# FW implementation of SMILE SXI Radiation Shutter Control System

Anders Lerum Alme


Master of Science in Physics

University of Bergen

June 2019

2019

FW implementation of SMILE SXI Radiation Shutter Control System

Anders Lerum Alme

http://bora.uib.no/

# Abstract

In 2023 a Solar Wind Magnetosphere Ionosphere Link Explorer (SMILE) satellite is set to launch to explore how solar wind interacts with the earth's magnetosphere and ionosphere. The SMILE space mission is a joint operation between the European Space Agency (ESA) and the Chinese Academy of Science, where scientists and industry are contracted to provide the satellite and its instruments.

A Soft X-ray Imager (SXI) which is one of those instruments, is designed to detect low energy electrons in this interaction. As the satellite will orbit the earth, it will during a percentage of the orbit come within the radiation belt of the earth, where there are high energy particles. To protect the detectors in the SXI instrument from the high energy radiation near the earth, a radiation shutter is being developed by a team at the University of Bergen to enclose the detector inside the satellite when needed.

This device is split into the radiation shutter mechanics (RSM) and the radiation shutter electronics (RSE) that is going to control the operation of the RSM. This work covers the continued development, implementation, testing and verification of the RSE.

This thesis explains the functionality of the RSE and the implementation of that system on a field-programmable gate array (FPGA). The RSE will be commanded by a central master that oversees the operation of the whole SXI instrument. A reliable communication protocol is designed and implemented to be able to communicate with the master. The RSE will be able to perform different operations, including opening and closing the shutter, reading different sensors related to the shutter operations and it will be able to protect the SXI detectors should the master data process unit (DPU) fail.

The system has been tested with both simulations with a test bench and on a breadboard. These tests have been done to check that the RSE operates as desired and that a breadboard version of the shutter will be able to open and close.

# Acknowledgements

First and foremost, I would thank my supervisor and the man responsible for the design and development of the Radiation Shutter Electronics, *Professor Kjetil Ullaland*. He has been a tremendous help with developing an actual useful design and excellent guidance in writing this thesis, especially when I was struggling at different times. I would thank him for unbeknownst to me, nominating me to be in charge of buying in snacks and drinks for the Christmas party, which I assume is the highest honour a student of his can achieve.

I give my thanks to *Ove Lylund*, whose work this thesis continues, *Chief Engineer Bilal Hasan Qureshi* who have both tested my design and collaborated with me in the testing of the RSM, and *Senior Engineer Shiming Yang* for his development of the hardware I got to use. I would also thank *Senior Engineer Georgi Genov*, who helped us set up the measurements and the rest of the SMILE team for their work with the SMILE project.

For tips, tricks and solutions to problems, both *Ola Grøttevik* and *Associate Professor Johan Alme*, who also helped with my thesis, have my gratitude. Also, *Magnus Rentsch Ersdal* has my thanks. He helped me with making a working test program when I was handed a python program for the first time in my life.

At last, I would like to thank my friends and my family for support and encouragement. My time at the University would not have been the same without them.

# Content

# Abbreviations

| | | | |
|---|---|---|---|
| **SMILE** | Solar Wind Magnetosphere Ionosphere Link Explorer | **TID** | Total-Ionising Dose |
| **ESA** | European Space Agency | **EBB** | Elegant breadboard |
| **SXI** | Soft X-ray Imager | **RAM** | Random-Access Memory |
| **RSM** | Radiation Shutter Mechanics | **CPU** | Central Processing Unit |
| **RSE** | Radiation Shutter Electronics | **VHDL** | Very High-Speed Integrated Circuit Hardware Description Language |
| **DPU** | Data Process Unit | **PWM** | Pulse Width Modulator |
| **PSU** | Power Supply Unit | **RoR** | Remote Memory Access Protocol over Regular Byte stream DAQ Protocol |
| **HDRM** | Hold Down Release Mechanism | **UART** | Universal Asynchronous Receiver-Transmitter |
| **PCB** | Printed Circuit Board | **SPI** | Serial Peripheral Interface |
| **SEL** | Single-Event Latch-up | **VVC** | VHDL Verification Component |
| **CMOS** | Complementary Metal-Oxide-Semiconductor | **GUI** | Graphical User Interface |

# 1 Introduction

## 1.1 Background

The radiation from the sun makes it possible for life to be sustained on earth, but the radiation has immense destructive powers too. In 2012 a solar flare just missed the earth [1]. This flare could have caused a mass blackout. In 1989 a solar storm took out the power transmission system in Quebec, Canada [1]. To be able to understand these and be better able to forecast events, the space weather must be studied. Space weather is phenomena that are the constant interactions between the sun and the magnetosphere of the earth.

ESA has a vision of getting a better understanding of these effects. The SMILE mission is planned to launch as a mission to further explore the full connection between the sun and the earth. The satellite will be placed outside the magnetosphere to observe. There, an SXI will be used to map the magnetosphere and look at the emission from the solar wind. [2]

## 1.2 About this thesis

The objective of this thesis is to continue the development and implementation of a control unit called the radiation shutter electronics (RSE). Lylund and the SMILE team started the development of this project in [3]. The RSE will control the RSM which will enclose the detectors of the SXI instrument when the satellite is within the radiation belt of the earth, as there are high energy particles there which can damage the detectors. As such, the instrument and the RSM will be explored before the requirements of the RSE is explained in Chapter 2. The RSE's primary purpose is to drive the stepper motor in the RSM, so a stepper motor and how to drive it and how power will be saved will be explained. Also, the FPGA that is going to be used will be explained.

The development of the RSE is then focused upon. This thesis moved the project from using a microcontroller to utilise an FPGA. The microcontroller was abandoned in favour of an FPGA because it has not featured in many space missions, which results in a lack of space heritage. The FPGA has been used in previous space missions, so it is proven to work reliably in space.

The fundamental pieces of the RSE are the stepper driver, the register bank which stores all necessary information, and the communication module that lets the central master of the instrument command the RSE. To be able to communicate with the RSE efficiently and reliably, a new small communication protocol was developed to reduce the overhead from the communication standard used previously. The commands that are going to be used to control the RSM need to be well defined and are explained in Section 4.4.

The RSE shall perform different tasks that need to be well defined. Therefore, the different modules have been properly developed to work independently and having a simple hierarchy in order to get a simple and understandable design. The design has been developed with care to ensure that the modules were behaving as expected. The system has also been tested continuously in simulation. The RSE has been developed with a version control tool and is stored on the University of Bergen's git lab repository. Finally, the system has been tested on the bench with an elegant breadboard of the RSE and RSM. A master DPU simulator was modified and used to test by sending signals in accordance with the communication protocol.

## 1.3 Thesis outline

**Chap 2: Requirements for the electronics**

This chapter describes the objective of this work. The SXI instrument and the RSM will be looked at and explained first as these provide the fundament for the RSE. From this, the RSE is described with the radiation considerations, the FPGA that is selected and a close look will be had on the stepper motor as the main purpose is to drive it.

**Chap 3: RSE development**

In this chapter, the choices that were done for the development will be explored. First, why the microcontroller was abandoned for an FPGA will be explained, before a more general outlook on FPGAs and microcontrollers will be had. Then one of the main features of a sequential system, the state machine, will be looked on. The chapter rounds of with a discussion of developments strategies.

**Chap 4: Functionality**

The functionality of the RSE will be defined in this chapter. The core design will first be looked upon before the register that is to be used are defined. Then the procedures and commands will be established before the communication protocol that has been developed is explained.

**Chap 5: Firmware and implementation**

This chapter describes the firmware that has been developed to satisfy the requirements. How the different modules are working and implemented are gone through in detail. First, the top-level design is looked at before the individual modules are described.

**Chap 6: Test and development**

How the development of the system was done is explained first in this chapter. Then the modified DPU simulator is explained, and the modifications are highlighted. The rest of the chapter describes how the tests on board were done, and the results of those tests are shown.

**Chap 7: Summary and conclusion**

In the final chapter, the work is summarised, and the results of the test are discussed.

**Appendix A**

This appendix explains the communication protocol that was used in the previous iteration.

# 2 Requirements for the electronics

## 2.1 Objective

In the SXI Instrument document [4] the SXI's energy band is described to extend up to 5 keV. As seen in Figure 1, the SMILE satellite will orbit the earth and come within the earth's radiation belt. In the radiation belt around the earth, some protons reach above 10 MeV and electrons reaching above 0.5 MeV [5]. The SXI instrument must be protected against these particles when the satellite is within reach of the radiation belt of the earth, as radiation there can degrade or destroy the instrument. The satellite might have to perform manoeuvres to calibrate different instruments, and the SXI needs to be protected against stray particles. To protect the instrument from stray particles and high energy radiation, a Radiation Shutter is required.

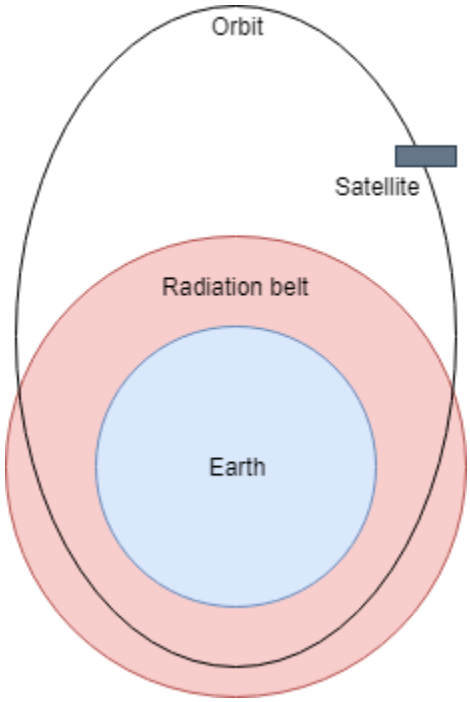


*Figure 1 SMILE satellite's orbit illustration*

An internal team at the University of Bergen called the SMILE team develops the mechanical part of the Radiation Shutter. The main objective of this work is to create an electronic system to be able to operate the RSM from commands.

## 2.2 Soft X-ray Imager Instrument

On ESA's webpage about the instruments belonging to the SMILE mission, they describe the SXI like this:

> "The SXI is a wide-field lobster-eye telescope using micropore optics to spectrally map the location, shape, and motion of Earth's magnetospheric boundaries, including the bow shock, magnetopause, and cusps, by observing emission from the solar wind charge exchange (SWCX) process. The SXI is equipped with two large X-ray-sensitive CCD [Charge-Coupled Device] detectors covering the 0.2 keV to 2.5 keV energy band, and has an optic field of view spanning 15.5° × 26.5°." [6]

According to [4] the scientific objective of the SXI is to image the X-ray emission produced when solar wind ions interact with neutral atoms in the exosphere of the Earth. These X-ray lines have intensities that peak in the cusps and magnetosheath. The flow of solar wind and energy into the magnetosphere can be imaged from the density boundaries of the X-ray emissions. The SXI telescope combines imaging with spectroscopy to obtain information on the composition of the solar wind that generates the solar wind charge exchange X-ray emission. This way changes in the solar wind reaching the magnetosphere can be detected. [4] The SXI is illustrated in Figure 2.

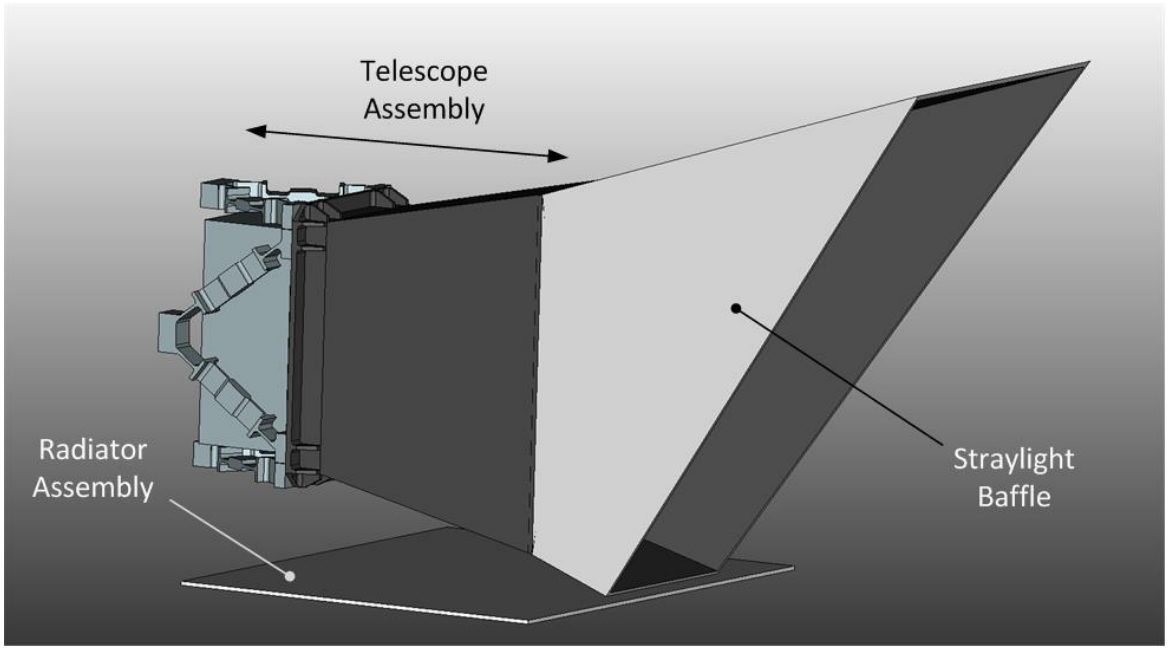


*Figure 2 3D rendering of the SXI instrument. One of two possible configurations [4]*

The SXI instrument has four units in addition to the x-ray imager instrument itself. These are explained in the instrument interface control document [4] as:

- Straylight baffle: Avoid light into the system from the Sun or the Earth.
- Telescope Assembly: Optical telescope structure
- Radiation Shutter: Shutter mechanism
- Focal Plane Assembly: CCD detectors and front end electronics

### 2.2.1 Electronics box

In addition to these components, there is a separate chassis containing all the backend electronics. In there is the DPU, the power supply unit (PSU) and the radiation shutter control electronics (RSE).

## 2.3 Radiation Shutter Mechanics

This section is mainly based on the RSM design report [7] from the SMILE team. The RSM is the shutter itself, as shown in Figure 3. The RSM frame is mounted in the telescope assembly. The RSM Rotary Actuator on the top of the figure in rotates the RSM Door Leaf in with the bearings on each side of the Rotary Actuator. The RSM Launch Lock is a pin puller that keeps the Door Leaf in place during launch. Besides, there is the RSM End Switches, the RSM End Switch Trigger and the RSM End Stops.

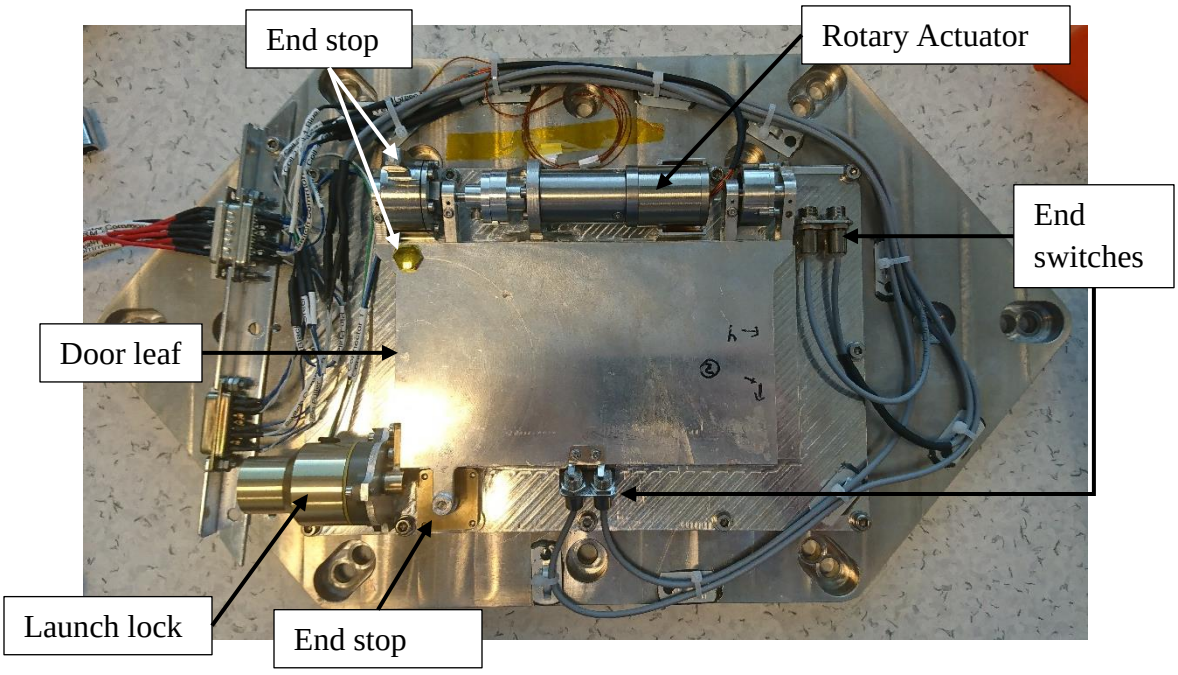


*Figure 3 RSM prototype*

The main components of the RSM are made from aluminium. Where the door leaf is in temporary physical contact with other elements like the end stops are made from INERMET (Heavy tungsten alloy) and hardened stainless steel. These places where there is contact are prone to cold welding.

When a metal surface impacts another metal surface, there is a natural oxide level on the surface that is impacted. This layer would naturally re-oxide on earth. In space, this layer is broken irreversibly. Breaking this layer will create a pure metal to metal contact, and this enables welding. This layer is either degraded over time from impacts or broken from vibrations might lead to oscillating movements that also enables welding, called fretting. These processes are called cold welding [8], and may happen on the RSM during launch as the door leaf might vibrate heavily.

### 2.3.1 Rotary actuator

The rotary actuator is the motor that is used to rotate the door leaf. It consists of a phySPACE stepper motor and a gearbox that comes preassembled from the manufacturer Phytron. Some parameters of the phySPACE stepper motor are listed in Table 1. The stepper motor allows the use of a relatively simple control system to drive it. How a stepper motor works are explained in 2.4.3.

The gearbox has a small diameter and gives a high torque/mass ratio. A drawback of the high ratio is that the gearbox has many mechanical components.

*Table 1 phySPACE stepper motor parameters [7].*

| **Parameter** | **Value** | **Note** |
|---|---|---|
| VSS stepper motor | 200 | full steps per revolution |
| Pole pairs | 50 | |
| Half-step mode, phases | 8 | |
| Electrical half-steps | 400 | |
| RS Opening angle, ° | 100 | |
| Gearbox ratio | 192 | |
| Half-step speed, $s^{-1}$ | 500 | |
| Opening time, s | 44 | |
| Motor speed, RPM | 75 | or $75/60 = 1.25\ s^{-1}$ |

By knowing the key parameters of the stepper motor and the mechanism, we can determine the number of half steps per second and the rotational speed. With a half step speed of 500 steps per second, we obtain an opening time of around 41 seconds at a rotational speed of 75 rounds per minute. This is well within the recommended maximum speed of 100 rounds per minute for dry lubricated motors. A close-up picture of the rotary actuator is shown in Figure 4.

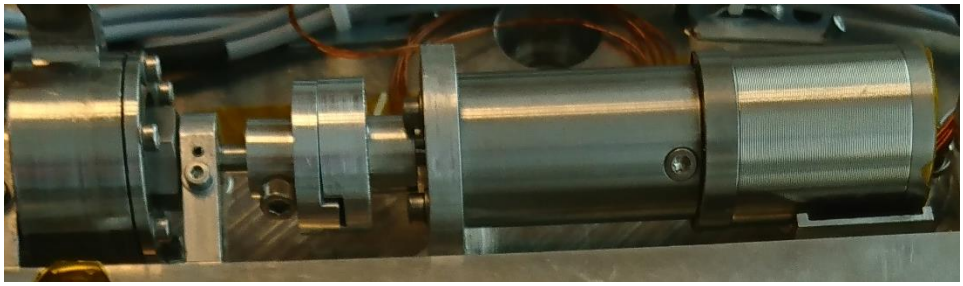

*Figure 4 Close-up of the rotary actuator with bearing*

### 2.3.2 Hold Down Release Mechanism

The purpose of the hold down release mechanism (HDRM), on the RSM Launch Lock, is to hold the door leaf secure in place during launch and then release it before the instrument starts operating. The door leaf will be at risk of moving around under launch if it is not held in place. The HDRM pin is inserted before launch. In the locked position the leaf will be in contact with the end stop. This contact might lead to cold welding. To reduce this risk, a bronze hub is used for the leaf/pin interface. Should the end stop and leaf fuse, the rotary actuator must be driven at max torque to try to force the shutter open. A close up picture of the HDRM is shown in Figure 5.

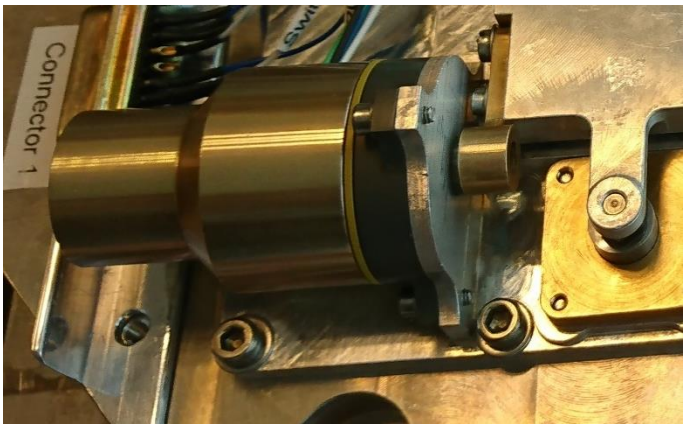


*Figure 5 Close-up of Hold Down Release Mechanism*

### 2.3.3 End Switch

End switches are used to signal that the leaf has reached the end position. The two redundant systems each have its pair of end switches for the two end states. Upon contact with the leaf, the switches will be activated. There is some slack in the switches after they activated to allow for a gap between the nominal end position and the end stops.

### 2.3.4 End Stops

In case the end switches should stop working, the rotation of the shutter needs to be stopped, and the end stops mechanically stops the leaf from rotating. The end stop also uses a spring to preload the leaf against the stop during launch.

## 2.4 Radiation Shutter Electronics

The purpose of the RSE is to control the RSM described in detail in [7], which includes:

- Operation of RSM Rotary Actuator (a stepper motor with gearbox)
- H-bridge motor driver circuit
- Sensor read-out for the temperature sensor
- Sensor read-out for the RSM Sensors (“Shutter open” and “Shutter closed”-switches)
- A communication link between the DPU and the RSE

The RSE needs to be an independent system. To accommodate its purposes the necessary circuits and an FPGA is placed on a printed circuit board (PCB), where the FPGA can control the other circuits. A system overview produced by the SMILE team is shown in Figure 6, where the motor control and the motor driver are central to the RSE. In the earlier stages of the project, the DPU through the RSE was supposed to control the HDRM.

The Radiation Shutter is designed to be fully redundant. I.e., the stepper motor (RSM Rotary Actuator) has a double set of windings, and a double set of temperature sensors, enabling independent control from two individual electronics boards. All feedback switches and actuators, as well as communication channels, are also redundant.

For the RSE in the electronics box, this means that there are two independent PCBs connected to their redundant PSUs and DPUs. There are no cross connections between the dual redundant systems, and only one redundant system will be powered at the time.

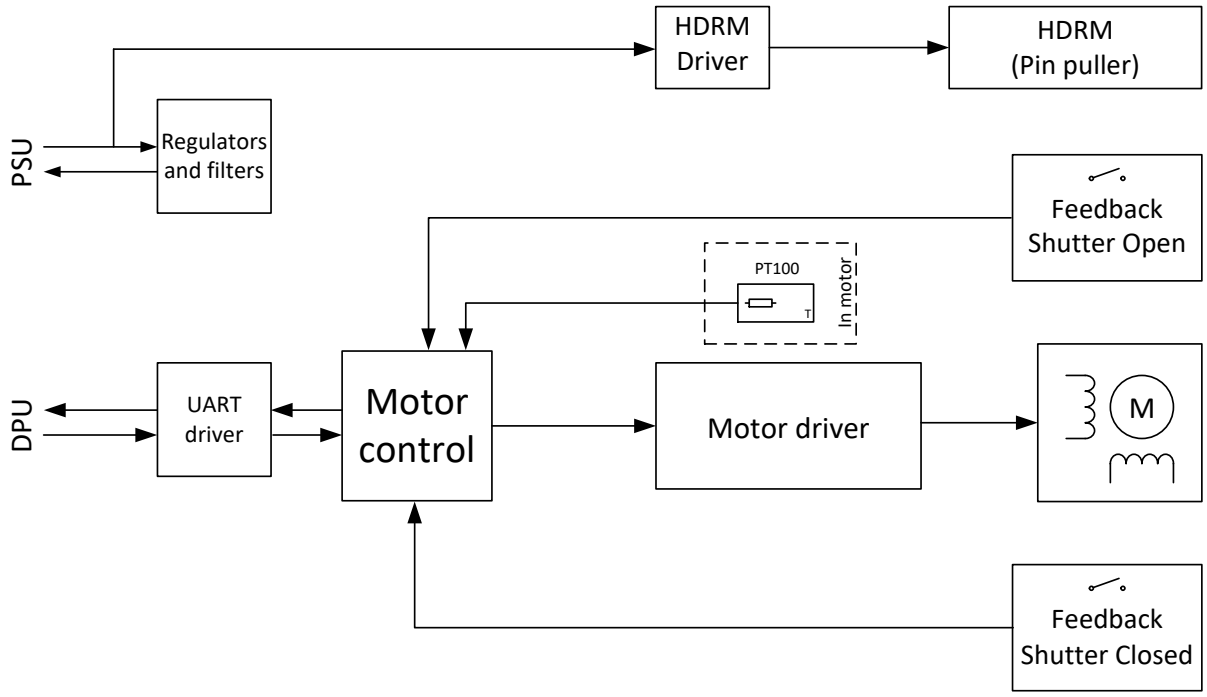


*Figure 6 RSE overview*

### 2.4.1 Radiation considerations

Radiation effects are divided into two main effects. The first effect is Single Event Effects, that is a stochastic effect that can happen anytime. As the name implies, these effects happen in a single event as either a non-destructive effect such as corrupting data or as a destructive effect where the corruption permanently damages or destroys the circuit. An example of a non-destructive effect is single event upsets, where a radiation effect occurs in a memory node. This stored data will be in an erroneous state but can be overwritten with new valid data. A destructive effect is a single-event latch-up (SEL) where an ion-generated charge triggers the bulk of complementary metal-oxide-semiconductor (CMOS) technology to produce a low-impedance path between ground and power. This path can create a feedback loop that maintains a high current through the path [9].

The other effect is the total-ionising dose (TID). This is the energy that is absorbed by the technology per mass when the technology is exposed to ionising radiation. This is measured in rad. This effect is an accumulated effect as that degrades the performance and potentially the functionality of material like insulators that are common in CMOS technology [9].

The RSE needs to be protected and able to withstand these effects as it will break otherwise. Therefore all SMILE electronics shall be immune to destructive SEL and protected against other Single Event Effect. All electronic components must also be able to withstand a TID of 60 krad [10]. In case critical components are marginally tolerant to the applicable dose, spot shielding will be applied.

### 2.4.2 FPGA

The FPGA that is going to be used needs to be able to meet the requirements for radiation tolerance. In addition, it needs to be reliable and not draw too much power. The choice of using an FPGA is explained in Section 3.1.3.

An anti-fuse FPGA from Microsemi will be used for the main motor controller, either RTAX250 or RTAX1000, depending on design needs. NanoXplore NG-MEDIUM (NX1H35S) FPGA is kept as an option, but this is a brand-new component, so little or no space heritage exists for it.

#### Microsemi RTAX250

The RTAX family is the second generation of Microsemi's products for space applications. The RTAX250 FPGA has registers that are hardened for single event upsets. Being hardened, the registers are immune against single event upsets at a linear energy transfer of less than 37 MeV/cm$^2$/mg. The FPGA will survive a TID up to 300 krad, which is above the demand of 20 krad. It is also immune against SEL with a linear energy transfer up to 117 MeV/cm$^2$/mg. The RTAX series also comes with a low power option, which saves up to 80% of static current compared to the standard versions in the worst-case scenario [11].

### 2.4.3 Stepper Motor

A stepper motor is an electrical motor that moves a single step when the magnetic field changes as a consequence of switching the direction of the current flow in the field coils. There are two principal types of stepper motors, as illustrated in Figure 7. The bipolar stepper motor, which has one coil per phase and needs two switches for each phase. The unipolar stepper motor, which has one coil for each phase and one switch for each phase. The current flow in the coil is reversed by flipping the switches [12].

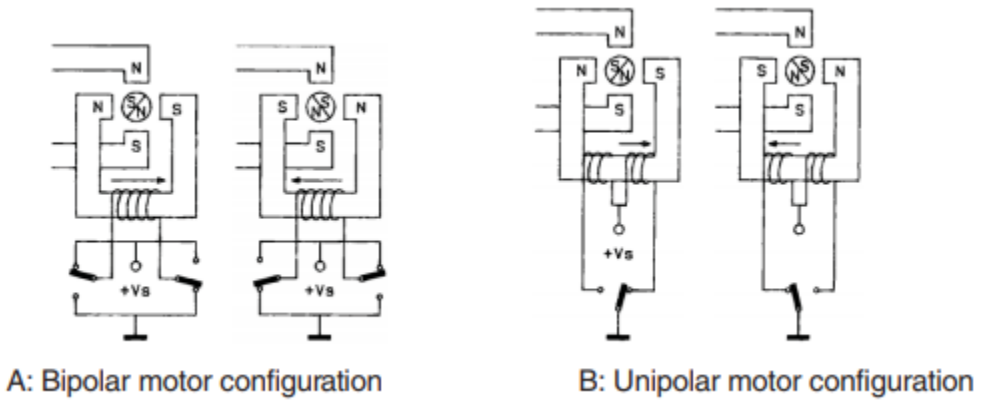


*Figure 7 Stepper motor configuration [12].*

Our motor uses a bipolar configuration, and the advantage of the bipolar configuration is that there is only one coil with low winding resistance. The unipolar has a double winding with a higher winding resistance because of the thinner wire that is required. The advantage of the unipolar configuration is that is can have a simpler driver circuit for switching, whereas the bipolar requires a more complex driver circuit [12]. As previously discussed, the stepper motor used in the RSM is a phySPACE stepper motor. This stepper motor is bipolar and needs a driver circuit that fits it. By switching the direction of the current flow through the two coil pairs, the stepper motor is driven.

### Stepper driver circuit

A common way to operate the two coil pairs is by using an individual H-bridge for each coil, as shown in Figure 8. This circuits main components are the four transistors and the coil forming an H-shape in the middle. The transistors have their own regulatory circuits that set the voltage on them to turn them on and off.

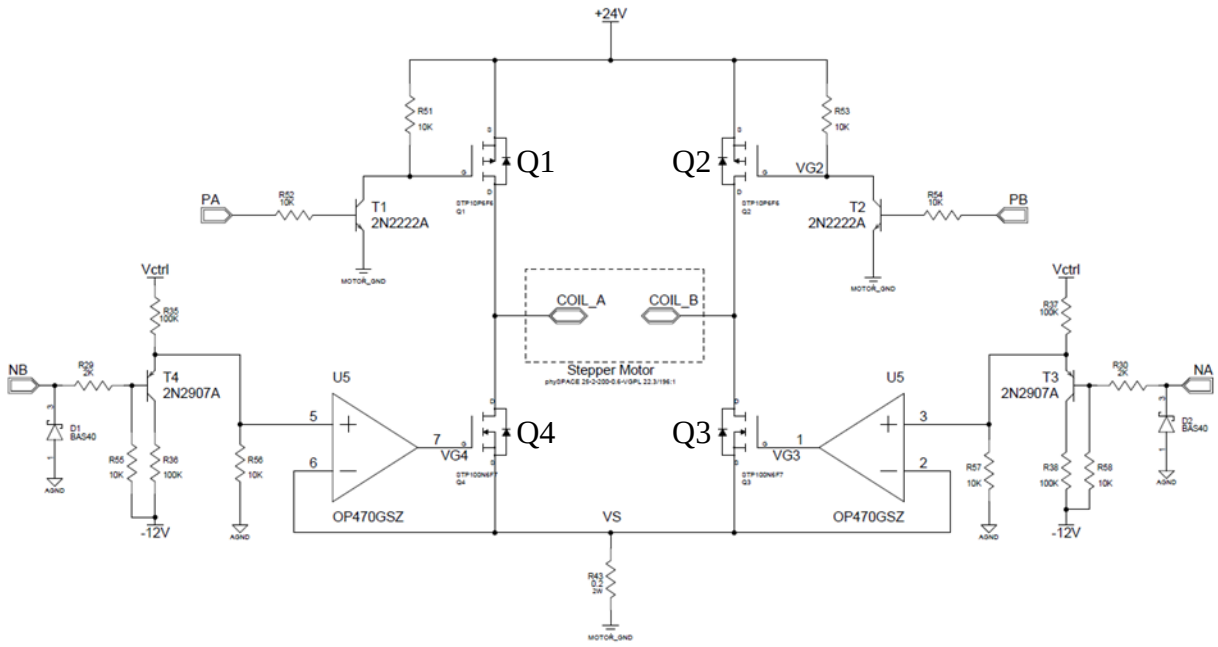


*Figure 8 Stepper motor driver circuit H-bridge*

The current through the transistors will excite the coils which in turn steer the stepper motor. Current will flow through the transistor pair Q1 and Q3, or Q2 and Q4. Q3 and Q4 are also forming constant current sources, enabling a controlled current flow in the inductors. The controlled current is set by the $V_{ctrl}$ pulse, which is derived from a filtered pulse width modulator (PWM) source, connected to the FPGA. The current through the power transistors results in a voltage drop over $R_{sense}$, which is fed back via the operational amplifiers to control the power transistors. The terminals labelled PA, PB and NA, and NB is connected to the FPGA, again one set for each coil.

## Step sequence

A change in the current through the coils will produce a step in the motor. With one pole pair to move the stepper, there are four full steps in each electrical cycle. Typically for stepper motors, each step will move 1.8° or 7.5°. With full steps, there will always be current through both the coils. Instead, the current can be turned off before switching to the other side as an intermediate step. This method is called half-step and gives eight half-steps instead of four full steps, as seen in Figure 9 [12].

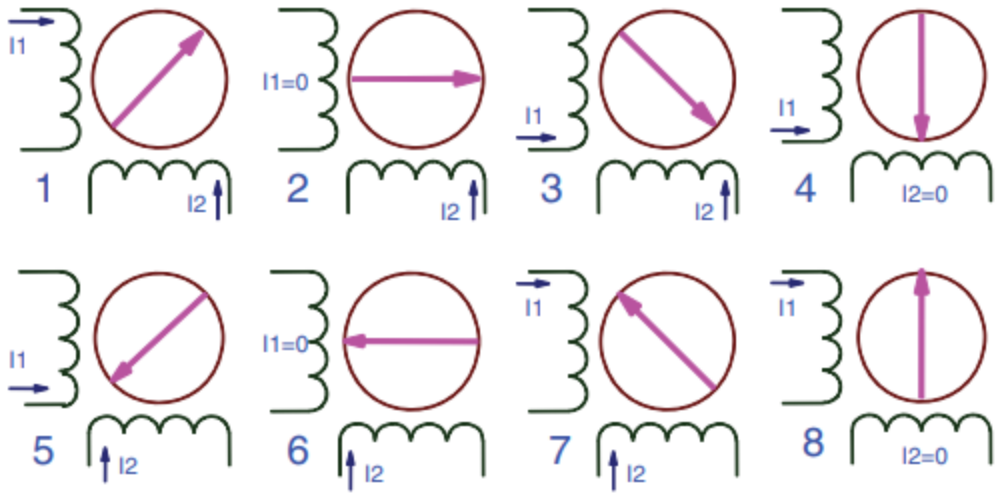


*Figure 9 Half-step sequence for a two-phase bipolar motor [12]*

Using half steps effectively doubles the resolution of the stepper motor but comes at the cost of getting only about 70% torque [12]. The torque margin is more than large enough for us, and we can get a smoother operation with less mechanical stress by using half-step mode.

As shown in Figure 9, the current in one coil is sustained in one direction for three out of eight phases before turning off, and changing direction. Each motor-on sequence is always started by a programmable length settling time, in which transistor Q3 or Q4 is fully on, and the current limiter circuit controls the full current as described earlier. After that, current chopping is enabled, where transistor Q3 or Q4 is turned off and on again at regular programmable intervals, resulting in a current decay determined by the motor coil inductance, before the appropriate transistor is turned on again. By turning the inductor current on and of faster than the inductor manages to change the current significantly, we can hold the current at a nearly constant level. The current level will fall and be raised again to the current plateau where the current is at the set motor current level, before being allowed to fall again. This chopping of the motor current results in a significant reduction in average power as the power source will deliver less current in total.

As the settling time only is needed to get the motor current up to the desired current plateau where the current will stay if the corresponding transistor is on, it will have little effect on the torque the motor produces. The chopping is used to save current while simultaneously holding the inductor current near constant. As long as the inductor current stays at the current plateau and do not fall towards zero, the torque should be constant. The torque will have a breakpoint where the chopping

causes a significant drop in the inductor current. As the current then will go towards 0, there will not be any power to create the torque. As torque will be a function of the inductor current, the main component of increasing the torque will then be the motor current as this sets the inductor current.

To be able to operate the motor with half steps, the transistors in the driver circuits needs to be turned on in the correct order, as shown in Figure 10. The driver needs to go through eight stages, corresponding to the eight half-steps. In order to reverse the motor direction, we need to run the half step sequence in the opposite direction by switching the A and B coil's driving sequence. As eight steps are needed for each direction, a state machine will need at least 16 states to drive the stepper motor.

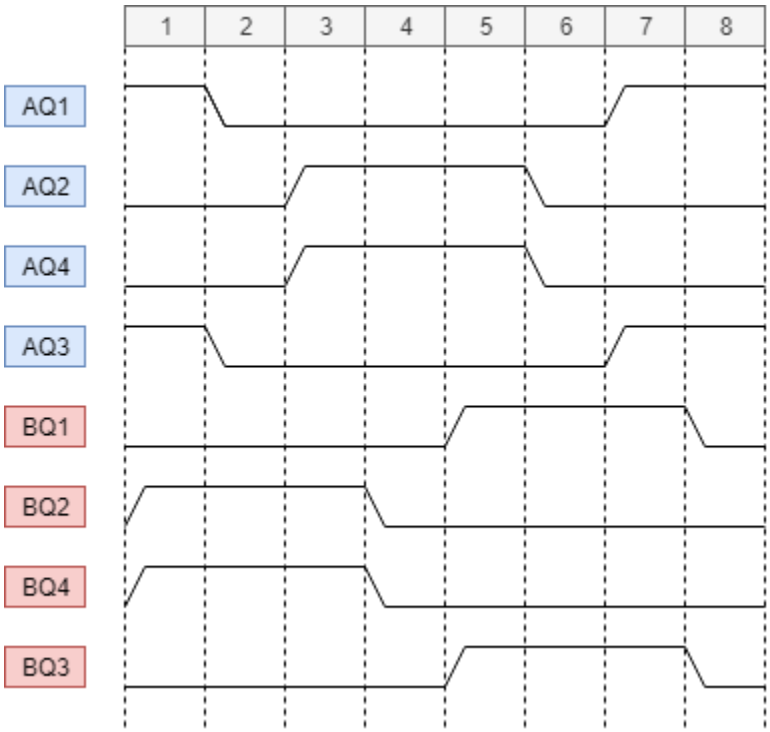


*Figure 10 H-bridge electrical cycle states for coil A and coil B for 8 half step phases*

# 3 RSE Development

## 3.1 Software vs Hardware

In the first iteration of the RSE, a stepper driver breadboard was developed with a driver software placed on a microcontroller to drive a stepper motor. In this iteration, an elegant breadboard (EBB) was designed by the SMILE team, that is closer to the real version but still using commercial components. When this iteration of the RSE is done, an Engineering Qualification Model can be created, where the identical components to the ones used in flight will be used to test the functionality and the fit. At last, a Pre-Flight Model is made with the components that are going to used, and when it has passed the tests, it is upgraded to the Flight Model, which will be launched.

The RSE needs a firmware to control the stepper driver circuits as described in Section 2.3. There is no official definition of firmware, but in general, the term firmware is used about a code that is placed onto a device to control the low-level functions of the device. Typically, this is a software code that uses a microcontroller to execute the code. Another way of looking at this is to say that a hardware description language is also a firmware as it is synthesised and place onto an FPGA. At the core, it is, however, hardware instead of software.

### 3.1.1 FPGA

FPGA is an integrated circuit that consists of configurable logic blocks. These logic blocks are built up by using two different methods. Our FPGA uses anti-fuses to program the FPGA. The use of anti-fuses makes the FPGA one-time programmable. Other FPGAs can use static Random-Access Memory (RAM) or flash memory to hold the configuration of the logic blocks. A simplified FPGA floorplan is illustrated in Figure 11.

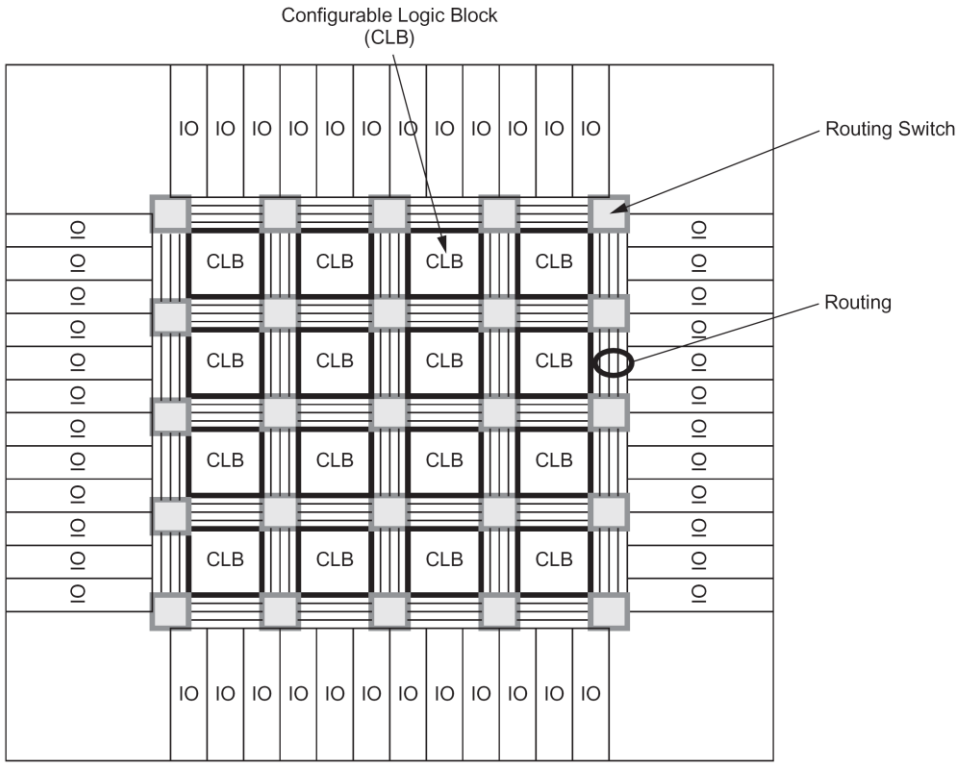


*Figure 11 Simplified FPGA floorplan [13, p. 630]*

Vendors sell the FPGA without a configuration, and the customer can then program the FPGA as desired. In our project, a reconfigurable FPGA is used to test the firmware. One-time programmable FPGA is often used in space projects, as they have proven to be reliable in a radiation environment [14].

### 3.1.2 Embedded system

"An embedded system is a microprocessor based system that is built to control a function or a range of functions", see for example [15]. The embedded system is a part of a larger system where its job is to control a function. One such microprocessor-based system is a microcontroller. "A microcontroller (sometimes called an MCU or Microcontroller Unit) is a single Integrated Circuit (IC) that is typically used for a specific application and designed to implement certain tasks" [16].

A microcontroller consists of multiple components as a Central Processing Unit (CPU), Integrated memory such as RAM and peripheral interfaces such as I/O ports. The CPU is the brain of the system that controls everything that happens.

Microprocessors come in different bit sizes, which indicate how wide the data bus is. As such an 8-bit microcontroller will utilise an 8-bit data bus. The bit size will limit the register widths, address bus and that every single instruction has a set range to work within. An 8-bit microcontroller can only work on 8 bits at a time, while a 32-bit microcontroller can work with 32 bits. This makes the smaller microcontroller useful if we want to save power. Whereas the larger microcontroller can use a more significant number of bits in its calculations, to get a more accurate result, or a number higher than the smaller microcontroller can handle. Being more precise comes with increased size, power usage, memory and price of the microcontroller.

The previous RSE design used an ATmega128 microcontroller, which is a low-power AVR 8-bit microcontroller. Microchip has a space version of the ATmega128 microcontroller called ATmegaS128. It is the same chip as the ATmega128 but has improved radiation toleration, and its main features are shown in Table 2. The original thought was to make a prototype on the ATmega128 and then transfer the system over to an ATmegaS128.

*Table 2 Features of ATmegaS128 [17]*

| **Features** | **ATmegaS128** |
|---|---|
| **Flash (KB)** | 128 |
| **SRAM (KB)** | 4 |
| **EEPROM (KB)** | 4 |
| **External Memory (KB)** | 64 |
| **General Purpose I/O pins** | 53 |
| **SPI** | 1 |
| **USART** | 2 |
| **ADC** | 10-bit, up to 76.9ksps (15ksps at max resolution) |
| **ADC channels** | 8 |
| **8-bit Timer/Counters** | 2 |
| **16-bit Timer/Counters** | 2 |
| **PWM channels** | 6 |
| **Operating voltage** | 3.0-3.6V |
| **Max operating frequency** | 8 MHz |
| **Temperature range** | -55°C to 125°C |

The ATmegaS128 has "[n]o Single Event Latch-up below a LET threshold of 62.5 MeV/mg/cm2@125°C[, and is] tested up to a Total Ionizing Dose of 30 krads(Si) according to MIL-STD-883 Method 1019" [17]. Microchip also states that the microprocessor "has been developed and manufactured according to the most stringent requirements of MIL-PRF-38535 International Standards and Aerospace AEQA0239 specification" [17]. MIL-PRF-38535 is the US military's performance specification that "establishes the general performance requirements for integrated circuits or microcircuits and the quality and reliability assurance requirements, which are to be met for their acquisition" [18].

### 3.1.3 FPGA or CPU

Since this project will go onto a satellite that will be launched into space, we need a technology that is approved for usage in space. The previous RSE utilised a microcontroller as described in Lylund [3]. An appropriate microcontroller for the mission was found, the ATmegaS128. The main problem with the ATmegaS128 for us is that it does not have space heritage. Space heritage means that none or few space missions have used this microcontroller. Having used a particular technology multiple times proves that it performs reliably in space. Besides, ESA has a rigorous set of test procedures to test software and technology that are lacking space heritage. This would have meant that we would have to prove that the microcontroller would work.

Some alternative microcontrollers with increased bit sizes were considered, but those would also come at an increased cost and system complexity. An external memory might also have been necessary. There are, however, multiple FPGAs with space heritage that we could then use instead. As these FPGAs have space heritage and hardware would be used instead of software, we would have an easier time to get the design approved, as hardware are easier to get accepted by ESA's test procedures. In this project, the lack of space heritage and the easier test procedures made us abandoned the microcontroller, and switch to an FPGA.

In addition to the reasons above we wanted to take a more general view on an FPGA vs an embedded system. In embedded systems, the microcontroller reacts to stimuli on the different inputs, and from the incoming data and stored information, the system creates the desired output. The FPGA on the other side has dedicated logic to respond to the stimuli coming into the system. So, while the microcontroller would have to let the incoming data got through the CPU to create an output, the FPGA creates a logic block directly between the input and the output to speed up the response scientifically. To create this configuration, a hardware description language is required. VHDL (Very High-Speed Integrated Circuit Hardware Description Language) is one such language. A general comparison of a CPU and an FPGA is shown in Table 3.

*Table 3 Comparison of CPU and FPGA adopted from [19]*

| | **CPU** | **FPGA** |
|---|---|---|
| **Overview** | Traditional sequential processor for general purpose applications | Flexible collection of logic elements and IP blocks that can be configured |
| **Processing** | Single- and multi-core MCUs and MPUs, plus specialized blocks: FPU, etc. | Configured for application; SoCs include hard or soft IP cores (e.g., Arm) |
| **Programming** | OSes, APIs run a huge range of high-level languages; assembly language | Traditionally HDL (Verilog, VHDL) |
| **Peripherals** | Wide choice of analogue and digital peripherals in MCUs; MPUs include digital bus interfaces | SoCs may include many transceiver blocks, configurable I/O banks |
| **Strengths** | Versatility, multitasking, ease of programming | Configurable for a specific application; configuration can be changed after installation; high performance per watt; accommodates massively parallel operation; wide choice of features: DSPs, CPUs |
| **Weaknesses** | OS capability adds high overhead; optimized for sequential processing with limited parallelism | Relatively difficult to program; long development time; difficult for floating-point operations |

## 3.2 State machine

One of the core components of a sequential digital system is a state machine. A finite state machine is a sequential system that consists of combinational logic and a state register, as illustrated in Figure 12. The outputs of the FSM are a function of the current inputs and past inputs. The state gives information about previous data. The present state stored in the state register is the culmination of everything that has happened thus far, see for example [20].

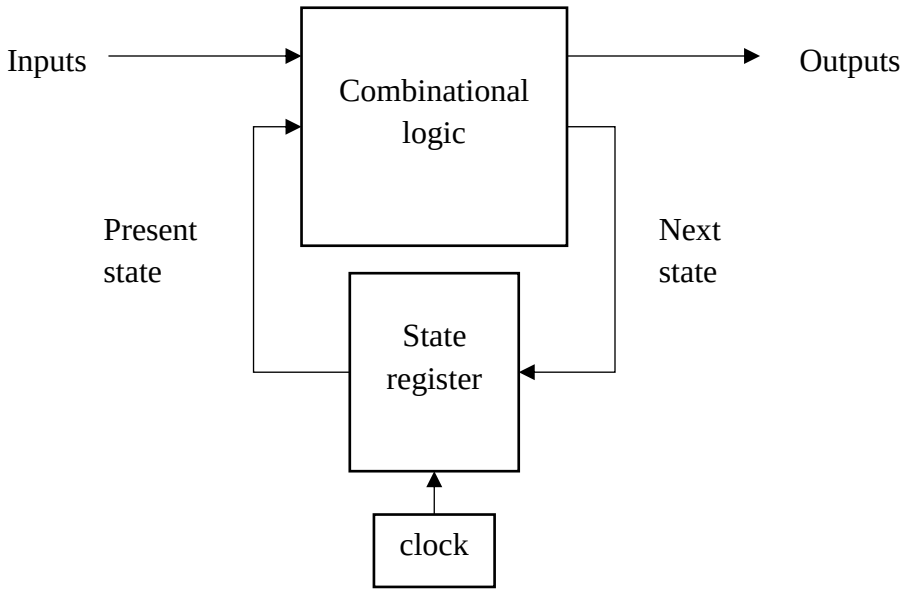


*Figure 12 Synchronous finite state machine block diagram [20]*

The state machine can be asynchronous or synchronous. So, either the state and the outputs only update on the triggering clock edge, or it updates as soon as one input changes value.

## 3.3 VHDL development strategy

To be able to create a design and implement it, we need a plan so we can prioritise and stage the development. There are a lot of different strategies for developing a firmware system. Various approaches were considered to be able to develop the firmware efficiently.

### 3.3.1 Waterfall development method

A common technique to use when developing a project is the so-called waterfall design strategy. The waterfall strategy bases itself upon that the requirements of the design need to be figured out before anything can be done. From these requirements, a design can be proposed. If the design is accepted, it can then be implemented.

At last, the implementation needs to be verified. No steep can be done before the completion of the previous. By going through the development step by step and locking down the different aspects before moving onto the next task, one can achieve good flow.

In the article “Understanding the pros and cons of the Waterfall Model of software development” Melonfire [21] comes with this explanation of the waterfall model:

> “Essentially, it's a framework for software development in which development proceeds sequentially through a series of phases, starting with system requirements analysis and leading up to product release and maintenance. Feedback loops exist between each phase, so that as new information is uncovered or problems are discovered, it is possible to "go back" a phase and make [an] appropriate modification. Progress "flows" from one stage to the next, much like the waterfall that gives the model its name.”

Tutorialspoint [22] says that the “[w]aterfall approach was first SDLC [Systems Development Life Cycle] Model to be used widely in Software Engineering to ensure [the] success of the project. (…) In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.” A typical waterfall is shown in Figure 13.

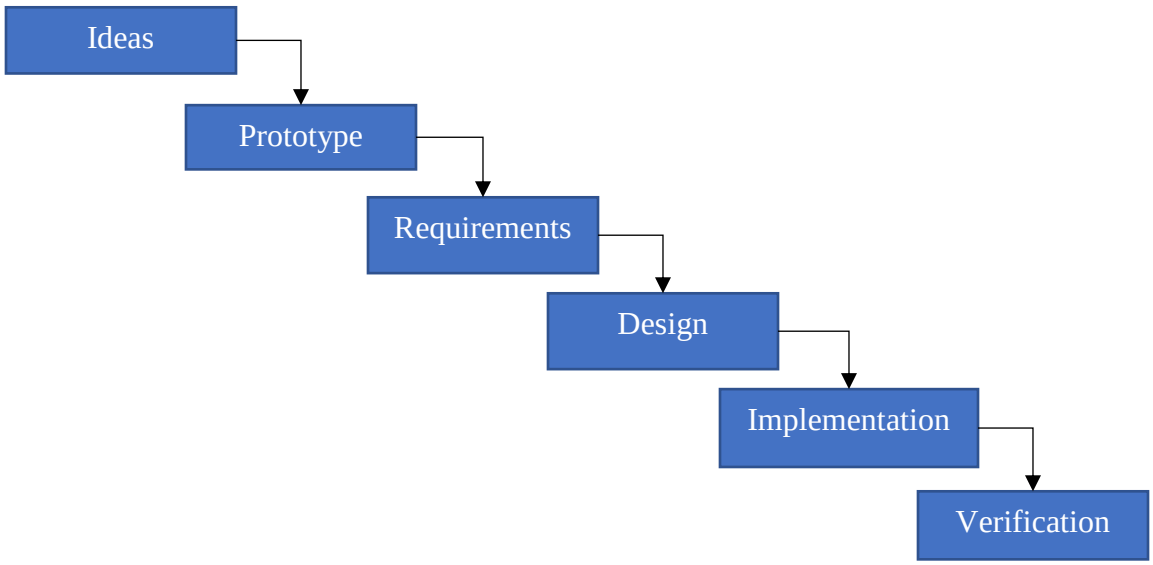


*Figure 13: Waterfall model*

There is an ideas phase where people pitch the goals of design formulated and different designs, and from that, the development of a prototype is possible. These two phases are often presumed to have already happened in the waterfall model. An idea about what the system is and how it should function must be given in advance. From these ideas, a prototype can be developed to see if the proposals are possible and find out what requirements are needed. If there is not a lot of external

demands on the system, it can be hard to start with coming up with the requirements list. In a large software project, it is often useful to create a smaller version in a more straightforward language like python at first to get a feel for the solution. Such a prototype lets the developers have a look at the system and figure out what precisely the customer desires. Often it is hard to know precisely what is desired at an early stage.

A modification upon the waterfall strategy is to split the design up into different modules and create localised "waterfalls" where we go through the four aspects for a single module which exist in a bigger system which is the main "waterfall".

The main problem with the waterfall strategy is that requirements might change during the development. Such changes would be difficult to implement since one has to go back and change the previously done work.

### 3.3.2 Lean development method

Another strategy is the lean software development method. According to Poppendieck and Cusumano [23], lean is more about a set of principles that are applied to development, and not a distinct practice.

Poppendieck and Cusumano [23] go on to explain the seven principles that lean is based upon. “Optimise the whole” is about seeing the software as a part of a larger system. Only then can the developer understand what the customer needs and want. “Eliminating waste” is the need to remove anything that does not contribute value to the customer or gives more or better knowledge about how to deliver value more effectively. “Build quality in” is also called “top-down-programming”. This method is about how smaller modules are continuously integrated into larger systems. “Learn constantly” is about how development is about developing knowledge and then placing that knowledge into a product. There are two ways to go about this. Either to learn first, or continuously learning throughout the development.

“Deliver fast” is how there are multiple in-house releases or releases to the customer. These releases are designed, developed and delivered repeatedly with small changes. “Engaging everyone” means that the software is only a part of a bigger system. There is different knowledge around in different departments and value might be lost if no one sees the system as a whole.

Decisions should be taken by the people who have the power and the knowledge about what they are deciding. “Keep getting better” is to realise that the known specific practices are often not the best for the current problem and that the system needs to adapt and improve over time [23]. How a development will flow is shown in Figure 14.

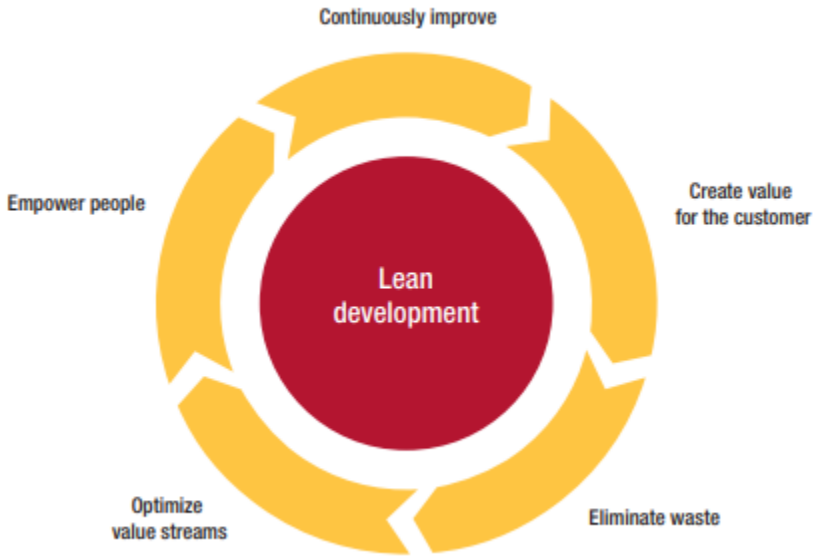


*Figure 14 Lean development [24, p. 23]*

### 3.3.3 Choosing a strategy

The Bergen SMILE team has put forth the ideas behind the system. From these ideas, Lylund has developed a software prototype to be used on a breadboard. They figured out most of the requirements during the creation of the prototype. This prototype was working and could run a stepper motor according to the current specifications at the time.

Since the SMILE project and the requirements for the RSE are changing during the development, the lean manufacturing strategy was selected. This project is the second iteration of the whole RSE project. Lylund and the SMILE team pitched the ideas and developed a prototype to work with a breadboard in the first iteration. The lean strategy bases itself upon reducing waste and focus on the value for the customer. In our case, this means that we focus on the necessary parts first and foremost. We will make a minimum viable product which can be accepted, and then we add functionality to it by continually adding changes and small new parts. This way, each step could

be designed, implemented, tested and accepted. This method let the requirements be nailed down for the barebone design. With a barebone design, we can expand upon it. While developing the system, we will learn about new situations and problems that we need additional requirements to solve.

# 4 Functionality

## 4.1 Design - VHDL

In the conclusion of Lylund [3], a potential FPGA design based upon his software design was proposed by the SMILE team, as shown in Figure 15. This design was our fundament. There are three primary tasks the design must accomplish. We need a method for communicating with the DPU, some way of storing essential information and a method for driving the motor. The components that make these three tasks possible is our barebone design. From that design, more functionality was added to protect the RSE.

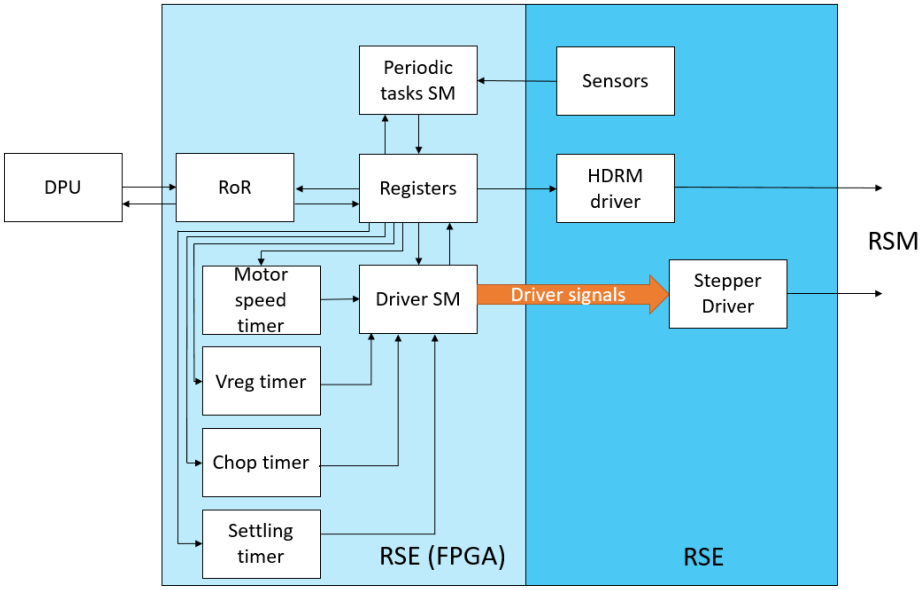


*Figure 15 FPGA design proposed by the SMILE team [3]*

## 4.2 Registers

If the DPU is going to be able to retrieve or send any data to the RSE, the RSE needs some registers to store information. The registers are composed of multiple flip flops to store a logic vector. With various registers, they can be defined in order with addresses. Individual bits can also be specified. Via specified registers with set address and given content, the DPU can reliably access the RSE to read or write data. Typically, on the hardware side, these registers are situated in the register bank and can be probed from other modules.

### 4.2.1 Status registers

The DPU needs to know the status of the RSE and the RSM in order to be able to perform operations. The status registers will always be readable, but not writeable for the DPU, as the RSE maintains them. Most of these registers will come directly from the corresponding modules and therefore, be updated immediately. The status registers are shown in Table 4.

*Table 4 Status registers*

| **Register** | **Address** | **Bits** | **Purpose** |
|---|---|---|---|
| Firmware Version | 0x00 | 0-7 | FPGA firmware revision number |
| Motor Temperature | 0x01 | 0-7 | Monitor motor temperature |
| Electronics Temperature | 0x02 | 0-7 | Monitor electronics/heat-sink temperature |
| Shutter status | 0x03 | 0 | Shutter is closed |
| | | 1 | Shutter is open |
| | | 2 | Shutter closure in progress |
| | | 3 | Shutter opening in progress |
| | | 4 | Shutter emergency closure initiated |
| | | 5 | Motor too hot |
| | | 6 | Electronics too hot |
| HDRM status | 0x04 | 0 | HDRM is armed |
| | | 1 | HDRM is activated (only valid for 1 sec) |
| Performed steps L | 0x05 | 0-7 | Number of steps performed for ongoing operation (LSB) |
| Performed steps H | 0x06 | 0-7 | Number of steps performed for ongoing operation (MSB) |
| Heartbeat count | 0x07 | 0-7 | Incremented for every heart-beat register request |
| Processor status | 0x08 | 0 | Heartbeat missing |
| | | 1 | Reset armed |

### Firmware version

The firmware version register contains the current firmware revision number so that any revisional changes can be detected. The set number also gives the DPU something stable to read during testing as the value is not changed. During testing on board, we can read this register, and since we know what to expect, we can detect any errors in the communication.

### Motor temperature

This register is the current sampled motor temperature. The motor temperature will be regularly updated as long as the ±12V rails are enabled. The heat will be measured with a periodic readout of the thermistor voltage. The motor temperature will increase if a motor operation is in progress. Rest of the time, nothing should be happening, so there should be no generation of heat. So, most of the time, the temperature should be following structural temperature.

### Electronics temperature

The electronics temperature register is the current sampled temperature of the electronics driving the motor. The temperature is read together with motor temperature in the same procedure. Like the motor, the driver electronics temperature should increase when the motor is running, and other than that, the temperature should be stable.

### Shutter status

To keep track of the shutter operation and position, we need a register to keep up with the state of the shutter. The definition of every single bit in the shutter status register is shown in Table 4. We need to know if the shutter is open or closed. The easiest way of knowing this is to see if the corresponding end switch is engaged. Should the end switches stop working, the last run command needs to be remembered by the DPU. If an operation is running, no other operation can be accepted, so the current running operation must be able to be read by the DPU. At last, the DPU needs to know if the motor or electronics temperature is exceeding safe limits.

### HDRM status

This functionality will not be used beyond the EBB. The HDRM status register indicates the status of the HDRM, as described in Table 4. There are only flags indicating if HDRM has been armed or activated, as no sensors are measuring the HDRM. The HDRM must be armed before it can be

activated. The HDRM will stay armed for 1 sec. If any other write command than HDRM activate is performed, the HDRM will disarm. The point of arming the HDRM is to ensure that the shutter is not released before the satellite is in place. The HDRM active will only be active for one second.

#### Performed steps L/H

These two registers count the number of steps the motor has taken during the ongoing or last operation. The low register contains the least significant byte, and the high register contains the most significant byte. When the maximum number these two registers can hold is reached, the number will wrap-around back to 0.

#### Heartbeat count

The primary purpose of this register is the safety handling of the radiation shutter. This heartbeat count is set up to protect the instrument by closing the shutter in case of a DPU failure. The DPU must access this register minimum every 30 sec or the RSE will initiate an emergency closure. This register will start with a 0, and each access will increment the content. In order to prevent overflow, the number will wrap-around.

#### Processor status

The status of the RSE firmware system, as described in Table 4. The two signals here are the heartbeat missing, which indicates that the heartbeat has gone 30 sec without having been read and that an emergency closure is initiated. The reset functionality is designed the same way as the HDRM. This means that there is a reset armed signal that needs to be set before the system can be reset to prevent any accidental resets.

### 4.2.2 Control registers

The DPU needs to be able to control the operation and to do that we need some control registers. These registers need to be read/write register. The control register is shown in Table 5.

*Table 5 Control registers*

| Register | Address | Default value | Purpose |
|---|---|---|---|
| Motor current | 0x20 | 100 | Set stepping current. 0xFF for max available current |
| Settling time | 0x21 | 20 | Ramp up time allowed to reach set motor current before chopping start, in steps of 4 microseconds |
| Chop duty cycle | 0x22 | 150 | Step Motor chop duty cycle time in units of clock cycles relative to 256 |
| Max acceptable motor temperature | 0x23 | 170 | For motor protection. 0xFF for no temperature protection |
| Max acceptable electronics temperature | 0x24 | 170 | For electronics protection. 0xFF for no temperature protection |
| Max steps for operation L | 0x25 | 255 | Number of steps allowed before the operation is aborted (LSB) |
| Max steps for operation H | 0x26 | 255 | Number of steps allowed before the operation is aborted (MSB) |
| Enable ±12 V | 0x27 | 0 | Set to 1 to enable ±12 V rails |
| Max motor current | 0x28 | 205 | Maximum allowed motor current. In order to set motor current higher, this register must be changed first. |

### Motor current

The motor current register controls the duty cycle of a PWM, in order to set the current level for the motor driver. The output of the PWM is the $V_{ctrl}$ shown in Figure 8. The PWM starts with an output of '1' and counts until the motor current value is reached, where it switches to '0'. The register can contain 0-255, where 255 will be the maximum current.

### Settling time

The settling time register is in control of the duty cycle of a PWM, but this PWM will only send one pulse on each start and not wrap around for a new pulse. As the settling time signal will only be used in the start of turning on a new set of motor driver transistors, it is is easier to give the settling time a reset signal that initiates a single pulse instead of having to ignore the rest of the pulses.

### Chop Duty cycle

Chop Duty cycle controls the duty cycle of another ordinary PWM, in order to control the current chopping. Chopping is described in Section 2.4.3. The chopping will start with a '1' and switch to a 0 when the register value is reached. Should the register be set to the maximum value 255, then there will be no chopping so that the current from the transistors will be DC.

### Max acceptable motor temperature

This register contains the maximum allowed motor temperature. The measured temperature is compared to the maximum allowed, and should it exceed the maximum the motor operation will have to wait to let the motor cool down. This maximum limit is to ensure that the motor does not overheat and degrade consequently. Should we want to ignore the temperature measurement, the maximum acceptable temperature is set to the maximum register value, as no measured value can exceed it.

### Max acceptable electronics temperature

The max acceptable electronics temperature is set as the mac acceptable motor temperature, to protect the driver circuit electronics. The driver transistors will degrade and might stop working should they overheat. The temperature is measured and compared to the maximum the same way the max acceptable motor temperature works, and also halt any running operation should the temperature exceed the maximum allowed.

### Max steps for operation

The max steps for operation limit the open/close shutter – max number of steps operation. The operation will run until it reaches the maximum number of steps where it will end. The operation is explained in Section 4.4.1.

### Enable ±12 V

The LSB of this register will turn on and off the power rails to the driver circuits. Turning off the power will stop any commands from happening as a driving current cannot be created. Not having the power on will also stop any leakage through the transistors so that power is saved.

### Max motor current

Max motor current acts as a safeguard on the motor current register, so that the motor current cannot be set above the max current without increasing the max motor current. The maximum current should be set so that the motor and the driver electronics don't overheat to extensively.

## 4.2.3 Debug registers

A debug register can be written or read-back for verification. These registers, as shown in Table 6, are used for various RSE debugging tasks.

*Table 6 Debug registers*

| **Register** | **Address** | **Default value** | **Read/ Write** | **Purpose** |
|---|---|---|---|---|
| Disable blinking LED | 0x30 | 0 | R/W | 0x01 Disable the blinking LED |
| Seconds since access | 0x31 | NA | R | Number of seconds since last heartbeat register access |

### Disable blinking LED

To be able to easily check that that communication works while testing the system on board the disable blinking LED register is used. This register will be used to control a blinking LED on a test pin on the breadboard. The blinking LED will be on when the FPGA is programmed or reset. If the LED is blinking the system is properly reset and functioning. Then the disable blinking LED register can be written to, to disable the blinking led. If the LED stops blinking, then the communication is working.

#### Seconds since access

This register enables monitoring of the heartbeat register timer during testing, for example, to see that it counts the correct amount before initiating the emergency closure.

### 4.2.4 Command register

A command is used to perform RSE operations like opening the radiation shutter and closing it, by writing to the control register. The register is automatically reset to zero after receiving a new command and will therefore always read 0x00, and is shown in Table 7.

*Table 7 Command registers*

| Register | Address | Default value | Purpose |
|---|---|---|---|
| Command register | 0x40 | 0 | Register to take commands. Commands listed in Table 8 |

## 4.3 Operational procedures

After power-up and initialisation of the RSE, the DPU should check the RSE status registers to confirm normal operation. In particular, the heartbeat counter register should be accessed regularly to prevent emergency closure of the RSM, see Section 4.2.1. After that, the DPU should configure the parameters of the control registers according to specifications and requirements. Then, after renewed confirmation of nominal conditions, the commanding of RSM opening or closure can be performed.

### 4.3.1 Emergency closure

The heartbeat function work as a register that needs to be accessed to safeguard the system. The RSE is dependent on commands from the DPU to perform operations. Should the DPU malfunction or the communication link not work, the RSE will need to protect the SXI instrument as the RSE does not know if the satellite is within the radiation belt of the earth or not. Should this happen, an emergency closure needs to be performed.

The emergency closure will be initiated 30 seconds after the last heartbeat was received. At this point, the power rails to the driver circuit are dependent on the enable ±12V rails register. The

register needs to be overridden as the rails could be either on or off. When turning the power rails one, we need to wait one additional second to ensure the power rails are fully on, as they will have some delay turning on. During this if the DPU can reaccess the RSE it should be able to cancel the emergency closure and return to normal operation, as being able to access the RSE at this stage should indicate the DPU is still working.

After the power rails have been on for a second, an emergency closure will be initiated. This closure will go on until the end switch is engaged. This operation will also be able to cancel, as to more rapidly get the shutter open again should the DPU be able to access the RSE quickly after the emergency procedure was initiated. When the shutter is closed, or the operation was cancelled, the RSE will wait for a new heartbeat before any new motor operation is allowed.

An emergency closure will adhere to the max acceptable temperature registers, to not destroy the radiation shutter. Should the temperature be disregarded, the max acceptable temperature needs to be set to the maximum register value. This way the operators of the satellite will be able to select if they want a fast closure at the expense of the degradation of the radiation shutter, or if a slower closure with the risk of degrading and damaging the instrument be acceptable. As destroying the radiation shutter will either shut the instrument inside or leaving it exposed to high energy radiation, while saving the radiation shutter can damage the instrument.

### 4.3.2 Temperature reading

Two temperatures need to be read: the motor and the driver electronics. There is a pt100 thermistor on each that is connected to an analogue-to-digital converter. The minimum and maximum temperature allowed on the motor are from -150°C to 70°C [7] and on the electronics -55°C to 80°C [10]. The value of the two temperatures will be sampled each second as long as the power rails are on.

## 4.4 Command operations

A command will result in activation of the required procedure, or an error response, as discussed in Section 4.6. If, for example, the motor or electronics temperature is too high, the operation is not permitted, but if the motor or electronics temperature becomes too high during RSM closure/opening, the RSE will suspend the operation until the temperature is below the threshold.

In the latter case, no intervention is required from the DPU side, but in case of an emergency, the operation may be aborted, in order to adjust the temperature thresholds before reissuing the RSM operation.

There is a set of sporadic operations that operates the stepper motor or the HDRM or are used to reset the RSE. These commands can be performed at any time, from the DPU. The command register is made to accept these commands into it and then perform the corresponding action. The command register is a write-only register, which clears directly after having been written to. All commands are shown in Table 8.

*Table 8 Commands*

| **Register** | **Address** | **Command identifier** | **Purpose** |
|---|---|---|---|
| Open Shutter Stop at end | 0x40 | 0x01 | Open Shutter, run motor until end stop detected |
| Close Shutter Stop at end | 0x40 | 0x02 | Close Shutter, run motor until end stop detected |
| Open Shutter Max no of steps | 0x40 | 0x04 | Open Shutter, run maximum no of step regardless of end-stop detection |
| Close Shutter Max no of steps | 0x40 | 0x08 | Open Shutter, run maximum no of step regardless of end-stop detection |
| Emergency close Stop at end | 0x40 | 0x10 | Close Shutter as fast as possible, run motor until end stop detected |
| Arm Reset | 0x40 | 0x20 | Arm Reset Function |
| Reset | 0x40 | 0x22 | Reset RSE FW |
| Arm HDRM | 0x40 | 0x40 | Arm Activate Hold Down and Release Mechanism |
| Activate HDRM | 0x40 | 0x42 | Activate Hold Down and Release Mechanism |
| Cancel command | 0x40 | 0x80 | Cancel any ongoing command |

Five commands start the motor, two that are necessary to reset the RSE, two for activating the HDRM and a final command that cancels whatever is ongoing. The HDRM commands will not be used after the EBB, as they are not needed.

### 4.4.1 Motor commands

The motor needs to have at least one open and one close shutter command. Since the shutter has end switches for open and closed, these can be used to end a command. In case of a malfunction in one of the end switches, there needs to be an alternative set of commands to open and close the shutter. These operations will be suspended should a temperature reading exceed the maximum allowed temperature, to allow the electronics and/or the motor to cool down. Temperature readings will be performed regularly so that the operation can continue.

#### Open/Close Shutter – Stop at end

These commands are designed to be utilised during the regular operation. The purpose of these is to open or close the shutter. This operation is performed until the end switch is engaged.

#### Open/Close Shutter – Max no of steps

The stop at end commands is reliant on that the end switches work. In case the end switch stops working, there needs to be another method of getting the shutter open and closed. These two commands work independently of the end switch and will then work if the end switches fail. The max number of steps command uses a set control register to run. The operation will start and count the number of steps. It will run until it reaches the set maximum number of steps, before ending the operation.

#### Emergency close – Stop at end

The emergency close – stop at end command operates the same way that the normal close shutter stop at end command. It was held as an option that this command could ignore temperature readings, but it was decided that the command should wait if the motor or electronics got too hot. Therefore, this command is now identical to the regular close shutter command. If the motor or electronic temperature should be ignored, then the max allowed temperature registers should be set to max.

### 4.4.2 Other commands

There is a couple of other commands that are not directly related to the stepper motor.

#### Arm Reset & Activate Reset

The arm reset command will arm the reset function. This signal will hold reset armed for 1 sec, where it will be unarmed if nothing is written to the RSE. If anything, other than a activate reset is written, it will unarm. The activate command will activate the armed reset signal. Only when the reset is armed and activated within 0 seconds after it was armed, will a reset of the RSE be performed.

#### Arm HDRM & Activate HDRM

Arm HDRM arms the HDRM signal, in the same way, the arm reset work. After 1 second or should any other write operation than activate HDRM be performed, the HDRM signal will be unarmed. The activate HDRM command needs to come within that time limit to activate the armed HDRM signal. The two HDRM commands will only be used on the EBB as they are not needed after. After EBB the PSU will activate the HDRM.

#### Cancel command

This command will cancel any other command. As it is a write command, it will unarm both HDRM and reset. It will also end any motor operation, including an emergency that has started from a missing heartbeat. For example, should an emergency closure be initiated from a missing heartbeat, and the DPU is still working, the closing operation can be cancelled, and the shutter reopened without having to wait for the shutter to close.

## 4.5 Communication

The DPU needs to be able to access the RSE and write and read data to and from it. As the RSE does not act independently, it acts as a slave that reacts to commands from the DPU. A protocol is then necessary to use to facilitate the communication between the RSE and the DPU, so commands can be sent from the DPU to the RSE and the RSE have some way of reporting back to the DPU.

A protocol defines a dependable interface where the communication between multiple components can be performed predictably and must weight different needs against each other. There are robust

protocols like the Remote Memory Access Protocol over Regular Byte stream DAQ Protocol (RoR) protocol that is a secure protocol which is good at roughing out errors, but it also has much overhead and therefore slower with transferring actual data bits. Different protocols prioritise being small and with little overhead. This means a high percentage of the payload are data bits. A smaller protocol might sacrifice robustness so it cannot detect some errors that a more robust protocol can detect.

The previous iteration of the RSE used the RoR communication standard in [3]. A simpler protocol was possible to use as the communication was going to be performed point to point. Point to point communication means that there will only be one master and one slave, and the communication will be directly between them. A single slave and master make using a slave address unnecessary as there is only one slave. A simpler protocol could be used.

As the DPU and RSE communicate point-to-point, a simpler interface was possible and less resource demanding. Neither a slave address nor a chip select is necessary. However, we need to do some checking. As the RSE must be autonomous, it does not need to be able to send data from the RSE without being prompted from the DPU. At the low level of the transmission, the baud rate is defined as the number of symbols that are sent per second, as there are methods to include more than one bit in a single symbol transmission. In our project, a standard I/O pin is used that either is set high or low. This results in the baud rate are the same as the bit rate.

A standard Universal Asynchronous Receiver-Transmitter (UART) is suitable for a simple communication link. The drawbacks of the UART is that it cannot support multiple slaves and masters, both the master and slave must use a set baud rate and the limit on the data frame. There is only one slave and one master in our system. The baud rate has to be set the same at both the master and the slave for both systems to be able to interpret the incoming data. This problem could make a UART undesirable as the technology might come with a set baud rate or a couple of selected baud rates. Given the whole project are developed to be used on an FPGA, the limits on the baud rate are how fast the system clock can sample it and the sampling rate, and other than that the baud rate is selectable. Since we are not limited in the same way the DPU is, this meant in effect that the DPU could set a baud rate that it is limited to, our system can adapt to it.

A serial peripheral interface (SPI) interface would require more pins than a UART as the UART only needs a receiving and transmitting pin, whereas the SPI would need a chip select signal and a clock signal. The SPI is great at full-duplex mode as data is sent in both directions during a transmission. Developing a UART that supports full duplex is more complex to do. The SPI would introduce a second clock to the system as the master will send its clock to the slave. Two clock domains would give us a problem with having to cross the clock domains, but can be solved by letting the clock only do the sampling and putting up slow write signals, and then letting the internal clock react to the rising edge of the write signal for new data.

I2C main advantage is that it supports multiple masters and up to 1008 slaves, which is not useful in this project. It also requires a more complex solution than a UART or an SPI. [25]

As a UART was to be utilised, an internal protocol on top of the character level was developed that satisfied our demands. The UART handles the character level, which handles the low-level bits, and the RSE protocol that handles the top-level communication that looks at the characters that are sent.

The system clocks frequency is 10 MHz. We are going to send eleven bits each time the UART is used. There are about 20 registers that should be read once each second. With the proposed RSE protocol, there are four times this with overhead, so we end up with at least 880 transactions each second. Using the RoR protocol would add five times the overhead increasing this to 4400 transactions each second.

Going with the RSE estimate, some transactions must happen each second, but also a lot more could happen during a second if a start-up of the motor is initiated. A baud rate of 100 kHz was proposed as it would give us some speed, but not make it a high-speed system as it is not needed. The motor will use around 40 seconds to go from open to closed and vice versa, so the speed of the transmission does not have to be set as fast as possible.

As the speed is not a huge problem, and we are also able to use a half-duplex system. As the DPU is always the only one to initiate a transaction, there are no problems with making sure a collision does not occur with a half-duplex system. A full-duplex mode would be possible, but this would have required using queues and a more complex system with a different philosophy. As we have

already set the speed low, it would be easier to set the speed higher than creating a full-duplex system, if we wanted a faster system.

## 4.6 RSE protocol

A simple serial communication part, such as a UART only creates a method of sending a set of data bits. What goes into these data bits in what order needs to be defined as the UART does not define order. There exist multiple large complex protocols, but there are no standardised smaller protocols. Therefore, an internal protocol was developed that satisfied all our demands. As we do not want much unnecessary overhead, it contains only two layers. The character level drives the communication on the low level, and the package level decides what to put into the character level in what order, as shown in Figure 16.

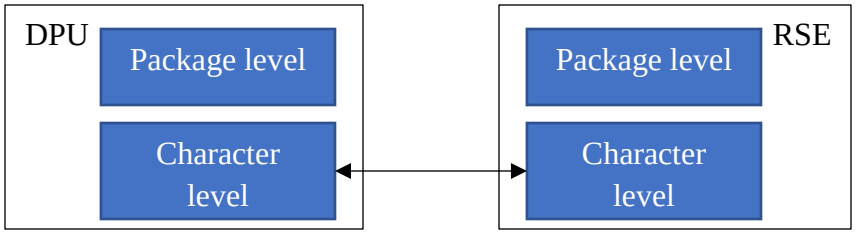


*Figure 16 RSE protocol levels in operation*

### 4.6.1 Character level

The character level is a single transmission package and is the low level of transmission. The character consists of a start bit, eight data bits, one parity bit and a stop bit, as shown in Figure 17. The UART must handle this level by receiving and checking each bit in order, before checking them overall. The UART must also handle sending out the bits in the correct order and calculate the parity bit.

Start and stop bit are necessary to detect a new package is being received. A package that will not be able to detect at the right place where to start if there is no start bit. Data bits transfer the data from one component to the other. If a bit flips during the transmission, the parity check on the parity bit will be able to detect this. A parity bit can help the stability of the system by rejecting faulty characters. Adding a parity bit comes at the cost of more overhead on the character level. There could have been implemented some hamming code to restore the package, but it was considered more straightforward to reject the package and let the DPU know that an error occurred.

To be able to see if the communication lines break, the idle level is set ordinarily high. The parity bit and the stop bit will be wrong if the line breaks so that the UART will accept no characters. The last error that the character level needs to handle is if a bit cannot be validated. So, if a bit is not stable in the middle of it, it must be rejected.

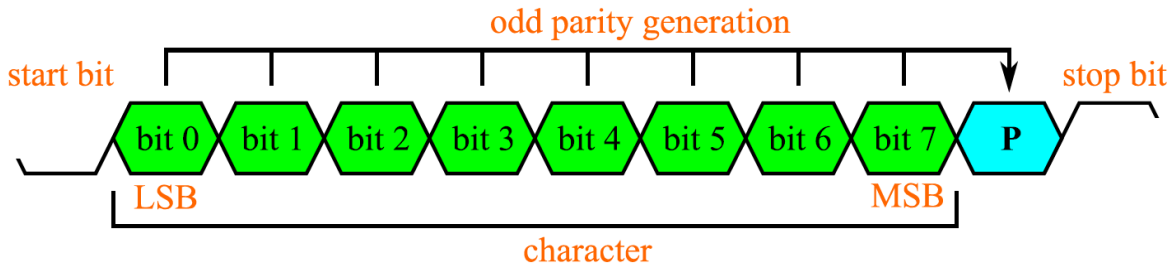


*Figure 17 UART character [26]*

### 4.6.2 Package level

The package level is the high-level part of the communication. A high-level lets the DPU and RSE make sense of what is put in the characters and in what order characters should be sent.

The primary purpose of this protocol has been to keep the overhead on the package level low. We have point-to-point communication, so no invocation with a slave address is needed. A write and a read transaction are defined in the protocol. The DPU is the one to initiate the communication, and the RSE respond on the command. Both commands need a write/not-read (wnr) bit and an address in the memory to interact with. As the highest defined register has an address of 40, so only 7 bits are necessary. This lets the wnr bit be placed in the same character as the address to save one character in transmission. Should however there be 8 bits required for the address, the UART could be extended to use 9 data bits instead of requiring more character to send the wnr bit.

In the response message, the response should include the whole command received so that the DPU can check if the correct command was performed. If it was not the DPU can send the correct command again, to write over the faulty command. The command could be dropped from the response to achieve even less overhead, but by having it in, the DPU gets some help in verifying the that the correct command was performed. Otherwise, the DPU would have to do a read operation, as well as having to check that the correct data had been written.

There is no cyclic redundancy check in the protocol to make it more error proof and no self-checking. To reply with the command and letting the DPU overwrite the error fast was a better solution, as implementing self-checking would make the system more complex.

There is a sequence number in the response message. This lets the DPU keep track of how many commands have been performed and is a fast way for the DPU to figure out that something was skipped or went wrong in the previous interactions with the RSE. The DPU should then proceed to read all the registers to check that everything is as expected and the DPU should figure out if it must reset the RSE. The whole memory write and read procedure is illustrated in Figure 18.

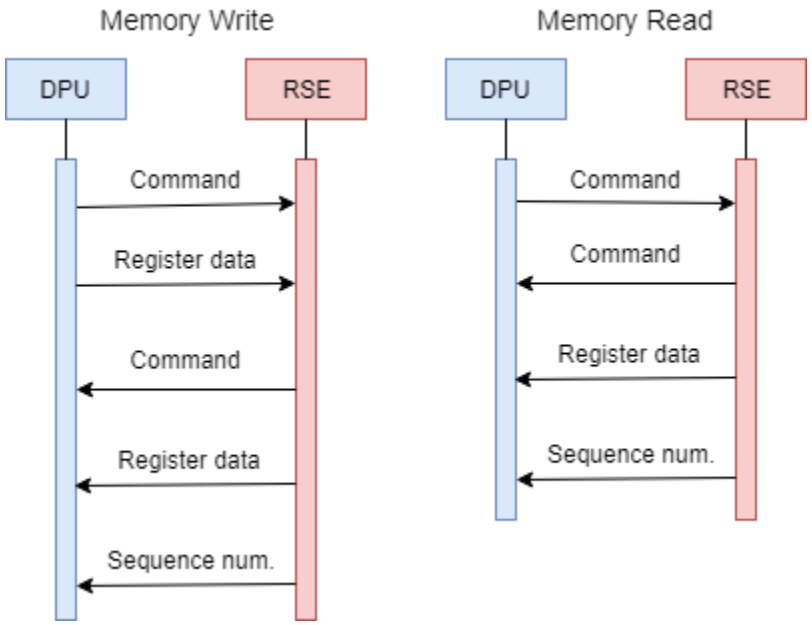


*Figure 18 RSE protocol memory write and read transaction*

### Read operation

The read operation is going to be performed most, so it been prioritised to keep it as small as possible. So, it is necessary to provide a wnr bit and the address, which could be done in just one character. This was then selected to be the read command, as seen in Table 9. The response would include the command, the status of the register that had been read and a sequence number, and is shown in Table 10.

*Table 9 Read command format*

| **8 bits** |
|---|
| Command |
| 0AAAAAAA |

*Table 10  Read response format*

| **8 bits** | **8 bits** | **8 bits** |
|---|---|---|
| Command | Status | Sequence number |
| 0AAAAAAA | BBBBBBBB | SSSSSSSS |

### Write operation

The write operation is designed the same way the read operation is, but it also needed the 8 bits that are to be written to the desired register. The write command would include two characters, as described in Table 11. The first containing the command, and the second with the data bits to be written into the register. The response would mirror the command and the data bits that have been written in addition to the sequence number, as shown in Table 12. This lets the DPU see that the command has been performed correctly or incorrectly.

*Table 11 Write command format*

| **8 bits** | **8 bits** |
|---|---|
| Command | Register |
| 1AAAAAAA | RRRRRRRR |

*Table 12 Write response format*

| **8 bits** | **8 bits** | **8 bits** |
|---|---|---|
| Command | Written value | Sequence number |
| 1AAAAAAA | RRRRRRRR | SSSSSSSS |

### Error response

In the case that an error occurs, the RSE will provide an error indicator and an error message, so that the DPU can see what went wrong. The response message also contains the sequence number, as it might indicate other errors if it has skipped, as shown in Table 13.

*Table 13 Error response format*

| **8 bits** | **8 bits** | **8 bits** |
|---|---|---|
| Error indicator | Error message | Sequence number |
| 11111111 | BBBBBBBB | SSSSSSSS |

The different errors are listed in Table 14. The low numbered error messages tell that something went wrong on the low level (parity, validation, timeout). The higher numbered error message indicates that there is a top-level problem, i.e. not allowed.

*Table 14 Errors*

| Response | ID | Description |
|---|---|---|
| Parity error | 0x01 | Message parity error |
| Data validation error | 0x02 | A received bit could not be validated as '0' or '1' |
| Transmission timeout | 0x03 | Slave timed out waiting for remaining words from the master |
| Not writable register | 0x04 | The register is read-only |
| Wrong address | 0x05 | No such register |
| Not allowed | 0x10 | Not allowed since another command is already being performed |
| Shutter already closed | 0x11 | Not allowed since the shutter is already closed |
| Shutter already open | 0x12 | Not allowed since the shutter is already open |
| Motor temperature too high | 0x13 | Not allowed since the motor is too hot |
| Electronics temperature too high | 0x14 | Not allowed since the electronics are too hot |
| 12V rails not active | 0x15 | Not allowed since the ±12V rails are not on |
| Exceeding max current | 0x16 | Not allowed to set current above the max motor current |
| Unknown command | 0x21 | The received command is undefined |

### Timing

Given that a half-duplex transmission, the RSE or the DPU are not allowed to start transmitting before the end of the stop bit on the command or response was received, as illustrated in Figure 19. This would give the system feeding the UARTs data some slack to finish its operations and go back to idle.

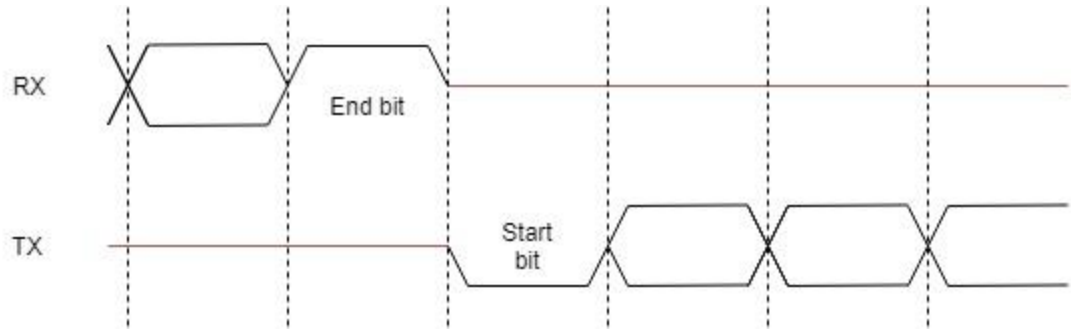


*Figure 19 Start of new transmission timing*

The write operation might timeout in between the two parts of a write command, or the RSE might timeout in between two characters. This timeout is called a transaction completion timeout. To make sure this timeout does not happen, a maximum of 11 baud delays between successive words is allowed. If this requirement is breached, the RSE will issue an error message if it is waiting for the DPU.

The RSE might also timeout after it has received a command. This timeout is a response timeout. If the RSE has not started sending the response within 11 baud delays, it should clear and be ready for a new command so that the DPU can start a new transaction. The slave would need 11 baud delays to be able to recognise a silent link. Eleven baud delays were selected for both timeouts as it is the length of one full character from start to stop bit, so a silent link can be recognised should either the DPU or the RSE have mistakenly started a receiving operation.

# 5 Firmware implementation

## 5.1 Top level design

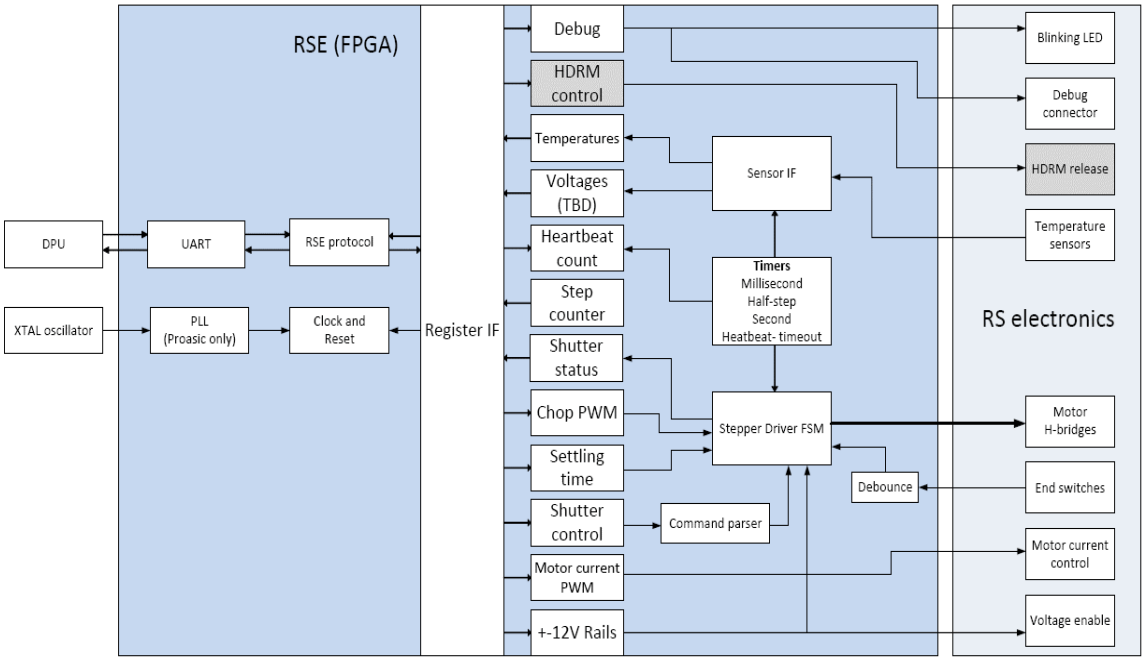


*Figure 20 Top level design*

At the top level, as shown in Figure 20, there is a UART that handles the low-level communication, and an RSE protocol that handles the top-level communication towards the DPU. The data received is put into the register bank, and requested data is pulled from there. The register bank stores all the software registers and connects wires from other places, so they act as read-only registers. The data registers are then sent out in records to the different modules that need them. Some modules send back value to the registers. These are read-only register as modules control them.

In the top level, there is a couple of timer modules. There is a clock generator module that generates a pulse each millisecond and second. These signals are used to time the other processes, so they run synchronously to each other.

The design is split up where it is appropriate. The stepper driver is reliant on a lot of in signals, but to keep the size of it down it only connects different incoming signals to the output when they are

required. Dividing up the modules made the testing simpler as an error could be isolated to its module that produced the wrong signal. Also, by using regularity as a philosophy for the different modules, the modules were simpler to build fast and test.

## 5.2 UART

UART is a component that handles the low-level communication with the DPU. The UART must control the transmission and check that all the communication is legal. All the communication is done in baud periods. Therefore, each step has a counter which is counting the clock periods in that step and checking if it reaches the number of clock pulses per baud period, before entering the next step. When no communication is occurring, the transmission lines are set high. The start bit is then low so that a new message can be detected. Then comes the eight data bits from the least significant to the most. Then a parity bit comes, before a stop bit, which is high.

The UART consists of two parts, with support functions. One part handles the transmission of packages from the RSE to the DPU. First, the transmitter needs to be started from a valid data signal. Then the transmitter needs to fetch the desired data that is to be sent. This data is then wrapped with a start bit, parity bit and a stop bit. All the data is sent through the transmitting data line. First out in the transmission, the start bit is transmitted for one baud period. Then the data bits will be sent for one baud each. A counter is utilised to count which data bit is in progress of being sent. While the data bits are being sent, the odd parity bit is being calculated in a separate process. A separate process makes it so that the parity bit does not have to be calculated during one single clock cycle. The calculated bit is then sent out for one baud period. At last the stop bit is sent out for one baud period to complete the transmission.

The receiver part controls the reception of packages from the DPU to the RSE. The receiving transmission line is monitored to check for incoming data. First, all the data from the transmission line is synchronised through two registers. The synchronisation is done to avoid metastability problems, so the signals are usable in the UART clock domain. The synchronised bit is sampled each clock cycle into a sampling register. This register oversamples the bit by a factor of clock pulses per baud period. The oversampling, together with a counter, can then look at the middle bits of the transmission. The five middle bits are compared, and if all of them has the same value, the sampled value is accepted. If they have different values, an error is indicated. That way, the bit

used is sampled when the transmission is at its most stable, and all the middle bits are the same value ensures that a proper sample is utilised. This process is illustrated in Figure 21.

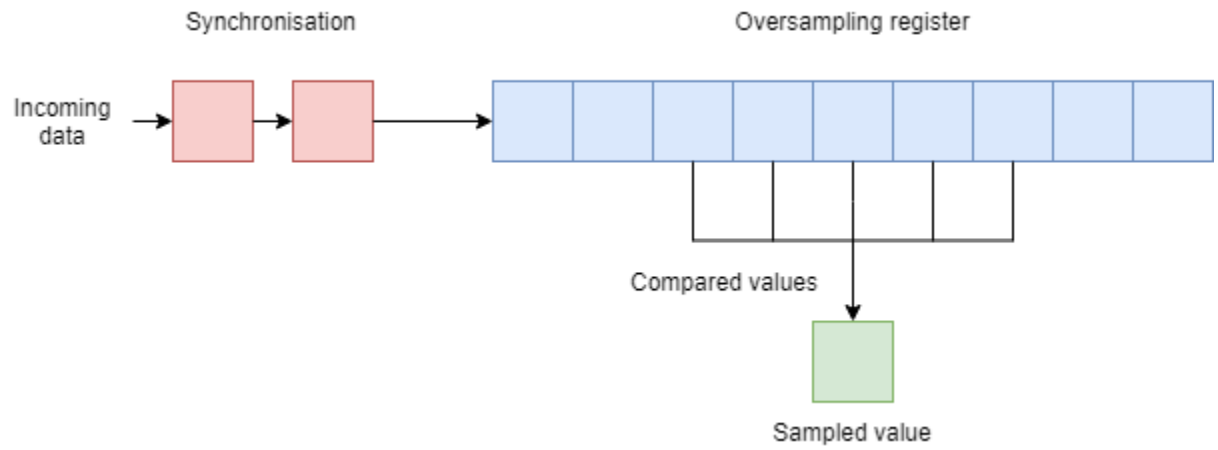


*Figure 21 Sampling the value of incoming data*

When the transmission line goes low, a start bit is detected. This start bit will then fill up the sampling register. The sampling register is looking for a start word where half the register is high, and the other half is low. The start word is timed, so the next steps samples when the whole sampling register is full of the first single bit. If the search word appears, the next step waits so the whole bit is in the sampling register. Then it checks the middle bits if they are all low. If not, something is wrong, and the state goes back to idle, where it waits for a new search word to be found. Should all the bits be 0, the counter is reset and, the process goes to the next step. There the data bits are collected. They are checked if they are valid and then put into a register, where a counter places them in the correct order. After the data bits are collected, a parity bit is calculated, and the parity bit from the package is received. The calculated and the received parity bit is then compared. If they are unequal, an error flag is set. At last, the stop bit is collected, and a flag is set indicating that a new word has been received.

## 5.3 RSE protocol

The RSE protocol handles the top-level communication of the transmission in accordance with the protocol. The implementation of the top-level is done by using a state machine, as illustrated in Figure 22. Since all communication is initiated from the DPU, the state machine can wait until it receives something. The UART will deliver the data and a valid data bit when new data is received. It will also deliver flags for low-level errors such as parity bit error and data validation error. The state machine has one idle state, and one state for each of the words that are to be sent and received during the transmission, in addition to two wait states. It also has an error state where an error message is produced, which the state is sent to if an error has been produced.

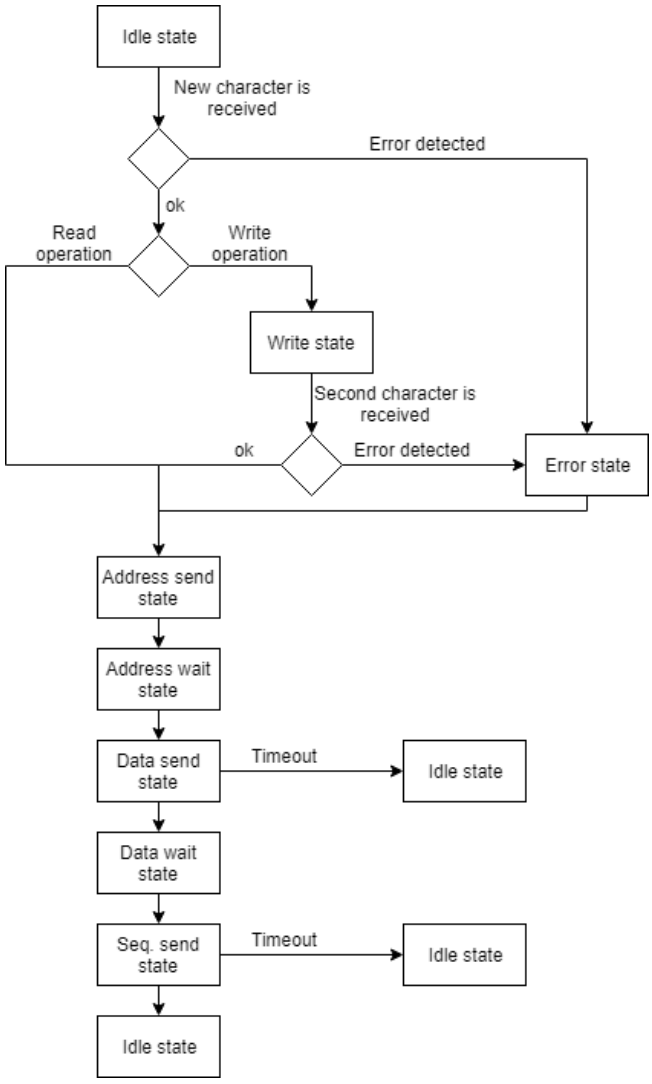


*Figure 22 RSE protocol state machine*

First, in the idle state, the code will be waiting for the UART to indicate that a new word has been received. Then the parity check flag will be check. Then the write/not read bit is checked. This determines whether a write or read procedure is to be performed. This bit is put in a variable to keep it for later purposes. Since it is a variable, it can be utilised at once. If it is a write operation, the state is sent on to the write state, and the timeout counters are reset. However, if it is a read operation, the address that is received is check if it exists before the read operation is performed by setting the cs and read signal high. The state is then set to address send.

The write state waits first for a new data valid flag to be set. Then the data validation, parity, existing address and writable address are checked for in that order. This order is used so that the lows level error is given priority. This is done as an unwritable address is an input error from the DPU, but a validation or parity error is a much more critical error as there is something wrong with the system.

Then which of the address that is written to is checked. If the command register is tried to be written to, checks must be done to check if it is legal to write the command that is received. First, the command that is written needs to be checked if it is one command that warrants checks. If the command is related to the shutter first, the 12V rails need to be checked if they are on. Then the temperature of the motor and the electronics must be check if they are too hot. Then the status of the shutter is checked to see if it is in the process of opening or closing. At last, if it is an open command that has been received, check if the shutter is already open, and vice versa for the close command.

If the command is another known command, no checking must be done. However, if it is not a known command that has been received, an unknown command error is set. If no error was detected a standard SBI write to the register is performed, the timeout counters are reset, and the write data is placed in a data register inside the register bank, and the state is sent on to the address send state. If an error has been detected, the corresponding error message is put into the data register to indicate what kind of error has occurred, and the state is sent to the error state.

In the address send state if a read command has been performed, the data read is put into the data register. The data read is not ready before this state and is then put into the data register. Then the address and the wnr bit is feed to the UART so that it can be sent. When the UART tells that it is

ready to send, the valid data signal is set to initiate the transmission in the UART. It will then enter the address wait state. This wait state waits only one clock period so that UART gets time to reset the ready signal, so the data send state will not imminently happen. It also resets timeout counters.

Given that the shutter will use seconds to go from open to closed, and just the transmission uses a baud rate much smaller than the clock frequency, one clock period is not that critical. A method where a done signal was utilised instead of the ready signal was used early in the project but discarded to get a clearer interface. The done signal only last one clock period, and so the ready signal which lays high is easier to handle and made the waveform simpler to understand during the simulation.

The data send state waits for the UART to be ready again. When the ready signal arrives, it will send the data bits to the UART so that they can be sent. This procedure is performed while a transmission timeout counter is running. It will timeout at a set constant where it will go back to the idle state as it is defined in Section 4.6.2 that if the transmission is not initiated within 11 baud periods, the DPU can send a new package. To implement this the timeout counts twice so that the first package finishes before it starts counting for the second.

The same procedure is applied in the data wait state as in the address wait. This state resets the timeout counter and waits a clock period so that the ready signal has time to go low again. Then the sequence number state applies the same procedure as the data send state, but it will send out the sequence number register value, before adding 1 to it for the next transaction. The addition is done to track the number of interactions from the DPU to the RSE.

For the error state, the data register is set to the corresponding error message when the state is set to the error state. Setting the data register then ensures that the correct error message is produced since the flag telling what is causing the error often last only a single clock period. The data register holds the data from the register bank when read, or the data to be written is held in this register during a write operation. The data register is what that is sent back during the data word of the package. The error state itself replaces the write state where the state machine waits for the data to be received. It also works as a stage in between the idle state when a character has been received, and the address send state where the response is generated, in case of a read operation. The address to be sent is set to 1’s indicating an error has occurred, and the cs is set to 1.

The check that controls that the received address is valid and the check for a writable address are both placed in functions as the to help the readability of the state machine. A simple function name is used, and in them, they will check the received address against all legal addresses.

## 5.4 Smile register bank

This module was generated using the Bitvis register wizard early in the development. The generation of a register bank created an SBI interface, and it has been used to access the register bank from the RSE protocol code. This SBI interface was chosen to be used early so it could be used for learning to utilise the Universal VHDL Verification Methodology tool from Bitvis. There is a top level of the register bank, and it consists of two parts: the SBI controller and the core where the registers are located, with two records in-between them.

The SBI controller is an SBI interface and two decoders, one for writing to the register bank and one for reading from the register bank. The record from the register bank contains all the register values. In the read process, if read and chip select is set, the address is used to chose which register value is placed into the read data output. The only case where it is different is that if the heartbeat register is read, it produces a heartbeat count accessed signal that is used in the heartbeat module in Section 5.7. This is done to simplify the heartbeat module. The write process checks if the write signal and chip select signal is set. If the write signal is set, the address is checked, and the corresponding register signal in the record is set to the write data value, and the write enable signal is set for that register. All the individual write enables is or-ed together to create a general write signal. This is done because some other modules must see if a general write was performed.

The register bank contains registers for each of the defined writable and static registers. Most of the read-only registers are signals come in as through a signal in a record and are just passed directly on to the SBI. The registers are also connected to the different records going out as they are needed in other modules. There could have been a general record out, but since most modules only use a few of the registers, individual records were created after where they are going to. This also made the naming of the records symbolise more what they contain. The command register is unique since it is cleared after holding the new value for one period.

## 5.5 Debugging module

To make the first test on the board easy a debugging module was created. The purpose of this is to have a blinking LED on startup. This LED makes it simple to see that the programming of the board was successful. The communication could also be checked if the blinking LED could be turned off, as visual feedback is provided instantaneously from the led. The LED will start blinking again on reset. This has been great to check with when problems occurred as a sent command was received on the board, but no response came back. This reduced the amount of probing and time used on other techniques to troubleshoot.

The debugging module code is implemented as a process with a linear-feedback shift register. The shift registers output change value roughly every second. The process was made to be independent, and while the sec pulse from the clock generator could be used, it would make it rely on other modules, and not be fully independent. A normal binary counter could have been implemented instead. Since the counting order does not matter, the LFSR has a simpler logic, and there was a wish to try using an LFSR, it was decided to be made that way.

The debugging module was created early in the project after the communication and register bank was done. This was to enable testing of accessing the register bank on an FPGA early in the development.

## 5.6 Clock generator

A clock generator was made as a millisecond and a second pulse was required. The pulses were required to be independent, so multiple modules could rely on it to keep them synchronous. This central clock generator makes the other modules dependent on signals outside them instead of using counters inside them. However, having a synchronous clock pulse and reducing the number of counters was considered to be better. It was simpler to develop and made the readability better because of the reduced size of other modules.

The clock generator is built with a conventional counter where a variable counter is used to count from 0 to a constant holding the value of clock pulse per millisecond. This counts each clock pulse. When it reaches the constant value, a pulse is set, and the counter is reset. This generates the

millisecond pulse. Then the same process is repeated to create a sec counter. The millisecond pulse is used to count, and a second pulse is created.

In addition, an arm timeout process is used to timeout the arm signals. This process counts on each ms. There is an uncertainty in wherein the ms counting process the ms counter is when the arm timeout is reset, so an additional counter was used for arm timeout. The arm timeout is reset by the different modules that use this process. When reset the counter will count for one sec, where an arm timeout signal is set. This is sent back to the modules that need it.

## 5.7 Heartbeat

The purpose of the heartbeat module is to be able to control that DPU communication is working. If the communication lines stop working, the instrument needs to be protected, as the RSE does not know anything about the outside and what is happening. The DPU is required to read the heartbeat register at least once each 30 sec. This lets a timer be created to count until the communication can be assumed to be broken. When the heartbeat register is read, the counter is reset back to 0. The counter counts on each second pulse. The counter will remain at max value if it is reached. This is to prevent the counter from looping around and deactivate the emergency closure. If it counter reaches the timeout constant, a heartbeat missing signal is set.

Each access increases the heartbeat register's value by one. Incrementing is a simple way to see that the heartbeat register has been accessed. The heartbeat module monitors the SBI interface into the register bank and performs the addition if the heartbeat accessed signal is high. The heartbeat register value is sent back to the register bank where it is just connected to the SBI interface directly.

## 5.8 HDRM

HDRM is the module that is supposed to release the shutter after the satellite is in place. The shutter is locked in place with a hold-down mechanism during launch. This needs to be activated by a current to release. The HDRM module provides a signal to activate this.

The HDRM requires a command to arm it, and then a command activating it. If any other write operation than the activate command is performed while the HDRM is armed, it will unarm. This

is to make sure that the shutter is not released before it should, and thereby doing damage to the shutter and its surrounding.

The process will wait for an HDRM arm command. If an arm command and a command write are written to the register bank, the arm signal will be set. When the arm command is received, the HDRM resets the arm timeout procedure in the clock generator. This will give back a signal if a too long time goes before the activate command is received. Then if a write command is performed to the register bank and the arm signal is high, the command register is checked if it is written an HDRM activate command to. If it is, the activate signal is set, and else the armed signal is reset. If the timeout signal is set, the arm signal is also reset. The module sends the arm and activate signals to the register bank to the HDRM status register.

## 5.9 Reset generator

The reset generator operates much in the same way the HDRM works. It is designed to be able to reset the whole design while from a command. To perform a reset an arm and a activate command is needed to be written in succession to the command register within one sec of each other. This is to ensure no single wrong package can by accident reset the system.

The reset generator runs a process that checks if an arm reset command is written to the command register. If it is, the reset armed signal is set, and the arm timeout is reset in the clock generator. Then if the timeout is activated, the arm signal is reset. However, if the arm is set and a write command is received, the process will check if a reset activate is written to the command register and then set the reset activate signal. If it is not a reset activate command, the reset arm will be unarmed.

## 5.10 Stepper motor modules

Most of the stepper motor could be placed in a single module. To stop it from bloating, it was simpler and easier to split it into different modules to handle the different purposes. This increased the number of global signals but was at the benefit of getting a better hierarchy. The modules have been tried to look alike and using the same methods to achieve the same results.

## 5.11 Switch debounce

Switches usually jump between closed and open, and there are no clean edges. To deal with this, a debounce module is needed. Two switches need debouncing, one that detects that the shutter is open, and one that detects that the shutter is closed. The debounce are not affected by the other code and are not required to be in sync with anything else. Since this is the case and the clock generator was providing a millisecond pulse, that was utilised to keep it simple.

The debounce process receives the millisecond pulse and from that counts until it reaches the set debounce time constant. When it reaches the constant, the current input value from the switch is checked against the previous value. If they are the same value, the incoming value is let through, if not the previous value is used.

This method is used so that if the stepper motor samples in a transition period of the switch where the signal will be jumping between high and low, it will have to run another step. The alternative is not to have a debounce module and just let the signal straight through. It could just as well work, as the noise on the signal is only present when the switch is transitioning, which means it is either as good as open or closed. The momentum of the shutter might be good enough to close the shutter fully. There is also a closing step in the stepper motor to let the stepper sequence finish, and this might also be good enough to flip the switch value properly. However, at an early stage in the development, nothing was known about this, and so to ensure there are no problems the debounce was developed and utilised.

## 5.12 Pulse width modulator

The PWM's job is to create a digital signal that is modulated to be on and off in cycles. This signal is used to turn a transistor on and off, to achieve certain currents as described in Section 2.4.3. In this project, one PWM is used to control the current in the motor, and one to chop the signal on the transistors. This is to limit the power used.

The motor current is connected to an output which is connected to the current source, and by turning it on and off, a corresponding current is delivered. The chop signal is delivering a constant chopped signal, and this is used in the stepper driver to chop the signal on the Q3 and Q4 driver transistors, as seen in Figure 8.

The PWM takes in a register value. A process will count from 0 until the register value. During this, the modulated signal is high. When the counter reaches the register value, the modulated signal is set low. The counter then goes to the maximum value and then loops around.

A goal of the PWM was to design it as universal as possible so it could be reused.

## 5.13 Settling time period

The settling time module is a module should provide a high signal up until a particular time. This is done to drive the signals on the transistors driving the motor for the settling period. This allows the current through the driver transistors to be settled at the desired value before any power saving methods are applied. In this project, the settling time period signal is supposed to be high for the settling period, and then the signal is chopped to save power. The settling time period module is to be similar to the PWM module, except that it is missing the wrap-around. So, the module needs to be reset each time the counting is to be performed.

The way this is implemented is by creating two counters. One that counts until every four microseconds and one that counts those microseconds until the settling time constant is meet. When the settling time is reached, a settling done register is set. When the settling done register is set, the settling pulse output is set low, where it was high before the settling was done. When the register is set, the counters will continue counting, but they do not affect anything.

## 5.14 Half step synchroniser

To synchronise the steps in the stepper motor, and ensure all steps are equal, an independent step counter was developed to ensure this. This module is consisting of a counter that counts until the half step constant is meet, where an out signal is pulsed. This signal is used to proceed in the stepping order in the setter driver. The stepper driver is required to reset the half step synchroniser to sync the stepper driver sequence. This was a simple way of getting the stepping in sequence, instead of having to sync the sequence to the half step by waiting for the pulse.

## 5.15 Stepper control

To know what command should be performed, the command register must be interpreted. To interpreted it, a stepper control module was made to detect changes in the command register and pass the interpretation on to the stepper driver. Since the command register is cleared one clock cycle after a new command has been placed in it, so all new commands are easily detected by checking if the command register is not cleared.

To make this simple, the command register is checked if it holds a value which corresponds to a motor relevant command. Illegal commands are rejected in the communication part, so all commands that are received can be expected to be legal. For example, this will stop a new open command to override an ongoing close command. So, in the procedure the command register is checked, and then all the command flags are set low, and the new stepper command signal is set high. Then the command is checked which command is being received, and its corresponding flag is set high.

The only special case is the cancel command which checks if an ongoing command is being performed. If it is the cancel command flag that is set high, or else it is kept low, and the new stepper command flag is set low, so it never goes high. While it is not necessary for the stepper driver, it is done to ensure the new stepper command are always set with a command flag and not trigger any checks in stepper driver. This also ensures no unnecessary interactions across modules.

The flags are held high until a new command is received. This is so that the stepper can just check on the flag to know which command is going on, and thereby which end conditions to check for.

## 5.16 Stepper driver

The stepper driver is the component that controls the stepper motor operation. The stepper sequence must be fulfilled in the correct order, as shown in Figure 9. The signals that go into the stepper sequence comes from other independent modules, to reduce the size of the stepper driver module. The module handles the power rails, the status of the motor, emergency mode, checking the temperature, and driving the stepper sequence.

For the power rails in normal mode, they are to follow the LSB in the enable 12V rails register, but in the emergency mode, they need to be powered on to ensure that the shutter closes. Since the rails might be off to save power when the emergency mode sets in, it needs to be able to override the register. The easiest way to perform this is to check if the emergency state is not in the normal operation mode and override it should that be the case.

The shutter status register is the only way for the DPU to get to know what is going on with the motor within the RSE. The stepper driver places the corresponding signals from other processes into the bits in the register where they belong where it is possible. For the closing and opening in progress status bits, the state of the stepper process is checked if the state is in the closing or opening sequence. The emergency status is checked the same way the power rails is checked.

The test procedure needs to check if the motor temperature or the electronics temperature exceeds the maximum allowed temperature. This set two registers indicating if the motor or the electronics are too hot. This lets the sequence check if one of these registers is set so it can wait for cooling down.

The RSE might lose communication to the DPU, and should it an emergency mode is needed. A state machine handles the emergency mode. There is a normal mode state where the communication is working, and heartbeats are received. When a missing heartbeat is indicated with a signal, the state goes to the heartbeat missing state. This will power on the power rails, overriding the register controlling the rails normally. There if the heartbeat is received again, the state goes back to the normal state. If a second pulse is received as it will come one second after the missing heartbeat signal, then the emergency initiated signal is set, so the stepper driver process will start closing the shutter. When the shutter is closed by end switch or max number of steps, the finish motor command is sent back from the stepper process. The emergency initiated is then reset, and the emergency finished state is set. If it, however, receives a cancel command, it will stop the stepper process as normal, and the emergency state is set to the emergency finished, as a heartbeat is imminent as a cancel command was received from the DPU. In the emergency finished state, it will wait for the heartbeat missing signal be set low, before going back to the normal state.

The emergency is based on the missing heartbeat from the heartbeat module. So, the RSE needs a way to protect the detector if the DPU stops working and needs a reset. The rails need to be

powered on. Then the state takes a second to ensure the rails are properly turned on. This also gives some time to let a heartbeat come as a reaction on the power rails turning on and stop the emergency before it is initiated. Then in the close process, the closing procedure needs to respect the too hot condition and the max number of steps. This is to ensure that the RSE does not degrade and destroys itself, and thereby leave the detector exposed to high energy radiation or that the shutter is closed without the possibility of opening it again. The switches might also malfunction and get stuck to a value, so it can't be used to check if the shutter is closed. Then in the finished state, it will just check if the heartbeat is received before going back to normal operation.

The stepper driver is the process that goes through the stepper sequence. This process sits in idle with all the transistors off, before while waiting for a new command to come from the stepper control, or an emergency is initiated by the emergency process. When a new command is received, the counters in the settling time module, the half step module and the number of steps performed value is reset. This is to ensure that the number of steps register is kept at its value until a new command is performed. The flags from the stepper control are checked to find out what command is to be performed. The state is then set to the appropriate stepper sequence. There are two sequences, one for opening the shutter and one close sequence.

As described in 2.4.3, the stepper needs 16 states to reach the whole stepper sequence in both directions. The stepper sequence is built up so that there is a step zero, the normal steps from the stepper sequence, initial steps where signals to other modules are set, cooldown steps, a step nine, and an all off step. When started for the first time in a sequence or restarted after a cooldown, step 0 is used instead of step 1. This step is first in an initialisation step, before going into the main step. The main step runs until the half step is received, before going into a new step. There are initialization steps where there are changes to the transistors, and it lasts only one clock period. The main steps will send out the settling time reset when the half step is received when it is necessary to power on a new transistor.

In step eight, when the half step is received, different checks must be performed to check if the operation is finished. The test sequence is vital as specific conditions like temperature or cancels needs to take precedence over other conditions. First, the number of steps is increased here. This

is a simple way of ensuring that the number of steps is progressed, as the eighth step is the end step of a single sequence, and the start point might be in either step zero or step one.

The first check is to see if it was an emergency has been initiated. It will then check if it has been cancelled or if the end has been reached. If it was not an emergency that initiated the operation, or it is an opening operation that is being performed, the first thing to check is if the current operation is cancelled. This will send the state to the ninth step to finish the sequence before ending the operation, however, if the operation is not cancelled, which operation that is being performed needs to be checked so that the correct end conditions can be checked.

For the stop-at-switch commands, the switch is checked if it is set. If it is, the ninth step is the next to end the operation. If the switch is not set, the electronics and motor temp must be check if they are too hot, which in case the operation goes to the cooldown state. Else a new step is run. For the max number of steps operation, the number of steps is checked if the max number of steps is reached. If it is the operation goes to the ninth step to end, and if not, it must check the temperature and either go to the cooldown state or run a new step.

The cooldown states turn off all the transistors and then rechecks the temperature each half step. Then if the temperature has lowered to or under the max acceptable temp the cancel flag is checked if the operation has been cancelled during the cooldown. If not, it will go to step zero for a new step. This is since the only conditions that use the cooldown states is if a new step is to be performed. If the operation ends, the transistors are turned off and will cool down. As no new commands are allowed as the motor or electronics are too hot, so nothing will heat them.

When ending the operation, the ninth step is utilised. At the start, one transistor keeps low at the very first step, even though it should have been high, step nine is used to compensate for this. The final step is the all off step that turns of all transistors. This is used as a clean-up state where all the transistors are turned off. It provides a simple universal test for testing, as it is the end state for both the closing and opening sequence. This lets a check on the state make sure the operation was finished cleanly, as the only clean exit is going through this state.

# 6 Test and development

The RSE protocol was developed with these steps:

1. Development and testing of the communication with the DPU
2. Test of communication on board with a simple program
3. Development of motor and other modules
4. Further testing in a test bench
5. Expanded python program for testing on board
6. Testing on board

## 6.1 Development of the firmware

The communication was developed first as Bitvis provides a register tool to create a register bank with an SBI interface, and two test methods. This tool is called Bitvis Register Wizard. These were desired to be utilised in the project and came with an SBI and UART example. This did let a single test bench for the register bank be generated to make sure the register bank worked. In it, each register was accessed via an SBI interface and read. All writeable registers were written different values to and then checked.

As the register bank was working, the communication modules to access the register bank from the DPU was to be created. The RSE protocol was decided to be used, and so on a high-level module was created to fulfil this. To test this, we created a new test bench with the RSE protocol and the register bank. The RSE protocol is connected to the UART via two records. Two procedures were created to make a write operation and a read operation over the records. Having tested this, the UART was created to get the whole communications aspect ready. The test bench was expanded, as illustrated in Figure 23, and different procedures replicating the RSE protocol was created to perform the tests.

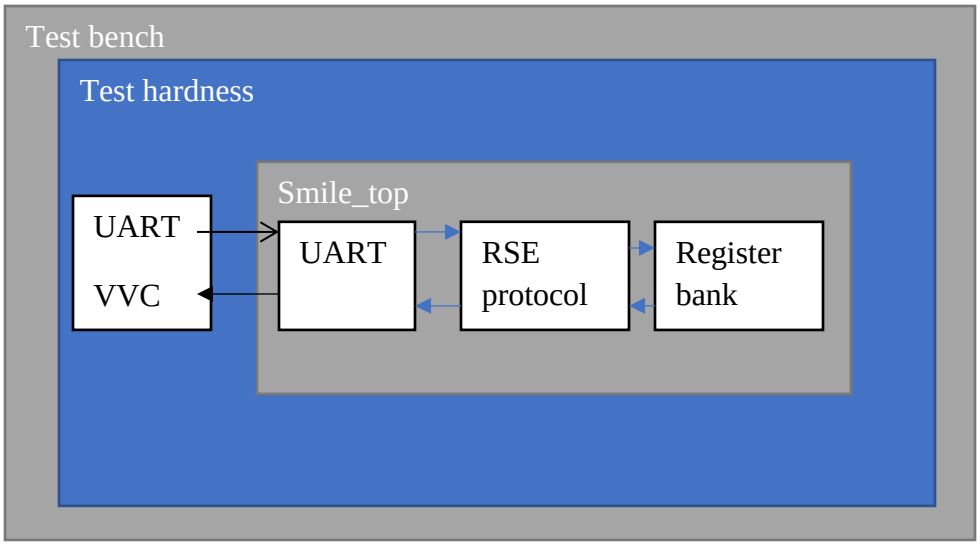


*Figure 23 Test bench for communication*

The Bitvis Universal VHDL Verification Methodology framework was incorporated into the tests, and their UART VHDL Verification Component (VVC) was put into the test bench. A testbench harness was used to instantiate the UART VVC, the RSE components, the engine for the framework and the clock process. This was done to reduce the size of the test bench file and separate the instantiation as it would not be modified a lot afterwards.

This was the fundament for the testbench that was expanded upon. In the test bench, the register default was read before the write operation was performed to all writeable registers. As the normal operation was working, the different illegal cases were tested to check if the corresponding error message were received. First, multiple read transmissions were sent before the RSE had time to respond. This does not work as the UART will receive the observed characters, but the RSE protocol ignores them as they come in during states that do not care about any new received signal. The only way this works is if the DPU initiate sending a word after the sequence number have started sending from the RSE. Then the UART will have received the whole word as the RSE protocol has had enough time to get back to idle. Then unknown commands are sent and rejected. The parity bit is changed in the VVC to use even parity. This makes it so that all communication from the DPU is rejected. This is then changed back to odd parity before the test bench tries reading and writing non-existing register and writing to read-only registers. At last, transaction timeouts is performed while writing both correctly and faulty. All of this worked in the simulation.

As the communication was working the functionality could be developed. The other functions, as described in Chapter 5, was developed. First, the functionality that did not affect driving the stepper motor directly was developed, and after that, the stepper driver modules were developed. These modules were continuously tested in the test bench by accessing the registers that controlled the functionality and from there check that the correct output was produced, probing different signals to check if they behaved as intended and at last we did see if the RSE behaved as expected in communication with the DPU. As all functionality is either dependent on commands from the DPU or running independently, we would trigger different functions by writing to the RSE, or wait for functions to trigger.

## 6.2 DPU simulator

Lylund created a DPU simulator to test his code. His thesis [3] gives the following description of the DPU simulator:

> "The DPU simulator is created in Python, which is a high-level, general-purpose programming language. Python has simpler syntax compared to the alternative programming languages: C, C++, and Java. It also supports a large variety of libraries and development tools, such as serial communication and Graphical User Interface (GUI). This enables complex programmes to be created in a relatively short time, so it was decided to use Python as the framework for constructing the DPU simulator" [3].

This software has been taken as the starting point of the present RSE GUI. During this work, the software has been restructured into several python modules, each with dedicated functionality. Some new functionality has also been added, for example, access to new registers, regular access to the heartbeat register and ability to open and close the radiation shutter multiple times sequentially. The latter functions needed a major software change in order to run continuously in the background, which was implemented employing the Python QTimer class, which provides repetitive and single-shot timers. The "Open Close Cycles" button sets a flag which triggers a software finite state machine controlling the open-close cycles.

The new GUI also implements the RSE communication protocol instead of the previous RoR, see Figure 24. This example shows how the heartbeat register is accessed every second, and from the log, we can see how the sequence number is incremented for each access.

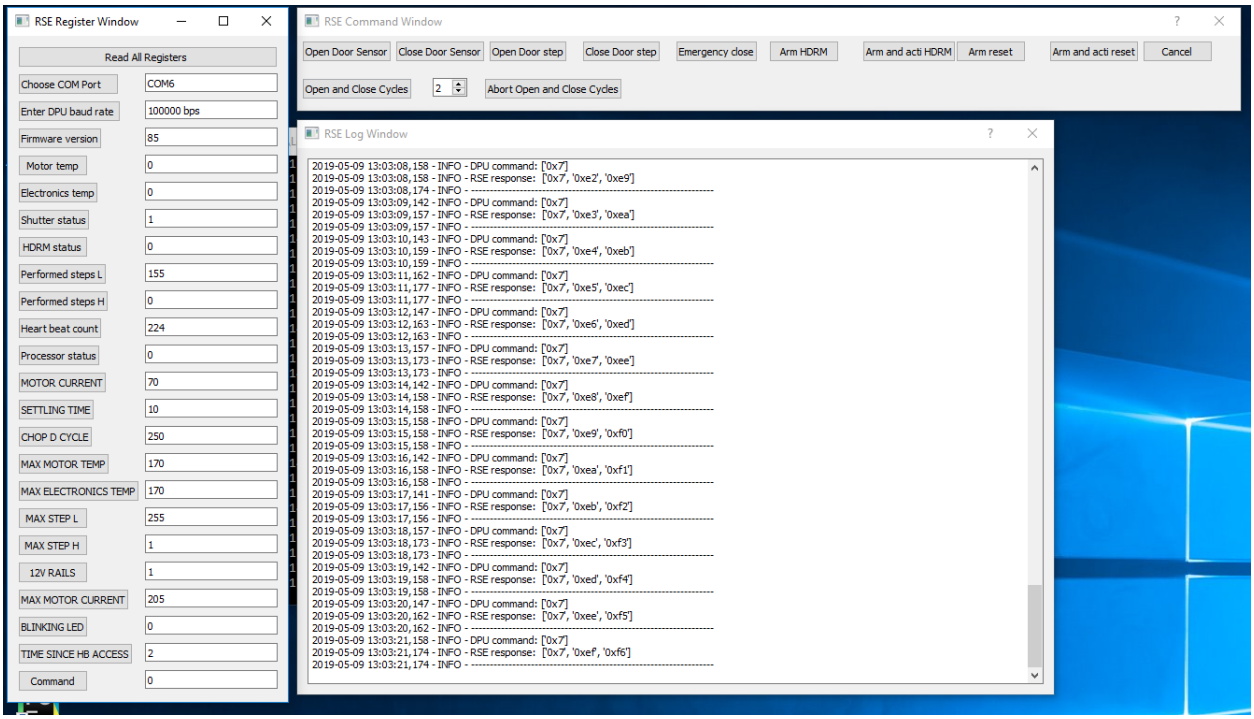


*Figure 24 Modified DPU simulator for operating and testing the RSE*

### 6.2.1 RSE Register Window updates

Previously we had to choose a register and then write a value or read it. Using the drop-down menu and writing to a register was tedious when trying to write to multiple registers. We changed every register label to buttons that read the register, and each register got a text box where we could write the desired value. Pressing enter triggers a new function that sends the proper command to the RSE. If the user writes an illegal number or a number not within 0 to 255, this now caught by an exception routine, preventing wrong values to be written to control registers or the software to crash. As previously mentioned, some new registers have also been added.

### 6.2.2 RSE Command Window updates

The RSE register window simplifies the process of writing commands to the RSE by specific command buttons. In principle, there should be a single button per command, but for the arm and

activate functionality, single buttons are needed. This is because it is difficult to click two buttons quickly enough to be within the required one second time window for the valid arming time, see Section 4.2.4. So, the button for arming and activating HDRM and reset was combined in the GUI for testing proposes.

In addition, as previously mentioned, an “open and close cycles” function was implemented. This function runs multiple open and close until end switch commands in succession. This allows long term testing of the hardware without manual intervention.

### 6.2.3 RSE Log Window updates

A terminal was used, but we wanted a log window instead. Using that we could easily connect the window to a file to read afterwards. With the extensive RoR communication protocol, the whole process of a single write or read were shown. The RSE protocol allowed a simpler log message as we just printed the characters that were sent to the RSE and the response message. A comment was added in case of an error or for commands. The sequence number would also increase on the screen so we could see if any packages were lost.

When we implemented the emergency close feature on the FPGA, we had to create a background function that provided heartbeats to the RSE. This function stopped the RSE from closing the radiation shutter after 30 sec while testing the RSM.

## 6.3 Hardware tests

After thorough simulations, the firmware is ready to be tested on the physical system. The EBB is connected to the FPGA development board, and the RSM is attached to the RSE via the RSE-RSM harness.

To put the firmware onto the FPGA, it must be compiled. First, a tool like Libero synthesises the code. Synthesis is a process that analyses the code and finds syntax errors (language grammar error) and static semantic errors (ex. wrong type assignment). Compiling the code generates an executable object code [20]. These languages contain a sequential list of statements.

During the compilation of code, the constraints of the code are added. An I/O constraints file is needed. This file makes the FPGA set inputs and outputs to desired pins on the FPGA. The FPGA pins are then connected to components on the PCB board. This way, for example, standard I/O pins, LED or switches can be accessed from the FPGA. On a microcontroller, in contrast, there are set registers that correspond to hard-wired components, and the registers cannot be chosen to correspond to a different pin. [27]

In addition, there is a timing constraints file that is necessary. “Timing constraints represent the performance goals for your designs. The software uses timing constraints to guide the timing-driven optimisation tools in order to meet these goals” [28]. These constraints are set to give information to the tool about the delay from ports, minimum speed of the clock domain and identify critical logic paths that need a maximum delay or more clock cycles [28]. This is used when the tool generates the floorplan of the FPGA and decide where to put the logic. By putting logic and nets around the floorplan, the delay and timing constraints can reach a satisfactory level.

A floorplanning constraints file can be used to create regions on the FPGA and assign logic and nets to these regions [28]. This is practical as modules that belong together can be placed close to each other. A set floorplan will also allow a set space for different systems, where there are conflicts over the usage of the area. A set floorplan also enables reconfiguration of certain blocks when desired. As our system are rather small, and it will be burned into an FPGA, a dedicated floorplan was not used.

The firmware was put onto a ProASIC 3E, and the DPU simulator was connected through I/O pins. We did also compile the firmware for use on the RTAX250 FPGA. On that, we would use 40% of the combinational cells and 36% of the sequential cells, ending up with a total of using 39% of all the cells.

### 6.3.1 Testing of communication on board

A blinking-LED module is included as a firmware module to ease the first test. The main idea is that the I/O pins toggles regularly after power-up and reset, and the blinking can be inhibited by writing to a control register. Proper firmware operation can thus be verified immediately by observing the blinking LED activity, and proper communication link operation can be confirmed

by writing to the inhibit register. The FPGA I/O pin can either be connected to a physical LED or simply to a test pin that can be probed.

### 6.3.2 RSM bench test

The main functionality of the RSM can be verified by running the motor in order to open or close the radiation shutter. Full functionality is simply verified by commanding “open door stop at end” or “close door stop at end” while observing the movement of the door leaf. In addition, the motor parameters should be tuned by setting the optimum motor current, half-step settling time and the chop duty cycle. The figures gathered from the oscilloscope, view the current through one of the coils driving the stepper motor where the current will go in one direction for 3 out of 8 half steps before switching off the current for one half step, and then turning it one in the other direction for another 3 half steps as illustrated in Figure 9.

### 6.3.3 Settling time

At the onset of the coil current, there is a dedicated “Settling time”, allowing the current to reach the desired value before current-chopping is started. As can be seen from Figure 25, the needed settling time is very short, about 40 µs before reaching the current plateau. The initial current peak shown in the figure is related to the speed of the current regulation circuitry, specifically the speed and the slew-rate of the opamp. At the start of the step, the current control is regulated to maximum, before the set current is reached, and it takes a few microseconds before the speed and the precision of the current regulator.

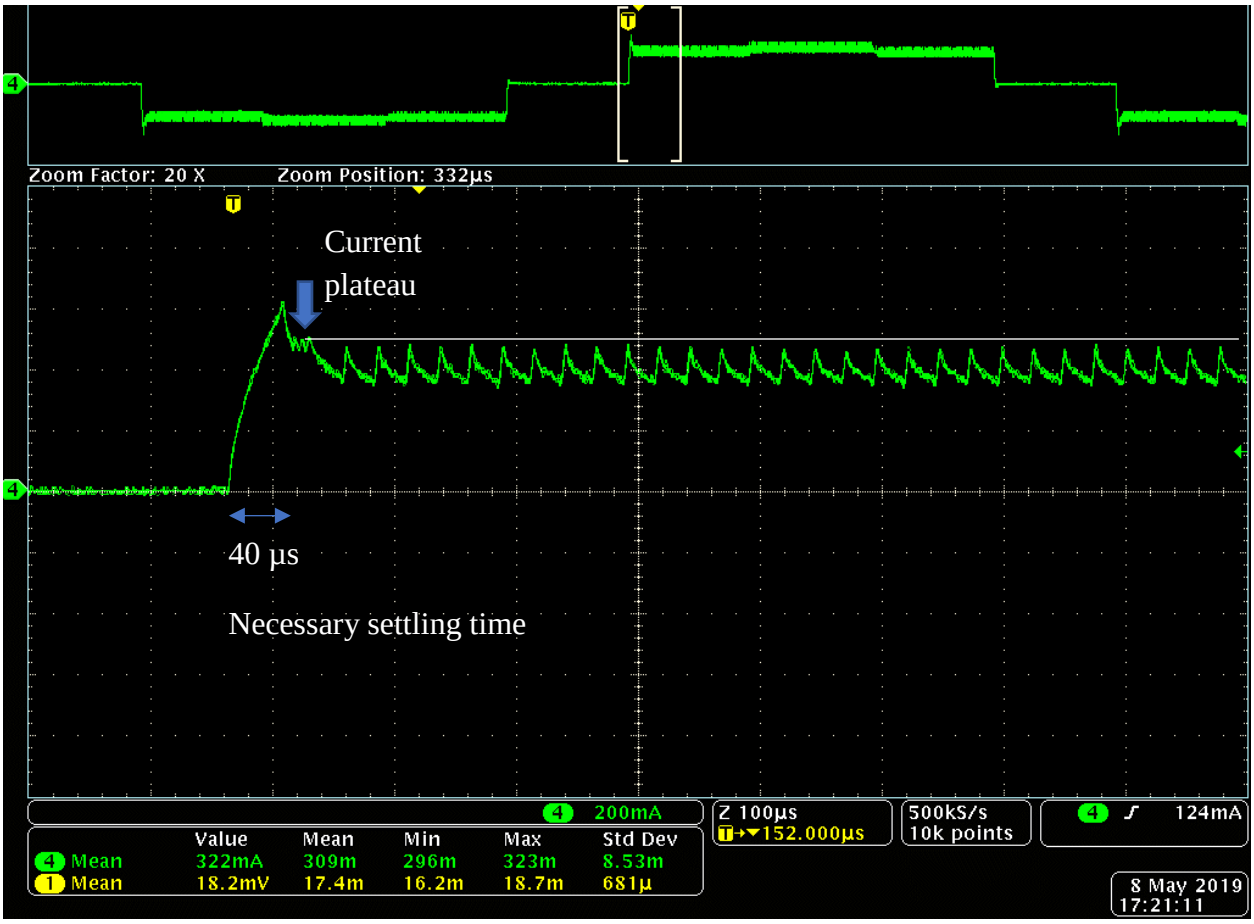


*Figure 25 Ideal settling time peak*

The set settling time has very little influence on the motor torque, as described in Section 2.4.3. Therefore, it can be set as low as possible in order to save power by keeping the chopping period as long as possible for every current phase. An experiment can be done in order to verify the power saving effect of lowering the settling time. Trying to settle with a value of 10 gives us a settling period of 40 µs, and a register value of 50 corresponds to a settling period of 200 µs. Using 10 instead of 50 represents a settling time difference of 160 µs per step. A step is 2 ms, and the power saving effect can be calculated if we assumed negligible power consumption during the chopping period:

$$\frac{10 * 4\mu s}{50 * 4\mu s} = \frac{160\mu s}{2000\mu s} = 8\%$$

There will be 8% saving of power using the lower settling time. Observing the power supply meter, we saw a change from 58 mA for 200 µs settling time and 51 mA for 40 µs, representing a change of about 12%. Too high settling time can be seen in Figure 26.

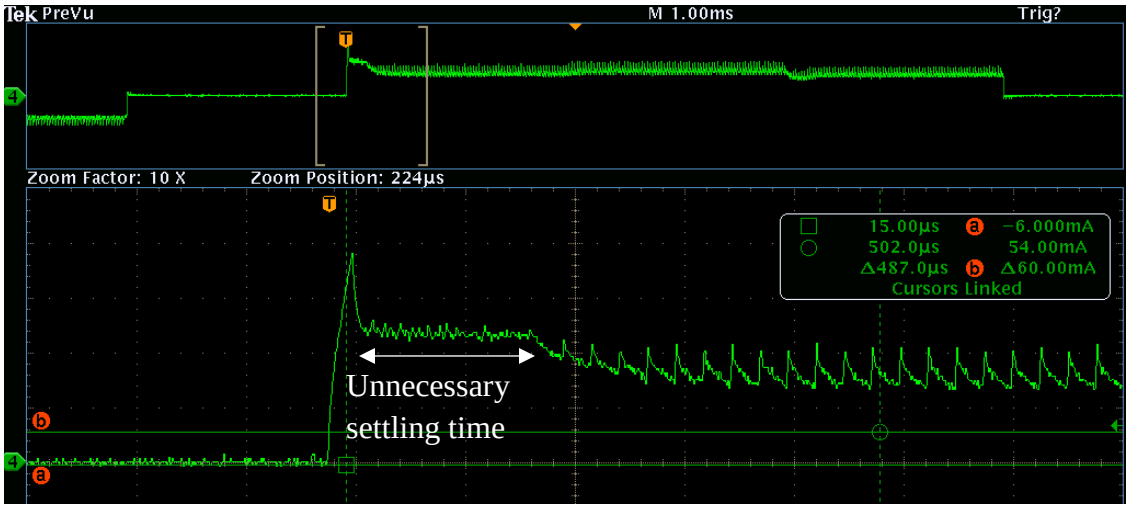


*Figure 26 Too high settling time peak*

We also tried setting the settling time to 0, as shown in Figure 27. This had no settling time, and therefore, chopping started immediately. With a high enough chopping and current, we could still reach the plateau.

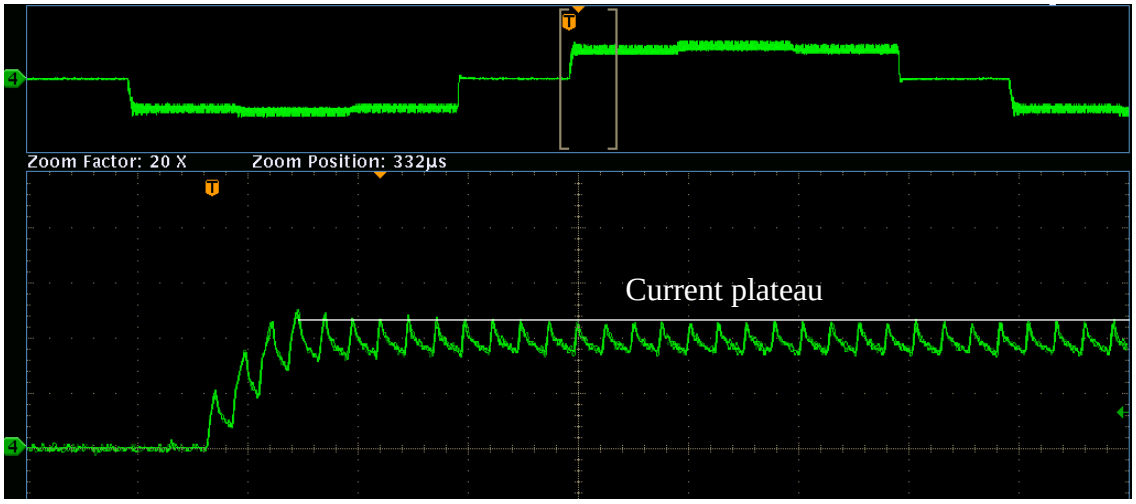


*Figure 27 Zero settling time peak*

The result of the settling time measurements is listed in Table 15, and the actual motor current is drawn in Figure 28, and the power supply current is drawn in Figure 29.

*Table 15 Settling time test (Motor current 100, Chop duty cycle 150)*

| Settling Time | Actual Motor Current (mA) | Power Supply Current (mA) |
|---|---|---|
| 0 | -- | 125 |
| 20 | 400 | 139 |
| 50 | 415 | 145 |
| 100 | 420 | 165 |
| 150 | 425 | 180 |
| 200 | 425 | 200 |
| 250 | 425 | -- |

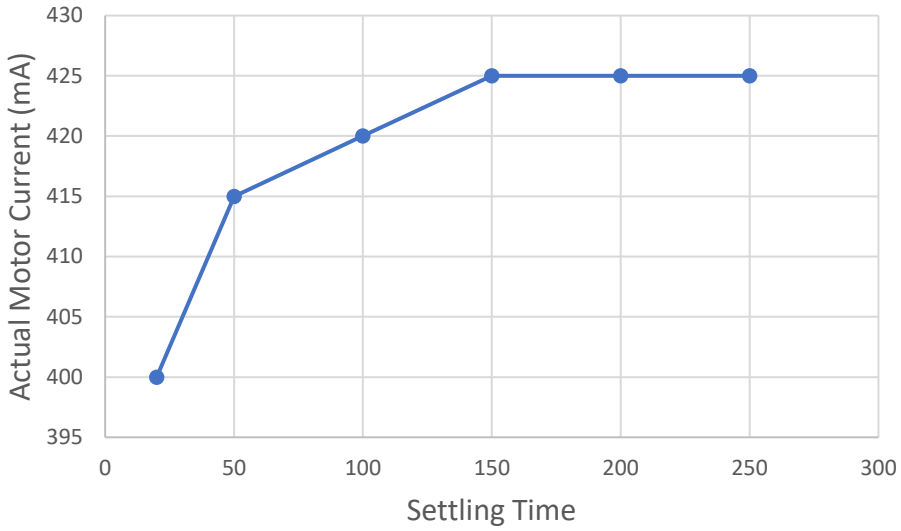


*Figure 28 Motor current as a function of settling time*

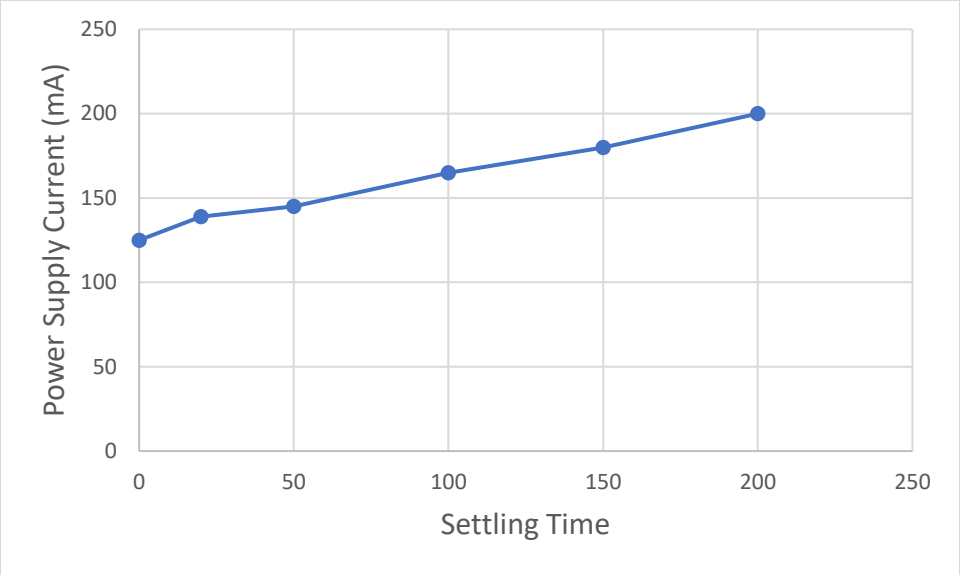


*Figure 29 Power supply current as a function of settling time*

### 6.3.4 Chopping

As mentioned in 6.3.3, the settling time allows the current to reach the desired value before current-chopping is started. The chopping signal is constantly going, but the settling time signal overrides it, by locking the output to 0 for as long as desired. After the settling is done, the chopping controls the current. The chopping lets the transistors be charged and discharged perpetually. The recharge needs to reach the current plateau again. As can be seen in Figure 30, the current is chopped where the current falls off, before being charged back up again. This process moves the average current down. In Figure 31, the same operation is done without chopping.

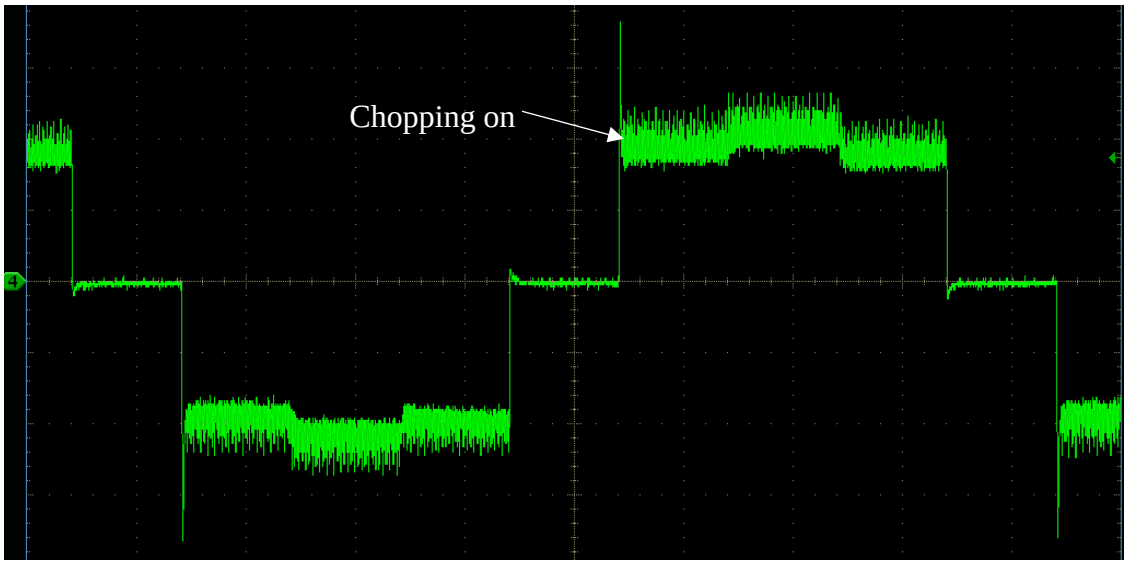


*Figure 30 Normal chopping*

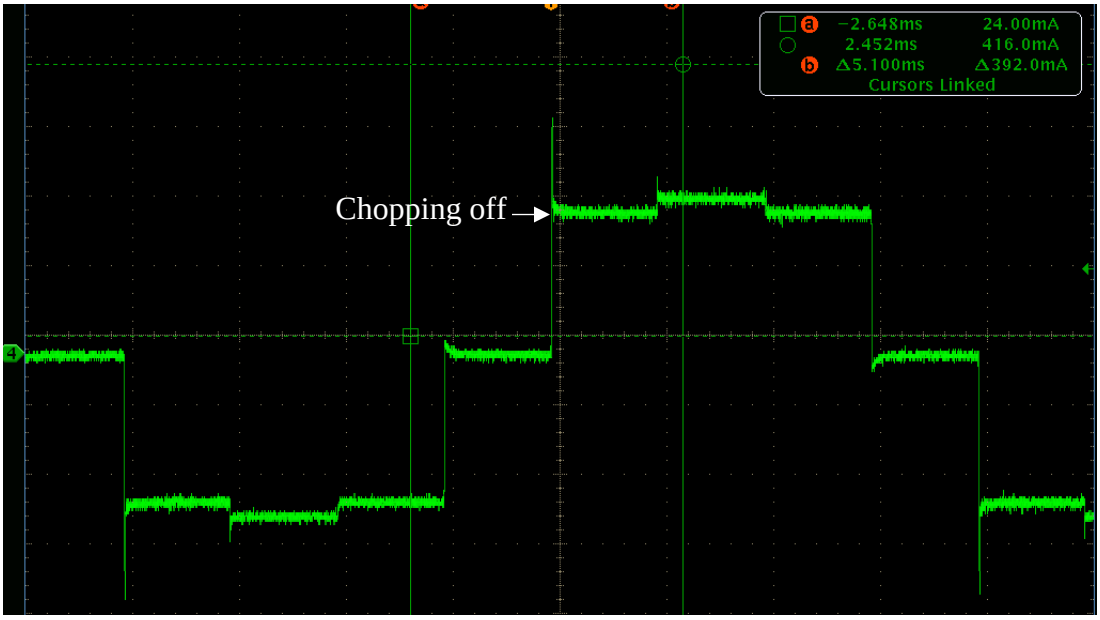


*Figure 31 Without chopping*

Settling time is only active for the ramp-up of the current. The current is then primarily set by the chopping and the motor current register. The chopping needs to be high enough so that the circuitry can recharge itself, as seen in Figure 25. The current needs to reach the plateau again, or the current will fall off, as seen in Figure 32.

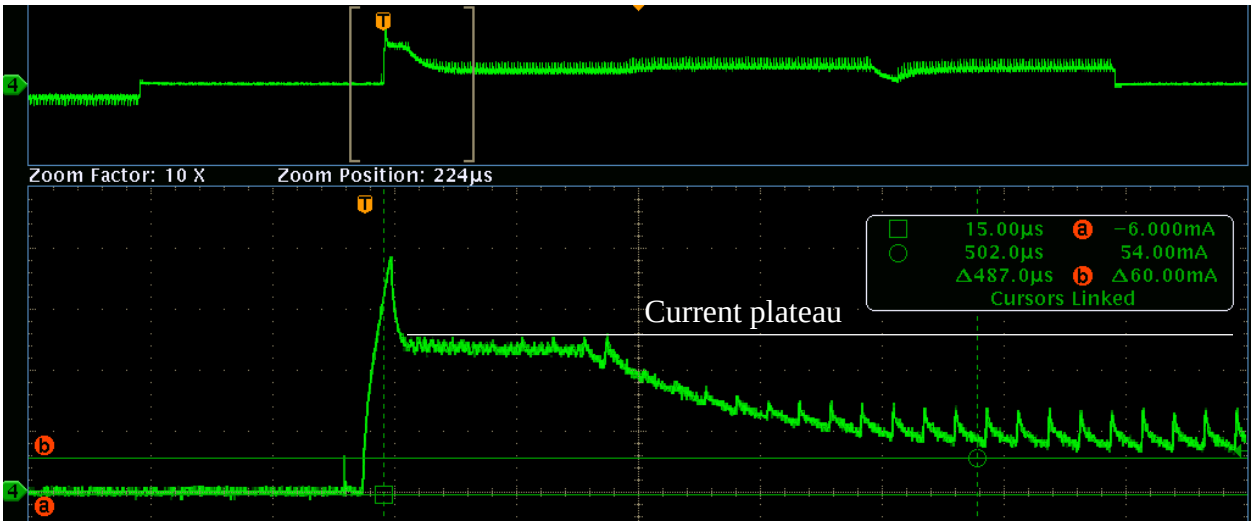


*Figure 32 Too low chopping*

The chopping needs to be set so that the circuitry has time to recharge the transistors. The optimum chopping will only just reach the current plateau again before letting the transistors discharge again. Staying at the current plateau is counterproductive as we want to minimise power usage. Looking at Figure 33 and Figure 34, we can see that the effective time duty cycle where the current ramps up are 2 µs of 25 µs.

$$\frac{2\ \mu s}{25\ \mu s} = 8\%$$

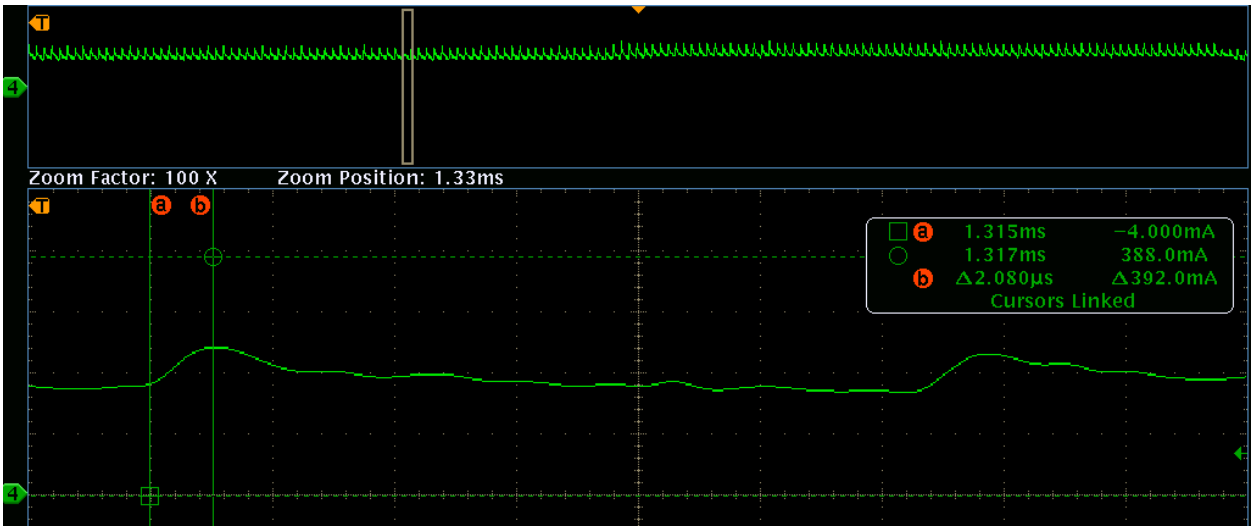


*Figure 33 Effective on-period of one chop*

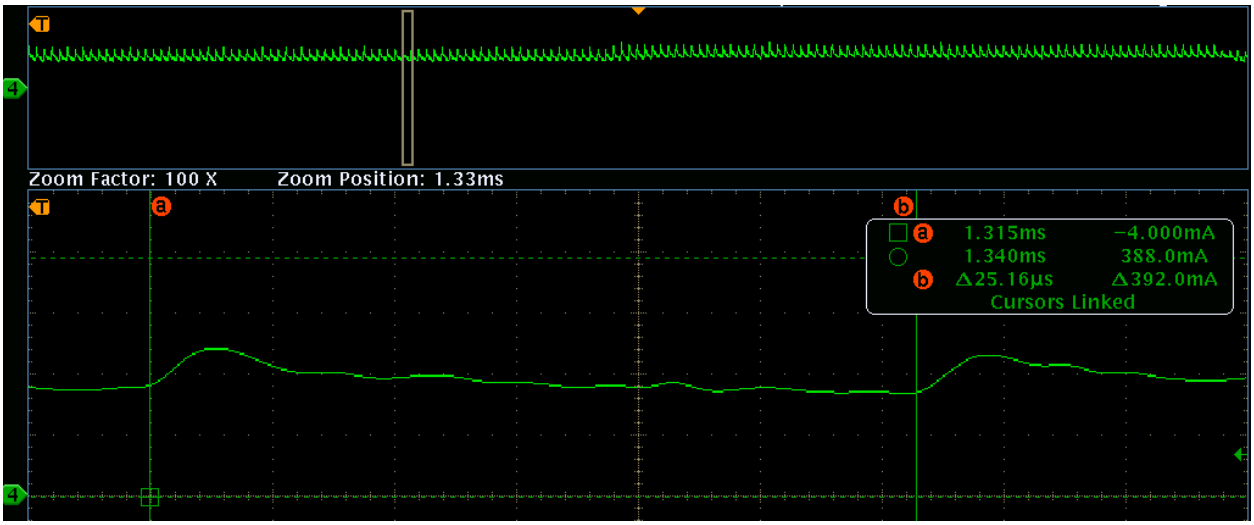


*Figure 34 Whole period of one chop*

The test results are listed in Table 16. At lower chop duty cycles, settling time created a current before the transistors lost the charge, as shown in Figure 32. At a chop duty cycle value of 123, the shutter did get a high enough current to open the shutter. As seen in Figure 35, the current ramped up in between 120 and 130. However, looking at Figure 36, the current drawn from the power supply only increased while the motor current stopped just before 450 mA at a chop duty cycle of 140. Comparing the two graphs, we can see that we will use unnecessary power at higher chop duty cycles.

*Table 16 Chop duty cycle test (Motor current 100, Settling time 50)*

| Chop D Cycle | Actual Motor Current (mA) | Power Supply Current (mA) |
|---|---|---|
| 100 | 0 | 23 |
| 110 | 0 | 24 |
| 120 | 57 | 30 |
| 122 | 164 | 40 |
| 123 | 244 | 50 |
| 124 | 280 | 63 |
| 125 | 312 | 70 |
| 130 | 396 | 99 |
| 140 | 444 | 124 |
| 150 | 448 | 140 |
| 160 | 448 | 181 |
| 180 | 456 | 285 |
| 200 | 440 | 390 |

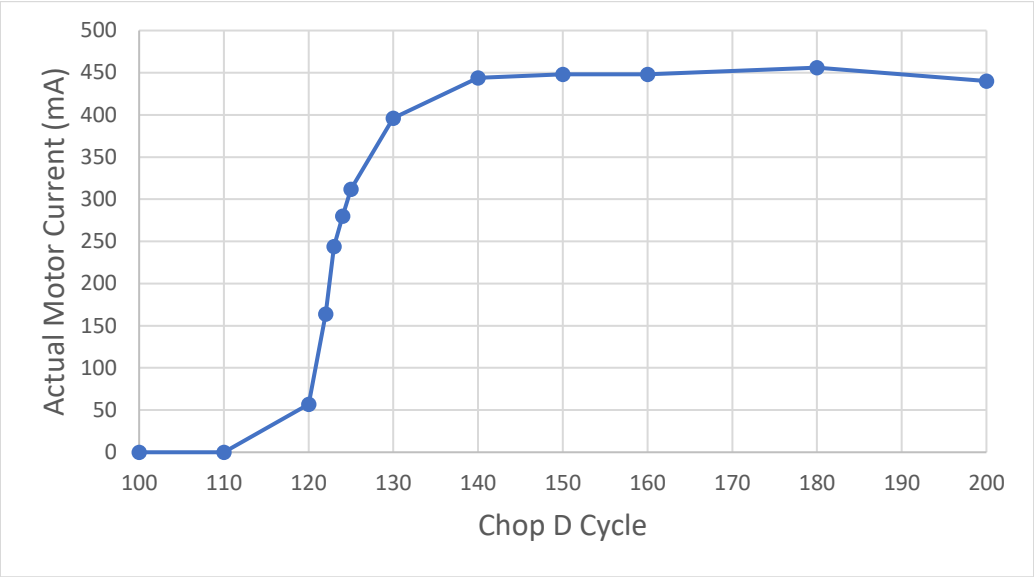


*Figure 35 Motor current as a function of chop duty cycle*

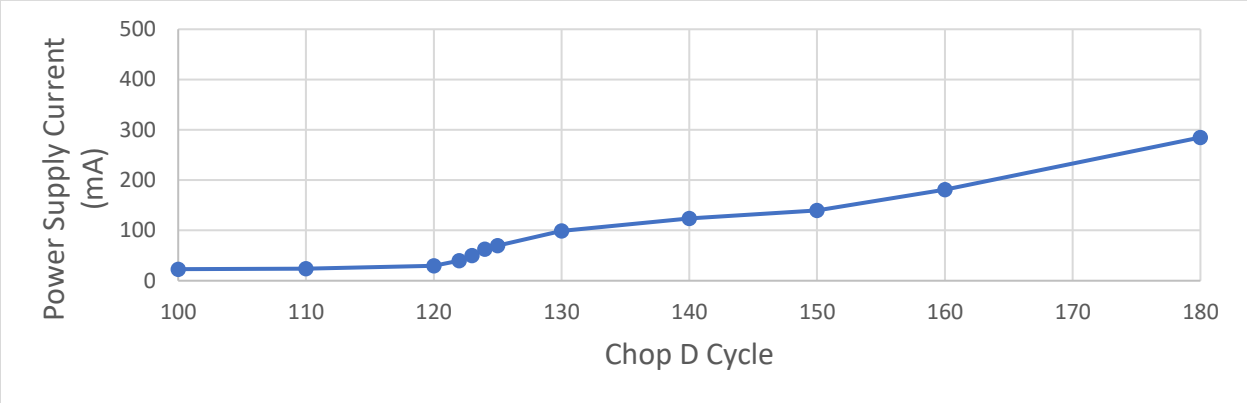


*Figure 36 Power supply current as a function of chop duty cycle*

### 6.3.5 Motor current

The motor current is the main driver of the current. This sets the current plateau that the chopping and settling time should reach. As can be seen in Table 17, the current increases linearly with the motor current register. From these results, Figure 37 and Figure 38 with trendlines were made. We could see a linear increase in current as we increased the register value.

*Table 17 Motor current test (Chop D cycle 150, Settling time 50)*

| Set Motor Current | Actual Motor Current (mA) | Power Supply Current (mA) |
|---|---|---|
| 50 | 205 | 60 |
| 75 | 335 | 105 |
| 100 | 408 | 145 |
| 125 | 508 | 190 |
| 150 | 608 | 230 |
| 170 | 680 | 275 |
| 175 | 715 | 280 |
| 180 | 720 | 290 |
| 200 | 800 | 340 |

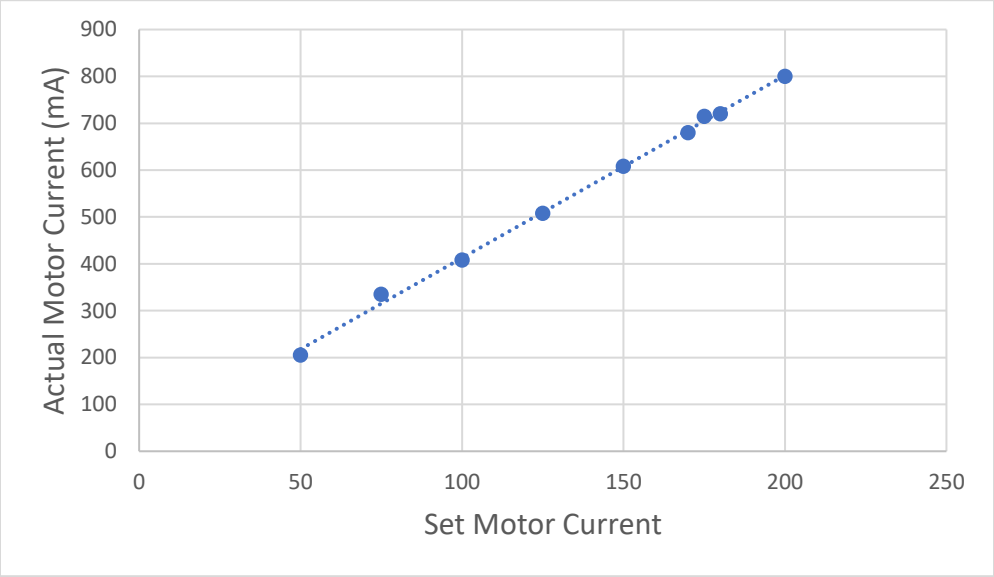


*Figure 37 Actual motor current as a function of set motor current with trendline*

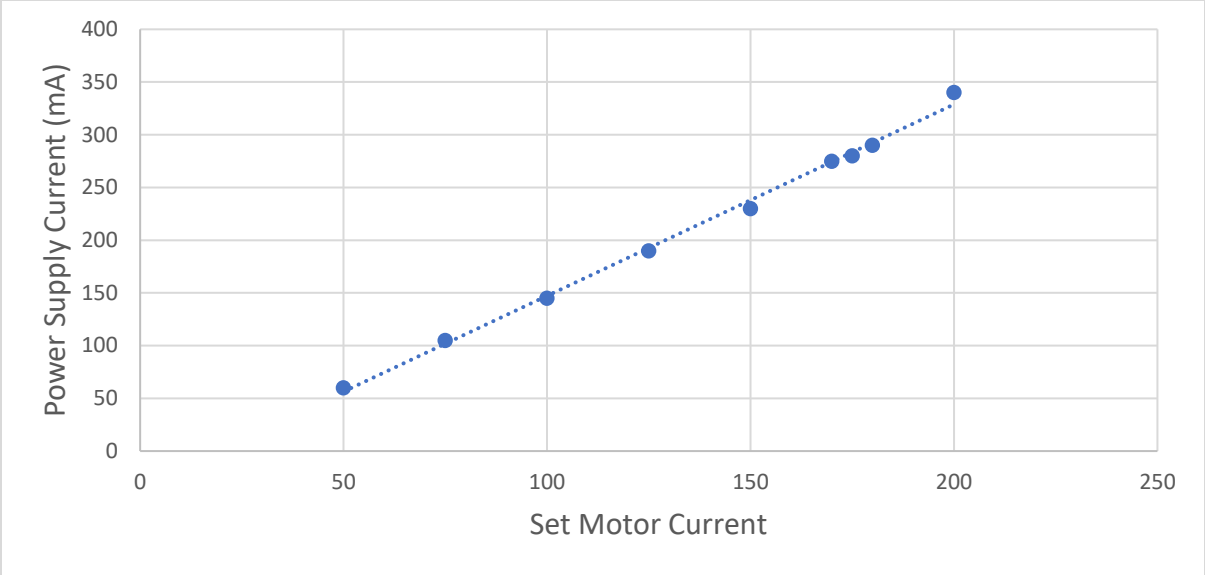


*Figure 38 Power supply current as a function of set motor current with trendline*

# 7 Summary and conclusion

The system was developed as a second stage of the RSE. The requirements that are explained in chapter 2 were adapted from the first iteration described in Lylund [3] and the design document of the RSM [7], and edited when needed, to work with the FPGA version of the RSE. In the first iteration, Lylund and the SMILE team created a breadboard that was able to run a simple stepper motor. We ended up with an elegant breadboard of the RSE that is run with an FPGA on a development board. The elegant breadboard is almost ready to be taken to the next stage as only some small alterations to the RSE was requested at a late stage of this work. The only main missing part of the RSE is the temperature readings. An SPI needs to be created to access the analogue-to-digital converter that reads out the temperature of the motor and the driver electronics. The allowed temperature for the motor is in a range of 220, so the register can be tuned to have a resolution of almost 1 degree as the register range is 256.

In this work, we started with a quick project on the previous communication protocol, before discarding it early on in favour of developing a simpler protocol. The RSE protocol that is described in Section 4.6 was developed to work with a register bank and a UART while providing far less overhead than the previous protocol. The communication with the register bank and the debugging module was implemented first to get a test on board done as soon as possible. When this was working, the rest of the functionality from Chapter 4, like the control of the stepper motor was developed and implemented into the system. Each module has been individually created and place into the system, where new tests were added to check that the new module behaved as wanted. We have

This new functionality was continuously tested as it was developed with a test bench in Questa. As we learned more about the system and methods, we discovered new problems and cases that may fail us. As a result of this, we finalised and created new requirements to fit the problems and corner cases that arose. Many of these issues are logged in the git revision tool. For example, we had the settling time set in steps of microseconds. This proved to be too small when we tested it on the board, so we changed it to use steps of 4 microseconds instead. Developing the system, we have taken into consideration what can go wrong, like the end switches stop working, and designed an alternative that can go around the problem.

The system has been designed with modulation in mind. We have implemented all modules internally, and only the register bank has been generated from a tool. No intellectual property cores have been used in the design. All modules have been designed to be as independent as possible.

As we can see in the results in Section 6.3, we achieved the same characteristics that were reached in [3]. Motor current is the primary factor in setting the desired current in the coils to operate the stepper motor. The chopping will save current as long as it high enough to not let the current drop off. Setting the settling time higher than necessary will only waste power, and we do lose some inductor current going by not having a settling time. Ideally, we would want the chopping the be able to hold the inductor current constant, but just bearly. As for the settling time, we want it to get the correct inductor current as fast as possible, but also allow chopping to start once the current is in place. Then the motor current register can be used to adjust the current.

# References


[1] NASA, “Near Miss: The Solar Superstorm of July 2012,” NASA, 23 07 2014. [Online]. Available: https://science.nasa.gov/science-news/science-at-nasa/2014/23jul_superstorm. [Accessed 14 05 2019].

[2] ESA, “Smile summary,” ESA, 06 03 2019. [Online]. Available: http://sci.esa.int/smile/59137-summary/. [Accessed 24 04 2019].

[3] O. Lylund, “Design and Development of the SMILE SXI,” University of Bergen, Bergen, 2018.

[4] SMILE team, “SMILE Instrument Interface Control Document - ICD Soft X-ray Imager - SXI [Internal rapport],” University of Bergen, Bergen, 2016.

[5] B. Mauk, N. Fox, S. Kanekal, R. Kessel, D. Sibeck and A. Ukhorskiy, “Science Objectives and Rationale for the Radiation Belt Storm Probes Mission,” *Space Science Reviews*, pp. 3-27, 7 09 2012.

[6] ESA, “SMILE instruments,” ESA, 26 02 2019. [Online]. Available: http://sci.esa.int/smile/59140-instruments/. [Accessed 1 5 2019].

[7] BCSS, “SMILE SXI Design Report for RSM 002,” University of Bergen [Internal Report], Bergen, 2018.

[8] A. Merstallinger, M. Sales, E. Semerad and B. Dunn, “Assessment of Cold Welding between Separable Contact Surfaces due to Impact and Fretting under Vacuum,” ESA Communication Production Office, Noordwijk, 2009.

[9] R. Baumann and K. Kruckmeyer, “Radiation handbook for electronics,” Texas Instrument, Dallas, TX, 2019.

[10] SMILE team, “FPGA and Firmware Requirements Specification for RSE [Internal rapport],” University of Bergen, Bergen, 2019.

[11] Microsemi, “RTAX-S/SL and RTAX-DSP Radiation-Tolerant FPGAs,” Microsemi Corporation, Aliso Viejo, 2015.

[12] T. Hopkins, “AN235 Application note Stepper motor driving,” 14 11 2012. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/57/c8/7c/c1/0d/91/46/89/CD00003774.pdf/files/CD00003774.pdf/jcr:content/translations/en.CD00003774.pdf. [Accessed 01 05 2019].

[13] N. H. E. Weste and D. M. Harris, CMOS VLSI Design, Boston: Pearson, 2010.

[14] M. Wirthlin, “FPGAs operating in a radiation environment: lessons learned from FPGAs in space,” IOP Publishing for SISSA MEDIALAB, Oxford, 2013.

[15] S. Heath, Embedded Systems Design, Oxford: Newnes, 2003.

[16] M. Gudino, “What is a Microcontroller?,” Arrow Electronics, 26 02 2018. [Online]. Available: https://www.arrow.com/en/research-and-events/articles/engineering-basics-what-is-a-microcontroller. [Accessed 03 05 2019].

[17] Microchip, “ATmegaS128 - Datasheet,” 13 12 2017. [Online]. Available: https://www.microchip.com/wwwproducts/en/ATmegas128. [Accessed 26 04 2019].

[18] US Defense Logistics Agency, “PERFORMANCE SPECIFICATION INTEGRATED CIRCUITS (MICROCIRCUITS) MANUFACTURING GENERAL SPECIFICATION FOR DEPARTMENT OF DEFENCE UNITED STATES OF AMERICA,” 06 12 2018. [Online]. Available: https://landandmaritimeapps.dla.mil/Downloads/MilSpec/Docs/MIL-PRF-38535/prf38535.pdf. [Accessed 26 04 2019].

[19] Arrow Electronics, “FPGA vs CPU vs GPU vs Microcontroller: How Do They Fit into the Processing Jigsaw Puzzle?,” Arrow Electronics, 5 October 2018. [Online]. Available: https://www.arrow.com/en/research-and-events/articles/fpga-vs-cpu-vs-gpu-vs-microcontroller. [Accessed 03 05 2019].

[20] K. L. Short, VHDL for Engineers, Harlow, UK: Pearson Education Limited, 2014.

[21] C. Melonfire, “Understanding the pros and cons of the Waterfall Model of software development,” TechRepublic, 22 09 2006. [Online]. Available: https://www.techrepublic.com/article/understanding-the-pros-and-cons-of-the-waterfall-model-of-software-development/. [Accessed 03 05 2019].

[22] Tutorialspoint, “SDLC - Waterfall Model,” Tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm. [Accessed 28 04 2019].

[23] M. Poppendieck and M. A. Cusumano, “Lean Software Development: A Tutorial,” *IEEE Software,* pp. 26-32, 26 06 2012.

[24] C. Ebert, P. Abrahamsson and N. Oza, “Lean Software Development,” *IEEE Software,* p. 23, 21 08 2012.

[25] Sparkfun, “I2C Sparkfun,” Sparkfun, [Online]. Available: https://learn.sparkfun.com/tutorials/i2c/all. [Accessed 24 04 2019].

[26] P. Horváth, “SMILE SXI PSU - Regular Bytestream DAQ Protocol Definition,” Budapest, 2018.

[27] DIGILENT, “What is a Constraints file,” DIGILENT, [Online]. Available: https://reference.digilentinc.com/learn/software/tutorials/vivado-xdc-file. [Accessed 20 05 2019].

[28] Microsemi, “Libero SoC v11.6 Users Guide,” 9 2015. [Online]. Available: https://www.microsemi.com/document-portal/doc_download/130850-libero-soc-v11-6-user-s-guide. [Accessed 20 5 2019].

[29] P. Horváth, “SMILE SXI PSU - RMAP over RBDP Protocol Definition documentation,” Budapest, 2018.

# Appendix

## RMAP over RBDP protocol

RoR stands for Remote Memory Access Protocol (RMAP) over Regular Byte stream DAQ Protocol (RBDP). The protocol came about by implementing the RMAP on top of the RBDP to create an interface that fulfils the responsibilities of the slave. This protocol lets the creates a read and a write operation to access the registers in the slave.

The RoR protocol defines a protocol that uses the UART at low-level, but it uses 9 data bits instead of 8. Otherwise, the character level is the same as in the RSE protocol. The package level is where the RoR protocol is different from the RSE protocol.

At the packet level, the RoR protocol operates with query packets and response packages. Only the master can generate query packages, and the targeted slave creates a response package. There are five different query packages: Invocation, instruction, register address, write data high and write data low. Each query has a unique header telling the slave which query is sent, and then the remaining five bits provide addresses R/nW and data to write. The MSB in all characters from the master is set to 0 to indicate this character comes from the master. In the response characters, this is 1. This is done so that other slaves on the communication line can easily check if it is a master command that needs to be check if they are going to respond to, or if it is a slave responding.

The response from the slave consists of three characters. The first is the confirmation character, confirming the master that the package has been received. The header of the query character is repeated and the second to the LSB indicate if a timeout happened. The LSB tells if the query was rejected. The next character will contain either the register content that has been read, no data or the error message that facilitated the rejection. The last character contains a CRC number calculated from the two first characters using the polynomial $g(x) = x^8 + x^2 + x + 1$.

Out of this a read operation and a write operation is created. The read operation uses the invocation character, follows with the instruction character and ends with the register address character. This lets the data from the register come in response to the register address character. The write operation goes all the way by using the write data high and write data low characters also. More about the RoR standard can be found in [29] and [26].

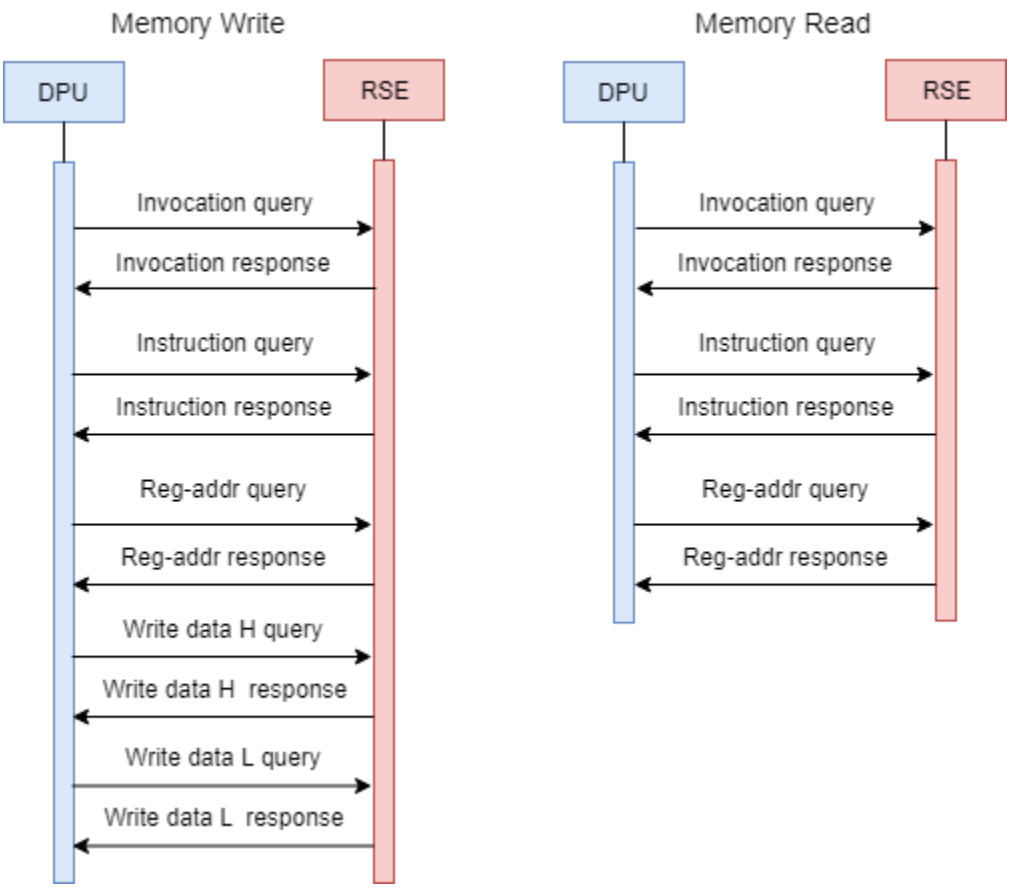


*Figure 39 RoR memory write and read*

### *Timing*

One baud delay is required before any response from the slaves or new transaction from the master can be initiated. The delay is necessary so that all the slaves can recognise a silent channel, and then they can prepare to receive an invocation. By demanding a quiet channel before a new invocation can take place, a continuous transaction between the master and one of the slaves can happen. The other slaves can sleep and don't have to check on each character from the master. The slave also needs some slack, so it has time to prepare for the response. The RoR was developed with software in mind, so this provides the software with some time to react to the character that has been received and act upon it.