Design of an Embedded Readout System for the ALOFT Gamma-Ray Detector Instrument

A thesis by

Mats Fredrik Heigre

for the degree of

Master of Science in Physics

Department of Physics and Technology

University of Bergen

September 2018

Abstract

Birkeland Center for Space Science has proposed a campaign known as the Airborne Lightning Observatory for FEGS & TGFs (ALOFT) to study Terrestrial Gamma-Ray Flashes (TGFs). TGFs are the most energetic natural phenomena occurring in the Earth's atmosphere, and are important to our knowledge about the relationship between the Earth and space. The ALOFT campaign will use a gamma-ray detector instrument built by the University of Bergen which will be mounted to the NASA ER-2 High-Altitude Airborne Science Aircraft.

This work covers the design and development of the embedded software used to offload and operate the detector readout system of said instrument. A similar instrument was built and flown in 2017. The new instrument differs from this by being implemented on a System on a Chip (SoC) embedded platform, reusing relevant modules from the old instrument. The software has been implemented with the FreeRTOS Realtime Operating System (RTOS). Design considerations to limit complexity, and the impact of the radiation environment the instrument is to be operated in, has been performed trough implementation of a checksum algorithm, cyclic rewriting of registers, and modular design strategies.

A verification system has been realized with a prototype hardware setup, in which test systems has been added to process synthetic TGF-events in the software and hardware. Test with emulated data and a Telnet control interface has been successfully implemented. The current implementation focuses on modularity, and thus offers a very good framework for further development of the instrument when campaign specifications are decided.

Preface

The following work was accomplished at the Department of Physics and Technology at the University of Bergen in the period between August 2017 and September 2018. The work of this thesis marked an unofficial beginning of the development of ALOFT, and thus no information and documentation existed except for that of the FEGS-II campaign. Developing such a complex system was done without much prior experience, and no hands-on experience with the C programming language and embedded systems was present at the beginning of the work.

Acknowledgments

First and foremost, my two supervisors *Associate Professor Johan Alme* and *Professor Kjetil Ullaland*, have both inspired and influenced my work, in which i would like to express my deepest gratitude. During the work, a personal highlight was witnessing the UiB-built MXGS instrument being launched to the International Space Station. Getting to work in an environment with the passion witnessed from both the *space physics group* and the *microelectronics group* that day, has been one of the most rewarding experiences during my stay at the University of Bergen.

On the work related to the space physics, much of it would not have been possible without the help I received from *Associate Professor Martino Marisaldi* and *PhD Candidate Chris Alexander Skeie*. *Senior Engineer Shiming Yang* has provided invaluable insight into the FEGS BGO instrument, and *Researcher Boris Wagner* gave advices which helped during the steep learning curve of learning to program in a new language.

I would like to express my gratitude for my fellow co-workers whom I shared office with at room 312, and other PhD candidates from the microelectronics group. I hope I will be able to stay in contact with all of you as these two years have been a very enjoyable experience. I would also like to thank my family and friends for keeping my spirit high. Especially *Alexander Nikolai Nesse*, *Hogne Andersen*, and *Håvard Skibenes* have played central roles to this. An extra special thanks goes to *Chris Alexander Skeie*, *Magnus Rentsch Ersdal*, and *Martino Marisaldi*, for proofreading both during the drafting period and final revisions.

Most importantly, my lovely *Marie Aadland* has provided me with encouragement and support. Thank you for always being there. This work would really not have been without you.

Mats Fredrik Heigre

Contents

Preface ii
Nomenclature vii

Acronyms **ix**

1 Introduction **1**
1.1 History 1
1.2 Background and Motivation 2
1.3 Objective of this Thesis 2
1.4 Thesis Outline 3
1.5 Citation Principles 4

2 High-Energy Atmospheric Phenomena and the Radiation Environment **5**
2.1 Terrestrial gamma-ray flashes 5
2.2 Instrumentation 10
2.3 Research and Ongoing Campaigns 11
2.4 The Radiation Environment 14
2.5 Radiation Design Considerations for ALOFT 16

3 FEGS BGO Instrument - Readout System **19**
3.1 System Architecture 19
3.2 BGO Data Acquisition Unit 20
3.3 Readout Control Unit 21
3.4 Data Processing Unit 23
3.5 Operation of FEGS BGO during Flight Campaigns 25
3.6 Review 25

4 ALOFT Design **27**
4.1 System On a Chip 27
4.2 System Requirements 28
4.3 Xilinx Digilent Zybo SoC Trainer Board 30
4.4 FreeRTOS - A Realtime Operating System 31
4.5 Multithreading 32
4.6 Memory allocation 34
4.7 First Stage Boot Loader 34
4.8 Storage 35
4.9 User Interface 35
4.10 Cockpit Experiment Control Panel 36

4.11 GPS interface . . . 37
4.12 Chapter Summary . . . 37

5 Embedded Readout Development **39**
5.1 System architecture . . . 39
5.2 Boot Sequence . . . 42
5.3 SD Memory Card Initialization . . . 44
5.4 Internal Message System . . . 44
5.5 User Interface . . . 47
5.6 Command Interpretation System . . . 51
5.7 Configuring the Firmware . . . 52
5.8 Data readout . . . 56
5.9 Test System for Synthetic Data . . . 59
5.10 Known Problems . . . 61

6 Testing **63**
6.1 Prerequisites for the tests . . . 64
6.2 Synthetic data read-in . . . 66
6.3 Test of Readout . . . 67
6.4 Test of System in Stand-Alone Mode . . . 69
6.5 Summary . . . 70

7 Conclusion and Outlook **71**
7.1 Conclusion . . . 71
7.2 Further work . . . 72

Appendices **75**

A Coding Style **77**

B Git Repository - Directory Tree **79**

C Methodology and Software Overview **81**
C.1 Hierarchical Design . . . 81
C.2 Debugging and Version Control . . . 81

D Operating manual **83**

E Command Sheet & System Message Codes **85**
E.1 Command Sheet . . . 85
E.2 System Message Codes . . . 85

F Tutorials **89**
F.1 FreeRTOS Bootloader generation in Xilinx SDK . . . 89
F.2 Example project with AXI4 Lite peripheral on the Zynq-7000 . . . 89

Acronyms

ADC Analog to Digital Converter.

ALOFT Airborne Lightning Observatory for FEGS & TGFs.

ASIM Atmosphere-Space Interactions Monitor.

BATSE Burst and Transient Source experiment.

BGO Bismuth Germanate Oxide.

BRAM Block RAM.

CGRO Compton Gamma Ray Observatory.

CMOS Complementary Metal-Oxide Semiconductor.

DAU Data Acquisition Unit.

DHCP Dynamic Host Configuration Protocol.

DMA Direct Memory Access.

DPU Data Processing Unit.

DRAM Dynamic RAM.

ECP Cockpit Experiment Control Panel.

EGSE Electric Ground Support Equipment.

FEGS Fly's Eye GLM Simulator.

FEGS BGO Fly's Eye GLM Simulator (FEGS) Bismuth Germanate Oxide (BGO).

FIFO First In First Out.

FPGA Field Programmable Gate Array.

FSBL First Stage Boot Loader.

GOES-R Geostationary Operational Environmental Satellite-R.

GPIO General Purpose I/O.

GUI Graphical User Interface.

IP Internet Protocol.

ISR Interrupt Service Routine.

LEO Low Earth Orbit.

LVDS Low Voltage Differential Signaling.

LwIP Light-weight Internet Protocol.

MIO Multiplexed I/O.

MMIA Modular Multi-spectral Imaging Array.

MUX Multiplexer.

MWC Memory Write Command.

MXGS Modular X- and Gamma-ray Sensor.

NOAA National Oceanic and Atmospheric Administration.

PMT Photomultiplier Tube.

PPS Pulse Per Second.

RAM Random Access Memory.

RCU Read-Out Control Unit.

RDP Remote Desktop Protocol.

RHESSI Reuven Ramaty High Energy Solar Spectroscopic Imager.

RREA Relativistic Runaway Electron Avalanche.

RTOS Realtime Operating System.

SCDP Scientific Data Packet.

SEE Single-Event Effect.

SLC Single Level Cell.

SoC System on a Chip.

SPI Serial Peripheral Interface.

SRAM Static RAM.

TCP Transmission Control Protocol.

TGF Terrestrial Gamma Ray Flash.

TLC Tripple Level Cell.

TLE Transient Luminous Event.

UART Universal Asynchronous Receiver-Transmitter.

UDP User Datagram Protocol.

USB Universal Serial Bus.

Chapter 1

Introduction

1.1 History

Lightning and its loud vibrant sound is something most people are familiar with. Ancients believed the bolt of light coming down from the heavens was a sign of furious and enraged gods, such as Thor from Norse mythology and Zeus from Greek mythology. It was not until the 16th century that the true nature of this phenomenon was revealed. The legend has it that Benjamin Franklin was the one who conducted the experiment resulting in the scientific breakthrough. The experiment consisted of a kite attached to a leyden jar using a silk string, an iron key, and some thin metal wire. The goal was to confirm the hypothesis that lightning was an electric phenomenon, by accumulating charge into the leyden yar.

In 1845 Michael Faraday linked electromagnetic radiation with electromagnetism by showing that polarized light traveling through transparent materials responded to magnetic fields. This discovery was later in the 1860s explained by James Clerk Maxwell through four equations that unified electricity and magnetism as the same force. It explained how a time-varying electric field acts as a magnetic field, *the electromagnetic spectrum.*

Then, in 1886 so called "whistlers" were heard on a 22 km telephone line in Austria. These "whistlers" are *Very Low Frequency* radio signals now associated with lightning discharges. They are audible in audio-frequency range as alternating whisteling from a high to low frequency of about 1000 cycles per second. [1]

This brought on a new type of radio science, and with it, a tool to unlock the mysteries of thunderclouds was born.

Over the time it has been realized that lightning is not the only electric phenomena occurring in the Earths atmosphere. Transient Luminous Events are one type of such phenomena. TLEs happen at a much greater altitude than regular lightning and have the property of being visible to the naked eye just as lightning, although very faint. Due to their relatively short duration, actually spotting them without any equipment can be hard. Figure 1.1 displays a compilation of various photographed TLEs and their relative sizes. Although visually impressive, there is no evidence that TLEs emit any energetic radiation [2].

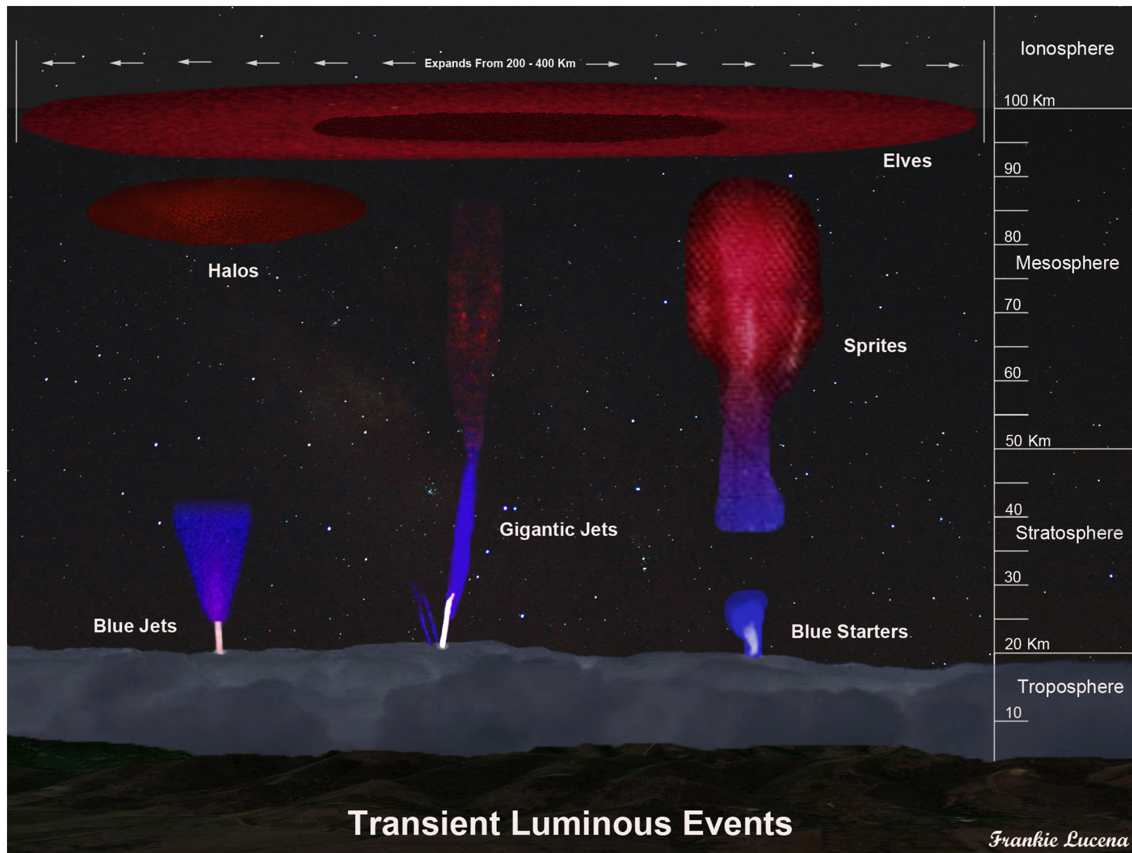

Figure 1.1: A compilation of various photographed TLEs and their relative sizes. *Credits: Frankie Lucena, Spaceweathergallery.com*

1.2 Background and Motivation

Understanding the relationship between the Earth's magnetic field and the atmosphere is the primary goal of the *Birkeland Center for Space Science*. Currently, four different groups of scientists are working on separate research areas to understand this relationship.

The focus of one of the groups is to understand the mechanism behind gamma radiation detected from thunderclouds. By flying aircraft and spacecraft platforms equipped with scientific instruments above thunderclouds, the goal is to understand high-energetic natural occurring phenomena of the atmosphere, such as Transient Luminous Events and *Terrestrial Gamma Ray Flashes*, or *TGFs* for short.

Detecting these phenomena is non-trivial, as they are happening on a sub-microsecond scale. Thus, building detectors with fast readout systems that can sustain the harsh environments found in and at the edge of space, are required to properly resolve the event spectra.

1.3 Objective of this Thesis

The primary scope of this thesis is to initiate the process of developing and designing software for an embedded readout system for the Airborne Lightning Observatory

for FEGS & TGFs (ALOFT). ALOFT is an instrument which will be built by the University of Bergen, and its primary goal is to aid in the study of Terrestrial Gamma Ray Flashes. ALOFT will also be used to study other weaker atmospheric phenomena.

ALOFT will be very similar to the FEGS BGO instrument which flew in the summer of 2017 on the NASA ER-2 High-Altitude Airborne Science Aircraft. FEGS BGO is a scintillator based instrument with a decentralized readout system. ALOFT will also be scintillator based, but implemented on a single embedded System on a Chip (SoC) platform. The SoC will contain a processor, and programmable logic in the form of an FPGA module. This thesis will primarily focus on the software executed on the processor.

The instrument will be flown at an altitude of 20 km, and thus will be operated in an environment with higher radiation levels than the ones found at ground levels. Determining the severity of this environment's impact on the electronics used in the instrument, and making design considerations accordingly, if any, should be done.

This thesis aims to define which parts of the instrument that should be implemented on the programmable system. As many as possible of these functions are also implemented. The instrument must be able to store the data gathered by the detectors to a storage device. The bandwidth of the write operation to the storage device must exceed the expected maximum data rate of 1.98 MB/s. A user interface accessible trough ethernet for instrument control, should be implemented. The status of the system should also be logged to a log file on the storage device, for retrieval post-flight. A test system should be implemented by which synthetic data can be read from a file, and used to simulate data output from the scintillator detectors. Feasibility of implementing the software in an operating system environment contrary to a bare bone environment should also be studied.

When successfully implemented, the end product is a system capable of receiving data from a synthetic data file, process it the same way as the FEGS BGO instrument, and store it on a storage device. Albeit implemented on a single SoC, using less power and with higher performance and reliability.

1.4 Thesis Outline

Chapter 2: High-Energy Atmospheric Phenomena and the Radiation Environment

The first part of this chapter provides the necessary background to give a brief understanding of the physics behind TGFs. This is required to understand the type and usage of instrumentation which is described in the second part of the chapter. The second part also covers some of the campaigns which are currently used in research related to TGFs. In the third part, the radiation environment at the operating altitudes of ALOFT, and its impact on the electronics of ALOFT are investigated.

Chapter 3: FEGS BGO Instrument - Readout System

This chapter provides a summary of the readout system and Electric Ground Support Equipment (EGSE) of the FEGS BGO instrument. The chapter is provided due to the importance of the FEGS BGO instrument to the design of ALOFT, and should give a better understanding of the parts or functionalities migrated to ALOFT.

Chapter 4: ALOFT Design

This chapter provides a comprehensive discussion and documentation for the hardware and software to be used in the ALOFT readout system. The chapter covers the system requirements, elaborates on the choice of hardware and software, reuse of parts from FEGS BGO, and gives an introduction to the operating system being used by the readout system.

Chapter 5: Embedded Readout Development

This chapter documents the software side of the readout system, along with the readout architecture. The chapter aims at giving an in-dept understanding of how the different software modules work, and documents the overall software architecture.

Chapter 6: Testing

This chapter covers the testing of the system, along with the results and a description of setups used when performing the tests.

Chapter 7: Conclusion and Outlook

In this chapter the ALOFT readout system with emphasis on the software side is reviewed. Results from chapter 6 are discussed, and a conclusion is provided. The chapter also includes information on work that is yet to be completed, or should be implemented based on the conclusion.

Appendices

This part of the thesis contains appendices which contains extra details of how the system works. They can prove useful for an operator of ALOFT, or for future developers to review the system.

1.5 Citation Principles

Citation principles for this thesis are that all references that are listed before the sentence punctuation is a reference to the original content of the information provided in that specific sentence only. References after a sentence punctuation are referring to the information provided in all the sentences prior to the reference, up to the last reference, or beginning of the paragraph. For larger sections containing multiple paragraphs that all uses the same reference only, a small text is provided which cites the reference.

Chapter 2

High-Energy Atmospheric Phenomena and the Radiation Environment

2.1 Terrestrial gamma-ray flashes

In 1991, scientists discovered something strange. The Compton Gamma Ray Observatory (CGRO) along with its payload Burst and Transient Source experiment (BATSE), originally intended to measure photons from galactic gamma-flashes picked up signals coming from the Earth [3]. The gamma-ray flashes BATSE observed originated from thunderclouds down on Earth. Although energetic radiation had previously been detected radiating from thunderclouds, this was a brand new phenomenon [4]. The Terrestrial Gamma Ray Flash (TGF) had been discovered. Figure 2.1 gives an artists illustration of how a TGF may appear. Gamma radiation (pink) is being radiated into space, and electrons (yellow) and positrons (green) lines up to follow along the Earths magnetic field lines due to them having electric charge.

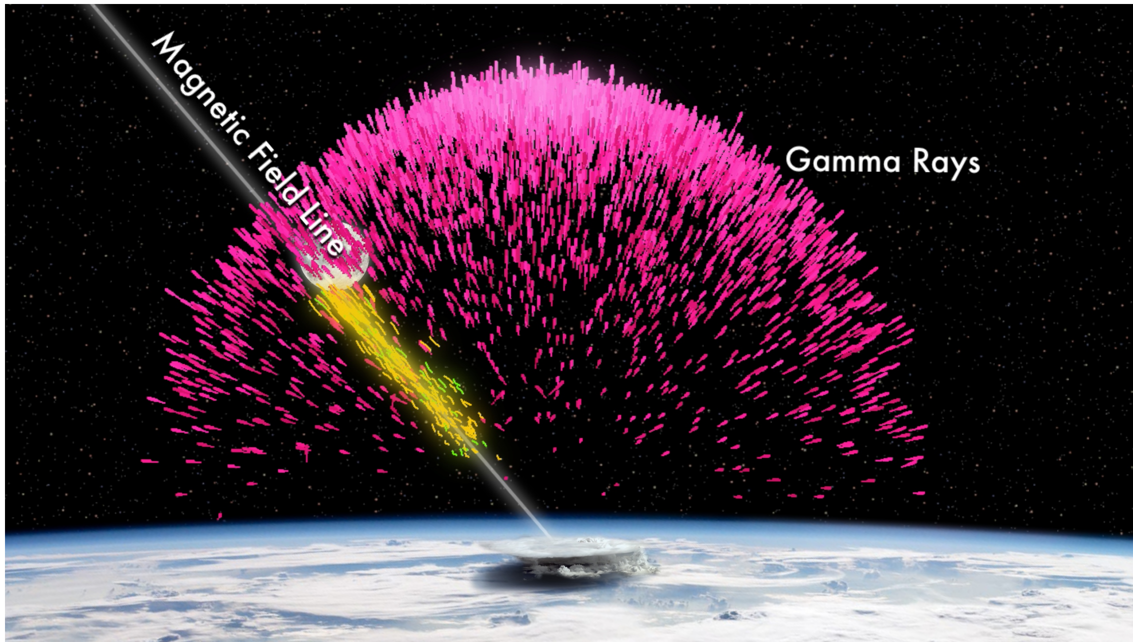

Figure 2.1: An artists illustration of a TGF event. The TGFs originates from regions lower than 20 km and typical duration is in the order of a millisecond [2, chap. 4.1, 4.6]. Gamma-rays in pink, electrons in yellow, and positrons in light green. *Credits: NASA*

2.1.1 Origin of runaway electrons

The following section is based on [2].

The most common mechanism for generating energetic radiation such as x- and gamma-rays in the atmosphere, is through *bremsstrahlung*. Other mechanisms exist as well, such as nuclear decay. In the atmosphere bremsstrahlung occurs between so-called *runaway free electrons* and the atoms present in air. Runaway electrons are electrons with a high amount of kinetic energy. Actually they are so energetic that they behave relativistic. Bremsstrahlung will be further explained in section 2.1.2.

The runaway mechanism was first proposed in 1925 by C.T.R Wilson. He discovered that electrons could become high-energetic when placed in electric fields. If the energy gained from the field exceeded the loss from interacting with other particles in the air, the electron will "run away". Electric fields where the energy gain becomes positive is known as break-even fields, and can be calculated by equation 2.1 where n is the relative air density compared to that of sea level.

$$E_b = 2.18 \cdot 10^5 V/m \cdot n \tag{2.1}$$

For runaway electrons to propagate large distances, a field strength about 30% higher than $\mathrm{E_b}$ is required. This field strength is at comparable level to the maximum field strengths seen in thunderclouds. The graph in figure 2.2 shows energy loss of an electron per unit length as a function of kinetic energy. Energy loss comes from the effective frictional force. eE (horizontal line) shows the force produced by a strong

electric field. For an electron to reach a "runaway state", the loss from frictional forces must be less than the energy gained. This is the case when the kinetic energy of an electron surpasses the energy threshold ϵ_{th}. Such electrons are often referred to as "seed" electrons.

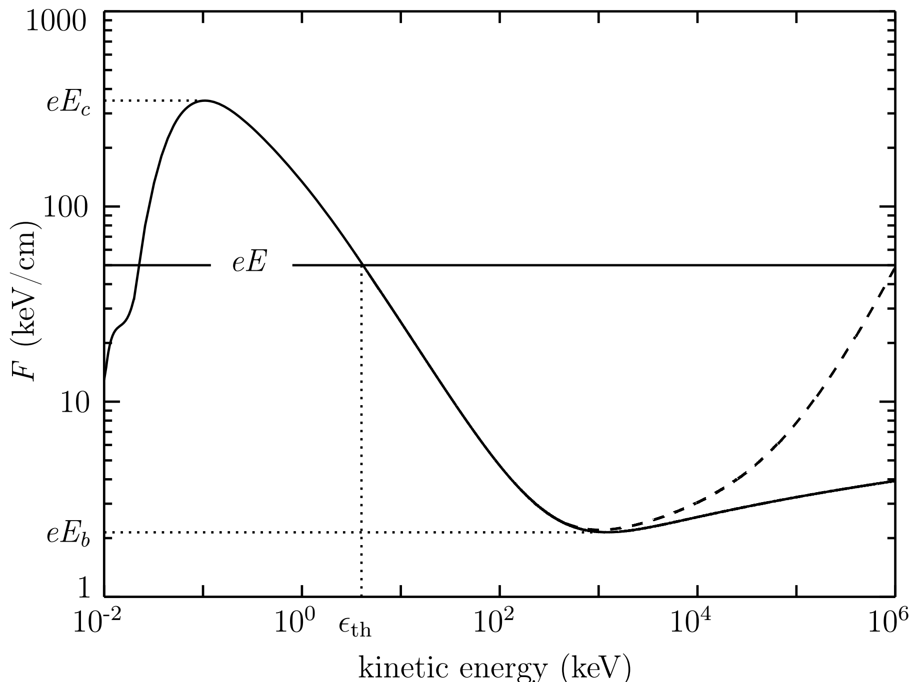

Figure 2.2: The effective frictional force experienced by a free electron (or positron) moving through air at standard temperature and pressure as a function of kinetic energy. [2].

It is suggested that seed electrons are provided by cosmic radiation- or radioactive decay-sources. The seed electrons may cause an electron avalanche known as a Relativistic Runaway Electron Avalanche (RREA). RREAs result in runaway of a high number of electrons. However, if the electric field strength exceeds the maximum frictional force $\mathrm{eE_c}$, all free electrons may runaway without the need of an external seed particle. This is known as "cold" or "thermal" runaway, and is thought to happen during the propagation of lightning.

2.1.2 Bremsstrahlung

One very interesting property of matter can be observed when a fast moving electron such as a runaway electron gets within close proximity of an atomic nucleus. As the nucleus has a positive electric charge and the electron is negative, the coulomb force will make the electron and the nucleus attract each other [5, p. 154]. Comparing the mass of a nucleus with that of an electron, the former is larger by order of magnitudes. When applying Newton's third law (2.2), it becomes evidently that the electron will undergo the greatest deviation from its state of movement when in the electrical field of the nucleus. The path of the electron will start curving towards the nucleus. Velocity is a vector quantity consisting of speed and direction, meaning that the electron experiences a change in velocity although the speed is constant. This radial acceleration produces some captivating properties, as the electron starts

emitting electromagnetic radiation. It should be noted that when the emission happens, the electron speed is decreased to keep the conservation of energy. [6, cap. 34-5].

$$F_{\text{nucleus}} = F_{\text{electron}} \tag{2.2}$$

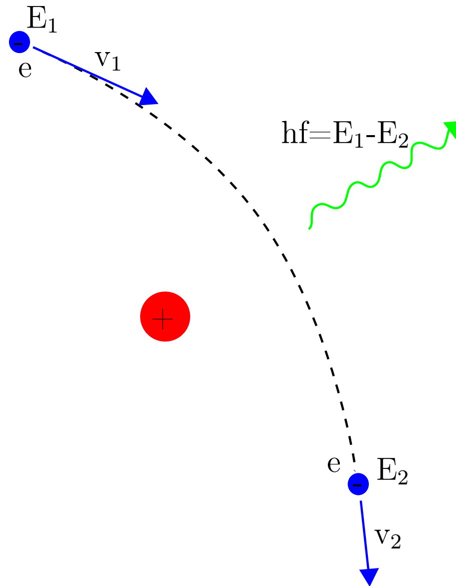

Figure 2.3: Relativistic electron passing by a positive nucleus, resulting in Bremsstrahlung-emission. Illustration by Trex2001 under public domain.

This type of radiation is known as breaking radiation, or by its more common german name *bremsstrahlung*. The amount of energy the photon obtains from the incoming electron depends only on the kinetic energy of the electron, and its distance to the nucleus when passing. The closer the electron gets to the nucleus, the more energy gets released via the photon. As there are almost endless combinations of distances and electron energies, the energy of bremsstrahlung emissions take form of a continuous spectrum as seen in figure 2.4 with low energies being the most abundant. This is caused by electrons having a higher probability to pass a nucleus from a larger distance, thus giving off less energy. [7]

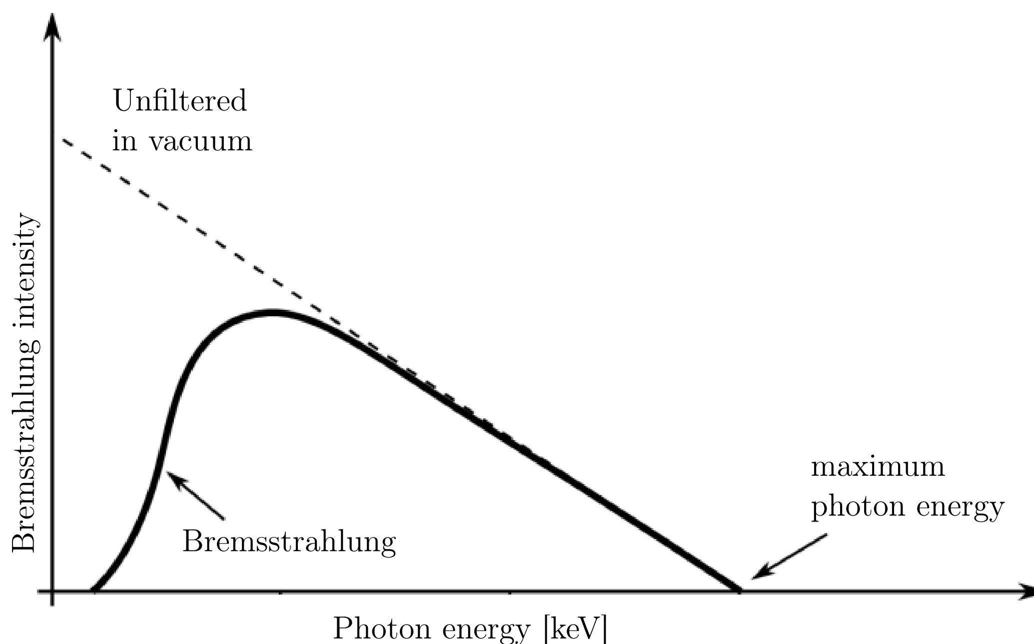

Figure 2.4: Bremsstrahlung intensity as a function of photon energy. The maximum photon energy is limited by the incoming photon's energy loss when passing the nucleus. The sharp decrease in low energy emission is due to absorption in the detector materials. [7]

Usually photons produced by bremsstrahlung are classified as x-rays. The classification comes from it being an energetic photon with an energy exceeding 1 keV and the generating source is an electron. If the source of generation is from nuclear decay, the high energetic photons are called gamma-rays. [2] An alternative classification is to categorize gamma-rays as all photon energies above 1 MeV.

TGFs are the most energetic photon phenomenon with natural origin on Earth, with energies up to tens of MeV [8]. With such high energies, they are classified as gamma-rays, even though the source mechanism fits with the classification of x-rays. The energies does also extend down to tens of keV for less energetic events.

2.2 Instrumentation

This section covers the instrumentation used to detect the photon emissions coming from TGFs. The section source is primarily [9].

2.2.1 Scintillators

Scintillators are a widely used material that combined with read-out and front-end electronics can be used as an instrument in high-energy physics research, dosimetry, medicinal physics, and environmental radiation protection. It takes use of a phenomenon known as luminescence which describes the illumination of a material without the material being heated. In scintillators, luminescence is caused by a particle that hits the scintillator medium. The particle passes through the medium and excites or ionizes some of the atoms in the medium. The excited atoms will emit photons upon deexcitation. The emitted photon energy range depends on the medium used, and the emitted light intensity is proportionally dependent on the energy deposited.

This way, scintillators are useful for converting deposited energy into light, or down-converting photon frequencies to values easier to survey. For this thesis, the gamma- and x-ray photons emitted from TGFs are the incoming particles of interest. By the use of a scintillator with a medium of Bismuth Germanate Oxide (BGO), the large energy deposited by high frequency gamma- and x-ray photons can be converted to light in the near visible spectrum.

2.2.2 Photomultiplier Tubes

Scintillators produces a number of low energy photons for each particle it gets bombarded with, and the goal now is to convert these photons into a sufficient strong electric signal so that it can be further handled. Figure 2.5 shows a schematic of a Photomultiplier Tube (PMT) connected to a scintillator that outputs low energy photons. PMTs converts the photons into electrons by the use of the photoelectric effect. When the photon is converted, multiple electrodes called dynodes accelerates the electrons towards the anode by means of a high voltage. When an electron bumps into the dynodes, it knocks loose more electrons and essentially causes an avalanche of electrons. These "new" electrons are known as secondary emission and increases the current measurable at the anode.

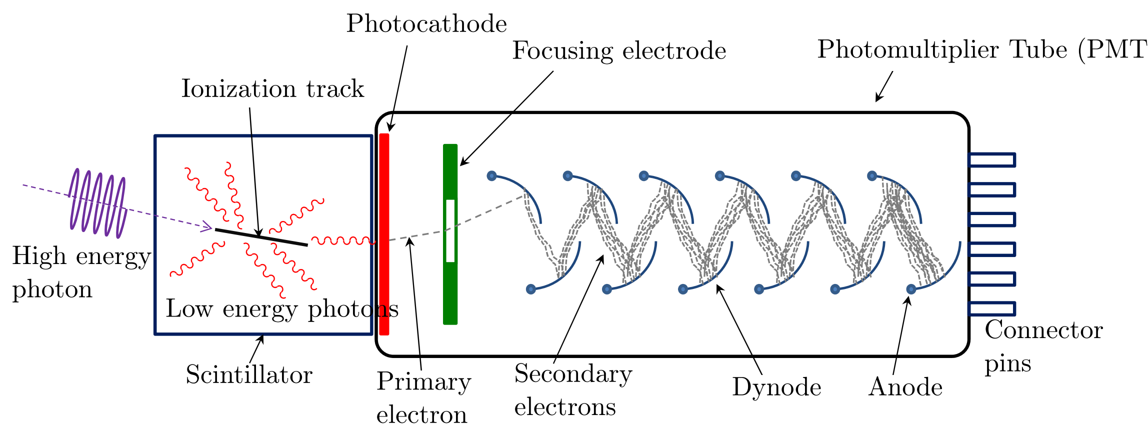

Figure 2.5: Schematic of a photomultiplier tube showing the inner workings and arrangement of components. As has been described, the primary electron causes an avalanche producing a lot of secondary electrons that increases the current at the anode. *Source: Wikipedia, Page name: Photomultiplier*

2.3 Research and Ongoing Campaigns

The Birkeland Center for Space Science located in Bergen is a Norwegian Center of Excellence with the objective of understanding the relationship and interactions between Earth and Space. There are currently two big science campaigns related to TGF detection and understanding, FEGS BGO and ASIM, with FEGS coming to an end. A new campaign, ALOFT, wil continue were FEGS left off.

2.3.1 Atmosphere-Space Interactions Monitor - ASIM

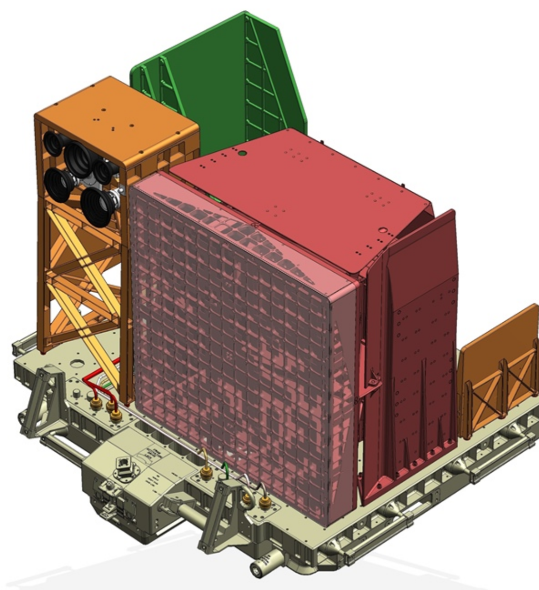

Figure 2.6: Computer aided design model of the Atmosphere-Space Interactions Monitor (ASIM) [10]

The Atmosphere-Space Interactions Monitor (ASIM) contains two instruments, the Modular X- and Gamma-ray Sensor (MXGS) and the Modular Multi-spectral Imaging Array (MMIA) seen in red and orange in figure 2.6. MXGS was developed at the Birkeland Center for Space Science and carries two sets of detectors for different energy ranges. The low energy detector is a 128 by 128 pixelated CZT (CdZnTe)-detector sensitive to the 15 keV to 400 keV spectral band. The pixelation combined with a coded mask in front of this detector makes it possible to determine the TGF source direction. For high energies, BGO detectors sensitive to the 200 keV to 40 MeV spectral band is used. [10]

One of the main questions for Atmosphere-Space Interactions Monitor (ASIM) to answer, is to find out how common TGFs are. BATSE detected 10 TGFs annually, while the Reuven Ramaty High Energy Solar Spectroscopic Imager (RHESSI) detected 100 annually [11]. ASIM was successfully launched and mounted to the Columbus module on the International Space Station in April 2018, with operations ongoing. With preliminary results, it will detect around 800 TGFs annually.

Figure 2.7: ASIM can be seen mounted on the far right of the Columbus module on the International Space Station in this illustration. *Credits: ESA Concurrent Design Facility*

2.3.2 Fly's Eye Geostationary Lightning Mapper Simulator - FEGS-II

The Fly's Eye GLM Simulator (FEGS) instrument was originally built to test and validate lightning sensors to be used on National Oceanic and Atmospheric Administration (NOAA)'s Geostationary Operational Environmental Satellite-R (GOES-R) series of weather satellites. FEGS was mounted to a Lockheed ER-2 High-Altitude Airborne Science Aircraft operated by NASA, which flied over thunderclouds to validate the sensors. ER-2 is the civilian successor of the well famous U2 "dragon lady" aircraft called U-2S. It is capable of operating at mission altitudes exceeding 70,000 feet and thus can be flown above the originating altitudes of TGFs. [12]

The University of Bergen was offered some of the spare room allocated to FEGS to piggyback their own instrument. This has later been known as the FEGS BGO instrument. FEGS BGOs mission is to study TGFs along with other weaker atmospheric phenomena. It is built of three BGO-detectors of the same type as the MXGS instrument. FEGS flew during April and May 2017, with the data obtained from these flights having an ongoing review at the time this thesis was written.

Figure 2.8: The ER-2 aircraft during a NASA campaign. *Credits: NASA*

2.3.3 The Airborne Lightning Observatory for FEGS & TGFs - ALOFT

The Airborne Lightning Observatory for FEGS & TGFs (ALOFT) campaign is the successor of FEGS BGO and will use the same NASA ER-2 High-Altitude Airborne Science Aircraft platform to perform its mission. Even though the ASIM instrument is operational, there is still a lot of need for an aircraft mounted instrument. The main reason for this is that ASIM is in Low Earth Orbit (LEO) and therefore needs to peek through a lot of atmosphere to detect atmospheric phenomena. This dense atmosphere blocks out many of the dimmer phenomena.

With an instrument mounted to an aircraft, the distance between the detector and phenomena, and thus also the amount of atmosphere between them gets decreased. This means that a greater signal to noise ratio can be achieved, and ALOFT is therefore capable of detecting phenomena which would be bellow the noise floor of ASIM.

In this thesis, the name ALOFT will be referring to the BGO detector instrument which is developed in this thesis and flown in the ALOFT campaign. Technically,

the correct name of the instrument would be a separate name in the *ALOFT II* suite, as the instrument suite of FEGS-II as a whole was named ALOFT. However, as no name has been given to the new instrument, and to avoid further confusion to anyone involved in the project, the single new BGO detector instrument will be referred to as *ALOFT* only.

2.4 The Radiation Environment

As will be seen in section 2.5, radiation can cause many undesirable behaviours in electronics. There are also very important considerations to make regarding the use of humans in radiation environments, for instance astronauts. Therefore, applications that are going to operate in radiation environments must be designed to tolerate the radiation exposure with satisfying results.

For high-altitude atmospheric and LEO space applications, there are many design similarities and considerations that can be made, as the environmental conditions regarding radiation are quite similar. These similar environmental conditions comes from both application taking place partially or totally outside of the atmosphere, which at lower altitudes would acts as a shield against high-energetic particles. As both applications are also within Earth's magnetic field, they are still shielded to some degree from high-energetic cosmic radiation.

The substantial source of radiation in an airborne or space-based instrument is cosmic radiation. Cosmic radiation is a collection of particles originating from the universe, with the Sun being the major contributor to the amount received on the Earth. The amount originating from the Sun can be seen heavily increased during a *Solar Particle Event*, such as a *flare* or *Coronal Mass Ejection.* Outside of the atmosphere, the cosmic radiation consists of 85% protons, 13% α-particles, and 2% heavy ions. The received dose from cosmic radiation at 1600 and 5000 meters above sea level is twice and seven times the dose received at sea level, respectively. [13]

Some of these particles can cause problems by their own, but they can also interact with atoms in the atmosphere, causing secondary emission such as gamma-rays through bremsstrahlung, high-energetic neutrons, protons, and pions. Secondary emissions increases the total number of radiation particles as they go through the atmosphere, but at lower altitudes the atmospheric density is high enough to absorb most of the particles. This can be seen in figure 2.9 which shows the relative neutron flux as a function of altitude, obtained from a balloon flight experiment [14].

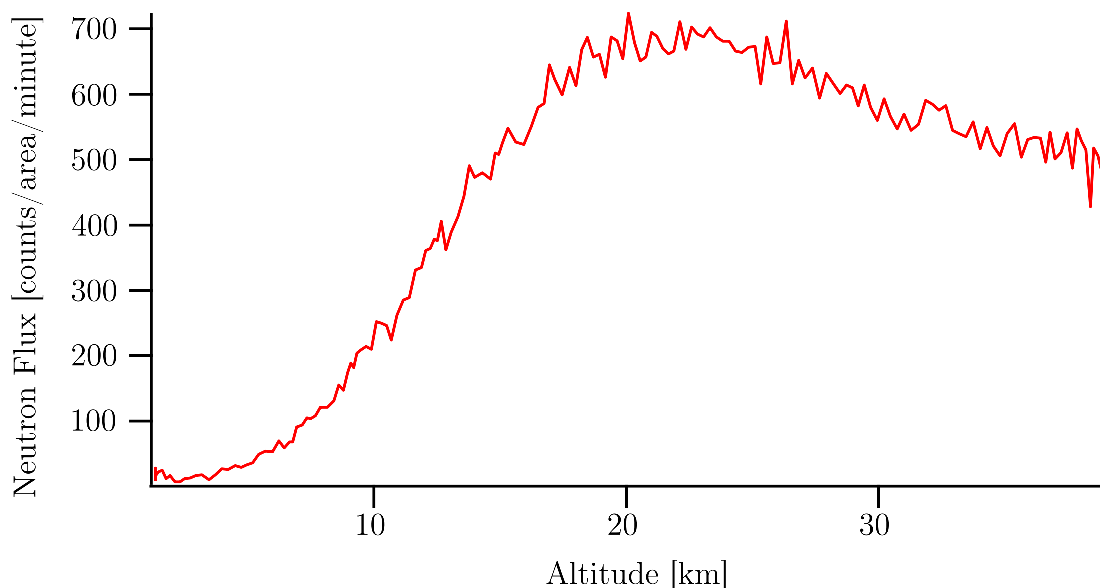

Figure 2.9: Neutron flux as a function of altitude, measured from a balloon flight experiment. The effects of secondary emissions and atmospheric absorption can clearly be seen. [14]

Neutrons have the highest flux of the secondary emission particles. Figure 2.10 shows the neutron energy spectrum measured and calculated at 40,000 Feet above ground [15]. At 60,000 Feet which is similar to the flight altitude of the ER-2 Aircraft, the neutron flux has been measured to be around 1.3 neutrons/cm^2/second [16].

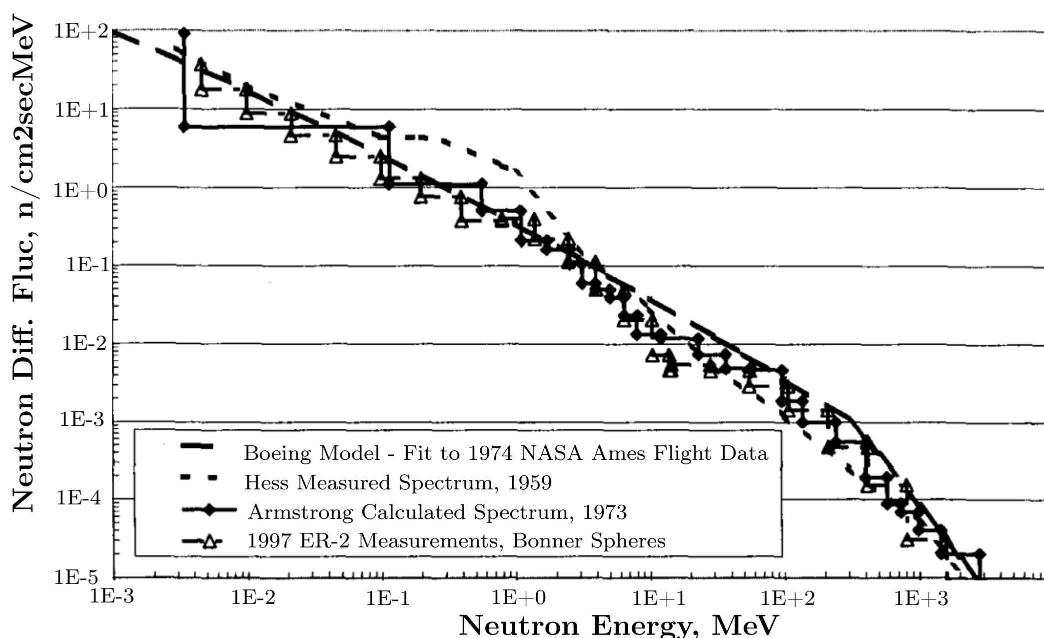

Figure 2.10: Neutron energy spectrum in the atmosphere at 40000 feet above sea level. [15]

As charged particles travels along the magnetic field lines of Earth's magnetic field,

radiation flux levels are higher at larger latitudes. Measurements at jet aviation altitudes during a solar minimum have shown a 2.5 to 5 times increase in radiation flux, in the polar regions compared with flights at equatorial latitudes. [17]

2.5 Radiation Design Considerations for ALOFT

When designing an instrument to be operated at high altitudes or in space, proper consideration to the radiation environment must be taken to avoid the undesirable effects of radiation. During the 2017 FEGS-campaign the ER-2 aircraft was operated at 20 km altitude. The flight trajectory was between Georgia and Colorado in the Continental United States [18]. For this trajectory, radiation is evidently lower than a trajectory were the aircraft is flown at higher latitudes. However, as seen in section 2.4, any equipment at these altitudes will be exposed to much higher radiation intensity than at sea level. In addition to the radiation with cosmic origin, the ER-2 will also be exposed to the radiation originating from the TGFs. As was described in section 2.1.2, TGFs can have photon energies of tens of MeV. They do, however, have such a low occurrence that it it believed that their impact on the electronics can be neglected.

The effects of radiation can be divided into two categories; Long-term effects, and Single-Event Effect (SEE). As the name indicates, long-term effects happens over a longer time period than SEEs, and encompasses effects caused by the *Total Ionizing Dose* accumulated, and *displacement damages*. Displacement damages are not particularly applicable to Complementary Metal-Oxide Semiconductor (CMOS) microelectronics, and due to the relatively short mission periods of ALOFT, the long term effects are in general not applicable to the electronics discussed in this thesis and will therefore not be discussed further. [19]

2.5.1 Single Event Effect (SEE)

SEEs are caused by the interaction with single charged particles and can cause great problems in electronics. Their effects can be divided into soft and hard errors. Soft errors are recoverable although they may have corrupted the data content in registers and other memory devices. Hard errors however, induces permanent damage to the device. [19]

Some hard error effects are *Latchup*, *Burnout*, and *Gate Rupture*. Latchups causes permanent loss of functionality due to low-resistance paths forming between V_{DD} and GND in CMOS chips. Limiting the occurrence of latchups can only be performed through changes to the CMOS design, and is therefore outside of the scope of this thesis. Burnouts and Gate rupture effects on sub-micron metal-oxide-semiconductor transistors are rare and minimal, but can contribute to a reduction in lifetime expectancy which is not important to ALOFT. [20]

Single-Event Upsets and Single-Event Transients

With the introduction of Dynamic RAM (DRAM) that used semiconductor technology in the 70's, vendors were perplexed with the discovery of bits on the devices randomly being flipped. Intel later identified the events being caused by α-particles

originating from trace amounts of uranium and thorium in the microelectronics package colliding with the silicon. When an α-particle strikes silicon, it generates an electron-hole as the particle loses energy. These electron-holes act as carriers, and if collected into the diffusion terminals of transistors, charge can be collected. This increase in charge causes a current spike which is known as a *Single-Event Transient*. If the charge is of comparable size to the node charge of the transistor, the bit value stored on the transistor can be flipped. If the bit is flipped, the event is categorised as a *Single-Event Upset*. Both single-event transients and single-event upsets are classified as "soft errors". [21]

Single-event upsets caused by α-particles have the approximate same occurrence on the ground as in aircraft, as the particles originate from the package. The problem can be reduced by using high-quality materials for the package production. However, single-event upsets caused by high-energetic neutrons with energies larger than 10 MeV are 300 times more probable in the atmosphere than on the ground. When compared, the contribution of single-event upsets caused by α-particles in atmospheric applications are insignificant compared to the ones caused by high-energetic neutrons. Incidents caused by α-particles are also quite rare, and therefore in general can be neglected. [15]

2.5.2 Calculating the risk of SEEs

SEEs poses a risk to ALOFT, as they may alter register and storage values. Equation 2.3 gives the number of SEEs per hour, were N_{bits} is the number of configuration bits on the device, σ_{bit} is the bit cross section given in cm^2/bit, and $\phi_{neutron}$ is the neutron flux per cm^2.

$$\frac{SEE}{hour} = N_{\text{bits}} \cdot \sigma_{\text{bit}} \cdot \phi_{\text{neutron}} \cdot 3600 \tag{2.3}$$

ALOFT will be implemented on a 7 series FPGA from Xilinx. This is further described in chapter 4. The bit cross section of this FPGA series is $6.99 \cdot 10^{-15} cm^2/bit$ [22]. In section 2.4, the neutron flux at 60,000 feet was found to be around 1.3 neutrons/cm^2/second. The bitfile used for the FPGA has a size of 16.8 million bits. Only a smaller fraction of these bits are actually used in the design, but to make a conservative estimate all are assumed critical to the system.

For an 8 hour flight, this makes the probability of one single SEE to be around 4‰. Conservative design rules do, however, suggest that design considerations should be taken on the assumption of a case ten times worse. This means that the design should be implemented with the assumption on a 4% likelihood of an SEE occurring during a flight.

These numbers are very low, especially as the ALOFT instrument is non-critical to the safety of the pilot of the ER-2, or any equipment for that matter. As will be seen later in the thesis, some radiation design precautions have however, been implemented in the software.

Chapter 3

FEGS BGO Instrument - Readout System

The development of the ALOFT readout system is the main scope of this thesis, but as discussed in section 2.3, the design and architecture of ALOFT will be heavily inspired by the FEGS BGO instrument due to their mission similarities. This chapter will explain the readout system of the FEGS BGO instrument. Figures and the majority of the information is based on [23].

3.1 System Architecture

Figure 3.1 illustrates the readout and system architecture of FEGS BGO. There are three major blocks in addition to the external Electric Ground Support Equipment (EGSE). The major blocks are dotted and colour coded in light grey to represent abstract components, namely the Data Acquisition Unit (DAU), the Read-Out Control Unit (RCU), and the Data Processing Unit (DPU). Each will be further explained in their respective subsections. In each of the major blocks, dark grey blocks illustrates hardware with either firmware or software if present.

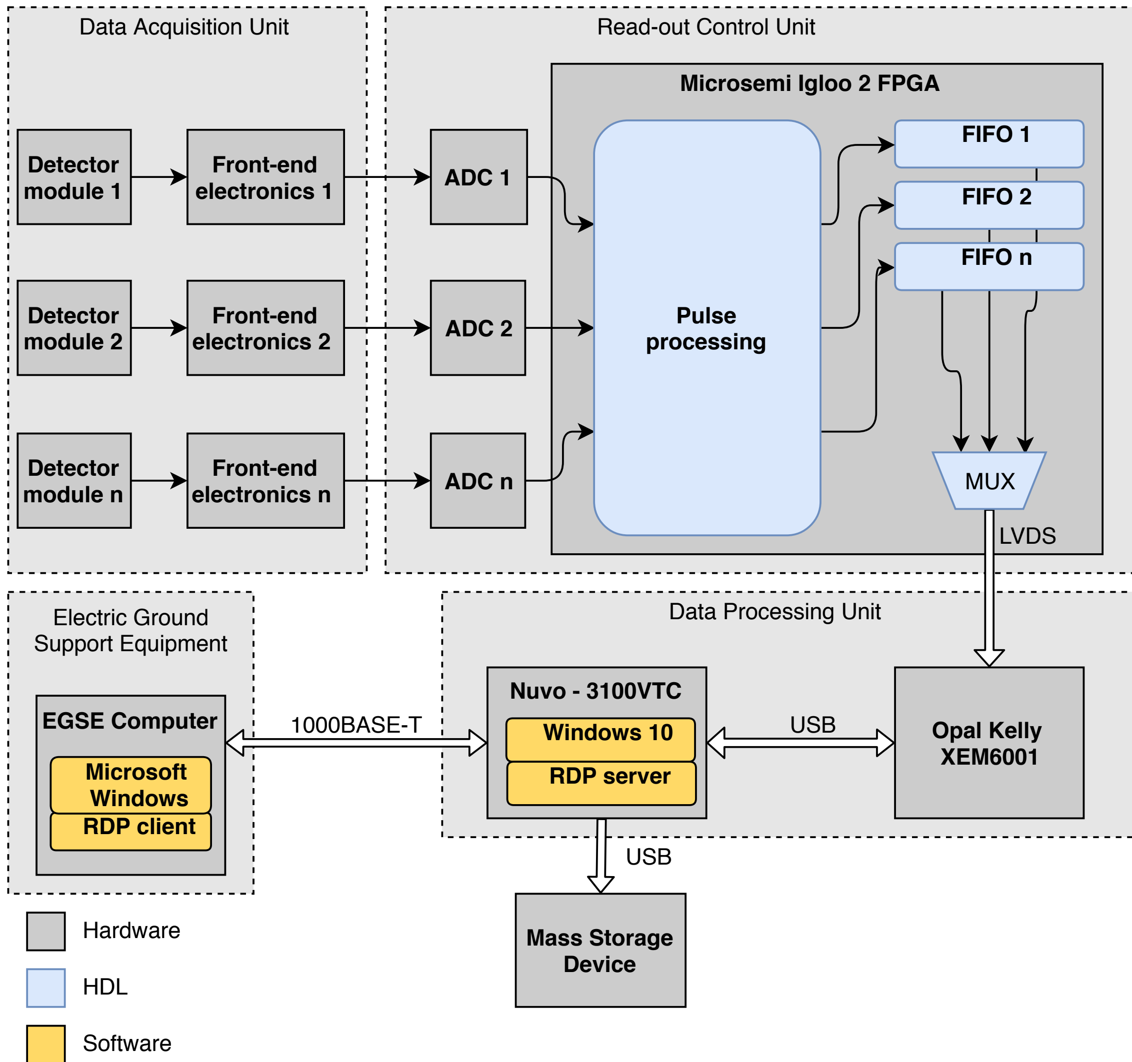

Figure 3.1: Block diagram of the FEGS BGO instrument readout and system architecture. The three major blocks in light grey encompasses the Data Acquisition Unit (DAU), Read-Out Control Unit (RCU), and Data Processing Unit (DPU), in addition to the Electric Ground Support Equipment (EGSE).

3.2 BGO Data Acquisition Unit

The DAU is a mechanical construction of three BGO *detector modules*, one fast *detector module*, and voltage dividers. Each BGO detector module is a separate BGO scintillator crystal with a PMT.

The output from the PMTs are low-pass filtered with a corner frequency of around 20 MHz. This is to remove any high-frequency components before each PMT-signal is amplified by a preamplifier. The relative position in the readout-chain of the *preamplifier*, *low-pass filter* and *detector module* can be seen in figure 3.2. The figure also contains the Analog to Digital Converter (ADC) and ADC driver modules which are described in section 3.3.

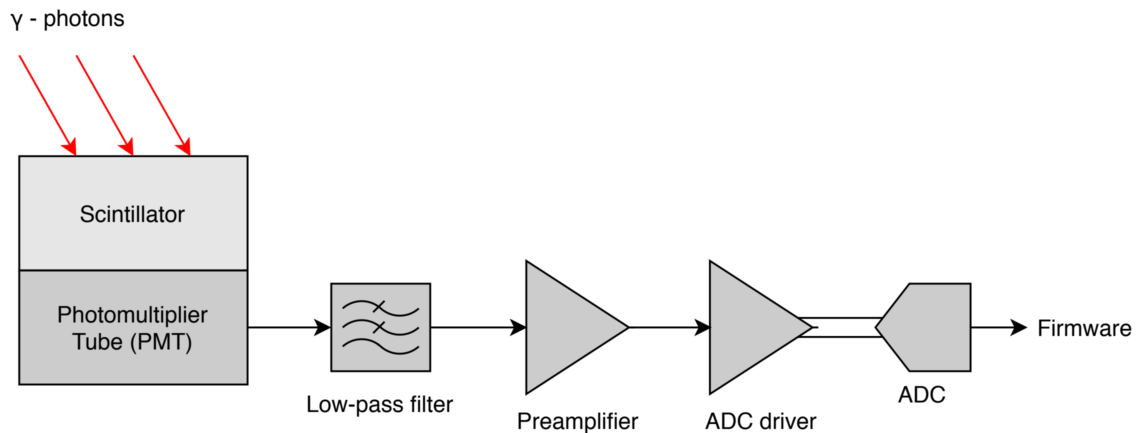

Figure 3.2: Relative position of the detector modules, front-end electronics, and ADCs in the FEGS BGO instrument.

3.3 Readout Control Unit

The RCU contains three ADCs and a *Microsemi Igloo 2* Field Programmable Gate Array (FPGA).

As the PMTs outputs analogue signals, it is necessary to convert them into digital signals to efficiently process and handle them. This is done by the three ADCs, one for each PMT channel. Each ADC runs in a differential signaling setup. By using differential signaling, common mode noise and interference gets rejected. The overall performance is also increased as the signaling is balanced, and the dynamic range gets increased by a factor of two.

To create the differential signaling from the single ended analog PMT output, an ADC driver is placed in front of the ADC in the readout-chain. This is illustrated in figure 3.2. The ADCs used are the 12 bit RHF1201, operating at a 36 Msps sample rate which is the same frequency as the main oscillator in the system.

The RCU is also connected to a GPS receiver called Copernicus II from the company Trimble. The receiver provides a 1 Hz Pulse Per Second (PPS)-signal accurate within 60 ns rms. [24]

3.3.1 Firmware

Each ADC outputs a 12-bit data stream which is fed into the firmware running on the flash based Igloo 2 FPGA. Figure 3.3 provides a block diagram of the various pulse processing performed in the firmware. Each PMT channel is handled concurrently in the firmware.

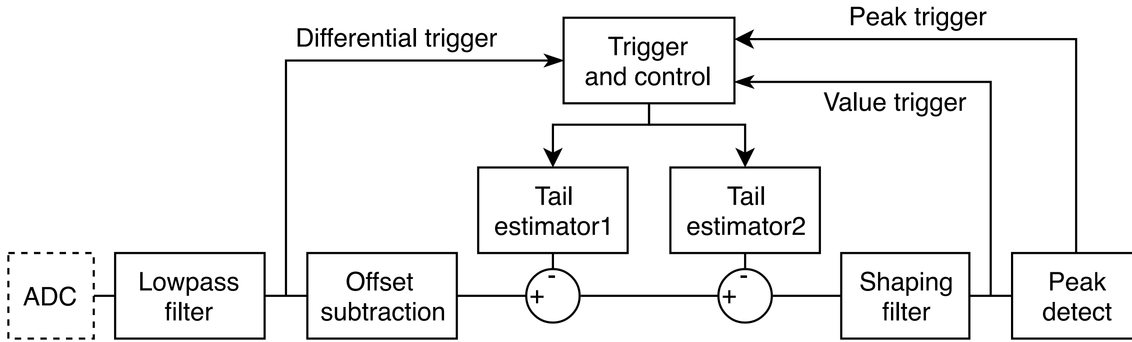

Figure 3.3: Block diagram of the pulse processing performed in the FEGS BGO firmware, implemented on the Igloo 2 FPGA. Solid blocks are implemented on the FPGA.

Interfacing the firmware is performed through a set of registers. Each module in the firmware contains a control register, and some also have a status register. The control registers are 8 bit with 14 bit addresses.

Scientific Data Packets

Data from each of the firmware data channels are outputted in the form of a 48 bit data packet called a Scientific Data Packet (SCDP). Table 3.1 displays the SCDP normally outputted from the firmware.

2 bits	**4 bits**	**1 bit**	**1 bit**	**1 bit**	**1 bit**	**6 bits**	**12 bits**	**20 bits**
Flag bits	Address	OVF	Fast	Valley	Spare	Fast time tag	Energy	Time tag
'00'	00 AA	'0'	'0'	'0'	'0'	FF FFFF	EE..EE	TT..TT
47 46	45 42	41	40	39	38	37 32	31 20	19 0

Table 3.1: Normal SCDP.

The normal SCDP is transmitted for any gamma ray event were a sufficient delay exists between two events. The first two bits are set depending on the *OVF* and *Fast* flags. The address bits indicate which BGO detector module the data originates from. Each SCDP contains a *fast time tag* and a *course time tag*. These are set using the PPS-signal generated by the GPS receiver. The *fast time tag* has a resolution of 27.78 ns, and the course time tag has a 1 µs resolution. The *energy*-field is the detected energy with a value range from 0 to 4095.

When the *OVF* and *Fast* flags are set, the generated SCDPs are called *Fast SCDP*, *Overflow SCDP*, and *ADC Sample SCDP*. Table 3.2 illustrates the different SCDPs based on the flags.

OVF	**Fast**	**Type**
0	0	Normal
0	1	Fast
1	0	Overflow
1	1	ADC Sample

Table 3.2: SCDP types.

Fast SCDP only differs from the normal SCDP by the *Fast* flag being set. *Fast SCDPs* are created when the time between two event-pulses are too close to be separated. The *Overflow SCDP* is transmitted when the ADC overflows for more than one clock cycle. The fast time tag is then unused, and the energy-field is replaced by the number of clock cycles the overflow exists.

2 bits	**4 bits**	**1 bit**	**1 bit**	**1 bit**	**1 bit**	**6 bits**	**12 bits**	**20 bits**
Flag bits	Address	OVF	Fast	Valley	Spare	Unused	Overflow duration	Time tag

Table 3.3: Overflow SCDP.

The *ADC Sample SCDP* is transmitted when the RCU is commanded to operate in *continous* or *triggered* sample mode. In these modes, the raw unfiltered 12 bit ADC value is included in the SCDP. Table 3.4 displays the ADC Sample SCDP.

2 bits	**4 bits**	**1 bit**	**1 bit**	**1 bit**	**1 bit**	**6 bits**	**12 bits**	**20 bits**
Flag bits	Address	OVF	Fast	Sample number			Energy	Time tag
'00'	00 AA	'1'	'1'	N	N	NN NN	EE..EE	TT..TT
47 46	45 42	41	40	39	38	37 32	31 20	19 0

Table 3.4: ADC Sample SCDP.

SCDPs for each respective channel are placed in separate First In First Out (FIFO) memories. A Multiplexer (MUX) operated with round-robin scheduling is then used to mux through the different FIFOs and makes the data available on a Low Voltage Differential Signaling (LVDS) serial data line. The round-robin scheduling assigns the total available time equally to each separate channel.

3.4 Data Processing Unit

The DPU is connected to the RCU using the LVDS link. The DPU consists of the OpalKelly XEM6001 FPGA, and the Nuvo-3100VTC in-vehicle computer.

3.4.1 OpalKelly XEM6001

Communication to and from the Nuvo computer is handled over Universal Serial Bus (USB), so the OpalKelly XEM6001 is used as an interfacing adapter between the RCU LVDS and USB interface. The LVDS link provides a maximum data transfer rate of 330 k packets per second. Saturation is only possible when the instrument is running in the continuous sample mode, and equals to a data rate of 1.98 MB/s for 48 bit SCDPs.

XEM6001 features the Xilinx Spartan-6 FPGA, which is well equipped to handle the LVDS interfacing work. [25].

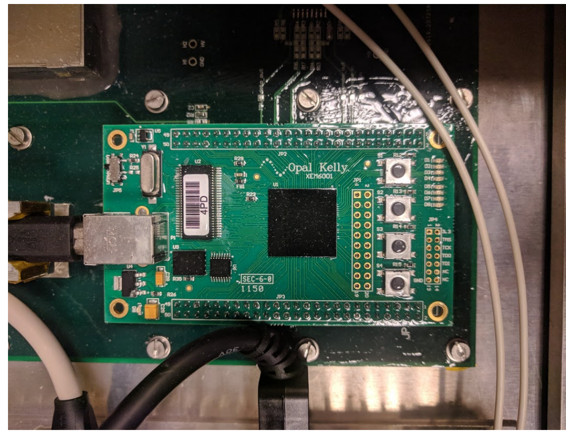

Figure 3.4: Opal Kelly XEM6001 mounted in the FEGS BGO instrument.

3.4.2 Nuvo-3100VTC

The Nuvo-3100VTC is a fan-less in-vehicle computer that features small dimensions and the required specifications to operate as DPU in FEGS BGO. With the used hardware configuration it has been budgeted to use up to 47 W. The Nuvo is used as DPU by writing the SCDPs received from the RCU onto a storage medium. The Nuvo also handles firmware configuration through the register interface, and system monitoring. This is performed by a C# software program called EGSE-software. The Graphical User Interface (GUI) of the program can be seen in figure 3.5.

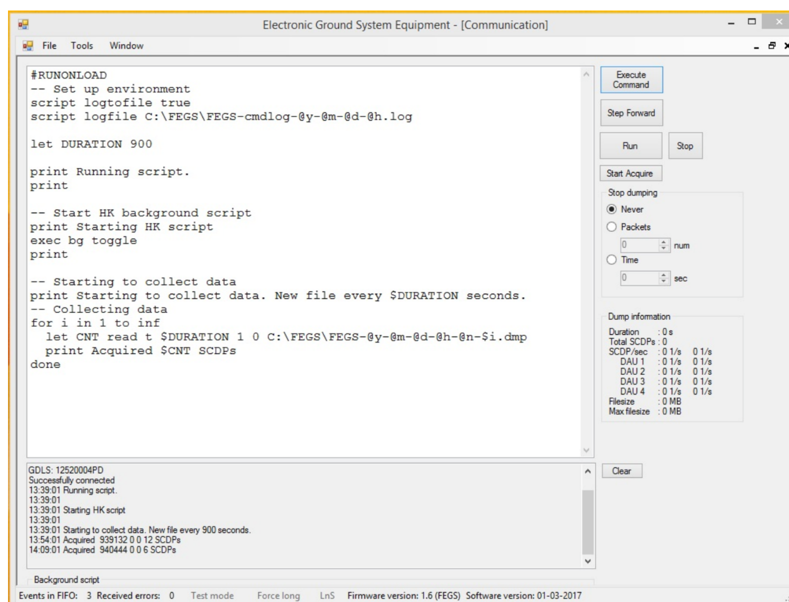

Figure 3.5: GUI of the FEGS BGO instrument Electric Ground Support Equipment (EGSE).

The operating system on the Nuvo is a Realtime Operating System (RTOS)-version of Microsoft Windows 10 which is configured to accept Remote Desktop Protocol (RDP)-connections necessary for remote access by the EGSE.

The Nuvo also features watchdog functionality. Watchdog is a hardware mechanism

which works by software resetting a timer before it reaches a predefined value. If the software hangs or otherwise fails to reset the timer, the hardware will automatically restart. This ensures that the Nuvo restart if the software behaves abnormally.

For storage of the acquired data, an industrial grade Single Level Cell (SLC) USB flash drive is used. The flash drive is connected to the Nuvo using USB version 2.0.

3.5 Operation of FEGS BGO during Flight Campaigns

To test and configure the FEGS BGO instrument, a cable harness connected to the Nuvo can be pulled through the payload bay so that it is accessible at the aft. External equipment connected to the instruments and airplane itself are normally termed EGSE, but in this thesis the term will only reference systems connected to the FEGS BGO and later the ALOFT instrument. The harness consists of an HDMI video out cable to connect a monitor to the Nuvo, a USB-B extension cable for easier access connecting and removing the USB flash drive, and a category 5E ethernet cable. By connecting an external computer to the ethernet cable, an ad-hoc network using static Internet Protocol (IP)-addressing can be created.

When connected, the EGSE-computer can connect to the Nuvo using RDP. RDP does a pass-through of the display and audio of the Nuvo to the EGSE, and sends mouse- and keyboard-input from the EGSE to the Nuvo. The RDP-connection is primarily used to control the EGSE-software running locally on the Nuvo.

3.6 Review

Development of the FEGS BGO instrument was designed with reuse of software and hardware from the ASIM campaign in mind. Reuse is great for shortening the development time of an instrument, but if not done properly can make for a less elegant solution. This is what happened to FEGS BGO.

As the new ALOFT campaign is built on the FEGS campaign, this is an excellent opportunity to revise the design of the FEGS BGO instrument for it to better accommodate the campaign requirements. In short, revising should remove the use of proprietary software, and move the design onto a more efficient hardware solution.

Chapter 4

ALOFT Design

In section 1.3 it was introduced that the Airborne Lightning Observatory for FEGS & TGFs (ALOFT) will be an embedded version of the FEGS BGO instrument for which the readout system was described in chapter 3.

FEGS BGO was presented as a decentralized system, with functionality spread over multiple separate devices. ALOFT will use the same detector modules, but the firmware and DPU will be implemented on a System on a Chip (SoC) embedded platform.

Due to the expected mission similarity and reuse of firmware, detector modules, and DPU functionality, the proposed design is based on the assumption of ALOFT having overall similar requirements as those valid for FEGS BGO.

4.1 System On a Chip

Due to the constantly development of ever more complex systems, the use of SoC-designs has become increasingly common. SoC is the term of systems combining different hardware modules through a common bus, on a single chip. Example of such modules could be memory units, I/O-peripherals, or a microprocessor. [21]

As the hardware modules are placed closer together than in conventional systems, delays are much smaller, and higher bandwidth can be obtained. This reduces some of the problems associated with building high-speed systems. Typically the modules can be interchanged as they have been individually pre-designed, adding a high degree of customizability-options for the system designer. SoC-designs with an FPGA-block also increases customizability as additional blocks can be implemented on the FPGA logic after the actual chip has been produced. From a designers perspective, using a SoC-design increases the time that can be used in the *behavioral* and *structural* design domains. These domains concern how the system behaves and which parts that needs to be added to fulfill the desired behavior. Consequently, less time is needed in the physical domain where tasks such as floorplanning and actual chip geometry-design takes place. See for example [21].

Naturally, development time is reduced as sophisticated systems can be designed without going through the hassle of coming up with every single subsystem design. SoC designs also aids in making the final product hierarchical, and decreases

development time.

The SoC used for ALOFT is a dual-core microprocessor system embedded on an FPGA. The microprocessor system in such systems are referenced as the *programmable system*, and the FPGA as the *programmable logic*.

ALOFT uses the firmware from the FEGS BGO RCU, and much of the same functionality as was found on the FEGS BGO DPU. As the firmware is already written in the hardware description language VHDL, and thus intended to run on an FPGA, it will be ported to the ALOFT programmable logic. This is currently being done at the University of Bergen, and the results should be available in [26] when published. Functionality earlier handled by the DPU are easiest implemented using software, and thus wil be executed on the programmable system.

With most of the readout system implemented as VHDL and software on the embedded SoC platform, it can easily be customized to meet additional mission and design requirements later in the development. The remaining part of this thesis will primarily focus on the design and implementation of the software responsible for the DPU-like functionality.

4.2 System Requirements

The required functionality of the software can be divided into four tasks. The primary task is data storage. This means to receive SCDPs from the firmware and store them safely to a mass storage device. It is important to perform this task with sufficient speed to avoid introduction of any bottlenecks, which would result in data loss.

The second task is firmware configuration. As has been elaborated in section 3.3.1, the firmware is interfaced through a set of configuration and status registers. ALOFT's software must be able to configure the firmware through this interface, preferably from a configuration table easy accessible when operating the instrument.

The third task is to log the status and health of the system. As the software is in control of both the firmware and the rest of the readout system, it has a very good position for monitoring the overall status of ALOFT. The log must be written to a nonvolatile memory to ensure its survival, should the instrument be power cycled.

The fourth and final task is to provide a user interface to the instrument. The interface must report information about ALOFT, and provide a way of manually controlling the instrument. The interface should be easily accessible even when the instrument is mounted in the ER-2 Aircraft. Preferably, the user interface should also be available from ground services while in-flight through the ER-2 Instrument Network described in section 4.9.

These four tasks must all be started automatically when power is connected.

4.2.1 Reliability

ALOFT will have fairly limited human interaction capabilities while in-flight. The only interaction available will be through the user interface covered in section 4.9

and the pilot operated Cockpit Experiment Control Panel (ECP) which is covered in section 4.10. In essence, this means that the instrument needs to be highly reliable. Reliability is defined as the probability of a system or component to function as intended for a specified period of time, and generally it is as relevant for software as it is hardware.

Hardware reliability in ALOFT is mostly relevant to the burn in phase, as the general life time of electronics surpasses the use of the instrument. Extensive hardware testing must be performed to ensure the removal of weak components.

The failure rate of software is at its peak during the *test and debugging* phase. When this phase is completed and the software is deployed, the probability of failures are usually at a minimum. The failure rate is constant beyond this point if no changes are made. Upgrades can add additional complexity which will increase the overall failure rate. In addition, as complexity increases, the maintainability decreases, which again increases the probability of failure when the software is upgraded. [27]

Focus has been on keeping complexity low to increase maintainability. This has to a large degree been achieved through regularity and the use of a hierarchical design strategy. Regularity and hierarchical design are further explained in appendix C.1.

4.2.2 Hardware requirements

Data storage

The data storage requirements are based on two factors; required data rate, and total amount of data. As seen in section 3.4, the FEGS BGO instrument had a maximum data rate of 1.98 MB/s. This maximum data rate was, however, only reached when the instrument operated in the continuos sample mode. As the limiting factor in FEGS BGO has been removed in ALOFT the transfer rate is no longer limited to the same rates. Higher data rates are not required for ALOFT, but it should be emphasized that capabilities beyond the required data rate will ensure a lower possibility of the software to induce a bottleneck in the readout chain.

ER-2's maximum mission duration is 8 hours, providing about 7 hours of data collection at the desired altitude. Normal missions last around 6.5 hours giving 5.5 hour of data collection. Theoretically, the instrument could generate around 50 GB of data if operated in the continuos sampling mode for a max duration flight. 50 GB is, however, an unrealistic overestimation. Results from the FEGS campaign indicates that even very conservative estimates does not surpass 7 GB of data. In addition to the collected data, some space must be reserved for the system logging, configuration table, and as will be introduced in section 4.7, a bootloader. These files are negligible in terms of size compared to the collected data.

4.2.3 Software requirements

It has been decided that the software will be written in the programming language *C*. C has the quite unique capability of providing syntax for both high- and low-level programming. It is also architecture-independent and thus suited as a general purpose language which is needed to perform the four main functionalities assigned

to the ALOFT software. C++ would be another option, but is not truly supported by the FreeRTOS operating system introduced in section 4.4 without making changes to the kernel.

4.3 Xilinx Digilent Zybo SoC Trainer Board

A Xilinx Digilent Zybo SoC Trainer Board has been chosen to be used during development. The board contains the Z-7010 SoC discussed in section 4.3.1. It also features peripherals such as gigabit ethernet, SD memory card capability, USB with On The Go support, HDMI, Audio in/out, Multiplexed I/O (MIO)- and General Purpose I/O (GPIO)-ports.

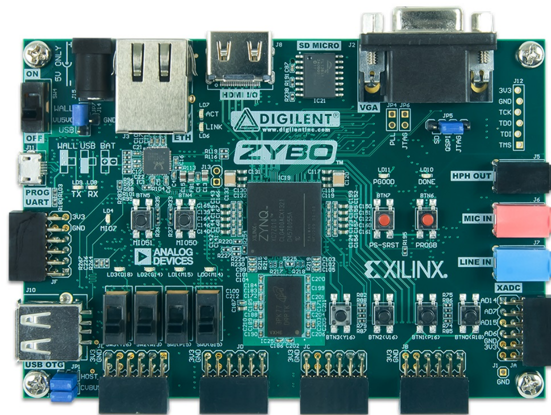

Figure 4.1: Xilinx Digilent Zybo board [28]

Nonvolatile memory

The Zynq-7000 contains two Serial Peripheral Interface (SPI) bus controllers. These can be used to interact with SD memory cards with serial mode support or other hardware such as various sensors, real-time clocks, etc. [29]

On the Zybo development board, a micro SD connector can be found on the reverse side of the board. Enabling the SPI controller in Xilinx' software program *Vivado* enables use of a connected micro SD memory card both as a mass storage device by any application running on the board, and as a boot medium. To use the micro SD memory card as a boot medium, the Mode jumper (JP5) must be positioned to select "MicroSD", and the card must contain a bootloader on a FAT 16/32 file system. [30]

Bootloaders are further covered in section 4.7.

4.3.1 Zynq-7000 Architecture

For development, the smallest member of the Xilinx Zynq-7000 All Programmable SoCs-family, the Z-7010 has been used. The Z-7010 features a dual-core ARM® Cortex™-A9 MPCore™ based programmable system, and a high performance, low-power Xilinx programmable logic, built with 28 nm process technology. A block diagram of the Z-7010 architecture is seen in figure 4.2, with the *programmable*

system embodied in light grey, and the *programmable logic* in dark grey. [29]

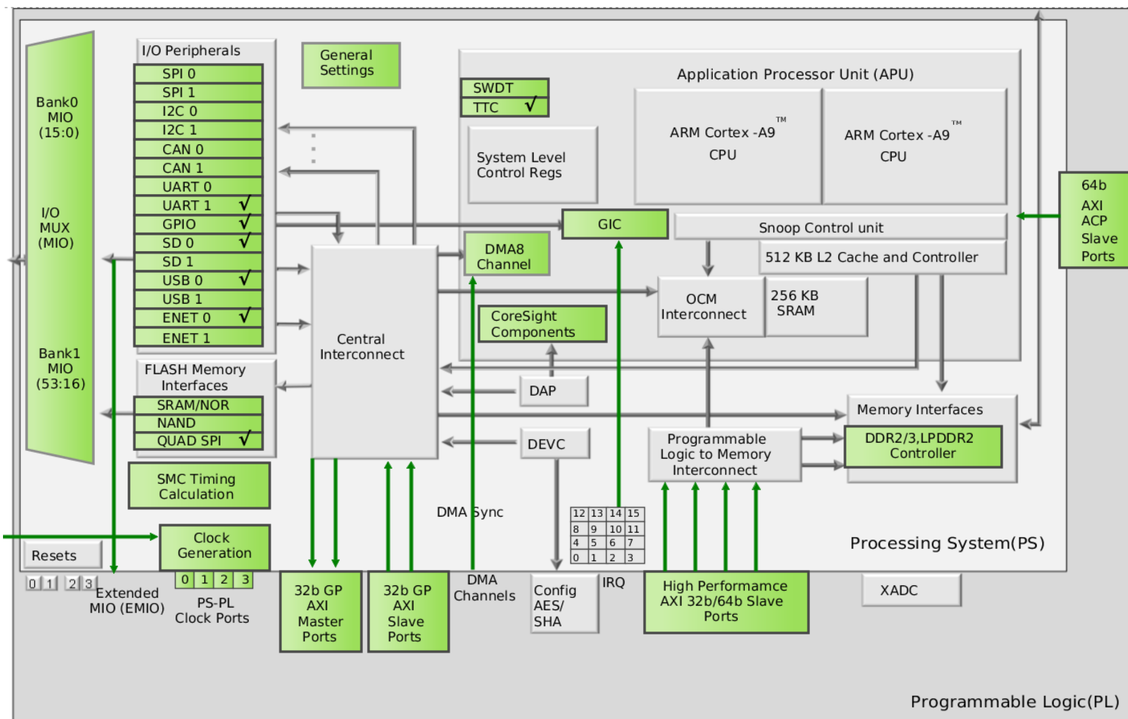

Figure 4.2: Block diagram exported from Xilinx Vivado of the Zynq-7000 architecture. The *programmable system* containing many of the subsystems is contained in a light grey block, while being interconnected with the *programmable logic* marked in dark grey.

The Zynq-7000 system has two gigabit peripherals. They can be seen in figure 4.2 as *ENET0* and *ENET1*. *ENET0* will be used to provide the network required for the user interface further described in section 4.9. On the Zybo development board, a Realtek RTL8211E-VL PHY controller is used to provide a 10/100/1000BASE-T ethernet connection interface to the peripheral.

4.4 FreeRTOS - A Realtime Operating System

Each of the software tasks described in section 4.2 could by varying degree be implemented directly on the programmable system without an operating system. However, due to the complexity of having multiple different functionalities that may not require the same priority in terms of processing time, it becomes very hard to find arguments against the use of an operating system.

An effective step in cutting down on complexity is to use multithreading. Multithreading will be explained in section 4.5.

FreeRTOS, or Free Real Time Operating System, is essentially an operating system to run bare bone applications with full multithreading support. It does not offer much more than that, and this separates it from running a Linux distro such as PetaLinux on the system. Using a Linux distro would probably cut some development time as there are quite a lot of resources and "out of the box"-solutions easily

implemented to do the job asked by ALOFT. A Linux distro would however draw more resources than FreeRTOS, as the solutions would not be specifically designed for the task at hand, but be adapted to fit in a "good enough"-solution.

As FreeRTOS is an RTOS, it can operate under strict defined time constraints. FreeRTOS seems like a perfect compromise between constraining computing- and power-resources, and providing an efficiently platform for development of a complex system. FreeRTOS' core source files are also conform to MISRA coding standard guidelines [31].

4.5 Multithreading

The following section along with subsections are based on [32].

Applications can contain multiple tasks which themselves can occupy separate threads. For a single core system only one task can run at a time, but multithreading makes it possible to switch between these tasks so that they behave as if they were running concurrently. The actual speed is, however, limited to the execution of one task at a time. Dual core systems, such as the one found on the ARM Cortex-A9, enables the possibility of running two concurrent tasks. This could be beneficial if the system will ever require data rates which saturates the single core. For now, ALOFT is designed to only use one of the cores, as it should provide enough performance. Designing ALOFT for single core usage also cuts down on system complexity and challenges related to development.

4.5.1 Scheduler

As only one task can be executed at a given time in a single core system, a framework for deciding which task to be executed is essential to a multithreading system. This decision process is known as *scheduling*. To help in deciding which task to execute, the tasks are marked with a priority. There will also always be an *idle*-task created by the operating system's kernel that will have the lowest priority. For tasks to be able to run, they need to have a higher priority than the idle task. Scheduling is handled by a "scheduler", and FreeRTOS has one built into the kernel. It operates in a *round-robin* policy mode, meaning that each task of the highest priority will be executed by turn. It does not, however, guarantee that the available processing time will be spread equally between them.

Up until now, a simplified view of tasks being either in a *running* or *not running* state has been described. The reality of the FreeRTOS kernel is however somewhat more complex, as there are actually four different task states; *Running*, *Ready*, *Suspended*, and *Blocked*. Figure 4.3 illustrates how task transitions are performed when different functions are called. If all tasks with the highest priority are placed in the blocked or suspended state, the scheduler will move on to the tasks with the second highest priority.

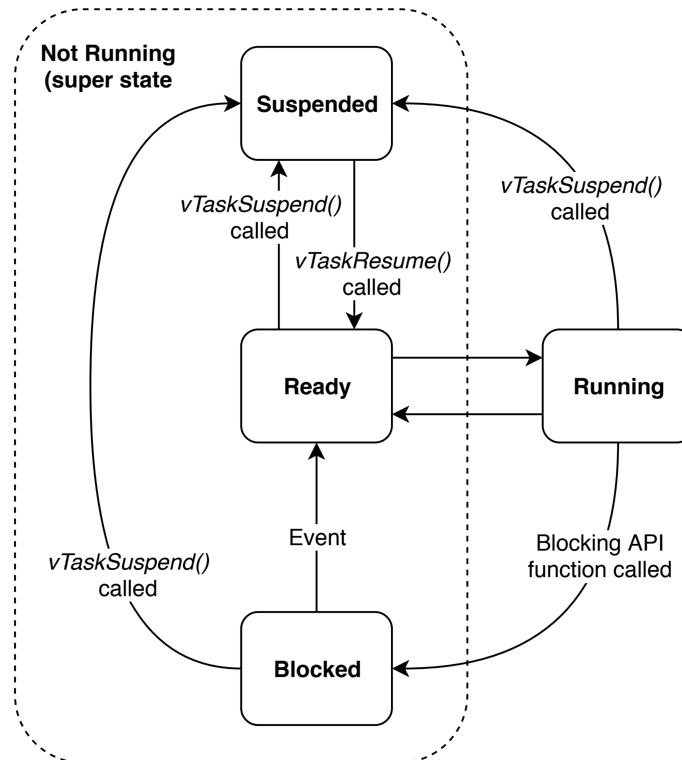

Figure 4.3: State machine for the four different task states available in FreeRTOS, Running, Ready, Suspended, and Blocked. Modified figure from [32].

For ALOFT, the scheduler algorithm is in a *prioritized pre-emptive scheduling with time slicing*-configuration. This means that if a higher priority task is put into the ready state when a lower priority task is in the running state, the lower priority task will immediately be pre-empted out of the running state so that the state is available for the high priority task. Maximum task priority is set to 8.

To avoid the highest priority tasks in ALOFT starving the lower priority tasks of run-time, all tasks are put into the suspended state for a limited time period when they have reached the end of instructions, unless deleted. Putting tasks into the suspended state is done with the $vTaskDelay()$ function call. It takes a number of clock-cycles as input. If the function is combined with the $pdMS_TO_TICKS()$-function, suspending can be defined in milliseconds.

Idle task

When the scheduler is started upon boot, the idle task will be created automatically. As it has the lowest priority of all the tasks, it will only execute when there are no other tasks in the ready state. The idle state can be used as a way to measure how much of the processing time that is actually used by the system. As it does not process any significant work, it reduces the processor's power usage when operating in the running state.

4.6 Memory allocation

In programming, there are three ways of allocating memory. *Static*, *automatic*, and *dynamic*. Automatic and dynamic allocates the memory at run-time, therefore making it nearly impossible to determine the actual memory footprint. They can also produce some nasty thread-behavior if not used with great care, as the program might run into memory leaks or prove non-deterministic with regard to execution time. Static allocation allocates the application writer's requested amount of memory at compile time. Usage of static allocation creates a deterministic memory footprint, and execution time can be specified by the application writer.

As the MISRA-C guidelines have been applied to this project, no use of dynamic memory allocation is permitted unless great care is taken [33, p. 34, Dir. 4.12]. This also covers automatic allocation. To comply, all usage of memory in the ALOFT software has been statically allocated to its greatest extent.

4.7 First Stage Boot Loader

References for this section can be found in [29].

The configuration of the programmable logic is stored in Static RAM (SRAM)-type internal latches, and thus is volatile. This means it has to be reloaded every time power is cycled on the Zynq-7000 device. Reloading can be performed by the programmable system using a bitstream-file.

The application running on the programmable system is stored in a memory module known as the *On-Chip Memory*. The Zynq 7000 uses a power-on reset function which resets all reset-capable registers when power is connected. Thus the application must be uploaded to the on-chip memory every time power is cycled. Uploading to the on-chip memory is handled by a *Read-Only Memory* known as the BootROM.

The device serving the boot image is known as a Master boot device, and in ALOFT this is the micro SD memory card. A First Stage Boot Loader (FSBL) contains the programmable system application, and can in addition also contain the bitstream used to configure the programmable logic. The FSBL has in the previous chapter, and will be further, referenced to as the *bootloader*.

In ALOFT, the bootloader has been created so that it contains the FreeRTOS programmable system application, and also the bitstream to configure the programmable logic. The bootloader is placed on the micro SD memory card. With this solution, no external computer is needed to program the ALOFT instrument as it is done automatically from the SD-card partition when power is connected. The bootloader has been generated using Xilinx' software program *SDK's* built-in bootloader-tool along with bitstream exported from Xilinx Vivado. Changes to the software or bitstream requires regeneration of the bootloader files. A tutorial on how to generate the bootloader has been written and posted to the internal wiki-page of the University of Bergen. Link to this tutorial can be found in appendix F.

4.8 Storage

For development, a 128 GB Samsung Evo Plus Micro-SDXC card has been chosen. The card is rated to class 10, meaning it can sustain a 10 MB/s minimum sequential write speed [34]. With the expected data rate and capacity requirements elaborated in section 4.2.2, this card will not pose any risk of introducing bottlenecks related to these requirements. The card uses 3D Tripple Level Cell (TLC) NAND technology, so the reliability may prove too low for use for other than testing purposes. SLC-based cards are more suiting to the task, but usually comes with an increased cost. For development, criteria was put on speed and capacity, and thus TLC technology proved the best from an economically standpoint.

File System

According to [29], the Zybo can only be booted from FAT 16/32 file systems with max card density of 32 GB. As the card used for development has a larger density, it has been formatted into two separate partitions were only a smaller 32 GB partition is used. Xilinx provides a library for usage of the FAT 32 file system. FAT 32 does however inherent a problematic attribute; The industry standard has volume tables limited to 2^{32}-1 sectors. This makes the maximum individual file size the system can handle around 4.3 GB, not compliant with the up to 7 GB of data being accumulated. Section 5.8 will cover how this is solved by spreading the data over multiple smaller files instead.

4.9 User Interface

Section 3.5 described how the user interface of FEGS BGO was available only through the EGSE.

One feature of the ER-2 aircraft which FEGS BGO did not use, was the *ER-2 Instrument Network*. In addition of providing aircraft housekeeping data and NTP time services to the instruments, the ER-2 instrument network can also provide a simple communication link between an instrument and ground web services. The service is handled over an Iridium or Inmarsat satellite link, depending on requirements. Iridium packets are, however, often lost, and the link provided by Inmarsat has a cost of $5.88/MB for data traffic above 6 MB/hour/aircraft. [35]

To be able to access the user interface while both at ground and in-flight through the ER-2 Instrument Network, it has been found that running a *Telnet* server on ALOFT provides a good solution. This non-GUI solution is extremely lightweight which severely reduces resource usage on the instrument itself, and data usage is very low compared with the RDP solution used on FEGS BGO. This means that the interface can be used both with a direct point-to-point link between the instrument and an external EGSE computer, and with the ER-2 Instrument Network for remote in-flight access, without adding large costs for data traffic. More resource-intensive tasks such as running a GUI overlay can instead be offloaded to the EGSE or ground services using standalone software. With the GUI overlay, the user will be able to send and receive commands with a much more user-friendly interface with for example command-predefined buttons, instead of the command-line type of interface

provided by telnet.

Accessing the instrument trough telnet still requires a cable harness as used on the FEGS BGO instrument, and as of now requires the physically removal of the micro SD memory card to offload data. In section 7.2.4, it is proposed how the data offloading may be performed over TCP/IP at a later step in the development.

4.9.1 Communication protocol

Telnet is a widely used protocol used over an ethernet connection. It is compliant with all large operating systems such as Linux, Windows, and Mac OS. Open source software for communicating with telnet clients and servers are also widely available for these OSs. Telnet makes accessibility easy and standardised, and removes the need for proprietary software. Connection with the instrument will be on a closed network provided by the ER-2 Instrument network, or a direct point-to-point link. It should be noted that telnet does not provide any security as the data is sent in plain text. However, as the network is closed, security such as the one offered by the Secure Shell (SSH)-protocol is not a necessity, and would only add additional overhead.

Sending packets of bits over a network using telnet is handled with either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). The ER-2 Instrument Network requires instruments to use UDP, although it does not contain the error detection and correction offered by TCP [35]. This, however, means higher transfer rates can be achieved due to the minimal packet overhead. Data costs of the Inmarsat link are also reduced as total data amount is lower.

4.10 Cockpit Experiment Control Panel

During flight, interaction with instruments is in addition to the ER-2 instrument network available through the Cockpit Experiment Control Panel (ECP). The ECP is operated by the pilot and consists of a simple ON/OFF switch, and two status lights; an "ON" light, and a "FL" fail light. Request of additional switches is possible, but in general it is expected that the instrument should handle itself when power is provided.

The "ON" light should be illuminated as long as the instrument receives power. Illumination of the fail light should only be used to signal the pilot that there is a problem with the instrument. However, it is allowed for the "FL" light to be illuminated while the instrument is powered up, but should be turned off when data acquisition has started. The standard go-to procedure when the "FL" light illuminates, is for the pilot to cycle instrument power with some timing constraints. If the problem persists, a decision of aborting the mission will be made. [36]

As ALOFT will piggyback with the FEGS instrument, it is not expected that it will have a dedicated switch and status lights on the ECP. However, if this decision is changed, it would be a feature easily implemented in the software by making it toggle a set of GPIO pins.

4.11 GPS interface

The Copernicus II GPS receiver used to generate the PPS-signal for the RCU on FEGS BGO, will also be used on ALOFT. The receiver is interfaced using the serial communication standard RS232 with low voltages, and can therefore be handled directly by the programmable system. [24]

4.12 Chapter Summary

In this chapter some of the design fundamentals to ALOFT has been decided. The system will be implemented on a Xilinx Zynq Z-7010 SoC with an embedded dual core ARM Cortex-A9 processor and 28 nm programmable logic. A micro SD memory card will be used to store the data coming from the detectors, and hold other files needed by the system.

The software running on the processor will be implemented in the FreeRTOS Realtime operating system, due to the software complexity. FreeRTOS enables multithreading which reduces the complexity of implementing all the features required.

The required features are to handle offloading of the detector data to a micro SD memory card, provide a user interface for controlling the instrument, log the system status to a log file, and configure the firmware running on the programmable logic.

User interface is to be provided through a telnet server which will be accessible both on the ground and while in-flight through the ER-2 Instrument Network.

Chapter 5

Embedded Readout Development

As recommended by [32], a demo application provided by FreeRTOS has been used as a template for developing the ALOFT software. The most promising one was the *lwIP Echo server* made with Light-weight Internet Protocol (LwIP) v2.0.2. It comes bundled with FreeRTOS 10 in Xilinx SDK 2018.1. The template contains a bare minimum framework for hosting a telnet server for a variety of different hardware platforms. By default, the implemented functionality is to echo back any message it receives on port 7.

To make the FreeRTOS able to function in a hardware environment, a set of hardware drivers are needed. These are provided through software called a Board Support Package. By using the LwIP template, the necessary board support packages comes preconfigured to allow use of the network hardware. To allow use of the fat32 filesystem, Xilinx's library called *xilffs* has been added manually to the board support package. The package has been configured to allow usage of functions capable of reading and writing full strings, not available by default.

5.1 System architecture

Figure 5.1 displays the architecture of the design developed in this thesis. The DAU and ADCs are identical to the ones used on FEGS BGO. Data from the ADCs are inputted into the firmware on the programmable logic. The programmable logic is connected to the programmable system using two separate AXI-buses. The figure gives a somewhat simplified view of the interconnection between the programmable system and programmable logic. A more detailed view of the interconnection with each of the AXI-buses can be found in section 5.8 and 5.9.

Software on the programmable system can read and write to files to the micro SD memory card. These includes a log file, a file containing configuration parameters for the firmware, a file containing data used for testing, and a number of files storing the data produced by the detectors. In addition, the bootloader is stored on the same partition.

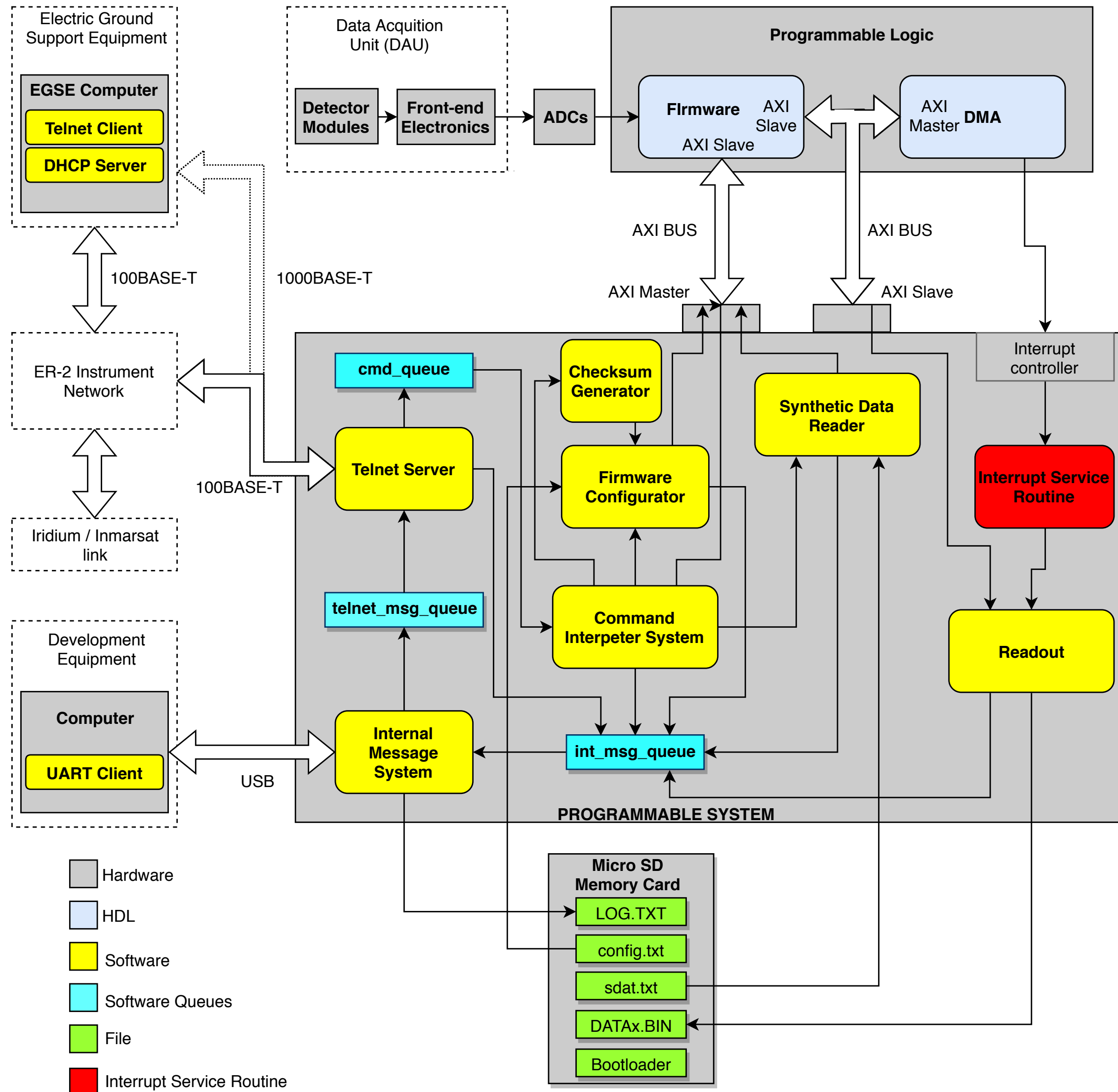

Figure 5.1: Architectural design of the ALOFT software.

Interprocess communication between the tasks has been solved with the use of queues which are marked in a cyan colour in the figure. In general, this was only needed for the system logging and user interface functionality. The use of modularity also removed some of the need for interprocess communication as the tasks works mostly independent of each other. Modularity is further explained in appendix C.

Table 5.1 lists the priorities set for each of the software tasks implemented in the programmable system. In the programmable system in figure 5.1, the tasks have been marked in yellow. A short description of each of the tasks follows, and are further documented in their respective sections later in this chapter.

Name	**Priority Level**
Readout	8
Synthetic Data Reader	7
Checksum Generator	6
Firmware Configurator	5
Internal Message System	4
Command Interpeter System	3
Telnet Server	2
Idle Task	1

Table 5.1: Priority level of the different tasks

Readout

The *Readout*-task is responsible for offloading the detector data transmitted from the firmware onto the micro SD memory card. It is triggered to start offloading when the programmable system receives an interrupt from the programmable logic. It is important for this task to complete the offloading before a new interrupt is sent, to avoid loosing data. Therefore, the task is executed with the highest priority. The *Readout*-task is further documented in section 5.8.

Synthetic Data Reader

The *Synthetic Data Reader* is used to write synthetic test data to the firmware to verify the readout system. The task is executed with priority 7, which is the second highest in the system. This is to avoid interruption by the scheduler doing a context switch to another task, as it would make it harder to determine the timing characteristics of the relationship between data being inputted into the firmware, and readout. The *Synthetic Data Reader* is further documented in section 5.9.

Checksum Generator

The *Checksum Generator* calculates a checksum used to verify the content of the configuration file used by the *Firmware Configurator*. As the *Firmware Configurator* is dependent on this calculation, the task is executed with a priority of 6, one priority over the *Firmware Configurator*. The *Checksum Generator* is further documented in section 5.7.1.

Firmware Configurator

The *Firmware Configurator* parses configuration parameters from the file checked by the checksum generator, and uses them to configure the firmware on the programmable logic by writing to its registers. The task is executed with a priority set to 5, one priority lower than the *Checksum Generator*. The *Firmware Configurator* is further documented in section 5.7.

Internal Message System

The *Internal Message System* is a centralized messaging system used to provide logging and task status monitoring in a multithreading environment. The messages are written to a log file for permanent storage, the UART debugging interface over USB, and to the *Telnet Server* for use in the Telnet user interface. Due to the importance of the logging, the task is executed with the highest priority of the tasks which are not polled by other stimuli. The priority is set to 4. Further documentation of the *Internal Message System* can be found in section 5.4.

Telnet Server

The *Telnet Server* hosts the user interface, which can be accessed either by a computer connected with a point to point link, or through the ER-2 Instrument Network. The server outputs the messages received by the *Internal Message System*, and can receive user input commands which are parsed and transmitted to the *Command Interpreter System*. Due to its dependence on the *Internal Message System*, it is executed with a priority set to 3. Further documentation on the *Telnet Server* can be found in section 5.5.2.

Command Interpreter System

The *Command Interpreter System* is responsible for executing the correct response based on user input through the user interface. Due to its dependence on the *Telnet Server*, it is executed one priority lower, at 2. As the system has been design to handle itself, user input is regarded as having the lowest importance, and thus this task has the lowest priority in the system. The *Command Interpreter System* is further documented in section 5.6.

5.2 Boot Sequence

Figure 5.2 demonstrates the boot sequence of the software. When power is connected and the software application has been loaded from the bootloader, the *queues* used in the application are created. Next, the file system partition on the micro SD memory card is mounted for access by the application. As neither the readout system nor the internal message system will function without it, it is essential that the mounting task is initiated prior to them. The micro SD memory card initialization is further described in section 5.3. If the initialization is successful, a register in the firmware containing the firmware number is read. Reading the register serves two purposes; Verifying the correct version number, and to ensure the programmable system has access to the registers of the programmable logic.

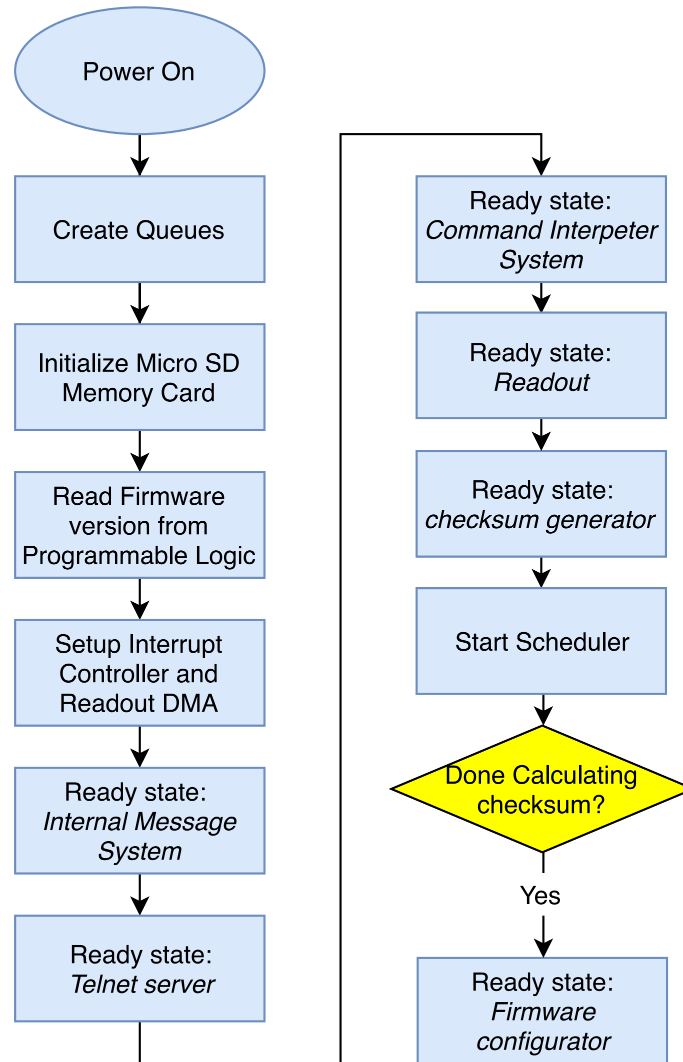

Figure 5.2: Boot sequence for the ALOFT software application.

Next, the *interrupt controller* on the processor and the Direct Memory Access (DMA)-module used by the data readout are configured. The DMA is further described in section 5.8. When configured, the tasks handling the *Internal Message System*, the *Telnet Server*, the *Command Interpeter System*, the *Readout*, and the *Checksum Generator* are all created and put into the *ready state*. None of the tasks can be put into the running state before the FreeRTOS scheduler has been started. When the scheduler is started, the readout task has the highest priority and thus is the first to be put into the running state. Section 5.8 explains how the readout task is then immediately put into the blocked state.

The checksum generator task has the second highest priority. As the readout task is put into the blocked state, the checksum generator is put into the running state. When the checksum generator has completed, the *firmware configuration* task is created and put into the running state.

5.3 SD Memory Card Initialization

To use the SD memory card as an application storage device, it must be mounted by the operating system before it can be used. Mounting is a process in which the operating system identifies the type of file system present on the storage device, and initializes it for read and write operations. When completed, the file system on the storage device is incorporated into the file system of the operating system through a mount point where it can be accessed by an application.

In ALOFT, the syntax for mounting and checking for a successful mount has been placed in a separate function. The function is called during the boot sequence as seen in figure 5.2. This mounting function could have been implemented as a separate task, but should only be required to run once during boot. A standalone function is however sufficient to avoid overcomplicating the implementation.

The function comprises two parts; mounting the SD memory card, and checking if the mount was successful. The first part is done by declaring a file system pointer later used by all tasks using the file system, a path to specify which logical drive the card is to be mounted to, and last, a variable to set a *delayed mount*.

As there is only one storage device to be used, the logical drive path is set to *0:/*. As will be seen in section 7.2.1, the finalized system may be designed to use different devices for the different files. In such case, the logical drive path must be set accordingly. The *delayed mount* option, if set, means that the file system will not be mounted until access to the card is requested by the application. For ALOFT, delayed mount has been disabled for easier debugging of storage device-related problems during boot.

5.4 Internal Message System

To enable monitoring of the system health, each task and function can report their status which can be viewed through the user interface and a log file. In figure 5.3 the architecture of the implemented message system can be seen. When a task is in the running state, a message can be sent into the *Internal Message Queue*. The queue is 16 elements deep. Messages are always written to the end of the queue which operates with a First In First Out (FIFO) scheme. When writing to the queue, the scheduler is denied context switching. This is to avoid other tasks being put into the running state, which would result in only parts of a message being written to the queue.

The messages consists of two parameters implemented with the C-object type struct; a system message code, and a message in plain text. The system message codes are further described in section 5.4.1. A task called the *Internal Message System* reads message by message from the receiving end of the queue.

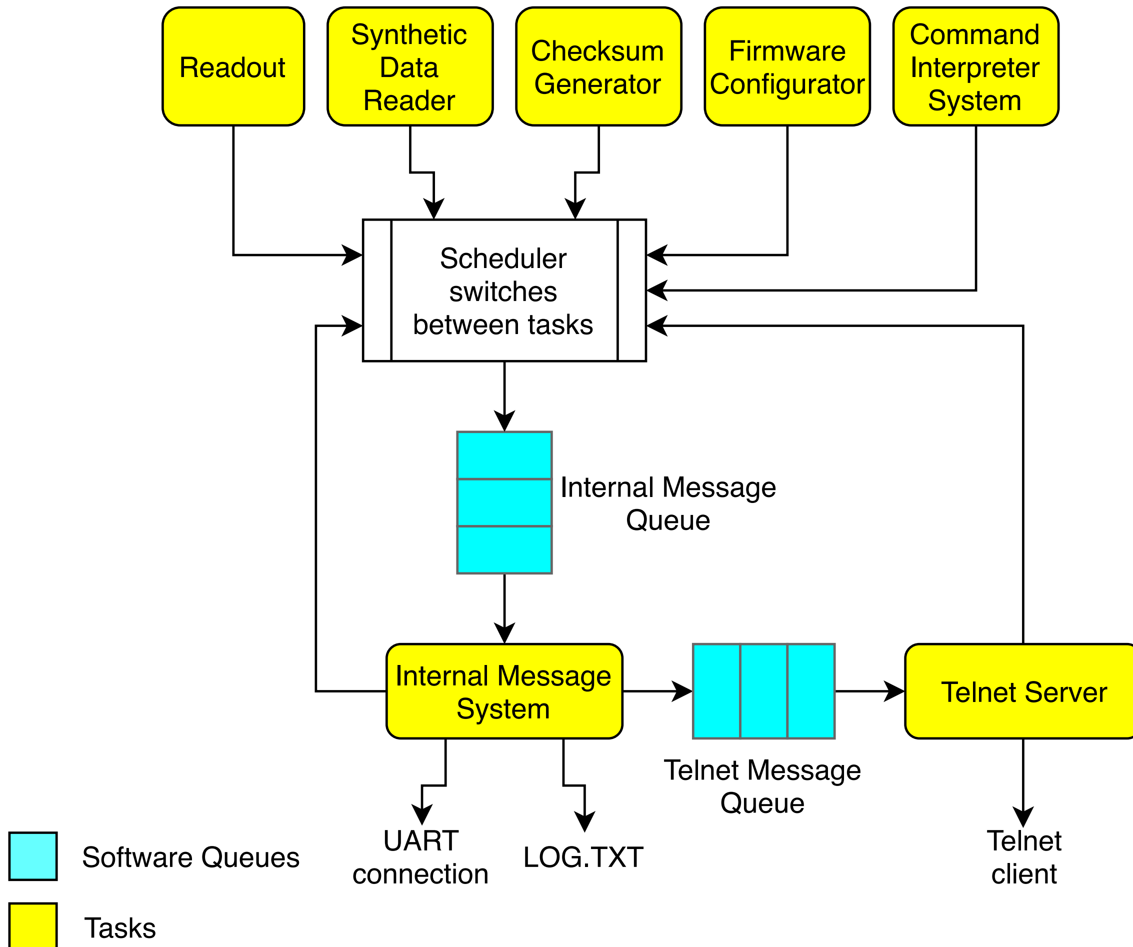

Figure 5.3: Design architecture and flow of the messaging system used to report information from the different tasks.

Figure 5.4 illustrates the flow of the task which is implemented as a loop. First, the task checks if the internal message queue is empty. If it is, the task is put into the blocked state for a total of 10 ms to avoid starving the processor of processing time. If the queue is not empty, a log file is opened from the micro SD memory card. If the file is not already present on the file system, the file is created by the software.

After the file has been opened, one message from the queue is written to the end of the file. As the software always writes to the end of the file, there is no risk of loosing data by overwriting the log content. When completed, the file is closed. Next, the message written to the file is printed to the Universal Asynchronous Receiver-Transmitter (UART)-interface which is further explained in section 5.5, and written to the end of a queue named the *Telnet Message Queue*. The *Telnet Message Queue* is 16 elements deep, and is used by the *telnet server* task documented in section 5.5.2.

The internal message system task has been implemented with the highest priority of the tasks apart from the ones only placed in the ready state once or on request. This is to ensure that the internal message queue is never overflowed, which could result in debugging-critical messages being lost. To ensure tasks with lower priority are not starved of processing time, the task is put into the blocked state for 10 ms

between each message readout.

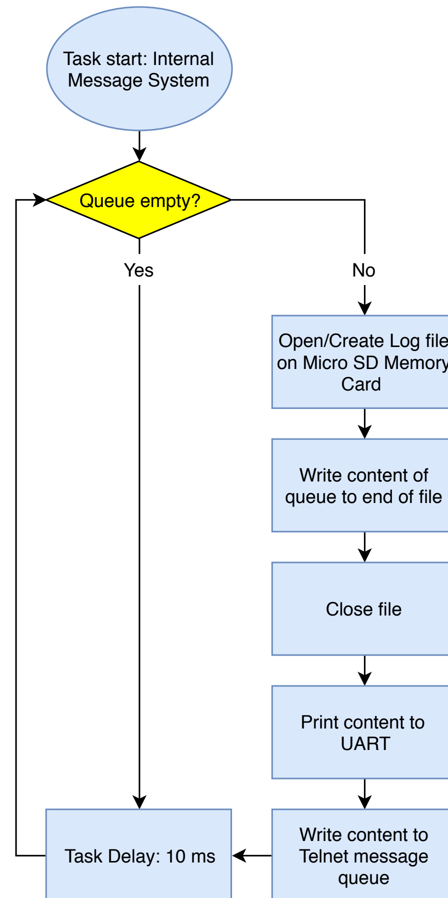

Figure 5.4: Design architecture and flow of the messaging system used to report information from the different tasks.

5.4.1 System Message Codes

All status messages can be categorised as either informational, warning, or error. The first element in a message is a 3-character code which has been called a *system message code*. These are used to help identify the nature and whereabouts of the message. The four main types are characterized by the first character in the 3-character number, and can be seen in table 5.2.

System Message Code	Type	Description
0xx	OTHER	Used for display formatting and similar uses
1xx	INFO	Information message
2xx	WARNING	Indicating that something might be wrong
3xx	ERROR	Something in the system has failed

Table 5.2: Types of system message codes implemented in the software.

The second character tells which subsystem the message was sent from. Table 5.3 displays the mapped numbers. The only exception to messages coming from the originated subsystem, are those related to task handling such as creating or deleting tasks. They should always be identified by the subsystem number '0'. A full list of codes can be found in appendix E.

System Message Code	**Subsystem**
x0x	Task
x1x	Queues
x2x	SD memory card
x3x	Telnet server
x4x	Command Interpretation System
x5x	Readout
x6x	Checksum Generator
x7x	Firmware Configurator
x8x	Synthetic Data Reader

Table 5.3: System Message Code-identities for different subsystems.

5.5 User Interface

As discussed in section 4.9, using a telnet server to host the instrument's user interface has been found to fulfill the necessary requirements. In figure 5.1 the telnet server task can be seen in the system architecture of ALOFT. The telnet server provides a full user interface, capable of both outputting information, and receive user input from a network client. Further information about the telnet server is found in section 5.5.2. In addition, a UART interface has been implemented to help debugging during the development.

5.5.1 UART

In section 5.4, it was explained how the *internal message system* printed messages to the UART interface. As the messages are the same as the ones available with the telnet interface, the UART interface can be used as a secondary way of monitoring the system status. Input-capabilities through the UART interface has not been implemented due to its intended usage as a debugging tool. Messages to be transmitted are not stored in any way by the software. This means that no message history can be obtained upon connecting through the UART interface, which differs from the telnet interface.

On the Zybo, the UART interface is available over USB. The USB port on the Zybo is a shared connector between a JTAG, and the UART interface. It can also be used to power the Zybo. The JTAG interface has been used to program the system during development. This bypasses the much more time consuming process of creating and booting from a bootloader, which is inefficient when development is performed in fast iterations. From the USB connector, an FTDI FT2232HQ *USB-to-UART bridge*

converts the USB packets to UART data. The default protocol parameters needed for the connection can be seen in table 5.4. During development, the text based modem control software *Minicom* was used to connect to the UART interface.

Parameter	**Value**
Baud Rate	115200
Character length	8 bits
Parity	No
Stop bit	1

Table 5.4: Default protocol parameters for the UART-interface.

5.5.2 Telnet Server

The implemented telnet server has been developed based on the FreeRTOS *lwIP echo-server*-demo. The LwIP is configured to run in socket-mode as it is used in a multithreading environment. It should be noted that it is not a full telnet server, but uses the standard telnet TCP/IP port 23, and access can be performed using a standard telnet client. Figure 5.5 illustrates the flow of the telnet server as implemented in ALOFT. When the task is started at boot-time, the network is configured according to the settings which will be explained below.

The ER-2 instrument network normally requires the connected instruments to use 100BASE-T networking. It is possible to set the PHY controller to always use 100BASE-T, but instead auto-negotiation has been kept activated. This enables the possibility of using 1000BASE-T which is rated for much higher data transfer rates. Auto-negotiation is the feature in which network hardware automatically detects and configures itself to communicate over the network with the fastest available standard of data transfer rates. With auto-negotiation activated, ALOFT will automatically be configured to use the 100BASE-T standard when connected to the ER-2 instrument network, and 1000BASE-T when connected using point-to-point linking with a computer which supports the 1000BASE-T standard. This is beneficial due to the much higher transfer rates provided by 1000BASE-T if data offloading to a computer over a point-to-point network connection is ever implemented. This is proposed in section 7.2.4.

If the auto-negotiation fails to detect the speed of the connection, the software is halted. Therefore, proper connection should be ensured before the system is booted. Alternatively, auto-negotiation should be disabled by default.

The ER-2 instrument network uses static IP-addressing and contains a Dynamic Host Configuration Protocol (DHCP) server. The DHCP server offers the IP addresses to the instruments. Addresses are in the 10.9.x.x and 10.6.x.x private domain, depending on which of the two available aircraft are used. On ALOFT, the telnet server is configured to use DHCP. With DHCP activated, the instrument receives an IP address from the ER-2 DHCP server automatically. Should no IP address be received, the telnet server does a failover to the settings seen in table 5.5.

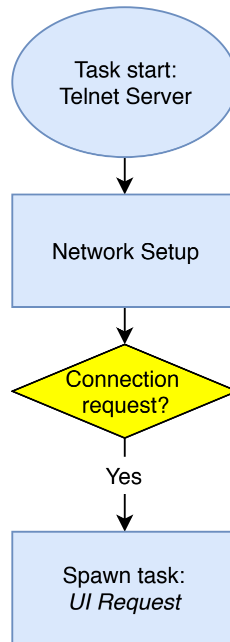

Figure 5.5: Flow chart of the telnet server task.

Board IP	10.42.0.2
Netmask	255.255.255.0
Gateway	10.42.0.1

Table 5.5: Default configuration parameters should the DHCP-request fail.

When the telnet server task has successfully set up the network, it awaits a connection request on network port 23 from a telnet client. If connection is made successfully, a new task is spawned called *UI request*. A flow diagram of the *UI request* task can be seen in figure 5.6.

The task starts by checking if the client sent any commands by reading from the socket. If the socket is empty, the task goes on to check if the internal message queue is empty. If also the queue is empty, the task goes back to start. If the queue is not empty, one message is read from the front of it, and written to the socket so that the client can receive it.

If a command is received from the client, the task goes on to parse the received string. Each command can consist of up to three parameters; The command, an address, and a value. The parsing function separates these parameters with a *space*, as delimiter character. When successfully parsed, the three parameters are sent using structs to a queue called the *command queue* as seen in figure 5.1.

If a client disconnects, the task is deleted.

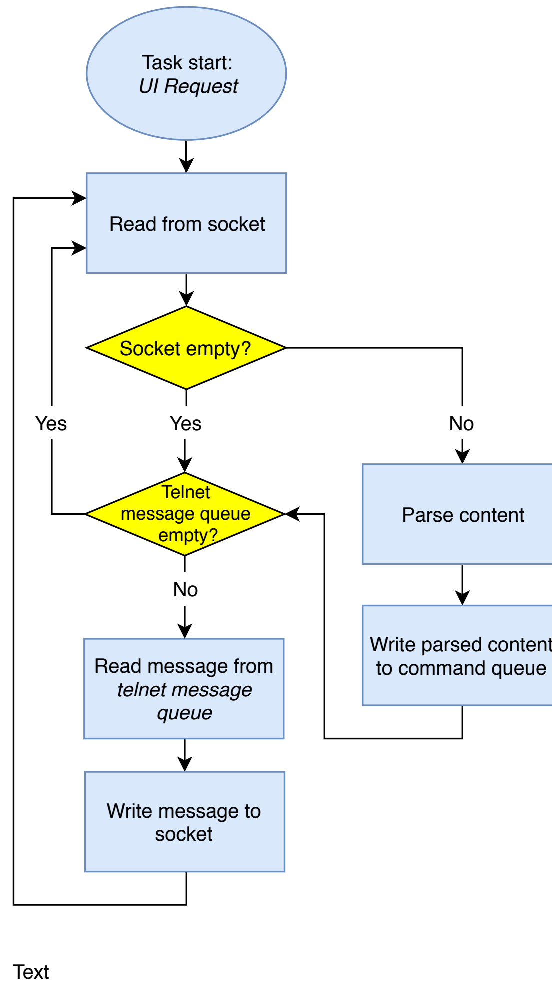

Figure 5.6: Flow chart of the task handling the user interface.

Listing 1 illustrates the user interface presented when connection is made through the UART interface and the system is booted. The telnet message queue is 16 elements deep. This means that only the last 16 messages processed by the *internal message system* are written to the socket when a telnet client connects.

```
110 :: Clearing internal message queue: SUCCESS
110 :: Clearing ethernet message queue: SUCCESS
110 :: Clearing command queue: SUCCESS
000 ::








000 :: ------ ALOFT detector system ------
000 :: Type 'help' for list of commands
102 :: SD initialization: RUNNING
120 :: f_mount: OK
103 :: SD initialization: DONE
170 :: Firmware version: 209
102 :: Readout: RUNNING
152 :: Readout: Awaiting interrupt
102 :: Checksum Generator: RUNNING
160 :: Checksum Generator: Calculation complete
102 :: Logger: RUNNING
102 :: CIS: RUNNING
102 :: Telnet server: RUNNING
link speed for phy address 0: 1000
Board IP: 10.42.0.2
Netmask : 255.255.255.0
Gateway : 10.42.0.1
331 :: Telnet Server: DHCP request timed out
131 :: Telnet Server: Configuring default IP of 10.42.0.2
```

Listing 1: User interface when connected through the UART interface.

5.6 Command Interpretation System

As was seen in section 5.5.2, the telnet server parses and sends any received commands to the *command queue* seen in figure 5.1. The *command queue* is 8 elements deep. Figure 5.7 illustrates the flow of the task called the *Command Interpretation System*. The task is created at boot time as documented in section 5.2, and has been implemented as a loop.

When the task is in the running state, it polls the command queue for content. If the queue is empty, the task is put into the blocked state for 10 ms to avoid starvation of processing time for the lower priority tasks. If, however, the queue is not empty, one command is read from the front. As was explained in section 5.5.2, the commands can consist of up to three parameters. A switch case statement decides which action to perform based on the first element of the command. Some actions have been implemented as separate functions placed within the task, while others

are implemented as separate tasks such as the *Synthetic Data Reader* described in section 5.9.

When the task has been requested into the *ready* state, or the appropriate function has completed, the task is put into the blocked state. This is to avoid starving lower priority tasks of processing time.

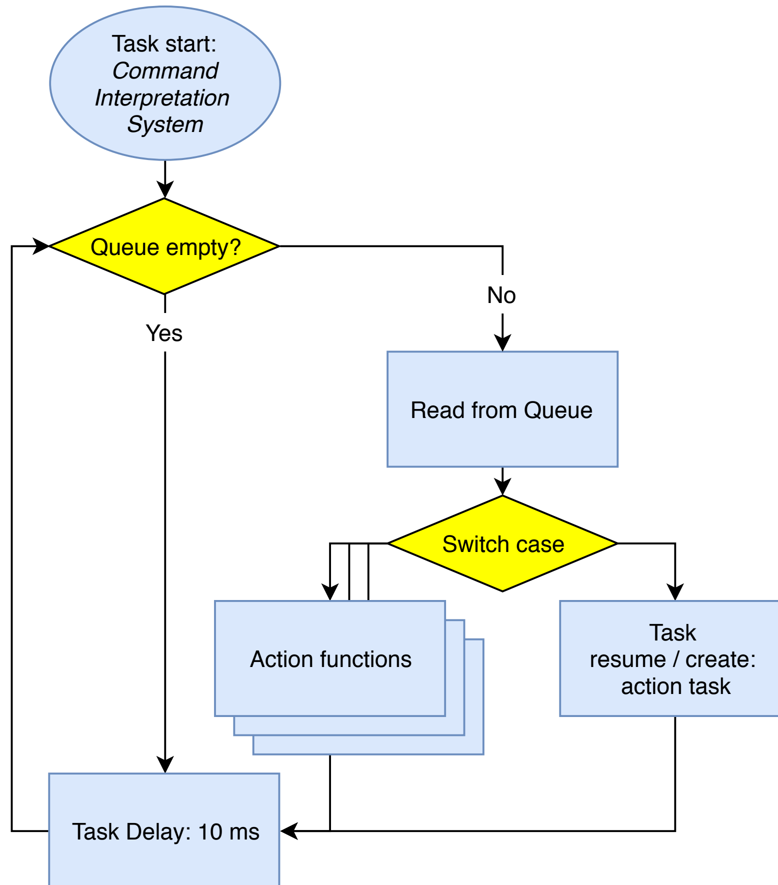

Figure 5.7: Flow chart of the task handling the command Interpretation system.

5.7 Configuring the Firmware

Upon boot, the firmware in the programmable logic is configured with default register values. As the default configuration may not be the desired for the mission, a reconfiguration with register parameters stored in a configuration table on the micro SD memory card is initiated by the software. Figure 5.8 illustrates a simple view of the flow of the configuration task responsible for firmware configuration, named *pl_config*.

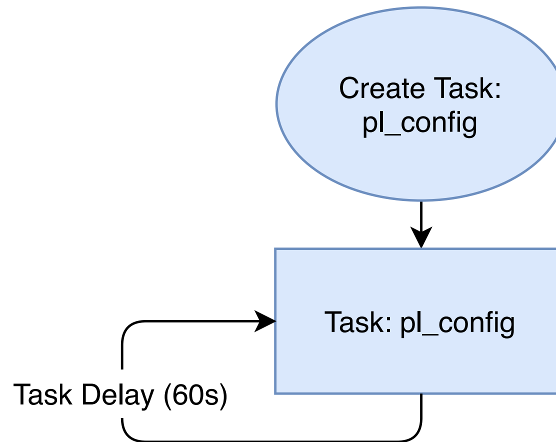

Figure 5.8: Behavioural flow of the programmable logic configuration task.

Due to the possibility of SEEs corrupting the configuration registers, the register values are regularly updated after the initial boot-time configuration. To automatically update the configuration registers, the task has been implemented as a loop which is paused for a set period between each run. When the task is running, it blindly overwrites the content of the configuration registers with the parameters found in the configuration table on the micro SD memory card.

Another possible way of solving the challenge of errors induced by SEEs, would be to read the values of the configuration registers and compare them with the configuration table. This would, however, require more processor time due to the extra read operations, but would in a development phase may provide more accurate information about the severity of the problem.

For testing purposes, the period of which the task is paused has been set to 60 seconds. This is very short, but was used to properly test the feature. From the SEE-estimates calculated in section 2.5.2, it was found that the possibility of a SEE to occur on the Z-7010 during an 8-hour flight is bellow 4%. The estimate was calculated using the total number of bits on the chip. However, the configuration registers in the firmware only account for a tiny percentage of the total number of bits on the chip. This means that the probability of an error that can be resolved by the automatic reconfiguration functionality, is very unlikely to occur. Possibly, the automatic reconfiguration feature will therefore not be required on the finalised system, or at least a much longer delay should be used to limit the processor usage.

Figure 5.9 illustrates the internal flow of *pl_ config*. Initially after the task has been created, it opens a file named *config.txt* containing the configuration table from the micro SD memory card. One potential problem that could arise, is if the configuration table itself has become corrupted by SEEs.

There are multiple ways to overcome this problem. One would be to use redundancy by storing N number of copies of the original configuration table file. As each file would be stored physically in different areas of the memory, it would be statistically less likely for all copies to change their content as N increases. The problem with this approach is that it would occupy a lot of memory, and require much processing time as each copy of the file would need to be compared. It would, however, introduce a

kind of error correction code to the system, that would make it possible to continue operating by rewriting all the copies with the content of the most abundant version.

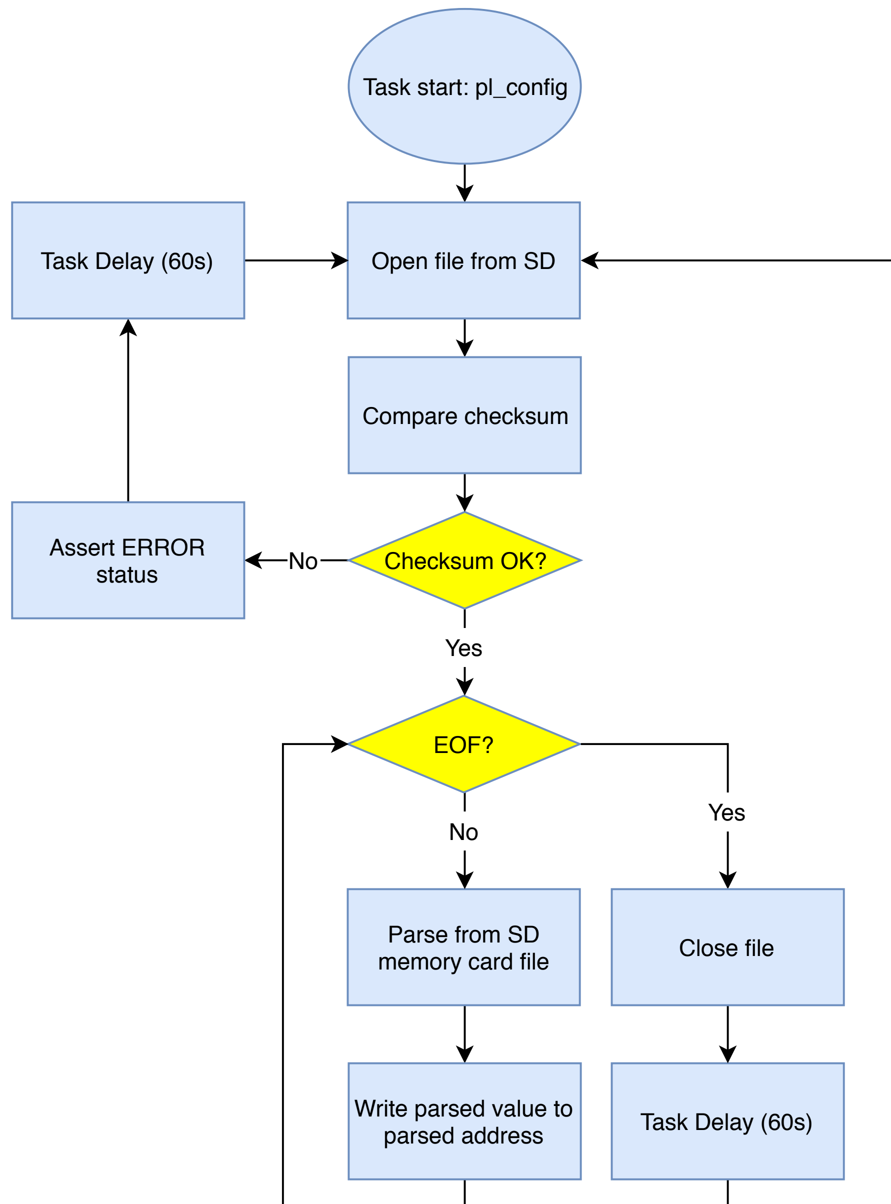

Figure 5.9: Flow chart of the programmable logic configuration task.

For now, error detection will be sufficient. For error detection, a checksum algorithm is used. A checksum algorithm calculates a value known as a "checksum" which will depend on the content of the file. Checksum calculation is further explained in section 5.7.1. If the file content changes, the checksum value will also change. This way, the checksum of the configuration file can be compared with a checksum calculated at boot-time. If they prove to be of different values, the file has been modified and an error message will be thrown.

If the file containing the configuration table is successfully opened, the read pointer is checked to not have reached the end of file. If the result is negative, a parse-function is initiated. Listing 2 illustrates the correct syntax for the configuration

file, for the parsing function to be able read it.

```
XPAR_BGO_TOP_0_BASEADDR,0x10,1
XPAR_BGO_TOP_0_BASEADDR,0x20,1
0x40000000,0x30,1
```

Listing 2: Snippet of configuration listings in config.txt.

Each valid configuration listing contains an *address* parameter, an *address offset* parameter, and a *value* parameter. Each listing must be on separate lines in the file, as the parser uses the newline character to differentiate them. Each of the parameters must be separated by the comma character.

The *address* parameter can be in the form of a defined name in the software such as the configuration listing on line one and two in listing 2, or a hexadecimal number as seen on line three. For readability, it is strongly suggested to use the software defined names. The *offset* and *value* parameters must be in the hex or integer form.

When a listing has been parsed, the value of the *value* parameter is written to the address *address* + *offset*. This parsing and writing is done for each line of the configuration table, until end of file is reached. The file is then closed, and the task is put into the blocked state for a total of 60 seconds, before starting from the beginning.

Manual changes to the configuration table can be achieved through direct modification of the file using a computer capable of mounting the micro SD memory card, or through the telnet user interface with the Memory Write Command (MWC). The MWC-command syntax can be found in appendix A.

5.7.1 Checksum generator

The checksum generator calculates the checksum of *config.txt* at boot time. In addition, it also re-calculates the checksum when the file is manually modified using the MWC-command. Figure 5.10 illustrates the flow of the checksum generator task named *cksum_gen*.

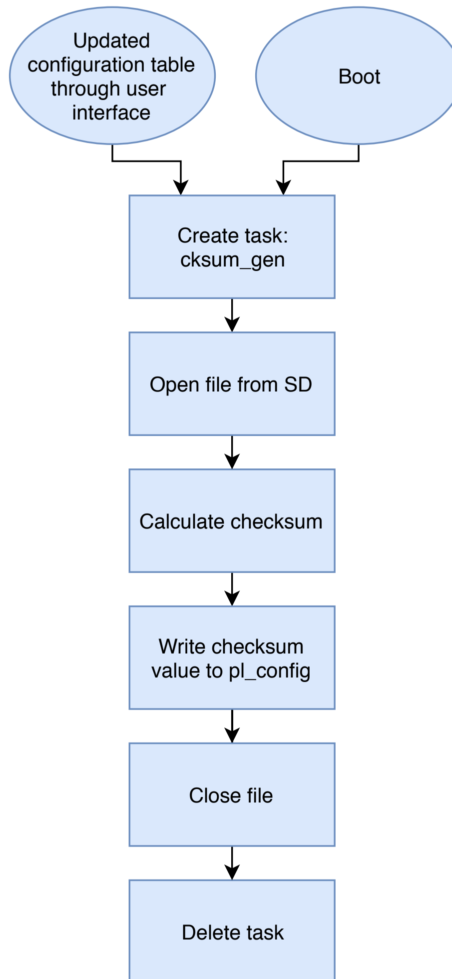

Figure 5.10: Flow chart of the checksum generator *cksum_ gen*.

When the task has been created, *config.txt* is opened from the micro SD memory card. Each character in the file is then read into an array of the unsigned integer-type *uint8_ t* allowed by the MISRA guidelines. The separate values are then added together to form a large number stored in a variable. If one of the characters in *config.txt* changes, the total value changes. When calculation is completed, the result is written to the firmware configurator task *pl_ config*. The file is then closed, and the task is deleted.

5.8 Data readout

Readout and storage of SCDPs from the firmware is the main priority of the software. Figure 5.11 illustrates the readout architecture designed in cooperation with Alexander Nikolai Nesse at the University of Bergen. Design of HDL architecture on the programmable logic is the work of Nesse and is further documented in [26].

BGO Top is the firmware from FEGS BGO. The *Stream wrapper* is an interfacing module written by Nesse to port BGO TOP to the AXI bus. When the stream wrapper transmits an SCDP, the *T_ last* signal is toggled high. As the AXI bus can be configured both as 32 and 64 bit, there are two options for transmitting the 48 bit SCDPs. Option one would be to transmit the packages over the 64 bit bus,

padded with zeros. Transmitting a padding is a waste of resources, and it has thus been decided to instead use the 32 bit bus with a *Width wrapper* module by Xilinx.

The width wrapper uses the T_last signal to buffer a total of ten 48 bit SCDP transmissions. This is equivalent to fifteen 32 bit packages, which is further written into a *FIFO* memory module. 10 packets offers a good tradeoff between speed as transferring chunks of data is faster to perform by the processor due to reduced number of instructions, and the time packets are stored in volatile memory. SCDPs in the FIFO is written to a Block RAM (BRAM) module using a stream to memory mapped DMA module operating in direct register mode. The DMA is activated by a *T_last* signal transmitted by the width wrapper when data is transmitted to the FIFO.

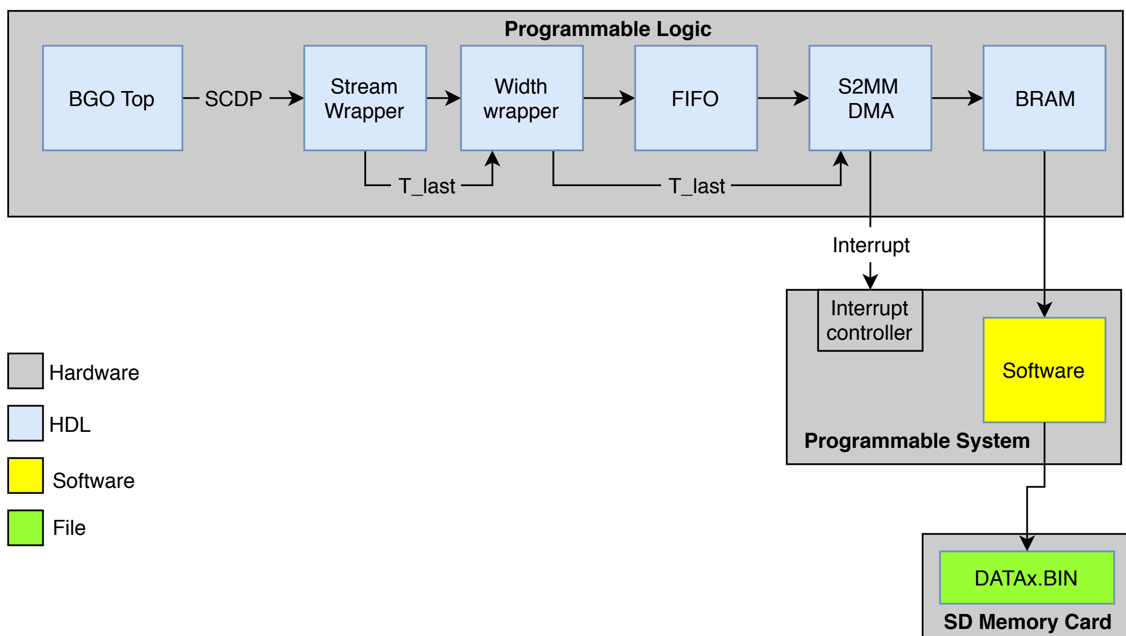

Figure 5.11: Data flow of the hardware readout.

The DMA has been configured to transmit an *interrupt* signal when a transmission from the FIFO to the BRAM has been completed. This *interrupt on complete* signal is received by the interrupt controller in the programmable system. Figure 5.12 illustrates the software behaviour when the interrupt is set. The readout task is created at boot-time, and then immediately put into a suspended state. The interrupt then triggers the task to resume. The SCDPs in the BRAM are read, and then written to a file on the SD memory card. When the task is done, it returns to a suspended state, awaiting a new interrupt from the DMA.

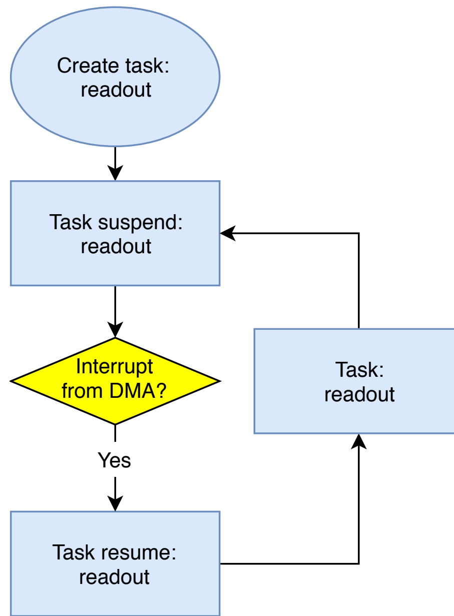

Figure 5.12: Behavioural flow of the software readout task.

Figure 5.13 illustrates the flow of the readout task as well as the interrupt handling flow. At boot-time the readout task *readout* is created, running with the highest priority compared to the other software tasks. The task is implemented as a loop, and a task suspend command is immediately called at the start of it.

The interrupt from the DMA makes the processor switch to an Interrupt Service Routine (ISR). Syntax allowed in an ISR is very strict, and the performed work possible is therefore very limited. The solution has been to call an ISR-safe task resume command to resume the readout task. In addition, the ISR resets the interrupt status, and re-enables the DMA run control.

When resumed by the ISR, the readout task opens a binary file on the micro SD memory card called *DATA1.BIN*. If the file does not exist, it is created. Due to the limitations of the fat file system as explained in section 4.8, the file size is limited to 2^{32}-1 Byte. Therefore, each time the readout task opens the file, it checks its size. If the file is larger than 4 GB, the file name to be opened next time the task loops is changed to *DATA1.BIN*. The number in the file name increases each time this happens. The number starts at zero. Changing the threshold of when to create a new file based on file size can be easily modified by a variable in the code.

Using a much lower threshold could prove useful as it would create many smaller data files instead of a few large ones. Only a file which is open is at risk of getting corrupted if power is removed during a write operation to the file. As the time spent writing to each file is reduced when the threshold value is reduced, less data will be corrupted should the instrument be exposed to a loss of power.

After successful file opening, a small loop routine reads the content of the BRAM into an array. When completed, the array is written to the file which is then closed. By first writing the BRAM content to the array, instead of directly to the file, speed is increased. This is due to the file writing command having a much larger overhead than writing to the array, and should therefore be used conservatively.

Testing during development proved up to 95 % speed increase when this method was applied.

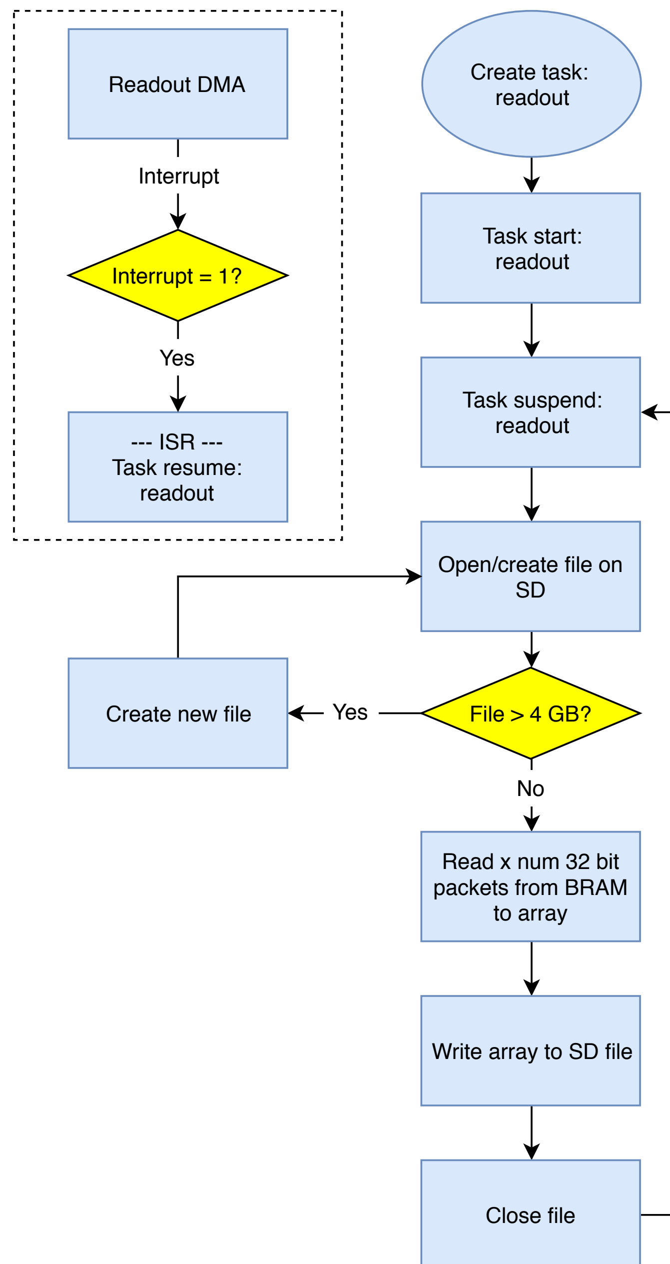

Figure 5.13: Readout task flow.

5.9 Test System for Synthetic Data

To verify the readout chain, a system has been added to send synthetic data into the firmware, and then operate the instrument as if the data was received from the ADC channels. Figure 5.14 illustrates the architecture developed in cooperation with Alexander Nikolai Nesse. Design of HDL architecture on the programmable logic is the work of Nesse and is further documented in [26]. Changing data input source is performed with a MUX.

When a *test*-command is initiated through the user interface, a software task called

synthetic_data_reader is created. This can be seen in figure 5.15 which illustrates the task behaviour. The task parses synthetic data from a file on the micro SD memory card into the FIFO in the programmable logic. When completed, the software sets a GPIO signal high to indicate completion. The task is then deleted.

The GPIO signal initiates data readout of the FIFO. The data is streamed to the MUX connected to the firmware.

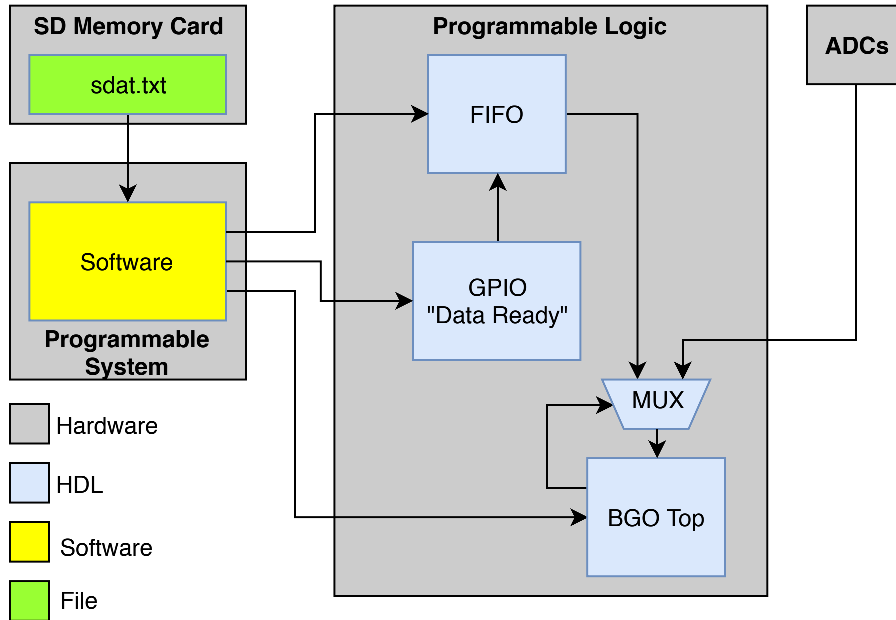

Figure 5.14: Data flow of the synthetic data read-in system.

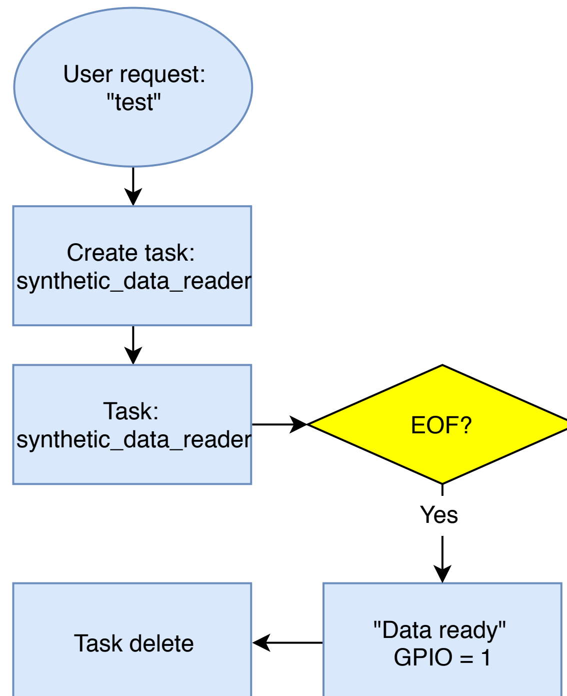

Figure 5.15: Behavioural flow of the synthetic data reader task.

When the synthetic reader task is created, it first configures the firmware to work

with the test system. This is done by instructing the firmware to switch the MUX input from the ADCs to the synthetic data stream through a firmware register.

Next, a file named *sdat.txt* containing synthetic data in the form of line by line integer values is opened from the micro SD memory card. The file is read line by line, and the read values are written to the FIFO. An option has been added to print the parsed data to the UART terminal for debugging purposes, known as *verbose mode*. It should be noted that parsing with the verbose mode activated is very slow due to the processing-intensive task of printing such a large amount of data to UART. Verbose mode is therefore by default set to *off*.

When the end of file has been reached, the file is closed, and the *data ready* GPIO signal is toggled. At completion, the task is deleted.

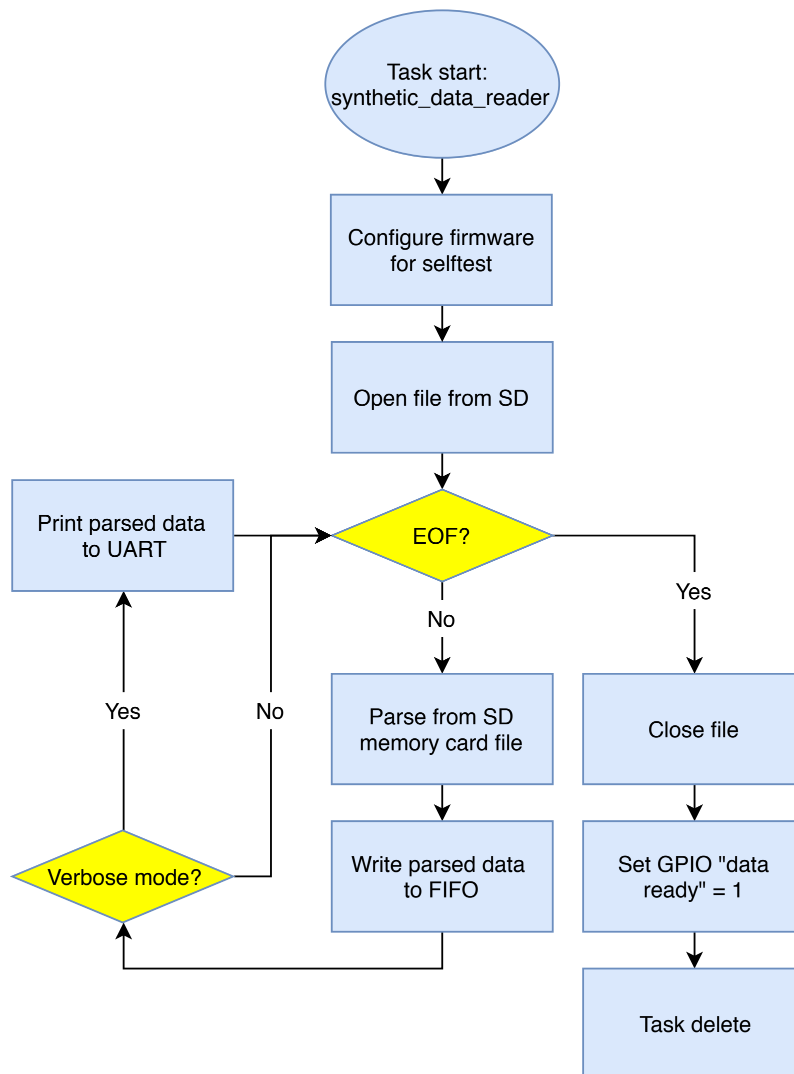

Figure 5.16: Flow of the synthetic data reader task.

5.10 Known Problems

The git repository found in appendix B contains an updated issue tracker, were access can be given by request if entitled. Two major issues not related to the code

itself, is however highlighted bellow.

Linker removes libraries after SDK reboot

The linker in Xilinx SDK 2017.4 has a tendency to remove the *-lxilffs* flag, used to include the xilinx fat file system library into the board support package. The flag is removed upon reboot of Xilinx SDK, and by re-exporting the bitstream from Xilinx Vivado. Currently the only solution is to edit the system.mms file in the board support package, by removing xilffs, compile, and then add it again. Identifying and then finding a solution to the problem required much time, and postponed much of the progress at the start of the project. [37]

Xilffs-library has string functions disabled by default

The standard inclusion of the xilffs-library comes with string functions disabled. As this library needs to be re-added every time Xilinx SDK is restarted (as described in section 5.10), it is important to remember to re-enable these functions. This is done by editing the definition of *FILE_SYSTEM_USE_STRFUNC* to equal to 2 in *xparameters.h* found in the include directory under the Board Support Package directory of the project.

Chapter 6

Testing

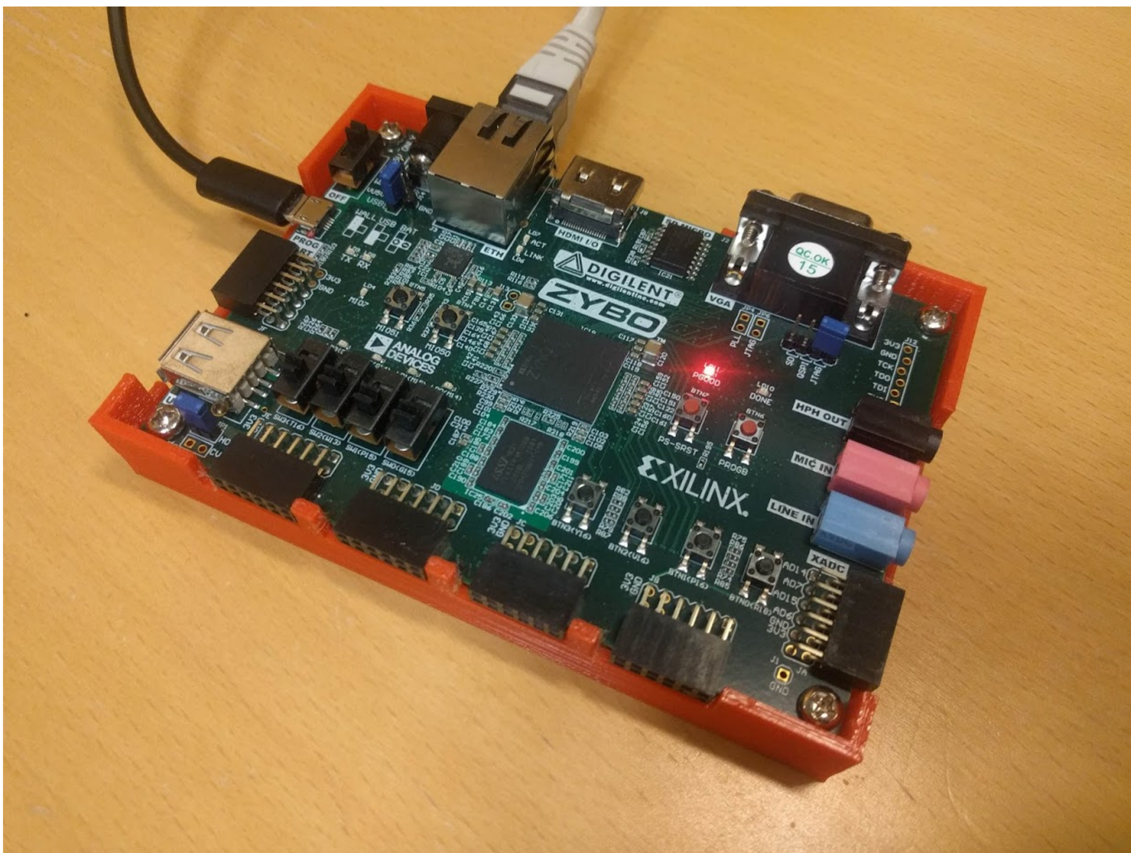

Figure 6.1: Test setup showing the Zybo board connected using a CAT6 ethernet, and USB cable.

Testing of the system was carried out at the University of Bergen. The test setup consisted of the Zybo trainer board connected to a computer running Linux. The board was connected through a CAT6 ethernet cable connected in apoint-to-point configuration, and a USB cable connected to the UART/JTAG port (port J11). Power was provided through the USB cable, and the micro SD memory card used was a 128 GB Samsung Evo Plus Micro-SDXC.

Although the different subsystems of the software have been tested during development, this chapter will present the tests used to verify that the objective of the

thesis as seen in section 1.3 has been fulfilled. The reason for not presenting the other tests is that their results are generally not so easily quantifiable. They are also of little interest seen from a overview and current system status perspective.

The presented tests are a *synthetic data read-in* test which verifies its usage as a test system, a *readout* test to verify the readout chain and that data is correctly stored on the micro SD memory card, and a test were the system is operated as a standalone system similar to how the finalized instrument will operate.

6.1 Prerequisites for the tests

6.1.1 Client-side Software

With the implemented solution, very little software is required on the client-side of this setup. The client computer runs a full-size desktop version of the Linux distro Ubuntu 16.04 LTS.

To properly connect the board to the computer without the use of a separate gateway hosting a DHCP server, the DHCP server service has been hosted on the client computer. This is easily done in Ubuntu by creating a new ethernet network connection in the network manager, and through the configuration set it to share the connection over IPv4. The shared connection IP address is by default within 10.42.0.x, with the computer acting as gateway located at IP address 10.42.0.1.

For connecting to the telnet user interface, the virtual package *telnet-client* of version 0.14-40 has been used. It features a telnet client command line interface software, running from a terminal window. *Telnet-client* comes bundled with Ubuntu 16.04 LTS, and the version is the latest release for the used version of Ubuntu as of August 2018.

Software for connecting to the UART interface was the text-based modem control program *Minicom*. The software is not default to Ubuntu, and version 2.7-1build1 var installed using the *Advanced Packaging Tool* from the default library sources. This version was the latest release for the used version of Ubuntu as of August 2018.

Test software

For triggering and monitoring of the integrated logic analyzer further described in section 6.1.2, Xilinx Vivado version 2018.1 was used on the client computer. Access to the analyzer is through the same USB port hosting the UART interface, and executing the analysis thus had to be performed from the same computer.

Programming both the programmable logic and programmable system of the Zybo was performed with Xilinx SDK version 2018.1. The same version of the software was used to generate the bootloader files using the tutorial from appendix F.

6.1.2 Integrated Logic Analyzer

A Xilinx IP-block called an Integrated Logic Analyzer has been added to the programmable logic. The module buffers signals a set number of clock periods with

probes connected to the different signal terminals of interest. The analyzer can be used to monitor signals both internally to the programmable logic. As the programmable system is embedded in the programmable logic, the signals going in and out of it can also be monitored. For the following tests, the sample data depth was set to 8192 clock cycles.

6.1.3 Synthetic data

To test the system, a file containing synthetic data samples was used with the test system documented in section 5.9. The file was created during the development of FEGS BGO to test the firmware, but has been modified to be correctly parsed by the software test system of ALOFT. The file contains 39600 12-bit values and thus has a range of 0 to 4095. The data simulates noise along with multiple TGF-events as outputted by the 12-bit ADCs. Figure 6.2 is a plot generated with a MATLAB script of the values found in the file.

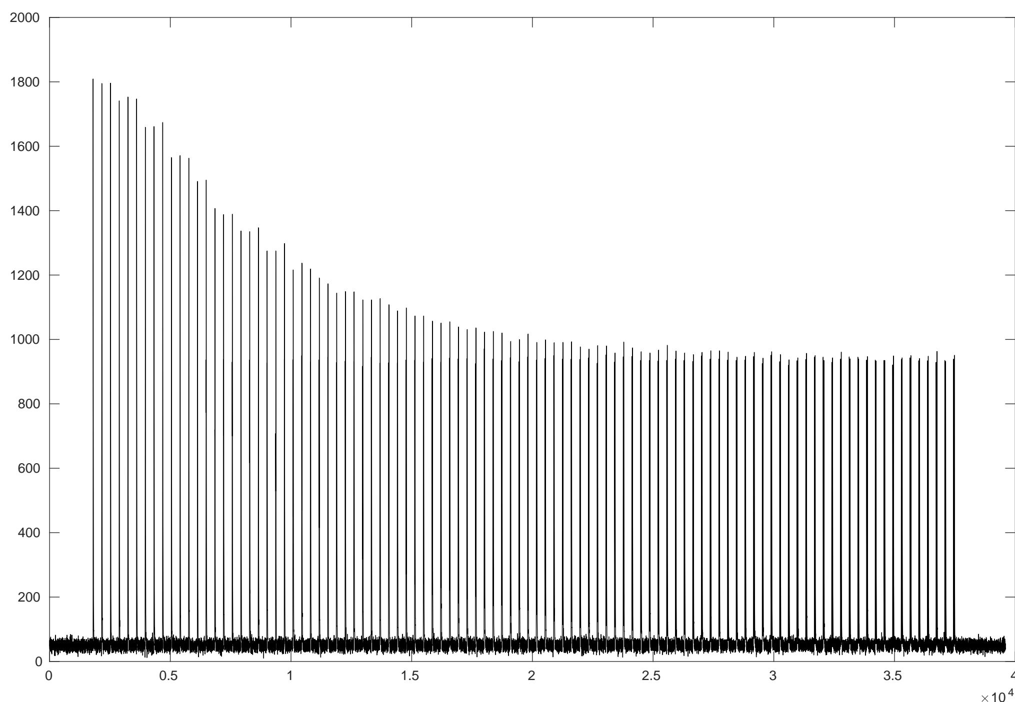

Figure 6.2: Synthetic data used with the test system in ALOFT to verify the readout chain. The Y axis displays the 12 bit ADC input value in range 0 to 4095, and X axis displays the sample number.

As can be seen from the plot, the around 2000 first sample points are just pure noise. It is, however, expected that the firmware will mark the first sample value above 0 as an event as it differs from the background noise measured to be 0 when started. After the first real event at around sample point 2000, multiple events follow with decreasingly lower values. Not easily seen in the plot, the events are also occuring more frequent later in the file in order to test the firmware for the *overflow* and *fast*

SCDPs. At the end of the file, the values goes back to noise only.

6.2 Synthetic data read-in

As testing the readout system is dependent on the test system described in section 5.9 which sends the synthetic data into the front of the readout chain, this was the first test performed.

Listing 3 displays the output seen in the UART interface when the *test*-command was issued through the telnet interface. As can be seen, the synthetic data reader task is started and put into the running state. Verbose mode was set to *OFF*. The task reaches the *end of file*, and thus completes. From this output it can also be seen that something has happened with the rest of the system, as the readout task has been triggered. This is further covered in section 6.3

```
102 :: Synthetic data: RUNNING
102 :: Synthetic data: Verbose mode OFF
102 :: Synthetic data: EOF
103 :: Synthetic data: DONE
150 :: Readout: Number of 32 bit packets written to SD: 15
152 :: Readout: Awaiting interrupt
```

Listing 3: Output from the UART interface after the *test*-command was issued through the telnet interface.

By the output in listing 3, it has been verified that the synthetic data reader task has been executed. Listing 4 shows the first ten values from the synthetic data file. It should be noted that in the file itself they are separated by the newline character. To verify that the test system works as intended, these values must be compared with what is actually being registered as being put into the firmware.

```
47 66 63 45 40 60 56 46 61 55
```

Listing 4: First 10 values written to the firmware from the file containing the synthetic data.

The intended behaviour of the test system was described in section 5.9, but in summary; The software writes the data from the file to a FIFO module on the programmable logic. When completed, the software asserts a GPIO pin to indicate its completion. When the GPIO pin is asserted, the FIFO writes its content to the firmware module *BGO TOP*.

Figure 6.3 shows the wave diagram output from the integrated logic analyzer when triggered by the GPIO pin signal being set high. The yellow line indicates the the location of the trigger point.

Figure 6.3: Wave diagram from the integrated logic analyzer, showing the 10 first data values being outputted from the FIFO when the GPIO signal is asserted.

Wave *axi_gpio_0_gpio_io_o* is from a probe connected to the GPIO pin. Wave *design_1_i/fifo_generator_0_dout* is the 12-bit data line output of the FIFO connected to the input of *BGO TOP*. By comparing the values outputted from the FIFO with the values of the synthetic data file in listing 4, it can be seen that these are the same. In addition, the data is not outputted from the FIFO before the GPIO pin is set high, as designed.

6.3 Test of Readout

As the primary goal of the instrument is to store data from the BGO detectors, testing of the readout chain has been the main priority in the testing phase. The following test was performed to see if the firmware interrupted the processor at the correct time, and if offloading was performed to the micro SD memory card. The results shown in this section is the result of the synthetic data test described in section 6.2. From listing 3 it could be seen that the readout was initiated.

Figure 6.4 is a screenshot of the wave diagram output from the integrated logic analyzer, configured to trigger on the interrupt signal transmitted by the DMA used for readout.

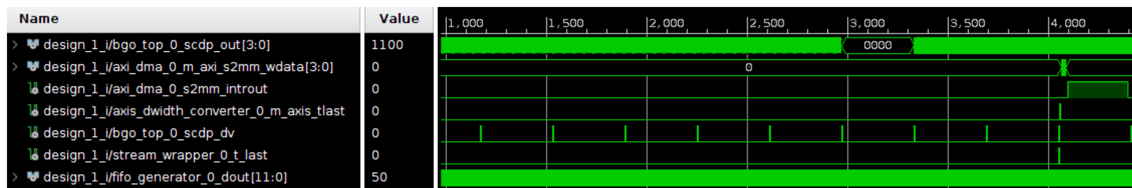

Figure 6.4: Wave diagram of data readout from the programmable logic.

In the wave diagram, data can be seen constantly flowing out of *bgo_top_0_scdp_out* which is the *BGO Top* module described in section 5.8. For each valid SCDP outputted, the data valid signal *bgo_top_0_scdp_dv* goes high. When 10 of these packages has been sent through the stream wrapper, the *t_last* signal *stream_wrapper_0_t_last* is asserted. This triggers the *t_last* signal *axis_dwidth_converter_0_m_axis_tlast* from the width wrapper, to signal the DMA to begin transferring data into the block memory. This may be easier seen in the wave diagram seen in figure 6.5. Figure 6.5 is a zoomed-in version of figure 6.4.

Only 9 of the data valid signals are shown in the first diagram, as the first event was detected when the firmware first received data. This was explained in section 6.1.3. The output of the DMA has the signal name *axi_dma_0_m_axi_s2mm_wdata* and can be seen starting and quickly completing data transfer when the *t_last* signal from the width converter is asserted.

When the DMA transfer has completed, it triggers the interrupt *axi_ dma_ 0_ s2mm _ inrout*. This verifies that the interrupt trigger of the programmable logic has been correctly set up.

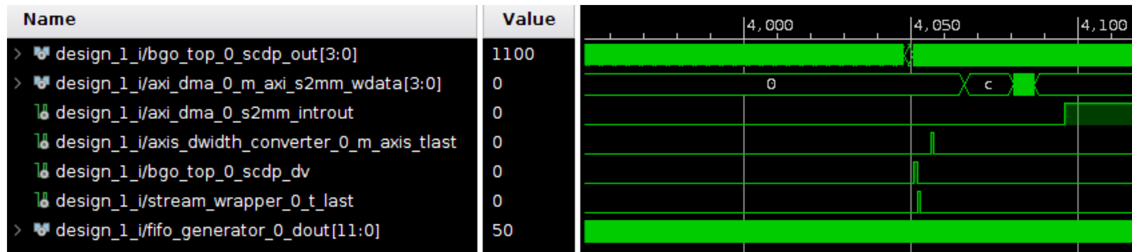

Figure 6.5: Wave

As described in section 5.8, the ISR should, if set up correctly, de-assert the interrupt signal. This can be seen happening at the end of figure 6.4. Listing 5 shows a snippet of the 36 first data bytes stored in the binary file on the memory card after running the test. The software does not have any functionality to tell SCDPs apart, so no newline character is inserted. Listing 5 has, however, been formatted so that each SCDP are on a separate line. The firmware also outputs the SCDPs in a reversed order. The listing has also been formatted to display the data in the correct order.

```
01 18 21 F1 8B 15
02 8E 00 01 8B 15
00 23 22 11 8B 1E
01 18 21 F1 8B 1F
02 8E 00 01 8B 1F
00 22 22 51 8B 28
```

Listing 5: Non-reversed SCDPs in hexadecimal format, separated by newline character.

By converting the hexadecimal numbers into binary numbers, the result is seen in table 6.1. These are the correctly formated SCDPs and can be compared with table 3.1 of the packet format from section 3.3.1.

Flag	Adr.	OVF	Fast	Valley	Spare	Fast t.t	Energy	Time tag
00	00 00	0	1	0	0	01 1000	0010 0001 1111	0001 1000 1011 0001 0101
00	00 00	1	0	1	0	00 1110	0000 0000 0000	0001 1000 1011 0001 0101
00	00 00	0	0	0	0	10 0011	0010 0010 0001	0001 1000 1011 0001 1110
00	00 00	0	1	0	0	01 1000	0010 0001 1111	0001 1000 1011 0001 1111
00	00 00	1	0	1	0	00 1110	0000 0000 0000	0001 1000 1011 0001 1111
00	00 00	0	0	0	0	10 0010	0010 0010 0101	0001 1000 1011 0010 1000
47 46	45 42	41	40	39	38	37 32	31 20	19 0

Table 6.1: SCDPs outputted from the readout test, and formatted to the normal SCDP format.

By comparing the SCDPs with the ones expected, they have been verified to be correct. The expected values was found by [26], by running an hardware description language testing workbench with the synthetic data on the firmware.

6.4 Test of System in Stand-Alone Mode

As the finalized system will be operated using only the telnet user interface, it was of interest to see if the system could be operated in a stand-alone configuration The setup of this test is somewhat different as the system is booted from the bootloader placed on the micro SD memory card. As the Zybo was programmed through JTAG and also powered through the same port for the other tests, it was decided to power it from a USB power bank to verify its independence in this test.

As with the finalized system, this also meant that the only way of monitoring and controlling the system was through the telnet interface. Figure 6.6 shows the test setup and execution.

Figure 6.6: Test setup showing the Zybo board connected using a CAT6 ethernet, powered by a battery pack.

The Zybo is powered off the USB power bank, and connected to the computer using a CAT6 ethernet cable. Seen in the picture, the computer was able to connect to the telnet user interface in the terminal window. Issuing the *test*-command through the interface was also verified as working. Afterwards, it was verified that the log file and data file had been created and contained the expected content.

It was noted that when booting the system using the bootloader instead of programming via JTAG, the time for the system to become ready was greatly reduced.

6.5 Summary

Synthetic Data Read-in

Through the synthetic data read-in test in section 6.2, it has been verified that the test system described in section 5.9 correctly parses the content of a file containing synthetic 12-bit data values, and writes them to the firmware.

Test of Readout

The readout test documented in section 6.3 verified that the data outputted from the firmware was correctly stored to the file on the micro SD memory card. It also verified that the current implemented functionality of the firmware by [26] works as intended.

Test of System in Stand-Alone Mode

Running the system off a battery bank, and booting the application from the bootloader on the micro SD memory card was performed in this test. The results verified that the system behaves as intended, and could be controlled through the telnet interface.

Chapter 7

Conclusion and Outlook

7.1 Conclusion

This thesis has described the specification, design and development of the software layer for the ALOFT readout system. The readout system has been developed on an embedded SoC platform containing both a programmable system, and programmable logic. Early in the project, it was found that implementing the software executed on the programmable system on top of the real-time operating system FreeRTOS would yield the most optimal solution due to the required software complexity. The developed software is capable of writing scientific data packages outputted from the firmware onto a micro SD memory card, and throughout testing indicates that the system is fast enough to not induce any bottlenecks. A single core processor has been demonstrated to prove sufficient for the current version of the software.

The firmware of the FEGS BGO instrument is currently being ported by another master student to run on the programmable logic used in ALOFT. Interfacing this firmware is performed through configuration and status registers. Automatic configuration of these registers from a configuration table on the micro SD memory card has been implemented in the software on startup. Afterwards the configuration is repeatedly performed with a 60 second interval to limit the impact of soft errors caused by the radiation environment ALOFT operates in. Based on calculated estimates of the radiation induced error-occurrence, this feature may not be needed, but has been kept for testing purposes. Error detection using a checksum algorithm has also been added to detect corruption of the configuration table.

Manually configuring the registers and reading the values of the status registers can be performed through the implemented user interface. The user interface is available through a telnet server, and therefore only an ethernet cable is needed to control the instrument. The user interface has two functions; Printing status messages, and receiving user input. Status messages are sent from a common centralized hub called the internal messaging system. All software subsystems can send status messages to this hub. This makes it very easy to add additional subsystems to the software at a later point. In addition, the messages are tagged with a unique identification code to help identify the type of message, and the subsystem of origin. The messages are also printed to any connected UART terminal, and logged to the micro SD memory

card for later access.

The fundamentals of all the software subsystems are in place, but some modification to the code may be required to make the finalised system behave exactly as intended. The designed architecture offers high modularity through the use of multithreading and functionality separation. As each task is to a large degree independent, removing, adding, and modifying tasks and functionality requires little effort. This is very useful to the further development of ALOFT, as the exact requirements are yet to be determined.

7.2 Further work

At this point, further testing of the software is needed. When the firmware has been successfully ported, a synthetic data test will be able to verify much of the system.

There is still work to do on the readout system of ALOFT. As has been described earlier, concurrent to the production of this thesis, Alexander Nikolai Nesse has worked on porting the firmware used on the FEGS BGO to the Zynq-7000 platform, and it will be described and documented in his thesis [26]. It is expected that his thesis will provide an updated status of the readout system, and possibly also provide request of additional requirements to the software. However, the following are short notes on suggested work and functionality that should be added to the system.

7.2.1 Using Multiple Storage Devices

The system developed in this thesis uses only one storage device; the micro SD memory card. Having all the files on one such device can be adequate in a test environment, but may require some redesigning in the future. It is not expected that the finished instrument will be using the Zybo board, but the one used may very well have micro SD memory card capability. It would, however, probably be better to distribute the files to different devices. This would be to avoid accidents were the bootloader or configuration file is accidentally modified or deleted when, for example, making the storage device ready for a new flight by deleting the accumulated data and log files. These devices would probably also benefit from being USB flash drives instead, as they are generally much more robust and easier handled than the tiny micro SD memory cards.

7.2.2 Improve Checksum Algorithm

The current checksum algorithm uses summation to calculate the total value of the configuration file. If one of the characters changes, the total value changes.

One problem with this simple checksum algorithm is, that if multiple characters changes, the total value may still be equal to the value calculated for the uncorrupted file. It is also in-sensitive to the order of the characters changing. It is suggested that the implemented checksum algorithm is either improved on these weaknesses, or swapped for other checksum algorithms which addresses these weaknesses.

7.2.3 Packet Format Compatibility with the ER-2 Instrument Network and Data Reduction

The packets transmitted by the telnet user interface must be verified to be compatible with the ER-2 Instrument Network. The maximum packet size is normally 550 bytes, but can be increased by request if the Inmarsat link is used. In general, this means that packet size will not be a problem with the currently implemented system. Packet frequency may, however, be a problem, as there is a rate-limit of one packet every ten seconds over the Iridium link. This can be increased on the Inmarsat link, but in general may require the user interface packets to be buffered before being sent over the network.

For better efficiency, both within the system and when using the user interface, a redesign of the message format may be valuable. The current system transmits both the system message code in addition to the message in plain text. Efficiency would be very much improved by only transmitting the code, and then have standalone software on the client side translating the codes to plain text.

7.2.4 Data offloading over TCP/IP

For the FEGS BGO instrument, the solution for offloading data was to remove the flash drive the data was stored on, and insert it in a separate computer for offloading. The flash drive was connected to the FEGS BGO using a USB extension cable which was part of the harness for instrument access. This harness provided easy access to offloading, as physical access to the instrument is limited.

Currently for ALOFT, the data is stored on a micro SD storage device which is mounted underneath the Zybo development board. Although the Zybo development board will not be the actual board used, micro SD mounting is typically similar. Offloading data in such a system requires direct physical access to the instrument, and other solutions should be looked at. The proposed solution is to implement large quantity data transfer over TCP/IP with the ethernet interface. The interface supports gigabit transfer speeds, and thus the current limiting factor would be the read rate of the storage device which typically is found between 30-100 MB/s. It is suggested to look at ways to implement the *File Transfer Protocol* to host an FTP server on the Zybo. Such a solution would require the use of TCP instead of UDP as used by the current EGSE-connection. This would be to ensure the offloading data does not get corrupted while being transferred.

7.2.5 Graphical User Interface

Interfacing with ALOFT is done through the telnet server. The user interface provides a minimal way of monitoring the system and perform operations through a series of commands. For easier readability of system monitoring and performing command operations, the development of a simple GUI is suggested. A GUI would provide a more intuitive way of monitoring the instrument as for instance graphs could be added, and commands could be issued by typing in forms and pressing virtual buttons.

A GUI could be implemented in the form of a web-server or a standalone program running on the EGSE. A solution using a web-server would need implementation on the instrument's programmable system. This would increase the processing load the system handles, but it would remove the need for special software on the EGSE. A standalone GUI software-solution running on the EGSE could interpret the messages coming from the existing telnet server, and convert virtual button-presses in the GUI software to predefined commands already present in the telnet user interface.

7.2.6 Limitations of the Z-7010

One limitation that caused some problems during development was the limited number of look-up tables and BRAM-cells available on the Z-7010. Although most problematic to the development on the programmable logic, integrating the programmable system into it required good insight into the signalling happening on the programmable logic. As was seen in section 6.1.2, this was handled using an integrated logic analyzer. Adding additional probes and longer sample periods requires a large amount of look-up tables. As the resource limit was easily reached, the solution was to use only a limited number of probes at a time. This was time consuming, as changes to the analyzer requires the bitstream to be regenerated, a process that takes between 15 and 20 minutes on the computer used.

The synthetic data read-in system suffered from the limited number of BRAM-cells on the chip. The reason was that the FIFO could not be set deeper than 16384 elements, meaning that writing the synthetic data file in the test performed in section 6.2 had to be performed in three sequences for it to process the whole file. To speed up development time, it would be beneficial to use a larger device to overcome these debugging problems, although the Z-7010 seems large enough for the finalized system.

Appendices

Appendix A

Coding Style

This project follows the *Linux Kernel coding style*, with the exception of using four-spaced indentation instead of eight-spaced. Guidelines for the use of C language in critical systems by MISRA have also been adopted. Maximum code line width is set according to the ISO/ANSI screen size of 80 characters except when the code contains text that will be printed to the screen. In such case, the text printed must follow the ISO/ANSI standard with a max length of 80 characters. [38]

Naming convention Naming convention for C-code in the project has been to use snake case due to its increased readability over camel case. This means that compound words and phrases are separated with an underscore, so that camel case becomes camel_case. This is different from camel case that would become camelCase. Characters should also be all lowercase. Named instances such as variables, functions, etc. that are declared globally should always have a descriptive name. Those that are only declared locally are allowed with a somewhat less descriptive name, but should not decrease the maintainability of the code.

Handles and queues should always end with *_handle* or *_queue* to decrease the chance of using the wrong item when referencing.

Appendix B

Git Repository - Directory Tree

The following is a directory tree of the git repository containing the system developed in this thesis. The directory tree only shows files and folders relevant to the work presented in this thesis.

The repository can be pulled from the University of Bergen's gitlab, if access has been granted:

```
https://git.app.uib.no/master/ALOFT_embedded_solution_repo.git
```

```
ALOFT_embedded_solution_repo
└ aloft_embedded
  ├ FSBL (bootloader versions located here)
  │ └ 6_9_2018
  │   ├ BOOT.bin
  │   └ output.bif
  ├ Programmable Logic
  │ └ aloft_v0_2
  │   ├ aloft_v0_2.sdk
  │   │ ├ aloft_v0_2_ps
  │   │ │ └ src (application files located here)
  │   │ └ aloft_v0_2_ps_bsp (BSP sources located here)
  │   └ aloft_v0_2.xpr (project file for Xilinx Vivado)
  └ synthetic_simulation
    └ sdat.txt
```

Appendix C

Methodology and Software Overview

This appendix contains a summary of the methodologies applied when developing the software side of the ALOFT readout system. The source to the following sections can be found in [21].

C.1 Hierarchical Design

The software developed in this thesis has been designed with the goal of having an hierarchical design. This "divide and conquer"-strategy means that the software has been divided into separate modules until each functionality can easily be comprehended in a separate module. This "modularity" is evident by using tasks in FreeRTOS (modules), and keep each task function in separated files. The total system complexity has in this way been decreased, as the functionality of a module may be used by other different modules, instead of having to write and maintain multiples of specialized code sections which can only be used by one functionality. This is called *regularity* and a prime example in ALOFT is the reuse of the code used to write messages to the *internal message system*.

Modularity also comes with the benefit of being a nice aid when debugging. As each module should ultimately only have one functionality, the module itself can be tested as a standalone system. As it is also easier to understand which functionality a specific module provides, determining the origin of an anomaly in a complete system is also generally easier.

C.2 Debugging and Version Control

Debugging

As the main part of this thesis has been software development and architectural design, debugging has been the number one most time-consuming task. When an anomaly to the intended behaviour has been found, the routine has been to first identify what intended behaviour was expected. The next step has been to postulate an hypothesis for what might be the cause of the anomaly, and ultimately test the hypothesis to make the necessary changes.

Version Control

Version control has been handled using *GIT*. The repository can be found linked in appendix B.

Appendix D

Operating manual

The following is a simple manual for operating the ALOFT through the telnet interface.

Before Boot

Insert the micro SD memory card with the bootloader into the Micro SD-slot found on the backside of the Zybo. The card must also contain the file with the synthetic data called *sdata.txt*. The file containing the configuration parameters called *config.txt* is not required. All files must placed in the root directory. A 10/100/1000BASE-T cable must also be connected between the EGSE computer and the Zybo. The computer should have a running DHCP server.

Boot

Apply power to the Zybo, and turn it on using the onboard switch. After about 10 seconds, the telnet server should be available. The time may vary between the usage of DHCP and Static IP allocation.

Accessing Telnet

From a EGSE computer running Ubuntu, the telnet server can be accessed using the following CLI command:

```
telnet <IP>
```

The IP address can be found using a host discovery tool such as Nmap, or if no DHCP-server is running, access can be made with the default IP address:

```
telnet 10.42.0.2
```

D.0.1 Reading binary data post flight

Reading large binary files can be done with ease by Python. First, navigate to the directory containing a DATAx.BIN file obtained from ALOFT. Then issue the following command in CLI to launch python:

```
python
```

After launch, read the file as binary data:

```
>>> file0 = open('DATAx.BIN', 'rb')
```

The content can then be put into an array for easier handling:

```
>>> array0 = file0.read()
```

This can take some time, especially if the file is still on the memory card. The speed is, however, most likely limited to the maximum read rate of the memory card used, depending on the computer used. Upon completion, the two following commands can be issued to check the number of array elements (returning just over 500 million elements in the example below), and the content of the array. Note: reading the whole array is time consuming unless a specific range has been specified using brackets.

```
>>> len(array0)
544768000
```

```
>>> array0
```

And last, range specified readout reading the first 1024 elements of the array:

```
>>> array0[0:1023]
```

It is advised to use Python for future data handling due to high data handling rates. It also comprises a large toolbox of advanced data processing tools through libraries.

Appendix E

Command Sheet & System Message Codes

E.1 Command Sheet

Table E.1 is a preliminary list of all available commands with description to operate ALOFT from a telnet client. Characters "<" and ">" displays the placement of user parameters. To illustrate, writing the value 0xFFFFF to register address 0x50000000 can be issued by the Memory Write Command (MWC) by typing the following into an active telnet connection and press enter:

```
mwc 0x50000000 0xFFFFF
```

Any offset to the address must be included in the address parameter.

Command	**Description**
test	Does a full readout using synthetic ADC-data taken from a file on the SD-card.
mrc <address>	Reads the value of the specified register address
mrc -a	Lists all registers with their respective value.
mwc <address> <value>	Writes <value> to register <address>.
mwc -a	Rewrites all registers from the default configuration file on SD.

Table E.1: Available commands in the ALOFT Telnet user interface.

E.2 System Message Codes

System Message Code	**Type**	**Description**
0xx	OTHER	Used for display formatting and similar uses
1xx	INFO	Information message
2xx	WARNING	Indicating that something might be wrong
3xx	ERROR	Something in the system has failed

Table E.2: Types of system message codes implemented in the software.

System Message Code	**Subsystem**
x0x	Task.
x1x	Queues.
x2x	SD memory card.
x3x	Telnet server.
x4x	Command Interpretation System.
x5x	Readout.
x6x	Checksum Generator.
x7x	Firmware Configurator.
x8x	Synthetic Data Reader.

Table E.3: System Message Code-identity for different subsystems.

Code	Type	Subsystem	Description
102	INFO	Tasks	The named task is now running.
103	INFO	Tasks	The named task has been stopped or has completed.
110	INFO	Queue	The named queue has successfully been cleared.
120	INFO	SD Card	Memory card has been mounted.
121	INFO	SD Card	End of file has been reached.
130	INFO	Telnet	DHCP request success.
131	INFO	Telnet	System has been configured with the default IP. See table 5.5.
132	INFO	Telnet	A client has connected to the telnet server.
150	INFO	Readout	Displays the number of scientific data packages that has been written to SD.
151	INFO	Readout	A new datafile has been created on the SD-card.
152	INFO	Readout	The task awaits an interrupt from the ISR.
160	INFO	Cks. Gen	Calculation of the checksum has completed.
170	INFO	FW Config	Firmware version number.
230	WARNING	Telnet	DHCP is not activated.
231	WARNING	Telnet	An unknown command was received.
232	WARNING	Telnet	A client has disconnected from the telnet server.
250	WARNING	Readout	The maximum individual file size allowed by the system has been reached.
310	ERROR	Queue	The named queue was unsuccessfully cleared.
320	ERROR	SD Card	Failed to mount the SD card.
321	ERROR	SD Card	Could not open file in the specified task.
322	ERROR	SD Card	Could not seek in file in the specified task.
323	ERROR	SD Card	Could not read from file in the specified task.
324	ERROR	SD Card	Could not write to file in the specified task.
325	ERROR	SD Card	Could not close file in the specified task.
330	ERROR	Telnet	Error adding N/W interface.
331	ERROR	Telnet	DHCP request timed out.
332	ERROR	Telnet	Error reading from socket. Socket will be closed.

Table E.4: Description of System Message Codes.

Appendix F

Tutorials

The following is a list of tutorials created during the project. The tutorials have been published on the internal wiki page of the University of Bergen.

F.1 FreeRTOS Bootloader generation in Xilinx SDK

For tutorial on generating a bootloader for FreeRTOS in Xilinx SDK, see [39].

F.2 Example project with AXI4 Lite peripheral on the Zynq-7000

Tutorial on creating an example project with an AXI4 Lite peripheral on the Zynq-7000 can be found in [40]. The peripheral is then accessed from the ARM A9 micro processor.

Bibliography

[1] R. Helliwell, "Whistlers and related ionospheric phenomena," 1965. doi: 10.1029/GL012i006p00393

[2] J. Dwyer, D. Smith, and S. Cummer, "High-energy atmospheric physics: Terrestrial gamma-ray flashes and related phenomena," *Space Science Reviews*, vol. 173, 2012. doi: 10.1007/s11214-012-9894-0

[3] G. Fishman, P. Bhat, R. Mallozzi, J. Horack, T. Koshut, C. Kouveliotou, G. Pedleton, C. Meegan, R. Wilson, W. Paciesas, S. Goodman, and H. Christian, "Discovery of intense gamma-ray flashes of atmospheric origin," 1994. doi: 10.1126/science.264.5163.1313

[4] M. McCarthy and G. Parks, "Further observations of x-rays inside thunderstorms," *Geophysical Research Letters*, vol. 12, No. 6, pp. 393–396, 1985. doi: 10.1029/GL012i006p00393

[5] H. Cember and T. Johnson, *Introduction to Health Physics.* The McGraw-Hill Companies, Inc., 2009. ISBN 978-0-07-164323-8 Fourth Edition.

[6] R. Feynman, R. Leighton, and M. Sands, *The Feynman lectures on Physics, Mainly Mechanics, Radiation, and Heat.* Addison-Wesley Publishing Company, 1977. ISBN 0-201-02010-6-H Sixth printing.

[7] R. P. M. van der Boog, "Energy calibration procedure of a pixel detector," 2013, bachelor thesis of applied physics.

[8] M. Marisaldi, F. Fuschino, C. Labanti, M. Galli, and F. Longo, "Detection of terrestrial gamma ray flashes up to 40 MeV by the AGILE satellite," *Journal of Geophysical Research*, vol. 115, 2010. doi: 10.1029/2009JA014502

[9] S. Dyer, *Wiley Survey of Instrumentation and Measurement.* Kansas State University, 2001. ISBN 978-0-471-22165-4

[10] ASIM.dk, "ASIM payload," http://asim.dk/payload.php, payload overview page, Accessed: 03-07-2018.

[11] N. Østgaard, "ASIM – challenges and possibilities," 2010, presentation from AGILE workshop.

[12] Lockheed Martin, "The U-2S Dragon Lady - product page," https://lockheedmartin.com/us/products/u2.html, accessed: 03-07-2018.

[13] T. Thorsteinsen, “PHYS231 kompendium,” 1995, summary from the course "Physics of Radiation".

[14] Z. Gibson, A. Nagata, M. Morikawa, T. Sakai, T. Shimizu, and Y. Takahashi, “High altitude dependence of ionizing radiation from cosmic rays,” 2016, poster.

[15] R. Schrimpf and D. Fleetwood, “Selected topics in electronics and systems - radiation effects and soft errors in integrated circuits and electronic devices,” vol. 34, 2010.

[16] A. Taber and E. Normand, “Single event upsets in aircraft avionics,” vol. 40, No. 2, 1993.

[17] P. Goldhagen, “Overview of aircraft radiation exposure and recent ER-2 measurements,” 2000. doi: 0017-9078/00/0

[18] N. Østgaard, H. Christian, J. Grove, and M. Quick, “Gamma-ray observations at 20 km altitude above thunderstorms,” *Geophysical Research Abstracts*, vol. 20, 2017.

[19] K. Label, “Radiation effects on electronics 101: Simple concepts and new challenges,” 2004.

[20] M. Silvestri, S. Gerardin, F. Faccio, and A. Paccagnella, “Single event gate rupture in 130-nm CMOS transistor arrays subjected to x-ray irradiation,” 2009.

[21] N. Weste and D. Harris, *CMOS VLSI Design - A Circuits and Systems Perspective.* Addison-Wesley, 2011, vol. Fourth Edition. ISBN 978-0-321-54774-3

[22] G. Bruni, “Temperature effects on soft error rate due to atmospheric neutrons on 28 nm FPGAs,” 2014, master thesis.

[23] K. Ullaland, “FEGS BGO design report,” 2016.

[24] Trimble, “Copernicus II GPS receiver,” https://cdn.sparkfun.com/datasheets/Sensors/GPS/63530-10_Rev-B_Manual_Copernicus-II.pdf, reference Manual. Accessed: 23-08-2018.

[25] Xilinx, “Opal Kelly XEM6001 - product page,” https://www.opalkelly.com/products/xem6001/, accessed: 03-07-2018.

[26] A. Nesse, *Electronic Readout SoC Design for Measurement of Gamma Radiation (preliminary title).* University of Bergen, 2018, unpublished as of 11.09.2018.

[27] J. Pan, “Software reliability,” *Dependable Embedded Systems*, 1999.

[28] Xilinx, “Digilent zybo SoC trainer board product page,” https://www.xilinx.com/products/boards-and-kits/1-4azfte.html, accessed: 03-07-2018.

[29] ——, “Zynq-7000 All Programmable SoC Technical Reference Manual,” 2015, version 1.12.1.

[30] Digilent, “ZYBO FPGA Board Reference Manual,” 2016.

[31] FreeRTOS, “Coding standard and style guide,” https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html, accessed: 03-07-2018.

[32] R. Barry, "Mastering the FreeRTOS real time kernel, a hands-on tutorial guide," 2016, pre-release.

[33] MISRA consortium, "MISRA C:2012, guidelines for the use of the C language in critical systems," 2013.

[34] SD Association, "Speed class," https://www.sdcard.org/developers/overview/speed_class/index.html, overview of the SD Standard. Accessed: 03-07-2018.

[35] NASA Airborne Sensor Facility, "ER-2 instrument network communication notes," https://asapdata.arc.nasa.gov/sensors/ER2_Comms_Guide.pdf, instrument network in the ER-2. Accessed: 23-08-2018.

[36] Dryden Flight Research Center, "ER-2 airborne laboratory experimenter handbook," 2002.

[37] forums.xilinx.com, "Linker-problem-with-xilffs," https://forums.xilinx.com/t5/Embedded-Development-Tools/Linker-problem-with-xilffs/td-p/788267, accessed: 03-07-2018.

[38] The Linux Kernel, "Linux kernel coding style," https://www.kernel.org/doc/html/v4.10/process/coding-style.html, accessed: 05-06-2018.

[39] M. Heigre, "Tutorial, FreeRTOS FSBL," https://wiki.uib.no/ift/index.php/FreeRTOS_FSBL, tutorial on generating FSBL-files in Xilinx SDK. Accessed: 23-08-2018.

[40] ——, "Tutorial, Creating example project with AXI4 Lite peripheral in Xilinx Vivado," https://wiki.uib.no/ift/index.php/Creating_example_project_with_AXI4_Lite_peripheral_in_Xilinx_Vivado, tutorial on creating an AXI4 Lite peripheral and access it with the Zynq-7000 ARM A9 µP. Accessed: 23-08-2018.