to the reset value instead of the data in value.

Slices

Slices are blocks containing FFs, LUTs and some additional logic. There are different types of slices varying in the additional logic. Figure 2.1 is the schematic of an l-type slice of an 7 series Xilinx FPGA. The LUTs are located on the left side. The multiplexer, right of them, allows to connect multiple LUTs together to create a bigger LUT. The carry logic is right of these multiplexers and allows the implementation of fast ripple carry operations such as addition. The right half of the image contains the FFs and the multiplexer that decide their input. The Clock, Clock Enable and Soft Reset signals enter on the bottom left and are shared between all FFs in a Slice.

Routing Resources

The routing resources are used to transfer signal between slices or between slices and hard cores. There are two types of routing resources, one for logic signals and the other for clock signals. Logic signal routing resources are used to connect data outputs to data inputs. Clock resources are used to distribute clock signals. Clock signals have a much higher fanout than data signals. Furthermore, clock signals have to reach each component at almost the same time.

Hard Cores

Hard cores are logic blocks, that have a fixed function instead of generating them through LUTs and FFs. They are used to add functionality to the FPGA that would otherwise not be available or would have otherwise a too high resource consumption. A simple example are serial to parallel input converters. They are used for protocols that use transfer speeds that are otherwise to fast for the FPGA. An example of such a fast protocol is PCIe. Multiplication is slow and space consuming when implemented in LUTs. For fast calculations DSPs are included in the FPGAs. Using LUTs to create bigger memory block is a waste of resources. For this reason, the FPGA contains blocks of RAM. The number of hard cores has increased with each generation of FPGAs.

PLL and other Clock Resources

The FPGA has multiple types of resources for the creation and handling of clocks. The Phase-Locked Loop (PLL) can be used to create clock signals. The clock signals can be temporarily disabled by clock gates. Clock buffers can recover external clocks for internal use. The types of clock resources available vary between different FPGAs and FPGA manufactures.

2.1.1. Current Developments

The current trend in FPGA development is the addition of more hard cores. One example is the addition of new DSPs for AI calculations in some new FPGAs. Another trend is the

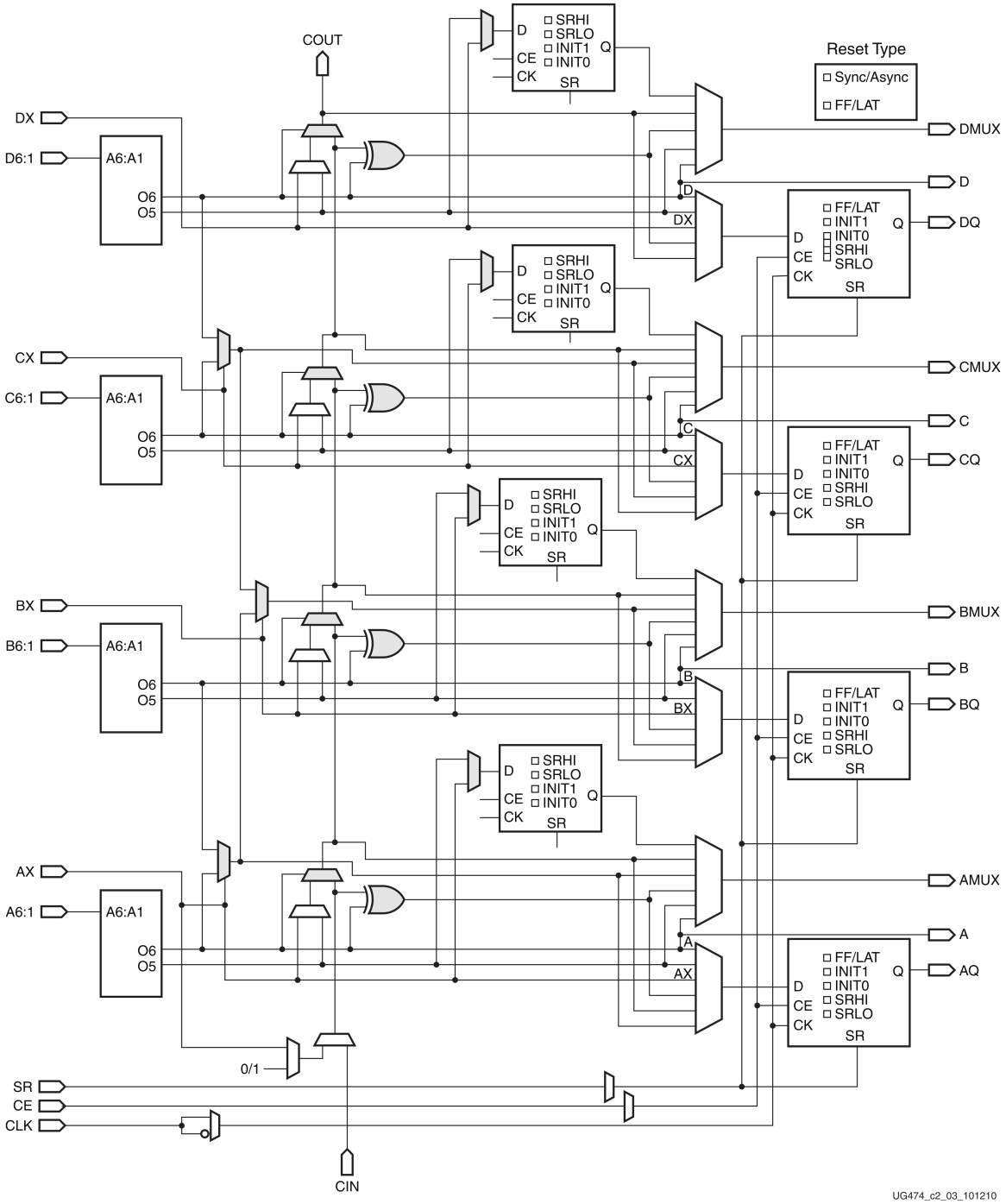


Figure 2.1.: 7 Series FPGAs configurable logic block (https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf)

addition of more RAM to the FPGA. In the current development the RAM access speed has become more and more the new bottleneck of the system. With more available gates per chip and more DSPs the compute power increases. This increases the need for fast data access. New storages such as MVME SSD get faster rapidly and the new transceiver cores of FPGAs allow to create 100 Gbps network connectivity. DDR memory, on the other hand, had nearly no performance increase in the same time. To counter this problem some FPGAs have integrated HBM memory [20]. This development allows memory accesses of more than 1 Tb per second. This removes the memory bottleneck.

2.1.2. FPGA Programming

Creating a program for an FPGA is more complex than writing an application for a CPU. The first step is to create the Register Transfer Level (RTL) code. The RTL is a synchronous abstraction layer that describes the way registers and combination logic is connected together. The RTL is described in a Hardware Description Language (HDL) such as VHDL. It is common practice to simulate the RTL before deploying it. Only if the generated RTL is correct the compilation for the FPGA is started. In the synthesis compilation step the RTL is converted into a setup of FPGA components such as LUTs, FFs, routing resources and hardened cores.

There is further information needed to put the RTL onto the FPGA. Constraint files are used to provide the necessary information about the used IO resources, clocks and timing constraints. There is no guarantee that it is possible to put a given RTL and constraints on an FPGA. There may be not enough resources on the FPGA or the set clock speed is too fast for the selected FPGA. The implementation compile step tries to place all components on the selected FPGA and creates the routes between them. If this step is successful, the bitstream for the FPGA can be created, otherwise the RTL and/or the constraints have to be adapted. This implementation compiler step is NP-hard and the input is so big that not all possible placings can checked. Therefore it is possible that a placement and a routing exist, but the compiler does not find it.

2.1.3. VHDL

VHDL is an HDL used to describe the RTL for FPGAs and Application-Specific Integrated Circuits (ASICs). Figure 2.2 contains ABRO, introduced in Section 1.1, written in VDHL. Figure 2.3 shows the created RTL. This RTL can be synthesized for an FPGA. The result is displayed in Figure 2.4. This implementation assumes that the input signals are synchronized with the clock CLK. If inputs are not synchronized, it is necessary to sync them with double buffering input registers to avoid meta stable signals reaching the logic.

The RTL creates the behavior, expected by the SMoC, as long as all input signals are synchronized with the clock.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity abro is
    Port ( clk : in STD_LOGIC;
           a : in STD_LOGIC;
           b : in STD_LOGIC;
           r : in STD_LOGIC;
           o : out STD_LOGIC);
end abro;

architecture Behavioral of abro is
 signal gotA, gotB : STD_LOGIC;
begin

abro : process(clk) is
begin
if rising_edge(Clk) then
    if r='1' then
        gotA<='0';
        gotB<='0';
        o<='0';
    else
        gotA<= gotA or a;
        gotb<= gotB or b;
        o<= (gotA or a) and
            (gotB or b);
    end if;
end if;
end process;

end Behavioral;
```

Figure 2.2.: ABRO in VHDL

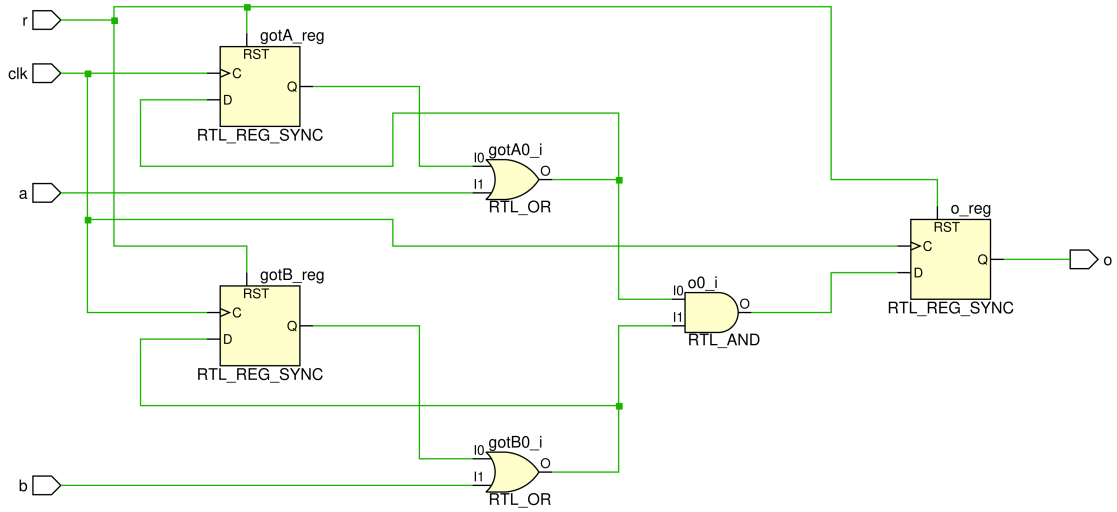


Figure 2.3.: RTL generated from VHDL ABRO in Vivado

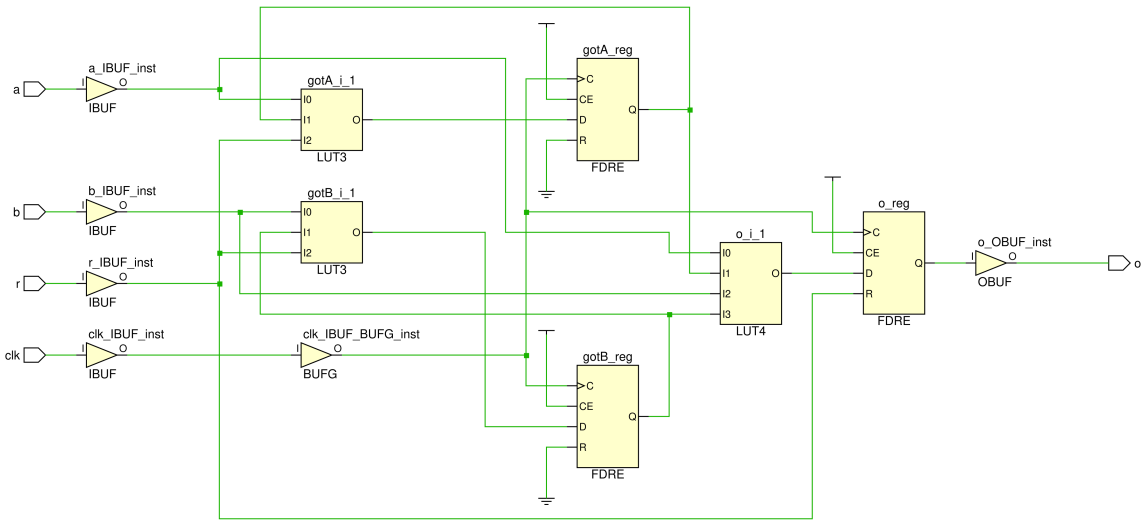


Figure 2.4.: VHDL ABRO synthesized with Vivado for FPGA

Constraint Files

Constraints provide additional information about the RTL and the target hardware. The first type of constraints are IO constraints. These constraints describe the relations between the input and output signal of the RTL and the physical pin on the IC. Furthermore, it restricts the modus and voltage the pin is using. The most common IO voltages are 1.2 V, 1.35 V, 1.5 V, 1.8 V, 2.5 V and 3.3 V. Low Voltage Complementary Metal Oxide Semiconductor (LVCMOS), Low-Voltage Differential Signaling (LVDS) and Low-Voltage Transistor-Transistor Logic (LVTTL) are the most common operation modes.

Another type of constraints are clock declarations. Clock declarations ensure that clock signals are using specific clock routing resources. Furthermore, this information is used to check if the implemented design violates the setup and hold times of the FF. In some cases the setup and hold time restriction are too hard. Timing constraints allow to relax this restriction. A false path declaration removes any timing checks from a signal. This can be used for constant values. An example is a configuration that is only written during reset. Another timing constraint is the Multi-Cycle Path (MCP). This constraint allows to increase the number of cycles of the setup time. This allow to have parts of the logic run at a slower speed.

2.1.4. Xilinx: Vivado

The Vivado Design Suite[1] is the IDE provided by Xilinx for their FPGAs[2] and FPGA-based SOCs[3]. Vivado replaces ISE[4], the previous IDE from Xilinx.

2.2. Raspberry Pi

The Raspberry Pi[5] is a single board computer. The Raspberry Pi was the starting point of the modern development of the single board computer market. The Raspberry Pi Family is still the reference for other single board computers due to its low price and good performance.

2.3. ATmega

ATmega is a family of microcontrollers using the AVR instruction set. The members of this family vary in size. Some of them have specialised features such as USB or SRAM interfaces. An example for an ATmega with USB is the ATmega32u4[6]. One of the most famous uses of ATmega microcontrollers is in arduino[7] boards.

AVR[8] is a RISC architecture for microcontrollers. The bytecode is designed to be easy to decode. Most of the instruction only take 1 clock cycle.

2.4. Stepper Motor

A stepper motor [8] is a motor designed for precise rotation. The rotation of the motor is divided in smaller steps. Many stepper motors have 200 steps per rotation. The most common type of stepper motor is the hybrid stepper motor.[9]

Hybrid stepper motors have a rotor with a permanent magnet and a stator with two independent coils. Figure 3.3 visualizes the operation of a stepper motor. Magnetizing the coils in the correct order creates a rotating magnetic field. The rotor is following this magnetic field. The rotor of real stepper motors has more than two poles to create smaller steps.

[1] https://www.xilinx.com/products/design-tools/vivado.html
[2] https://www.xilinx.com/products/silicon-devices/fpga.html
[3] https://www.xilinx.com/products/silicon-devices/soc.html
[4] https://www.xilinx.com/products/design-tools/ise-design-suite.html
[5] https://www.raspberrypi.org/
[6] https://www.microchip.com/wwwproducts/en/ATmega32u4
[7] https://www.arduino.cc/
[8] http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-0856-AVR-Instruction-Set-Manual.pdf#_OPENTOPIC_TOC_PROCESSING_d94e25246
[9] https://www.orientalmotor.com/stepper-motors/technology/hybrid-stepper-motors-v-hybrid-control.html

2.4.1. H-bridge

To drive a hybrid stepper motors, the controller needs to be able to power its coil in both direction. The setup necessary is called an H-bridge[10]. In an H-bridge setup, each end of the motor coil is connected to a push-pull driver.

2.4.2. Current Development of PowerFETs

The improvement in the production technique for Gallium Nitride Field-Effect Transistor s (GaNFETs) brought the production cost down. This allows the use in many power electronic products, such as motor drivers. The advantages of GaNFETs are that they have lower resistance, faster reaction time and lower gate capacities. This allows to switch loads in the MHz range and up to the GHz range. GaNFETs provide interesting timing requirements [9]. This timing problem exceeds the capabilities of most controllers. GaNFET are not used here to keep the focus on Timed SCCharts and to have a higher range of possible controllers.

2.4.3. Controller Types

Motor controllers can be categorized in two categories: open loop and closed loop. Open loop controllers have no feedback from the motor regarding its position. Open loop controllers are simple to build, but are not able to recognize any errors such as missed steps.

Closed loop controllers use sensors to read the position of a motor to recognize errors, such as missed steps. The controller can try to catch up by repeating the missed step. The setup of the sensor is often complicated and increases the cost of a motor setup.

A common version of a motor controller is an open loop controller that has an over current protection. This controller type uses PWM to limit the current running through the motor. In this setup the motor controller has no feedback of the motor position and is therefore in open loop configuration, but the over-current protection allows the use of higher voltages, this allows to run the motor faster.

The drop in price of embedded compute power led to the development of a new type of stepper motor controller. This new type of motor controller simulates the stepper motor. The difference of simulated motor current and real motor current allows to conclude the motor position. This sensorless closed loop motor controller can detect errors, such as lost steps, as if it has position sensors. This type of controller is easier to setup since no sensors are needed. Furthermore it can directly replace an open loop stepper motor controller without any modification to the motor setup.

Bendjedia et al. [2] describe the implementation of a sensorless closed loop controller using Kalman filtering.

Simulation and PID based controllers need to run continuously. Dynamic ticks, on the other hand, are designed for reactive programs (input and/or time triggered). An

[10] https://blog.digilentinc.com/what-is-an-h-bridge/

open loop stepper motor controller is a reactive program and therefore it is used in this report.

2.5. CAD

Computer-Aided Design (CAD) is used in all modern forms of production. CAD programs are used to create models of products. These models are then exported in standardized formats to be used by other CAD programs and for the production of the part.

KiCad

KiCad[11] is a CAD program for the creation of PCBs. KiCad is used to create schematics and PCB Layouts.

OpenSCAD

OpenSCAD[12] is a CAD tool to create 3D models. OpenSCAD uses a Domain Specific Language (DSL) as user input. One of the application of OpenSCAD is to create 3D models for 3D printing.

FreeCAD

FreeCAD[13] is an open source parametric modeler. It can be used to create 3D objects from 2D scatches. The 3D modes can be converted to 2D technical drawings.

2.6. Production Techniques

The different components of the demonstrator require different manufacturing techniques.

2.6.1. 3D Printing

3D printing is an additive manufacturing technique. The 3D model of the part is sliced into layers. This is done by a tool such as slic3r.[14] The 3D Printer[15] creates the object then by adding material slice by slice.

[11] http://kicad-pcb.org/
[12] https://www.openscad.org/
[13] https://www.freecadweb.org/
[14] https://slic3r.org/
[15] https://opensource.com/article/19/8/3D-printers

2.6.2. Soldering

Soldering is used to fix components on a Printed Circuit Board (PCB) and to get a conductive connection between the pads on the PCB and the pins of the components. There are two methods of soldering used for this report, soldering with a soldering iron and solder[16] and reflow soldering with a hot air gun and soldering paste.[17]

[16] https://www.makerspaces.com/how-to-solder/
[17] https://www.build-electronic-circuits.com/reflow-soldering/

3. Design

This Section describes the basic structure of the demonstrator and the related design decisions. Furthermore, the concepts for an adjusted VHDL code synthesis for SCCharts is presented, as well as the structure of the tick environment.

3.1. Demonstrator

The task of a demonstrator is to highlight features of a product, here Timed SCCharts with dynamic ticks. The advantages of Timed SCCharts are that they handle and adapt to physical time and react to events immediately or with delay. In combination with a dynamic tick environment it is possible to reduce the number of calculations, leading to a lower power consumption.

The most imported requirement for a demonstrator is a good visual feedback that is related to the promoted features. A spectator has to grasp the concept of the demonstrator in a short amount of time and has to be able to understand that the system is running correctly. A demonstrator can be enhanced by analysis. The analysis results can be used for further explanations. A demonstrator with analysis can also be used for further development.

3.1.1. Demonstrator Selection

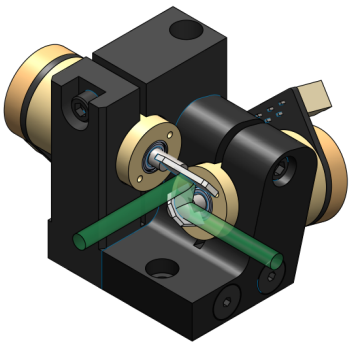
There are many ways to demonstrate the capability of Timed SCCharts. Three of them were taken in closer consideration.

Galvo-based laser scanner

A galvometer (galvo) is an electromechanical instrument that rotates in a position depending on the applied current. The high reaction speed of galvos can be used to create fast servo motors with limited rotation range. To achieve this, a closed loop controller is added to the galvo. On a range of eight degrees most galvos can reposition between 10000 and 100000 times per second. Mounting two galvos, with mirrors attached, in a 90 degree angle creates a scanner. A laser pointed at these mirrors can be redirected to create a projection of an image. Figure 3.1a is a model of this setup.

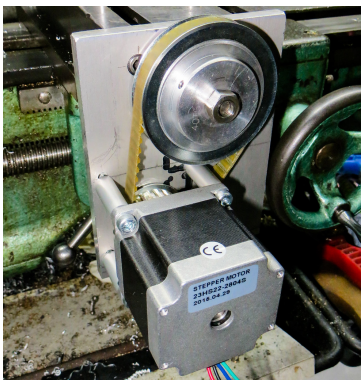
Redirection of light beams has further applications. Keeping a Free-Space Optical Interconnect aligned [4] for an example.

A laser scanner has a good visual feedback. In case of a timing error the projected image gets distorted. Furthermore, with up to 100k positions per second it is possible



(a) A galvo-based scanner (ScannerMAX Compact-506 6thD: http://6thd.eu/cz/produkt/scannermax-compact-506/)

(b) Traffic light (Sn00zerman CC BY 3.0 https://www.thingiverse.com/thing:1458548)



(c) Stepper motor (tudedude CC BY-NC-SA 2.0 https://www.flickr.com/photos/53354512@N00/29346359636)

Figure 3.1.: Symbolic images of the potential demonstrators

to demonstrate the speed of SCCharts on an FPGA. The downside of the laser scanner is that analysis of errors is complex since the position of the galvos has to be captured. The closed loop control circuit of the galvo is often analogous. Fast Analog to Digital Converters (ADCs) are needed to replace the analogous circuit with an SCChart-based digital control loop. This increases the demands to PCB design and hardware components.

Traffic lights

The idea of this demonstrator is to build a model of a road junction with traffic light inducing pedestrian crossing. A pedestrian traffic light was also used as an example when times SCCharts were introduced [15]. Furthermore, a traffic control is a common example application for synchronous programming languages due to their safety criticality [10].

The traffic light is a Timed system. This can be used to demonstrate the capability of Timed SCCharts to create periodical signals. Figure 3.1b illustrates an exemplary setup. A pedestrian can request to cross the road by pressing the button. This event is, depending on the state of the traffic light controller, handled immediately or after a delay. Therefore this demonstrator has every type of event possible in Timed SCCharts. The downside of this model is the lack of a visual feedback. The difference in reaction time is not recognizable and there is no way to know if a longer waiting time of a pedestrian is due to the state of the controller or a bad reaction time.

Stepper motor controller

A stepper motor is a special type of motor designed for precise rotation. The stepper motor can move in fixed steps. Most stepper motors move in steps of 1.8 degrees. A

stepper motor has two different coils. The stepper motor coils have to be powered in a specific order and direction to create the rotation. Figure 3.1b presents an example of a stepper motor in industrial use. T. Bourke used these control signals as a motivation for real-time in the synchronous programming language Esterel in his dissertation [5]. The amount of rotation, done by a stepper motor, is known by the controller, even if the stepper motor is in an open loop configuration. A good visual feedback can be created by mounting a disk on the stepper motor.

The complexity of a demonstrator with a stepper motor can be increased by using a simple motor signal generator with additions, such as an over current protection for the coils. Precise control signals allow to run the motor much faster. In case of a timing error in the control signal at high RPM the motor will not only loose steps, but also stop completely. This is a simple but powerful demonstration. Furthermore, is it easy to analyze the correctness, delay and jitter of the motor control signals. Lastly there is no lower limit for rotation speed. This allows to run the controller software on slower platforms for comparison. The downside of the stepper motor is that there is more hardware needed to create good visual feedback.

Selection

The stepper motor is the best of this three demonstrators for highlighting the features of Timed SCCharts. It has a good visual feedback and can demonstrate all features of Timed SCCharts. One of the deciding factors was the ability to analyse the performance. The clear expected behaviour combined with the wide rage of operation frequency allows to benchmark an SCChart on a wide range of platforms.

3.1.2. Demonstrator Setup

The goal is to demonstrate that the stepper motor is running precisely. This can be achieved by adding partial disks to the stepper motor. Figure 3.2 displays a setup of two stepper motors that are mounted in a 90 degree angle. The partial disks, mounted on the motor, are passing through each other. If one of the stepper motors looses a step, the disks will collide.

The work on the demonstrator can be split into two parts. The first part is the basic set of components necessary to drive a stepper motor. In the second part this setup is used to create the setup of the demonstrator.

Basic Stepper Motor Setup

A stepper motor creates a rotating magnetic field by magnetizing coils. The rotor is locked to the magnetic field. There are multiple setups possible. The most common is the hybrid stepper motor. The hybrid stepper motor uses two coils and a rotor with a permanent magnet. The half step control system pattern can be used to run this kind

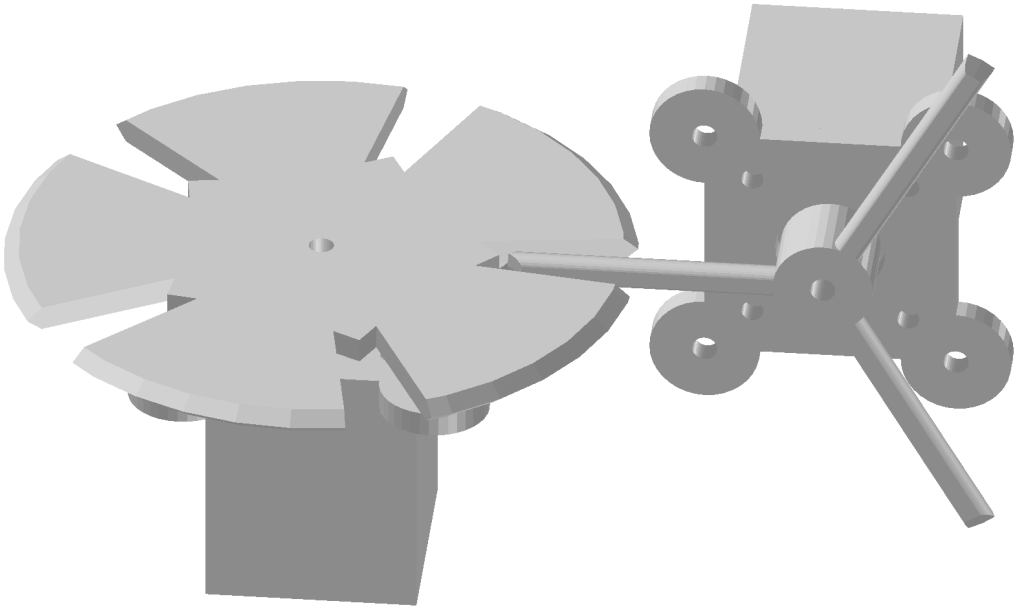


Figure 3.2.: CAD image of a stepper motor based demonstrator

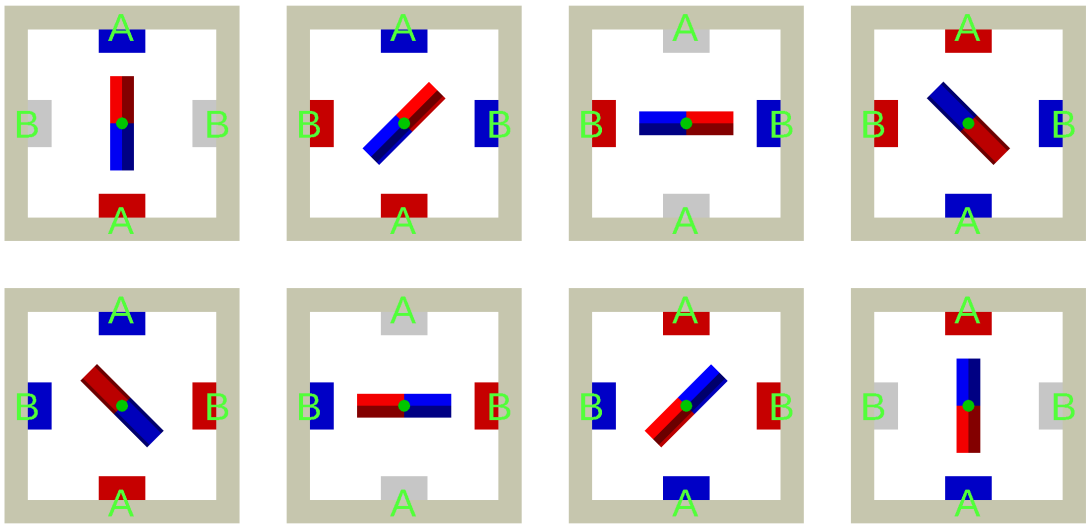


Figure 3.3.: The 8 states of a stepper motor in half step mode. The images are ordered clockwise.

Wire	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
A+	VCC	VCC	-	GND	GND	GND	-	VCC
A-	GND	GND	-	VCC	VCC	VCC	-	GND
B+	-	VCC	VCC	VCC	-	GND	GND	GND
B-	-	GND	GND	GND	-	VCC	VCC	VCC

Table 3.1.: Powering of the stepper motor wires in half step mode

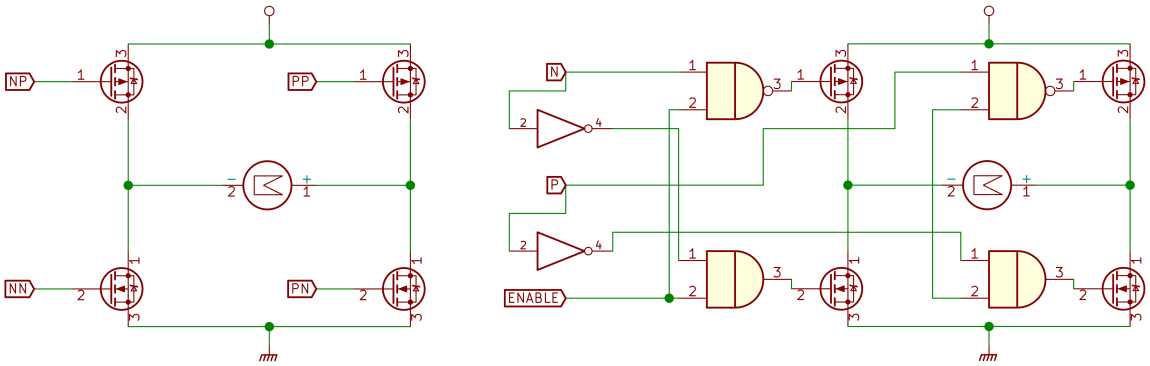


Figure 3.4.: Simplified schematics of an H-bridge

of motor. Figure 3.3 displays the eight different stages of a hybrid stepper motor in half step mode. The necessary input signals for the coils are listed in Table 3.1.

Motors consume significant power, therefore they cannot be powered by the controller directly. A driver is needed between the controller and the motor. The driver converts the low power digital signal into high power signals for the motor. The coils of a stepper motor have to be powered in both directions. The setup of power transistors that can power a system in both direction is called an H-bridge. Figure 3.4 contains the schematic of an H-bridge on the left side. The transistors that are in series between GND and VCC may never be turned on at the same time. Otherwise GND and VCC would be shorted. To ensure this, some H-bridges do not allow to control the four transistors directly. Instead, an interface is used that has for every half of the H-bridge a signal that carries the polarity and an enable signal for the entire bridge. This setup is on the right side of Figure 3.4.

Inductive loads, such as a coil, resist changes in current flowing through them. This leads to a reduction in motor power at higher speeds. This problem can be circumvented by increasing the used voltage. To prevent the motor from drawing to much current at low speeds a current limitation is needed. This is done by measuring the current drawn and using the motor coil and H-bridge as a step down (buck) converter topology to limit the current.

Many types of ICs can be used as a controller. In this report three different types are used: an Micro Controller Unit (MCU) in form of an Atmega32u4, a Micro Processor

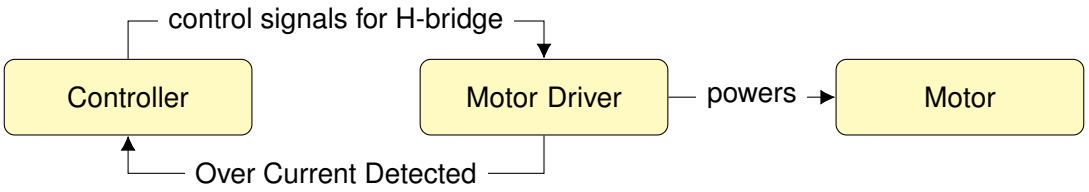


Figure 3.5.: Basic demonstrator setup

Unit (MPU) in form of a Raspberry Pi 3 and an FPGA in form of an ARTIX A7-35T. These controllers are selected due to their position in their respective domains. They have low to medium calculation performance and are affordable.

The controller gets input from the user by buttons. The user gets feedback about the internal state via LEDs. The last input is the speed signal generated by an external signal generator. The external signal generator has multiple advantages over an internal. The use of an external signal generator make a comparison between the platforms fair. All platforms get exactly the same speed signal. Furthermore the interface of the controller towards the user is simplified since the signal generator has its own interface.

Figure 3.5 contains the setup of the basic motor controller components set. The controller sends motor control signals for the H-bridge to the driver. The H-bridge on the driver powers the motor. If the motor is drawing too much current, the motor driver detects this and sends this information back to the controller. The controller has to react to this by disabling the H-bridge.

Demonstrator Setup

In Figure 3.6 the complete demonstrator setup is displayed. The controller is connected to two motor drivers. Each motor driver is connected to a stepper motor. The motors are mounted in a 90 degree angle on the motor mount, as visualized in Figure 3.2. MotorS has three sticks attached to the motor shaft. The sticks are in a 120 degree angle to each other and in a 90 degree angle to the motor shaft. MotorD has a disk with five cut-outs mounted to its shaft.

If the motors are in the correct initial position and the motors are run with a speed ratio of three to five, the disk and the stick will pass through each other. A timing error of the controller leads to lost steps. The consequence is that the disk and the sticks are no longer synchronized and would hit each other. The disk and sticks running through each other are the indicators that everything is running as expected. A collision on the other hand would be clear indicator that something went wrong. This creates the visual feedback for the user.

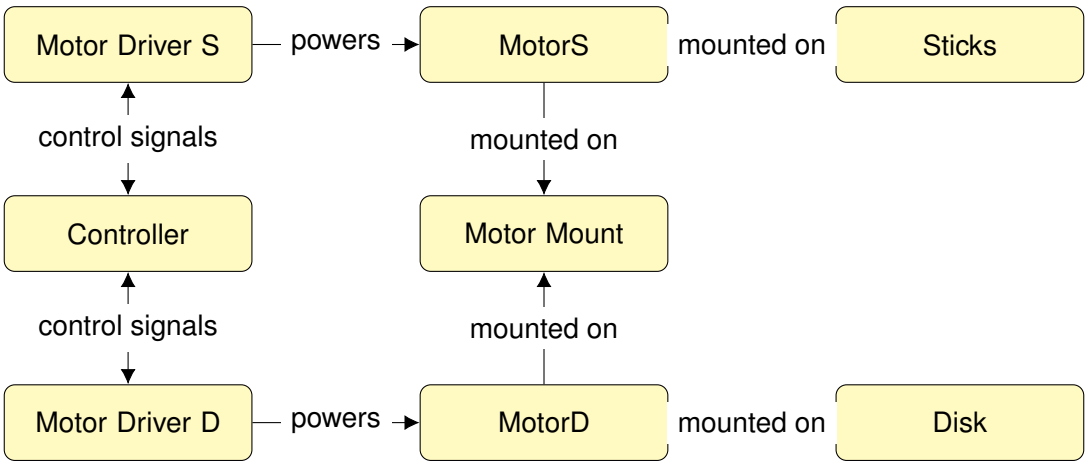


Figure 3.6.: Demonstrator setup

3.2. SCCharts to VHDL Compiler

The SCCharts compiler contains a dataflow-based compilation approach that converts an SCChart into a netlist. Johannsen [11] describes how to use the netlist to create VHDL code. Unfortunately, the original implementation is no longer functional and therefore has to be recreated.

The netlist generated by the SCCharts compiler is a list of calculations, that have to be executed in a given order. Variables can be written multiple times and read between these write operations in one tick calculation. For normal imperative code, such as C or Java, these statements are statically scheduled and result in a sequential sequence. This sequential approach is not suitable for FPGAs. A signal on an FPGA cannot have more than one driver and can only change during clock events. The consequence is that each statement of the netlist would consume at least one clock cycle. Variables that are written multiple times have to be split up in multiple signals to allow parallel calculations. This is done by the Static Single Assiment (SSA) transformation, available in the compiler. A netlist in SSA form can then be used to generate VHDL code. This is done by converting the complete netlist into unclocked logic expressions.

An alternative, that we do not pursue further, would be the usage of variables in an VHDL process, and thereby moving the parallelization problem from the SCCharts compiler to the VHDL compiler.

The last step is to add registers. Figure 3.7 visualizes the created RTL. All input signals and all nets that are used in the next tick are buffered in registers. The registers store the new data at the rising edge of the tick signal, and thereby initialize the tick calculation. In this setup the tick signal is the clock signal of the RTL.

This compilation approach, further referred to as original compiler, translates the SMoC of SCCharts into the SMoC of VHDL.

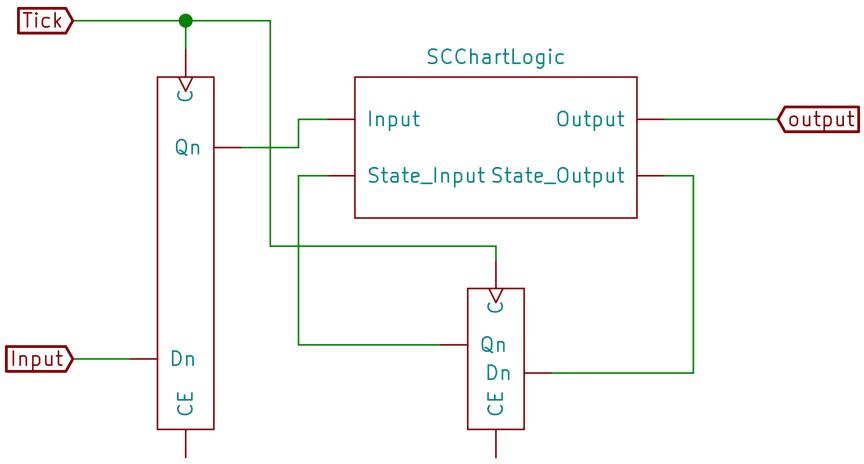


Figure 3.7.: RTL structure of the VHDL code generated by the SCChart to VHDL compiler

3.2.1. Limitations of a single Clock Approach

To understand the problem with this approach it is important to understand the different kind of uses of the SMoC. The hardware of all computers is defined in the SMoC. Clock signals control every register in the system and calculations are done in the time the registers have a stable output. It is possible to have different clock signals, but the transfer of data between different clock domains has always a penalty. This is the SMoC at the hardware level. All components at the hardware level are bound by the SMoC.

The SMoC can also be used as a theoretical construct in programming language design. SCCharts use SMoC in this capacity. It defines the fundamental structure of an SCChart program. The implications of the SMoC are limited to the SCChart program itself. Other parts of the program, such as the environment, use the interfaces of SCCharts that are based on the SMoC, but are not forced to use the SMoC model as their own Model of Computing (MoC).

Joining the hardware and the theoretical SMoC removes the abstraction layer between the SCChart and its environment. The consequence is that the hardware clock has to be slow enough for the complete tick calculation to be done in one clock cycle. A complex SCChart has a low tick rate but other components in the environment may have to use the same clock. This slows down the fast operation of the environment immensely. Furthermore, every version of the SCChart would change the clock speed again. This makes it impossible to have IO protocols implemented in the same clock domain.

It is necessary to decouple the low-level SMoC clock from the high level SMoC ticks.

3.2.2. Pipelined Approach

The term "pipelined tick function" can be interpreted in different ways. The first interpretation is that the netlist of the tick function is broken down into smaller sections. The sections are separated by a register. This stretches the calculation out to multiple clock cycles and allows to start a new tick function calculation on every clock cycle. The calculation of multiple ticks at the same time stands opposite to the premise of synchronous languages. There are only two applications for this approach. If a design requires multiple instances of an SCChart, a pipelined code can calculate multiple instances with the same hardware at the same time. The other approach is to use the pipelined code in a dynamic tick environment with fuzzy timing and prioritized events. This setup allows the dynamic tick environment to retroactively decide when to start the tick function. This approach may be worth further studies.

The interpretation we use here, is that there is only one tick calculated at a time but the logic calculation is spread over multiple clock cycles. The difference is that the operation blocks can be used multiple times in one calculation. Another advantage is that complex operations, such as division, can be implemented by sequential algorithms, such as long division. This approach reduces the consumption of LUTs and increases the consumption of FFs. In contrast to LUTs, FF are not a limiting resource on FPGAs[1].

The following paragraph outlines an approach to build a pipeline from a netlist. The described approach is not perfect. The problem of finding the best pipeline is likely at least NP-Hard.

The first step to build a pipeline, is to find the longest data path. The calculations on this path are optimized for speed by splitting them up into smaller chunks. The size of the chunks depends on the target clock speed or LUT depth. The following process is repeated with the longest remaining path until no path is left. The selected paths are analyzed for potential fragmentation. The goal is to find a fragmentation that allows to reuse as many existing logic blocks as possible. New logic blocks are created for all operations that cannot be implemented by reusing logic blocks. The new logic blocks are optimized for space efficiency. Optimization for calculation time is only done if necessary. The paths will get shorter with every iteration. This allows more aggressive space optimization.

Many necessary steps are left out in the description. For example it is necessary to track for every net the clock cycle in which it gets valid. The register, storing the net, can change every clock cycle. This must be traced too. It is not necessary to store nets, if they are no longer needed. Every iteration can change the start or endpoint of the remaining paths or split them in multiple sections.

This approach is more on the speed optimized side. In many cases a space optimization is more important than time optimizations. The reason for this is that LUTs (and DSPs) are a precious resource on FPGAs. A better space optimization can be achieved by setting the targeted calculation time before the compiler run. With a fixed time, it is possible to use more optimizations focusing on space.

[1] https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf page 12.

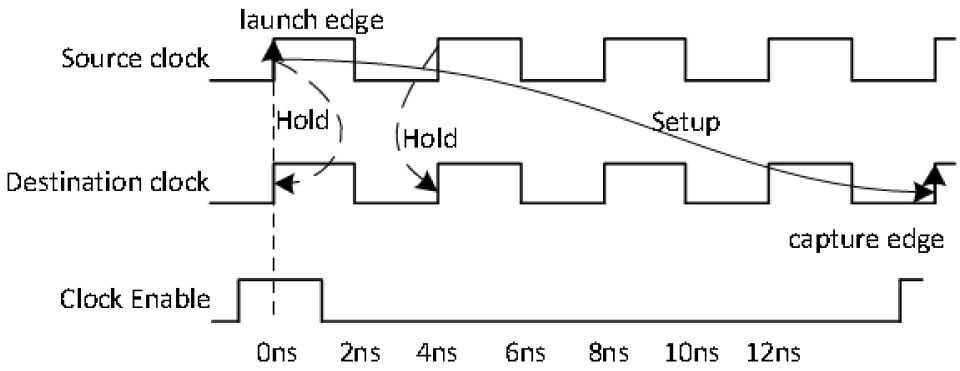


Figure 3.8.: Multicycle path with setup multiplier of four (Source: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_1/ug903-vivado-using-constraints.pdf)

This approach is preferred over the multicycle path based, due to its potential in space efficiency. However, in this report we focus on the development of an demonstrator and consider this topic future work. Therefore is this approach not used in the implementation of the demonstrator.

3.2.3. Multicycle Path based Tick Function

A Multi-Cycle Path (MCP) is a data path that has more than one clock cycle time to get from the data input register to the output register. Defining an MCP moves the setup time to another edge of the clock signal.

Figure 3.8 shows the setup and hold timings of a multicycle path with a length four. The setup is moved forward by three rising edges of the clock signal. The hold is kept at the same clock edge. The output of the input register to the multicycle path has to be stable for four clock cycles. The example uses a CE signal to ensure that the registers are changing every four clock cycles.

This technique can be used to embed the big logic calculation of the SCChart in a fast clocked environment. To use this technique the RTL and interface has to be changed. The adapted RTL and interface is shown in Figure 3.9.

The new interface has separate Tick and CLK signals. CLK is the clock signal for the FFs. To start a new tick calculation, the Tick signal must be set to '1' for one cycle of CLK. The Tick signal is connected to the CE port of the input and state register. This way the input data is stored in the input registers at the next rising edge of the clock. The tick trigger signal is delayed for the specified amount of clock cycles by being passed through the delay registers. Before reaching the last of the delay register the CE of the output registers is set to '1' for one cycle. As a consequence the calculation result is stored in the output registers at the next rising clock edge. The output of the last delay register is written to the TickDone signal. TickDone is a new signal of the interface. This

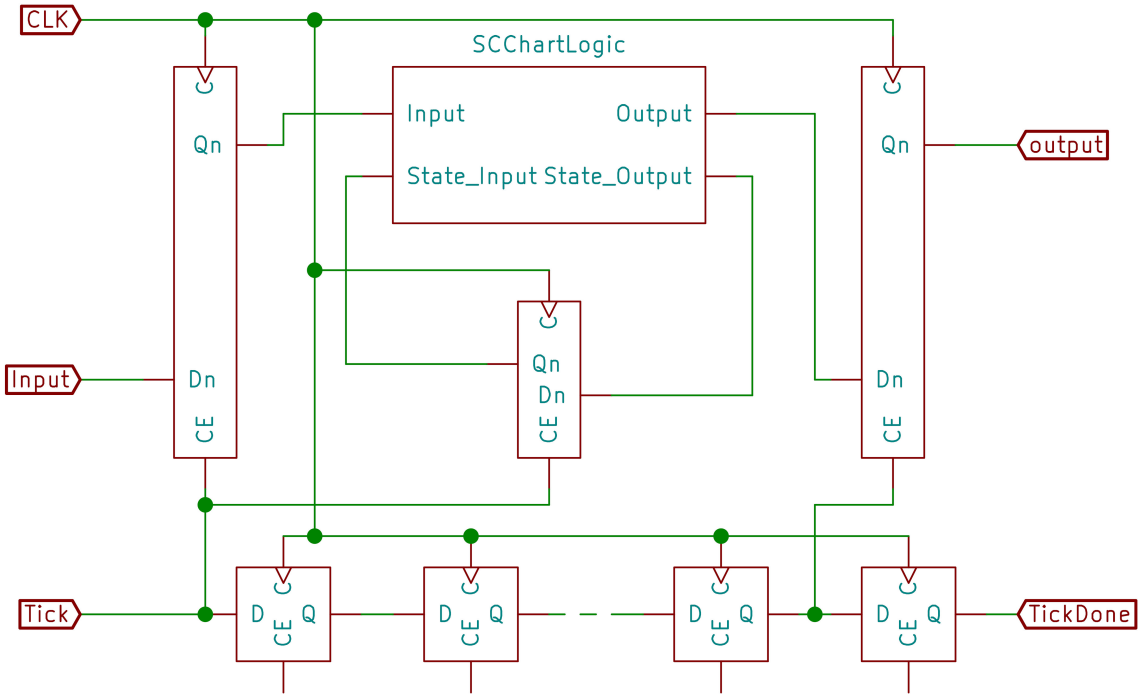


Figure 3.9.: RTL of the VDHL code generated by the improved SCChart to VHDL compiler

signal indicates that the tick calculation is done.

The TickDone signal is used by the environment to delay the triggering of a tick calculation while a calculation is already running. Starting a tick calculation while a calculation is already running would lead to an undefined state of the state registers.

The input of the logic is stable between the start of two ticks. The output register keeps all intermediate results of the logic from leaking out. This way the output is a valid output in every clock cycle. The number of delay registers defines the maximal length of the multicycle path. The length of the output register can be set via VHDL generics. For the generated MCP constraints a variable can be used to set the length of the multicycle path. This allows to change the calculation time without recompiling the SCChart.

The MCP approach is a simple and effective way to get the SCChart to VHDL compiler in a state suitable for the use in FPGAs. This approach is implemented for the purposes of this report. For productive use it is recommend to extend the compiler with the ability to generate the necessary timing constraints for multicycle paths[2].

[2] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_1/ug903-vivado-using-constraints.pdf

3.2.4. Wrap Up

Both approaches decouple the tick and clock signal. Therefore they solve the most important problem. However there are more problems with the original approach, for example the used SCChart compile chain and SSA is optimized for sequential platforms as mentioned before. This and other problems have to be addressed before a complex compiler such as the pipeline based is developed. More information in this regard can be found in Section 6.1.2. The MCP-based option is much simpler to implement and keeps most of the structure described by Johannsen [11]. Therefore this approach is implemented here.

3.3. Environment

The environment, displayed in Figure 1.4, has to be implemented for every target platform. The components of the environment are the time management module, the trigger management system, the input module, and the output module. These modules have clear interfaces and can be implemented and replaced independently. Figure 3.10 contains the pseudocode of a dynamic tick environment tying these components together. There is one additional signal path: The tick signal is fed back to the time management. The time management needs the tick signal to know when the last tick function was called.

```
while true do
    repeat
        (eTrigger, input) = ReadInput();
        (tTrigger, deltaT) = UpdateTime(sleepT, tick);
        tick = eTrigger or tTrigger;
    until tick;
    (output, sleepT) = Tick(input, deltaT);
    WriteOutput(output);
end
```

Figure 3.10.: Pseudocode of a dynamic tick environment

3.3.1. Time Data Type

In *Time in SCCharts* [15] the time signals deltaT and sleepT were introduced as real numbers. Real numbers are easy to handle, and they allow to define any time interval by using seconds as a base unit. A milisecond is just written as 10^{-3}. However it is in general not possible to store real numbers in computers. There are two main approximations for real numbers: Floating point numbers and fix point numbers.

Floating point number based time

The advantage of floating point time is that they are easy to handle. They can be used as if they are real numbers. Furthermore, they have a huge range of possible values. A 32 bit floating point number can store time values from the shortest measurable time interval to the age of the universe. The problem is that the mathematical operations have precision errors. The precision of an addition depends on the difference of the values. $a + b = a$ if a is much bigger then b. A 32 bit floating point counter, counting nanoseconds will never reach the value one second. Even if the error of addition, in the selected application, is small, the error is accumulated due to the delta-based interface design. The adding up of deltaT is a form of chain dimensioning. The problem with chain dimensioning is that not only the values are added up but also the errors. This leads to ever increasing errors in time tracking.

The last downside of floating point numbers is that for calculations a Floating Point Unit (FPU) is needed. Not every platform has an FPU. FPUs are more complicated and slower in calculation than integer units. On FPGAs floating point operations take much more resources compared to integer operations.

Fix point number based time

A fix point number is not a basic data type of modern computer architectures. Fix point numbers are created from integers. The simplest fix point number is an integer and the declaration of an denominator regarding the integer. An example of a time type using integer and a declared denominator is TimeSpec[3].

The advantage of this time types is that they have no calculation errors. Chain dimensioning, as specified by the interface, is therefore no problem. With fix point numbers a tight timing can be created. One disadvantage of fix point numbers is that they are more complicated to handle. Another problem is that their range is limited and therefore more bytes are needed to express big numbers. The handling of fix point time types can be improved by creating a smart constructors and a math library for them.

Conclusion

For the environment implementation of this demonstrator we prefer fix point numbers. Floating point numbers could be used in some cases when the time keeping of the system is not precise anyway or the smallest and biggest deltaT value are close enough together, or the rounding error aggregation is no problem. However, here we have hard requirements regarding timing and must prevent imprecisions and computation slowdown. Further improvements to the time data type are suggested in the Section 6.1.3.

[3] https://en.cppreference.com/w/c/chrono/timespec

4. Implementation

This chapter presents all components of the built demonstrator, illustrated in Figure 4.1. This includes the developed software and components only needed for the analysis. The first section provides an overview of the demonstrator, including descriptions of the interfaces and the functionality. The next three sections cover its main components: controller, power stage and motor assembly. The last section presents the components necessary to run the analysis.

4.1. Demonstrator

Figure 4.1 shows an annotated image of the demonstrator. The core component is the motor assembly on the right side. It contains two stepper motors mounted in a 90 degree angle. One motor has a disk and the other has three sticks mounted to its shaft. If the motors are running synchronized, they can pass through each other. Each stepper motor is connected to one motor driver, located in the middle. The stepper motors and the

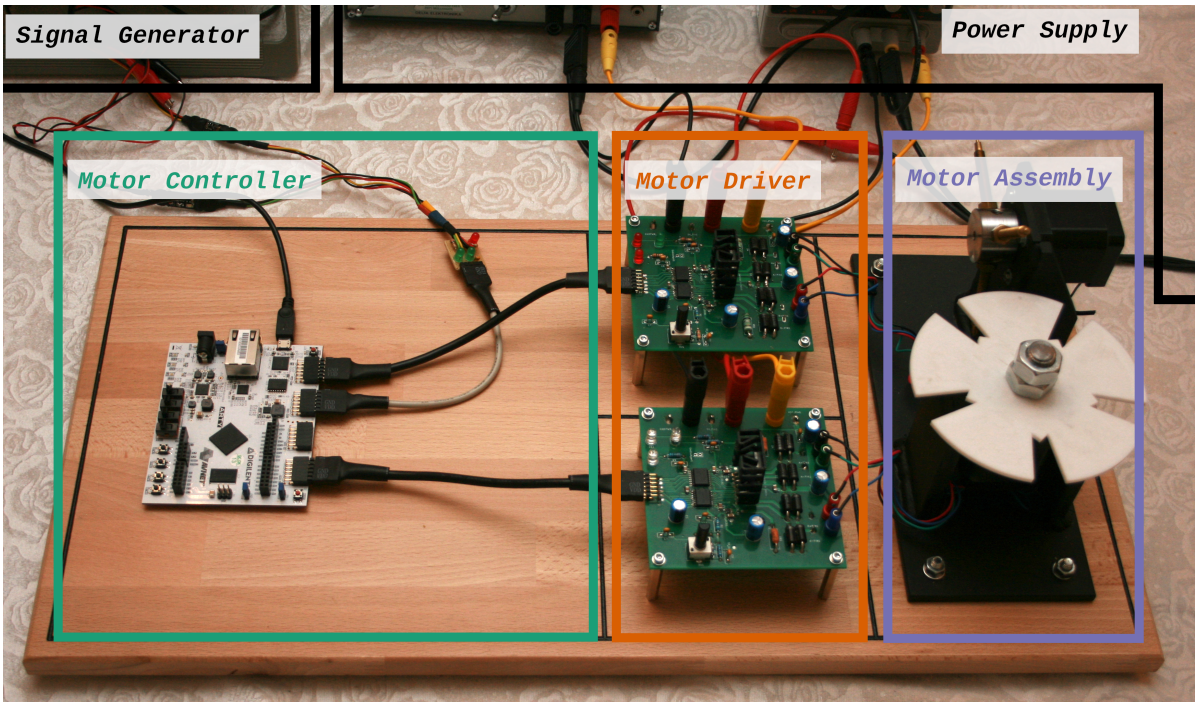


Figure 4.1.: Demonstrator setup

motor drivers are connected with four wires, connecting each of the motor's two coils to an H-bridge on the motor driver.

The control signals for the H-bridges of the motor drivers are received via a PMOD-cable from the motor controller. The motor controller, on the left, gets a square wave signal from a signal generator (above), representing the target speed of the system. This signal is used to generate the motor control signals.

An important feature of the motor driver is the over current detection. If the motor draws too much current, an overcurrent detected signal is sent to the controller. This signal is used by the controller to implement an over current protection.

The setup, displayed in Figure 4.1, is powered by three different power supplies, visible in the upper part. The logic of the motor driver is powered by a 5 V power supply that is not on the photo. The two power supplies on top of the image are each connected to one motor driver. This setup allows a better surveillance of the current drawn by each motor, and is otherwise not necessary.

4.2. Motor Assembly

The motor assembly is the core of the demonstration. This section describes the main features of the motor assembly. Technical drawings in Appendix A, provide more detailed information about the components. The drawing of the motor assembly in Figure 4.2, allows the inspection of the most important components of the motor assembly, the rod and the disk assemblies.

The disk assembly (Appendix A Sheet 11) consists out of five components: the disk mount, the disk, a grub screw and two hex nuts. The disk mount (Appendix A Sheet 6) is machined form an M14 bolt. A grub screw is used to mount it on the motor shaft. The disk is put on the bolt and fixed in place with two countered hex nuts. The disk is 3D printed using white PETG[1]. The disk (Appendix A Sheet 7) has a diameter of 120 mm and is 3 mm thick and has five slots to let the sticks pass through.

The rod mount (Appendix A Sheet 8) is turned from S235 Steel. A grub screw is used to fix the mount on the motor shaft (Appendix A Sheet 12). The grub screw shares the hole with a rod. The set screw and rods are located on opposite sides of the motor shaft. The rods (Appendix A Sheet 9) are made out of M6 brass threaded rod. The overall length of the rod is 60 mm.

The backbone of the of the motor assembly is the motor mount, made from S235[2] steel (Appendix A, Sheet 10). The components of the motor mount are machined from 30x8, 50x8, 100x6 flat steel and welded together in the MIG welding process. The technical drawing of these components are in Appendix A on Sheets 1-5. The stepper motors are bolted to the motor mount with four M3 screws. The motor mount it self is mounted to the base plate with four M6 carriage bolts.

[1] https://filament2print.com/gb/blog/49_petg.html

[2] https://www.worldsteelgrades.com/1-0038-steel-s235jr-material/

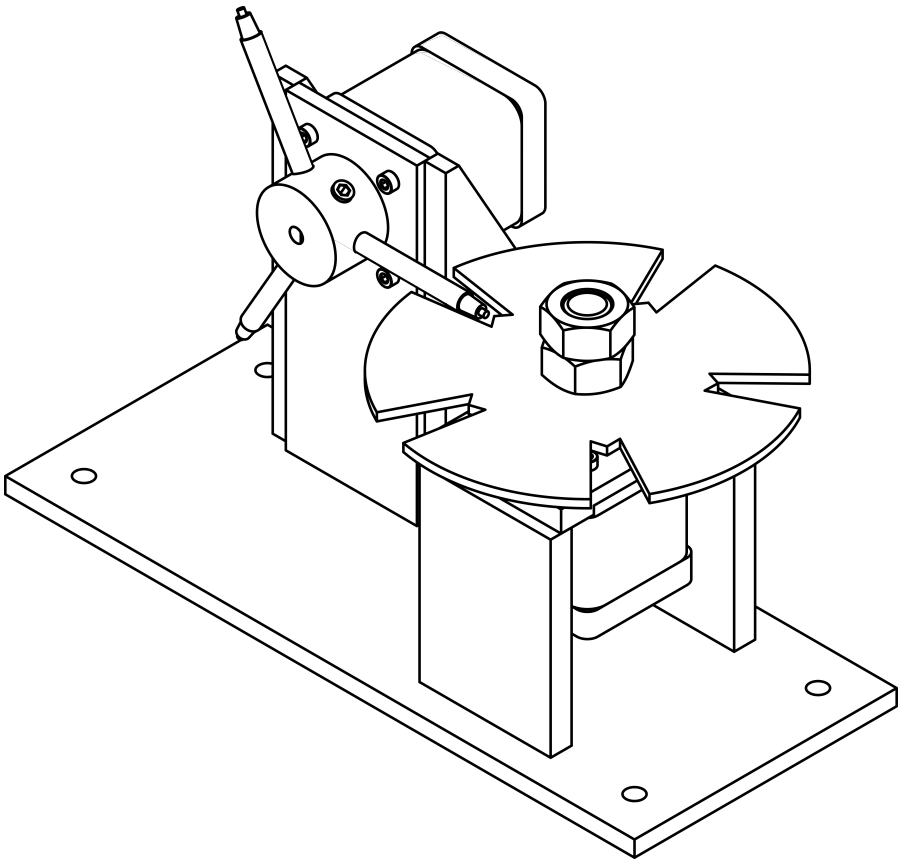


Figure 4.2.: Technical drawing of the motor assembly

The stepper motor used are 17HS5415P1-X6[3] manufactured by ACT MOTOR. It is a bipolar hybrid stepper motor with a 4 wires interface. The motor type is NEMA 17 [4].

4.3. Motor Driver

A 4 wire stepper motor has two coils. To run the stepper motor, these coils need to be magnetized in the right order and alternating direction. To drive the coils an H-bridge is needed. This section describes the design of an L298 based stepper motor driver. The L298 is a double H-bridge IC. The board is designed to interface with an FPGA via a PMOD header. Galvanic isolation and over current sensing are key features of this board.

4.3.1. Requirements

The basic requirement is that the motor driver can drive a 4 wire stepper motor. Therefore the board needs two H-bridges. To use high voltages at a low RPM, an overcurrent

[3] https://cdn-reichelt.de/documents/datenblatt/X200/17HS5415P1-X6DATASHEET.pdf
[4] https://www.nema.org/Standards/SecureDocuments/ICS16.pdf

1	A+ half-bridge
2	A- half-bridge
3	enable H-bridge A
4	Overload detected on H-bridge A
5	GND
6	VCC
7	B+ half-bridge
8	B- half-bridge
9	enable H-bridge B
10	Overload detected on H-bridge B
11	GND
12	VCC

Table 4.1.: Pinout PMOD headers

protection is needed. The overcurrent protection is part of the controller. For the controller to be able to react to an overcurrent, a digital feedback signal is needed. The stepper motor is a inductive load, therefore an overvoltage protection is needed for the H-bridges. Furthermore, capacitors are needed to buffer the changes in current drawn from the power supply. To protect the motor controller in case of any failure of the motor driver, the board needs to have a galvanic isolation.

4.3.2. Interface

The stepper motor driver interfaces with three other components: to the FPGA, to the power supply and to the stepper motor.

Interface to FPGA

To interface with an FPGA or another controller, this board uses a PMOD[5] 12 pin header. None of the standard PMOD™ interfaces fits for this board, therefore the General-Purpose Input/Output (GPIO) standard is used. Table 4.1 contains the used pin assignment.

Interface to PSU

The L298 board needs two different supply voltages: 5 V logic level and the motor supply voltage. The motor supply voltage may not exceed 45 V. It is advised to use a current limiting power supply to protect the motor in case of an error in the controller. The power is supplied to the board via three 4 mm banana plugs, see Table 4.2.

[5]https://reference.digilentinc.com/reference/pmod/specification

Function	Color	$V_{min.}$	$V_{typ.}$	$V_{max.}$
GND	black	-	0	-
VCC_{logic}	red	4.5	5	5.5
VCC_{motor}	yellow	-	-	46

Table 4.2.: Color code for power supply banana plugs

Function	Color
A+	black
A-	green
B+	red
B-	blue

Table 4.3.: Color code for motor banana plugs

Interface to Stepper Motor

The stepper motor is connected with 2.6 mm banana plugs. The board uses female connectors and the motor male connectors. The Table 4.3 shows the color codes.

4.3.3. Circuit Diagram

The first step in PCB design is the creation of the schematic. The schematic describes which components are used and how they are connected. It is common practice to split the schematic into multiple parts that are focused on one functionality. The circuit diagram of the motor driver is split up in six parts.

The core feature of this board is the Motor driver with current sensing, realized by a Dual H-bridge with shunt resistors. The the signal generated by the current sensing is used by the over current detection. The overcurrent detection needs a reference value do decide if the current limit is reached. It is provided by the adjustable reference voltage source. The control signal for the H-bridges and overcurrent detected signals are exchanged with the motor controller via a galvanic isolation. The galvanic isolation protects the motor in case of a fault in the motor driver. The power stabilization ensures the stability of the supply voltages and thereby the correct behavior of the ICs. The last schematic describes the function of the LEDs on the PCB.

The following sections contain descriptions of these components including the relevant parts of the schematic. The complete schematics of the motor driver and the other PCBs are in Appendix B.

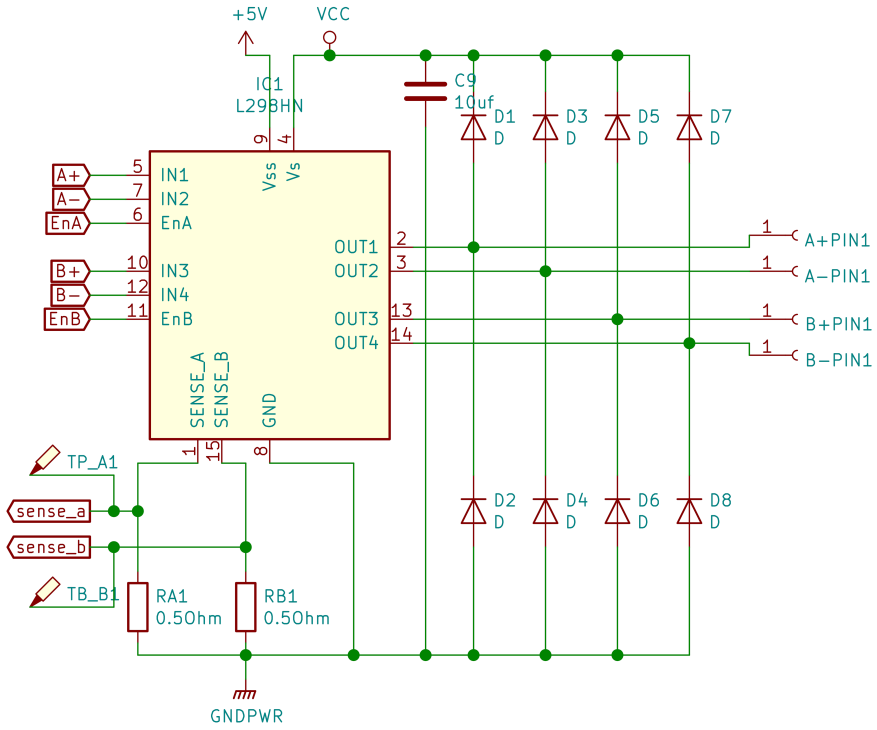


Figure 4.3.: Double H-bridge l298 with protection diodes and shunt resistors

H-Bridge

This board uses the dual H-bridge IC, L298[6]. The schematic is in Figure 4.3. The L298 IC has no internal over voltage protection. The over voltage protection is done by the diodes D1 to D8. An advantage of the L298 is that the low side of the H-bridges are not connected to ground. Instead the low side of each H-bridge is connected to their own sense pin. This allows the use of load sensing resistor between the H-bridge and the ground (RA1 for H-bridge A and RB1 for H-bridge B). The voltage drop over the resistor can be measured at the test pins (TP_A1, TP_B1) and is used by the over current detection.

Over Current Detection

The over current detection, Figure 4.4, is realized by the high speed comparator LMV7219[7]. The over current is detected when the voltage created by the load sensing resistors exceeds the reference voltage.

[6]https://www.mouser.de/datasheet/2/389/l298-954744.pdf
[7] http://www.ti.com/lit/ds/symlink/lmv7219.pdf

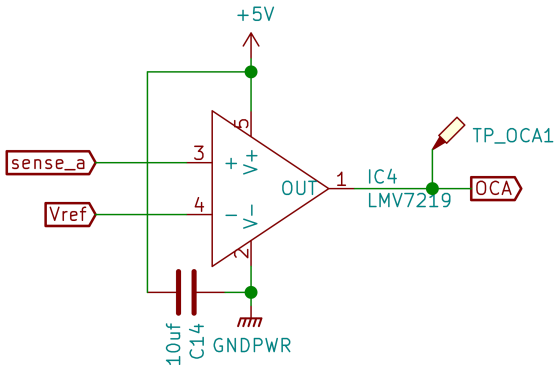


Figure 4.4.: Overcurrent detection

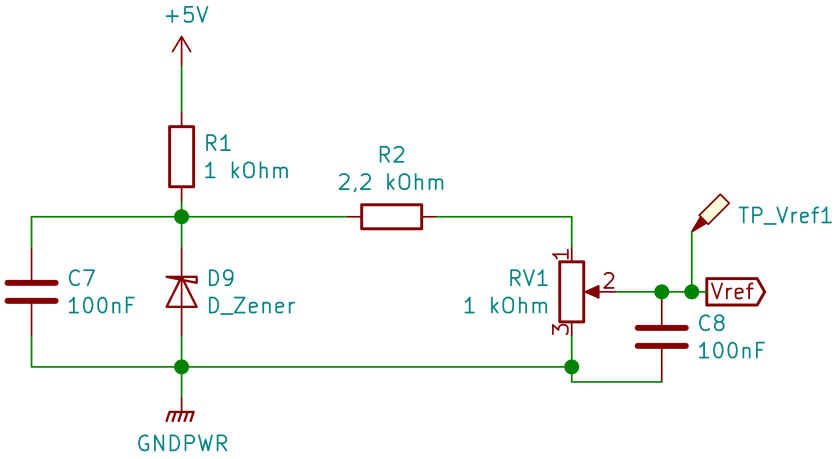


Figure 4.5.: Reference voltage source

Reference Voltage Source

The Zener diode based reference voltage source, Figure 4.5, produces a reference voltage between 0 and 1 volt. The voltage can be changed by RV1. The reference voltage, set with RV1, directly correlates to the current limit that is detected by the over current detection.

Galvanic Isolation

The schematic of the galvanic isolation is displayed in Figure 4.6. ISO7741[8] galvanic isolators are used for the galvanic isolation of the logic signals. ISO774x are high speed quad channel digital isolators. The ISO7741 variant has 3 forward and 1 backward channel. This digital isolator type is chosen for its fast propagation time of 7 ns and

[8]http://www.ti.com/lit/ds/symlink/iso7741-q1.pdf

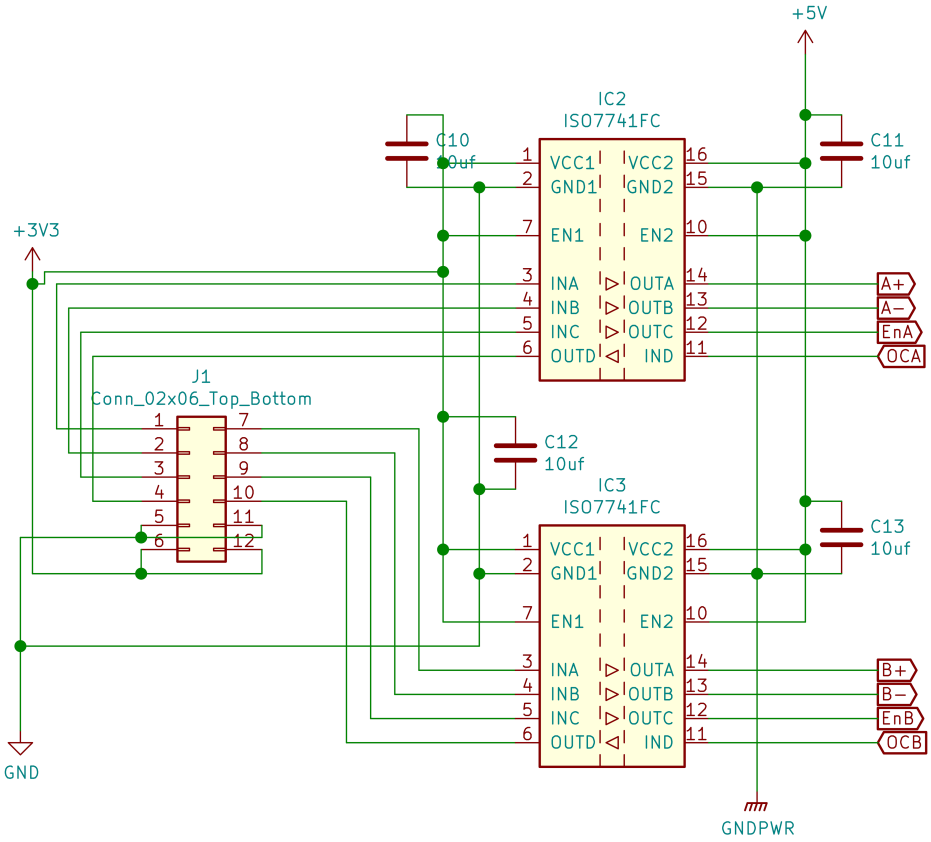


Figure 4.6.: Galvanic isolation

a low jitter. A fast isolator is necessary since the overcurrent protection is part of the controller. Therefore the overcurrent detected signal and the resulting shutdown signal for the H-bridge have to path the isolator, adding two propagation delays to the overall reaction time. Another benefit of faster isolators is that they have a lower impact on the analysis results.

Power Stabilization

All supply voltages are stabilized with capacitors, see Figure 4.7. Every IC has its own capacitor for a local power stabilization, displayed in their schematic. This board has two ground levels due to the galvanic isolation: GND and GNDPWR. GND is the ground level of the control board connected to the motor driver via PMOD header. The ground level GNDPWR is the ground level of the motor power supply. CG1 allows to couple the grounds with a capacitor or a resistor, to prevent a build-up of a too high difference in the two ground levels. If both ground levels are the same, GndJumpler1 can be used to connect them together.

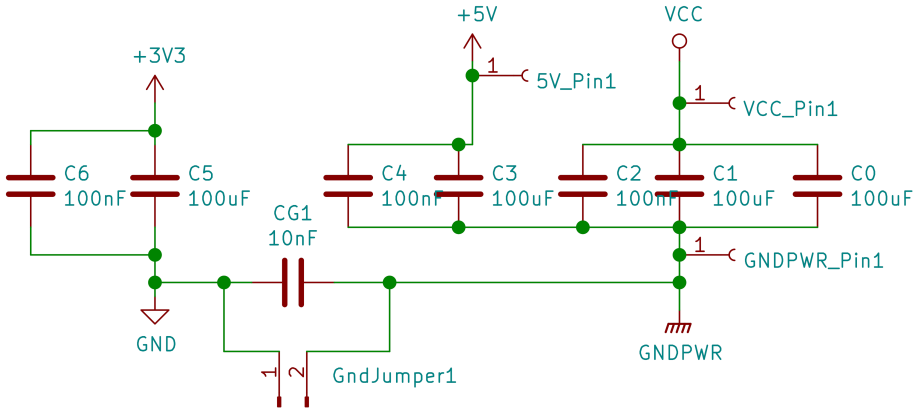


Figure 4.7.: Power stabilization

LEDs

Often overlooked or seen as a gimmick, LEDs can fulfill a vital role on a PCB. They allow to debug problems fast by showing information about the internal state of the PCB. There are multiple possible uses for debugging LEDs on a stepper motor driver, as a supply voltage indication and as an visualization for digital or analogous signal, such as over current detected, motor speed or any other control signal.

To keep the demonstrator simple, this board only uses supply voltage indication LEDs. Digital signals, such as over current detected, are visualized by the LEDs on the controller board. There are three supply voltage LEDs. The LED D10 indicates that the PMOD header provides logic level voltage. D11 is connected to the 5 V logic level supply voltage of the motor driver. The last LED D12 is connected to the motor supply voltage. The motor supply voltage is between 5 V and 45 V. For simplicity, the LED does not use a constant current source to compensate for the wide range of input voltages. This leads to a very dim LED at low motor voltages. Figure 4.8 presents the related schematic.

It is necessary to decide the form factor of each component before the schematic can be turned in a PCB. A combination of THT and SMD components is used to enhance the visual appearance of the board.

4.3.4. PCB Layout

The next step is to build the layout of the PCB itself. The main tasks are the placement of the components and the routing of the circuit path.

The placement of the components influences the complexity of the routing and performance of the PCB. The placement of the components is normally optimized to reduce the routing complexity. It is also possible to place components a way that aids explanations of the component's function and their interactions. This is an advantage in a demonstrator.

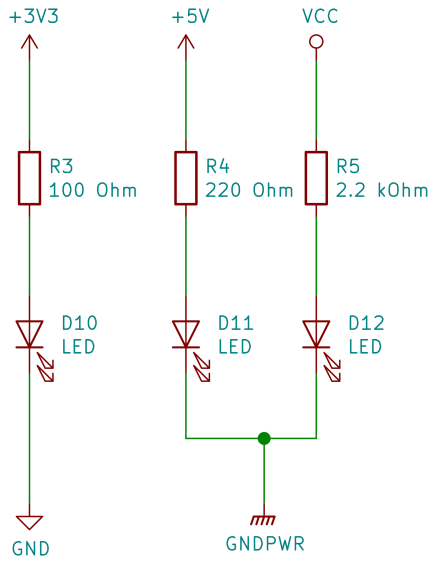


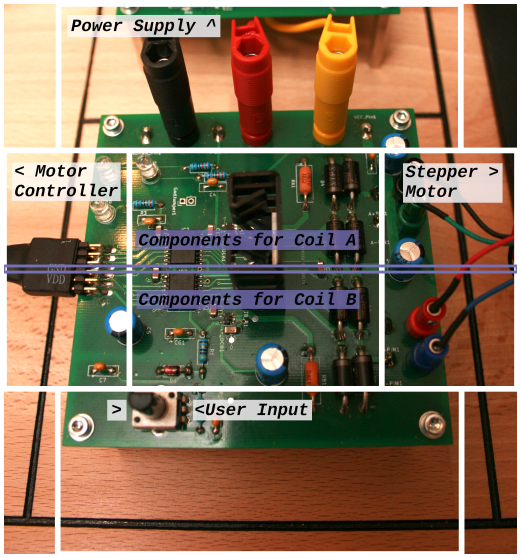
Figure 4.8.: Power indication LEDs

Figure 4.9 contains two images of the motor driver. The motor driver PCB design is optimized for demonstrations. The first optimization is the clear position of the interfaces. In Figure 4.9a the interfaces are annotated in white. On the top of the PCB are the connectors for the power supply. The right side of the board is reserved for the connection to the slave components, in this case the stepper motor. The bottom of the PCB contains the user interface, in this case the potentiometer of the reference voltage. The one exception of these rules are power LEDs. They are placed near the power connectors on the top. The left side is used for the connection to the master. The master of this board is the motor controller. This design pattern is used in all PCBs designed for the demonstrator.

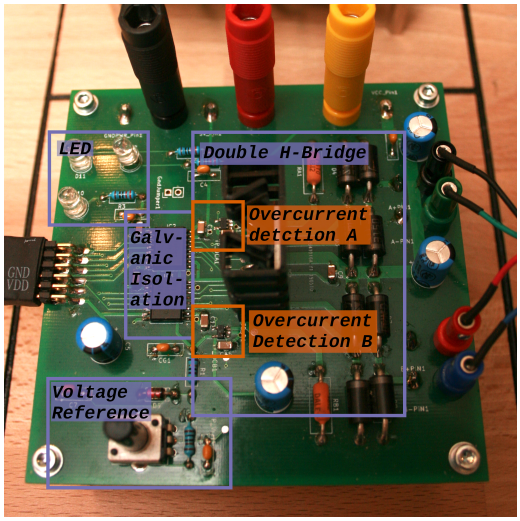
The other design role that is used for the PCB is to keep function groups together. Figure 4.9b shows the positions of the function groups, described in the previous sections.

The last big design decision is specially for this PCB. This motor driver has two H-bridges, one for each motor coil. The blueish lines in 4.9a split the galvanic isolation, overcurrent detection and the H-Bridges in two. Components above the lines drive coil A, and coil B is driven by components below.

Figure 4.10a is the 2D CAD model of the PCB. This kind of view is used during the design of the PCB. The 3D version, Figure 4.10b, is generated out of the same board file, but is not used during development, except to check the appearance of the board before it is produced.

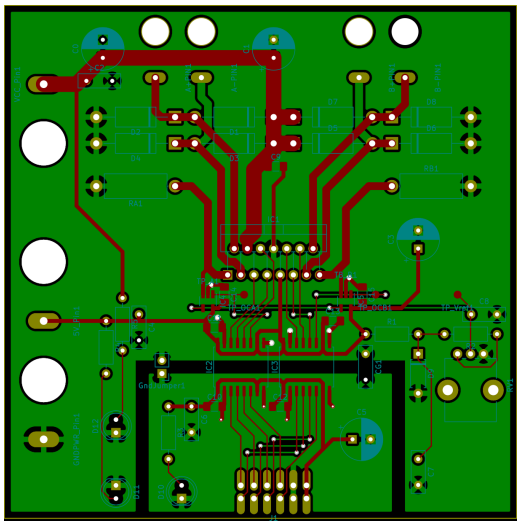


(a) Structure overview

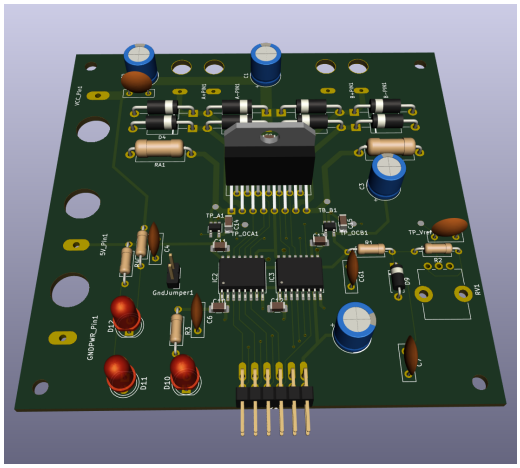


(b) Functional groups

Figure 4.9.: Annotated motor driver images



(a) PCB



(b) PCB in 3D

Figure 4.10.: Motor Driver

4.4. Motor Controller

The base of any controller is the hardware. The primary controller hardware, used in this report, is an FPGA. The alternative platforms are a Raspberry Pi and an ATmega. They are used to run the Controller SCChart in a Dynamic Tick Environment (DTE), see Section 4.5 and 4.6.

4.4.1. FPGA

The Arty A7[9] is a development board for the Artix 7 35 and Artix 7 100. The board is produced by DIGILENT. We use an Artix 7 35 for this demonstrator. An advantage of the board is that a USB programmer is already included and the board can be powered via USB. The features of the board, used for the demonstrator, are the four 12 pin GPIO PMOD[10] headers, the buttons and the LEDs. The PMOD headers are used to connect to the motor driver and the LEDs and buttons as user interface. The onboard 100 Mbit Ethernet is used in analysis to get the data of the logic analyzer circuit to the PC.

The Artix 7 line-up of FPGAs is designed for performance per watt and low cost. The 7 series [13] of FPGAs continue the trend that the amount of special purpose slices, such as Digital Signal Processor (DSP), are increased.

4.4.2. Raspberry Pi

The Raspberry Pi is a single board computer and the origin of the modern development of them. In the test setup a Raspberry Pi 3 is used with Raspbian as the operation system.

The Raspberry Pi is a single board computer has a 40 pin GPIO header. To connect PMOD boards to a Raspberry Pi, an adapter is needed. The simplest possible adapter are loose wires. This approach is not practical to use and less reliable than a PCB based adapter. The big benefit of designing an adapter PCB is the ease of use and the repeatability of a setup. The Pi2PMOD board is a daughter board for the Raspberry Pi3 B. It converts the GPIO pin header into 3 PMOD pin headers. Furthermore the adapter allows to power the Raspberry Pi with 4 mm banana jacks and displays the current state of the GPIO pins with LEDs.

Interface The board has two 4 mm banana plugs for power supply: a black one for ground and a red one for 5 V VCC.

To connect to the Raspberry Pi the 40 pin GPIO pin header is used. The mounting holes of the Raspberry Pi are also available on this board. This allows to use bolts to mechanically connect this board to the Raspberry Pi.

The board supports up to three PMOD boards that can connect to the PMOD headers A, B and C. The three PMOD headers have the distance defined by the PMOD standard. This allows the usage of slave boards with multiple PMOD headers.

The board has a power LED for the 5 V power rail and three groups of eight LEDs, one group for each PMOD header. For every group a freeze signal exists to freeze the current state and to display it as long as needed. The LEDs represent the PMOD pins 0 to 7 from right to left.

Interface Interconnection The board is designed as a passive adapter between the Raspberry Pi 3 header and PMOD headers. The only active components on this board are

[9] https://store.digilentinc.com/arty-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/
[10] https://www.digilentinc.com/Pmods/Digilent-Pmod_%20Interface_Specification.pdf

PMOD	PMOD pin	LED group	LED number from left	GPIO	Header pin
A	0	left	8	8	24
A	1	left	7	10	19
A	2	left	6	9	21
A	3	left	5	11	23
A	4	left	4	12	32
A	5	left	3	13	33
A	6	left	2	14	8
A	7	left	1	15	10
B	0	center	8	0	27
B	1	center	7	1	28
B	2	center	6	2	3
B	3	center	5	3	5
B	4	center	4	4	7
B	5	center	3	5	29
B	6	center	2	6	31
B	7	center	1	7	26
C	0	right	8	18	12
C	1	right	7	20	38
C	2	right	6	19	35
C	3	right	5	21	40
C	4	right	4	22	15
C	5	tight	3	23	16
C	6	right	2	24	18
C	7	right	1	25	22

Table 4.4.: Connection between the interfaces

the LED drivers that are used to minimise the effect of the logic level virtualisation on the logic signals themselves. Table 4.4 describes the connection between the three interfaces.

The PMOD headers A and C are designed as SPI PMOD connectors. The pin header B is optimized for fast IO by being able to write data to the GPIO without rearranging the bits.

Furthermore, the GPIO pin 17 enables the live view of the data on PMOD header A, in case the Jumper 2 is set accordantly. The same is true for GPIO pin 26 PMOD Header B and Jumper 3 and GPIO pin 16 PMOD Header C and Jumper 1. The led D24 is controlled by GPIO pin 27 and the LED D25 is the power led of the 5 V power supply.

Circuit Diagram The only IC used in the schematic, Figure 4.11, are level-controlled D-latches. The D-latches are used to decouple the LEDs for the data signal. This is

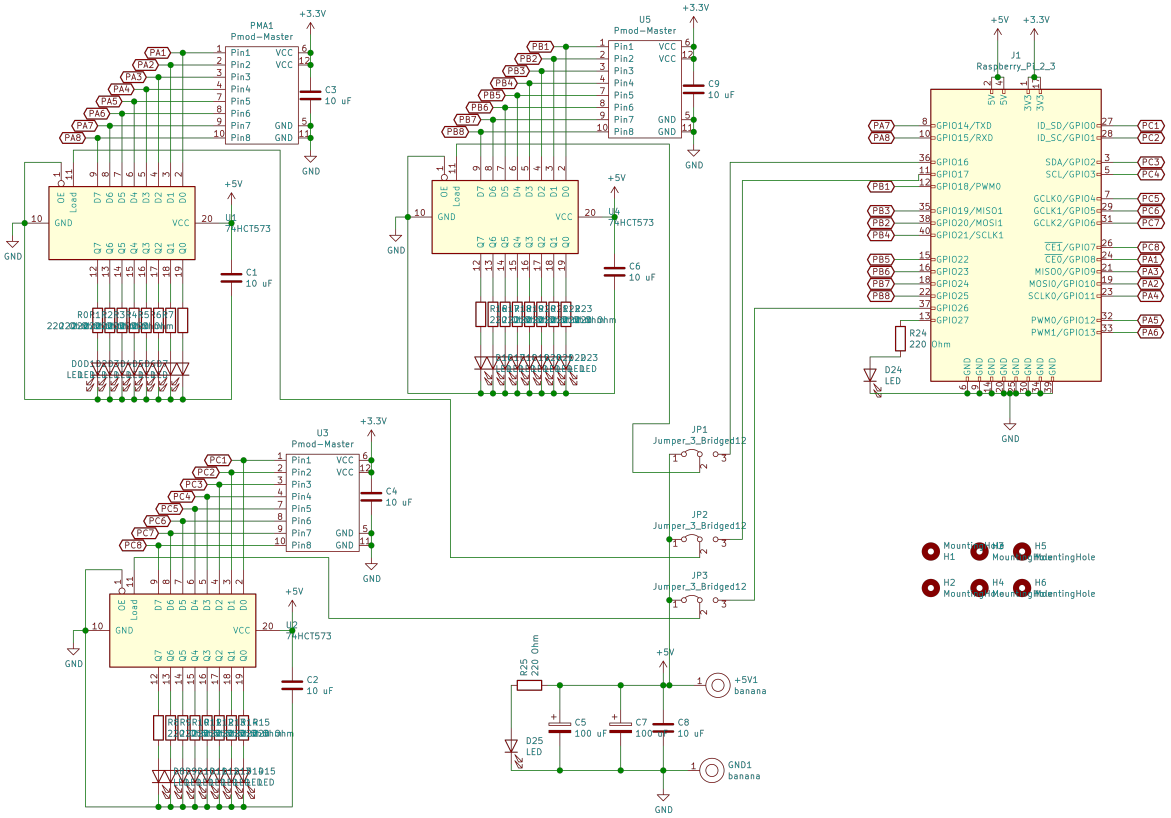


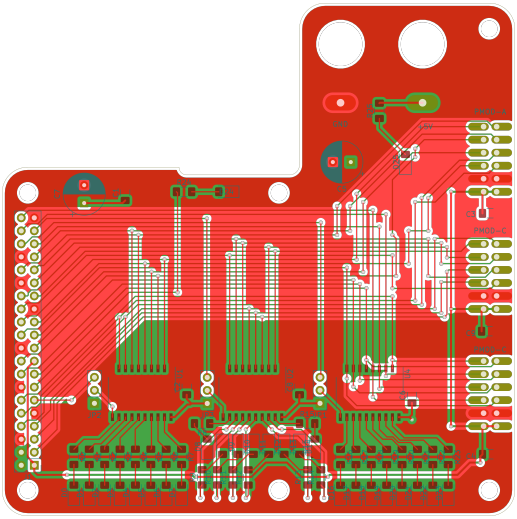
Figure 4.11.: Pi2PMOD schematic

necessary to stop any distortion the LED would apply to the logic signal. Furthermore, it protects any connected board with low maximal output load. The used latches are of the type 74xx573. There are multiple possible 74 families, that can fulfill this role. The only important requirement is that 3v3 CMOS signals are recognized, while the supply voltage is 5 V.

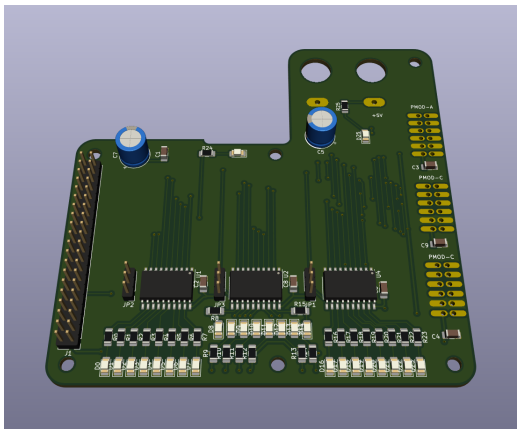
PCB layout The PCB layout follows the rules described in Appendix C. The power supply enters the PCB on the top. Master connectors are on the right, slave connectors on the left side. User input and output is positioned on the bottom with the exception of power status LEDs. Figure 4.12a is the CAD view of the board and Figure 4.12b the 3D version.

4.4.3. ATmega

This microcontroller board features an ATmega32u4 running at 8 MHz and on 3.3 V. Peripherals can be connected at the 3 PMOD headers. The MCU is programmed via USB. The USB port is a native USB device and can therefore also be used for other



(a) PCB



(b) PCB 3D

Figure 4.12.: Pi2PMOD

common USB applications, such as to exchange data or to simulate a HID[11].

The ATmega32u4 is an AVR-based MCU. The AVR architecture is widely used in many maker projects. The good documentation, feature rich instruction set and the good software and hardware support are the key reasons for the popularity of AVR MCUs.

The key selling point of the Atmega32u4 is the full speed USB port. The hardware USB port enables fast communication with PCs and allows direct programming with a bootloader. Since the USB port is a native USB port and not a USB to serial converter, it can be used to simulate HIDs or other USB device types.

The ATmega series of MCU is supported by a wide rage of open-source tools such as avr-gcc, avrdude, ardruion and many more. The Ardurio Leonado uses the same MCU, therefore the arduion bootloader can be use to program the ATmega32u4. There is only a small alteration needed. The Leonado board runs at 16 MHz, while this board runs at 8 MHz. Therefore the PLL of the USB core needs to be configured accordantly.

A lower core frequency is necessary, because this board has a supply voltage of 3.3 V. The Leonado board uses 5 V. The advantage of 3.3 V is that this way a direct connection between this board and a Raspberry Pi or FPGA can be created. The downside is the lower core frequency. This decision may be revisited in a later version of this board[12].

Interface This PCB has three PMOD connectors: PMOD-B, PMOD-D, PMOD-F. The naming is based on the IO Port of the ATmega mainly used for the PMOD connector. The pin numbers of Port-B and PMOD-B line up exactly. This allows fast IO operations. PMOD-B has the limitation on possible loads during ISP programming since the ISP pins are part

[11] https://www.usb.org/hid

[12] With the implementation of the logic Sniffer (Section 4.8.3), the necessity of 3.3 V communication is removed. A new revision of this board can therefore operate at 5 V.

of Port-B. This limitation can be resolved by using a bootloader and USB programming. PMOD-D is connected to Port-D, but with mixed up pin connection. The fourth pin of the PMOD-D header can be connected to Port-E pin 6 by changing the position of a jumper (JP5). This way PMOD-D has a external event pin on the pins that are used for over current detection on the motor driver. This can be used to implement a fast over current protection. IO operations on PMOD-F are slower because some of its pins are connected to Port-C. If the JTAG interface is used, the upper four pins of this header cannot be used.

This board has an native USB 2.0 client port and a standard 10-pin AVR-ISP programming header for ISP programming. This header can be used to program a bootloader. This programming header shares some pins with the PMOD-B header. Note that, it can be necessary to remove any slave board from PMOD-B before programming. The main usage of this header is the flashing of the bootloader that enables programming via USB.

The ATmega 32u4 has a JTAG interface for debugging. This pin header shares pins with the PMOD-F header. The upper 4 pins of PMOD-F can not be used, if JTAG is used.

The board has four banana plugs for power supply: The black one is the ground, the green is the input for the 3.3 V supply current used for the ATmega, the red is directly connected to the USB port and can be used as a power source for a DC-DC converter to create the 3.3 V for the ATmega.

Circuit Diagram Figure 4.13 shows the complete schematic of the board. Most of schematic is standard or a design already used on one of the other boards. There are two points worth mentioning. The first are the two LEDs (D1,D2), that can be controlled by IO pins. They can be used as debug LEDs. The other thing is the use of a ferrite beat as a filter for the 3.3 V voltage for the ATmegas ADC.

PCB Layout The PCB layout follows the rules used in this report for PCBs. Power and other supply wires are entering the PCB on the top. Master connectors are on the right, slave connectors on the left side. This implies that slave boards are always located right to the master board. User input and output is put on the bottom with the exception of power status LEDs.

The most critical nets are the D+ and D- of the USB interface. They are kept always as close as possible and have the same length. Figure 4.14a is the CAD view of the board and Figure 4.14b is the 3D view.

4.5. SCCharts Controller

This section describes the motor controller software. This software takes the speed signal as an input and derives the motor control signals for the two stepper motors. The two motor rotation speeds always keep a ratio of 3 to 5, to avoid collisions. Four button

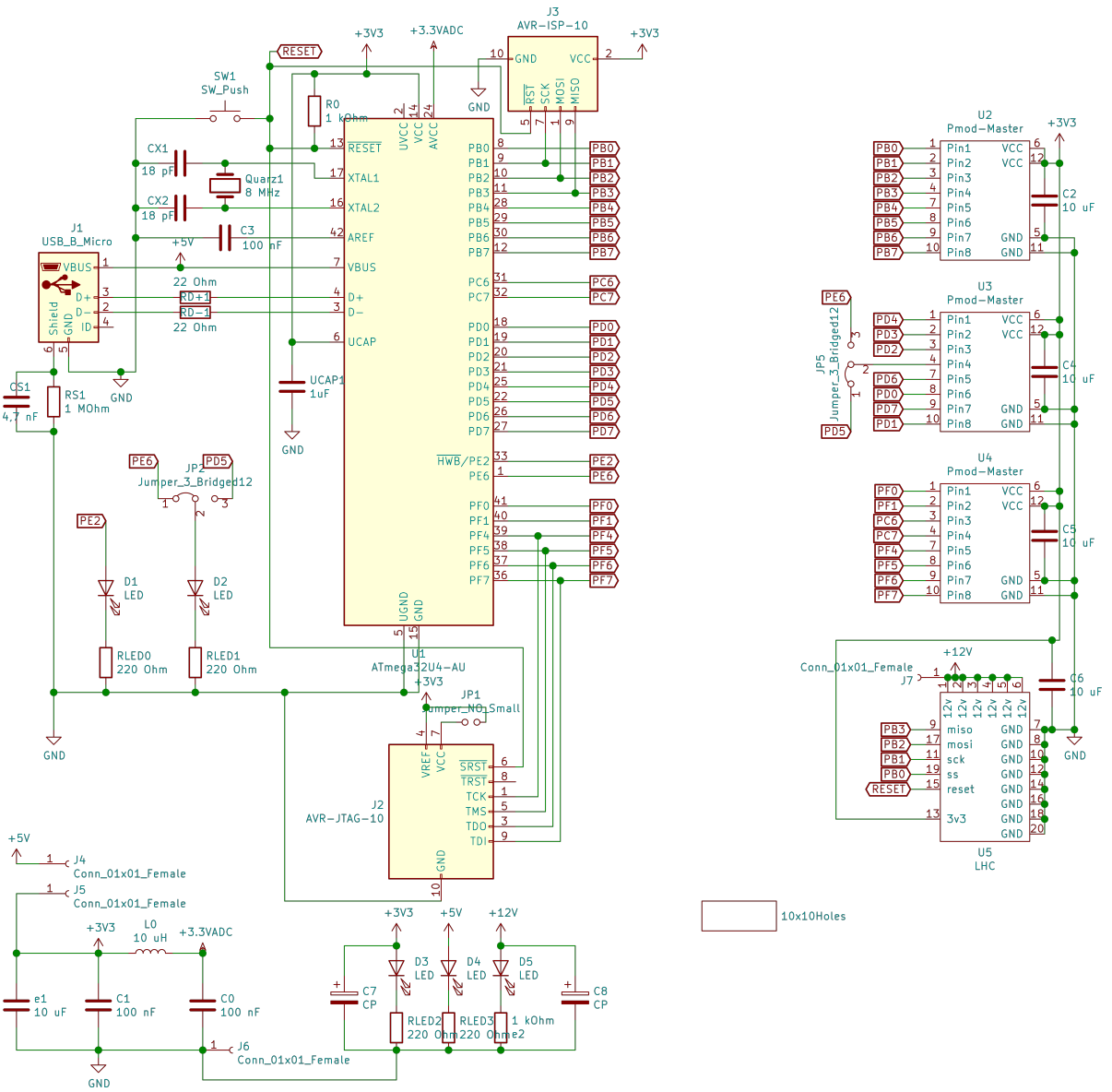
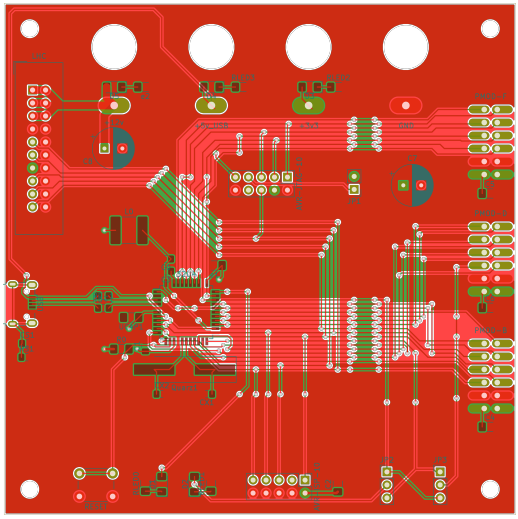


Figure 4.13.: ATmega32u4 schematic

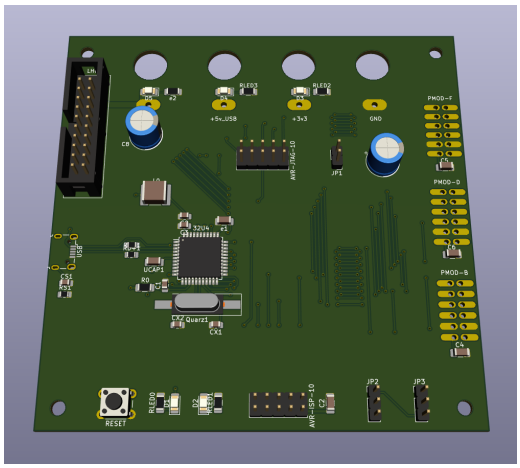
inputs are captured and debounced by the software, and used to influence the positions of the stepper motor. The buttons are used to calibrate the initial positions before the motors run. The disk and the rods would most likely collide, if the initial position is not set. The other feature of the motor controller software is the overcurrent protection. The motor is turned off for a fixed amount of time if an overcurrent is detected.

4.5.1. Motor State SCChart

In Section 3.1.2 the half step control pattern for stepper motors was introduced. The motor SCChart, visualised in Figure 4.15, creates these control signals. The output is



(a) PCB



(b) PCB 3D

Figure 4.14.: Atmega32u4

designed for the the modified H-Bridge control signals displayed in Figure 3.4 on the right side. Both sides of the half-bridge are always oppositely polarised. This allows to use only one polarity signal for the H-bridge instead of one for each half-bridge. The single polarity signal has to be negated for one of the half-bridges. The input interface has two signals move and dir. The dir signal specifies the direction of the motor. If the move signal is true, the motor controller moves a half step in the specified direction. This is done by transitions between the states. The SCChart has a state for each of the eight half steps. The states have entry actions that set the output signals. The name of each state is derived from the states output. The capital letters "A" and "B" indicate the respective coil of the motor is powered. They are followed by a lower case letter indicating the polarity, "p" for plus and "m" for minus.

4.5.2. Overcurrent Protection

Section 3.1.2 introduced the idea of using higher voltage and a current limiter. The basic idea is to use the coil of the motor, the H-bridge and the protection diodes as a Buck converter[13]. To keep the coil always magnetized, the Continuous Conduction Mode (CCM) of the Buck topology is used. A possible implementation of this mode is constant off-time PWM. In this implementation the power is disconnected for a constant time. During this off-time the coil is discharged through the protection diodes. The over current protection SCChart, Figure 4.16, has three states: Wait, Power, Cooldown. Only in the power state the outgoing enable signal is true. As long as the incoming enable signal is false, the SCChart is in the Wait state. If the incoming enable signal is activated, the transition to the Power state is taken. If an over current is detected, while in the

[13] http://www.learnabout-electronics.org/PSU/psu31.php

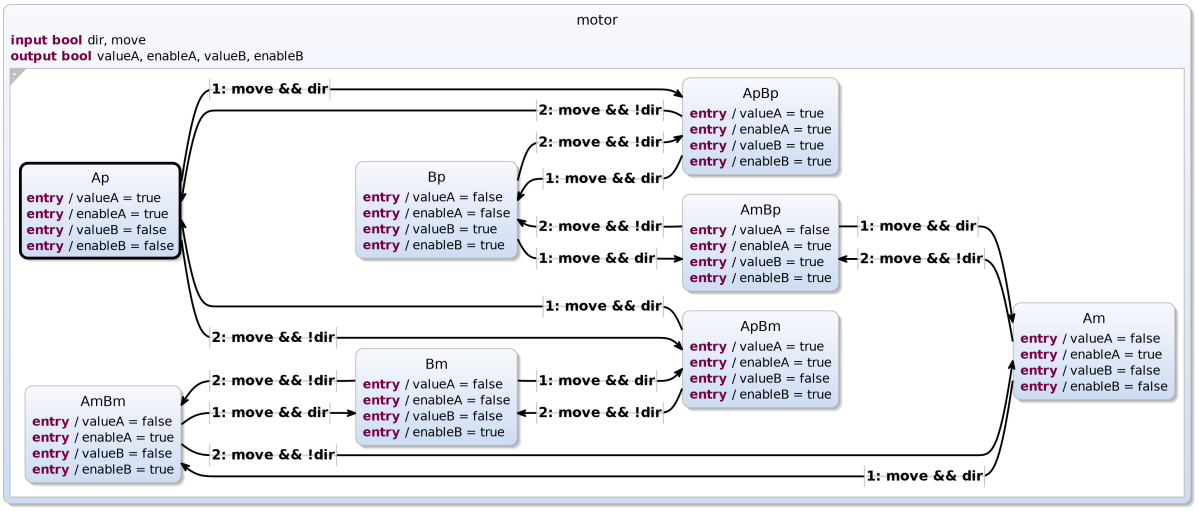


Figure 4.15.: SCChart half step state controller

Power state, the transition to the Cooldown state is taken. The Cooldown state is left after the selected fixed time to either the Wait or Power state, depending of the incoming enable signal. The off-time has to be selected based on the speed of the H-bridge and over current detection, the coil and the acceptable value of current change during an PWM cycle.

The transition between the Power and Cooldown state has another condition. The state machine has to be in the Power state for at least 1500 ns before the transition can be taken. This blind time has its origin in the parasitic induction of the resistor used to measure the current. In the moment of activation of the H-bridge the coil is still charged. The current, flowing through the coil, tries to flow to the ground through the resistor. The parasitic inductance of the resistor is not charged yet and blocks the current flow. This leads to a brief voltage spike until the inductance is overcome. This voltage spike would trigger the transition to the Cooldown state if there were no blind time. This loop would repeat itself until the coil is completely discharged. The length of the blind time depends of the parasitic induction. The blind time has to be much smaller than the off-time, otherwise there would be a risk of a run away situation in which the current decrease during the cooldown is smaller than the increase during the blind time. In these cases the outgoing transition of the Cooldown state would need another guard that stops the leaving until the over current event has passed.

4.5.3. Edge Detection

The edge detection, described in the reduced controller, is described in its own SCChart (Figure 4.17) for better visual clarity in the flowchart of the complete controller.

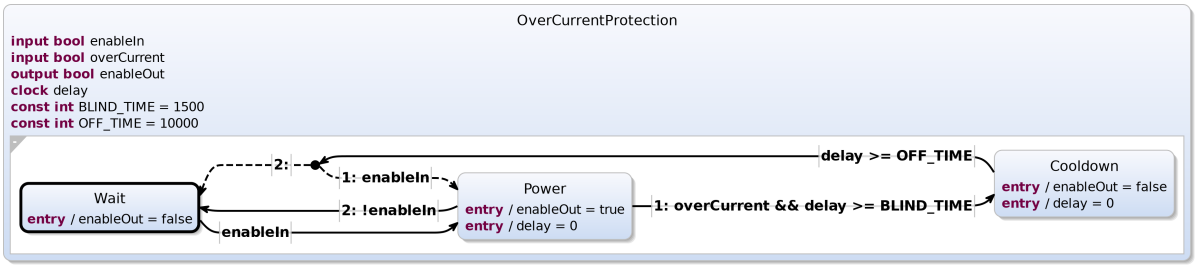


Figure 4.16.: SCChart overcurrent protection

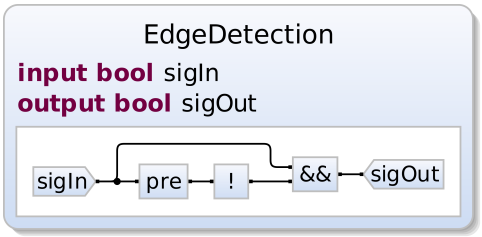


Figure 4.17.: SCChart edge detection

4.5.4. Multi Click

The user interface has four buttons that are used to rotate the motors individually, one button per motor and direction. To facilitate longer rotations it is possible to press the button for more than one second and the button click is repeated at a fixed rate. The multi click SCChart (Figure 4.18) creates the click repetitions and debounces the incoming button signal.

The debouncing is done by the delay of 50 ms on the transition to the FirstStep state. The used clock is reset by the self transition of the Disabled state. This setup ensures that the button is pressed continuously for at least 50 ms. If the button stays pressed for the next 950 ms, the transition to the Off state is taken. As long as the the button stays pressed the state machine changes between the On and the Off state, generating 10 button presses per second.

4.5.5. Speed Signal Divider

The motor with the rods has to run faster than the motor with the disk. The ratio between the rotation speeds is 3 to 5. To accomplish this, the input speed signal is split up into five sub steps for one motor and three sub steps for the other motor. The Speed Signal Divider (Figure 4.19) also takes the button presses as an input and moves substeps accordingly. To reduce the complexity of the SCChart, the manual movement and the movement from the speed signal cannot happen in the same tick.

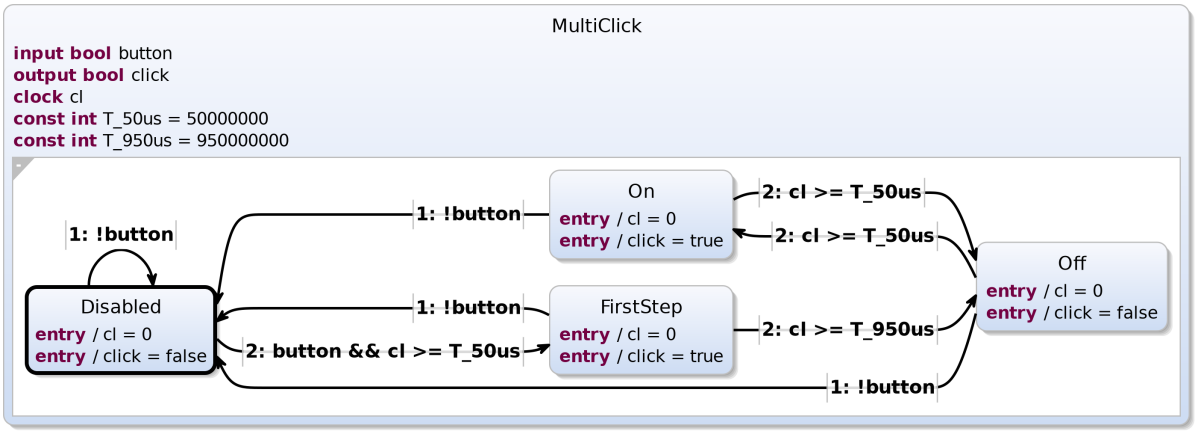


Figure 4.18.: SCChart multiclick

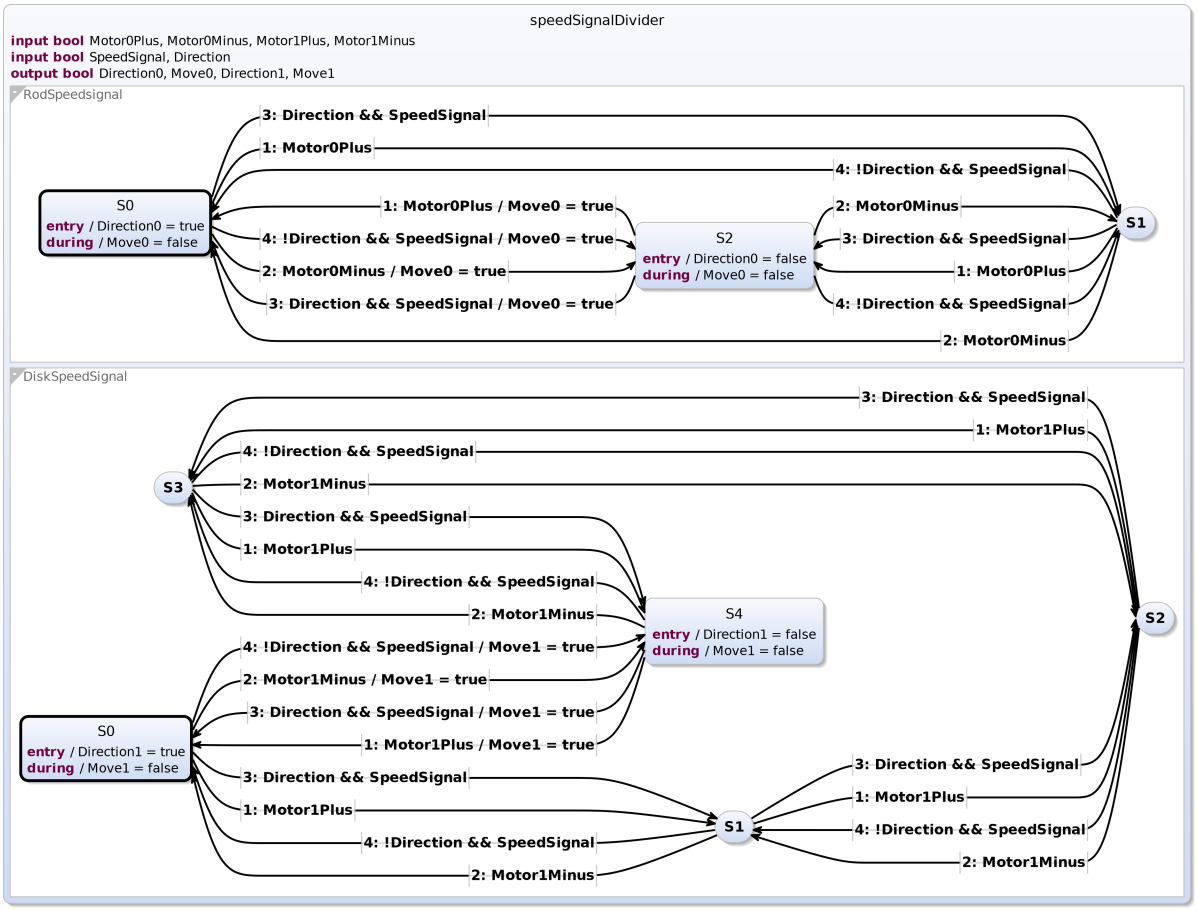


Figure 4.19.: SCChart speed signal divider

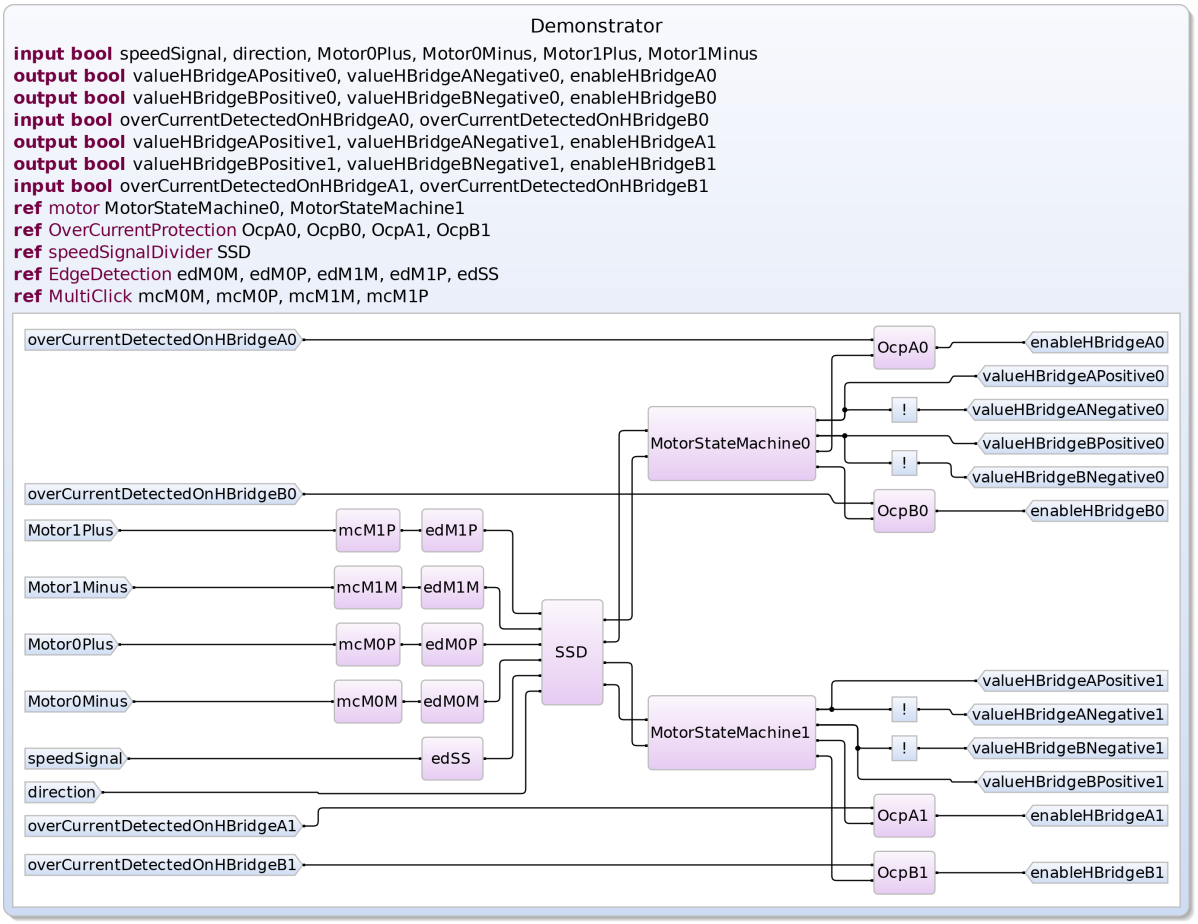


Figure 4.20.: Top-level SCChart controller

4.5.6. Controller

The top-level controller uses and combines the SCCharts presented so far. The usage of dataflow, instead of parallel regions, in this SCChart creates a good visualization of the program structure. This underlines the value SCChart provide by allowing state-based and dataflow-based programming

The controller in Figure 4.20 references two Motor State SCCharts that get their input from a Speed Signal Divider (SSD) SCChart. The inputs of the Speed Signal Divider are synchronized with Edge Detection (ed*). If the input is a button, a Multi Click module (mc*) is added before the edge detection. Putting the edge detection behind the Multi Click module allows the Multi Click module to be level-based. This reduces the size of the SCChart. An overcurrent protection ocp* is added for each of the 4 enable signals, to protect the motors.

4.6. Dynamic Tick Environment

One of the concepts of SCCharts is to keep SCCharts platform independent. SCCharts are run in platform-dependent environments to achieve this. The task of the environment is to capture the input data from the sensors, call the tick function of the SCChart, and provide the results to the actors. This report uses a special form of environment, the DTE. This environment type is designed for Timed SCCharts. The characteristic feature of DTE is that it only calls the tick function if it is necessary.

4.6.1. DTE in VHDL

The VHDL DTE uses 64 bit signed integer Int and Time data type. The used VHDL data type is signed(63 downto 0) and is imported from IEEE.NUMERIC_STD.

Figures 4.21 and 4.22 contain the VHDL code of the DTE. In Figure 4.21 the interface of the environment is declared. The interface differs from the interface expected from Figure 1.4. Figure 4.23 is a detailed visualization of the complete setup, including the connection between hardware, DTE, and the mulit-cycle tick logic. The VHDL DTE is hierarchically on the same level as the SCChart, and is located between the sensors and the SCChart. The DTE takes the sensor data as an input. The DTE forwards this data to the tick function and uses it to detect any change in the input signals input to detect events. The outputs of the SCChart are not used by the DTE, with the exception of the sleepT signal. All other outputs are forwarded to the actors. The timing and tick trigger signals are exchanged between DTE and tick function. The whole setup acts as a two state state machine, that is either in tick calculation state or in environment mode. The input type of the data from the sensor is STD_LOGIC. The output type is Boolean. This is done since in FPGA design STD_LOGIC is the standard type used for Boolean values. The original compiler uses Boolean instead of STD_LOGIC. This was not changed to keep the compiler as close as possible to the original compiler. Then the Behavior of the environment is defined. Strating with the internal signal declarations. The active signal indicates that the environment is running. The DTE has to run before the tick function. Therefore it is set to '1' on reset. This '1' is sent as a token of activity to the tick function via the TF_tick signal, if an event triggers a tick. As soon as the tick calculations are done, the token is received back via the TF_tickDone signal. The deltaTime signal stores the time passed. It is always a clock cycle ahead to compensate for the time passed during this clock cycle, in this case 10 ns. The pre_* signal stores the last inputs of the sensor for edge detection. This is also done during reset. Many SCCharts need some ticks to get ready to run. In these cases it is necessary to add a go counter that is counted down with every tick and ticks are triggered immediately as long this counter is not 0. In this case one initial tick is enough. This initial tick is triggered by the time management since the initial value of SleepT is 0.

Figure 4.22 continues with the main logic of the DTE. It has three variables: deta, trig

```
----------------------------------------------------------------------------------
-- Company: Kiel Univeristy
-- Engineer: Andreas Boysen
--
-- Creation Date: 09/23/2019 01:33:41 PM
-- Design Name: Dynamic Tick Environment
-- Module Name: Environment - Behavioral
-- Project Name: An FPGA based Demonstrator for Dynamic Ticks
-- Target Devices: Arty A7
-- Description: A Dynamic Tick Environment for the basic stepper motor controller setup.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity environment is
    Port ( clk : in STD_LOGIC;      -- clock signal
           reset : in STD_LOGIC;    -- reset signal
           asap : in STD_LOGIC;     -- if true the environment runs in asap mode.
           -- from IO (sensor input)
           IO_overCurrentDetectedOnHBridgeA : in STD_Logic;
           IO_overCurrentDetectedOnHBridgeB : in STD_LOGIC;
           IO_speedSignal : in STD_LOGIC;
           IO_direction : in STD_LOGIC;
           -- to SCChart
           --- State signals
           TF_tick: out std_logic;
           TF_tickDone : in std_logic;
           --- input/output
           TF_overCurrentDetectedOnHBridgeA: out boolean;
           TF_overCurrentDetectedOnHBridgeB: out boolean;
           TF_speedSignal: out boolean;
           TF_direction: out boolean;
           --- DTE Time Signals
           TF_deltaT: out signed(63 downto 0);
           TF_sleepT: in signed(63 downto 0)
           );
end environment;

architecture Behavioral of environment is
-- activ signal is '1' iff the envrionment is running and 0 iff the tick calculation is running
signal  active : STD_LOGIC:= '1';
-- pre registers for edge detection
signal  pre_overCurrentDetectedOnHBridgeA :  STD_Logic;
signal  pre_overCurrentDetectedOnHBridgeB :  STD_LOGIC;
signal  pre_speedSignal :  STD_LOGIC;
signal  pre_direction :  STD_LOGIC;
-- time counuter
signal deltaTime : signed(63 downto 0) := x"0000000000000000";
-- trigger for the Tick calculation.
signal nextTick : STD_LOGIC;

begin
-- output of the trigger signal
TF_tick <= nextTick;
```

Figure 4.21.: Dynamic tick environment in VHDL (environment.vhd) part A

```
process(clk) is
-- variables for trigger system
variable delta : STD_LOGIC;
variable trig  : STD_LOGIC;
variable timingEvent : STD_LOGIC;
begin
    if rising_edge(clk) then
        if reset = '1' then
            -- reset all values
            active <= '1';
            deltaTime <= x"000000000000000a" ;
            pre_overCurrentDetectedOnHBridgeA <= IO_overCurrentDetectedOnHBridgeA;
            pre_overCurrentDetectedOnHBridgeB <= IO_overCurrentDetectedOnHBridgeB;
            pre_speedSignal <= IO_speedSignal;
            pre_direction  <= IO_direction;

        else
            -- trigger calculations
            if (active = '1') then
                -- buffering input for edge detection in next environment cycle
                pre_overCurrentDetectedOnHBridgeA <= IO_overCurrentDetectedOnHBridgeA;
                pre_overCurrentDetectedOnHBridgeB <= IO_overCurrentDetectedOnHBridgeB;
                pre_speedSignal <= IO_speedSignal;
                pre_direction  <= IO_direction;
                -- output buffering
                TF_overCurrentDetectedOnHBridgeA <= '1' = IO_overCurrentDetectedOnHBridgeA;
                TF_overCurrentDetectedOnHBridgeB <= '1' = IO_overCurrentDetectedOnHBridgeB;
                TF_speedSignal <= '1' = IO_speedSignal;
                TF_direction  <= '1' = IO_direction;
                -- detecting edges on all inputs
                delta := (pre_overCurrentDetectedOnHBridgeA xor IO_overCurrentDetectedOnHBridgeA) or
                        (pre_overCurrentDetectedOnHBridgeB xor IO_overCurrentDetectedOnHBridgeB) or
                        (pre_speedSignal xor IO_speedSignal) or
                        (pre_direction xor IO_direction);
                -- detecting timing event
                timingEvent := '1' when TF_sleepT <= deltaTime else '0' ;
                -- triggering next Tick
                trig := delta or asap or timingEvent;
                -- writing trigger variable to signal
                nextTick <= trig;
                -- deactivate environment, if tick triggered
                active <= not trig;
                --  time update
                if trig = '1' then
                    deltaTime <= x"000000000000000a" ;
                    TF_deltaT <= deltaTime;
                else
                    deltaTime <= deltaTime + x"000000000000000a" ;
                end if;
            else
                -- no new Ticks, while Tick calculation
                nextTick <='0';
                -- reactivate, if Tick function done
                active <= TF_tickDone;
                -- updating time, while Tick is calculated
                deltaTime <= deltaTime + x"000000000000000a" ;
            end if;
        end if;
    end if;
end process;

end Behavioral;
```

Figure 4.22.: Dynamic tick environment in VHDL (environment.vhd) part B

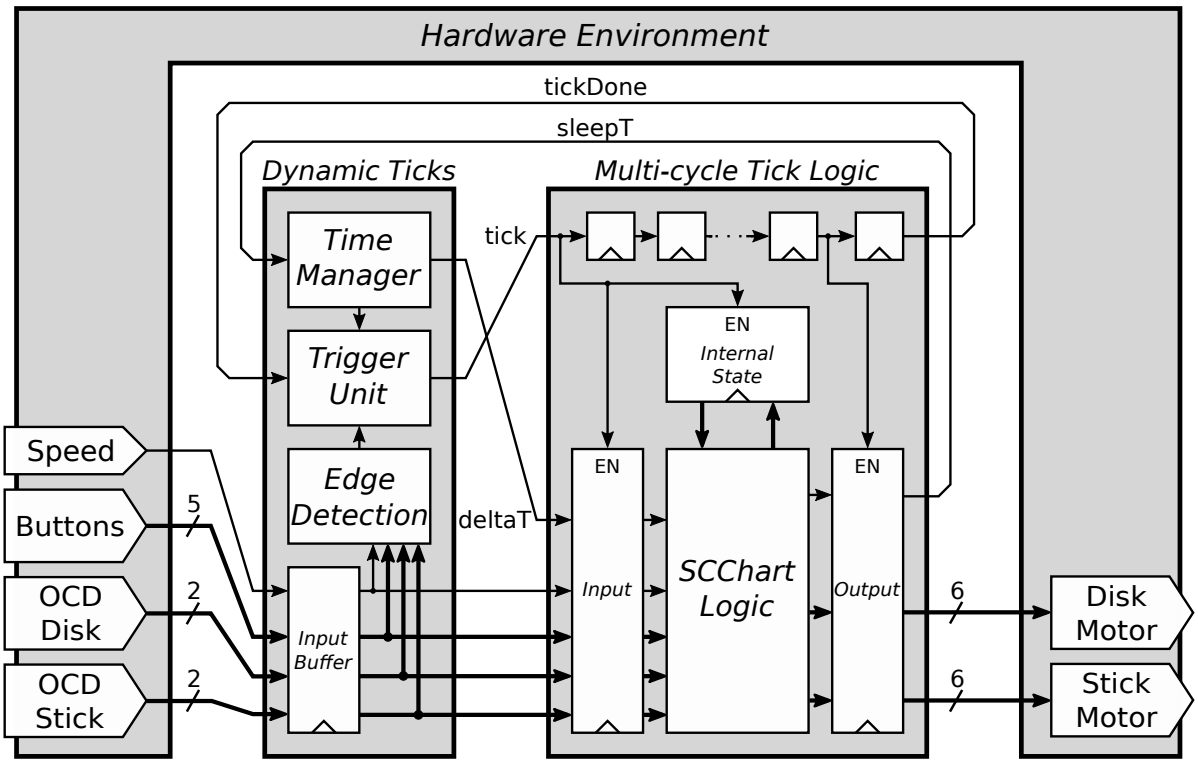


Figure 4.23.: Implementation of a dynamic tick environment

and timingEvent, that are used for the event calculation. To understand this code, it is important to understand the difference between signals and variables. Signals are all written at the same time and all readings are done on the signal value set at the last clock cycle. The consequence is that even if in line 84 the pre_overCurrentDetectedOnHBridgeA signal is changed to the current value of IO_overCurrentDetectedOnHBridgeA, the comparison between these two signals in line 94 uses the value that pre_overCurrentDetectedOnHBridgeA had before the assignment on line 84. Variables on the other hand are calculated sequentially. This is often used for operations over arrays, such as *any*, that check if any bit of a vector is true. Variables allow to declare a simple for loop over the elements of the vector instead of writing all elements explicitly. In this program the variables are used to collect all events, triggering a tick. In line 94ff all inputs were checked for changes and the result is stored in the variable delta. The timing event is stored in timingEvent on line 99. These events are combined in line 101 to the trig variable. The use of variables makes the code much more readable. The variable trig is written to the signal nextTick in line 103. The variable trig is needed by the time management beginning in line 107. Without the use of variables the time management's if-statement would have to contain all variable calculations used to create trig. The time management is a simple counter that is in case of a tick trigger written to the TF_deltaT signal and reset, and otherwise just increased by the cycle length.

The else case beginning in line 113 is executed during the tick calculation. It prevents the start of any new tick calculation by setting nextTick to '0'. The active signal is set

to the TF_tickDone signal value to reactivate the DTE after the tick calculation is done. This activation of the DTE takes a complete cycle, increasing the minimal runtime of the DTE by one cycle. It is possible to activate the DTE immediately instead by changing the condition of the if statement in line 82 from active = '1' or FT_tickDone ='1'. This is not done here to give the environment at least one clock cycle to react to the new output. The last operation is the increasing of the time counter to keep track of time. Increasing the time counter every cycle instead of once by the length of the tick calculation allows to use implementations with changing or unknown calculation times.

4.6.2. DTE in C

The C implementation uses the approach described in Figure 3.10 with a small variation. The tick trigger signal is not fed back to the time management. Instead the time management has two functions. The function trigger_timing returns true if a timing event occurs. update_timing is called directly before the tick function to get the current deltaT.

The basic concept of the C DTE is derived for the code generated by the SCChart compiler in the netlist approach. All data of the modules are stored in a struct and and modified by functions. The code generated from SCCharts has two functions. One function resets the struct and the other calculates the tick. The modules of the environment have the same function, but have another function to create an instance of their struct. The struct of the environment is defined in the header (Figure 4.24), it contains pointers to the structs of all other modules. This allows the main program loop (Figure 4.25) to call all modules without any knowledge of their inner functions. The volatile char pointer s is used to stop the main loop out of other processes.

The init_environment() function calls the init function of all other modules and creates the struct of the environment. The implementation in Figure 4.25 is the Raspberry Pi version of the environment and therefore contains some Raspberry Pi specific calls. The function reset_environment uses the same technique, it also calls the resets of all modules.

The run_environment function calls the step function until the SCChart is terminated or the loop is interrupted by another thread. The step function, step_environment does two things different than the pseudocode in Figure 3.10. First it is adapted according to the mentioned changes in the time management interface and second, instead of a inner loop waiting for an event, the call of the tick is optional. This change allows to stop the execution of the loop between every input read instead of only after a tick was triggered.

The environment is the only module, that is (mostly) platform independent. All other modules have to be implemented platform specific. The following paragraphs contain the ideas for these modules for the platform Raspberry Pi and ATmega.

Raspberry Pi Specific Modules The GPIO pins of the Raspberry Pi are used to communicate with the motor driver. This is done with the library bcm2835.h. It is not possible

```
#ifndef HEADER_ENVIRONMENT
#define HEADER_ENVIRONMENT
#include "input.h"
#include "timing.h"
#include "output.h"
#include "scchart.h"
#include "input.h"
#include "timing.h"

#include "piio.h"
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    InputData*     i;
    TimingData*    t;
    OutputData*    o;
    TickData*      d;
    volatile char* s;
} EnvironmentData;

EnvironmentData* init_environment();
void reset_environment(EnvironmentData* e);
void run_environment(EnvironmentData* e);
void step_environment(EnvironmentData* e);

#endif
```

Figure 4.24.: Dynamic tick environment in C, header file (environment.h)

to change all outputs at the same time to a new value. The Raspberry Pi GPIOs use two functions: set and clear. The set function sets all values, defined by an mask to '1'. The clear function does the same except it sets the value to '0'.

The input module also uses the bcm2835.h to read the input from the GPIO pins. The struct of the input module is used to store the input values of the last call to be able to detect edges.

The timing module uses the timespec_get system call to get the current time to calculate deltaT. The timespec data is then converted into a 64 bit integer, to be used by the SCChart. timespec_get on a Raspberry Pi is not a precise time source. Using the integrated cycle counter of the Raspberry Pi would be much more precise. This option was not used here, since timespec_get is the common way for time access. Furthermore, accessing the cycle counter requires root permissions which might be undesirable for such a program.

ATmega specific modules The GPIO pins of the ATmega can be accessed directly by reading out of registers (variables). No additional library is needed. The more complex part of a DTE on an ATmega is getting the time. The biggest data type, supported by the AVR-C compiler, is 64-bit Int. int64_t is used as the time type. The time is tracked by a 16 bit counter. The counter overflow event is activated and used to increase the 64 bit integer. This process is not as simple as it sounds. The problem is that the counter and the thread have to be synchronised. After an interrupt occurs it takes at least seven cycles before the 64 bit integer is updated. This problem is solved by adding a new

```c
#include "environment.h"

EnvironmentData* init_environment(){
    // malloc
    printf("init\n");
     EnvironmentData *e;
    volatile char * c;
    e = (EnvironmentData *)malloc(sizeof(EnvironmentData));
    c = (char *) malloc(sizeof(char));
    if (e == 0 || c== 0 )
        {
            printf("ERROR: Out of memory\n");
            return 0;
        }
    // Raspberry Pi specific
    printf("sub_PIIO\n");
    initPiIo();
    // sub_init
    printf("sub_input\n");
    e->i = init_input();
    printf("sub_timing\n");
    e->t = init_timing();
    printf("sub_malloc TickData\n");
    e->d = (TickData *) malloc (sizeof(TickData));
    printf("sub_reset TickData\n");
    reset(e->d);
    printf("sub_outpit\n");
    e->o = init_output();
    printf("sub_char\n");
    e->s = c;
    printf("inint done\n");
    return e;
}

void reset_environment(EnvironmentData* e){
    reset_input(e->d, e->i);
    reset_timing(e->d, e->t);
    reset(e->d);
    reset_output(e->d,e->o);

}

void run_environment(EnvironmentData* e){
    while(!(e->d->_TERM)) step_environment(e);
}

void step_environment(EnvironmentData* e){
    int u, v;
    u=update_input(e->d, e->i);      // reading input
    v=trigger_timing(e->d, e->t);    // checking dynamic tick trigger
    if (u||v) {                      // if trigger
        update_timing(e->d, e->t);   // update time
        tick(e->d);                  // SCChart Tick
        update_output(e->d, e->o);   // output results.
    }
}
```

Figure 4.25.: Dynamic tick environment in C, source file (environment.c)

Boolean that is set to true if the counter is at least at 2^{15} by a comparison interrupt. The overflow interrupt sets it to false again. If the value read from the time is small but the boolean indicates, that is should be big, an overflow event occurred, but was not handled yet. Therefore the combined time value has to be increased by 2^{16} to account for this.

4.7. Tool Chains

To run and deploy the software on the demonstrator hardware, some tool chains are needed. This section provides a short overview of the most important tool chains used in this demonstrator.

SCChart to FPGA

The first step is to convert SCChart code to VHDL. The SCChart to VHDL compiler is implemented as described in the Section 3.2.3, integrated in KIELER and used for this compilation step. The generated code uses MCPs. A constraint generator is implemented and used to generate the necessary MCP constrains from the VHDL code. The VHDL code and the constraint files are loaded into the Vivado design suite and combined with the DTE and other environment code. From here on the Vivado tool chain is used, including synthesis, implementation, bitstream generation and programming.

SCChart to Raspberry Pi

KIELER is used to convert SCCharts to C code. This code, transferred to the Raspberry Pi, is combined with the environment code and compiled with the GCC.

SCChart to Atmega

Again KIELER is used to generate the C code from the model. Then the code is then combined with the environment code and compiled with avr-gcc. avr-objcopy converts the object code to hexfile format, needed for avrdude. avrdude is used to flash the hexfile onto the Atmega.

4.8. Analysis-Specific Components

The previous sections contain everything necessary to run the demonstrator. This Section contains the components that are only needed for the analysis. The complete demonstrator is a timing critical system by design, to show the achieved precision. To be able to test the limits of the system, without destroying it, a reduced setup with one motor is used. The reduced controller replaces the controller in the analysis. To capture data, a logic sniffer is used. The PMOD Sniffer is designed, to keep the logic sniffer galvanically isolated from the rest of the system. The advantage of this approach is that

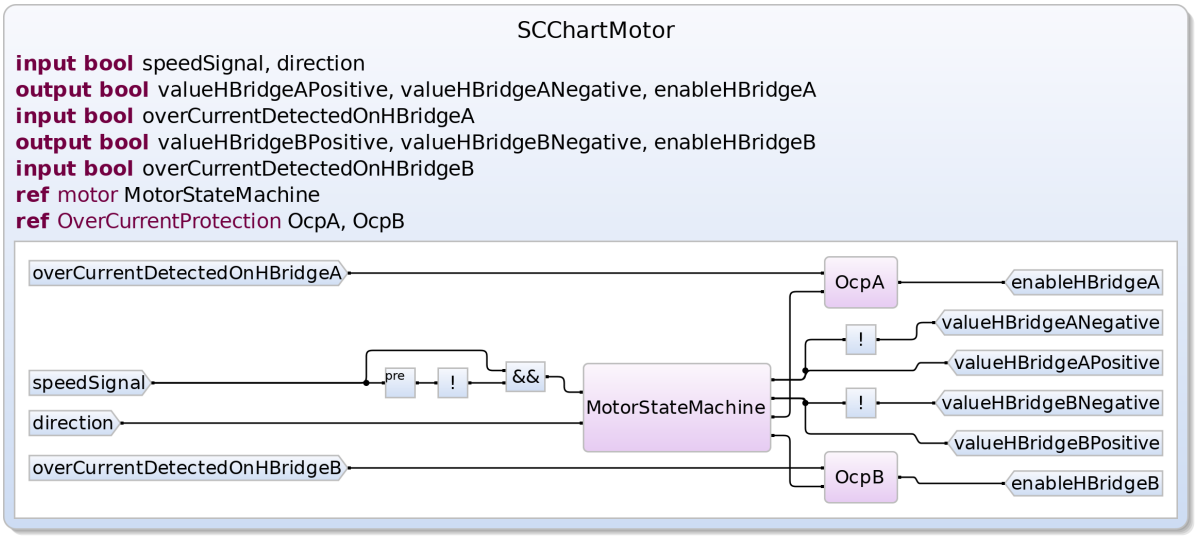


Figure 4.26.: SCChart for the reduced controller

the logic sniffer and the control signals that are observed can operate on different logic levels.

4.8.1. Reduced Controller

The motor state controller and two over current protections can be combined to a simple motor controller, see Figure 4.26. The motor controller takes a speed signal and a direction signal as input and outputs to control the motor. All other input and outputs are connected to the interface of the l298 stepper motor driver. The incoming square wave is not synchronized to the tick rate. It is necessary to synchronize the signal to the tick clock. This is done by the inlined the edge detection.

4.8.2. Logic Sniffer

A logic sniffer with compression was implemented in VHDL to log the input and output of the controller. The compressed data is sent per Ethernet to a computer, where it is decompressed and converted into the Value Change Dump format. A galvanic isolation is necessary to protect the logic sniffer. The PMOD Sniffer is created for this purpose.

4.8.3. PCB: PMOD Sniffer

The PMOD Sniffer board is designed to be inserted between a PMOD master and a PMOD slave. The board allows the monitoring of the traffic with minimal distortion of the data signals between PMOD slave and master. The application of this board is to copy signals at their origin for the external logic analyzer. This minimizes the effect of the logic analyzer to the signals and protects the components from each other.

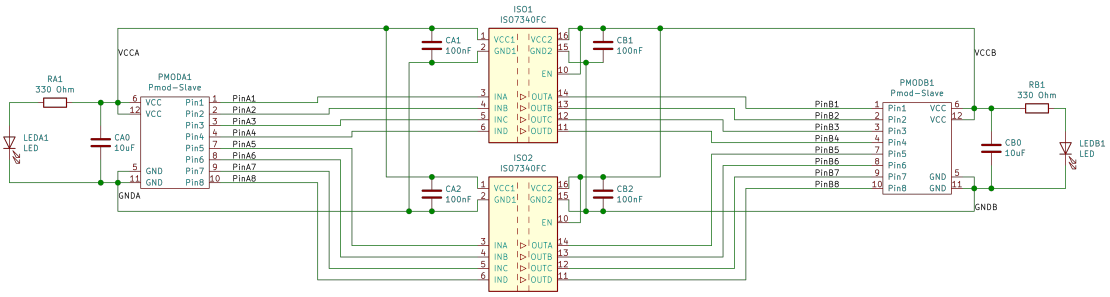
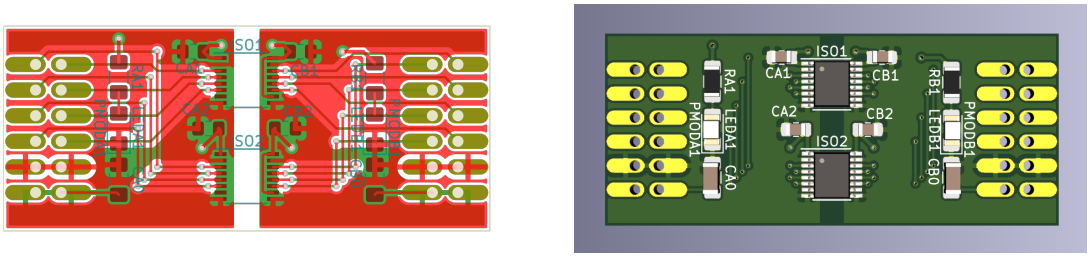


Figure 4.27.: PMOD sniffer schematic



(a) CAD model

(b) PMOD sniffer PCB 3D

Figure 4.28.: PMOD sniffer PCB

Interface

PMOD-A is equipped with a stackable pin header. This pin headers are male on one side of the PCB and female on the other side. PMOD-B is a PMOD slave. The output on PMOD-B is a galvanically isolated copy of the data, revived on PMOD-A.

Circuit Diagram

The ISO774x family[14] of digital isolators fulfills all requirements. They have a wide input range from 2.25 to 5.5 volts. The schematic in Figure 4.27 shows that two ISO7340 are used. The ISO7740 and ISO7340 have the same pinout and there is no library for the ISO7740 available yet. Therefore the library of the ISO7340 is used in the schematic. The rest of the design is standard. The power of the ISOs is stabilised with 100 nF caps (CA1, CA2, CB1, CB2). The power received from the PMOD header is also stabilized with 10uF caps (CA0, CB0). Each power rail has a led (LEDA1, LEDB1) to indicate its presence. Both PMOD header are slave PMOD headers. This is done to have both slave connectors on the same side. The same would be true, if two PMOD masters had been used.

PCB Layout

A smaller form factor of the ISO7740 was used to keep the board compliant with the PMOD standard. This allows to use multiple sniffers next to each other on a PMOD devices. The design is mirrored. Each side has a power-indicating LED and the 10 uF near the PMOD header. The 100 nF capacities are placed close to the ISOs. The board uses ground-plain on top and VCC planes on bottom. Figure 4.28a is the CAD model of the PCB and Figure 4.28b is the 3D view.

[14] https://www.ti.com/lit/ds/symlink/iso7741.pdf

5. Evaluation

The purpose of the analysis is to characterize the performance of dynamic ticks, different compilation approaches and different controller hardware.

The complete demonstrator is a time critical system due to the collision that occurs in case of a missed step. To be able to test the limits without any risks, the reduced setup is used for tests. In the reduced setup the speed signal is generated by a signal generator. This signal is fed in the motor controller. The motor controller generates the necessary control signals for the motor driver. The motor driver powers the motor accordingly and gives feedback in form of the over current detected signals. The motor is mounted in the motor base, to have the same resonance frequencies in all test cases. The disk assembly is attached to the motor during tests.

The signal between motor controller and power stage and the signal from the signal generator are captured for analysis. For this analysis the environment is programmed to output an additional signal that indicates whether the controller is calculating the tick or the environment is active itself.

The basic tests are conducted by setting up the controller and enabling the power supply. The speed signal is then increased to the target value. The system then has time to stabilize. Only after a stabilization period of at least ten seconds the logic sniffer is started. This approach is chosen to remove effects of the startup. Otherwise it would not possible to create a fair comparison of the different motor controllers.

5.1. Variables of the Test Setup

The test setup can be changed by replacing the motor controller and by changing the external inputs. The parameters of the motor controller that can be changed are the underling hardware, the type of environment and the compiler strategy. The three hardware options are FPGA, Raspberry Pi, and ATmega. The environment is either ASAP or dynamic and the SCChart compiler strategy is either netlist-based, state-based or priority-based. These three variables can be set in any combination with the exception that netlist-based compiler strategy is the only strategy available for FPGAs.

The two external inputs influencing the tests are the speed signal and the power supply. The speed signal is only limited by the hardware speed. The motor voltage is limited by the maximum voltage of the motor driver (46 V).

The final variable of the test is the value selected for the overcurrent detection.

There are many factors influencing the performance of the stepper motor. It is important to understand why they can be kept at a fixed value during all tests and how the results are influenced by this decision. The motor mount influences the resonance

frequency of the motor. The motor mount dampens the vibration enough that a high RPM can be achieved without interference.

All tests are performed with the disk assembly attached to the stepper motor. The load of the stepper motor has to be changed in an evaluation of the quality of the stepper motor controller code. It is, however, not necessary for the evaluation of the environment.

The motor controller code is fixed for all tests. The goal is to evaluate the performance of dynamic tick environments. Furthermore, this setup can be used to compare code optimisation in SCChart, but this is not part of this report.

The capture time of this test is set to around 30 seconds. This time is enough to capture more than 10000 data points. This value is a compromise between the time and resources invested in the test and the type of outliers that can be detected.

5.1.1. Data Capturing and Limitation

The data is captured with the Arty A7 board. This approach has the disadvantage that the logic analyzer and the controller share the same 100 Mhz clock. This leads to an advantage for the FPGA-based motor controllers of up to 10 ns in reaction time. The 10 ns are added to the analysis results to display worst case behavior of the used implementation. The alternative is to use a generated random jitter of 10 ns. This approach was rejected since the 100 Mhz clock, this source of this jitter, is only needed by the logic sniffer. The DTE could run at much higher frequencies and thereby reducing the jitter even more.

A further advantage of the FPGA is that no PMOD sniffer is needed, this reduces the jitter by up to 2.5 ns. The 10 ns interoperation delay of the PMOD sniffer has no effect on the analysis since the delay is on all signals that are captured. Removing the 2.5 ns jitter from the results has a negligible effect on the results and is therefore ignored.

Inputs such as the supply voltage, current limit reference voltage and the speed signal can drift over time. It is important to notice that these input changes can limit the precision of the analysis.

Time Measurement

Figure 5.1 displays all important time points of a tick in a synchronous program. An event occurs at the time T_e. The synchronous system starts its calculation with a delay (Δi) at the time T_{cb} due to handling of wake-up procedures and the time to read inputs. The calculation takes (Δt) time and ends at T_{ce}. The last step is the output of the data that is finished at T_{oe}. There are two time segments related to the output: Δo, the time it takes to output the data after the calculation and Δoi, the time the output of the system is invalid because the writing of outputs has not yet completed. Δoi is important for system development. Other components must be aware of the time frames in with the received data is invalid or this time frame must be so short that it has no effect. Δo

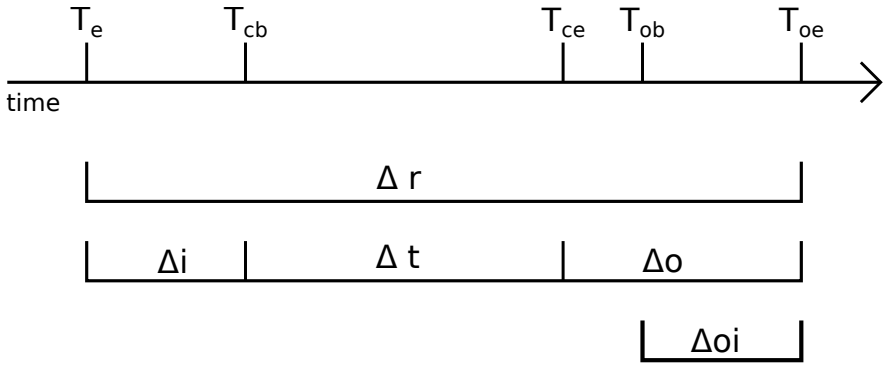


Figure 5.1.: Different time measurement points and resulting intervals.

is only used in performance analysis. Δi and Δo are the time frames spend in the DTE while Δt is the time frame used by the tick calculation function.

T_e, T_{ob} and T_{oe} are the only times that can be measured without changes to the program. To capture T_{cb} and T_{ce} the system has to provide the information. Outputting this information negatively has an impact on the measured time. Therefore Δr, the reaction time, is used in the analysis. The future work section 6.1.1 contains proposals how to capture all time points without influencing the runtime.

The order of the points in time, displayed in figure 5.1, is valid for the used systems but is not true for all implementations. Depending of the implementation, T_{ob} can be as early as T_{cb} and T_{oe} can be before T_{ce}.

5.2. Basic Performance Analysis

The first test run is designed to evaluate the general performance of all controller variants. Furthermore, this test is used to evaluate which of the controllers are capable of running the over current protection. For every platform the best performing combination of environment type and compiler strategy is determined. The selected combinations are then used for further tests.

5.2.1. Analysis Technique

There are fourteen possible combinations of hardware, environment and compilation strategy to create a motor controller. The basic test setup is used to capture data for all of these motor controllers. These data sets are analyzed for the delay between the input of the function generator and the resulting output changes. In Figure 5.1 this is Δr.

Another performance metric is the number of ticks per second. This metric can show the calculation performance of different platforms, if the ASAP environment is used. Furthermore, a comparison between the ticks per second of dynamic and ASAP

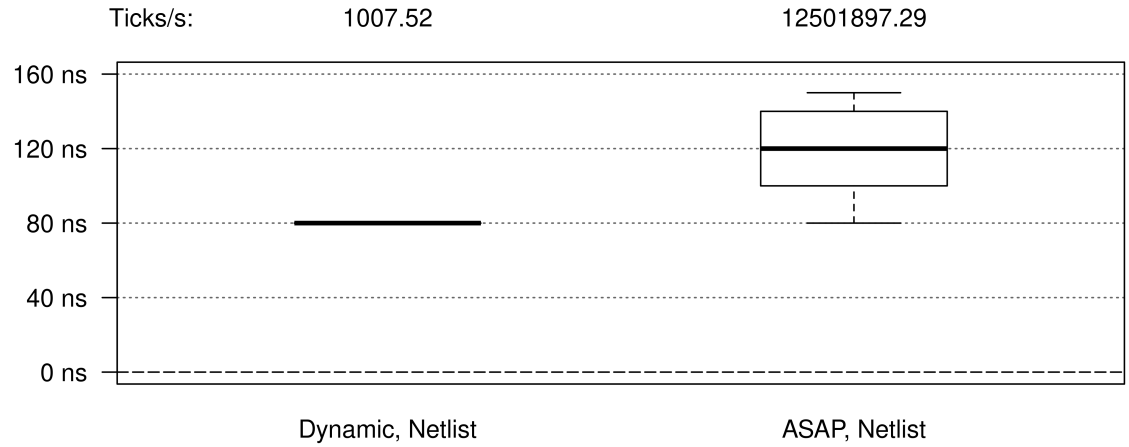


Figure 5.2.: Reaction time for different environment types on an FPGA

environment can prove or refute the hypothesized power saving potential of dynamic ticks.

5.2.2. Test Parameters

The test is run at 400 steps per second and at 5 V. 400 steps, accordingly 60 RPM, is slow enough for any valid controller and fast enough that 5 V will not lead to an over current event. The acceptable variance of these values are +/− 3 steps per second and +/− 0.5 V supply voltage. The supply voltage restriction could be chosen even bigger since there is no feedback to the controller.

5.2.3. Test Results and Interpretation

The ticks per second value of the FPGA controller with ASAP environment exceeds 12 million. The logic sniffer on the other hand can only handle around 2-3 million events per second. This makes it impossible to capture the events over the 30 second capture period. To handle this effect, we capture the ticks per second separately for the reaction time.

The performance of the two FPGA-based controllers is visualized in Figure 5.2. The dynamic tick environment has a constant reaction time. This is possible since there are no two events close enough together to influence each other. The ASAP environment on the other hand has a variance from 1 to 2 calculation times. The reason for this is that the events can occur at any time during the current tick calculation, adding the remaining calculation time to the reaction time. The advantage of an FPGA-based

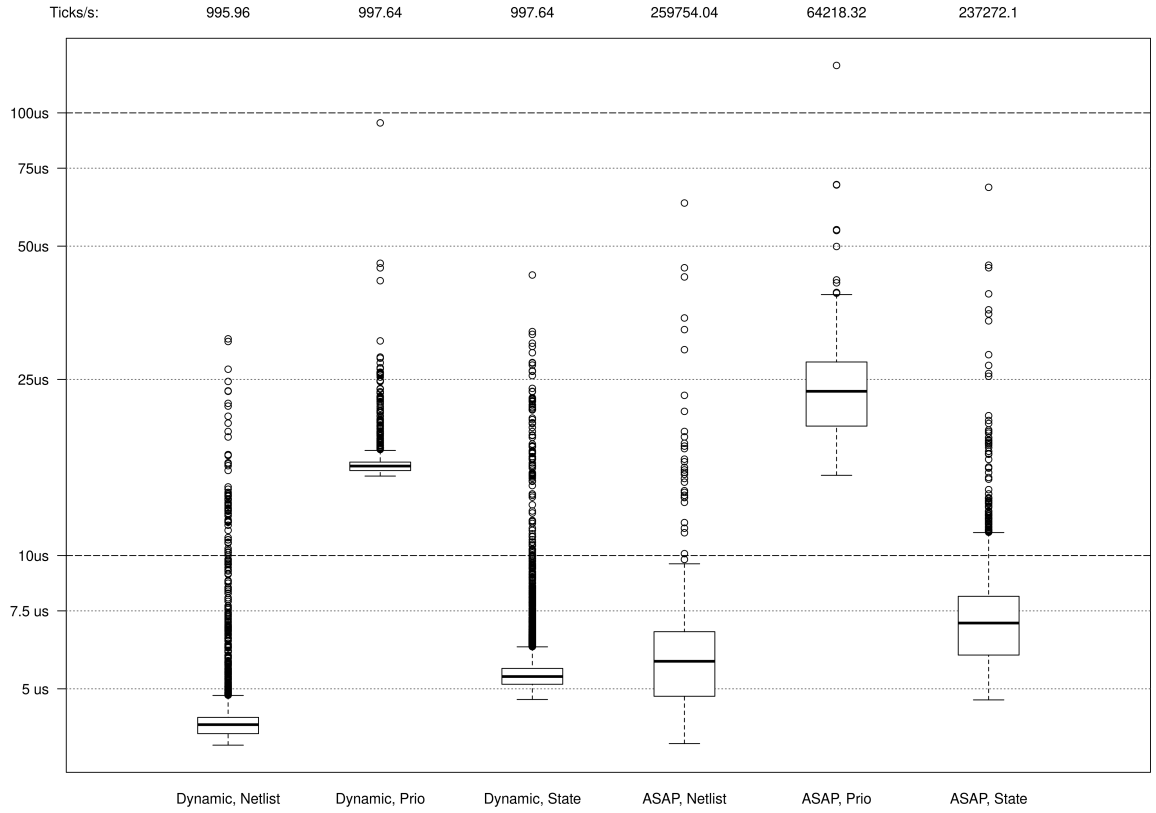


Figure 5.3.: Comparison between different compilation approaches and environment types on a Raspberry Pi (logarithmic scale)

controller is the fast and fixed runtime of the tick function. The calculation time was known during compilation and now confirmed by this test.

Figure 5.3 contains the boxplots and ticks per second regarding the performance of Raspberry Pi based controller. This figure uses a logarithmic scale. The expected number of dynamic ticks for this setup is around 1000. This number is directly dependent on the input speed signal. 800 events for the input signal: 400 rising edges and 400 falling edges. Additionally, 200 events from the blind time (transition between Power and Cooldown state). Only every other step creates this timing event since a transition between two states either enables or disables exactly one coil. All dynamic controllers have a value near 1000, proving that they are working correctly.

The most obvious difference to the FPGA-based controller are the outliers in the reaction time. These outliers have a calculation time that is up to 10 times bigger than their average. There a 2 possible sources of this reaction time jitter: the kernel or tick calculation. Splitting the reaction time up into the time spend in the environment and the time of the tick calculation shows that the runtime jitter occurs in both phases.

Therefore we can conclude that these outliers are created by the kernel.

A standard Raspberian is not a real time capable system. Kernel calls can take an undefined amount of time. This interruption by the kernel can take more than 350 μs [1]. Using a real-time[2] patched kernel could reduce the interrupt length to around 75 μs. None of the tests had a long interruption. The Raspberry Pi had no other loads and no user input was done during the test, to reduce the kernel activity to a minimum. A real-time kernel and isolation of the controller process on a single core are measures that would reduce the number of outliers, but can not remove them. On the other hand would any other load on the system increase the number of outliers. Therefore these results are a fair representation for the achievable performance on a Raspberry Pi.

The comparison between the ASAP tick environment and the dynamic tick environments on the Raspberry Pi present results that are in line with the results of the FPGA. The data confirm the expectation that the ASAP tick environments reaction time is up to one tick calculation time slower than the reaction time of the dynamic tick environment.

The comparison between the compilation approaches shows that the netlist-based approach has the best performance.

The test on the Atmega shows that none of the approaches are able to perform the necessary 1000 ticks per second. Therefore the Atmega based controllers are excluded from all tests. The most likely reason for the low performance is the complexity of the generated code in comparison to the processor speed. The ATmega core frequency is 150 time slower than the Raspberry Pis. Furthermore, the core of the ATmega has only 8-bits and is single core, creating increased cost for time calculation and time tracking.

The final analysis, generated from this dataset, compares the results of the FPGA with the best compiler strategy on the Raspberry Pi. Figure 5.4 visualize the differences in the performance of the FPGA and the Raspberry Pi, also regarding the ASAP and DTE environment.

Consequences for further Tests

There are two conditions a controller has to meet to be able to run the over current protection: the maximum average reaction time and the worst-case reaction time have to be in an certain limit.

The over current protection is described in Section 4.5.2. The important aspect is that the delay between the over current detected and the disabling of the H-bridge has to be shorter than the power-off timing, to prevent a run away situation. This sets the maximum average reaction time. Two reactions have to be faster than the 10 μs off-time. Therefore the average reaction time has to be 5 μs or lower.

To define the maximum worst case reaction time, the current increase per second has to be calculated. Figure 5.5 displays the motor running under test conditions. The first two signals are the voltages over the two shunt resistors on the motor driver. The third

[1] https://emlid.com/raspberry-pi-real-time-kernel/

[2] https://www.linuxfoundation.org/blog/2013/03/intro-to-real-time-linux-for-embedded-developers/

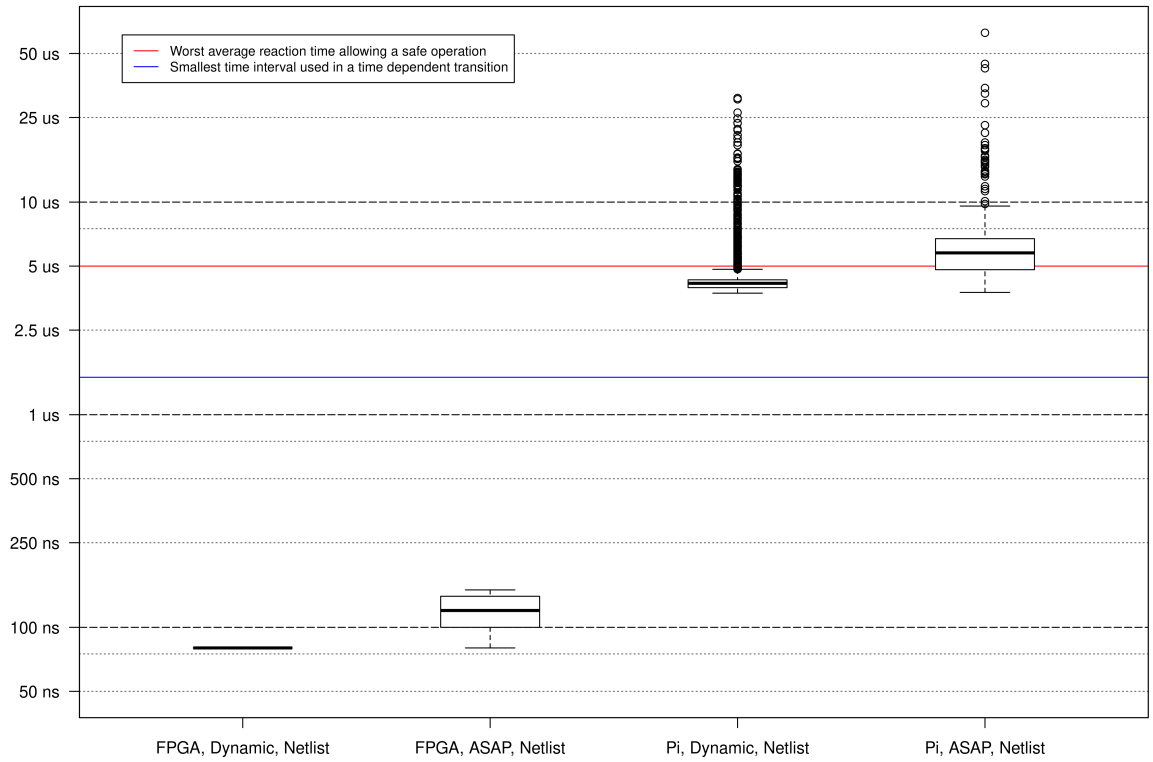


Figure 5.4.: Comparison between FPGA and Raspberry Pi (logarithmic scale)

signal is the supply voltage and the last signal is a trigger signal. The voltage over the shunt resistors rises from 0 V to 500 mV in 7.5 ms. At a resistance of 500 mΩ this equals to 1 A in 7.5 ms or 133,3 ampere per second. This value is linearly dependent on the supply voltage. The goal is to run the motor at 30 V, therefore the expected current increase per second is around 800 A. This approximation is possible because the static resistance is much smaller than the dynamic resistance. Setting the acceptable current variation to 100 mA, 10 preset of the target current, leads to a maximum worst case delay of 125 μs.

Both FPGA-based controllers fulfill these conditions. In Figure 5.4 the red line indicates the 5 μs limit. The netlist-based controller in a dynamic tick environment is the only controller on the Raspberry Pi meets the criteria. Therefore the dynamic and netlist-based controllers are used in the further tests.

5.3. Max stable RPM

The maximal achievable speed of the stepper motor depends on the hardware and the precision of the control signals. A consequence for the interpretation of analysis results

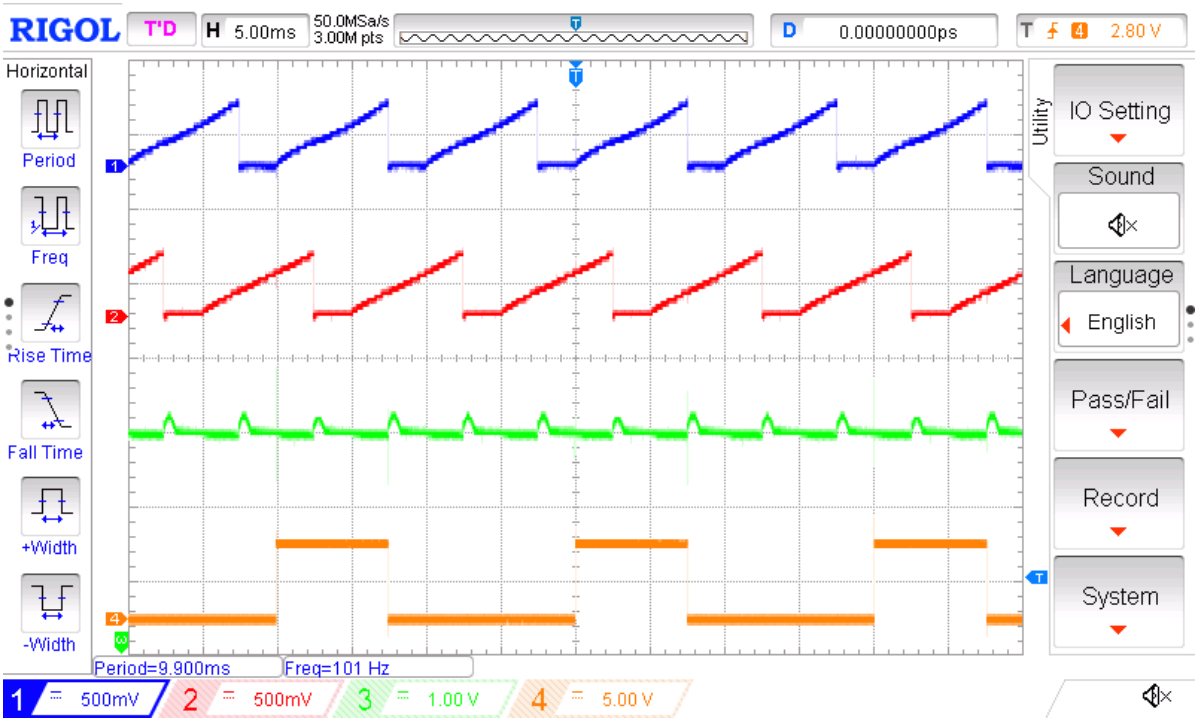


Figure 5.5.: Oscilloscope image captured during basic test

is that results with the same max speeds have no significance. Different results on the other hand indicate difference in the quality of the control signals.

In this test the best performing controllers are compared by their maximal step rate. The test is performed at 30 V and 1 A current limit. The maximum step value is found by manually increasing the speed until the motor stops. The test is repeated multiple times to minimize the influence from the manual test execution.

The maximum speed of the FPGA-based controller reaches 32000 steps per second (4800 RPM). The Raspberry Pi based controller achieves a speed of 15000 steps per second (2250 RPM). The lower value for the Raspberry Pi is expected due to the outlier in the reaction time. These interruptions are not only limiting the max RPM, but are already audible at much lower speeds.

The expressiveness of this test is limited due to a feedback loop between motor and power supply. Figure 5.6 displays this feedback loop. The top two signals are the voltage drop over the shunt resistors while the third is the motor supply voltage. The last signal is the trigger source and can be ignored. The supply voltage varies up to 5 V depending on the motor speed. A theory is that energy oscillates between kinetic energy in the rotor and electric energy in the capacitors. Energy is pumped in the rotor, increasing its speed. This changes the rotor position from following the rotating magnetic field to leading it. The consequence is that the motor is becoming a generator, converting back kinetic energy to electrical. This increases the supply voltage. As soon as the rotor is slow enough, the cycle begins again. The generated energy is not visible on the image

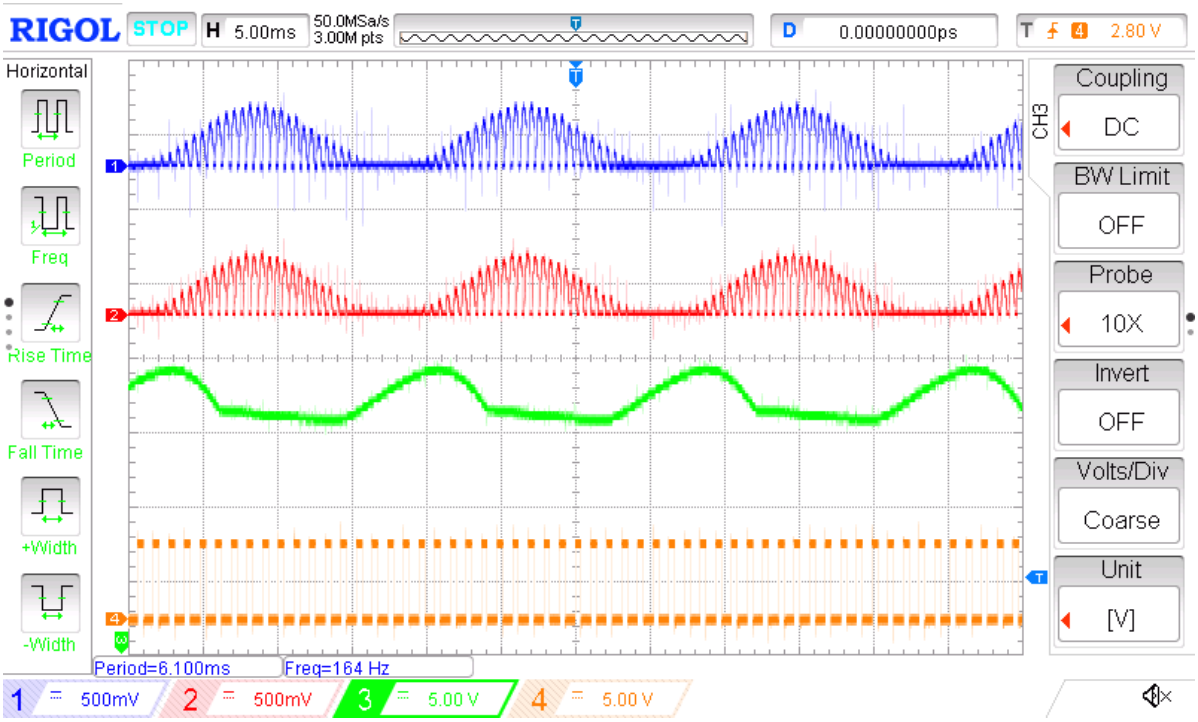


Figure 5.6.: Oscilloscope image of the oscillations

since the generated current flows through the protection diodes and not the H-bridge.

A match between the resonance reference of the motor assembly and the used capacity or the buck converter in the power supply are possible origins of this osculation. A possible solution to this problem is to decouple the power source from the motor with a diode and addition of enough capacitors to store not only the energy from the coil but also from the motor rotation.

Repeating the test with these modifications was not deemed necessary, due to significant difference in the maximal rotation speed.

5.4. Timing Event Jitter

The basic test has no delayed reaction and therefore no measurable timing events. This test uses the fixed off-time in the over current protection to measure the precision of timed events.

This test is performed by setting the motor speed to 0 in a motor state that powers both coils. The supply voltage is 10 V and the current limit is 0.5 A. The used controllers are the best performing controllers determined in the basic performance test.

In this setup both over current protections are active. This way the input event “over current detected” of one coil can occur promptly before or after the timed events created by the over current detection of the other coil. The length and variance of the off-times are used to measure the reaction time jitter.

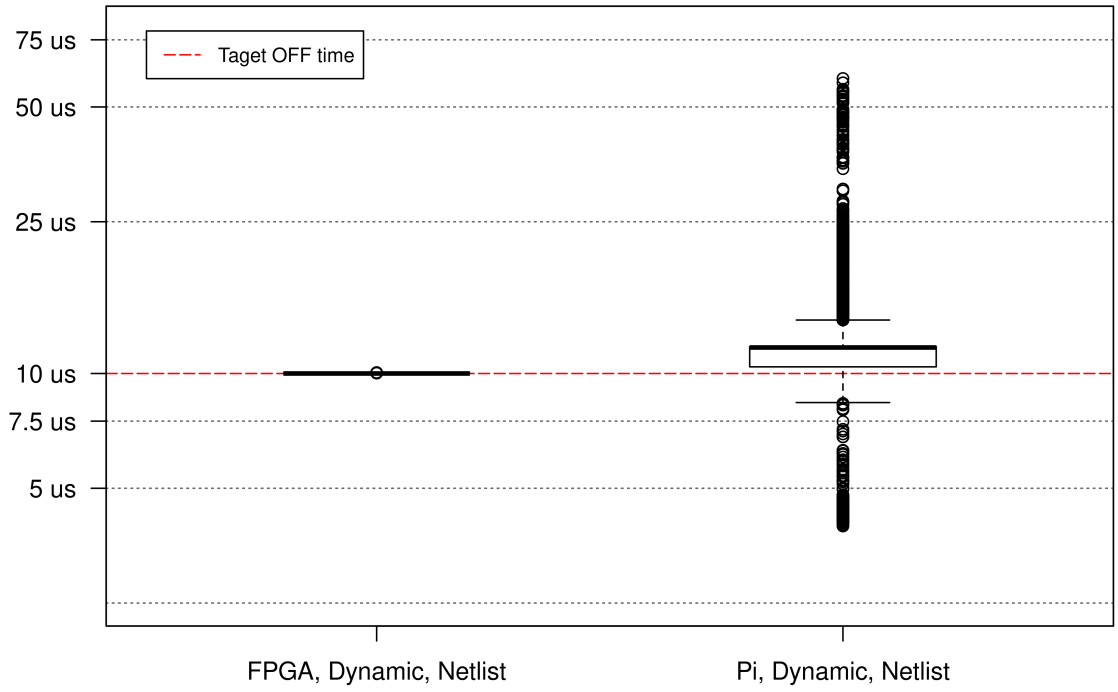


Figure 5.7.: Comparison of time spent in the Cooldown state (logarithmic scale)

The capture time is kept at 30 seconds, resulting in over a million data points. The test results are visualized in Figure 5.7, with boxplots on a logarithmic scale. The FPGA timing is perfect with the exception of a few outliers. The maximal outlier is less than one tick calculation time bigger than the expected value. These outliers are created by an over current event that is less than one tick calculation prior to the timing event. The results of the Raspberry Pi display two types of outliers: off-times that are too long and off-times that are too short. Outliers that are much later were already expected, due to the results of the first test. Timed reactions that are too early are indirectly created by the outliers in the runtime of the tick function. The real off-time is between the output phases of the two ticks. The timing, on the other hand, is taken during the input capturing phase of the environment. Therefore the off-time is extended, if the 2nd tick calculation takes longer than the first and shortened, if it is the other way round. In other words, if a long tick calculation time puts calculations behind the real-time, the calculations tries to catch up with the real-time, resulting in shortened off-times.

Implementing and evaluating techniques to reduce jitter and its influence are worth further investigation.

5.5. Summary

The code generated from SCCharts is too complex to be run on an Atmega. The consequence is that the code generation of the SCChart has to be optimized, to create code that can be used on devices with low calculation capability. This result is strengthened by the fact that the Raspberry Pi is at its limits, when it is running the motor controller.

The Raspberry Pi with Raspbian is not a real-time capable system. This impacts the timings of the control signals. This limitation has its origin in the OS and not in the CPU. Therefore it is possible to use the Raspberry Pi hardware as a real-time system by removing the OS and using the cycle counter as a clock. The problem with this approach is its complexity: Any service used from the OS has to be reimplemented in a non interrupting way. The preferred solution to this problem is to minimize the influence of the OS to the minimum by using a real-time kernel and isolating the program on a CPU core. If a jitter time below 100 μs is needed, it is recommended to add a real-time capable system to create the control signals.

FPGAs are by design real-time capable systems that are only limited by their clock speeds and size. This analysis has demonstrated that SCChart, running on an FPGA, can produce reliable and precise control signals.

The nearly perfect timing created by the FPGA demonstrate that the performance of Timed SCCharts are only limited by the available calculation power and the real-time capability of the platform it is running on.

SCCharts in ASAP tick environments react on average in 1.5 tick function calculation times to an input event. The maximum reaction time is two times the calculation time. Even an environment that knows all future events cannot produce a better worst-case reaction time, only a better average. The introduction of dynamic ticks improves the average reaction time to a value much closer or equal to the calculation time. Dynamic ticks produce in most cases a perfect schedule of tick calculation. Only in situations where multiple events are in a burst that is shorter than one tick calculation a better schedule would be possible. There are still improvements possible that improve the average reaction time. The test demonstrates that the performance is nearly perfect. The consequence is that improvements in the tick functions runtime have a much bigger influence on the absolute value of the average than any further optimization to the environments tick scheduling. Increasing the calculation time of the environment, to get an even better schedule, would most likely decrease the overall performance.

Another advantage of dynamic ticks, demonstrated by these tests, is the reduction of tick calculations. This advantage depends on the type of input and the time a tick calculation takes. It is therefore not possible to describe the advantage with a fixed value. This test, however, proves that using a DTE removes all tick calculations that are not necessary.

6. Conclusion

The goal of this report is, to demonstrate the capabilities of Timed SCCharts in dynamic tick environments. This is achieved by the creation of a hardware-based demonstrator.

The design of the hardware demonstrator is focused on flexibility and evaluability. A stepper motor controller is selected as the base of the demonstrator. A single stepper motor would only allow to demonstrate the achieved precision with analysis data. A setup of two stepper motors was selected to provide a visual representation. The setup consists out of two partial disks, traversing each other. Any timing problems would lead to a collision, providing a good feedback to the observer.

The complete demonstrator is designed in a modular fashion to create a flexible platform. The clear interfaces between modules not only allow the replacement of components, but also provide opportunities for data capturing. This data is used to analyze and evaluate the behavior and performance of components. An FPGA was selected as the hardware to run the controller. FPGAs provide enough performance to run any controller and to provide the capability to capture information about the controller's behavior without impacting its performance. A program, running on an FPGA, is not influenced by an operating system or any other kind of software. This allows to monitor the performance without other influences, creating clear and expressive data.

A VHDL to SCChart compiler was implemented, based on an existing approach. This implementation of the compiler uses a new RTL structure that is compatible with DTEs. Using SCCharts to generate the RTL for FPGAs is in line with the current trend of high level syntheses.

The focus of the implementation is to create a setup that not only allows to understand the created precision, but also aids explanations of the function and role of each component. Modular design, the grouping of components by function and the usage of a mix of through hole and surface mounted components help to achieve this.

The flexible design allows to extend the evaluation from dynamic versus ASAP tick environments to performance comparisons of SCCharts on different platforms and between different compilation approaches. This substantiates the value of SCChart in embedded designs.

The analysis proves that Timed SCCharts can control systems with high real-time requirements. The dynamic tick environment is an important factor to increase the performance, while reducing the amount of computations. In the case of this demonstrator the dynamic tick environment used less than 0.01 percent of the ticks to perform the same task with a better performance than a ASAP approach. Making time a first class citizen has not only enabled this performance, but also increased the expressiveness of SCCharts.

6.1. Future Work

FPGA-based performance analysis is a valuable data source. This chapter contains ideas to use it to their full potential.

The SCCharts to VHDL compiler is far from its full potential, despite the performance, demonstrated in this report. Pointers to improvements of the SCChart to VHDL compiler and Timed SCChart in general are provided in this section.

6.1.1. Further Tests and Analysis

The analysis provides multiple starting points for further analysis. The big problem of the Raspberry Pi-based controller is the interference of the operating system. Removing the operation system and running the code bare metal would provide a better results. This is complicated and uncommon, contradicting the idea to use common platforms for performance analysis. A softcore on an FPGA is a better base for analysis, if the only goal is to analyze the performance of SCCharts code. The advantage of this approach is the possibility to log detailed and precise data about the functions run. Furthermore, it is possible to influence the precision of the time source and to simulate an environment in real-time.

The biggest advantage of an FPGA-based analysis is the high precision. The time on a CPU has a precision between 500 and 10000 nanoseconds, ignoring the timing error created by the kernel, which can exceed 350 μs. The time on an FPGA can be measured down to 2 nanosecond intervals, and even smaller intervals if high performance FPGA are used.

The Zynq-7 line-up of FPGA-based SoCs provides an interesting platform for tests and performance comparison. The combination of ARM cores with FPGA fabric allows many different types of tests. The FPGA part can run native VHDL code or a soft-core running program code. The ARM cores can be used bare metal or with Linux as an operating system. The fast interconnects between CPU and FPGA and the shared memory can be used to monitor or influence the program that is tested. An example for such a test would be an SCChart running under Linux, but the time for the DTE would be obtained from the FPGA. This setup would remove any timing errors caused by the OS that are not caused by kernel interruptions.

This setup can also be used to investigate another open question: Is it better to focus on developments that remove as much tick calculation time jitter as possible or on extension that mediate negative side effects of jitter?

6.1.2. Improvements of the SCChart to VHDL Compiler

The main problem of the SCChart to VHDL compiler is that the resulting RTL uses excessive amounts of floor space in the FPGA and has a low resource utilisation due to the long critical path.

Integrating SCCharts in the development of hardware descriptions could be worth further investigations. A combination of an improved version of the compiler with the

newly created dynamic tick environment and automated constrain generation could be a powerful tool.

Generating pipelined Code instead of using MCPs

With increasing complexity of the SCChart the length of the longest data path increases. This decreases the utilization of the resources. The pipeline-based compilation approach was introduced in Section 3.2.2. The advantage of this approach is the higher utilization of the resources.

Optimized Function Order

In contrast to CPUs, many operations on FPGAs need the bits of the input at different point in time. The bits of the output are also ready at different points in time. An example is the addition. The carry goes from Least Significant Bit (LSB) to Most Significant Bit (MSB). The LSB output is n carry delays earlier ready than MSB. An addition takes a LUT and n carry delays in time. Two additions just take two LUTs and n carry delays in time. An addition combined with an operation that also consumes one LUT of time, but has a carry that moves from MSB to LSB takes two LUT and $2n$ carry worth of time. The consequence is that the order of operations influences the run-time.

Another problem is that the operation order is not optimized for parallel execution. An often occurring pattern is a boundary check, `x := x>0 ? x : 0`, followed by some other operation, `x := x+y`; in this example. The SSA-form of this code is:

```
g := x1 > 0;
x2 := g ? x1 : 0;
x3 := x2 + y;
```

All operations have to be calculated sequentially. This code can be optimized for parallelism:

```
g := x1 > 0;
x2 := x1 + y;
x3 := g ? x2 : y;
```

In the optimized code it is possible to calculate the first two statements in parallel. This is an example of an optimization without an drawback. There are more techniques to optimize the netlist for parallel execution. Some of them increase the overall number of calculations to reduce the calculation time. This can be demonstrated with the same example by setting another boundary, such as 5 instead of 0:

```
g := x1 > 5;
x2a := x1 + y;
x2b := 5 + y;
x3 := g ? x2a : x2b;
```

This optimized code has one operation more than the original code, but has a shorter critical path, since the three first statements can be calculated in parallel.

This type of optimization trades space for computation speed, if the MCP-based compiler is used. The primary use of this type of optimization is in the generation of pipelined code, since it has no downside if an existing resource is available for the additional computation.

Use DSPs and other Hard Cores

Modern FPGAs have different special propose hardware slices. They range from simple multipliers over DDR interfaces to complete processor cores. Using these components increases the performance immensely. The most interesting type of hardened cores are DSPs. Using them increases the performance and reduces the used space. The main disadvantages of using DSPs and other hard cores is that the compiler has to know which resources are available, and the code becomes hardware and manufacturer dependent.

6.1.3. Smart Constructors for Time

Integer and floats are the possible data types for time in SCCharts. The interaction between clocks and other variables are currently done without casting. Therefore it is necessary to manually check the types and to add the time unit conversion. A smart constructor for time would have the form of a time unit: day, min, s or ms for example. These constructors can be used in combination with constants and variables: clock = 30 s or clock >= x min. The addition of these smart constructors create shorter and cleaner code: The statement clock >= x / 1000000000 would become much shorter: clock >= x ns. Another advantage is the increase in visibility of time dependent transition. This change can also be the base for host-specific clocks. A library, written in the target language, would provide the math functions for the host-specific clock type. The introduction of a clock data type would be a big advantage on platforms that need clock types that exceeds their bit width, such as 8-bit MCUs.

List of Figures

List of Tables

Acronyms

ADC Analog to Digital Converter

ASIC Application-Specific Integrated Circuit

CAD Computer-Aided Design

CCM Continuous Conduction Mode

CE Clock Enable

CLK Clock

CPU Central Processing Unit

DFA Deterministic Finite Automaton

DSL Domain Specific Language

DSP Digital Signal Processor

DTE Dynamic Tick Environment

FF FlipFlop

FPGA Field-Programmable Gate Array

FPU Floating Point Unit

GaNFET Gallium Nitride Field-Effect Transistor

GND GrouND

GPIO General-Purpose Input/Output

HDL Hardware Description Language

IC Integrated Circuit

IDE Integrated Development Environment

IIOT industrial internet of things

IO Input/Output

KIELER Kiel Integrated Environment for Layout Eclipse RichClient

LSB Least Significant Bit

LUT LookUp Table

LVCMOS Low Voltage Complementary Metal Oxide Semiconductor

LVDS Low-Voltage Differential Signaling

LVTTL Low-Voltage Transistor-Transistor Logic

MCP Multi-Cycle Path

MCU Micro Controller Unit

MoC Model of Computing

MPU Micro Processor Unit

MSB Most Significant Bit

OpAmp Operation Amplifier

PCB Printed Circuit Board

PLL Phase-Locked Loop

RTL Register Transfer Level

SCChart Sequentially Constructive Statechart

SMD surface-mount device

SMoC Synchronous Model of Computing

SPL Synchronous Programming Languages

SSA Static Single Assiment

THT Through-hole technology

VCC Voltage at Common Collector

VHDL Very high speed integrated circuit Hardware Description Language

Bibliography

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] M. Bendjedia, Y. Ait-Amirat, B. Walther, and A. Berthon. Sensorless control of hybrid stepper motor. In *2007 European Conference on Power Electronics and Applications*, pages 1–10, Sep. 2007.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, Jan. 2003. IEEE.

[4] G. Boisset, B. Robertson, and H. S. Hinton. Design and construction of an active alignment demonstrator for a free-space optical interconnect. *IEEE Photonics Technology Letters*, 7(6):676–678, 1995.

[5] T. Bourke and M. Pouzet. Zélus: a synchronous language with odes. In *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013*, pages 113–118, Philadelphia, PA, USA, Apr. 2013.

[6] T. Bourke and A. Sowmya. Delays in Esterel. In *SYNCHRON'09—Proceedings of Dagstuhl Seminar 09481*, number 09481 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 22–27 Nov. 2009.

[7] J. Colaço, B. Pagano, and M. Pouzet. SCADE 6: A formal language for embedded critical software development (invited paper). In *11th International Symposium on Theoretical Aspects of Software Engineering TASE*, pages 1–11, Sophia Antipolis, France, Sept. 2017.

[8] R. Condit and D. W. Jones. Stepping motors fundamentals. *Microchip Inc. Publication AN907*, pages 1–22, 2004.

[9] H. C. P. Dymond, J. Wang, D. Liu, J. J. O. Dalton, N. McNeill, D. Pamunuwa, S. J. Hollis, and B. H. Stark. A 6.7-ghz active gate driver for gan fets to combat overshoot, ringing, and emi. *IEEE Transactions on Power Electronics*, 33(1):581–594, Jan 2018.

[10] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, (9):785–793, 1992.

[11] G. Johannsen. Hardwaresynthese aus SCCharts. Master thesis, Kiel University, Department of Computer Science, Oct. 2013. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/gjo-mt.pdf.

[12] Y. Lin. Using FPGAs to solve challenges in industrial applications. *EE times*, 2011.

[13] B. Przybus. Xilinx redefines power, performance, and design productivity with three new 28 nm fpga families: Virtex-7, kintex-7, and artix-7 devices. *Xilinx White Paper*, 2010.

[14] A. Schulz-Rosengarten, R. von Hanxleden, F. Mallet, R. de Simone, and J. Deantoni. Time in SCCharts. In *Proc. Forum on Specification and Design Languages (FDL '18)*, Munich, Germany, Sept. 2018.

[15] A. Schulz-Rosengarten, R. von Hanxleden, F. Mallet, R. de Simone, and J. Deantoni. Time in SCCharts. Technical Report 1805, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018. ISSN 2192-6247.

[16] A. Schulz-Rosengarten, R. von Hanxleden, F. Mallet, R. de Simone, and J. Deantoni. Time in SCCharts. In T. J. Kazmierski, S. Steinhorst, and D. G. e, editors, *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2018*, pages 1–25. Springer, 2020.

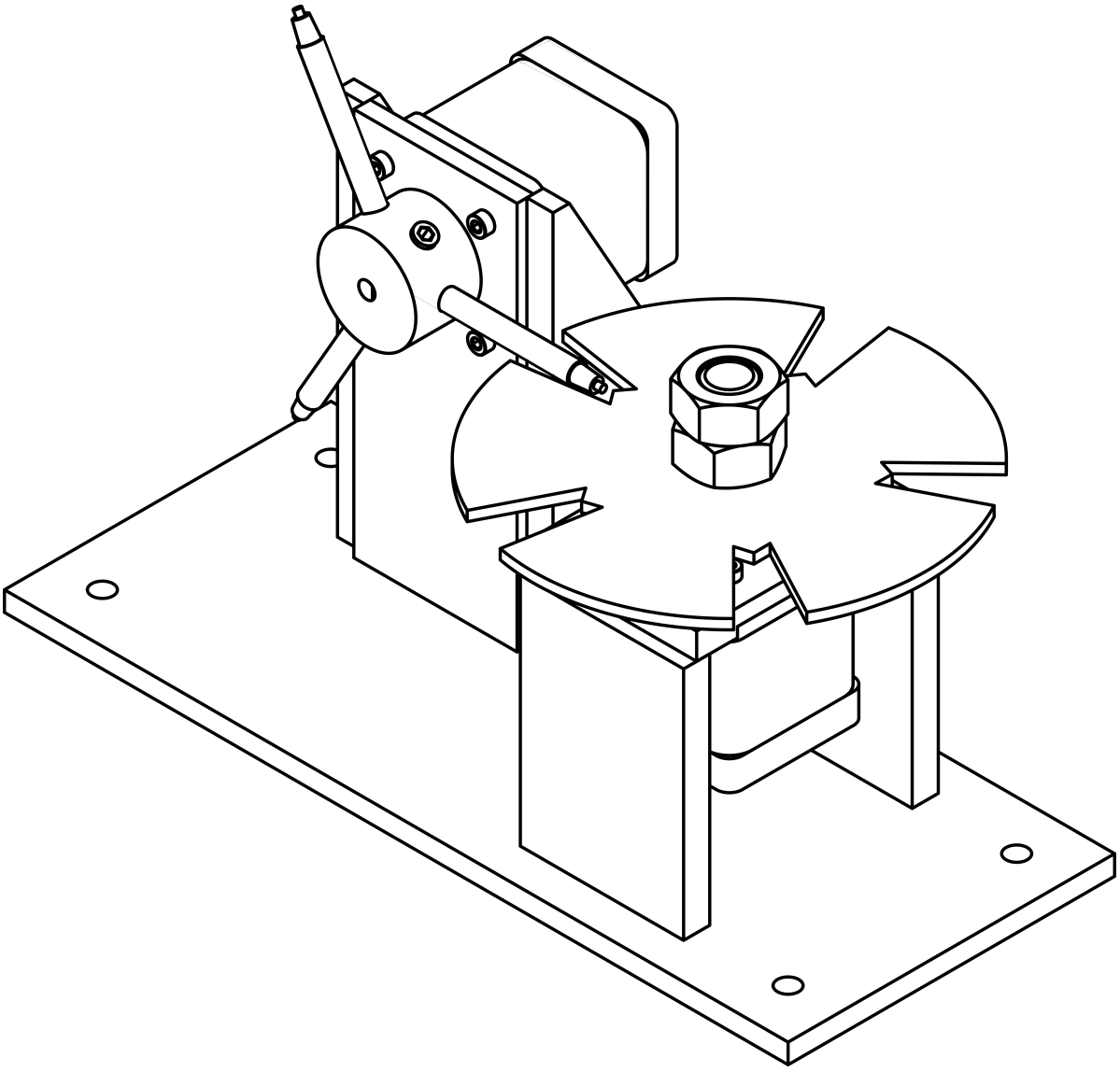
[17] R. von Hanxleden, T. Bourke, and A. Girault. Real-time ticks for synchronous programming. In *Proc. Forum on Specification and Design Languages (FDL '17)*, Verona, Italy, Sept. 2017.

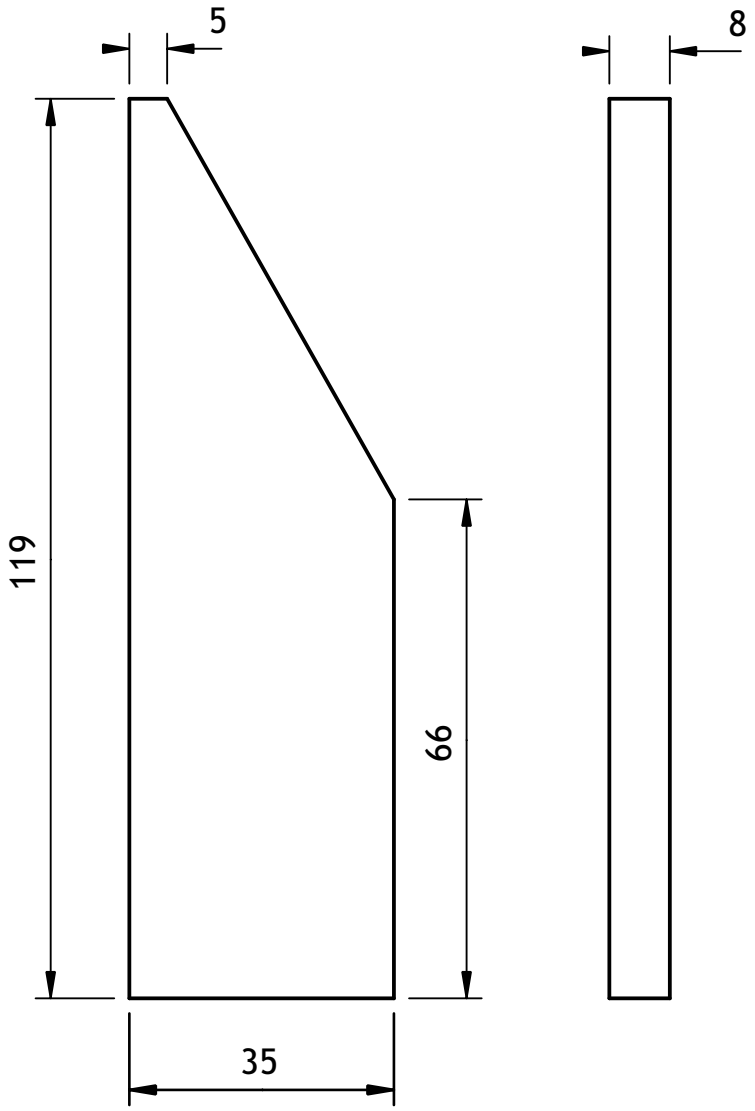
[18] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.

[19] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM. Long version: Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274.

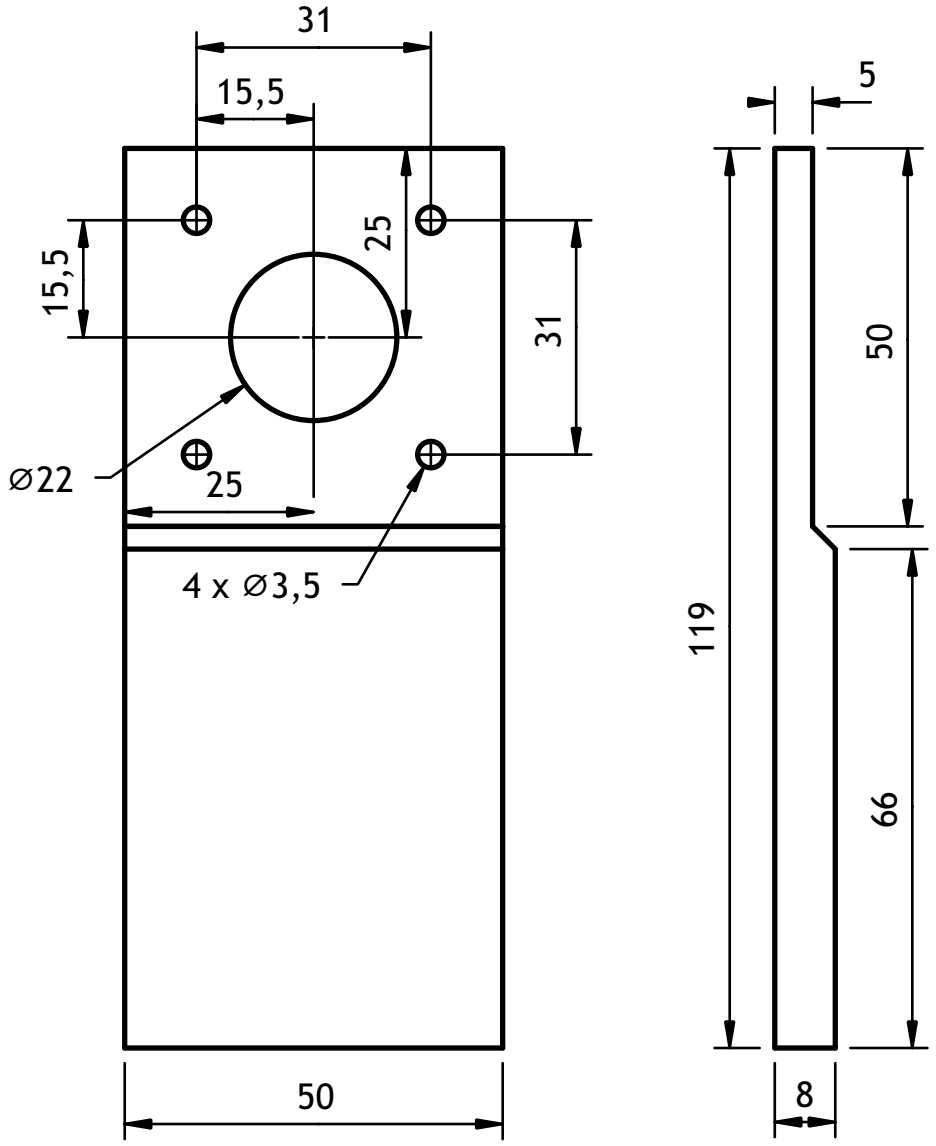
[20] M. Wissolik, D. Zacher, A. Torza, and B. Da. Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance. *Xilinx Whitepaper*, 2017.

A. Technical Drawings





Created by: Andreas Boysen	Title: reinforcement metal			
Supplementary information: tolerance: DIN-ISO 2768-1 g material: S235 weight: 0,2kg	Size: A4	Sheet: 1 / 14	Scale: 1:1	
	Part number: MTAB-01-001 Drawing number: MTAB-01-001-01			
	Date: 05/11/2019		Revision: REV A	



Created by:	Title:
Andreas Boysen	motor holder star

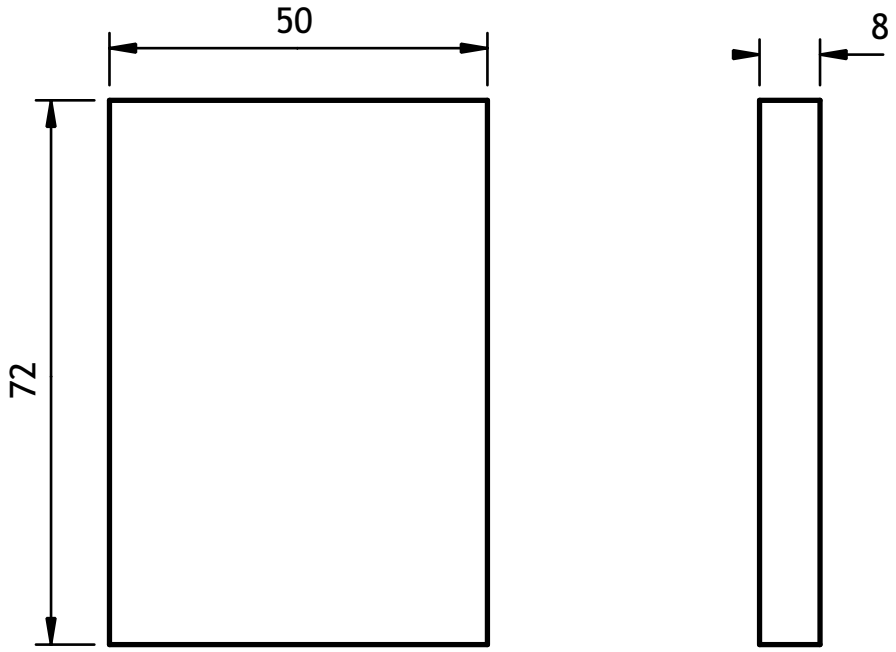
Supplementary information:
tolerance: DIN-ISO 2768-1 g
material: S235
weight: 0,27kg

Size:	Sheet:	Scale:
A4	2 / 14	1:1

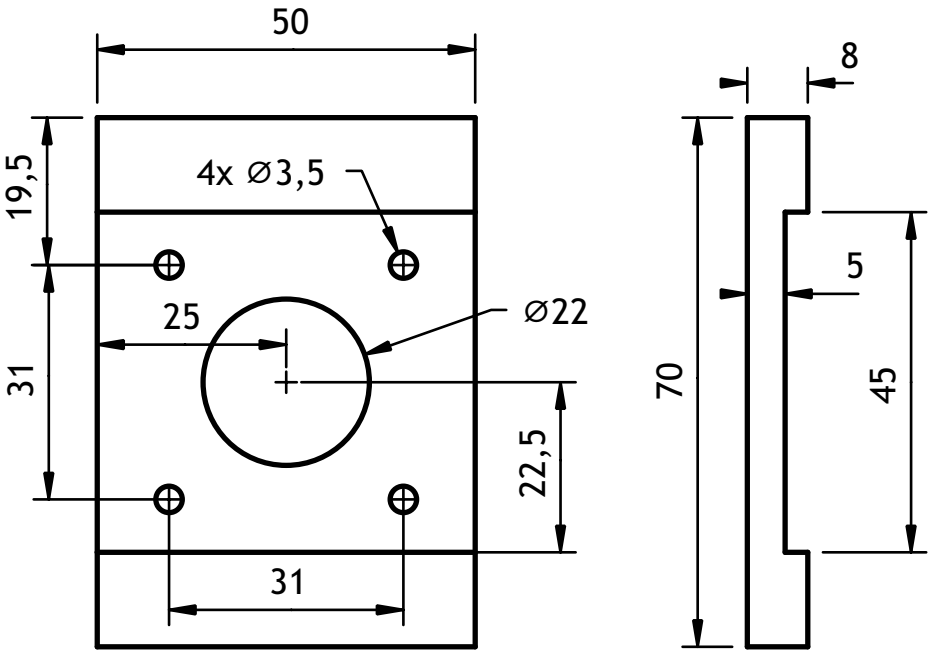
Part number:
MTAB-01-002

Drawing number:
MTAB-01-002-01

Date:	Revision:
03/11/2019	REV A

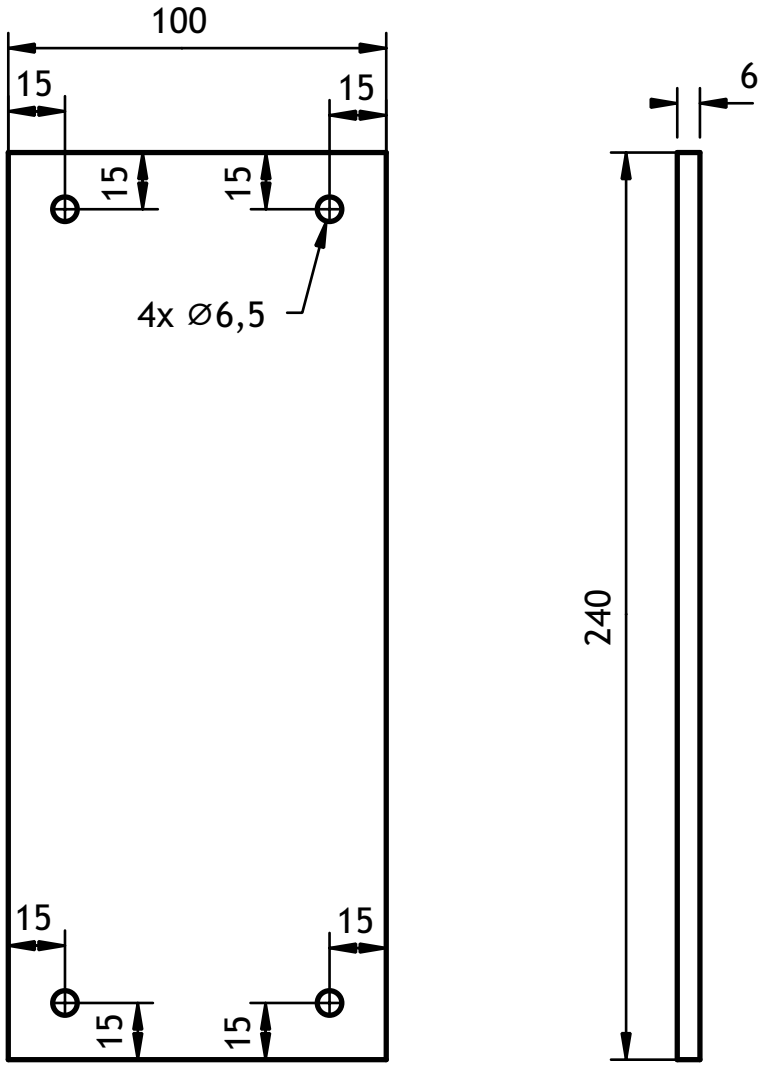


Created by:	Title:			
Andreas Boysen	pillar			
Supplementary information: tolerance: DIN-ISO 2768-1 g material: S235 weight: 0,22kg	Size: A4	Sheet: 3 / 14	Scale: 1:1	
	Part number: MTAB-01-003 Drawing number: MTAB-01-003-01			
	Date: 05/11/2019		Revision: REV A	



Created by:	Title:
Andreas Boysen	motor holder disc

Supplementary information:	Size:	Sheet:	Scale:
tolerance: DIN-ISO 2768-1 g material: S235 weight: 0,21kg	A4	4 / 14	1:1
	Part number: MTAB-01-004 Drawing number: MTAB-01-004-01		
	Date: 05/11/2019		Revision: REV A



Created by:	Title:
Andreas Boysen	ground plate

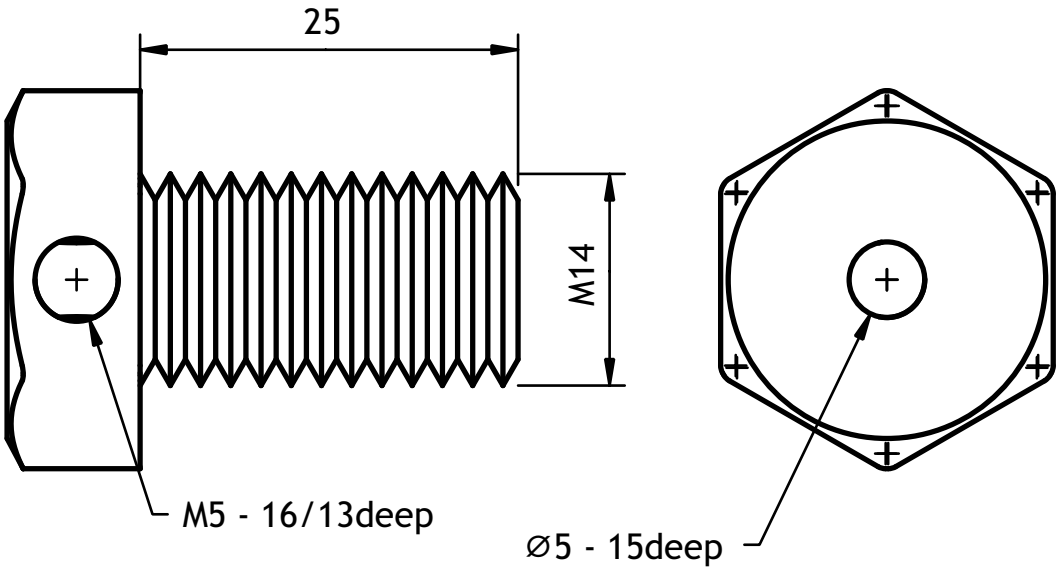
Supplementary information:
tolerance: DIN-ISO 2768-1 g
material: S235
weight: 1,13kg

Size:	Sheet:	Scale:
A4	5 / 14	1:2

Part number:
MTAB-01-005

Drawing number:
MTAB-01-005-01

Date:	Revision:
05/11/2019	REV A



Created by:	Title:
Andreas Boysen	disc mount

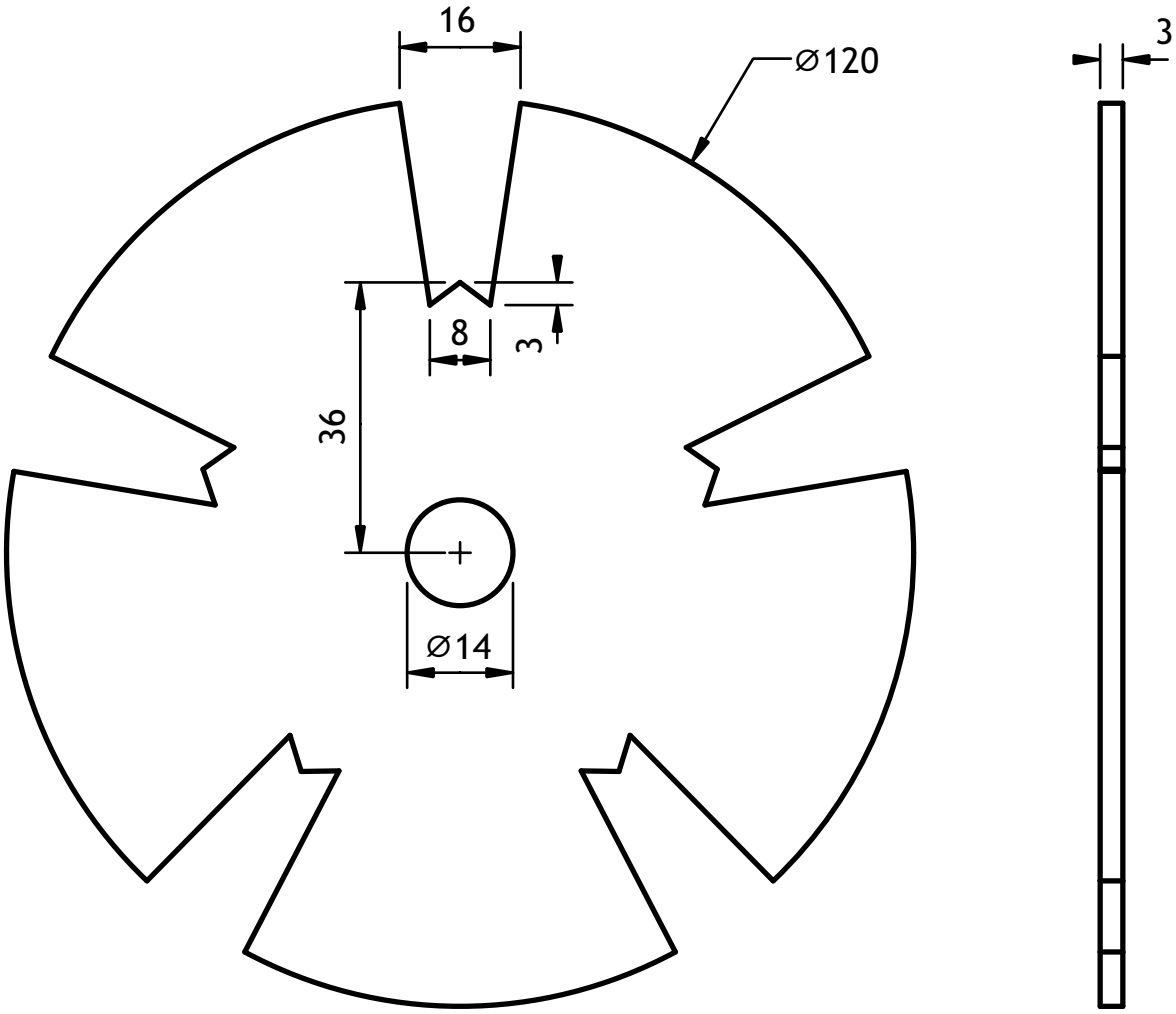
Supplementary information:
tolerance: DIN-ISO 2768-1 g
raw material: screw DIN 933 M14x25 8.8
weight: 0,1kg

Size:	Sheet:	Scale:
A4	6 / 14	2:1

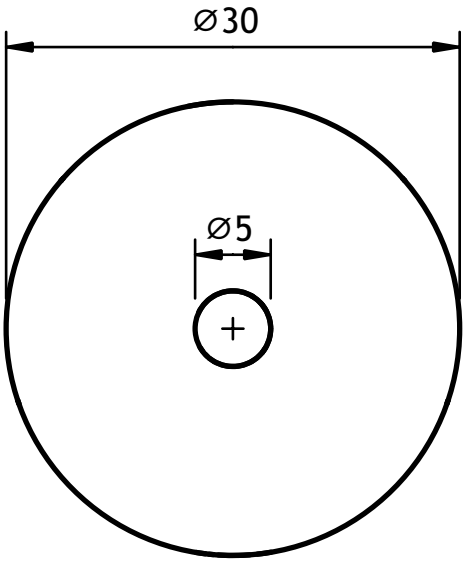
Part number:
MTAB-02-001

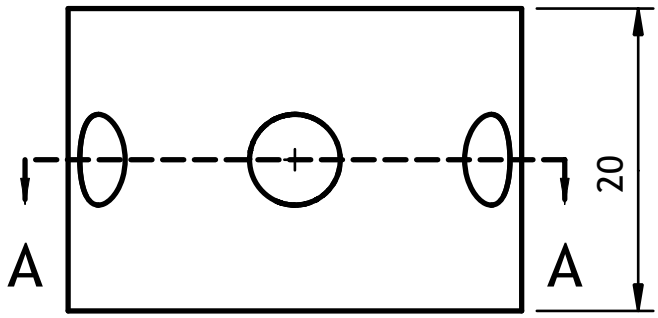
Drawing number:
MTAB-02-001-01

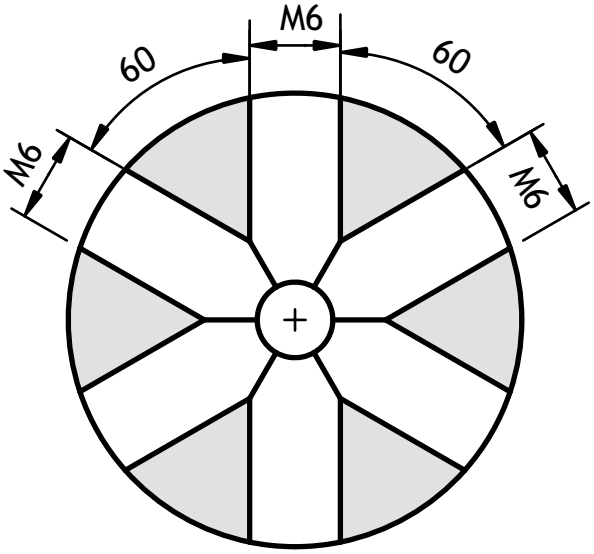
Date:	Revision:
05/11/2019	REV A



Created by:	Title:			
Andreas Boysen	disc			
Supplementary information: tolerance: DIN-ISO 2768-1 g material: PETG weight: 0,01kg	Size: A4	Sheet: 7 / 14	Scale: 1:1	
	Part number: MTAB-02-002 Drawing number: MTAB-02-002-01			
	Date: 05/11/2019		Revision: REV A	







cut A - A

Created by: Andreas Boysen	Title: rod mount

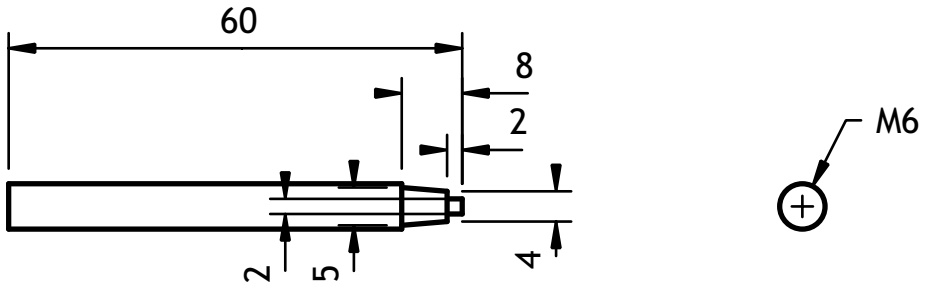
Supplementary information:
tolerance: DIN-ISO 2768-1 g
material: S235
weight: 0,02kg

Size:	Sheet:	Scale:
A4	8 / 14	2:1

Part number: MTAB-03-001

Drawing number: MTAB-03-001-01

Date:	Revision:
06/11/2019	REV A



Created by:	Title:
Andreas Boysen	rod

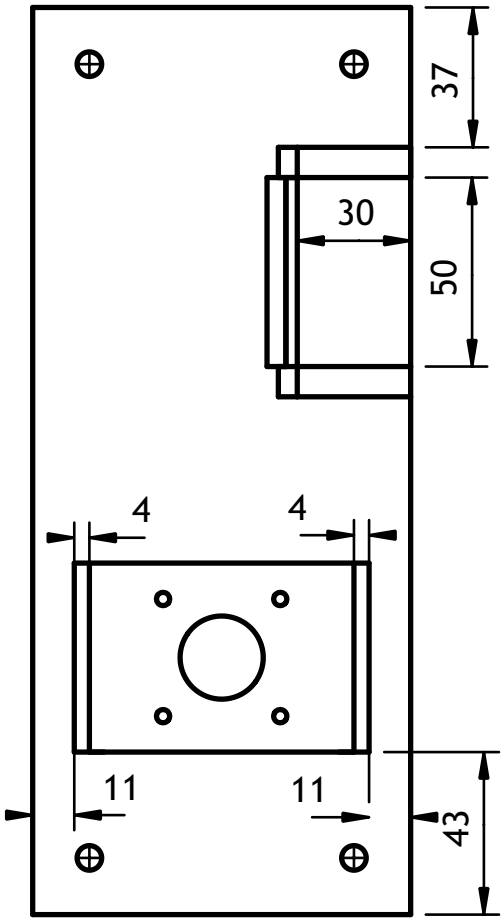
Supplementary information:
tolerance: DIN-ISO 2768-1 g
material: brass
weight: 0,005kg

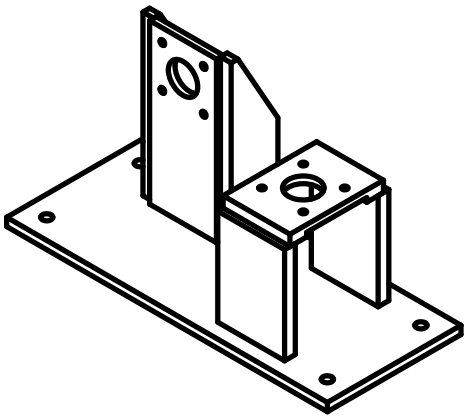
Size:	Sheet:	Scale:
A4	9 / 14	1:1

Part number:
MTAB-03-002

Drawing number:
MTAB-03-002-01

Date:	Revision:
03/11/2019	REV A





pos.	qty:	title:
1	2 pcs.	MTAB-01-001, reinforcement metal
2	1 pc.	MTAB-01-002, motor holder star
3	2 pcs.	MTAB-01-003, pillar
4	1 pc.	MTAB-01-004, motor holder disc
5	1 pc.	MTAB-01-005, ground plate

Created by:
Andreas Boysen

Title:
assembly drawing (welding)

Supplementary information:
-weld thickness: 3mm
-weld components all around

Size: A4

Sheet: 10 / 14

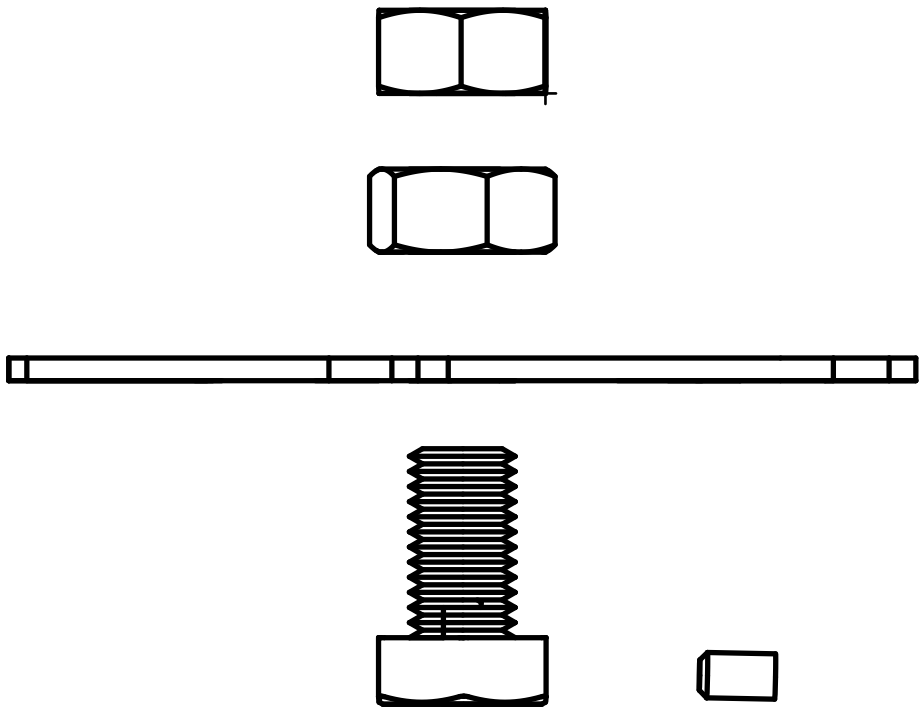
Scale: 1:2

Part number:
MTAB-01-000

Drawing number:
MTAB-01-000-01

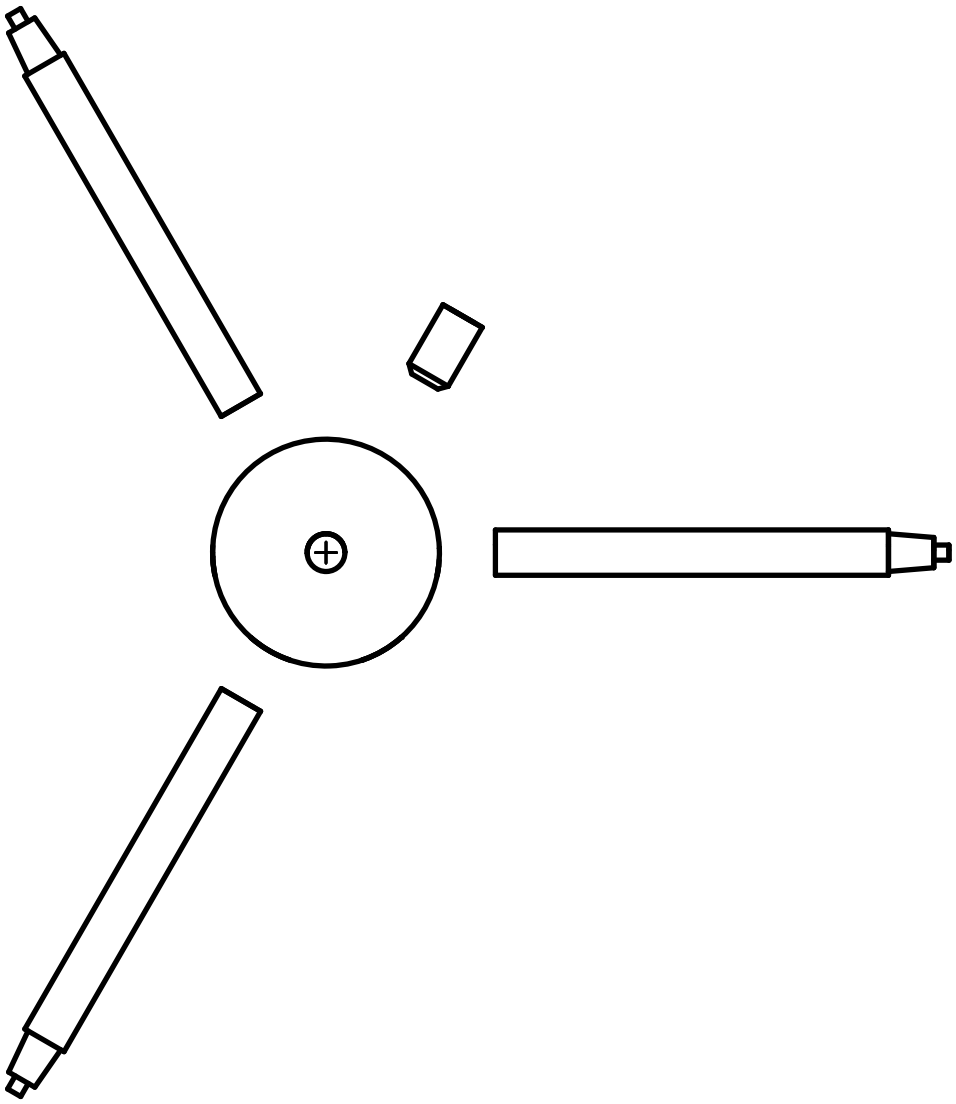
Date: 12/12/2019

Revision: REV A



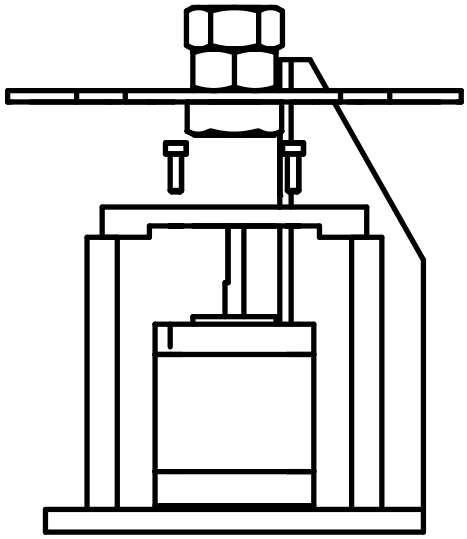
pos.	qty:	title:
6	1 pc.	MTAB-02-001, disc mount
7	1 pc.	MTAB-02-002, disc
13	1 pc.	grub screw M5x15
14	2 pcs.	nut M14

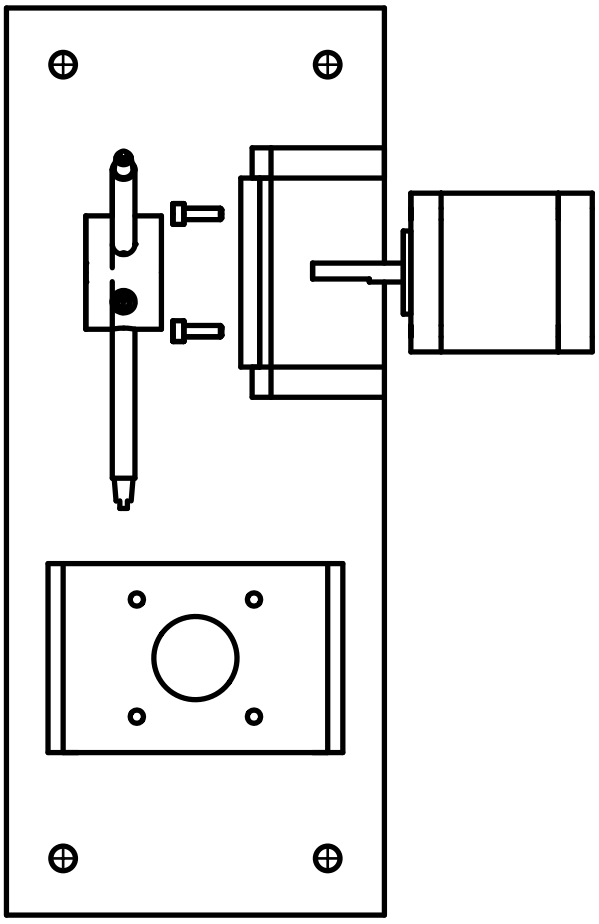
Created by: Andreas Boysen	Title: disc assembly		
Supplementary information: use Loctite	Size: A4	Sheet: 11 / 14	Scale: 1:1
	Part number: MTAB-02-000 Drawing number: MTAB-02-000-01		
	Date: 15/12/2019		Revision: REV A

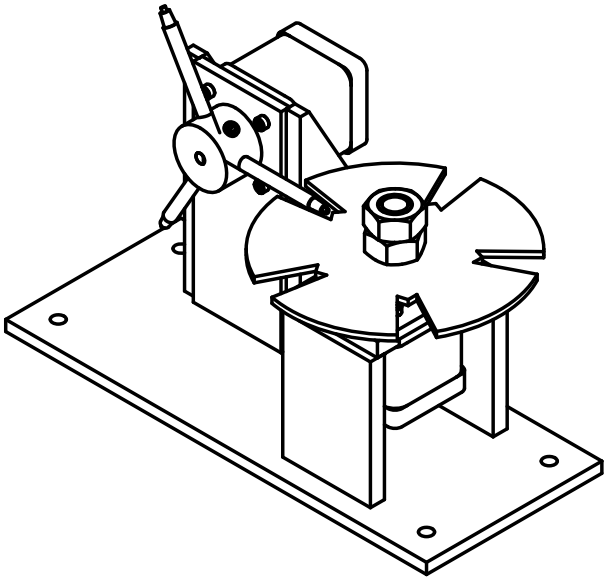


pos.	qty:	title:
8	1 pc.	MTAB-03-001, rod mount
9	3 pcs.	MTAB-03-002, rod
12	1 pc.	grub screw M6x15

Created by: Andreas Boysen	Title: rod assembly		
Supplementary information: use Loctite	Size: A4	Sheet: 12 / 14	Scale: 1:1
	Part number: MTAB-03-000 Drawing number: MTAB-03-000-01		
	Date: 15/12/2019		Revision: REV A







pos.	qty:	title:
	1 pc.	MTAB-01: base frame
	1 pc.	MTAB-02: disc assembly
	1 pc.	MTAB-03: rod assembly
10	2 pcs.	stepper motor
11	8 pcs.	hexagon socket screw M3x16

Created by: Andreas Boysen	Title: assembly drawing		
Supplementary information:	Size: A4	Sheet: 13 / 14	Scale: 1:2
	Part number: MTAB-04-000		
	Drawing number: MTAB-04-000-01		
	Date: 16/12/2019		Revision: REV A

pos.	qty:	title:	standard:	material:	comment:
		MTAB-01-000: base frame			welded construction
1	2 pcs.	MTAB-01-001, reinforcement metal		steel: S235	
2	1 pc.	MTAB-01-002, motor holder star		steel: S235	
3	2 pcs.	MTAB-01-003, pillar		steel: S235	
4	1 pc.	MTAB-01-004, motor holder disc		steel: S235	
5	1 pc.	MTAB-01-005, ground plate		steel: S235	
		MTAB-02-000: disc assembly			
6	1 pc.	MTAB-02-001, disc mount	(DIN 933)	steel	
7	1pc.	MTAB-02-002, disc		PETG	
		MTAB-03-000: rod assembly			
8	1 pc.	MTAB-03-001, rod mount		steel: S235	
9	3 pcs.	MTAB-03-002, rod		brass	
		standard parts			
10	2 pcs.	stepper motor	NEMAR 17		max. length: 50mm
11	8 pcs.	hexagon socket screw M3x16	DIN 912		
12	1 pc.	grub screw M6x15	DIN 913		
13	1 pc.	grub screw M5x15	DIN 913		
14	2 pcs.	nut M14	DIN 934		grade: 8.8

Created by: Andreas Boysen

Title: parts list

Supplementary information:

Size: A4

Sheet: 14 / 14

Scale:

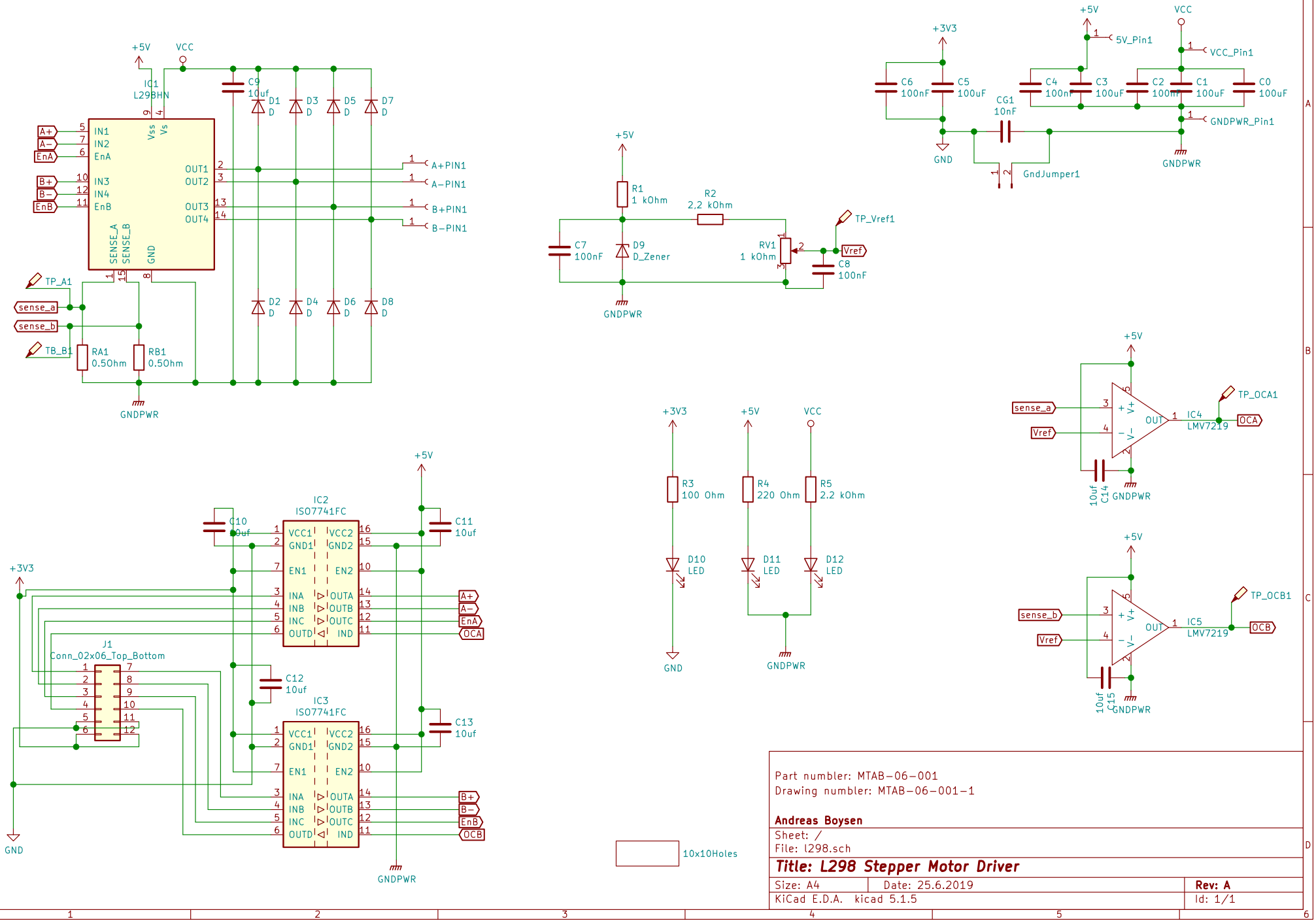
Part number:

Drawing number:

Date: 16/12/2019

Revision: REV A

B. Schematics



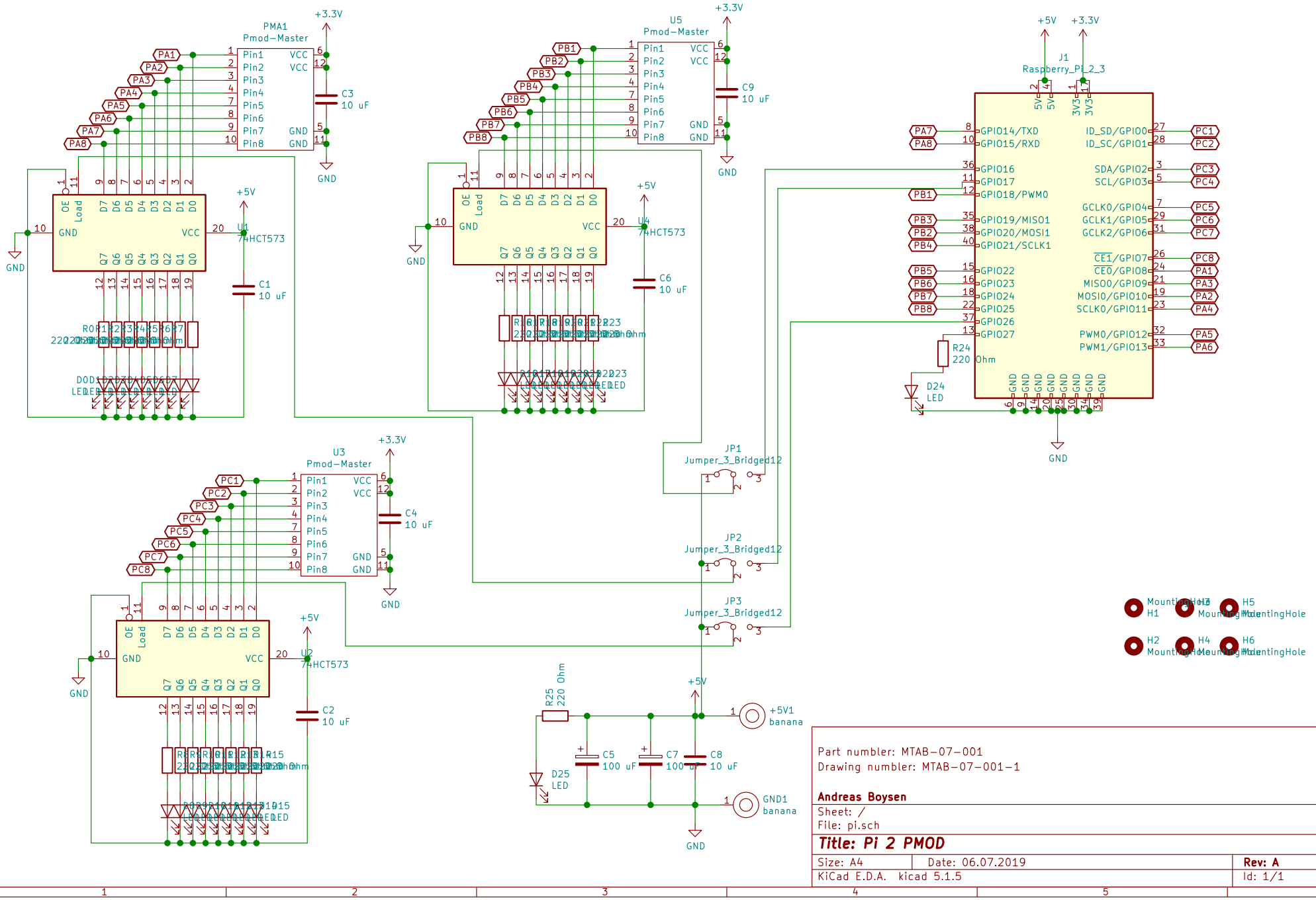
Part numbler: MTAB−06−001
Drawing numbler: MTAB−06−001−1

Andreas Boysen

Sheet: /
File: l298.sch

Title: L298 Stepper Motor Driver

Size: A4	Date: 25.6.2019	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1



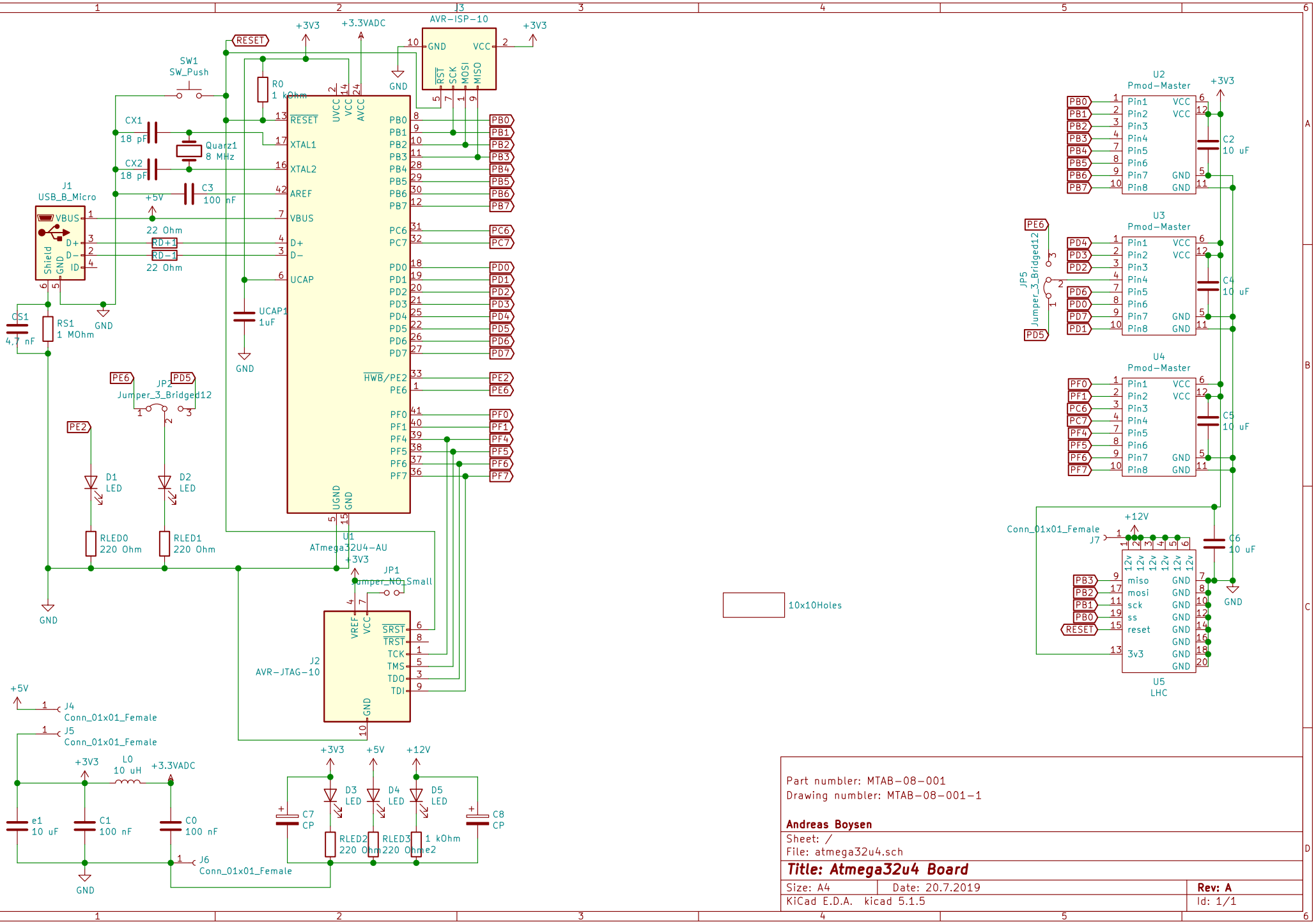
Part numbler: MTAB-07-001
Drawing numbler: MTAB-07-001-1

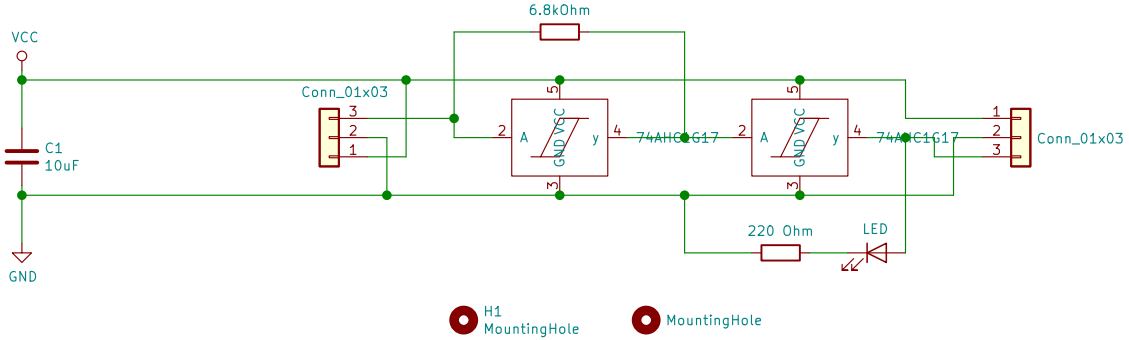
Andreas Boysen

Sheet: /
File: pi.sch

Title: Pi 2 PMOD

Size: A4	Date: 06.07.2019	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1


AVR-ISP-10
SW1
SW_Push
R0
1 kOhm
CX1
18 pF
CX2
18 pF
Quarz1
8 MHz
C3
100 nF
J1
USB_B_Micro
22 Ohm
22 Ohm
CS1
4,7 nF
RS1
1 MOhm
UCAP1
1uF
JP2
Jumper_3_Bridged12
D1
LED
D2
LED
RLED0
220 Ohm
RLED1
220 Ohm
U1
ATmega32U4-AU
JP1
Jumper_NO_Small
J2
AVR-JTAG-10
J4
Conn_01x01_Female
J5
Conn_01x01_Female
L0
10 uH
e1
10 uF
C1
100 nF
C0
100 nF
J6
Conn_01x01_Female
C7
CP
C8
CP
D3
LED
D4
LED
D5
LED
RLED2
220 Ohm
RLED3
220 Ohme2
1 kOhm
U2
Pmod-Master
U3
Pmod-Master
U4
Pmod-Master
JP5
Jumper_3_Bridged12
C2
10 uF
C4
10 uF
C5
10 uF
C6
10 uF
Conn_01x01_Female
J7
U5
LHC
10x10Holes
Part number: MTAB-08-001
Drawing number: MTAB-08-001-1
Andreas Boysen
Sheet: /
File: atmega32u4.sch
Title: Atmega32u4 Board
Size: A4
Date: 20.7.2019
Rev: A
KiCad E.D.A. kicad 5.1.5
Id: 1/1



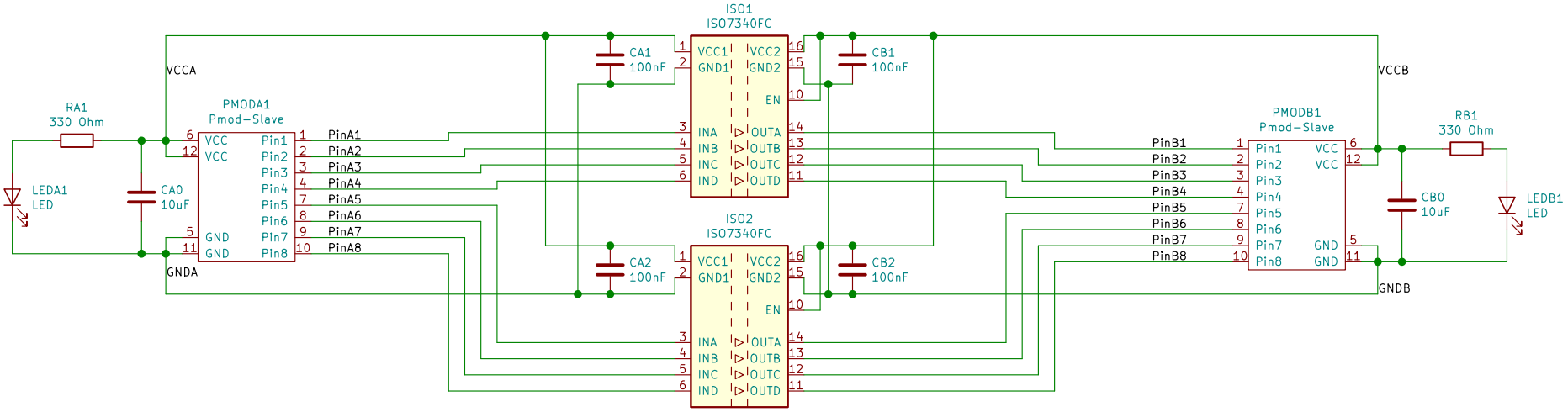
Part numbler: MTAB-09-001
Drawing numbler: MTAB-09-001-1

Andreas Boysen

Sheet: /
File: debouncing.sch

Title: Debouncer

Size: A4	Date: 17.7.2019	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1



Part numbler: MTAB−10−001
Drawing numbler: MTAB−10−001−1

Andreas Boysen

Sheet: /
File: pmodSniffer.sch

Title: Logic Sniffer

Size: A4	Date: 30.09.2019	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

C. Design Principles

A demonstrator is between a proof of concept and a full product. This influences the design principles used for designing a demonstrator. This section discusses some of this design principles for demonstrators.

Demonstration and analysis: A demonstrator is designed for demonstration and analysis of the features. The analysis part not only allows for further development, it also enhances the demonstration by displaying the analysis data.

Modular design with clear interfaces: This is more a general design rule but applies especially for demonstrators. Modular design enables the reuse of parts, replacement of faulty parts without complete replacement, and late changes in the setup. Good interfaces allow to design part after part. This is not be possible if a design decision inside one component leads to necessary changes in another component. A clear interface can also be utilized in a demonstration.

Don't overengineer, KISS: There are always better ways to do something that are more fancy, have more performance or efficiency. This is a demonstrator, not a final product. If it does not interfere with the purpose of the demonstrator: Keep It Simple, Stupid.

Decide and don't hide: In part selection, schematic and PCB design occur many situations where a problem can be solved by hardware design or in later stages by software. In these cases put the solution on a place where it enhances the demonstration. Never solve problems in a hidden manner. An example: Do not use a slow Operation Amplifier (OpAmp) to filter high frequency. Use a low-pass filter and a good OpAmp. Hiding the high frequency in this case limits the understanding a spectator can get from the demonstration. Furthermore, you are using a non-specified property of a component. This can lead to rare events in witch the high frequency is not filtered. Another problem is that this property can change in later revision of the component, and it is no longer possible to rebuild the demonstrator.

Design for failure: Designing with demonstration and testing in mind means to design with failures in mind. It is possible that nets that should not be connected are connected during testing. In a good design this error does not spread through the whole installation. In this way the damage is limited to one component. This can be done by the usage of current and voltage limitation, fuses and galvanic isolation. The interfaces are a good place for the galvanic isolation of digital signals.

No tests without testpoints: Integrate test points. Test points are essential for debugging and analysis of hardware. Build them as accessible as possible.

Mounting: Another often forgotten requirement is mounting. Every component must be mountable. One of the best and easiest ways to mount something are mounting holes.