Implementation of a Data Handling System for a Scientific Magnetometer on a CubeSat

Theodor-Adrian Stana

Master of Science Thesis
Stockholm, Sweden 2012
Implementation of a Data Handling System for a Scientific Magnetometer on a CubeSat

June 26, 2012

Author: Theodor Stana

Supervisor: Nickolay Ivchenko

Master of Science Thesis
Stockholm, Sweden 2012
XR-EE-SPP 2012:005
Abstract

Since their invention in 1999, CubeSats have become a widespread standard for small picosatellite missions. CubeSats allow for quick development of satellite payloads and launch in space without the high costs of a normal satellite. Emphasis during the CubeSat design process is placed on use of commercial-off-the-shelf (COTS) components and reuse of previously-designed units.

This report describes the interfacing of a scientific magnetometer, the Small Magnetometer in Low-Mass Experiment (SMILE) to such a CubeSat mission, the Space Weather using Ion spectrometers and Magnetometers (SWIM). Design of a complete platform for use in multiple such missions is presented here.

Modularity is one of the key aspects followed in the course of the work. A new board containing the analog pick-up and compensation circuitry for SMILE has been designed to fit inside a CubeSat frame. Additionally, the board contains circuitry for temperature measurements and gravity-gradient boom deployment. Modularity on the board is assured via short-circuit resistors, which can be soldered in case features are needed.

A full communication protocol has been developed and is presented as part of this work. Hardware implemented in an FPGA is used for filtering of compensation signals and storage to a Flash memory chip on the SMILE board. A modular, reusable and adaptable software stack for the flight microcontroller unit (FMCU) has been implemented for communicating to the SMILE instrument. The stack has been designed to be usable with different processors and communication interfaces. It can be run under an infinite-loop type application using interrupts, or as part of a real-time operating system task.
Acknowledgements

The author would like to thank the supervisor of this work, Dr. Nickolay Ivchenko, for the attention put into this project, his multiple advices and guidance offered in the course of the project. The discussions with him really helped me understand and learn things, and I am grateful for his patience.

Next, special thanks go to my family, my father Virgil, my mother Ecaterina and my fiancée, Ramona, for their unbridled support and energy they have given me during the course of this project. Without your encouragements, I would have found it hard to gather the strength necessary to pull through at some difficult times.

Another person that is due special acknowledgements here is Dr. Hien Bich Vo. I would like in this way to thank him for putting me up during my 3-week period spent in Puerto Rico, and for showing me as much of the beauty of the island as possible. Special thanks go to Hugo Pastrana, with whom I have had the pleasure of working and who has done some of the testing on the software stack. I would also like to thank him for the amazing trips around Puerto Rico he has arranged and the times we have had together. I have also had the pleasure of working and enjoying the company of Norberto Rios, Francisco Rivera, Ana Espinal-Mena and Christian Morales Otero. Discussions with them have truly been helpful and enjoyable and I am looking forward to meeting them again in the near future.

Last, but not least, I would like to thank here my colleague and friend Moysis Tsamsakizoglou, for the interesting discussions that we’ve had over coffee, for the opinions offered when asked, and for opposing and giving great feedback to this thesis. I trust we will do this again in the future.
5 FPGA Hardware Design

5.1 Timing in the SMILE Design ........................................ 32
5.2 DAC Output Signal Filtering ........................................ 32
  5.2.1 Filter Specifications ........................................... 34
  5.2.2 Matlab’s FDATool ............................................. 35
  5.2.3 Filter Coefficients ............................................. 36
  5.2.4 Hardware Implementation ....................................... 37
5.3 Communication Protocol Implementation ............................. 41
  5.3.1 The \texttt{spi\_slave} Block ...................................... 41
  5.3.2 The \texttt{chksum} Block ...................................... 42
  5.3.3 The \texttt{spi\_ctrl} Block ...................................... 43
5.4 Memory Storage ................................................... 48
5.5 Synthesis and Place-and-Route Results .............................. 50

6 Flight Software Design .............................................. 52

6.1 Hardware Layer .................................................. 53
6.2 Translation Layer ................................................ 53
6.3 Application Layer ................................................ 53
6.4 Main Task Function .............................................. 54

7 Testing and Validation ............................................... 60

7.1 SMILE Board ...................................................... 60
7.2 FPGA Hardware .................................................. 61
  7.2.1 DAC Signal Processing ....................................... 61
  7.2.2 Communication Protocol ..................................... 64
7.3 Flight Software .................................................. 66
7.4 Integrated System ................................................ 66

8 Conclusions and Future Work ........................................ 68

8.1 Conclusion ........................................................ 68
8.2 Future Work ...................................................... 69

Appendices

A SMILE Board Schematics ............................................ 72

B CubeSat Kit bus connections on SMILE board ......................... 76

C Adapting the SMILE Software Stack .................................. 79
# List of Figures

1.1 P-POD CubeSat launch mechanism. (Photo from [http://cubesat.org](http://cubesat.org)) ................................... 5  
1.2 Block diagram of SMILE design .................................................. 6  
1.3 The LEMI sensor ........................................................................... 7  
1.4 Concept of SMILE in the SWIM mission ....................................... 7  
2.1 Simple fluxgate ........................................................................... 10  
2.2 Operation principle for the simple fluxgate magnetometer ............. 11  
2.3 Two-core fluxgate ......................................................................... 12  
2.4 Response of a two-core sensor .................................................... 13  
2.5 Frequency response of a low pass filter ....................................... 14  
2.6 FIR filter .................................................................................... 14  
2.7 $N$-bit number with $F$-bit fractional length .................................. 15  
3.1 SPI communication mode ........................................................... 19  
3.2 Reply message to the ACS command. Each block is 16 bits wide. .... 20  
3.3 Process of the MEMFMT command .............................................. 21  
4.1 Schematic of the burnwire circuit ................................................ 25  
4.2 Schematic of the temperature measurement circuit ....................... 26  
4.3 The SMILE board on the CSK development kit ............................. 28  
4.4 Reference coefficients on X axis .................................................. 29  
4.5 Reference coefficients on Y axis .................................................... 29  
4.6 Reference coefficients on Z axis .................................................... 30  
5.1 SMILE spectrum .......................................................................... 33  
5.2 Matlab’s FDATool ....................................................................... 35  
5.3 8 bits, 11-bit fractional length .................................................... 37  
5.4 9 bits, 12-bit fractional length .................................................... 37  
5.5 7 bits, 10-bit fractional length .................................................... 37  
5.6 Comparison between the three representations ............................. 37
5.7 Block diagram of 250-Hz filter and relevant surrounding circuitry .......... 38
5.8 Waveforms for 250-Hz filter .............................................. 39
5.9 Simplified internal structure of the 250-Hz filter .......................... 39
5.10 Block diagram of 10-Hz filter and relevant surrounding circuitry ........... 40
5.11 FPGA communication protocol handling block diagram .................... 41
5.12 The SPI slave block ..................................................... 42
5.13 SPI slave state diagram .................................................. 42
5.14 The checksum computation block ........................................ 43
5.15 Checksum computation waveform ....................................... 43
5.16 The SPI control block .................................................... 44
5.17 Transmission state machine .............................................. 44
5.18 Reception state machine .................................................. 46
5.19 MEMRD state machine .................................................... 46
5.20 ACS response state machine .............................................. 47
5.21 MEMADDR response message. Useful bits are coloured. The rest of the bits are always cleared and read as '0' .................................................. 48
5.22 Memory storage hardware block diagram .................................. 48
5.23 Data packets stored to memory ........................................... 49
5.24 Time packets stored to memory. Only coloured bits are used; the rest are unused and read as zeros .................................................. 49
5.25 The sample_combine state machine ...................................... 49
6.1 SMILE software stack ..................................................... 52
6.2 The SMILE application flags (smile_flags) variable ....................... 54
6.3 The SMILE task adds an extra abstraction level to the software ............ 55
6.4 Flowchart of the SMILE task ............................................. 55
6.5 Flowchart for ACS command .............................................. 56
6.6 Flowchart for MEMFMT command ....................................... 57
6.7 Flowchart for MEMRD command .......................................... 58
7.1 Flash memory storage format for filter test phase .......................... 61
7.2 Filtered data format during test phase .................................... 62
7.3 X axis plots for data retrieved from the FPGA and calculated by Matlab .... 62
7.4 Y axis plots for data retrieved from the FPGA and calculated by Matlab .... 63
7.5 Z axis plots for data retrieved from the FPGA and calculated by Matlab .... 63
7.6 250-Hz filter spectrum for X axis ....................................... 64
7.7 Terminal output while testing the spi_slave block .......................... 65
7.8 Memory simulator state machine ........................................ 65
7.9 Reply to the ACS command during the testing phase. Each block is 16 bits wide. . . . . . 66

B.1 CSK bus connectors. The highlighted pins have routed connections. . . . . . . . . 77
List of Tables

2.1 Three-bit offset binary representation .......................... 15
2.2 Three-bit two’s complement binary representation .................. 15
2.3 Three-bit binary number representation with two-bit fractional length .................. 15
2.4 Three-bit binary number representation with five-bit fractional length .................. 16

3.1 SMILE modes of operation in SWIM mission .......................... 19
3.2 SMILE commands .............................................. 20

4.1 Resistor values for $[-100, 100]$ °C temperature range .................. 27
4.2 ADC values considering a 10-bit ADC on the FMCU .................. 27

5.1 The system-wide $M\_TIME$ clock vector .......................... 33
5.2 SMILE filter specifications .............................................. 34
5.3 Coefficients of the 250-Hz filter .............................................. 36
5.4 Coefficients for the 10-Hz filter .............................................. 38
5.5 Command flags in the $spi\_ctrl$ block .......................... 45
5.6 Response flags in the $spi\_ctrl$ block .............................................. 45
5.7 Synthesis and Place-and-Route details for the FPGA hardware .................. 50

6.1 SMILE software stack header files .............................................. 53

B.1 Short-circuit resistors for enabling connections on the CSK bus .................. 77
B.2 SMILE CubeSat Kit bus pin connections .............................................. 78

C.1 Macro prefixes in the $smile\_translation.h$ file .......................... 79
List of Abbreviations

ADC – Analog-to-Digital Converter
CSK – CubeSat Kit
DSP – Digital Signal Processing
FIR filter – Finite-Impulse Response filter
FMCU – Flight MCU
FPGA – Field-Programmable Gate Array
MCU – Microcontroller Unit
PDR – Preliminary Design Review
SMILE – Small Magnetometer In Low-mass Experiment
SPI – Serial Peripheral Interface
SWIM – Space Weather using Ion spectrometers and Magnetometers
UART – Universal Asynchronous Receiver/Transmitter
VHDL – VHSIC (Very High-Scale Integrated Circuit) Hardware Description Language
Chapter 1

Introduction and Background

1.1 CubeSat Technology

The CubeSat standard, developed and maintained by the California Polytechnic Institute of Technology (Cal Poly) in collaboration with the Space Systems Development Laboratory at Stanford University, is a standard describing a set of rules that developers should adhere to when developing a picosatellite. In addition, Cal Poly is the developer and maintainer of the Poly Picosatellite Orbital Deployer, or P-POD for short, which is the system used to launch CubeSats into orbit.

CubeSat mission schedules are designed to be completed within two to three years. This makes it ideal for implementation by teams of University students and offers them the possibility of going through the whole design process of a satellite mission. Most CubeSats launched to date are actually University projects. Many countries have launched their first satellites in the form of CubeSats.

The standard has become widespread in the space industry. Commercial off-the-shelf components offered by several companies are usually used in the design process, which allows the developers to focus on their mission payloads. This eases the development process and enables re-use of older designs, where appropriate. Because of this design process, many CubeSats are nowadays used as testing platforms for future technologies. Also, the fact that to some extent satellite components are mass-produced decreases their cost, which in turn decreases the cost of development for the satellite. In normal satellites, large costs are a result of using custom components and the long testing processes these components have to go through. In CubeSats, however, since the components are available commercially, they are tested by the producer, so no further testing costs are necessary on the developer side.

Since the launch mechanism is also defined by the standard and maintained and tested by Cal Poly for compliance with various launch vehicles (LVs), this is one less worry the developers need to have in their design process. The P-POD design [36] is presented in Figure 1.1. It is an aluminium box with a door and an opening mechanism. CubeSats placed inside the P-POD are released by a spring-based mechanism after the door is opened and slide along a set of low-friction rails. Since CubeSats are usually sent as secondary payloads on a larger mission, the P-POD design ensures that the launch of the CubeSats does not in any way interfere with either the main payload of the mission, or the LV.

CubeSats can come in different sizes. The standard 1-U CubeSat, which was the first to be described in the standard, is a 10×10×10 cm satellite with a maximum weight of 1.33 kg [37]. Sizes can then be extended on one dimension, adding units of height. Thus, a 2-U CubeSat is 10×10×20 cm, a 3-U CubeSat is 10×10×30 cm, and so forth.

Development of electronics for CubeSats is also made easy by companies providing picosatellite bus standards. Since printed-circuit boards (PCBs) are usually stacked up within a CubeSat, these standards define mechanical and electrical specifications for stack-up bus connectors, as well as how and where boards should be stacked up within the satellite.
One such component provider is Pumpkin Inc. A complete CubeSat package containing the CubeSat frame in different unit sizes, the CubeSat Motherboard (MB) containing a flight microcontroller (FMCU), a Development Board (DB) for all software development and testing, along with power supplies and other development tools necessary can be bought from Pumpkin for less than $9000. Apart from this, Pumpkin also offers various other components, such as electrical power systems, solar cell arrays, radio transceivers for downlink and uplink of telemetry data between a satellite and a ground station, etc.

An attracting feature of Pumpkin products is the multiple opportunities they offer. Most of the designs provided by Pumpkin are modular. All software development and testing can be done using the CubeSat Kit (CSK) DB, which is 100% pin-compatible to the MB. This means software written on the DB can be debugged and validated and then migrated to the MB. Multiple processor architectures are supported on both DB and MB by changing between pluggable socketed processor modules (PSPMs) in the case of the DB, and programmable processor modules (PPMs) in the case of the MB. More information on Pumpkin’s products can be found by accessing their webpage at [http://www.cubesatkit.com](http://www.cubesatkit.com).

1.2 The SWIM Mission

One such CubeSat mission and the subject of this work is the *Space Weather using Ion* spectrometers and *Magnetometers* (SWIM) design. It is a 3-U CubeSat developed by the Interamerican University of Puerto Rico in collaboration with various institutes in the US and in Europe. The project is supervised by Dr. Hien Bich Vo as the principal investigator of the mission. As with many other CubeSat designs, the main purpose of the mission is the education of students in practical space mission design.

A new boom deployment technology based on release of strain energy in bi-stable tape springs [35] is currently part of the intended payload of the satellite. The technology is developed by the United States Air Force Research Laboratory (AFRL) in collaboration between several institutes, including KTH. At the same time, NASA is developing a new attitude control system (ACS) for small satellites and the system is intended for trial on the SWIM CubeSat.

Finally, the *Small Magnetometer In Low-mass Experiment* (SMILE), a collaboration between KTH Stockholm and the Lviv Center of Institute of Space Research in Ukraine is part of the scientific payload of the satellite, along with a Langmuir probe. The purpose of the magnetometer, apart from collecting data relevant for the study of auroral regions, is to provide magnetic field data to the ACS system.

The SWIM mission began in June 2010 and it is under current plans due for launch in 2014, with no launch date being set as of yet. A Critical Design Review (CDR) for the mission is planned in August 2012. A list of objectives for the mission is presented below:

- develop a student satellite mission by a minority community in the United States;
acquire relevant space weather data to understand coupling of the Sun to the Earth;

increase the technology readiness level in different guidance, navigation and control sensors.

Participation of KTH in the SWIM mission started from early in the design process. In his MSc Thesis [38], Julio Zorita made an analysis of the spacecraft dynamics considering a gravity gradient boom with the SMILE magnetometer at the end. He developed an attitude simulator, as well as a mass model of the spacecraft. Various studies, tests and production iterations were made by local KTH teams in collaboration with the AFRL in order to create the gravity gradient boom of the satellite; work is still undergoing in that direction.

One final relevant point to be made here is the use of the CubeSat Kit (CSK) from Pumpkin. A CSK DB was sent by the team in Puerto Rico to KTH, so development on the communication protocol could be done on hardware close to the flight hardware. The FMCU used in the mission is the dsPIC33FJ256GB710.

1.3 The SMILE Fluxgate Magnetometer

Work on the SMILE digital fluxgate magnetometer has been going on for some time at KTH. The design has undergone multiple revisions, every time adding one more feature and fixing previous issues.

The SMILE design consists of the actual fluxgate core, the LEMI sensor, and the control electronics, consisting of both digital and analog components. A block diagram of the previous SMILE design is presented in Figure 1.2. The digital part is implemented inside an Actel ProASIC 3 FPGA [5]. Sensor readings are made using a 12-bit analog-to-digital converter (ADC) and multiplied by a set of reference coefficients over one excitation period to obtain a compensation value to feed to the cores. The excitation frequency for the sensor is 8 kHz. An on-board 4-Gb Hynix Flash chip [19] allows for non-volatile storage of data and communication protocol circuitry inside the FPGA allows for accessing and downloading the data stored on the Hynix chip. The data stored on the chip can later be used for analysis of the magnetic field.

The actual sensor, LEMI, presented in Figure 1.3, is a dual-magnetic core, three-axis sensor with volume compensation. It has a measurement range of approximately ±65 µT. More sensor characteristics can be found in previous publications on the instrument [16, 14, 15].
SMILE has been successfully used in several previous missions. The latest of these is the SQUID sounding rocket experiment [8], run under the European Space Agency’s REXUS programme. Around the same time, the instrument was used in the Polarized Gamma-ray Observer (PoGOLite) mission, as part of the ALBERT instrument. The experiment was supposed to measure the polarization of soft gamma rays, and due to the altitudes the experiment should have reached, SMILE was supposed be used to measure how auroral electron precipitation contributes to the gamma ray background [18]. Since the balloon that the ALBERT instrument was launched on in 2011 leaked, a re-flight is scheduled in July 2012.

In the context of the SWIM mission, the SMILE fluxgate magnetometer is located at the end of a gravity gradient boom, which is to be deployed via the SIMPLE boom described in [35]. A depiction of this is made in Figure 1.4. The instrument is used in order to provide data for the satellite’s attitude control system (ACS).

1.4 Scope of this Work

This work presents the implementation of the electronics and software for interfacing the SMILE instrument to the SWIM satellite. Since KTH has the expertise in the functioning of the instrument, it was
established that they would provide the instrument interface to the satellite.

The MSc thesis project presented here was started with this purpose. The goals of this thesis as specified at the beginning of the working period are:

- Define a hardware interface between SMILE and the SWIM CubeSat motherboard;
- Define a communication protocol between the flight microcontroller and the FPGA;
- Implement the communication protocol hardware inside the FPGA;
- Implement the software the flight microcontroller side.

This report begins with a theoretical background in Chapter 2. A high-level description of the SMILE communication protocol is then offered in Chapter 3, after which implementation details are described in Chapters 4 through 6. The SMILE board design is presented in Chapter 4, followed by a description of relevant FPGA hardware in Chapter 5 and the SMILE interface software in Chapter 6. Details on testing procedures used throughout the project are offered in Chapter 7. Finally, conclusions on the work are drawn in Chapter 8. Possible future work is also presented in the same chapter.
Chapter 2

Theoretical Considerations

2.1 The Fluxgate Effect

Fluxgate magnetometers have been used in different purposes since their invention in the 1930s. By winding an excitation coil around a ferromagnetic core and driving an alternating current through it, the material is periodically saturated, which causes the magnetic lines of flux to line up and the flux to be constant. Changing the polarity of the alternating current causes a change in the direction of the magnetic field, which means the flux would change. This change in flux produces a voltage and by winding a pick-up coil around the core, this voltage can be measured. The basic 1-core fluxgate magnetometer is shown in Figure 2.1.

With no external field applied, the voltage signal at the output of the pick-up coil displays equally-spaced peaks at one-half the excitation period (see Figure 2.2. When an external field is present at the core location, the core will be saturated for longer periods of time in the direction of this field. This translates in the voltage peaks in shifting left or right. The strength of the external magnetic field can be measured by measuring the time between two voltage peaks.

It is common for fluxgate magnetometers to have two cores instead of one. Such a setup is shown in Figure 2.3. The excitation current in this case causes the magnetic fields to cancel each other out when no signal is applied at the output, and no voltage exists at the pick-up coil. However, when an external field is present along the direction of the core, one of the cores will be saturated for a longer period of time, resulting in voltage peaks as shown in Figure 2.4. This voltage will now always be centered around half the excitation period. By measuring the amplitude of this voltage at twice the excitation frequency, the magnetic field at the core location can be derived.

Due to the fact that in the case of small magnetic fields the voltage at the pick-up coil would be small, most magnetometers use a feedback mechanism to balance the ferromagnetic cores at a zero-valued normal magnetic field. By measuring the signal at pick-up coil, an opposite field can be generated
Chapter 2. Theoretical Considerations

2.2 Digital Filtering

2.2.1 Digital Signal Processing and Filtering Basics

Digital signal processing (DSP) is a technology widely used when a signal needs to be conditioned to some extent. Whether it is signal characterization using Fourier transforms, or simple removal of unwanted components from a signal, DSP techniques can be employed to perform the operation. Signal processing was initially done using analog components, but with the introduction of the digital signal processor, where a signal is represented as a finite array of bits, storage of signals on common computer storage media is made possible for their later analysis. Digital signals are also less prone to usual errors affecting analog circuits, such as an analog circuit’s temperature drift. Flexibility is also larger on the DSP side. Since signal processing algorithms are written in software for digital signal processors, such a device can be reused to implement different algorithms, if need arises. Analog circuitry, once placed on the board, has its own distinct functionality which can only be replaced by resoldering components.

One of the more common applications of DSP is signal conditioning. An example of a filter’s frequency response is given in Figure 2.5. The figure shows the passband and stopband of a simple low-pass filter and its cutoff frequency. Signals whose frequencies are lower than the cutoff frequency are allowed to pass, i.e., are not attenuated, while signals which have a higher frequency than the cutoff are attenuated.

The basic principle behind filter design is that of convolution [10]. The output $y(t)$ of a signal $x(t)$
Chapter 2. Theoretical Considerations

2.2. Digital Filtering

Figure 2.3: Two-core fluxgate

passing through a linear-time invariant (LTI) system with the impulse response $h(t)$ is given by:

$$y(t) = x(t) * h(t)$$ (2.1a)

$$= \int_{-\infty}^{+\infty} x(\tau)h(t - \tau) \, d\tau$$ (2.1b)

$$= \int_{-\infty}^{+\infty} x(t - \tau)h(\tau) \, d\tau$$ (2.1c)

where $t$ and $\tau$ both represent time and $\tau$ of 2.1c was obtained from 2.1c by variable substitution: $\tau_{new} = t - \tau_{old}$. The $\ast$ operator symbolizes the convolution operation. In discrete-time, the integral is defined as a sum. Discrete-time convolution can therefore be defined as:

$$y(n) = x(n) * h(n)$$ (2.2a)

$$= \sum_{k=-\infty}^{\infty} x(k)h(n - k)$$ (2.2b)

$$= \sum_{k=-\infty}^{\infty} x(n - k)h(k)$$ (2.2c)

where $n = t/\Delta t$ and $\Delta t$ is the duration between two continuous signal samples.

Generally, an LTI system is considered to be causal, i.e., that the response of the system to an input signal does not depend on future values, only on present and past values. In the context of Equation 2.2c above, this translates to $h(k < 0) = 0$. Because the products of outputs before sample 0 of $x(t)$ in the case of discrete time, would be zero in Equation 2.2c, it can be redefined in the context of causal systems as:

$$y(n) = \sum_{k=0}^{\infty} x(n - k)h(k)$$ (2.3)

Considering a finite version of the previous equation, the output equation of an L-tap finite-impulse response (FIR) filter is obtained [34]:

12
Chapter 2. Theoretical Considerations

2.2. Digital Filtering

\[
y(n) = \sum_{k=0}^{L-1} x(n-k)h(k)
\]  \hspace{1cm} (2.4)

with its corresponding impulse response in the z-domain \[10, 34\]:

\[
F(z) = \sum_{k=0}^{L-1} h(k)z^{-k}
\]  \hspace{1cm} (2.5)

The \(z^{-k}\) element in Equation 2.5 is known as a delay element in the z-domain \[10\]. From this results the graphical representation of an FIR filter, presented in Figure 2.6. The \(h(k)\) elements are called the FIR filter’s coefficients; they are usually obtained by using a dedicated filter design software, such as the Filter Design Toolbox in MATLAB.

2.2.2 Number Representation

When an analog signal is digitized it is represented in the form of a vector of bits forming a binary number. Several ways of representing a binary number exist \[34\]. A number representation defines how the vector of bits can be translated into a decimal format, which is easier to read by humans, and how binary arithmetic works for that representation.

ADCs, for example, usually have an offset binary representation, whereby numbers represented on \(N\) bits in the range \([-2^{N-1}, 2^{N-1} - 1]\) are successively represented in binary steps of \(I\). This representation is equivalent to converting the binary number using normal non-signed integer representation and subtracting \(2^{N-1}\). Table 2.1 shows an example of offset binary representation.

By far the most common number representation is two’s complement. The most-significant bit (MSB) under this representation is the sign bit. For positive numbers, the sign bit is zero and the binary number is converted in a normal manner to a decimal number. For negative numbers, the MSB is one and,
assuming an $N$-bit number, $2^{N-1}$ is subtracted from the rest of the binary number. An example of two’s complement representation is given in Table 2.2. Summarizing, the two’s complement number is given by:

$$X = \begin{cases} \sum_{n=0}^{N-2} x_n 2^n & \text{if } X \text{ is positive} \\ -2^{N-1} + \sum_{n=0}^{N-2} x_n 2^n & \text{if } X \text{ is negative} \end{cases}$$

A quick comparison between Tables 2.2 and 2.1 reveals that conversion from offset binary to two’s complement representation can be done by flipping the MSB.

Fractional numbers can also be represented in fixed-point format by using two’s complement representation, by specifying a certain number of bits $F$ of the total number of bits $N$ as the fractional part of the number. Figure 2.7 shows a fractional binary number would be represented with the binary point placed after the $F-1^{st}$ bit. The range of a fractional binary format number is $[-2^{N-F-1}, 2^{N-F-1})$. For example, considering a binary number with a bit width of 3 and a fractional length 2, the range of numbers that can be represented is $[-1, 1)$ in increments of 0.25. Table 2.3 shows the numbers that can be represented in this format.

To keep number representations as accurate as possible, fractional lengths can also be larger than the actual bit width of a number. This can be useful in the case when the expected number range is less than $[-1, 1)$. Table 2.4 shows the same three-bit representation, but considering a fractional length of
Table 2.1: Three-bit offset binary representation

<table>
<thead>
<tr>
<th>Binary number</th>
<th>Decimal number</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>-4</td>
</tr>
<tr>
<td>001</td>
<td>-3</td>
</tr>
<tr>
<td>010</td>
<td>-2</td>
</tr>
<tr>
<td>011</td>
<td>-1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
</tr>
<tr>
<td>111</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2.2: Three-bit two’s complement binary representation

<table>
<thead>
<tr>
<th>Binary number</th>
<th>Decimal number</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-4</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 2.7: $N$-bit number with $F$-bit fractional length

Table 2.3: Three-bit binary number representation with two-bit fractional length

<table>
<thead>
<tr>
<th>Binary number</th>
<th>Decimal number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>-1</td>
</tr>
<tr>
<td>1.01</td>
<td>-0.75</td>
</tr>
<tr>
<td>1.10</td>
<td>-0.5</td>
</tr>
<tr>
<td>1.11</td>
<td>-0.25</td>
</tr>
<tr>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>0.01</td>
<td>0.25</td>
</tr>
<tr>
<td>0.10</td>
<td>0.5</td>
</tr>
<tr>
<td>0.11</td>
<td>0.75</td>
</tr>
</tbody>
</table>
5. The representation yields a number range \([-0.125, 0.125]\); the binary point is also shown in the table, and ‘-’ characters show unrepresented bits to the binary point.

<table>
<thead>
<tr>
<th>Binary number</th>
<th>Decimal number</th>
</tr>
</thead>
<tbody>
<tr>
<td>-100</td>
<td>-0.125</td>
</tr>
<tr>
<td>-101</td>
<td>-0.09375</td>
</tr>
<tr>
<td>-110</td>
<td>-0.0625</td>
</tr>
<tr>
<td>-111</td>
<td>-0.03125</td>
</tr>
<tr>
<td>-000</td>
<td>0</td>
</tr>
<tr>
<td>-001</td>
<td>0.03125</td>
</tr>
<tr>
<td>-010</td>
<td>0.0625</td>
</tr>
<tr>
<td>-011</td>
<td>0.09375</td>
</tr>
</tbody>
</table>

Because this representation is an extension of the two’s complement representation, binary arithmetic using fractional numbers is performed in the same way as two’s complement. When multiplying two binary numbers, the binary point is moved by the number of positions of the fractional parts of the two numbers added together, just as in decimal number multiplication. For example, when multiplying two numbers \(X_1\) and \(X_2\) with lengths \(\{N_1, F_1\}\) and \(\{N_2, F_2\}\), where \(F_n\) represent the fractional lengths, the result \(Y\) will be a number of length \(\{N_1 + N_2, F_1 + F_2\}\).

### 2.2.3 Quantization

Representing a quantity on a finite number of bits can lead to a certain loss of precision. This loss of precision is called quantization. Quantization errors occur whenever an analog quantity is stored in digital format, as for example the inputs to an ADC.

Because of this error when representing a number in the form of a bit vector, quantization can be considered as a form of noise affecting the input signal. Since quantization steps are the same within a range of numbers, as long as the input signal is uniformly distributed across a range, quantization noise can be considered as a form of additive white noise [31]. The upper bound on signal-to-noise ratio (SNR) with quantization noise is in this case 6 dB/bit [31].

In the context of DSP and digital filter design, quantization plays an important role. Filter design software such as the Filter Design Toolbox in MATLAB generate a filter’s coefficients in floating-point representation. If a filter is to be designed using a fixed-point representation (and it usually is in the context of an FPGA, since a floating-point datapath is a big area consumer), then the filter coefficients must be quantized. Quantization noise in this case affects both the passband and stopband of a filter, with stopband attenuation being most notably affected [31]. Because a filter’s coefficients tend to not be uniformly distributed along a range, the maximum SNR can usually not be obtained; a good rule-of-thumb for designing such a filter is to expect a 5 dB/bit SNR value [31]. With this value in mind, the filter coefficient bit widths should be selected to obtain the filter attenuation.

When quantizing the filter coefficients, the number representation is also something that needs to be kept in mind. When converting floating-point coefficients obtained from signal processing software to fixed-point, the fractional length of the representation needs to be selected appropriately based on the interval that the coefficients span. For example, consider a filter with 8-bit coefficients for which the highest-valued coefficient is 0.23246. Using a 7-bit fractional length representation would mean that these coefficients are in the range \([-1, 1]\). Although correct, representing the filter’s coefficients in this format would yield loss of precision because two MSBs from every coefficient are not used, and would introduce a quantization error. Instead, moving the fractional point two bits to the left, yielding a 9-bit fractional length, would make the representation interval \([-0.25, 0.25]\). This behaviour is highlighted in more detail in Section 7.2.1 of [31].
Chapter 3

The SMILE Communication Protocol

In order for data to be safely exchanged between the flight microcontroller (FMCU) and the SMILE FPGA, a dedicated communication protocol is defined. Since the SMILE magnetometer provides magnetic field readouts for the ACS system and since the space environment in which the satellite will operate is hazardous, error checking is used to ensure correct data transmission.

In order to keep the implementation modular, the decision has been made to divide work on two layers of representation:

- The **physical layer** refers to the hardware communication protocol in use. This defines parameters such as whether the communication is synchronous or asynchronous, the clock speed in the case of synchronous communication, data rates, etc.

- The **application layer** refers to a higher level of abstraction, which uses the characteristics defined by the physical communication protocol to implement various actions to be performed by the parties involved in data communication.

This chapter describes the communication protocol in use between the SMILE board and the SWIM motherboard and gives details about the error checking mechanism employed. The two layers of definition are generically described in the following sections and details about the implementations at the FMCU and the FPGA levels, respectively, are described in following chapters.

3.1 Operational Modes

Before the definition of the two layers of abstraction and how they relate to the context of the project, the operational modes of the SMILE instrument are described.

The SMILE fluxgate magnetometer is used to obtain magnetic field data for use in the attitude control system (ACS) of the spacecraft. Apart from this, during so-called scientific missions, the magnetometer will gather data about the magnetic field at the spacecraft location. The gathered data are saved to the Flash memory on the SMILE board and retrieved by the FMCU, which will then store it to the satellite’s SD (Secure Digital) card, for downlink back to Earth. This data would after that be analyzed.

While the compensation rate of the magnetometer is still the same 8 kHz as in previous missions, the sampling frequency, defined here as the storage frequency to memory, can have two values, as shown in Table 3.1. A **high-resolution** mode is defined for satellite latitudes of over 60°, where the auroral structure is rapidly-varying, and thus more measurements help to better understand this variation. Below 60° in latitude, the **survey mode** offers a sufficient frequency for analysis of the magnetic field data.
Table 3.1: SMILE modes of operation in SWIM mission

<table>
<thead>
<tr>
<th>Mode</th>
<th>Sampling frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-resolution</td>
<td>250 Hz</td>
</tr>
<tr>
<td>Survey</td>
<td>10 Hz</td>
</tr>
</tbody>
</table>

3.2 Physical Layer

Serial Peripheral Interface (SPI) [30] is used as the communication interface at the physical layer. It is a high-speed full duplex serial communication protocol designed for communication between digital circuits. It was proposed by Motorola and has seen many implementations from several chip vendors [25, 23, 11]. SPI is a master-slave communication protocol that can operate using two to four pins on the device:

- **SCLK** – serial clock, the communication clock defining the bit rate; it is controlled by the master of the communication;

- **SS** (optional) – slave select, pulled low by the master to enable communication. This pin can be neglected if there is only one slave device during communication. If there is more than one slave device, there will be one SS pin for each slave;

- **MISO** – master in serial out; it is common for the serial device to keep the MISO pin in a high impedance state while its corresponding SS pin is driven high by the master;

- **MOSI** (optional) – master out serial in. This pin can be neglected if the master does not send data to a slave, but only receives data from it.

SPI communication begins with the master pulling the SS line low. A series of 8/16/32 of SCLK pulses (depending on word length) are then used to send the bits. There are different modes of SPI communication [30], corresponding to different SCLK polarities and phases. Bit endianness can also vary from one implementation to another.

The choice in communication protocol was due to the possibility of having high data rates during transmission and the fact that the majority of SPI slave devices keep their MISO pins in high-impedance (tri-state) mode when the SS pin is set, thus allowing the use of the line by another SPI slave. One of the two SPI interfaces available on-board the FMCU is shared by SMILE with an ADIS16488 [12] inertial sensor. The FMCU is the master in the communication. To avoid reconfiguring the FMCU’s SPI interface every time there is a switch between the two slave devices, the SMILE SPI interface transfer mode is the same as that of the ADIS16488. On each SPI communication, 16 bits of data are transmitted between master and slave, starting with the MSB on the first clock and ending with the LSB on the sixteenth clock. SPI mode 3 [30] is used for data communication – see Figure 3.1. The idle state (when no communication is performed) of the SCLK pin is logic high (3.3 V) and sampling is done on the rising edge of each SCLK cycle.

![Figure 3.1: SPI communication mode](image-url)
3.3 Application Layer

At the application level, the communication protocol is implemented by means of various command characters sent by the FMCU to the SMILE FPGA. Because of the 16-bit word size used at the physical level, the actual command sent is two bytes; the first of these (the most significant byte) is zero, and the second the ASCII character. A full list of SMILE commands implemented for the SWIM CubeSat mission is given in Table 3.2. A brief description of each of the commands is made in the following subsections.

In order to assure correct sending of sensitive data, a checksum is appended to messages sent by the FPGA in response to the ACS and MEMRD command. During data reception, the checksum is computed by the FMCU and verified at the end of transmission with the one sent by the FPGA. The received message is only accepted upon reception of a correct checksum.

### Table 3.2: SMILE commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Character</th>
<th>Function</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMRD</td>
<td>a</td>
<td>Request one memory page</td>
<td>One page of memory data</td>
</tr>
<tr>
<td>MEMFMT</td>
<td>Q</td>
<td>Format Flash memory</td>
<td>none</td>
</tr>
<tr>
<td>MEMSTAT</td>
<td>F</td>
<td>Get address count of the SMILE Flash memory</td>
<td>Two words representing the memory status</td>
</tr>
<tr>
<td>ACS</td>
<td>A</td>
<td>Request ACS output</td>
<td>Single ACS output</td>
</tr>
<tr>
<td>REWIND_TOSTART</td>
<td>R</td>
<td>Rewind memory to zero address</td>
<td>none</td>
</tr>
<tr>
<td>REWIND_ONEPAGE</td>
<td>r</td>
<td>Rewind memory to previous page</td>
<td>none</td>
</tr>
<tr>
<td>MODE_HIGHRES</td>
<td>S</td>
<td>Set sampling rate to burst</td>
<td>none</td>
</tr>
<tr>
<td>MODE_SURVEY</td>
<td>s</td>
<td>Set sampling rate to survey</td>
<td>none</td>
</tr>
<tr>
<td>RECORD</td>
<td>-</td>
<td>Start and stop recording to memory. In the SWIM mission, this command is sent through an FMCU pin, instead of the communication channel.</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.3.1 The ACS command

This command is used by the flight software to retrieve measurements of the magnetic field for the attitude control system.

The ACS readout represents the current measurement of the magnetic field. As Table 3.2 and Figure 3.2 show, the SMILE FPGA responds by sending the three values of three components of the magnetic field to the FMCU by means of five sixteen-bit words, for a total of 80 bits. The first three words in the data string represent the magnetic field measured on the three axes and the last two are the checksum words.

The resolution of magnetic field component data sent by this command is 2 nT/bit.

![Figure 3.2: Reply message to the ACS command. Each block is 16 bits wide.](image-url)
3.3.2 The MEMRD command

The MEMRD command instructs the FPGA to start reading out and sending data stored in the Flash memory. Data in memory are stored in pages and read out on an 8-bit byte basis; one page is 2048-bytes long. The data are read from memory and sent to the FMCU in a page-wise manner. Once this command has been received by the FPGA, it will respond with 2048 bytes of data and thus 1024 words. An additional two checksum words are appended to the message sent by the FPGA. To get the next page of data in memory, the FMCU needs to perform the request again.

3.3.3 The MEMFMT command

Figure 3.3 shows the process of the MEMFMT command. This command is issued by the FMCU when the SMILE on-board memory is to be formatted. This is most often done after reading of a set of previously stored scientific data, in order to make room for new data storage.

The Hynix HY27UF084G2B Flash memory is erased on a block basis; each block is 64 pages in length. In order to erase all the blocks of the memory, the FMCU must issue the command once at the start of the formatting process and once at the end. The memory controller implemented in the FPGA takes care of re-issuing the block erase command to the memory chip multiple times until it receives the second command from the FMCU. The process starts with the FMCU issuing a REWIND_TOSTART, followed by the MEMFMT command to start the erase process. This ensures that the memory is erased starting with the first memory address. After the second MEMFMT command is sent to end the formatting process, the REWIND_TOSTART command is issued by the FMCU in order to return the address pointer to zero and store data saved during the next scientific mission from the start of the memory. The FPGA sends no reply to this command.

Since the memory is organised in 4096 blocks of data and erasing of one memory block takes ≤3 ms [19], formatting 4Gbits of data takes ≤12.3 seconds to complete.

3.3.4 The REWIND_TOSTART command

Apart from rewinding the address counter to zero before formatting, the REWIND_TOSTART command is also used when data stored to memory during the latest scientific mission is to be copied to the satellite SD card. Since the memory address counter in the FPGA is incremented upon writing data to memory, when the FMCU makes a request for reading out scientific data the FPGA would respond with data from memory blocks which have not yet been written. The FMCU thus has to start reading memory from
the first address in memory, and this is done by first issuing the \textit{REWIND\_TOSTART} command, after which it issues an appropriate number of the \textit{MEMRD} command.

The FPGA sends no reply to this command.

### 3.3.5 The \textit{REWIND\_ONEPAGE} command

This command instructs the address counter in the FPGA memory controller to be decremented by 1. One potential use of this command is to re-read a previously read page which has been incorrectly received by the FMCU. No response is received from this FPGA as a result of this command being sent.

### 3.3.6 The \textit{MODE\_HIGHRES} and \textit{MODE\_SURVEY} commands

These commands are sent to the FPGA to select between \textit{high-resolution} measurement mode, corresponding to data being stored to memory at 250 Hz, and survey mode, corresponding to 10 Hz storage frequency. No response is sent by the FPGA when this command is received; instead, internal FPGA logic assures selection between the two modes.

The \textit{high-resolution} mode is the default used for data storage to memory.

### 3.3.7 Error checking algorithm

The error checking algorithm employed for assuring data sent from the FPGA to the FMCU is at the time of writing of this report a simple summing checksum. Using the same checksum algorithm employed for downlink of data from the satellite to Earth was considered, however, this algorithm was not certain at the time of writing of this document and therefore, this simple way of calculating a checksum was chosen. The FPGA hardware is designed in such a way as to allow fast changing of the error checking algorithm for another, if the need arises.

The present algorithm offers no practical error correction. Its sole purpose is to make sure data sent from the FPGA to the FMCU has not been changed in the process of transmission. The words of a message are summed together and the result of this computation is appended in the form of two words at the end of the message.

It can be understood from this that if the positions of two data words are exchanged, the checksum would still be the same. However, this cannot occur in the case of this protocol, due to the fact that it is a simple point-to-point communication via SPI and the FPGA hardware simply sends one word after the other.
Chapter 4

SMILE Board Design

The SMILE board design has previously been through three versions of implementation. The latest design used in the SQUID mission [7, 18] was made on a 65×65 mm board and was intended to connect to several other boards in a configuration similar to that in the CubeSat. PCB specifications [26] from the Pumpkin CubeSat platform vendor indicate that board dimensions for PCBs inside the CubeSat are 90.17×95.89 mm in size. Thus, some sort of interface board had to be designed to add the SMILE board inside the SWIM CubeSat.

Using the older 65×65 mm board and designing a separate interface board that would adapt it to the new sizes, as well as adding a few extra components, was the first consideration made. It was soon determined that designing a new board to the correct sizes would be a more robust solution than an adaptor board. The SQUID design also lacked a dedicated programming connector for the FPGA. Because the SQUID design had several FPGAs which could be programmed in daisy-chain mode [3], programming was done through bus wires. In the context of the SWIM mission, only one Actel FPGA, the one on the SMILE board, was to be used in the design at the time of writing of this document. Adding a separate programming connector was therefore considered a more robust solution and it was used in this design.

Board design was performed using the Mentor BoardStation software suite from Mentor Graphics. Part of the design presented in this chapter is a reuse of the SQUID design; two sheets were added to the main schematic sheet of the SQUID SMILE design, adding the new components needed for the SWIM design. The additional board components were added in layout around the SQUID design.

In following sections of this chapter the process of designing the SWIM SMILE board and changes from the SQUID design are described in detail.

4.1 Schematic Design

The SQUID design of the SMILE board contained an Actel A3P250VQG100 [5] FPGA, a Flash memory chip from Hynix – HY27UF084G2B [19], analog circuitry necessary for signal conditioning at both the pick-up and the compensation sides, voltage regulator circuitry to provide the various voltage levels needed across the board, as well as an LT1180A [21] charge-pump RS-232 voltage level converter for debug purposes with a PC. A 2.5 V voltage reference was locally generated on the 5 V power supply.

Much of the old design is preserved and components are added around it to make up the new design. Modifications to the board design include the addition of a JTAG 10-pin programming connector for programming the Actel FPGA directly. The LT1180A UART level converter has been replaced by a Maxim MAX3222 [33] dual charge-pump circuit. The circuit has the same package and pinout as the LT1180A, but it can be supplied from a 3.3 V supply. This eliminated the voltage divider needed in the previous design on the FPGA UART RX pin.
The CubeSat Kit (CSK) System Bus used in the SWIM mission is a bus inspired from and similar to the widely-used PC/104 system bus [2]. Board mechanical specifications from Pumpkin are the same as those of PC/104 and the PC/104 connector, if used, is positioned as suggested in the original specifications. However, Pumpkin defines its proprietary bus with a different 104-pin connector and puts it forward as the default to use by connecting the bottom motherboard in the board stack-up to this connector. It is also the connector used to connect the boards in the SWIM CubeSat. The proprietary CSK bus connector consists of two 52-pin ESQ-126-39-G-D connectors from Samtec. Five connectors were obtained as sample orders from Samtec and placed as specified in [26]. Figure B.1 in Appendix B shows the pin connections relevant to SMILE and Table B.2 in Appendix B lists the CSK bus connections and the respective pins both on the bus connector side and the FPGA.

To enable reuse of the design in future similar missions, access is provisioned to pins on the CSK bus via 0 Ω resistors which can be soldered if the need arises. The CSK bus is designed to be used with several types of processors and some of these processors’ communication interfaces are usable via bus pins. Both I2C interfaces, both SPI interfaces as well as both UART interfaces on the CSK bus can be used. The I2C and SPI interfaces use the same FPGA pins (pins 28 through 31) and the 0 Ω resistor corresponding to the interface intended to be used should be soldered, while others are to be kept unsoldered. A list of all interfaces and the corresponding short-circuit resistor is offered in Table B.1, in Appendix B. An SN74LVC1G17 [28] Schmitt-trigger buffer from Texas Instruments is connected between the SPI/I2C clock signals and FPGA pin 28. Its purpose is to guarantee the clock signals controlling the two communication protocols are as clean as possible when reaching the FPGA. The two CSK UART interfaces share pins 43 and 44 of the FPGA together with the RS-232 voltage regulator. Refer to Table B.2 in Appendix B for a detailed list of CSK bus connections.

Four of the twelve CSK bus signals not connected to the CSK motherboard are also made available in the same way. The CSK \textit{USER0} to \textit{USER11} connections can be used to send signals exclusively between boards that are higher in the layer stack-up of the CubeSat. Signals \textit{USER0} to \textit{USER3} are connected to FPGA pins 81 through 78 respectively and can be enabled by soldering the corresponding 0 Ω resistors (see Table B.1 in Appendix B).

A wire cutter mechanism is present in the design. Its purpose is to activate the wire cutter for the SIMPLE gravity-gradient boom. The actual mechanical realization of the mechanism is not part of this work; only the electronic part is presented here. The electronic schematic of the burnwire mechanism is presented in Figure 4.1. A logic signal from the FMCU on the \textit{BOOM_DEPLOY} line controls activation of the cutter signal. When the \textit{BOOM_DEPLOY} signal is low, the Q2 transistor is turned off and Q1’s

![Figure 4.1: Schematic of the burnwire circuit](image-url)
gate is pulled high to 8 V, thus keeping it off. When \textit{BOOM\_DEPLOY} is high, Q2 turns on, connecting Q1’s gate to ground and turning it on. A current is then sent through the CUT+ wire to the boom; this current depends on the resistance of the burnwire mechanism between the CUT+ and CUT- wires. Boom feedback, while not formally defined at the time of writing of this report, is assumed to come in the form of a microswitch being toggled and connected to ground upon successful deployment.

To monitor the temperature of the boom-mounted LEMI sensor, an AD590 [9] temperature sensor is added to the sensor end of the boom. Temperature data can in this way be stored on the satellite SD card, sent to a ground station and taken into account while analyzing the data. The schematic of this circuit is presented in Figure 4.2. The circuit in Figure 4.2 was calculated considering a temperature range of $[-30, 40]$ °C and using the methodology presented in Chapter 4 of [32].

![Figure 4.2: Schematic of the temperature measurement circuit](image)

A different temperature range of $[-100, 100]$ °C was considered after the circuit was input to the schematic capture program of the Mentor BoardStation suite, so a recalculation is necessary.

Since the AD590 sensor’s temperature coefficient is 1 \( \mu \text{A/K} \), the current through the sensor is measured at the inverting input of the OPA170 over the 1 k\( \Omega \) R23 resistor, yielding 1 mV/K. Considering the new temperature range of $[-100, 100]$ °C, the voltage range expected at the op-amp’s inverting input is [173, 373] mV. Using the 2.5 V analog voltage reference existing on-board, the voltage at the op-amp’s inverting input is amplified through the OPA170 to be in the range [0, 2.5] V using the resistor circuits around the op-amp. The resulting signal is sent to an analog input of the FMCU and stored to the memory card.

In order to enable different temperature ranges, the resistors in the voltage divider circuits are changed as follows. With the [173, 373] mV voltage range on the inverting input, the transfer function of the op-amp can be calculated using the equation of a straight line,

\[ y = mx + b \] (4.1)

due to the fact that the op-amp is only used in its linear region [32]. Changing Equation 4.1 to have the input and output voltage yields

\[ V_{\text{out}} = mV_{\text{in}} + b \] (4.2)

Now, considering the 2.5 V voltage reference, it follows that the range of voltages at the output of the circuit is [0, 2.5] V, for a [173, 373] mV input, which translates to:

\[
\begin{align*}
0 &= 0.173m + b \\
2.5 &= 0.373m + b
\end{align*}
\] (4.3)

Subtracting the first equation from the second in Equation 4.3, \( m \) and \( b \) are obtained
Table 4.1: Resistor values for \([-100, 100]\) °C temperature range

<table>
<thead>
<tr>
<th>Resistor</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R21</td>
<td>510 kΩ</td>
</tr>
<tr>
<td>R22</td>
<td>820 Ω</td>
</tr>
<tr>
<td>R33</td>
<td>11 kΩ</td>
</tr>
<tr>
<td>R35</td>
<td>44.2 kΩ</td>
</tr>
</tbody>
</table>

\[ m = 12.5 \]
\[ b = -2.16 \]

The resistors in Figure 4.2 can be obtained from \(m\) and \(b\) as follows:

\[
\begin{align*}
    m &= \frac{R_{21} + R_{35} + R_{33}}{R_{35} + R_{33}} R_{22} \\
    b &= V_{\text{REF}} \left( \frac{R_{22}}{R_{22} + R_{33}} \right) \left( \frac{R_{21}}{R_{21} + R_{35} + R_{33}} \right)
\end{align*}
\] (4.4)

The resistor values are shown in Table 4.1. Assuming an on-chip ADC on the FMCU configured for 10-bit operation from a voltage reference of 2.5 V, the values at the output of the ADC after conversion are shown in Table 4.2.

Table 4.2: ADC values considering a 10-bit ADC on the FMCU

<table>
<thead>
<tr>
<th>Temperature</th>
<th>Voltage at ADC input</th>
<th>ADC value</th>
</tr>
</thead>
<tbody>
<tr>
<td>173 K</td>
<td>0 V</td>
<td>0</td>
</tr>
<tr>
<td>373 K</td>
<td>2.5 V</td>
<td>775</td>
</tr>
<tr>
<td>3K (resolution)</td>
<td>3.2 mV</td>
<td>1</td>
</tr>
</tbody>
</table>

Should a different temperature range be expected at the instrument end, similar calculations can be performed and the resistor values changed, yielding the same range of voltages at the ADC input. The temperature resolution would change depending on the temperature range. Since this is a generic calculation, resistor values with 5% tolerance have been selected. In the purpose of the final mission, however, lower-tolerance resistors should be used.

Extra pins are provisioned for the extra connections to the SMILE end of the boom. Seven pins are added to the previous 15-pin connector, one of which corresponds to one of the twisted pair cables for the feedback signal from the boom deployment circuit. A twisted pair is connected to the two ends of the AD590 sensor for the temperature readout circuit and four of the pins are connected to the cutter resistor. Two pins are reserved for each of the cutter connections to ensure lower current flowing through each wire.

Finally, an LTC1911-1.5 [22] switched-capacitor voltage regulator from Linear Technology was added to the new design of the SMILE board as an alternative to the previous TPS76915 [27] low-dropout (LDO) regulator to provide the 1.5 V FPGA core supply voltage. The reason behind its addition is the added efficiency provided by switched-capacitor voltage regulators as compared to LDOs. Both circuits are present on the board, and selection between using one or the other regulator is made via 0Ω resistors.

The full schematics for the SMILE board are presented in Appendix A.

4.2 Board Layout

The SWIM SMILE board is, as in the SQUID design, a six-layer board. One of the internal layers is used as a ground layer connection, and power area fills are present across other layers for each of the supply voltages on board. A photo of the board placed on the Pumpkin CSK development kit is presented in Figure 4.3.
Layout of the analog pick-up and compensation parts of the board is almost the same as the SQUID design. Practically, the layout was taken from the previous design, centered on the current board, some components were moved around to accommodate new routing and new component placement, and finally the new components added to the design. The power area fills around some of the analog components were modified to make connections of new components to power nets easier.

The 0Ω resistors used to connect the communication channels to the FPGA are placed close to the CSK bus connectors. The Schmitt trigger circuit is placed close to pin 28 of the FPGA to assure the clocking signal for the I2C or SPI communication (depending on choice of protocol) is as clean as possible. Only one resistor for each FPGA pin should be soldered depending on choice of communication protocol. For the UART RX and TX lines, the 0Ω resistors corresponding to CSK bus connections are placed close to the CSK bus connector, and those corresponding to the debug connection are placed close to the charge-pump voltage regulator. Only one of three resistors for each line should be soldered at one time, to avoid damage to other UART circuitry.

Finally, the seven extra pins for connecting the new twisted pair cables are placed close to the old 15-pin connector.

4.3 SMILE Instrument Calibration

In order to calibrate the instrument, the same procedure was used as for the original SMILE design [16]. The LEMI sensor was placed in a magnetic shield and ADC readings were made for DAC compensation values for which the residual magnetic field was close to zero. Then, the compensation values were increased and decreased for each component and the ADC values re-read. Based on this, the set of reference coefficients were generated for the X, Y and Z components.

The coefficients are plotted in Figures 4.4 through 4.6.
Chapter 4. SMILE Board Design

4.3. SMILE Instrument Calibration

Figure 4.4: Reference coefficients on X axis

Figure 4.5: Reference coefficients on Y axis
Figure 4.6: Reference coefficients on Z axis
Chapter 5

FPGA Hardware Design

This chapter provides an insight into the hardware implemented in the FPGA. Since much of the design was inherited from a previous version of SMILE, some of the details about certain parts have been omitted from this work. The reader is referred to previous work on SMILE [15, 14, 18] for more information.

Hardware for the Actel FPGA was developed using the Microsemi Libero IDE [4] software pack. The hardware was described using the VHDL hardware description language and Libero ViewDraw [6] was used to connect the VHDL blocks into a design.

5.1 Timing in the SMILE Design

Timing within the SMILE design is kept using a 25-bit vector called $M_{TIME}$, shown in Table 5.1. The design of the system clock generation block is inherited from a previous SMILE design. The 32.768 MHz clock on-board the SMILE board is the input to a PLL block inside the FPGA, which generates a 65 MHz master clock. This clock, apart from being used by several blocks in the FPGA design, is also used to generate this timing vector.

The system clock generation block consists of two counters, generically named here low and high counters. Both of them are sensitive to the falling edge of the master clock. The high counter is an 18-bit running counter. Because its LSB toggles once every falling edge of the master clock, a clock signal with a divided-by-two-frequency is generated out of this. Similarly, the second bit divides master clock frequency by four. Signal frequencies of up to 250 Hz are generated in this way, which is the frequency of the MSB.

The low counter only increments when the high counter has all its bits set. Apart from generating the other, lower frequencies, this counter is used for generating a 1-Hz clock signal. It is reset automatically on a falling edge of the master clock if its value is 249, and since it counts 250 values of the 250-Hz clock generated by the MSB of the high counter, the 1 Hz frequency is thus generated.

Various clocks in the $M_{TIME}$ vector are used throughout the design. Various blocks also have part of the vector as an input, for design-wide synchronization purposes.

5.2 DAC Output Signal Filtering

Due to various factors, the SMILE design shows an unwanted spectral feature at around 1kHz. This unwanted feature, shown in Figure 5.1 needs to be cut off using a digital filter if sampling at lower frequencies is to be done.

In older SMILE designs where memory data were stored with a 250 Hz frequency, integration was
Table 5.1: The system-wide \( M_{\text{TIME}} \) clock vector

<table>
<thead>
<tr>
<th>Bit</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32.768 MHz</td>
</tr>
<tr>
<td>1</td>
<td>16.384 MHz</td>
</tr>
<tr>
<td>2</td>
<td>8.192 MHz</td>
</tr>
<tr>
<td>3</td>
<td>4.096 MHz</td>
</tr>
<tr>
<td>4</td>
<td>2.048 MHz</td>
</tr>
<tr>
<td>5</td>
<td>1.024 MHz</td>
</tr>
<tr>
<td>6</td>
<td>0.512 kHz</td>
</tr>
<tr>
<td>7</td>
<td>0.256 kHz</td>
</tr>
<tr>
<td>8</td>
<td>0.128 kHz</td>
</tr>
<tr>
<td>9</td>
<td>0.064 kHz</td>
</tr>
<tr>
<td>10</td>
<td>0.032 kHz</td>
</tr>
<tr>
<td>11</td>
<td>0.016 kHz</td>
</tr>
<tr>
<td>12</td>
<td>0.008 kHz</td>
</tr>
<tr>
<td>13</td>
<td>0.004 kHz</td>
</tr>
<tr>
<td>14</td>
<td>0.002 kHz</td>
</tr>
<tr>
<td>15</td>
<td>0.001 kHz</td>
</tr>
<tr>
<td>16</td>
<td>0.0005 kHz</td>
</tr>
<tr>
<td>17</td>
<td>0.00025 kHz</td>
</tr>
<tr>
<td>18</td>
<td>0.000125 kHz</td>
</tr>
<tr>
<td>19</td>
<td>0.0000675 kHz</td>
</tr>
<tr>
<td>20</td>
<td>0.00003875 kHz</td>
</tr>
<tr>
<td>21-25</td>
<td>0.00001 Hz</td>
</tr>
</tbody>
</table>

Figure 5.1: SMILE spectrum
done on 32 samples of compensation values computed at a frequency of 8 kHz. Since integration in the
digital domain is represented by summation, any spectral features are amplified. This problem is tackled
in the SMILE design for the SWIM mission. The data coming from the compensation block are first
filtered through a low-pass filter with 32 taps and downsampled to 250 Hz, providing data for storage in
the high resolution mode. The data output from the 32-tap filter are further filtered down to 10 Hz using
another 25-tap FIR filter with downsampling.

Filter design starts from a set of filter specifications and inputting these to a filter design software
tool. Various types of filters can be tried to see which provides the best frequency response according
to specifications. Once a selection is made, the filter design software is used to generate the filter’s
coefficients and quantize them into a fixed-point representation if desired.

It was established that an FIR filter should be used for filtering the DAC signal. The choice is due to
the simplicity in designing an FIR filter’s datapath and their guaranteed stability. Furthermore, in order
to reduce the area consumption of the filter circuitry, a fully-serial FIR filter topology was selected. Two
filter datapaths are implemented inside the FPGA, one to provide filtered data at 250 Hz, corresponding
to high resolution mode operation for SMILE, and one to provide filtered data at 10 Hz, corresponding
to operation in survey mode.

The following subsections present the design process, starting with the filter specifications, describing
the Matlab Filter Design and Analysis Tool which was used to generate the filter coefficients. A study on
how coefficient quantization affects the filter’s frequency response in the context of the current application
is also presented. This section ends with a description of the hardware implementation of the filter.

5.2.1 Filter Specifications

A set of specifications was derived from the need of cutting of the spectral features in Figure 5.1. These
specifications, shown in Table 5.2, were derived as stated below.

Since in the context of the SWIM mission there are two data rates for storing values to memory (see
Section 3.1), two sets of specifications exist, one for the burst mode, and one for the high resolution mode.
The cutoff frequency specifications are derived from the Nyquist frequency criterion and are based on the
output data rate. To avoid aliasing when downsampling a signal the output has to fall within an integer
frequency band [34] which is less than half the output frequency. The cutoff frequencies are therefore
selected to be less than half the output data rate. Attenuation parameters are derived from the need of
cutting off the unwanted frequency components.

<table>
<thead>
<tr>
<th>Name of parameter</th>
<th>Burst mode</th>
<th>Survey mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain (relative to DC) at cutoff frequency</td>
<td>-2 .. -4 dB</td>
<td>-2 .. -4 dB</td>
</tr>
<tr>
<td>Cutoff frequency</td>
<td>100 Hz</td>
<td>3 Hz</td>
</tr>
<tr>
<td>Output data sample rate</td>
<td>250 Hz</td>
<td>10 Hz</td>
</tr>
<tr>
<td>Bandpass nonflatness, max.</td>
<td>3 dB</td>
<td>1 dB</td>
</tr>
<tr>
<td>Stopband attenuation, min. (relative to DC gain)</td>
<td>-10 dB (at 200 Hz)</td>
<td>-10 dB (at 6 Hz)</td>
</tr>
<tr>
<td></td>
<td>-12 .. -15 dB (at 250 Hz)</td>
<td>-12 .. -15 dB (at 10 Hz)</td>
</tr>
<tr>
<td></td>
<td>-40 dB (at 800 Hz)</td>
<td>-50 dB (800 Hz)</td>
</tr>
<tr>
<td></td>
<td>-46 dB (at 1 kHz)</td>
<td>-60 dB (at 1 kHz)</td>
</tr>
<tr>
<td></td>
<td>-40 dB (at 1.2 .. 2 kHz)</td>
<td>-50 dB (at 1.2 .. 2 kHz)</td>
</tr>
<tr>
<td>Number of taps for the filter</td>
<td>32</td>
<td>25</td>
</tr>
<tr>
<td>Number of bits at the output</td>
<td>20</td>
<td>22</td>
</tr>
</tbody>
</table>
5.2.2 Matlab’s FDATool

The Filter Design and Analysis Tool (FDATool) from Matlab was used in analyzing and comparing different filter design methods. Its graphical user interface is shown in 5.2.

![Figure 5.2: Matlab’s FDATool](image)

The type of filter to be designed can be selected from the lower-left part of the GUI. The order of the filter, along with filter specifications (sampling and cutoff frequencies, passband and stopband attenuations, etc.) and other filter parameters can be selected from the GUI. The tool promptly shows the frequency response of the filter based on GUI field selections. Phase and magnitude responses, impulse and step responses, as well as pole/zero plots for the filter can also be viewed to further analyse the filter.

After the filter specifications have been input into the GUI, pressing the Design filter button generates the filter’s coefficients and shows the filter’s frequency/phase response (based on user selection). The coefficients can be output to the Matlab command line for further study, should that be needed. FDATool also aids the user in designing a custom hardware or software filter. Both single- and double-precision floating-point arithmetic can be used for computing the filter; fixed-point arithmetic can also be used, with custom coefficient, input, output, and internal filter quantization. Different selections made in the GUI are updated in the filter response window, which aids the user in studying how different quantizations affect the filter’s response.

Once the filter has been deemed to correspond to specifications, FDATool allows for outputting the filter’s coefficients in the form of a C header. What’s more, it can also generate an HDL file with the filter’s implementation. Either VHDL or Verilog can be selected for the output implementation, depending on the user’s selection. The HDL generation tool allows for different types of filter implementations, whether it is fully parallel, fully serial, a combination of these, or cascaded form design. Many other settings can be selected before generating an actual HDL filter. The reader is referred to the FDATool User’s Guide [1] and is encouraged to experiment with the tool for more information and better understanding of its usage.

To comply with the unity gain requirement for the filter, all filter coefficients are computed for unity gain. This means that for both filters, the coefficients sum up to the value 1. For the 250-Hz filter, considering the representation using 11-bit fractional length, this means that the sum of the coefficients is $2^{11}$. Similarly, for the 10-Hz filter, the representation has 10-bit fractional length, so the sum of the coefficients is $2^{10}$.
5.2.3 Filter Coefficients

Filter coefficient quantization can have a significant effect on the filter’s frequency response. As specified in Section 2.2.3, stopband sidelobe attenuations are most affected by improper quantization.

To study how coefficient quantization affects the frequency response for the SMILE filter, FDATool along with several of Matlab’s signal processing tools were used. First, the filter specifications were input to FDATool and when a filter’s frequency response was deemed appropriate, a Matlab script was generated from FDATool. This script was then modified to enable the study of quantization effects on the frequency response.

By experimenting with several types of filter design algorithms existing in FDATool, it was determined that a Chebyshev window would offer a frequency response closest to specifications, in both the 250- and 10-Hz filter cases. A 40dB stopband attenuation was selected for both the 250-Hz and the 10-Hz filters.

The filter coefficients for the 250-Hz filter are shown in floating-point representation in the first column of Table 5.3. Since the largest coefficient is 0.05694..., a bit representation with a fractional length 3 bits higher than the bit representation, e.g., 8 bits with 11-bit fractional length, should be enough to enable full-precision arithmetic without the filter response being affected. This representation would mean the range of values represented is \([-0.0625, 0.0625)\), of course rounded to the nearest binary representation.

Table 5.3: Coefficients of the 250-Hz filter

<table>
<thead>
<tr>
<th>Floating-point</th>
<th>Quantized</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00633497128751794</td>
<td>0.005859375</td>
<td>12</td>
</tr>
<tr>
<td>0.00582211906840849</td>
<td>0.00537109375</td>
<td>11</td>
</tr>
<tr>
<td>0.00841783300434312</td>
<td>0.00830078125</td>
<td>17</td>
</tr>
<tr>
<td>0.0115608236329254</td>
<td>0.01123046875</td>
<td>23</td>
</tr>
<tr>
<td>0.0152321679920831</td>
<td>0.01513671875</td>
<td>31</td>
</tr>
<tr>
<td>0.0193795429837648</td>
<td>0.01904296875</td>
<td>39</td>
</tr>
<tr>
<td>0.0230162697401703</td>
<td>0.0234375</td>
<td>48</td>
</tr>
<tr>
<td>0.0287228744493644</td>
<td>0.0283203125</td>
<td>58</td>
</tr>
<tr>
<td>0.033651259784532</td>
<td>0.033203125</td>
<td>68</td>
</tr>
<tr>
<td>0.0385313545682435</td>
<td>0.0380859375</td>
<td>78</td>
</tr>
<tr>
<td>0.0431798745497504</td>
<td>0.04296875</td>
<td>88</td>
</tr>
<tr>
<td>0.0474106304300389</td>
<td>0.04736328125</td>
<td>97</td>
</tr>
<tr>
<td>0.0510456415739634</td>
<td>0.05078125</td>
<td>104</td>
</tr>
<tr>
<td>0.0539262218077626</td>
<td>0.0537109375</td>
<td>110</td>
</tr>
<tr>
<td>0.0559231695584248</td>
<td>0.0556640625</td>
<td>114</td>
</tr>
<tr>
<td>0.0569452453687856</td>
<td>0.056640625</td>
<td>116</td>
</tr>
<tr>
<td>0.0569452453687856</td>
<td>0.056640625</td>
<td>116</td>
</tr>
<tr>
<td>0.0559231695584248</td>
<td>0.0556640625</td>
<td>114</td>
</tr>
<tr>
<td>0.0539262218077626</td>
<td>0.0537109375</td>
<td>110</td>
</tr>
<tr>
<td>0.0510456415739634</td>
<td>0.05078125</td>
<td>104</td>
</tr>
<tr>
<td>0.0474106304300389</td>
<td>0.04736328125</td>
<td>97</td>
</tr>
<tr>
<td>0.0431798745497504</td>
<td>0.04296875</td>
<td>88</td>
</tr>
<tr>
<td>0.0385313545682435</td>
<td>0.0380859375</td>
<td>78</td>
</tr>
<tr>
<td>0.033651259784532</td>
<td>0.033203125</td>
<td>68</td>
</tr>
<tr>
<td>0.0287228744493644</td>
<td>0.0283203125</td>
<td>58</td>
</tr>
<tr>
<td>0.0230162697401703</td>
<td>0.0234375</td>
<td>48</td>
</tr>
<tr>
<td>0.0193795429837648</td>
<td>0.01904296875</td>
<td>39</td>
</tr>
<tr>
<td>0.0152321679920831</td>
<td>0.01513671875</td>
<td>31</td>
</tr>
<tr>
<td>0.0115608236329254</td>
<td>0.01123046875</td>
<td>23</td>
</tr>
<tr>
<td>0.00841783300434312</td>
<td>0.00830078125</td>
<td>17</td>
</tr>
<tr>
<td>0.00582211906840849</td>
<td>0.00537109375</td>
<td>11</td>
</tr>
<tr>
<td>0.00633497128751794</td>
<td>0.005859375</td>
<td>12</td>
</tr>
</tbody>
</table>
Figures 5.3 through 5.5 show the frequency responses of the filter considering quantizations with different bit widths and fractional lengths. Frequency responses before and after quantization as calculated by Matlab are shown. Considering the range of the coefficients, representations on 8 bits with an 11-bit fractional length, 7 bits with 10-bit fractional length and 9 bits with 12-bit fractional length were studied. As can be seen also in the comparison between the three quantization schemes in Figure 5.6, the responses for 7-, 8- and 9-bit coefficients are fairly similar, with small deviations from the unquantized filter responses. An 8-bit coefficient length was selected for this design. The filter coefficients using the selected quantization mode are shown in column two of Table 5.3; the integer representation of these coefficients are shown in column three of the same table.

In the case of the 10-Hz filter, the filter coefficients are shown in the first column of Table 5.4. The maximum-valued coefficient is 0.07199..., which means a bit representation with a fractional length two bits higher than the length of the representation should be used, yielding the range of numbers of the coefficients \([-0.125, 0.125)\). An 8-bit representation was selected in order to keep the implementation close to the 250-Hz filter. The quantized coefficients are shown in the second column of Table 5.4, with the corresponding integer representation in the third column of the table.

### 5.2.4 Hardware Implementation

Upon deciding on a suitable frequency response, it was proceeded to actual hardware implementation of the two filters. The datapaths for the two filters are similar, with differences in the signals that control when a computation is performed. Both filters reuse the same multiply-accumulate (MAC) operation in a serial manner to perform the calculations. Signals from the X, Y and Z components are multiplexed to use the MAC datapath based on when they become available and the \(M_{TIME}\) global time vector controls when each input is selected.

The block diagram for the 250-Hz filtration circuitry is presented in Figure 5.7, and a waveform depicting the filter operation is presented in Figure 5.8. A simplified diagram of the internal structure of the filter is also presented in Figure 5.9. Bit 10 of the \(M_{TIME}\) signal is used as a clock for this datapath. Bits 11 and 12 are used as selection signals for multiplexing the inputs for the three components, and bit 12 is also used as a clock signal to the filter output registers. A rising edge on the \(load\) signal is used
Table 5.4: Coefficients for the 10-Hz filter

<table>
<thead>
<tr>
<th>Floating-point</th>
<th>Quantized</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00796630983910578</td>
<td>0.0078125</td>
<td>8</td>
</tr>
<tr>
<td>0.00923298241769831</td>
<td>0.0087890625</td>
<td>9</td>
</tr>
<tr>
<td>0.0140983691739623</td>
<td>0.013671875</td>
<td>14</td>
</tr>
<tr>
<td>0.0200248088581376</td>
<td>0.01953125</td>
<td>20</td>
</tr>
<tr>
<td>0.026866994168425</td>
<td>0.0263671875</td>
<td>27</td>
</tr>
<tr>
<td>0.0343739375909941</td>
<td>0.0341796875</td>
<td>35</td>
</tr>
<tr>
<td>0.0421997426057343</td>
<td>0.0419921875</td>
<td>43</td>
</tr>
<tr>
<td>0.049926429078645</td>
<td>0.0498046875</td>
<td>51</td>
</tr>
<tr>
<td>0.0570970287324377</td>
<td>0.056640625</td>
<td>58</td>
</tr>
<tr>
<td>0.06325565845166967</td>
<td>0.0625</td>
<td>64</td>
</tr>
<tr>
<td>0.0679901519783737</td>
<td>0.0673828125</td>
<td>69</td>
</tr>
<tr>
<td>0.0709722711673463</td>
<td>0.0703125</td>
<td>72</td>
</tr>
<tr>
<td>0.0719906318748865</td>
<td>0.0712890625</td>
<td>73</td>
</tr>
<tr>
<td>0.0709722711673463</td>
<td>0.0703125</td>
<td>72</td>
</tr>
<tr>
<td>0.0679901519783737</td>
<td>0.0673828125</td>
<td>69</td>
</tr>
<tr>
<td>0.06325565845166967</td>
<td>0.0625</td>
<td>64</td>
</tr>
<tr>
<td>0.0570970287324377</td>
<td>0.056640625</td>
<td>58</td>
</tr>
<tr>
<td>0.049926429078645</td>
<td>0.0498046875</td>
<td>51</td>
</tr>
<tr>
<td>0.0421997426057343</td>
<td>0.0419921875</td>
<td>43</td>
</tr>
<tr>
<td>0.0343739375909941</td>
<td>0.0341796875</td>
<td>35</td>
</tr>
<tr>
<td>0.026866994168425</td>
<td>0.0263671875</td>
<td>27</td>
</tr>
<tr>
<td>0.0200248088581376</td>
<td>0.01953125</td>
<td>20</td>
</tr>
<tr>
<td>0.0140983691739623</td>
<td>0.013671875</td>
<td>14</td>
</tr>
<tr>
<td>0.00923298241769831</td>
<td>0.0087890625</td>
<td>9</td>
</tr>
<tr>
<td>0.00796630983910578</td>
<td>0.0078125</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 5.7: Block diagram of 250-Hz filter and relevant surrounding circuitry
to latch the offset binary compensation data from the compensation register. The binary offset data is converted to a two’s complement representation before storage in the input registers.

A 5-bit counter internal to the 250-Hz filter block is incremented on every rising edge of the load signal and is used as an address counter to the filter coefficient ROM. When this signal reaches the value 31, an internal done_delay signal is set on the rising edge of the $M\_TIME(12)$ signal, which corresponds to the 8-kHz clock. The output done signal is set on the next falling edge of $M\_TIME(12)$. At the same time, the filtered value is latched to the output registers. The done_delay signal is used as a register enable signal for the output registers, so the filtration value is stored until the next is available.

In order to multiplex three components on the same datapath, three internal registers are used to store the value of the previous MAC operation. Each of these registers operates on the rising edge of $M\_TIME(10)$ and are successively enabled based on the states of bits 11 and 12 of the $M\_TIME$ signal, as seen in Figure 5.8. Four time slots exist during one period of the $M\_TIME(12)$ signal. Three of these slots are used for enabling the registers after the MAC operation for each components. When the done_delay signal is also set, the remaining time slot is used to clear the MAC registers and prepare the filter for computing the next set of filtered data. By the time the clear signal goes low, a new compensation value
Internally, the bit growth of the 250-Hz filter is of four bits, due to the values of the 8-bit coefficients described in Section 5.2.3 being multiplied-and-added 32 times to the 13-bit input. The multiplier is 21 bits wide and the adder, since it is where loss of precision can occur, is the full 25 bits in length. The synthesis tool is able to detect the fact that all coefficients used are positive values, so it infers a 7-bit ROM for the coefficients.

The 10-Hz filter has a structure similar to that of the 250-Hz filter. A block diagram of the 10-Hz filtration circuitry is presented in Figure 5.10. The clock of the 10-Hz filter is $M_{\text{TIME}}(15)$. Bits 16 and 17 of the $M_{\text{TIME}}$ vector are used as output register enables in a manner similar to the 250-Hz filter. Since values computed by the 250-Hz filter are stored in registers for one filtration period, no input registers are needed for the 10-Hz filter. A 5-bit modulo-25 counter is used as an address counter to the filter coefficients. As with the 250-Hz filter, there is a done_delay signal which is set on a rising edge of $M_{\text{TIME}}(17)$, which corresponds to the 250-Hz clock. The filter’s output done signal is set on the next falling edge of $M_{\text{TIME}}(17)$ after the done_delay signal is set.

The internal datapath of the 10-Hz filter is 31 bits long, as a result of a 3-bit growth due 20 bits at the input being multiplied by the 8-bit coefficients and added 25 times. As in the case of the 250-Hz filter, the synthesis tool infers a 7-bit ROM for the coefficients, due to their being all positive.

The VHDL code for the filters has been written in such a way as to enable re-use of the design, in case a decision is made later on to have different operation modes, with different storage frequencies. By changing the filter’s VHDL generics, the user may in principle copy the design of one filter and adjust its internal structure to suit the new needs. The filter coefficients can also be changed with new ones by copy-and-pasting the new integer coefficient values in the coefficient ROM table in the filter VHDL file.

Synthesis and place-and-route in the Libero IDE yielded a usage of 969 cells for the 250-Hz filter, and 1164 cells for the 10-Hz filter (see Table 5.7). Each filter has an area consumption of over 15% of the total FPGA area. Such large consumption is due to the use of shift-and-add multipliers inferred by the synthesis tool as a result of using the multiplication operator in VHDL. The fact that the datapath is multiplexed and several more registers are needed to store values of previous MAC calculations also adds to the area consumption. One possible improvement that can be made here is the usage of a fully serial multiplier to compute the multiplications. In principle, should a multiplier be designed to compute the value of multiplication between the present input and the present coefficient within the slot of one component computation, this solution would be a viable one to decrease area size. The serial multiplier should be able to compute the result within 250 µs in the case of the 250-Hz filter, and within 2 ms in the case of the 10-Hz filter, in order for a valid calculation on the convolution sums to be performed.
5.3 Communication Protocol Implementation

This section presents the implementation of the communication protocol presented in Chapter 3. To some extent, the same separation into so-called 'physical' and 'application' layers is maintained in the FPGA implementation. There are four separate hardware blocks that together handle the communication protocol.

Figure 5.11 shows a block diagram of the FPGA communication protocol hardware. Since the FMCU is the master in the SPI communication detailed in Section 3.2, the FPGA is the slave. Communication between master and slave begins with the FMCU sending a word bit-by-bit to the FPGA. The \textit{spi\_slave} block is responsible with shifting in the bits sent by the FMCU and presents a 16-bit word at its output. This block represents the 'physical layer' in the hardware implementation. Responding to the commands defined in Section 3.3 and therefore the 'application layer' is implemented via the \textit{spi\_ctrl} block. This block is responsible with responding to commands received from the FMCU and commanding the rest of the FPGA hardware based on the received command. The \textit{chksum} block computes the checksum to be appended at the end of the message.

The communication protocol hardware was first developed on a Xilinx Spartan-3 FPGA placed on a Digilent Basys board [13], after which the design was migrated to the Actel FPGA. Since the VHDL code used in development is the same, the software protocol is architecture-independent.

A detailed description of the implementation of each block in Figure 5.11 is given in the following subsections.

5.3.1 The \textit{spi\_slave} Block

The \textit{spi\_slave} block (Figure 5.12) is the first in the communication protocol; it is at its level that the physical communication protocol is implemented. Bits sent by the FMCU on the \textit{MOSI} line are shifted in on every \textit{SCLK} cycle, and simultaneously bits are shifted out on the \textit{MISO} line. To enable usage of other devices on the bus (such as the ADIS16488 IMU), the MISO line is kept in a high impedance state when the \textit{SS} line is high.

The block is implemented in the form of two state machines, one for input and one for output. The state diagram for the two state machines is presented in Figure 5.13. Each state machine consists of two states. In the \textit{idle} state, the state machines are waiting for the FMCU to pull the \textit{SS} line low and send the first \textit{SCLK} and in the \textit{shifting} state bits are shifted in and respectively out of the block. Two counters – one for input and one for output – are used to count the number of bits shifted by each state machine. Similarly, two 16-bit shift registers are used to send and receive the bits during one
communication frame. The done signal is a way of informing other blocks that the word at the rx_word output is valid; it is set upon reception of 16 clock pulses from the FMCU.

Because of the SPI mode selected, bits are sampled on the rising edge of SCLK (see Figure 3.1), the input state machine is triggered on the rising edge of the signal. Upon device reset, the input state machine is set into the idle state, the input shift register is cleared and the bit counter is cleared. The done signal is also set upon device reset, to signal that the state machine is idle. When the FMCU pulls the SS line low and sends the first SCLK, the done signal is pulled low and the state machine goes in the shifting state. Sixteen bits are then shifted in to the LSB of the shift register. Upon reception of the sixteenth SCLK cycle, the state machine is placed back in the idle state and the done signal is set, to signal correct reception of a word. Communication errors are avoided by not setting the done signal in case the SS line is pulled high by the FMCU before sending all sixteen bits of an SPI word. In such a case, only the bit counter is reset. If such a mid-word communication interruption occurs, sending/receiving the word should be reattempted from the start.

The output state machine works in a similar manner to the input state machine. However, the output state machine is sensitive to the falling edge of the SCLK signal, to ensure that bits to be sent via the communication channel are stable by the rising edge of SCLK, when they are sampled by the FMCU. The value at the tx_word input is written to the output shift register on the first falling edge of SCLK and sixteen bits are then successively shifted out of the MSB of the shift register.

Area consumption of the checksum block amounts to 76 of the 6144 cells available on board an Actel 250-kgate FPGA device (see Table 5.7).

5.3.2 The checksum Block

The checksum block shown in Figure 5.14 is tasked with building a checksum to be appended to an ACS or memory data message sent to the FMCU. The checksum is built from words received at the word_i input and after a number of SPI transmission cycles (three, if it is building the ACS checksum, or 1024 if building the memory readout checksum), the dword_o output contains a valid checksum double word. Using this double word as an input, the spi_ctrl block appends the checksum to the message sent via SPI.

In order to calculate a checksum based on valid words, the checksum block relies on the spi_ctrl block for controlling several signals. The clock input to the checksum block is the 1-MHz global clock (M_TIME[5]). A binary adder is used to build the simple summation-checksum described in Section 3.3.7. Because a
maximum number of 1024 additions of the words being sent are possible, the checksum is a double word, to make sure no overflows occur on the additions. Due to the 1024 additions, a maximum bit growth of 11 bits is possible, so the binary adder is 27 bits wide.

The $en$ and $clr\_acc$ signals control when the checksum additions are performed, and when the accumulator register is reset. These signals are driven by the $spi\_ctrl$ block as seen in Figure 5.15. Due to reasons pertaining to the implementation of the $spi\_ctrl$ block, the $en$ signal has the elongated shape seen in the figure and because there are multiple clock cycles while the $en$ signal is set, a rising edge detection circuit on this signal is used to inform the block when an addition can be performed, by setting the enable signal to the accumulator output register for one clock cycle.

Due to the transmitted word being latched at the output of the $spi\_ctrl$ block and the fact that this block is building a word out of two bytes when sending memory data, the rising edge of the $en$ signal alone would cause incorrect data being input to the accumulator. Thus, the enable signal to the accumulator is delayed for one more clock cycle, to assure correct checksum computation when sending memory data. This can be seen in Figure 5.15.

Area consumption of the $chksum$ block amounts to 186 of the 6144 cells available on board an Actel 250-kgate FPGA device (see Table 5.7).

### 5.3.3 The $spi\_ctrl$ Block

The block diagram of the $spi\_ctrl$ block is shown in Figure 5.16. The hardware for responding to commands from the FMCU is implemented in the form of several state machines that interact between them. Internal flags are used to implement the communication between the state machines, with each state machine checking for a certain flag before starting operation.

Five separate state machines are used to assure command response. A reception state machine handles responding to commands and setting internal command flags, which eventually trigger other state machines. A transmission state machine is used mostly as a means of signaling other state machines that a command is executing and replies are made via the SPI communication channel. The other three
5.3. Communication Protocol Implementation

State machines handle the ACS command, the MEMADDR command and the MEMRD command and set response flags, to signal the fact that a response is being sent. Command flags are shown in Table 5.5 and response flags are shown in Table 5.6. In these tables, flags are named according to their name in VHDL code.

The transmission state machine, shown in Figure 5.17 consists of three states. In the idle state, it waits for one of the other state machines to request sending a response. This is signaled by ORing together the send flags from the response state machines. In the send state, the machine waits for a rising edge of the done signal from the spi_slave block. When this happens, the state machine goes in the send_done state, which is the state being checked by command-responding state machines in order to advance. In this last state, the transmitting state machine checks the send flag to see if there are still words to be sent, and if not it returns to the idle state. Otherwise, it goes back into the send state in order to send another word via the communication channel.

The reception state machine is shown in Figure 5.18; it checks a received word from the spi_slave block and sets each flag upon detecting one of the commands. Generally, the flags for each command are active for one clock cycle in order to start another internal state machine. There are two exceptions from this: the flag corresponding to the MEMFMT command, which toggles every time the command is sent, and the flags corresponding to MODE_SURVEY and MODE_HIGHRES commands; each of these commands
Table 5.5: Command flags in the `spi_ctrl` block

<table>
<thead>
<tr>
<th>Flag</th>
<th>Purpose</th>
<th>Set by</th>
<th>Cleared on</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flg_mag</code></td>
<td>Start the ACS state machine</td>
<td>ACS command</td>
<td>Next clock cycle</td>
</tr>
<tr>
<td><code>flg_mem</code></td>
<td>Start the MEMRD state machine</td>
<td>MEMRD command</td>
<td>Next clock cycle</td>
</tr>
<tr>
<td></td>
<td>when no other memory operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>are executing</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>flg_memaddr</code></td>
<td>Start the MEMADDR state machine</td>
<td>MEMADDR command</td>
<td>Next clock cycle</td>
</tr>
<tr>
<td><code>flg_format</code></td>
<td>Start/stop the memory formatting process</td>
<td>MEMFMT command when no other memory operations are executing</td>
<td>The next MEMFMT command</td>
</tr>
<tr>
<td><code>rewind_tostart</code></td>
<td>Rewind memory to first page</td>
<td>REWIND_TOSTART command</td>
<td>Next clock cycle</td>
</tr>
<tr>
<td><code>rewind_onepage</code></td>
<td>Rewind memory back one page</td>
<td>REWIND_ONEPAGE command</td>
<td>Next clock cycle</td>
</tr>
<tr>
<td><code>flg_lowsrate</code></td>
<td>Change between sampling rates</td>
<td>MODE_SURVEY command</td>
<td>MODE_HIGHRES command</td>
</tr>
<tr>
<td><code>flg_who</code></td>
<td>Enable a 'whois' response</td>
<td>Reception of the '?' character</td>
<td>Next clock cycle</td>
</tr>
<tr>
<td><code>flg_loopb</code></td>
<td>Loopback unknown command</td>
<td>Reception of unknown command</td>
<td>Next clock cycle</td>
</tr>
</tbody>
</table>

Table 5.6: Response flags in the `spi_ctrl` block

<table>
<thead>
<tr>
<th>Flag</th>
<th>Signals</th>
<th>Set on</th>
<th>Cleared on</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flg_sendmag</code></td>
<td>ACS command is responding</td>
<td>ACS state machine in sending and checksum states</td>
<td>otherwise</td>
</tr>
<tr>
<td><code>flg_sendmem</code></td>
<td>MEMRD command is responding</td>
<td>MEMRD state machine in sending and checksum states</td>
<td>otherwise</td>
</tr>
<tr>
<td><code>flg_sendaddr</code></td>
<td>MEMADDR command is responding</td>
<td>MEMADDR state machine in sending state</td>
<td>otherwise</td>
</tr>
<tr>
<td><code>flg_who</code></td>
<td>Enable a 'whois' response</td>
<td>Reception of the '?' character</td>
<td>Next clock cycle</td>
</tr>
<tr>
<td><code>flg_loopb</code></td>
<td>Loopback unknown command</td>
<td>Reception of unknown command</td>
<td>Next clock cycle</td>
</tr>
</tbody>
</table>
respectively clears and sets a latch whose output represents the low_sample_rate output. In order to make sure that conflicting commands are not executed by the FPGA, e.g., that a format is not performed while the memory is reading or vice versa, one flag is not toggled if the other is set. To complete the previous statement, the command flag for the MEMFMT command is thus only set if the memory is not being accessed for reading or writing. The same holds true for the other two memory operations.

The reception state machine waits in idle state until a rising edge occurs on the spi_slave_done signal. Because some commands respond with a series of words via the communication channel and it is undesirable to cut a command’s execution at mid-point, the reception state machine also checks that no previous commands that require the communication channel are under execution, by checking that the transmitting state machine is in its idle state. If all these conditions are true, the received SPI word is latched into a register, the receiving state machine advances to the received state, where it checks the command, sets the corresponding command flag and returns back to the idle state on the next clock cycle.

There is one extra command that can be used for debug purposes. The FPGA responds to a 'whois' command (character '?' with the string "me". This can be used for quick debug of the proper operation of the communication protocol. In the case that the command is unknown to the FPGA, it is echoed back to the sender.

The state machine handling the ACS response is presented in Figure 5.20. It consists of three states, one in which it is waiting for its corresponding command flag to be set, one in which the three magnetic field components are successively routed to the tx_word output, and one state to receive the checksum from the checksum block and route it to the tx_word output.

When its command flag is set, the state machine goes from the idle to the send state, where an internal counter increments every time the transmitting state machine is in its send_done state. The
ACS state machine holds in the send state until this counter is equal to three, and thus magnetic field measurements on three axes have been sent.

At the same time it is routing the magnetic field data to the output, the ACS state machine also controls the checksum block into computing a checksum out of each magnetic field word. The enable signal to the checksum block is set while the state machine is in the send state and the word has not been sent (the transmitting state machine is in its send state). The checksum enable signal is cleared when the SPI word has finished sending, to inform the checksum-computing block that it can latch the new variable in its registers. This process is repeated for as long as the ACS state machine is in the send state. The checksum state is used to hold the ACS sending flag high long enough for the two checksum words to be sent via the communication protocol.

The most significant 16 bits of the 20-bit filtered data at the output of the 250-Hz filter are presented at the the mag_x, mag_y and mag_z inputs, thus filtered magnetic field data are sent as a response to the ACS command. The resolution of this data is \(2 \text{nT}/\text{bit}\).

The state machine handling the MEMRD command response has a similar behaviour to the one handling the ACS command. As can be seen in Figure 5.19, it consists of five states. Two clock cycles after the MEMRD command is decoded by the reception state machine, the MEMRD state machine goes into the read_page state if the memory is not written to or being formatted. In this state, it asserts a signal to the Flash memory controller, which in turn will send the page read command to the Hynix chip. Because the Hynix memory needs approximately 30 \(\mu\text{s}\) to execute the page read command and make data available, the MEMRD state machine waits for 64 \(\mu\text{s}\) in this second state. A more than double waiting time is used to make sure no errors occur. The state machine then goes to the read_byte state, where the read_byte output signal to the Flash controller is toggled twice and an SPI word is built out of the two bytes received from memory. In the send state, the MEMRD state machine waits for sending to be complete (for the transmitting state machine to be in the send_done state) and increments its corresponding word counter. The state machine then returns back to the read_byte state to build another word. This process is repeated until 1024 words have been sent to the FMCU via SPI, at which point the state machine goes to the checksum state.

Control of the checksum block while sending memory data is done in exactly the same way as for the ACS state machine. The only exception is that for the MEMRD command response state machine, the checksum is computed out of 1024 words. The checksum state is maintained as long as it takes to send the two checksum words. After this, the MEMRD command returns to the idle state.
5.4 Memory Storage

An older design used for storage of data to memory was modified in order to store filtered data values to the Hynix Flash chip. A block diagram of the filter storage hardware is presented in Figure 5.22. The sample_combine block is tasked with decomposing filter input values and sending them to the Flash memory buffer byte-by-byte. The buffer gradually fills up to the length of one page and when that happens, the AFULL (almost full) output is set, which triggers the FPGA flash memory controller, which in turn will start sending data to the Hynix chip.

Based on the current value of the low_sample_rate signal, the sample_combine block multiplexes the 250-Hz or the 10-Hz filtered data for storage to memory. The done signals from the two filters are multiplexed in a similar manner. Time data is also stored to memory once per second, and this data are interleaved with filtered memory data.
A data packet in memory consists of eight bytes. The most significant four bits of the packet are the packet header, which describes the information that is to be stored. Depending on the type of the packet, the 60 least significant bits (LSBs) of the packet are either three 20-bit vectors containing magnetic field data for the three components in the case of a data packet (Figure 5.23), or, in the case of time packets (Figure 5.24), 16 LSBs contain the current *seconds* counter, padded with zeros up to the 60-th bit.

The *sample_combine* state machine (Figure 5.25) is made up of three states. In the *idle* state, the machine waits for either a rising edge on the multiplexed *done* signal from the filters, or for the LSB of the *seconds* input to toggle. Magnetic field data is given priority over time data, and thus if both these conditions occur at the same time, magnetic field data is stored and time data is discarded. When one of the two events occurs, the state machine goes into the *data* or *time* state, depending on which type of data it is due to store. In both states, a 3-bit counter is used to count to the maximum number of 8 bytes that a packet consists of. In the *data* state, multiplexed filtered values are successively routed to the *sample* output byte by byte. The same occurs in the *time* state and with the *seconds* input. In both states, the *WE* output signal to the Flash buffer is set while counting up to the eighth byte, and cleared.
when the eighth byte is sent.

After compiling the design, the `sample_combine` block is placed in 188 cells of the total 6144 on board the Actel 250-kgate FPGA (see Table 5.7).

### 5.5 Synthesis and Place-and-Route Results

Upon full compilation of the design, the area consumptions (in number of cells) per design are listed in Table 5.7. The table lists the consumption of blocks designed within the scope of this thesis. The numbers in the table are subject to change from compilation to compilation, due to heuristics used by the synthesis and Place-and-Route tools.

<table>
<thead>
<tr>
<th>Component</th>
<th>Number of cells</th>
<th>Percentage of total</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filt_250</code></td>
<td>969</td>
<td>15.8%</td>
</tr>
<tr>
<td><code>filt_10</code></td>
<td>1164</td>
<td>18.9%</td>
</tr>
<tr>
<td><code>spi_slave</code></td>
<td>76</td>
<td>1.2%</td>
</tr>
<tr>
<td><code>spi_ctrl</code></td>
<td>619</td>
<td>10%</td>
</tr>
<tr>
<td><code>sample_combine</code></td>
<td>188</td>
<td>3%</td>
</tr>
<tr>
<td><code>checksum</code></td>
<td>186</td>
<td>3%</td>
</tr>
</tbody>
</table>

As can be seen, most of the area consumption is due to the two filters and the usage of the multiplier operator in VHDL. Changing the design to use either a serial multiplier, or a more area-efficient multiplication would yield better results in that sense.

The total design, including the compensation algorithm, Flash memory controllers, health care blocks and system clock generation circuitry, occupies 89% of the total 6144 cells on-board the Actel 250-kgate FPGA.
Chapter 6

Flight Software Design

While most of the flight software for the SWIM mission is developed by the team in Puerto Rico, the implementation of a software library for handling the data sent by the SMILE magnetometer was established as part of the contributions of the KTH team.

The SMILE interface software was designed as a software stack with three levels – see Figure 6.1. The lowest level in the hierarchy is the hardware, or low-level layer; it contains routines which access the FMCU’s registers for configuration of the various peripherals. Above this lies a translation layer which offers generic routines for accessing low-level routines. The final application layer defines a set of buffers for storing the various data sent by the SMILE FPGA along with a set of application routines to be called by the main software of the satellite. Application layer flags ensure that no conflicts exist between commands accessing the same component of the SMILE hardware. The application layer makes calls to the translation layer routines for accessing processor registers and sending the various commands defined in Table 3.2. These calls then translate into calls at the low-level layer.

![Figure 6.1: SMILE software stack](image)

Because of this organization, the interface software is modular [17]. Communication interfaces, drivers and even processors may in principle be interchanged by changing the translation layer routines to point to those of the actual communication driver in use. The SMILE software stack was designed to be used either under a simple infinite-loop-type embedded application, or as part of a real-time operating system (RTOS), as is often the case for satellites.

All the FMCU code for interfacing the SMILE instrument was written in the C programming language using the Microchip MPLAB [24] integrated development environment. Table 6.1 lists the C header files that define the interface routines for each layer in the software stack.

The following sections provide descriptions of the software stack layers.
### 6.1 Hardware Layer

The hardware layer is the lowest layer in the software stack. This layer defines low-level C routines which access the processor registers directly for configuring the various peripherals used in communicating to SMILE.

The communication protocol in use at the time of writing of this report is the SPI protocol, as specified in Section 3.2. Peripheral module SPI2 on the dsPIC33F is used for sending the various commands to the SMILE FPGA.

### 6.2 Translation Layer

The second layer in the SMILE software stack is the translation layer. It is implemented in the C header file `smile_translation.h` and it translates the low-level routines to the higher application layer. Its implementation in the form of C `#define` statements, or macros, means that it occupies virtually no code space in memory, due to direct compiler translation of macros [29].

In order to change the low-level routines, the macros in the `smile_translation.h` file should be changed with new definitions as appropriate. Macros are named starting with the `smile_<peripheral>` prefix, where `<peripheral>` is the peripheral to be accessed by the macro. Table C.1 in Appendix C shows routine prefixes used in the macros defined in the `smile_translation.h` file. When changing the translation layer macros to point to custom functions, the reader should refer to the guidelines presented in Appendix C and comments in the translation layer C file and use this as a model.

### 6.3 Application Layer

At the top of the software stack lies the application layer software. Its main purpose is sending the commands defined in Section 3.3. Apart from a set of application functions that are called by the main FMCU software to send requests to the FPGA, it defines a set of buffers for storing data received from the SMILE FPGA upon request.

Three buffers are defined in software for handling data sent by the FPGA. One 1026-word buffer stores the 2048-byte page data sent by the FPGA upon request of retrieving stored scientific data. The last two words in the buffer are the checksum (see Section 3.3.7) computed by the FPGA and checked by the FMCU to ensure correct data transmission. A five-word buffer is used to store current magnetic field readouts in use for the ACS system. As for the scientific data buffer, the last two words represent the checksum. Finally, one two-word buffer is used to store the current status of the FPGA memory handler and controller.

A set of application flags are also defined at this level. These flags are useful when using interrupt-based communication on the FMCU side. The flagging mechanism makes sure that no conflicting requests are made to the FPGA, e.g., that a memory erase is not requested while the FPGA is processing and replying to a `MEMRD` command. Similarly, to make sure that data sent by the FPGA is not stored in the wrong buffer, e.g., a `MEMADDR` command cannot be sent if a `ACS` command has not finished...
executing.

Figure 6.2 shows the SMILE application layer flags. They are implemented in the form of bits of a C char variable [29]. The appropriate flag is checked by an application layer function prior to sending a command. If the flag is set, the function returns an error and does not send the command to the FPGA. If on the other hand the flag is not set, the function sends the command to the FPGA and sets a flag. The flags are cleared inside the main FMCU software after the command has finished execution and data were received. This mode of operation avoids sending conflicting commands to the FPGA.

Each flag corresponds to a certain command being sent to the SMILE FPGA. Thus, the ACS flag is set when sending the ACS command, the MEMRD flag is set when sending a MEMRD command, and so on. The MEMADDR, MEMFMT and MEMRD flags are OR-ed together to make the memory flag which is checked by all functions performing memory operations. Since the MEMADDR, MEMFMT and MEMRD, respectively, are commands that cause the FPGA to access the Hynix Flash memory on the SMILE board, these functions check the memory flag before sending the command. The flag is also checked before sending the REWIND_ONEPAGE and REWIND_TOSTART commands. In a similar manner, the ACS, MEMRD and MEMADDR commands receive replies through the communication channel, their respective flags are OR-ed together to make up the communication flag. This flag is checked by functions sending these commands to the FPGA.

A separate record flag is set by the function sending the record command. In the context of the SWIM mission, the record command is sent via an MCU pin (see Table B.2), but the application function can be easily modified to allow for sending a command via the communication channel. The purpose of this flag – in the context of the SWIM mission – is, as shall be seen, to start or stop recording to memory. If the flag is set, recording is started; if it is cleared, recording is stopped. In later missions where the flagging mechanism is used, this flag can also be checked before sending memory commands, to make sure no conflicting commands are sent, or that the application layer software does not fall in an unknown state, where some flags are set and others are not.

Since interrupt-based communication is not used in the SWIM mission, the flagging mechanism is not used. It is however a useful tool when using interrupts to store data received from SMILE, and can be used in future missions.

<table>
<thead>
<tr>
<th>bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMADDR</td>
<td>MEMFMT</td>
<td>MEMRD</td>
<td>ACS</td>
<td>RECORD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2: The SMILE application flags (smile_flags) variable

6.4 Main Task Function

The main task function is located at the application level and uses the application-layer functions. Its purpose is to give an example of how the application-layer functions can be used. For the SWIM mission, it has been written with the intent of being part of actual flight software. In its operation, it can actually be thought of as a fourth level in the abstraction layer (Figure 6.3).

The SMILE software is thought of as being part of a stand-alone task in an RTOS executing on the FMCU of the SWIM satellite. It has in this sense been implemented as a task in the FreeRTOS operating system, which is the chosen RTOS for the SWIM mission at the time of writing of this document.

A flowchart diagram of the software task execution is presented in Figure 6.4. Upon RTOS initialization, the SMILE software stack is initialized and a ”soft” reset is performed via the SMILE reset pin. After this, the task goes in its infinite loop, where it ”takes” a binary OS semaphore. As a prime synchronization mechanism of the OS, a task that ”takes” a binary semaphore blocks until this semaphore is ”given” by either another task or an interrupt service routine. Many RTOSes give the possibility to pass data between tasks via a void pointer variable which is re-cast internally to the task. The idea
used in implementing the SMILE task is that another task running under the RTOS would “give” this semaphore, along with sending a specific command to the task via this void pointer variable when information is needed from SMILE. The SMILE task code then executes the procedures required under the communication protocol.

The commands to be sent to the task correspond to those in Table 3.2. These are defined in code via the SMILE_CMD macros in the smile.h file. The void pointer variable shall from now on be referred to by its actual name in code, taskcmd.

To execute the ACS command (Figure 6.5), the task sends the command to SMILE via the high-level application function. A series of five "dummy" transfers assure that five SPI words are received from SMILE, containing to the magnetic field data and the checksum. (Dummy transfers are necessary because of the operation of the SPI protocol on the dsPIC side. Writing a redundant value to the SPI register initiates a transfer and thus causes the FMCU to drive the SCLK line).
Based on the received data, the task calculates its own checksum and compares it to that received from the FPGA. Should the checksum be incorrect, the command is resent. This process is repeated three times; if for three times the checksum was incorrect, a soft reset is performed on SMILE and this is communicated to the calling task via the taskcmd pointer. The task that previously activated the SMILE task in need of information should therefore check this pointer for correct operation of the SMILE task.

The MEMRD operation executes in a much similar manner (Figure 6.7). Prior to executing it, however, the memory status is requested from the FPGA, to make sure no memory operations are executing at the time. If the formatting and recording bits in the FPGA response presented in Figure 5.21 are set, the memory address is read. This shows how many pages of memory have been written during the latest scientific mission, and thus how many pages should be read from the Flash memory. The memory is rewound via the REWIND_TOSTART command, and a number of MEMRD commands equal to the received FPGA memory address counter are then issued to retrieve data from SMILE into the memory data buffer. A waiting period of 70 µs is used after the MEMRD command, to allow for the 64 µs waiting period of the spi_ctrl state machine described in Section 5.3.3. After this, a series of 1026 "dummy" transfers store the data sent by the FPGA. The checksum is verified at the end, and if correct the data stored in this buffer are then copied to the satellite SD card for downlink back to Earth and the next MEMRD command is issued to the FPGA. This process is repeated until the maximum memory read at the beginning of the MEMRD cycle has executed, assuming of course that the read checksums are approved.

If on any occasion the checksum is not correct, the REWIND_ONEPAGE command is sent and the page read is re-attempted. Should the checksum be wrong a number of three times, the SMILE FPGA is reset and the memory reads are aborted. This fact is also signaled by the task via the taskcmd pointer.

Figure 6.5: Flowchart for ACS command
Figure 6.6: Flowchart for MEMFMT command

Formatting (Figure 6.6) is performed by first requesting the status from memory, checking that the recording flag is not set in the response words, and then issuing the MEMFMT command to the FPGA. Because, as seen in Section 3.3.3, formatting the whole 4 Gb of data on the Hynix chip takes 12.3 seconds, the task then enters in a blocked state for 13 seconds. After this time has passed and the task is placed in a running state by the RTOS kernel, the MEMFMT is re-issued to stop formatting inside the FPGA, followed by a REWIND_TOSTART command, to return the memory to a zero state and allow for the next round of data to be stored from a known location.

In order to start and stop recording of data to memory, the task requests the status, checks the recording bit in the status array, and sets the pin if the flag is cleared, or clears the pin otherwise. The RECORD SMILE application flag is also set or cleared simultaneously, based on the operation performed. This flag is checked prior to sending one of the REWIND commands, since rewinding the memory while data are stored would lead to corrupted data.
Execute command

MEMSTAT

Memory operation undergoing?

Y

store max address
REWIND_TOSTART

N

MEMRD
wait 70 us

Dummy send

Sent/Received 1026 words?

N

Checksum good?

Y

Complete

N

Checked 3 times?

Y

Max address reached?

N

Reset SMILE

Inform application

Figure 6.7: Flowchart for MEMRD command
Chapter 6. Flight Software Design

6.4. Main Task Function
Chapter 7

Testing and Validation

A mixture of testing strategies were used to validate the proper functioning of the various modules that the system comprises. Not all parts of the system were available at all times, so they had to be in some way simulated, or solutions were needed to fill their absence.

This was the case with the initial stages of testing the software stack during the time spent in Puerto Rico. Since a SMILE board was not available, SMILE was simulated using another microcontroller board. A similar situation occurred while developing the SPI communication modules for the FPGA. Because a board was not available at the time, and all boards available had hardware issues which made it hard to test the SPI protocol, another development board from Digilent was used to test it out.

A test-while-developing strategy was generally employed throughout the project. By finding errors and gradually fixing them, a functioning system yielded. This chapter presents testing methodologies for each of the parts comprising the system and described in previous sections.

7.1 SMILE Board

Different steps were taken for testing the SMILE board. Power regulator circuits and digital components (FPGA, Flash memory) were first soldered and tested for primary functionality. Before applying power, a short-circuit test was performed using a simple multimeter; after this, voltage was gradually applied at the power leads of the board. Once it was established that all power regulators worked as they should, the power-on reset circuitry was tested for correct operation, followed by testing of the on-board oscillator.

Simple logic was then implemented in the FPGA to test that it operates correctly. This included simple logic gates and clock divider circuits to check correct reception of the clock signal from the oscillator. Next, the Flash memory was tested by a simple VHDL state-machine which would first format the memory, then write the value $01h$ to one page, rewind, read back the page and test for correctness.

Resistors R15 through R18 and R25 and R26 were then soldered on the board, enabling access to SPI2 and UART1 modules on the FMCU, and the SPI slave and UART blocks were programmed into the FPGA. This part of the testing was done on the CSK development kit together with the FMCU, which was driving signals on the communication buses; the FPGA was configured to loop back characters received from the FMCU. The test served a double purpose: first, to test if these CSK bus connections were properly made on the board, and second, to test the SPI communication controller implemented in the FPGA.

The analog portion of the board was soldered after this and a LEMI sensor was connected to the board. An older SMILE design was programmed into the FPGA and the compensation mechanism was tested. This revealed some errors in the values of the resistors soldered on the board, which caused saturations in the zero-field of some axes due to too large pick-up signal amplification. These resistors were changed, and after it was determined that the compensation mechanism worked correctly for the fluxgate
magnetometer, it was proceeded to calibrating the SMILE sensor for the SWIM board. Calibration of the sensor is described in Section 4.3.

7.2 FPGA Hardware

Testing for the FPGA hardware was done in several steps. Before any logic was implemented in the FPGA, the VHDL code was simulated for correct operation using the Modelsim sofware. Logical code errors were in this way eliminated before downloading a bit file to the FPGA. Once the operation of a device under test was validated in simulation, the actual hardware was programmed into the FPGA and tested for correct operation.

The following subsections list steps taken to test out the actual FPGA hardware.

7.2.1 DAC Signal Processing

For testing the filter operation, the UART FPGA core used for communication in previous missions was first included in the system to send data to the PC for analysis. Then, the filter was included in the system and modifications were made to the UART core so as to enable the filter and start sending filtered data when the RECORD command was sent and memory was enabled. The second RECORD command disables the sending of filtered data. In the testing phase of the filter, the RECORD command was performed by sending the character ‘N’, instead of through the RECORD pin.

Unfiltered data were at the same time stored in the Flash memory. The data format for Flash storage used during the testing phase for the filter is shown in Figure 7.1. Data were sent from the SMILE FPGA to the PC in series of ASCII characters. Data in memory were organised in series of 16 bytes, each nibble of a byte being represented by an ASCII character in the FPGA output. Line feed and carriage return characters were sent at the end of this series of characters to aid in human readability. Thus, the output format to PC is 32 characters followed by line feed and carriage return.

Each of the 16-byte series contains six bytes of DAC output values – two bytes per axis – after the correlation and integration processes, six bytes containing a different shift for the compensation readings, and four other bytes containing timekeeping data. Since a page is 2048 bytes long, every page contains 128 16-byte values.

![Figure 7.1: Flash memory storage format for filter test phase](image)

The six bytes corresponding to the values for DAC compensation were relevant in this case, since they are the ones that are input to the filter. Every page in memory contained compensation values for each axis across 128 excitation periods. Considering the 250 samples per second sampling rate for the satellite and implicitly the filter, 32 unfiltered DAC readings correspond to one filtered reading. Therefore, one memory page of data contained the unfiltered values corresponding to 4 filtered values.

The following steps were followed in order to test the filtered data.

1. Start collecting data (enable the memory). Filtered readings are sent through UART once per filtering period. The unfiltered readings are at the same time stored to memory.
2. Store a sufficient amount of filtered values and then stop the reading. This also stops sending of filtered values.
3. Read $N$ filtered values starting from the first value sent.
4. Rewind the memory to zero and read $N/4$ pages.
5. Plot the spectra of filtered data for validation.

Due to the datapath being multiplexed between the three axes, it was necessary to first determine that the hardware filter worked correctly and with the right samples. Raw data read from the filter were in this purpose stored to a text file using the RealTerm serial terminal software. Filter samples were sent to the PC in a format similar to that of Flash memory data, except that for each filtered value, 8 characters per dimension were sent, followed by the same timing value sent at the middle of a memory sample. The filter’s outputs were sent unquantized to check for correct calculations and sign-extended to 32 bits to aid conversion from the ASCII to the numeric format. Figure 7.2 shows the filter data format sent to the PC during the testing phase.

Memory was then rewound to zero and unfiltered data were stored to another text file using the same serial terminal software. This file was read and the necessary data were stripped from each 32-character memory sample using Matlab. Converting from the hex value sent in ASCII to a numeric representation and using fixed-point arithmetic in Matlab via fixed-point ($fi$) objects, convolution computations were performed using the same coefficients as for the hardware filter. This assured computation using two's complement algorithms, instead of floating-point algorithms. Simultaneously, the text file containing filtered data was read and the data for each coordinate converted to a numeric value. At the end of this process, both Matlab-calculated values and those sent from the hardware filter were plotted, along with the difference between the two.

Figures 7.3 through 7.5 show the difference plots for each of the axes using 8192 filtered-value samples. As can be seen, for almost all the sampling time, the difference between the calculated values and actual hardware filter values is zero. There is only one discrepancy between the two occurring for the Y and Z axes. This discrepancy is most likely due to the terminal software inducing an error while storing to file when reading memory values through serial port. This in turn causes Matlab to convert the Y and Z axes readouts erroneously for that one sample and the computation to be off. The rest of the values are however the same, which led to the conclusion that the filter computations were performed correctly.

Noise spectra were then generated on filtered data gathered with the magnetometer inside a magnetic shield. The spectrum for the X axis is shown in Figure 7.6. It was in this way determined that the filter...
Chapter 7. Testing and Validation

7.2. FPGA Hardware

Figure 7.4: Y axis plots for data retrieved from the FPGA and calculated by Matlab

Figure 7.5: Z axis plots for data retrieved from the FPGA and calculated by Matlab
attenuated DC frequencies too much. This was due to the fact that the coefficients were not generated in Matlab so as to have unity gain at DC frequencies. The filter operation was however approved in this case and it flew in the SMILE design of the PoGoLite refly in July 2012. Results of the mission and actual operation of the device are still pending.

Using the new coefficients, filter spectra were again generated, but this time it was determined that the filter cannot be properly tested without the use of a separate voltage source in the form of a battery. Some 9 V batteries were used to collect data, but these did not have high enough capacities and they were depleted before conclusive testing and validation of the new filter coefficients could be performed.

7.2.2 Communication Protocol

Since the SPI communication controller consists of multiple blocks working together, each of the blocks was successively introduced in the FPGA design and tested. A Xilinx Spartan-3 FPGA on a Digilent Basys [13] board was used in the first stages of development, due to lack of a fully-usable board with an Actel ProASIC3 FPGA. Most of the commands were in this way developed and tested. The first stage of testing (that for the SPI slave block) was done with a Digilent chipKIT Max32 [20] Arduino-type board, followed by using the CSK development kit for communicating to the FPGA. The chipKIT was used to validate proper operation of the SPI slave block, by sending words through SPI and outputting the looped-back words sent by the FPGA to a PC.

On the Actel FPGA, this part of the design was tested exclusively on the CSK development kit and the SMILE board. The FMCU was configured to receive commands from the PC via the UART2 module and route these commands to the SPI2 module. Responses received from the FPGA were sent through the same UART module back to the PC.

First, the spi_slave block was tested via a simple loopback between the FMCU and the FPGA. After soldering resistors R15 through R18 on the SMILE board, the FMCU software was configured to send successive ASCII characters to and read the received characters from the FPGA via SPI. The sent and received characters were simultaneously sent to the PC via UART; the sent character was output first, the received character second, followed by carriage return and line feed characters. The terminal output in Figure 7.7 shows how the character sent by the FMCU when sending the first word is received from
the FPGA when sending the second, and thus that the SPI slave works correctly.

![Terminal output while testing the spi_slave block](image)

Figure 7.7: Terminal output while testing the `spi_slave` block

The `spi_ctrl` block and the command response mechanism was then tested. In order to test the sending of memory pages upon request via the `MEMRD` command, a simple hardware block was implemented in VHDL code to mimic the sending of bytes by the Flash memory controller. This block contained a state machine with two states, shown in Figure 7.8. Triggering from idle state to `counting` state was done by the `spi_ctrl` block setting the `read_page` signal high; the memory simulator then incremented a modulo-8 counter on rising edges of clocks when the `read_byte` signal was high. Since the `read_byte` signal is negated in the memory controller block, this behaviour mimics the operation of the Flash memory, which outputs a byte on every falling edge of the `RE` signal (see Figure 11 in [19]). The state machine of the memory simulator went back into the idle state after sending 2048 bytes.

The `MEMRD` command was then sent to the FPGA and it was checked first that a number of 2048 bytes were received as a response, and second that these 2048 bytes contained 256 series of modulo-8 counts, corresponding to what the memory simulator output.

Once this behaviour was confirmed, it was time to introduce the SPI controller inside the SMILE design. The ACS command was tested mainly by sending a DAC excitation word at the end of the three compensation words (see Figure 7.9), due to the fact that the compensation values changed from one reading to the next because of the compensation algorithm. This excitation signal was of constant value, so by checking that every time the command was sent, the word corresponding to the excitation signal was the same, proved that the sending of magnetic field values was correct and no logical errors were present in the response state machine, leading to words corresponding to a magnetic field reading being shifted.

Memory readout was tested by loading a version of the design that used UART for communication. The FMCU’s UART1 port was used to communicate to the FPGA, and UART2 was used to mirror the FMCU-FPGA communication to the PC. The proven UART communication control on the FPGA was used to format the memory and enable storage of data to memory. Values written to memory were then read and stored to a text file. Then, the SPI design was programmed back into the FPGA. Because the Hynix chip is a Flash memory, values written with the UART FPGA design remained, so the memory was
Figure 7.9: Reply to the ACS command during the testing phase. Each block is 16 bits wide.

read once again, this time using SPI. These readings were stored to a second text file and the contents of the two files were compared using Matlab; this yielded that values read via SPI were the same as those read via UART.

To conclude that the SPI controller worked as intended, the process was reversed, by first erasing, storing to memory and reading stored values via the SPI controller, and then downloading the UART controller design to the FPGA and re-reading the stored values. As expected, the values read via UART were the same as those read via SPI, which proved that the SPI controller worked properly.

The SPI communication speed was also tested using the FMCU and the flight software stack; tests were successful up to a data rate of 2 Mbps, which was the maximum clock that the FMCU could supply under the configuration that was used during this test. Since the on-chip PLL of the dsPIC33 was not being used, the core frequency could not be set higher than 4 MHz, which yielded a maximum settable frequency of 2 MHz for the SPI peripheral. However, it is expected that the SPI communication protocol would work up to the 15 Mbps [12] supported by the ADIS16488 chip.

7.3 Flight Software

From the beginning of this project, the plans were to develop a modular software stack that could be used with different hardware interfaces at the lower level. Since the considered protocol in use at that time was the UART protocol, software development and testing thus began in this sense.

Upon writing of the software stack functions, these were tested in collaboration with the team in Puerto Rico. Tests were performed to see that no logical errors existed, and correct them if they did. For these tests, the same Digilent chipKIT Max32 board was used, due to ease of development of software for it. The SMILE command response mechanism was simulated on this test board and used to check the software stack. The flagging mechanism was tested in this way and proved to be functioning properly.

After this, it was proceeded to adapting the software stack for SPI. Confirmation that the FMCU-side SPI mode corresponds to that of the ADIS16488 IMU was first made by scoping the FMCU pins with an oscilloscope while transmitting through SPI. The rest of the stack was then gradually introduced in the system and tested in conjunction with the Digilent Basys board, followed by the SMILE board when available and the SMILE Actel FPGA.

Once the FPGA hardware was deemed to be answering to commands in a proper manner, FreeRTOS was loaded to the FMCU and the application task was written. Gradual testing-while-developing was then used to validate the software stack to a final working version.

7.4 Integrated System

As can be understood from the rest of this chapter, the most significant part of testing was actually done on a system much resembling that of the satellite.

What has not been tested, due to lack of various resources, is the operation of the temperature measurement circuit, and the burnwire circuit. These could not be tested due to lack of the OPA170 operational-amplifier at the core of the circuit. Similarly, due to the lack of a boom deployment mechanism, the burnwire circuit was unable to be tested. However, in both these circuits’ cases, simulations were performed using a PSPICE tool from Linear Technology, to validate circuit operational parameters.

Before the testing period was concluded, one final test was performed in order to validate the operation
of the system as a whole, as well as the modularity of the software stack. The old UART controller FPGA core was modified to reply to commands in the way that the SPI controller does. The software stack was then modified so as the routines that the translation layer pointed to were UART driver routines. UART1 on the dsPIC FMCU was used to communicate to SMILE, as was used in testing the operation of the SPI communication protocol (Section 7.2.2). Small modifications were afterwards made to the SMILE task running under FreeRTOS, and the software was afterwards run. By checking the command responses using a terminal software on the PC as in Section 7.2.2, it was in this way validated that the software stack was indeed modular with respect to the communication interface used.
Chapter 8

Conclusions and Future Work

8.1 Conclusion

Implementation of a data handling system for interfacing the SMILE digital fluxgate magnetometer to a CubeSat mission was performed and described in the course of this work. The data handling system consists of FPGA hardware answering to commands sent by the FMCU and defined in Section 3.3.

Modularity was one of the key sought-after aspects throughout the design process. From the beginning, emphasis has been put on not only developing a system that can be interfaced to the SWIM satellite, but instead developing a platform that can be re-used in future CubeSat missions and extended according to need.

On the FMCU side, a fully-functional and modular software architecture has been implemented. It comes in the form of three layers, low-level layer for accessing processor registers and which should in principle allow interchanging communication protocols and even processors altogether. The translation layer above this is used to connect an application layer to the low-level layer. The application layer consists of functions to be called for sending the various commands to the FPGA. A set of buffers for storing data received from the FPGA are defined at the application layer. Application-layer flags are also defined at this level and can be useful in the event that interrupt-based communication is desired. The flagging mechanism is not used in the context of the SWIM mission.

The SMILE software stack has been designed to be used either as part of a simple embedded application running within an infinite loop, or as part of a stand-alone task within a real-time operating system. In this sense, a task running within the FreeRTOS operating system has been developed and used for validating the proper functioning of the SMILE communication protocol. This task has been developed so it can be used with as few modifications as possible within the flight software. It can also serve as an example of usage of the SMILE application-layer functions.

A new board was designed as part of the work for this thesis. Much of the board was inherited from the SMILE design used in the SQUID REXUS project. Apart from adapting the older design to the Pumpkin CubeSat Kit board specifications, additions were made to this design in order to add the functionalities required in light of the SWIM mission. Furthermore, the board has been designed with the possibility of its usage in future missions as well. Access has been provisioned to various pins on the Pumpkin CubeSat Kit bus allowing the use of different communication protocols by soldering the appropriate 0Ω resistors. The digital components on the board have been tested, as well as the analog part pertaining to sensor pick-up and compensation.

In addition, a scalable and re-usable digital filter structure with downsampling was implemented. A 250-Hz filter offers downsampling from an input frequency of 8 kHz, corresponding to the SMILE excitation frequency. Data at the output of this filter are stored to memory in high-resolution mode operation. Filtered data are also returned when magnetic field readouts are requested for the satellite.
attitude control system. Another 10-Hz filter with a structure much similar to the 250-Hz version offers a further downsampling to the 10 Hz frequency defined during survey mode operation. Selection between using one filter or the other is made via a command. One of the filters, the 250-Hz version, flew on the Albert balloon mission in late May. The filters could otherwise not be fully tested, due to lack of various hardware and a clean environment for the testing.

8.2 Future Work

Full testing and validation of the two filters is of course a primary concern and should be the first priority as to what concerns future work in this project. Fortunately, in case the coefficients are deemed incorrect, these can easily be changed due to the modular structure of the filter. Other possible future work here would be the inclusion of serial multipliers within the filter datapaths, in order to optimize the design for size.

The SMILE software RTOS task should be fully tested within the context of the flight software. Testing how the software task would perform with multiple other running tasks as would be the case in the SWIM mission should be one of the first tests performed on the software.

Finally, the addition of a cyclic-redundancy check algorithm for checking data sent from the FPGA to the FMCU can be performed. The SPI control hardware has been written in such a way as to allow easy addition of such an algorithm later on.
Bibliography


Appendix A

SMILE Board Schematics
Appendix B

CubeSat Kit bus connections on SMILE board
Appendix B. CubeSat Kit bus connections on SMILE board

Figure B.1: CSK bus connectors. The highlighted pins have routed connections.

Table B.1: Short-circuit resistors for enabling connections on the CSK bus

<table>
<thead>
<tr>
<th>CSK pin</th>
<th>Resistor</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1RX (UART)</td>
<td>R25</td>
</tr>
<tr>
<td>U1TX (UART)</td>
<td>R26</td>
</tr>
<tr>
<td>U2RX (UART)</td>
<td>R19</td>
</tr>
<tr>
<td>U2TX (UART)</td>
<td>R24</td>
</tr>
<tr>
<td>SCL1 (I2C)</td>
<td>R34</td>
</tr>
<tr>
<td>SDA1 (I2C)</td>
<td>R30</td>
</tr>
<tr>
<td>SCL2 (I2C)</td>
<td>R6</td>
</tr>
<tr>
<td>SDA2 (I2C)</td>
<td>R5</td>
</tr>
<tr>
<td>SCK1 (SPI)</td>
<td>R27</td>
</tr>
<tr>
<td>SDI1 (SPI)</td>
<td>R28</td>
</tr>
<tr>
<td>SDO1 (SPI)</td>
<td>R29</td>
</tr>
<tr>
<td>SSN1 (SPI)</td>
<td>R36</td>
</tr>
<tr>
<td>SCK2 (SPI)</td>
<td>R15</td>
</tr>
<tr>
<td>SDI2 (SPI)</td>
<td>R16</td>
</tr>
<tr>
<td>SDO2 (SPI)</td>
<td>R17</td>
</tr>
<tr>
<td>SSN2 (SPI)</td>
<td>R18</td>
</tr>
<tr>
<td>USER0</td>
<td>R49</td>
</tr>
<tr>
<td>USER1</td>
<td>R50</td>
</tr>
<tr>
<td>USER2</td>
<td>R51</td>
</tr>
<tr>
<td>USER3</td>
<td>R52</td>
</tr>
</tbody>
</table>
### Table B.2: SMILE CubeSat Kit bus pin connections

<table>
<thead>
<tr>
<th>CSK pin</th>
<th>FPGA Pin</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1.3</td>
<td>29</td>
<td>FMCU I2C2 data</td>
</tr>
<tr>
<td>J1.4</td>
<td>28</td>
<td>FMCU I2C2 clock</td>
</tr>
<tr>
<td>J1.13</td>
<td>28</td>
<td>FMCU SPI2 clock</td>
</tr>
<tr>
<td>J1.14</td>
<td>29</td>
<td>FMCU SPI2 MISO</td>
</tr>
<tr>
<td>J1.15</td>
<td>30</td>
<td>FMCU SPI2 MOSI</td>
</tr>
<tr>
<td>J1.16</td>
<td>31</td>
<td>FMCU SPI2 SS</td>
</tr>
<tr>
<td>J1.17</td>
<td>43</td>
<td>FMCU UART2 RX / FPGA UART TX</td>
</tr>
<tr>
<td>J1.18</td>
<td>44</td>
<td>FMCU UART2 TX / FPGA UART RX</td>
</tr>
<tr>
<td>J1.19</td>
<td>43</td>
<td>FMCU UART1 RX / FPGA UART TX</td>
</tr>
<tr>
<td>J1.20</td>
<td>44</td>
<td>FMCU UART1 TX / FPGA UART RX</td>
</tr>
<tr>
<td>J1.21</td>
<td>28</td>
<td>FMCU SPI1 clock</td>
</tr>
<tr>
<td>J1.22</td>
<td>29</td>
<td>FMCU SPI1 MISO</td>
</tr>
<tr>
<td>J1.23</td>
<td>30</td>
<td>FMCU SPI1 MOSI</td>
</tr>
<tr>
<td>J1.41</td>
<td>29</td>
<td>FMCU I2C1 data</td>
</tr>
<tr>
<td>J1.43</td>
<td>28</td>
<td>FMCU I2C1 clock</td>
</tr>
<tr>
<td>J1.47</td>
<td>81</td>
<td>CSK USER0</td>
</tr>
<tr>
<td>J1.48</td>
<td>80</td>
<td>CSK USER1</td>
</tr>
<tr>
<td>J1.49</td>
<td>79</td>
<td>CSK USER2</td>
</tr>
<tr>
<td>J1.50</td>
<td>78</td>
<td>CSK USER3</td>
</tr>
<tr>
<td>J2.1</td>
<td>32</td>
<td>SMILE record signal (from FMCU)</td>
</tr>
<tr>
<td>J2.2</td>
<td>27</td>
<td>Reset signal for SMILE (from FMCU)</td>
</tr>
<tr>
<td>J2.3</td>
<td>-</td>
<td>Boom deployment feedback signal (to FMCU)</td>
</tr>
<tr>
<td>J2.4</td>
<td>-</td>
<td>Temperature measurement signal (to FMCU)</td>
</tr>
<tr>
<td>J2.5</td>
<td>-</td>
<td>Boom deployment signal (from FMCU)</td>
</tr>
<tr>
<td>J2.13</td>
<td>31</td>
<td>SPI1 SS</td>
</tr>
<tr>
<td>J2.25</td>
<td>-</td>
<td>5V power supply</td>
</tr>
<tr>
<td>J2.26</td>
<td>-</td>
<td>GND</td>
</tr>
<tr>
<td>J2.30</td>
<td>-</td>
<td>8V power supply</td>
</tr>
<tr>
<td>J2.31</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

Adapting the SMILE Software Stack

This appendix offers guidelines on adapting the software stack for usage with either a new processor, a different communication interface, or custom low-level drivers. Comments in the SMILE software stack should be consulted for guidance when attempting to adapt the software stack.

Table C.1: Macro prefixes in the `smile_translation.h` file

<table>
<thead>
<tr>
<th>Macro prefix</th>
<th>Peripheral accessed</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>smile_io</td>
<td>I/O ports</td>
<td>Access input/output ports of the processor</td>
</tr>
<tr>
<td>smile_analog</td>
<td>Analog-to-digital converter</td>
<td>Access the ADC for temperature readout</td>
</tr>
<tr>
<td>smile_comm</td>
<td>Communication interface</td>
<td>Access the communication interface driver for sending and receiving data</td>
</tr>
</tbody>
</table>

The following generic steps should be followed when modifying the stack:

1. Implement drivers to access processor registers in the language of choice and provide C function prototypes in a file called `smile_lowlevel.h`. Make sure initialization functions with function name headers as specified in Table C.1

2. Change the functions in the translation layer to point to the low-level drivers. The `comm_put` and `comm_get` macros are necessary within the software stack, as they are used by the application layer to send commands to the FPGA.

   The `comm_transfer` macro is only relevant in cases where full-duplex communication is available and data is received at the same time as it is sent (i.e., SPI). In the case of other communication interfaces, this macro can be translated to a redundant check as below. This will avoid compiler errors of any kind.

   ```
   #define smile_comm_transfer(v) ( (v==1) ? 1 : 0 )
   ```

   In cases where no enable signal exists on the communication interface, the `comm_en` and `comm_dis` macros can be translated into NOPs as below (considering the MPLAB C30 compiler)

   ```
   #define smile_comm_en __builtin_nop
   ```

3. If the lower-level functions have initialization parameters, provide these in the form of macros at the translation layer. One example of this is given below. Here, the communication protocol initialization function has two parameters, defining the baud rate and the type of interrupt to use. These can be defined in a macro at the translation layer as below:

   ```
   #define smile_comm_init(baud_rate, interrupt_type) ...
   ```
Appendix C. Adapting the SMILE Software Stack

#define SMILE_COMM_INIT_PARAMS 115200,\nKTH_UART_USE_NOINT

Inside the software stack initialization function, the communication protocol calls the translation-layer macro with this parameter. In case the initialization function has no parameters, the macro can simply be left "open", as below:

#define SMILE_COMM_INIT_PARAMS

Both the I/O and analog driver initialization calls are provisioned with such a macro at the translation layer. The user should change these according to their lower-level function parameters.

4. Change the smile_data_t type definition in the smile.h file to have the number of bits expected on the communication channel.

5. Optionally change the smile_err_t type definition, if more errors are to be introduced. Such a case would however need modifications at the application layer as well, so it shall not be considered here.

6. Use the application-layer functions to retrieve data from SMILE.

The SMILE commands can also be changed by changing the SMILE_CMD_ macros in the smile.h file.