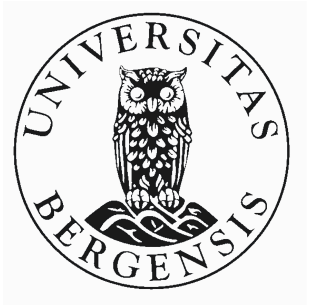
Development of Electronic Ground System Equipment for the ASIM Project

Thomas Bjørnsen

Master Thesis



Department of Physics and Technology
University of Bergen

June 2013

Abstract

ASIM (Atmosphere Space Interaction Monitor) is an instrument designed to monitor X-ray and gamma-ray bursts of terrestrial origin. ASIM is divided into two parts; Modular X-ray and Gamma-ray Sensor detect photons in the 15 keV to 20 MeV range, and Modular Multi-spectral Imaging Array takes images in the visible range. It will be mounted on the International Space Station (ISS) where it will be observing the earth's atmosphere.

The instrument is composed of two detector layers, each consisting of four identical units. The outermost layer is a low energy detector detecting photons ranging from 15 keV to 400 keV. This layer is made out of Cadmium Zinc Telluride (CZT) crystals and structured as a pixlated array. Each unit consists of 4096 pixels, giving the MXGS a total of 16384 pixels. The innermost layer is a high energy detector detecting photons ranging from 200 keV to 20 MeV. This layer is made out of Bismuth Germanate Oxide (BGO) crystals, and each unit consists of 3 BGO crystals and 3 photomultiplying tubes.

University of Bergen (UoB) is responsible for the development of both detector layers. The project is entering the final stage where the detector units are tested. Some test equipment and computer software is required to carry out these tests.

This thesis describes the development of tools needed to test the detectors. A primary objective of this work was to develop the equipment with scalability and reusability in mind. Several parts of the equipment can be adapted for later projects, thus reducing the time required to develop future testing equipment.

The test equipment is divided into two parts, and a computer program commands, monitors and gathers data from the detectors. All 4096 pixels of a CZT detector can be controlled individually by using a graphical user interface or through a script. The second part of the equipment is a physical interface, allowing a computer to communicate with the detectors. This is realized with a commercial FPGA based interface and with an embedded microcontroller supplying USB protocol. The primary objective is to convert from USB signals to the xlink protocol utilized by the detectors, but it also includes a buffer for temporary data storage.

The system has been subject of thorough testing, including both validation and verification tests and performance tests. It controls, monitors and acquires data from up to four detectors simultaneously. A working proof of concept existed a few weeks after project start, and the system has been used for several months.

Acknowledgements

The work described in this thesis was carried out at the department of physics and technology at University of Bergen.

First of all, I would like to thank my supervisor, Kjetil Ullaland. His guidance, help and feedback has been of great help throughout my work. A lot of credits also go to the chief engineers of the department, specifically Georgi Genov, Shiming Yang, Arne Olav Solberg, Maja Rostad and Dominik Fehlker, who have helped and assisted me.

Thanks also to my fellow students, Vegard Holsen and Augusta Pithalice, whom both contributed to this project. I would also like to thank my father, Willy, Anne Katrine and Alice, all of whom read and suggested improvements to this thesis. Thanks to UoB class of 09 and the microelectronics group for all the good time we shared.

Finally, I would like to thank the rest of my family for their support, and Vinni for your love and patience.

Contents

1 Introduction

1.1 History

Discovering gamma-ray bursts

During the early years of the cold war, nuclear bombs were detonated in high altitudes. A nuclear detonation produces radioactive particles that fall to the ground and drift with currents. Radioactivity is hazardous for living organisms, and detonating nuclear bombs in the atmosphere result in an increase in the collective dose of a population. Currents move the radioactive particles far from the explosion, thus people all over the world suffer from nuclear tests above the surface.

In 1963, a treaty was formed, prohibiting nuclear detonations in air, water and outer space. To monitor it was obeyed, USA built a series of satellites, "Vela", to detect x-rays, gamma-rays and neutrons. A nuclear detonation has a recognizable fingerprint, but the satellites also detected radiation that was not a nuclear detonation. Later satellites in the series had a time resolution that was good enough to calculate the origin of the source, and the non-detonation events they picked up were found to have a cosmic origin [1].

Discovering terrestrial gamma-ray flashes

To investigate gamma-ray bursts further, a new satellite, BATSE, was built. BATSE had detectors on every corner, so the direction of the gamma-source could be calculated. In 1994, the first observation of Terrestrial Gamma-ray Flashes (TGF) was detected and since then, data from several other satellites have been used in search of TGFs.

Only a few TGF events were detected by BATSE, but later satellites with better hardware detected many times more. Analyzing data from existing instruments

and compensating for poor time resolution, data suggests that TGFs are much more common than previously assumed [2].

Atmosphere Space Interaction Monitor (ASIM) project

ASIM is an instrument that will be mounted onto the Columbus module of the International Space Station (ISS) where it is going to observe the earth's atmosphere. It is composed of two instruments, Modular X-ray and Gamma-ray Sensor (MXGS) and Modular Multi-spectral Imaging Array (MMIA).

MXGS will be capable of detecting photons in both the x-ray and gamma-ray region. When such photons are detected, information connected with the event is transmitted to ground for analysis, along with images in the visible range taken by the MMIA instrument. For the first time scientists can systematically correlate TGFs with images in the visible range, which gives valuable input to their research [3]. University of Bergen (UoB) is not involved in the development of MMIA thus it will not be discussed further in this text.

1.2 About this work

My involvement in the ASIM project has been to manufacture Electronic Ground System Equipment (EGSE), which is hardware and software required to test detectors developed at the UoB. I have designed, developed, implemented and tested software and hardware, as well as modified existing tools. To some extent I have also been using the developed tools to test and debug the detectors.

1.2.1 EGSE

EGSE is a set of tools used through out the development. Developing hardware is much more time consuming than developing software. If the hardware allows it, test procedures should be developed in software. The development process is faster, it is cheaper, it is easier to expand and modify. It is also faster to fix bugs in software compared to hardware. Often advanced features from third party software can be exploited as well.

Existing implementation

Prior to this project, former students had already developed an EGSE platform. The main test procedures are implemented in software, and to communicate with the detectors, a hardware interface is used. This interface is realized with a programmable electric circuit manufactured by Opal Kelly Inc. Its primary job is to convert commands coming from the computer into something detectors understand, and vice-versa.

The EGSE software was developed in a programming language called LabVIEW. LabVIEW is not text based and it is not procedural or object oriented, like most text based programming languages. Rather it is a visual programming environment based on data flow. Data is processed by sending the input values through a set of "virtual instruments".

The Opal Kelly platform does not officially support LabVIEW, but the driver is shipped as a Microsoft Windows Dynamic Link Library (DLL). LabVIEW has native support of DLLs, thus it was still possible to use LabVIEW for the purpose of EGSE. However, there were a lot of stability issues in LabVIEW resulting from the driver.

No people with insight in the internal workings of the existing implementation remained at UoB. It is a large and complex program, and even minor changes could render the program useless. Finally, it was not designed with scalability in mind. So adding features or modifying the program would be both difficult and time consuming.

Because of these limitations and challenges it was decided that the software should be rewritten from scratch.

New implementation

My part was to rewrite the software in a text based programming language. One of the key elements was that it should be designed with scaling in mind from the start. The new implementation should also have scripting possibilities.

In the finished ASIM project, eight detectors are connected to a "Data Processing Unit" (DPU). The DPU has the overall control of the system, and during the development process, the EGSE takes the place of the DPU. Hence, the software is abbreviated as "DPU Emulator" and the hardware interface as "DPU Emulator Interface".

DPU Emulator is a computer program that commands, monitors and acquires data from the detectors. It is developed in a programming language called C#, and consists of a Graphical User Interface (GUI) of six windows. A simple script language is included, allowing for advanced commanding of detectors. All pixels of a pixelated detector can be configured manually through the GUI, or with commands in a script.

DPU Emulator Interface is a physical interface between a computer and detector. It can route packages to and from four detectors at the same time. I made only minor architectural changes to the hardware compared to the existing version.

The physical interface is implemented using an Opal Kelly XEM3001, which is a USB connected circuit board incorporating a Field Programmable Grid Array (FPGA). The DPU Emulator Interface is described in chapter 3.2, and the pinout is documented in table B.1 and B.2 as well as in figure B.1.

1.2.2 Structural Thermal Module (STM) Test

In the process of qualifying the ASIM instrument for space, several tests are performed. One of them is a thermal vacuum test of an STM version of the instrument. The thermal vacuum tests took place in January of 2013 at the facilities of Instituto Nacional de Téchnica Aeroespacial in Spain.

A small part of my work was related to this test. I was involved in developing tools, both software and hardware, used to read out temperatures during this test. I also participated in commissioning the developed tools.

2 Background

This chapter gives background theory for the preceding chapters. Chapter 2.1 starts by giving an introduction to the Modular Multi-spectral Imaging Array (MXGS) instrument and the constituents that is important for this text. That includes an overview of the two detector types of the MXGS, and in addition, some technical terms used throughout this text are explained.

An overview of the Opal Kelly XEM3001, which is used as the DPU Emulator Interface, follows in chapter 2.2. It gives a brief introduction of what it is and its capabilities. Some terms that Opal Kelly uses are also introduced and explained. Chapter 2.3 gives an introduction to development strategies, and some basic theory of computer programming.

2.1 Atmosphere Space Interaction Monitor

ASIM is designed to detect photons ranging from 15 keV to 20 MeV. It has a time resolution of 1 µs, and a relative time accuracy better than 10 µs. One of the detector layers is composed of 16384 pixels, which is used to locate the source of the photons.

Development of ASIM is a multinational collaboration project funded by European Space Agency (ESA) and lead by Terma, a Danish company. UoB is responsible for the BGO and CZT detectors, and Technical University of Denmark is responsible for the Data Processing Unit (DPU). Institutions in Spain and Poland are also involved in the ASIM project.

Unless stated otherwise, figures and tables in this chapter are from internal design documents, [3, 4, 5].

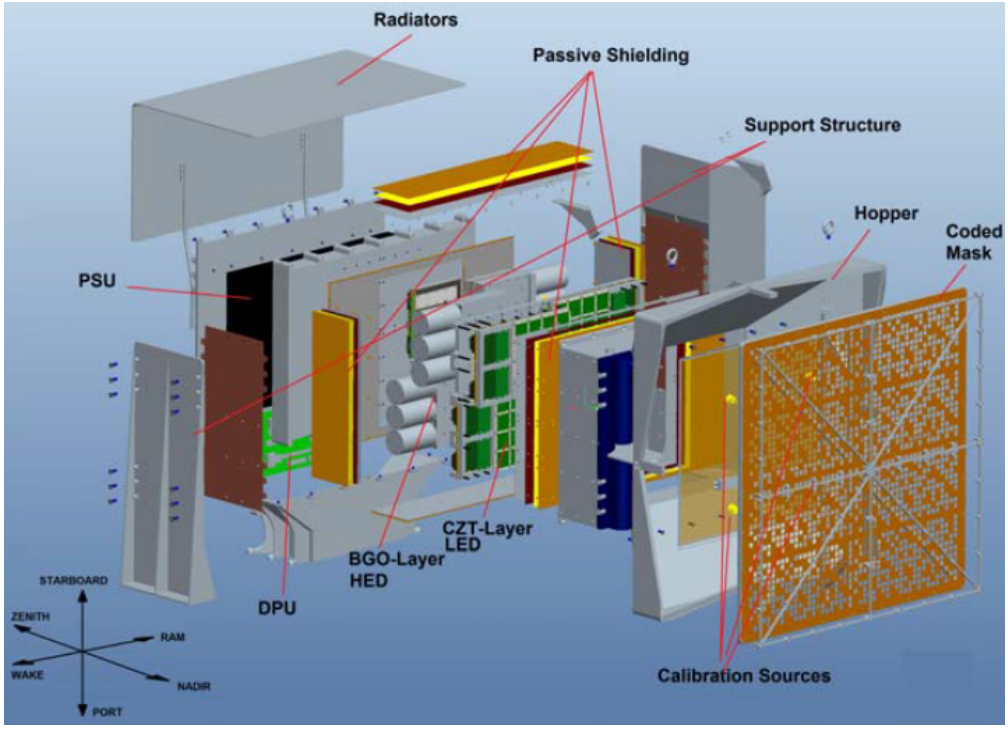


Figure 2.1: Exploded view of MXGS [3]

2.1.1 Structural overview

The MXGS is composed of several parts, and an overview can be seen in figure 2.1. Two different detector layers are stacked on the MXGS. The outer layer is a Low Energy Detector (LED) made of a Cadmium Zinc Telluride (CZT) alloy. The inner layer is a High Energy Detector (HED) made of Bismuth Germanate Oxide (BGO) scintillators. Each layer consists of four Detector Assembly Units (DAUs), so the complete system is composed of four BGO DAUs and four CZT DAUs.

DAUs are stand alone units that convert photons into digital data. First, photons are converted into analog electrical signals, where the electrical signal is proportional to the deposited energy. Next, the electrical signal is amplified, shaped and digitized, before it is sent to the DPU as a Science Data Packet (SCDP).

2.1.2 BGO

A BGO DAU has a detection efficiency of more than 60% for energies above 1 MeV. Peaks with a separation of approximately 500 ns can be distinguished, and it can

handle burst rates of 650 cts/ms.

Three BGO bars are mounted on every BGO DAU, and a photomultiplier tube (PMT) is attached to each bar. When a high energy photon hits a BGO bar, some, or all, of the energy is deposited. The absorbed energy is then reemitted as several less energetic photons. The PMT converts the emitted photons into electrical impulses, which is then amplified and digitized before they are sent to the FPGA for processing. When the FPGA detects the ADC peak value (Analog Digital Converter), a SCDP is generated and transmitted to the DPU.

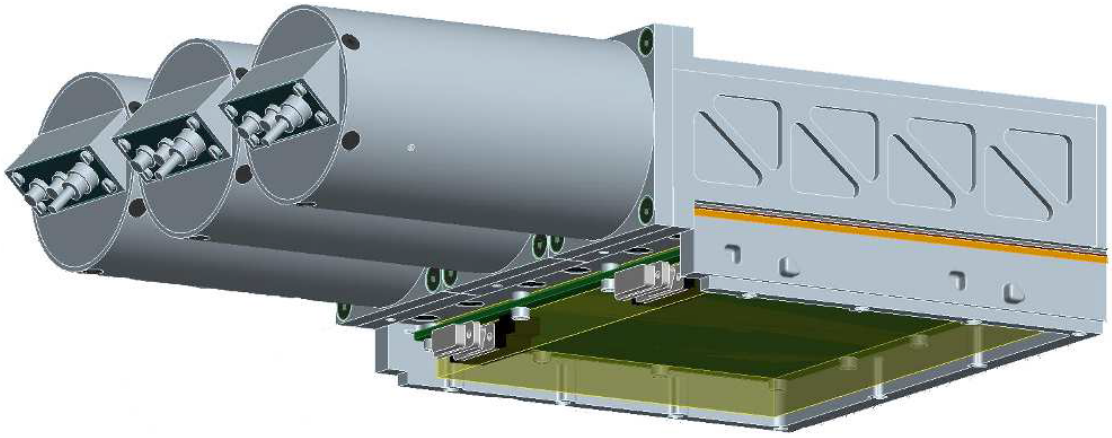


Figure 2.2: A BGO DAU with three BGO photomultiplier tubes

2.1.3 CZT

A CZT DAU is built as a pixelated array, divided into a 4x4 grid of "Detector Modules". A detector module is a 16x16 grid of pixels, thus giving a CZT DAU a total of 4096 pixels.

Four detector modules constitute a read out chain, as figure 2.5 shows, and share read out electronic. Each read out chain can separate hits down to approximately 4 μs. Because a read out chain share read out electronics, multiple hits occurring within this period is reported as multihit. Two bits are used for multihit, thus up to four multihits are reported per read out chain.

Outside the LED layer, there is a coded mask that attenuates photons and cast a "shadow" onto the detectors. Knowing the pattern of the coded mask and observing the shadow, the origin of the particles is calculated [6].

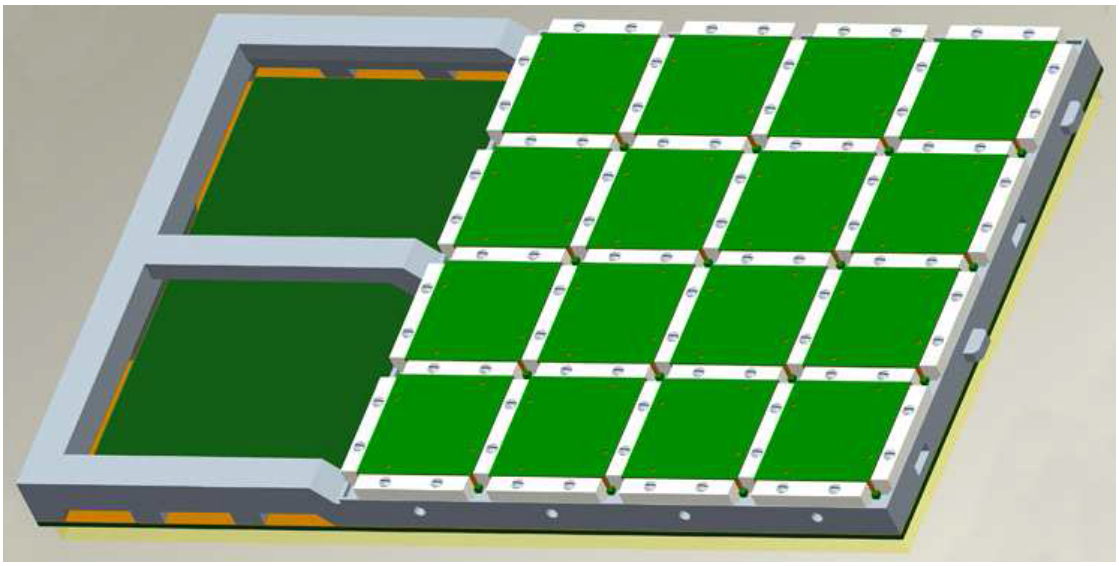


Figure 2.3: A CZT DAU with a 4x4 grid of detector modules

XA1.82 Integrated Circuit

To detect and process particle hits, an Application Specific Integrated Circuit (ASIC) is used. The ASIC in use is Gamma Medica Ideas "XA1.82", and is abbreviated as XA-ASIC. It has 128 analog input channels, each having a preamplifier, shaper and peak-hold. The energy output is a differential analog signal proportional to the energy of the particle.

An XA-ASIC has a volatile shift register where several settings are shifted in. Among the settings is a coarse and fine tuning of trigger threshold that is common for all pixels. It also has per channel fine tuning and the possibility to disable individual pixels.

To configure an XA-ASIC, 858 configuration bits are clocked into a pad named "RegIn". When clocking in a new configuration, the old configuration is clocked out on a pad named "RegOut". This allows for a daisy chain connection of several XA-ASICs, limited to 512 units [7].

Read out chains

Four detector modules constitute a read out chain, and share a bus interface to an FPGA. Figure 2.4 gives an overview of the read out chains and detector modules.

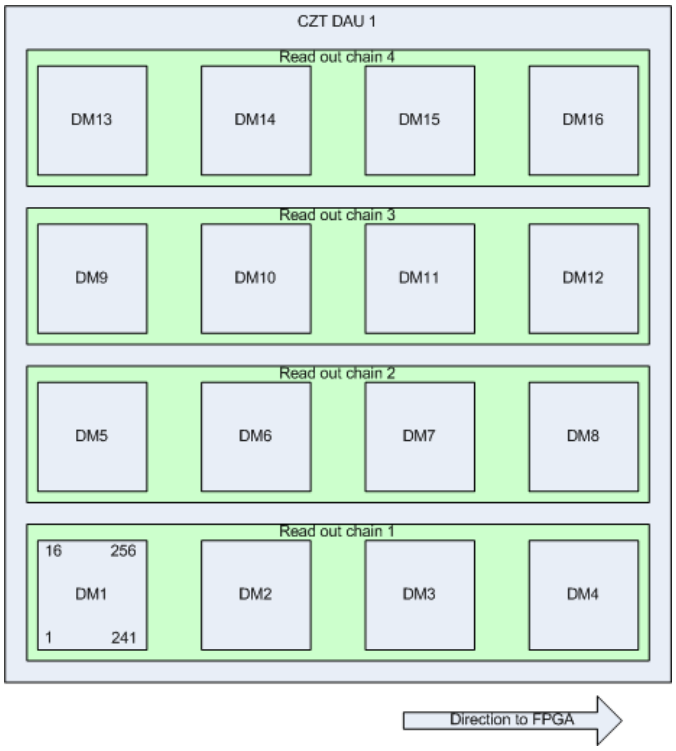


Figure 2.4: Organization of detector modules and read out chains of a CZT DAU

Daisy chain

A detector module carries 256 pixels and is equipped with two XA-ASICs. The eight XA-ASICs of a read out chain are connected in a daisy chain, where "RegOut" of one is connected to "RegIn" of the next. Figure 2.5 shows a diagram of the daisy chain connection, and table B.5 gives an overview of all 858 bits of the configuration register. The first bit in the table refers to the first bit in the shift register, thus it should be the last bit shifted in.

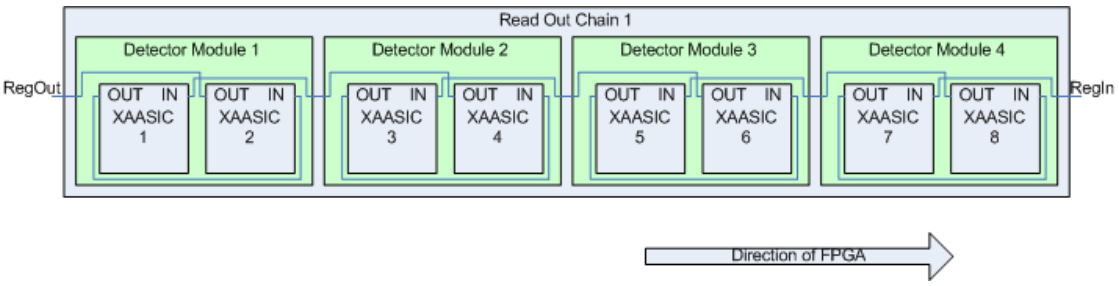


Figure 2.5: The order of the XA-ASICs in the daisy chain

2.1.4 Data Processing Unit (DPU)

The DPU is the data processing unit of the MXGS system. It has an FPGA with an embedded processor that is used to run software. The software is saved on a non-volatile EEPROM memory, and can be patched from ground if necessary. It also has 1 GB bulk memory where packages are temporarily stored.

The DPU is system master, and it commands, and monitors all eight DAUs. In addition, SCDPs from all eight DAUs is received, filtered and stored. Because of a limited downlink from ISS, only 140 MB is allowed transmitted per 24 hours. Therefore the DPU has an algorithm that decides whether or not an event is scheduled for downlink to a ground station [3].

2.1.4.1 Communication with DAUs

DPU and DAU communicate via a protocol called "xlink". Xlink is a Low Voltage Differential Signal (LVDS) with data strobe encoding.

Data strobe encoding

In data strobe signalling there are two signals, data and strobe. Exactly one of the signals changes every clock cycle, as can be seen in figure 2.6. The figure also shows that the clock is encoded within the data, and that XOR between data and strobe regenerates the clock. The receiver use the regenerated clock to sample the data, hence the system tolerate a skew of one bit time [8] between data and strobe. Another advantage is that it is very speed tolerant as long as it does not exceed the capabilities of the receiver. The sampled data is synchronized by the receiver, which allows sender and receiver to operate at different frequencies.

Xlink protocol

An xlink transmitter can send two different lengths. A "Long" word consists of 48 bits and a "Short" word consists of 24 bits. The length of the word is only transmitted indirectly, where the two first bits in a word indicate whether it is a long or short word. Sending a 24 bit word which is flagged as a 48 bit words, or vice versa, results in an error at the receiving end.

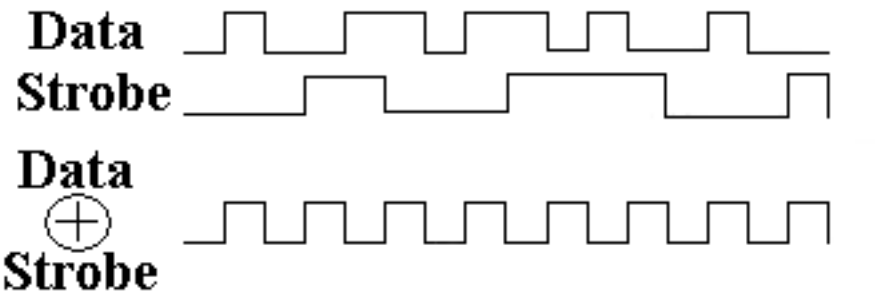


Figure 2.6: XOR between data and strobe regenerates the clock

Data packages

As discussed, the xlink protocol has two flagbits that indicate what sort of data it is. Two bits allow four different packages, where '00' imply a 48 bit packet, and all other implies a 24 bit word. The four packets, with flagbits, are summarized in table 2.1.

Flag bits	Packet type
00	**Science Data Packet** A particle event from a DAU is sent as a 48 bit word. It is composed of address, energy, timing information, and some BGO and CZT specific bits.
01	**Memory Dump Data Packet** Respons of a memory read command is sent as a 24 bit memory dump data packet containing 1 byte of data.
10	**Memory Read Command** All registers of a DAU can be read back by sending a 24 bit memory read command (MRC). When a DAU receives a MRC, it replies with a Memory Dump Data packet containing one byte of data.
11	**Memory Write Command** To command a DAU, the DPU sends a 24 bit memory write command (MWC). When a DAU receives a MWC, the data is written to the addressed register, and the command sent back to the DPU to acknowledge it was received.

Table 2.1: Summary of data packets available in the xlink protocol

2.2 Opal Kelly XEM3001

Opal Kelly produces devices with programmable FPGAs and onboard microcontrollers. The microcontroller is used to control the Phase-Locked Loop (PLL), and also works as a USB interface between the FPGA and computer. The USB communication is made transparent to the users by a software platform called "FrontPanel".

2.2.1 Input and output types

XEM3001 is highly flexible as it provides many virtual signals that can be set or read from a computer. The virtual signals are accessible through the FrontPanel driver, giving computer software easy interaction with the FPGA firmware. The names of signals are relative to the Opal Kelly XEM3001, so "In"-signals go into the FPGA, and "Out"-signals go out of the FPGA. In addition, it has 90 general purpose pins that may be used to connect peripherals.

There are three virtual I/O methods that serve three different purposes. XEM3001 has 32 inputs and 32 outputs of each type, as shown in table 2.2. For Wires and Triggers, each input and output is subdivided into 16 bits, giving a total of 512 signals of each type and each direction.

Endpoint type	**Address range**	**Sync/Async**	**Data type**
Wire In	0x00 - 0x1F	Asynchronous	Signal state
Wire Out	0x20 - 0x3F	Asynchronous	Signal state
Trigger In	0x40 - 0x5F	Synchronous	One-shot
Trigger Out	0x60 - 0x7F	Synchronous	One-shot
Pipe In	0x80 - 0x9F	Synchronous	Multi-byte transfer
Pipe Out	0xA0 - 0xBF	Synchronous	Multi-byte transfer

Table 2.2: Addresses of the various signals of the FrontPanel driver. Some of the different input and output properties are also listed [9].

Wires

WireIns and WireOuts can be regarded as regular level signals in a system. When a computer makes an API call to update WireIns or WireOuts, all Wires of the given direction are updated simultaneously.

Triggers

When a Trigger Out is triggered it is stored in a register until a computer makes the API call to update the TriggerOuts. When this happens, all TriggerOuts is transferred to the computer, and all bits that have triggered since the last API call are logical '1'. If it has not been triggered since the last API call, it is logical '0'. TriggerIns are updated and transferred to the FPGA individually as they occur, and is logical '1' for one clock period.

Pipes

The last I/O type is PipeIns and PipeOuts. Pipes are not subdivided into individual bits. They are meant for bulk data transfer, and can transfer data at a significant higher rate than Wires and Triggers.

Pipes are subdivided into two categories; ordinary Pipes and Block-Throttled Pipes. Block-Throttled Pipes differ from ordinary Pipes in that they only transfer data of the exact size of the configured block size. This gives an increased performance for small block sizes because of a quicker USB negotiation process [9].

2.3 Development strategies

2.3.1 Abstraction

Developing complex systems, whether it is hardware, software or other disciplines, abstraction is essential to success and is often applied at several levels. The contractee describes the desired end product, including description, main functionality and performance demands. The description is analyzed by the contractor, who divide the product into functionally separate blocks, often referred to as "modules".

A module has an interface between the inside and outside, and a description of how it is used. What is inside is irrelevant so long as it behaves as expected. A contractor may then become a contractee and delegate each module to subcontractors. The process of dividing complex systems into a set of modules is repeated until one reaches a comprehensible level of complexity.

The design process explained start at the top and work its way down, thus a top-down approach. Realization of the design is a bottom-up process, where the most basic modules are developed, integrated together to form a new module. This process then continue until the system is completed [10].

2.3.2 Reusing

There is a philosophy in UNIX stating that a program should "do one thing and do it well", and have a well defined interface of how it is used [11]. Practicing this philosophy, a program can be replaced at a later timer time, granted that it has the same interface.

A principle closely related to abstraction, is reuse. When designing a system using the top-down approach one should try to recognize modules that are systems in their own right. If a module is general enough, it may be used in other projects. There are several advantages of modularization, where increased productivity being important. Existing modules that are used in other projects also have higher quality, because they are already tested, both at a module level and system level, thus most errors are sorted out [12].

There are also drawbacks associated with reusing modules. The consequence of an error in a frequently used module is higher compared to a module used in a single project. As a result, the development process involves more planning and testing, and therefore a short term increase of expenses and production time [13].

Levels of reuse

Software distinguishes between three different types of reuse. Copying code from an existing project is the most commonly used method, but it is also the one with the least benefits. Customizing the copied code to the new project is also more likely to introduce bugs.

The second and third type includes a central library of finished modules. An obvious advantage is that bug fixes applied in the central library more easily integrates to projects utilizing the module. The difference between the two is that one type includes the module as source code, and the other link to a precompiled binary of the library.

In the hardware industry it is common practice to check if there exist a commercial product, either a component or intellectual property, that satisfy the requirements. If a product exists, this product is preferred to developing an equivalent product

in-house. The software industry exploits commercial alternatives to a less degree than the hardware industry [14].

Several factors contribute to the reluctance of reusing existing solutions. At a management level it is difficult to get funding and it is hard to administer a proper library. At a personal level, designers believe that their abilities exceeded that of previous designers, they want to innovate or they enjoy the challenge of programming rather than integrating an existing module [15, 14]. Software houses with the correct focus may reuse as much as 80% of the code, with the direct result of an overall productivity increase of 50-200% [13].

2.3.3 Programming

Hardware Description Language

FPGAs, being electrical circuits, are inherently different from CPUs. Developing FPGA firmware is, however, very similar to creating software to a CPU. A set of statements is written in a text file, and a program compile these statements into something that either an FPGA or a CPU understand.

Hardware Description Language (HDL) is used to describe the electric circuit of an FPGA firmware, where verilog and VHDL are the most common languages. An electric circuit has many concurrent operations that are performed simultaneously, thus statements in HDL may be written in any order.

Firmware written in HDL is modularised into "entities". An entity has a set of inputs and outputs, and several instances of an entity may be instantiated within a project. When converting code into firmware, the instances are substituted with the electric circuit described in the entity.

Procedural programming

A CPU takes a set of inputs, perform the requested operation and return the result. Once the previous operation is finished, it performs the next, which is best described in a sequential language. The programming syntax of a procedural language may be similar to a HDL language, but unlike HDL, statements are executed sequentially.

In procedural programming, frequently used code sequences are put in their own procedures and referenced when needed. This is often called "functions" in procedural languages and "methods" in object oriented programming. Essentially it

divides a program into several procedures that does one thing and does it well, thus reuse at a very low level. Contrary to HDL, a procedure is not substituted when the source code is converted to a program, thus the size of the end product is smaller when functions are used.

Object Oriented Programming

Today, many programs are developed to provide a service that is not sequential in nature. As an example, a graphical program is not used sequentially, thus better described with objects and events, e.g. "button number 4 was clicked", instead of sequences.

When developing object oriented, one treies to recognize objects and transform them into "classes", which are object oriented programming equivalent of modules. A class is an abstract definition of an object, and it may have properties, methods and events. For instance, a button has a size, location, text, and a mouse click event.

A class may also inherit from a parent class. One might have a "car"-class that has properties like width, length, weight, manufacturer, model and size of fuel tank. A hybrid car has all of those, but in addition it also has a battery. One could then create a "hybrid car" class that inherit all properties from the car class, but it has a capacity of battery property in addition to size of fuel tank. This solves a problem where a library is often very close to what is needed, except from a slight difference.

.NET is a platform on Microsoft Windows that provides a huge library of classes that is ready to use. Developers are relieved of programming basic classes, like buttons and textboxes, and may in stead focus on connecting them together to form new programs [16].

Application Programming Interface (API)

An API is an interface specification of a library. It includes a set of instructions that is available to external use, what input it expects and what output is expected. Often a functional description of the library and the individual instructions are also included.

2.3.4 Verification and validation

Throughout development, the product should be validated and verified. They are often distinguished as "are we building the right product" and "are we building the product right", for validation and verification respectively [17]. At the fundamental level it is possible to mathematically prove source code to be correct, but this is very difficult. An easier approach is to test a module by giving a certain input and compare the observed result with the expected result.

Testing hardware and software are broad subjects, each with many test strategies. A simple strategy is to do bottom-up testing, where a fundamental module is thoroughly tested before it is integrated into the next higher layer of abstraction. Once this layer is completed, an integration test is performed on the finished module. When this module is tested, it is integrated in a yet higher level, hence the process is repeated until the system is finished.

Below follows a summary of complications to keep in mind while developing and testing.

Hardware

Even though a module is proved to be correct, hardware operates in the physical world and is susceptible physical variances. Adjacent wires that are to close can contribute with "cross-talk" noise, affecting the voltage enough that it is interpreted wrong. Voltages also propagate at a limited speed, which may result in timing errors. Only a few combinations may result in the wrong behavior, making it very difficult to test hardware circuits [10].

Software

Software developers can in most cases expect hardware to work correctly. This eases the testing, but object oriented programming has another complication.

A class consist of many methods, some of which are "private" or "protected", meaning that they cannot be accessed from an object. These methods are accessed indirectly through "public" methods that make sure private methods have valid input. This gives developers good control of how a library is used, and it increases the robustness. For example, a protected method may contain a bug, but as long as the public method handles the input the bug never appears.

However, when a sub-class inherits from the parent class it does not face the same restriction. A sub-class is permitted to access protected methods that are not accessible from the API. Consequently, sub-classes may introduce bugs in parent classes, because it can bypass the code that handles unexpected input.

3 Electronic Ground System Equipment

EGSE is a set of tools used to test and characterize DAUs (Detector Assembly Units) that are in the final phase of development. A DPU Emulator Interface is a physical interface that allows a computer to communicate with DAUs through a USB connection. The DPU Emulator is computer program that commands and monitors as well as acquires data from DAUs through the communication interface. Consult chapter 1.2 for a more thorough review of the EGSE.

This chapter starts by discussing the planning phase of the new implementation. Chapter 3.2 gives a brief overview of the DPU Emulator Interface and how it was modified. Chapter 3.3 gives an overview of the constituent windows of the DPU Emulator, and what they are for. The DPU Emulator was subject to four tests to measure its performance. How it was tested, and the performance results, are discussed in chapter 3.4. Based on the test results, some changes are proposed in chapter 3.5, if higher performance should be needed at a later stage.

3.1 Planning EGSE

3.1.1 Discussion

Some requirements of the finished EGSE were that it

- Uses an Opal Kelly XEM3001 as interface
- Is capable of high speed data acquisition and low speed commanding
- Has a script based commanding language
- Is developed for scalability
- Is easy to modify and expand at a later time

Additionally, an ideal EGSE can be adapted to future projects, not necessarily related to ASIM. Adaption should be possible as user configurable settings, and not in source code.

3.1.2 Planning

The primary requirements were prioritized, but some measures was taken in an attempt to make it reusable for future projects. It was important that the most basic features were implemented first, so the new DPU Emulator gradually replaced the old. Throughout development, scalability was a key element that affected much of the code.

Work flow

The work flow is structured around the version control tool "git". Each project is developed in its own repository, and four repositories are used in total. The EGSE is divided in two repositories, one for DPU Emulator and one for DPU Emulator Interface. In addition, the STM software and benchmark program have their own repositories.

In total, around 15000 lines of code exist for all projects combined, thus to extensive to include in the thesis. The source code of all projects are published in the svn repository belonging to the ASIM project.

Feature priorities

First priority was to let the new implementation replace the old when it came to commanding DAUs. Registers should be referenced with names instead of numerical addresses. Data can be assigned to named bit fields, and unassigned bit fields keep their default value.

Second priority was to acquire SCDPs (Science Data Packets), but without data analysis at this stage. Data may be written as two files, where one is optional. The primary files contain data separated in the DAUs respective data fields. These files are processed by other programs, and saved as ASCII decimal files. The optional files have ASCII binary data and are used for human debugging.

Third priority was the possibility to configure CZT read out chains. Configuring a read out chain is an intricate and complex process. It is important to let the source code be tidy so it is easy to understand, debug and correct.

Support of four DAUs, extra script functionality and data analysis was planned to be added if time allowed it.

Addresses

From the EGSE point of view, four sets of addresses exist, but many electrical systems can be perceived using the same convention. The DPU Emulator Interface is a hub of several devices, and a device address is used to route packages to the correct device. Attached devices are sub-divided into modules. A module has a set of registers, each with its own address. The register may then be written to and/or read from. Finally, a register can be sub-divided into smaller chunks of bits.

A register address can be composed in two different ways. It can be split into two parts, where one range of bits is dedicated to module address and the other range to register address. Alternatively, it can be the algebraic sum of the register and module address. Granted that module address and register address are next to each other in the bit string, split addresses can be regarded as a subset of a algebraic addresses. Left shifting either register address or module address before summing the addresses, split address is achieved.

The EGSE implements algebraic addresses. By manually calculating the left or right shift of the addresses, it may be used in other projects if split addresses are used. Finally, it should handle unforeseen differences in other projects. This is realized by employing exceptions at every level.

Scripting

Scripts are used to send commands to detectors, and three sets of addresses are used per command. The first address specify what DAU to target, the next specify the module and the last specify the register. It should be possible, though not limited to, use names instead of numerical addresses for all of the above. Advantages include increased productivity, flexibility and readability, while at the same time reduce likelihood of bugs.

A register can be sub-divided into several fields, each with a name and default value. It should also be possible to assign a new value to parts of the register, and leave the rest with the default value.

The scripting language is a procedural language. This allows the script to have advanced functionality, and it can easily be expanded with extra features. Ideally,

a script can

- Have variables
- Perform arithmetic operations
- Process if-checks and for/while loops
- Call sub-procedures
- Read and write to files
- Acquire packets.
- Access FrontPanel

In the special case of MXGS, it should also be possible to configure XA-ASICs. Ideally, a script is able to enable and disable pixels, and gather data. Furthermore, it should process the gathered data and automatically determine an optimal threshold setting.

Configuring XA-ASICs

The finished EGSE should be able to configure a CZT DAU, both from GUI and script. The GUI layout of the window configuring CZTs is comparable to an actual CZT DAU, but for practical reasons it is limited to showing one read out chain at a time.

A CZT consists of 16 detector modules, each with two XA-ASICs. An XA-ASIC reads out hits from 128 pixels, all of which can be configured independently. In addition, each XA-ASIC has almost 30 settings affecting all pixels.

Each of the XA-ASIC configuration require an object in C#, and with 30 settings per XA-ASIC and 8 XA-ASICs per read out chain, this adds up to 240 objects. Including the 1024 pixels each read out chain has and multiplying by four read out chains, over 5000 objects are required. Designing a window with more than 5000 objects is impractical with the drag-and-drop method of Visual C# Express. So before starting to code, some questions are asked.

1. What would be the best strategy to construct a window with over 5000 objects?
2. Would it be possible to change the layout without doing the same change for all 16 detector modules?
3. Could the number of read out chains, detector modules and XA-ASICs be configurable?
4. Could it be implemented such that it would be readable and modifiable by others?
5. What would be the best strategy to save and load the configuration?

6. What is the best strategy to transform the configuration into DAU commands?

To answer the questions satisfactory, a good approach was to let the window be generated by code. With regularity and recursion, all of the mentioned challenges can be addressed properly.

Four detector modules constitute a read out chain. When the program is started, a procedure generates a window with one tab for every read out chain. For every tab created, a sub-procedure adds four detector modules, and so forth. This regular approach makes sure that the same layout applies to all detector modules.

To save the state of the window, recursion is utilized. The recursive method takes any window control as argument. If the control being processed is an input box, a checkbox etc., its value is saved to an XML-file. If the current control is a container, it calls itself, referencing this container. This way, the save procedure works its way recursively trough all controls in the window.

3.1.3 Test methodology

The program is split into several classes. Newly implemented features were tested once completed. When the behavior was verified, a layer of robustness was added so unexpected input is handled properly. Finally, the program was subject to more tests to check the behavior of unexpected input.

When the first stage was close to complete and well tested, output-pins of the DPU Emulator Interface was probed with an oscilloscope. Hand calculated patterns were compared with patterns emerging on the oscilloscope to further verify the behavior. In certain cases, it was compared with the LabVIEW implementation to see whether they were consistent with each other.

A DAU acknowledges a memory write command by relaying it back to the DPU. Therefore a loopback, where the output and input pins are short circuited, gives the expected response. This can be exploited to test that commands are sent and received reliably. In addition to the possibility of an external loopback, a test mode in the DPU Emulator Interface allows for an internal loopback. Once it was established that the program behaved as it should, it was tested with real DAUs before it was put in production.

Testing of the CZT configuration process is more complex, and is discussed in chapter 4.

3.2 DPU Emulator Interface implementation

3.2.1 Structure

The internal structure of the DPU Emulator Interface is divided into six logic blocks

1. Transmitter logic
2. Receiver logic
3. Data Packet Router
4. FIFO
5. Test logic
6. Sync generator

Structurally, only the sync generator is new compared to the old design. A graphical representation is shown in figure 3.1. For a table of signals and connections, reference appendix B.1 and B.2.

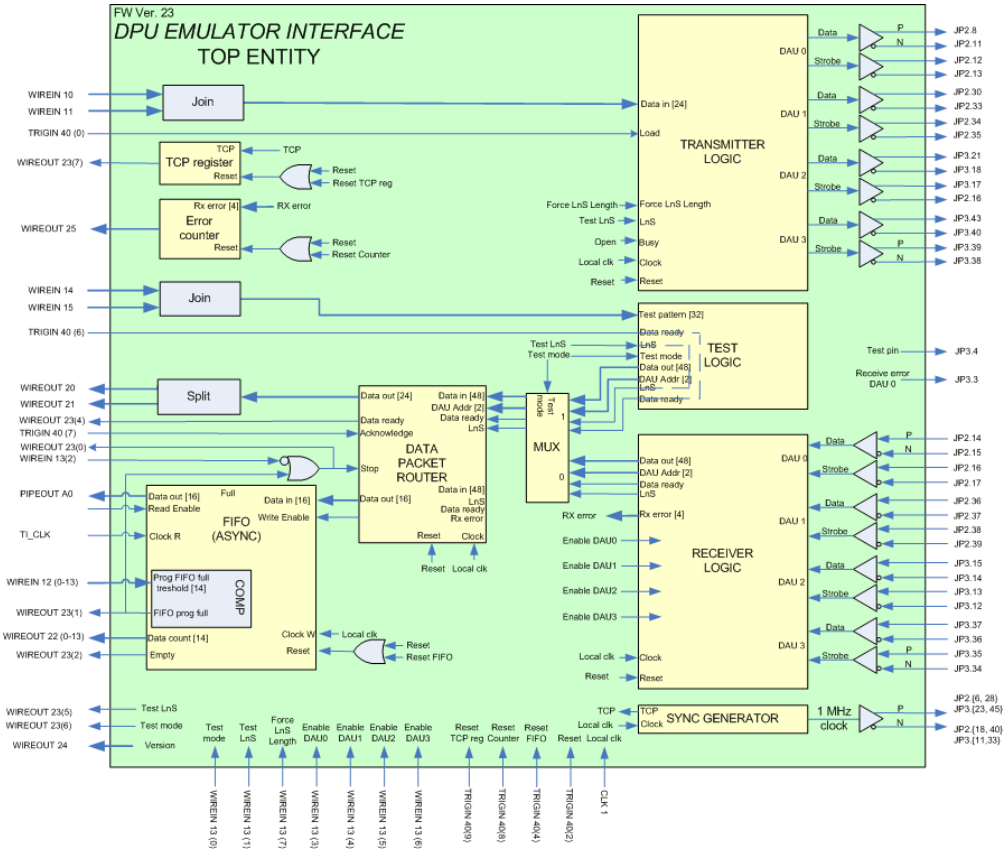


Figure 3.1: Internal structure and I/O signals of DPU Emulator Interface

3.2.2 Rewriting

Rewriting the firmware was divided into several phases.

Redesigning

The first phase established a working EGSE foundation. Some of the internal signals were level signals, but toggled with triggers. From an object oriented point of view, a function is generally enabled or disabled, not toggled. Consequently, all cases where Triggers were used to toggle between modes were changed to Wires.

Three TriggerOuts were used in the system. One indicated that a command reply was ready to be read, one that the FIFO was full and one that an error was received. Only the first is a proper trigger. Errors should be counted, and so long as the FIFO is full, a signal should remain asserted. Because there was an error in the Trigger Out implementation, and only one Trigger Out remained, it was replaced with a Wire Out and an acknowledge signal.

Finally, a DAU expects a 1 MHz Time Correlating Clock (TCP). The TCP is used to achieve a time resolution of 1 µs. Code emulating this clock was added in the firmware.

Support of four DAUs

Worst case scenario is when all DAUs send an SCDP at the same clock cycle and immediately send a new SCDP when the former is completed. To satisfy this timing constraint, four SCDPs must be relayed into the FIFO quicker than it takes one SCDP to be sent from a DAU. Calculations showed that this was satisfied and no architectural changes were needed.

An entity for multiplexing packets already existed in another project, and adapting it to xlink receivers required only minor changes. Once completed, it was simulated and tested.

Five bits is used by the xlink protocol for each word transmitted, thus a SCDP of 48 bits gives a total of 53 bits. The data packet router use one clock cycle to register a packet, and because the FIFO is 16 bits wide, it use three clock cycles to relay a packet into the FIFO. In total, four clock cycles is used by the data packet router per SCDP.

$$\text{SCDP rate} = \frac{18\ \text{Mbit/s}}{53\ \text{bit/SCDP}} \approx 340\ \text{kSCDP/s}$$

$$\text{Router capabilities} = \frac{36\ \text{Mwords/s}}{4\ \text{word/SCDP}} \approx 9.0\ \text{MSCDP/s}$$

Test mode

Adding three DAUs increased the SCDP rate to approximately 1.4 MSCDP/s. Test mode was modified to reflect this change, allowing the finished EGSE to be benchmarked for the worst case scenario. Support of short words was also added to test mode.

Increasing FIFO

The FIFO was enlarged to a depth of 2^{14} words. An SCDP occupy three words, hence the FIFO is capable of buffering 5461 SCDPs. One entry in the FIFO must remain unused as a buffer, thus 5460 is the SCDP capacity. With four DAUs, the FIFO is theoretically filled in 3.9 ms. The consequences of this are discussed in chapter 3.4.

$$\text{FIFO fill up time} = \frac{5460\ \text{SCDPs}}{1.4\ \text{MSCDPs/s}} = 3.9\ \text{ms}$$

3.3 DPU Emulator implementation

The DPU Emulator has a Graphical User Interface (GUI) that is intuitive to use. It consists of 6 windows, where the "Parent Window" is the main window. Two windows, "Configuration Window" and "Communication Window", are opened inside the Parent Window, while "XA-ASIC Window" and "Image Window" are opened as standalone windows. A menubar is used to control and program the DPU Emulator Interface. An operation manual of the program is included in appendix A.

Programming language used

Officially supported programming languages of Opal Kelly FrontPanel are C#, Visual Basic, Java, Python, C and C++. C# is a fully object oriented programming language. It uses the .NET framework, and its syntax is close to C and C++. Microsoft, who designed C#, explains it the following way [18]:

> C# (pronounced "See Sharp") is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, and Java programmers. [...] Contemporary software design increasingly relies on software components in the form of self-contained and self-describing packages of functionality. Key to such components is that they present a programming model with properties, methods, and events; they have attributes that provide declarative information about the component; and they incorporate their own documentation. C# provides language constructs to directly support these concepts, making C# a very natural language in which to create and use software components.

"Microsoft Visual C# Express" is an Integrated Development Environment (IDE) that is freely available. Programming a Graphical User Interface (GUI) in Visual C# Express is reduced to drag and drop in a "what you see is what you get" environment. Visual C# Express also includes auto completion of code, syntax checking, a debugger, and much more.

3.3.1 Addresses

To make the EGSE adaptable to future projects, addresses should be user configurable. This is the purpose of the "Configuration Window", which has one table for every level described in chapter 3.1.2. A variable is tied to every address, allowing scripts to address registers by name.

3.3.2 Communication with DAUs

Communication with DAUs is done through the "Communication Window", shown in figure 3.2. The window includes a textbox where DAUs are commanded, as well as functionality to acquire SCDPs. A simple script language allow for advanced commanding of DAUs. A quick reference of the scripting language is summarized in appendix B.3.

The class that implements the data acquisition proved more challenging than other classes. This is discussed in more detail in chapter 3.3.5.

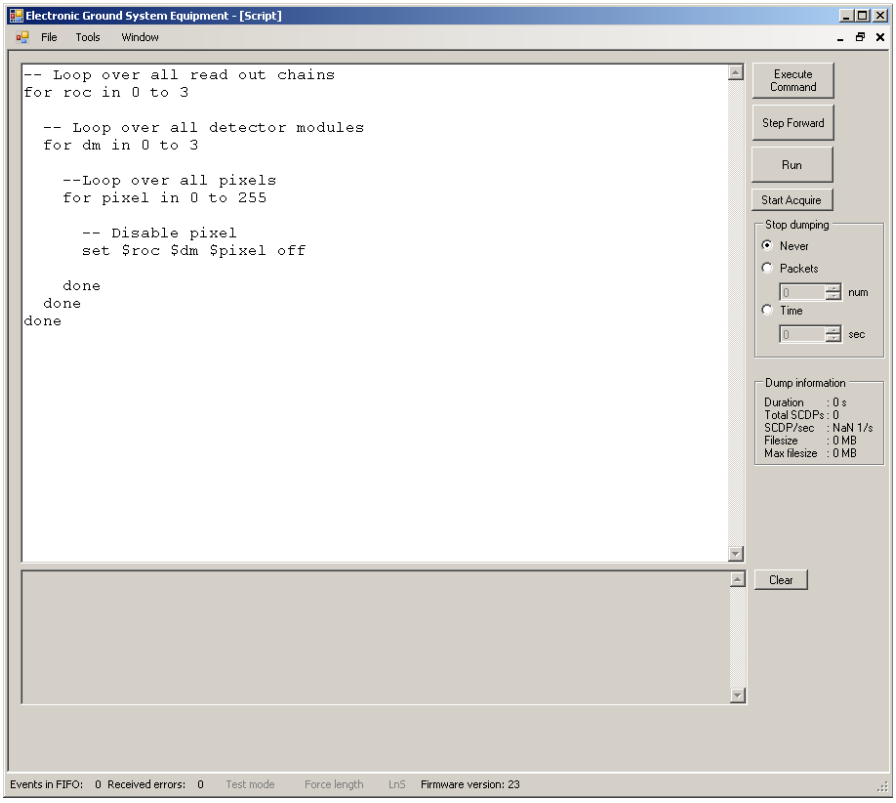


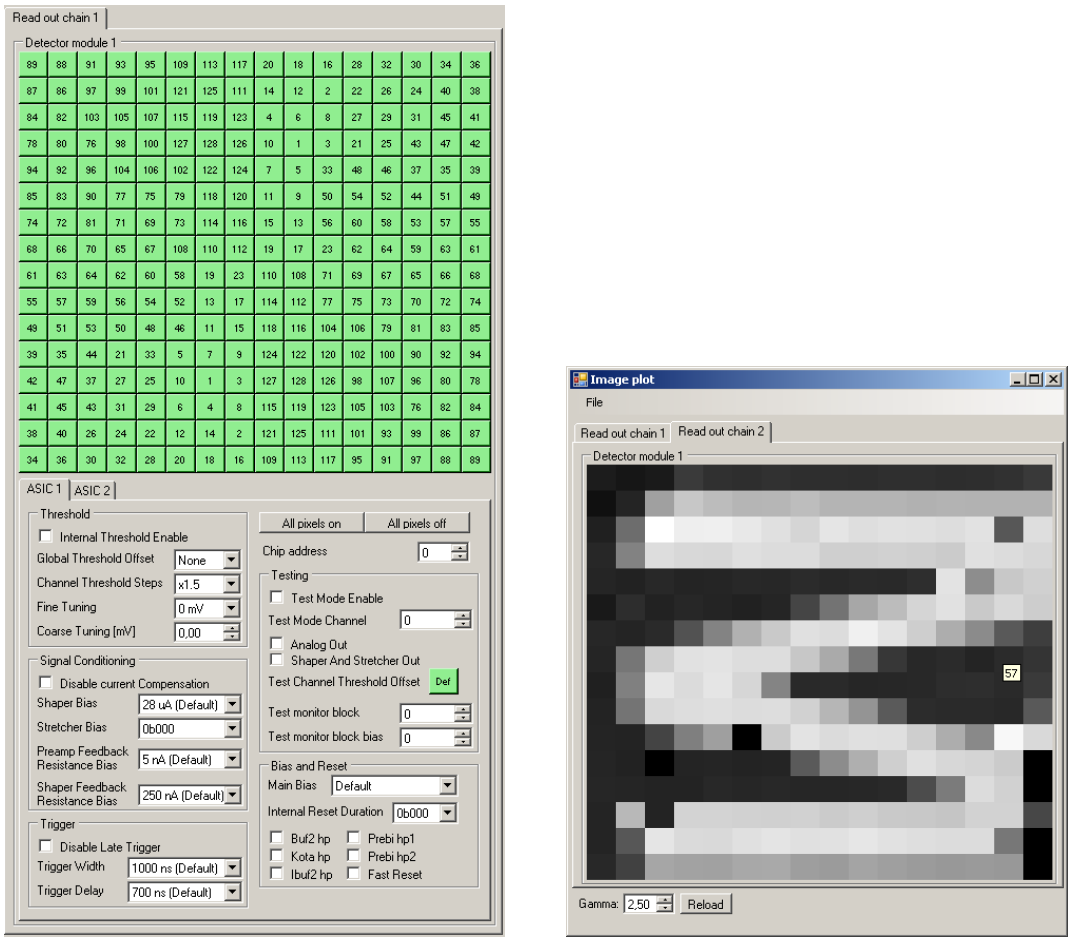
Figure 3.2: A screenshot of the Communication Window. The script in the textbox disables all pixels of a CZT

3.3.3 XA-ASIC Window

Figure 3.3a shows the layout of the XA-ASIC Window. From this window all aspects of all 32 XA-ASIC of a CZT can be configured graphically. The configuration can be saved to, and loaded from, an XML-file, as well as uploaded to a CZT DAU. All pixels can be enabled and disabled with a simple click, and pixel threshold is set from a right-click menu.

Uploading configuration

The process of uploading the CZT configuration to a read out chain is divided into four steps. First, the configuration is converted into a string of bits. Next, the



(a) XA-ASIC Window (b) Image Window

Figure 3.3: Figures showing one detector module of the respective windows

string of bits is uploaded to an XA configuration memory in the DAU. Finally, this memory is shifted into the daisy chain.

Combining figure 2.5 and the CZT and XA1.82 documentation, a configuration bit string (table 3.1) is generated and uploaded to the XA configuration memory. The XA configuration memory is then shifted inn big endian, starting with the lowest byte, thus the string is uploaded from left to right.

Testing

To convert the configuration into a string of bits, regularity is exploited once again. A separate method takes read out chain, detector module and XA-ASIC

<table>
<tr><td colspan="8">Read Out Chain 1
6864 bits</td></tr>
<tr><td colspan="2">DM 1
1716 bits</td><td colspan="2">DM 2
1716 bits</td><td colspan="2">DM 3
1716 bits</td><td colspan="2">DM 4
1716 bits</td></tr>
<tr><td>XA 2
858 bits</td><td>XA 1
858 bits</td><td>XA 4
858 bits</td><td>XA 3
858 bits</td><td>XA 6
858 bits</td><td>XA 5
858 bits</td><td>XA 8
858 bits</td><td>XA 7
858 bits</td></tr>
<tr><td colspan="8">Threshold, coarse, Bit 1 ... ... Dlt_Mode</td></tr>
</table>

Table 3.1: Configuration string of one read out chain.

as arguments. This method generates an 858 bit configuration string for the given XA-ASIC.

The string is generated in the same order as table B.4, thus making sure the order of settings is correct. Various settings were configured, and the configuration string of a single XA-ASIC was written to screen. The resulting output was compared to an expected result that was hand calculated.

Once it was established that it reliably produced configuration strings of 858 bits, two more methods were made. One method concatenates configuration strings of two XA-ASICs into a detector module configuration of 1716 bits. The other method concatenated four detector module strings into a read out chain configuration of 6864 bits. As a result, the structure of the overall read out process follow the layers in 3.1.

When it was exhaustively tested, the configuration procedure was tested together with a DAU. This process is described in chapter 4.

3.3.4 Image Window

Testing of the CZT started once it could be configured reliably, and chapter 4 is dedicated to the CZT test process. Acquired data showed that some pixels were noisy, meaning that they send false positives. Noisy pixels produce events at such a rate that the signal to noise ratio gets very low. The purpose of this window (See figure 3.3b) is to get a quick overview of noisy pixels so they can be disabled. It is also used to see if the pixel mapping seems correct.

A gray scale image is generated from the gathered data, where white is the most active pixels and black is the least active pixels. The gray scale is limited to 256 levels, so all pixels are normalized relative to the most active pixel. A side effect of normalizing relative to the most active pixel, is that the scaling pixel is always

white and "noisy".

The window has a user configurable gamma correction factor. When the pixel activity is similar this may be used to distinguish shades of gray.

3.3.5 Acquiring SCDPs

A computer is not a real time system, meaning that a computer program is occasionally put to sleep while other programs use the CPU. This gives a computer a relatively high latency, thus the DPU Emulator Interface requires a FIFO that buffers SCDPs. As briefly discussed in chapter 3.2.2, the FIFO is filled in 3.9 ms when data link between all four DAUs are saturated. Alternatively, it is filled in 15.6 ms when one DAU saturate the data link.

Typically, hits come in short bursts, and it is unlikely that the link is saturated for extensive periods of time. As an example, the data link is capable of continues rate of approximately 340 kSCDPs/s, but the BGO has its own FIFO and is capable of bursts of 650 hits/ms, nearly twice as much as the data link.

The DPU Emulator Interface FIFO is capable of buffering 5460 SCDPs, thus it can handle over eight full BGO bursts. However, the ideal situation would be that the EGSE was capable of handling a sustained saturation of all four data links. Therefore, the following chapter discus some measures that was taken to reduce packet loss.

Planning

A computer program can be put to sleep while it waits for something to happen. When it is put to sleep, its time slot on the CPU is given to another process. The minimum sleep duration is 1 ms and the operating system only assures it sleeps at least this long. Actual sleep durations depend on CPU load, operating system scheduler and program priority. A new CPU time slot is assigned anywhere from $1 - 20$ ms.

With no additional load, a sleep command of 1 ms was measured to 1.950 ± 0.001 ms. A delay of 1 ms cause a severe increase of data loss, as will be discussed in chapter 3.4. To reduce packet loss, sleep was abandoned in favor of a tight loop, which results in 100% usage of one CPU core while downloading.

Frequency accuracy on Opal Kelly XEM3001 is not guaranteed, but Opal Kelly experience approximately 50 ppm (parts per million) [19]. Assuming twice this,

xlink transfer at (18.0000 ± 0.0002) Mbit/s.

$$\begin{aligned}\text{Max rate of SCDPs } &= 4 \text{ DAU} \times \frac{18 \text{ Mbit}/(\text{s} \cdot \text{DAU})}{53 \text{ bit/SCDP}} \\ &= (1.358491 \pm 0.000015) \cdot 10^6 \text{ SCDP/s}\end{aligned}$$

The file size is unpredictable, but in worst case scenario 25 bytes is generated for every SCDP. This is very unlikely, thus only two significant figures is used in the following calculation.

$$\text{Max data rate } = 25 \text{ byte/SCDP} \times 1.4 \cdot 10^6 \text{ SCDP/s} \approx 35 \text{ MB/s}$$

A fast hard disk drive has a transfer rate of approximately 130 MB/s, so 35 MB/s takes a significant time to write. In less than 4 ms, it has to read the number of words in the FIFO, negotiate a USB time slot, download the content, and save it to disk. Consequently, the thread downloading data is time critical and the number of operations it has to perform should be reduced to a minimum. To further reduce packet loss, writing data to disk is delegated to another thread.

RAM buffers

Several types of RAM buffers exist in C#. For this project, only two were considered, "list" and "queue". Lists and queues were benchmarked for this particular implementation, and lists were used in the end (See chapter 3.4.1).

One CPU thread downloads data and writes it to RAM, and another thread reads from RAM and writes to disk. Two threads using the same list at the same time can potentially result in the wrong value being read or written. To avoid these complications, a "ping-pong"-buffer is utilized. In a "ping-pong"-buffer, a thread write to one buffer, while the other buffer is read by another thread. When it is finished reading, the buffers are swapped.

Improving write performance

When SCDPs are written to disk, they are split into columns so they can be analyzed by other computer programs. A straight forward method is converting the value into a string of bits and using string operations. A lot of slow mathematical operations are involved when transforming an integer into a binary string.

Fundamentally, everything is already represented in binary in a CPU, which can be exploited using bitwise operations. A bitwise operation is a fundamental logical operation operating at bit level. Examples of bitwise operations are shifting a number either right or left, applying logical AND or logical OR between two integers, etc. Bitwise operations are natively supported in CPUs, and are considerably faster than string operations.

To get a column of bits from a SCDP, to operations are used. First the SCDP is shifted to the right, so the desired column starts at the least significant bit. After it is position correctly, the bits to the left of the desired columns is removed. In practice, this means that they are ANDed with $2^N - 1$, where N is the number of bits in the desired column.

A benchmark comparison between the two methods showed that bitwise operations perform almost 40% better than string operations, in terms of write performance.

String write : 340±1 SCDP/ms
Bitwise write : 474±4 SCDP/ms

3.4 Benchmarking

Different parts of the program are subject to four different benchmarks. Some results are compared to the best possible performance, thus giving an indication of how well it is optimized. This also gives an indication of where there is potential for further optimization. Other tests show that it is impossible to satisfy all the timing constraints with the current architecture, and improvements are proposed in chapter 3.5.

Test procedure

A separate application was used for the benchmarks. All measurements are performed 100 times, unless specified otherwise. Uncertainty in a C# time measurement is unknown, so procedures executing in the order of milliseconds had two layers of averaging. An internal loop executes the procedure 100 times to decrease the uncertainty. All calculations use one standard deviation, unless stated otherwise.

Benchmarks involving SCDPs utilize test mode to generate data. Test mode simulates four DAUs sending a packet immediately after the former packet is completed. As previously discussed, this situation is unlikely. However, the scenario where all four data links are saturation indefinitely still serves as a good measurement of the system performance.

3.4.1 Buffers

A procedure fills two buffers (one list and one queue) with $2^{28} - 2$ bytes and reads the content back afterwards. The test reveals that queues spend 115% more time to fill and 74% more time to empty compared to lists.

Lists fill time	:	2.331 ± 0.006 ms
Lists read time	:	2.005 ± 0.002 ms
Queue fill time	:	5.010 ± 0.018 ms
Queue read time	:	3.485 ± 0.013 ms

3.4.2 APIs

Configuring an XA-ASIC read out chain or reading binning data both require several hundred commands. Each command involves two address lookups and several FrontPanel API calls. The purpose of this test is to locate where the bottleneck is, which can serve as a basis for optimizing the process. Both address conversion and Opal Kelly FrontPanels performance is tested.

Address conversion

Getting a register offset address includes a table lookup where all priorities are checked. The highest priority is looked up first and if a match is found, its address is returned and the search stops. If no match is found it tries a lower priority, and so forth. Consequently, the time requirement depends on the priority, and this benchmark tested the two extremes. The minimum time required is achieved when the input is a numerical value, and the address is returned directly. The maximum time is when it has to loop through all priorities.

Getting a register address consists of three steps. First it gets the module address, next it gets the register offset, and finally these are added together.

Get DAU address	:	0.0089 ± 0.0004 ms
Get register offset, min	:	0.0255 ± 0.0006 ms
Get register offset, max	:	0.395 ± 0.003 ms
Get register address, min	:	0.0262 ± 0.0004 ms
Get register address, max	:	0.418 ± 0.003 ms

Opal Kelly commands

Opal Kelly has measured how many Wires and Triggers operations are possible per second [9].

Update WireIns	:	1000+ per second	$\Rightarrow \sim 1.00$ ms
Update WireOuts	:	800+ per second	$\Rightarrow \sim 1.25$ ms
Activate Trigger In	:	1000+ per second	$\Rightarrow \sim 0.50$ ms
Update Trigger Out:		800+ per second	$\Rightarrow \sim 1.25$ ms

No uncertainty is given and the performance is very system specific, so a local benchmark was compared to their results. TriggerOuts are not used in EGSE and are left out of the test. WireIns used in EGSE have a wrapper on top of the FrontPanel driver. The wrapper allows for reading back Wire In values, and to set individual bits of a Wire In. Because of this, setting Wire In values are also benchmarked.

Set Wire In value	:	0.41 ± 0.04 ms
Update WireIns	:	0.875 ± 0.003 ms
Update WireOuts	:	1.2494 ± 0.0005 ms
Activate Trigger In:		0.4994 ± 0.0005 ms

Command sequence

With the current architecture in the DPU Emulator Interface, sending a command requires at least the following operations:

- Setting two Wire In values
- Updating WireIns to FPGA
- Activating a Trigger In to start serial transmission to DAU
- Updating WireOuts to read back reply
- Activating a Trigger In to acknowledge reply

Using the measured values for each operation and carrying out the calculation, one cannot expect a command to be quicker than approximately 3.44 ms.

$$\begin{aligned} \text{Duration of a command} &= 2 \times \text{Set Wire In delay} \\ &\quad + \text{Update WireIns} \\ &\quad + \text{Activate Trigger In} \\ &\quad + \text{Update WireOuts} \\ &\quad + \text{Activate Trigger In} \\ &= 3.44 \pm 0.06 \text{ ms} \end{aligned}$$

The class which is used to send commands employs the presented sequence, but in addition to performing these operations, it also converts variables to numerical addresses. Therefore it is expected to have slightly reduced performance. "Command sequence" below is just the operations, while "command API" is the operations and the best case address conversions.

$$\begin{aligned} \text{Command sequence} &: \quad 3.13 \quad \pm 0.02 \text{ ms} \\ \text{Command API} &: \quad 3.249 \pm 0.007 \text{ ms} \end{aligned}$$

3.4.3 Packet loss

Some SCDPs are lost when four DAUs produce SCDPs at full burst. The purpose of this test is to estimate the expected minimum packet loss that is realistic to achieve. This gives an indication of how well optimized the algorithm is.

The test is divided into three parts. First, a theoretical minimum packet loss is calculated based on the performance of Opal Kelly. Next, the local performance is measured and used to calculate a realistic packet loss. Finally, the actual packet loss is measured.

Discussion

It is not possible to determine if the downloaded data is genuine or garbage based on the content itself. Because of this, the program first reads the number of words buffered in the FIFO to determine how many bytes to download. In practice, this is done by updating the WireOuts.

Updating the WireOuts takes approximately 1.25 ms. Consequently, the effective time it takes to download the content is 1.25 ms more than it takes to download the actual data.

$$\text{Effective duration} = \text{Duration of Wire Out} + \text{Duration of data transfer}$$

The preceding tests use the following relation to calculate an effective transfer rate.

$$\text{Effective transfer rate} = \frac{\text{Bytes}}{\text{Effective duration}}$$

Minimum requirement for lossless performance

Using the rate of SCDPs from chapter 3.3.5, and knowing that each SCDP require 8 bytes, the bandwidth requirement is calculated.

$$\begin{aligned}\text{Minimum bandwidth requirement} &= \text{Size of each SCDP} \times \text{Rate of SCDPs} \\ &= 8 \text{ bytes/SCDP} \times 1.358491 \cdot 10^6 \text{ SCDP/s} \\ &= 10.36446 \pm 0.00011 \text{ MB/s}\end{aligned}$$

The downloaded data is buffered in a FIFO with a depth of $2^{14} - 2 \approx 2^{14}$ words and a width of 2 bytes. Rounding to 2^{14} results in an error which is negligible compared to the frequency error, thus ignored.

$$\text{FIFO size} = 2^{14} \text{ words} \times 2 \text{ bytes/word} = 32 \text{ kB}$$

Theoretical limit

With small block sizes, there is a large overhead because of the frequent negotiating processes, hence it is expected to have low transfer rate. As the block size increase, it is expected that the transfer rate steadily increases. The USB2 protocol is capable of 480 Mb/s, but with an effective throughput of around 35 MB/s [20]. So as the block size continues to increase, it is expected that the increase of transfer rate gradually declines as the transfer rate is approaching the maximum rate.

A theoretical minimum packet loss is calculated using Opal Kellys benchmark performance (Table 3.2). No data for 32 kB is provided, but plotting the table in a log-linear plot, figure 3.4, shows the expected behavior with an almost linear relationship.

Transfer length	**Transfer rate Pipe Out**	**Transfer rate Pipe In**
128 B	100 kB/s	100 kB/s
256 B	200 kB/s	100 kB/s
512 B	400 kB/s	300 kB/s
1.0 kB	800 kB/s	700 kB/s
4.0 kB	3.1 MB/s	2.8 MB/s
16.0 kB	10.4 MB/s	8.9 MB/s
64.0 kB	23.2 MB/s	20.8 MB/s
256 kB	32.7 MB/s	31.8 MB/s
1.0 MB	36.7 MB/s	36.5 MB/s
4.0 MB	37.9 MB/s	37.9 MB/s
8.0 MB	38.1 MB/s	38.2 MB/s

Table 3.2: Benchmark of the download performance of Pipes [9].

The transfer rate at block size of 64 kB and 256 kB is extrapolated and used as a rough estimate of the transfer rate at 32 kB. Letting R = transfer rate and L = $\log_2$(transfer length in kB), an approximate rate is:

$$\begin{aligned} \mathrm{R(L)} &\approx 4.75\ \mathrm{MB/kB \cdot s} \times L - 5.3\ \mathrm{MB/s} \\ \mathrm{R(32\ kB)} &\approx 18.5\ \mathrm{MB/s} \end{aligned}$$

Calculations suggest that a transfer length of 32 kB has an approximate transfer rate of 18.5 MB/s. Opal Kelly measured 800 Wire Out updates per second, thus 1.25 ms is added for this calculation.

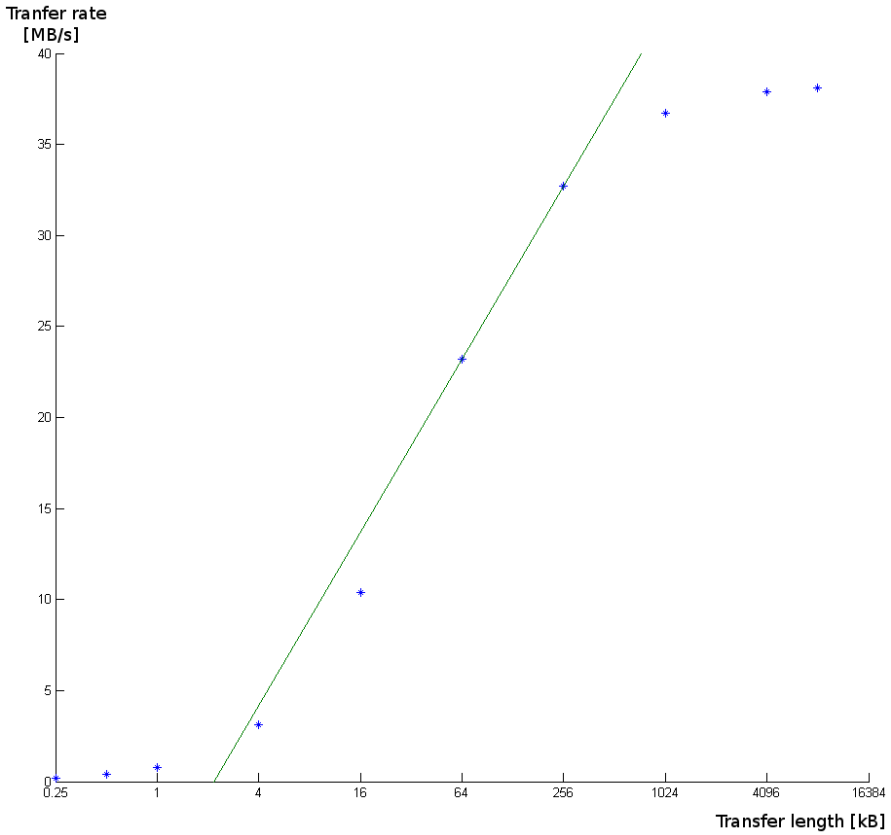


Figure 3.4: Ekstrapolation of Opal Kelly Pipe Out performance. A transfer length of 32 kB has an approximate transfer rate of 18.5 MB/s.

$$
\begin{aligned}
&\text{Approximate transfer rate} && && = 18.5\ \text{MB/s} \\
&\text{Duration of download} && = \frac{32\ \text{kB}}{18.5\ \text{MB/s}} && = 1.69\ \text{ms} \\
&\text{Effective duration} && = (1.69 + 1.25)\ \text{ms} && = 2.94\ \text{ms} \\
&\text{Effective transfer rate} && = \frac{32\ \text{kB}}{2.94\ \text{ms}} && = 10.6\ \text{MB/s}
\end{aligned}
$$

The minimum requirement is approximately 10.4 MB/s, but because no uncertainty is provided, it is not concluded that lossless performance is feasible from this estimate.

Practical limit

A benchmark measured the maximum transfer rate achieved in this environment. The FIFO was filled up to the maximum value, and the duration it took to download was measured. This shows a transfer rate significantly lower than Opal Kellys performance.

$$\text{Measured transfer rate} = 12 \pm 2 \text{ MB/s}$$

A practical limit is calculated from the basis of a local maximum transfer rate. It is first assumed that the downloaded data is not further processed.

$$\begin{aligned}
\text{Duration of download} &= \frac{32 \text{ kB}}{12 \text{ MB/s}} &&= 2.6 \pm 0.4 \text{ ms} \\
\text{Effective duration} &= (2.6 + 1.2494) \text{ ms} &&= 3.8 \pm 0.4 \text{ ms} \\
\text{Effective transfer rate} &= \frac{32 \text{ kB}}{3.8 \text{ ms}} &&= 8.6 \pm 0.9 \text{ MB/s} \\
\text{Minimum loss} & &&= (17.0 \pm 1.7)\%
\end{aligned}$$

The thread downloading data can probably be relieved of writing to RAM, thus be able to achieve this performance. In this implementation, however, writing 2^{28} bytes to a RAM buffer took 2.331 ± 0.006 s. Assuming a linear relationship, 32 kB written to a RAM buffer gives an additional delay.

$$\text{Writing to buffer} = \frac{2.331 \text{ s}}{2^{28} \text{ B}} \times 32 \text{ kB} = 0.2845 \pm 0.0007 \text{ ms}$$

Adding this delay into the calculation, gives a realistic minimum loss of

$$\begin{aligned}
\text{Effective duration} &= (3.8 + 0.2845) \text{ ms} &&= 4.1 \pm 0.4 \text{ ms} \\
\text{Effective transfer rate} &= \frac{32 \text{ kB}}{4.1 \text{ ms}} &&= 8.0 \pm 0.7 \text{ MB/s} \\
\text{Expected loss} & &&= (23 \pm 2)\%
\end{aligned}$$

Measured loss

To get an indication of how effective the implemented download algorithm is, a procedure acquires data 10 times, each time a duration of 30 seconds. Packets are enumerated sequentially in test mode, hence the number of lost packages is known. Calculations showed that the average packet loss was in agreement with the expected packet loss. As a conclusion, the implemented downloading algorithm is close to optimal, when the achieved transfer rate is taken into account.

$$\text{Packet loss } = \frac{\text{packets received}}{\text{packets received + packets lost}} = (26.5 \pm 0.9)\%$$

3.4.4 EGSE API integration

Originally the dataset API, which is used to convert from named to decimal addresses, used string operations. When a sub-section of bits is picked out from a string of bits, it is easy to see what bits are picked out, and consequently the source code is easy to read and debug.

Benchmarking the write performance (See chapter 3.3.5) showed that bitwise operations performed 40% better than string operations. Preliminary benchmarks showed that sending commands took roughly twice as much time as the theoretical minimum. With this in mind, it was optimized by changing from string operations to bitwise operations. As a result, the time used to send 1000 commands was reduced by 30%, from almost 6 seconds to 4 seconds.

Measurements

The DPU Emulator was temporarily modified to measure the time of a for-loop with 1000 iterations. Inside the for-loop there was a single memory read command, so its performance could be compared with the benchmarked performance. Surprisingly, this resulted in a rather large overhead of 19%.

In the process of locating why this was the case, more temporary modifications were applied. One modification removed script processing completely, and the command sequence from 3.4.2 was substituted. This should make the two tests directly comparable, but as "Sending command" below shows, it performs significantly slower.

The other modification kept script processing but removed the actual sending of a command. Everything except for FrontPanel operations was executed as normal. This measured the script parsing overhead, "Script parsing without sending commands" below. "Script parsing" below, is the normal behavior without modifications.

Sending command	:	3.77 ± 0.04 ms
Script parsing without sending commands	:	0.0053 ± 0.0001 ms
Script parsing	:	3.86 ± 0.04 ms

Performing only API calls required to send a command ("Sending command" above) resulted in a 16% higher execution time compared with the benchmark program. Why this is, remains unclear.

When the normal API is used the reply is compared with an expected reply. Thus in addition to the two address conversions, some if-checks and integer operations are performed. Neglecting these, which are in the order of nano- or microseconds, the following relation is a good approximation:

$$\begin{aligned} \text{"Command sequence"} &+ \text{"Script parsing without sending commands"} \\ &+ \text{"Get register address, min"} \\ &+ \text{"Get DAU address"} \approx \text{"Script parsing"} \end{aligned}$$

Carrying out the calculation gives

$$\text{Expected script parsing} = 3.81 \pm 0.04 \text{ ms}$$

This is in agreement with the measurements, thus the only delay that cannot be accounted for is the additional 16% overhead for the FrontPanel API calls in the EGSE software.

3.5 Proposed design improvements

Ideally, 5000 bytes of binning data should be read out in less than one second. With 1 byte per memory read command, this sets a timing constraint of 200 µs per command. Chapter 3.4 showed that a theoretical minimum is 3.13 ms per command, 15 times more than the timing constraint.

3.5.1 Reorganizing the FIFO

As discussed, the FIFO is 16 bits wide, and a SCDP is divided into three words. RAM in the Opal Kelly XEM3001 is divided into blocks of 18 kb [21].

$$\begin{aligned} \text{Size of FIFO} &= 16 \text{ bit/word} \times (2^{14} - 1) \text{ words} = 262128 \text{ bit} \\ \text{Blocks used by FIFO} &= \frac{262128 \text{ bit}}{18 \text{ kbit/block}} = 14.2 \text{ blocks} = 15 \text{ blocks} \end{aligned}$$

Whole blocks must be assigned for a specific width, thus the FIFO utilizes 15 of 16 available RAM blocks. Reorganizing the FIFO so one SCDP occupies one word, gives several advantages.

Changing width

On rare occasions, it was observed that one or two words of a SCDP did not get buffered in the FIFO. Consequently, the first word of the following SCDP becomes the last word of the previous SCDP. This causes a misalignment, and because there is no error detection, all data that follows is corrupted without warning. This problem is fixed by leaving room for one extra SCDP at the end of the FIFO. However, this had not been an issue if a SCDP was not split into three words.

Utilizing DAU address bits

Both BGO and CZT SCDPs have two unused bits in the address field, but these were left unused for unforeseen future changes. Instead the header bits were overwritten with the DAU address, as they have served their purpose. Because the FIFO is 16 bits wide, there was nothing to gain from removing the header bits. However, if a SCDP occupies one word in the FIFO, header bits could be removed. The same number of SCDPs could be buffered, but with 2 less bits, thus reducing the number of used blocks.

$$\text{Blocks used by FIFO} = \frac{5461 \text{ words} \times 46 \text{ bit/word}}{18 \text{ kbit/block}} = 13.6 \text{ blocks} = 14 \text{ blocks}$$

Adding flag bit

Increasing the width to 47 bits, does not increase the number of used blocks.

$$\text{Blocks used by FIFO} = \frac{5461 \text{ words} \times 47 \text{ bit/word}}{18 \text{ kbit/blocks}} = 13.9 \text{ blocks} = 14 \text{ blocks}$$

Letting a flag bit indicate whether or not a word is occupied, downloaded words that are all zeros could be discarded. This relieves the DPU Emulator of checking how many words to download, thus reducing the effective download duration with 1.25 ms. Using data from chapter 3.4, a new effective transfer rate may be calculated. In addition, bandwidth requirement is reduced from 48 to 47 bits.

In the following calculation, the delay of writing the data to RAM is included to get a realistic calculation. The number of bytes to download has no uncertainty. The correct number of byte was used in the calculations, but an approximation is shown for practical reasons.

Bytes to download	$= 5461 \text{ words} \times 47 \text{ bit/word} \approx 31.331 \text{ kB}$	
Write to buffer	$= \frac{2.331 \text{ s}}{2^{28} \text{ B}} \times 31.33 \text{ kB}$	$= 0.2785 \pm 0.0007 \text{ ms}$
Duration of download	$= \frac{31.33 \text{ kB}}{12 \text{ MB/s}}$	$= 2.5 \pm 0.4 \text{ ms}$
Effective duration	$= (2.5 + 0.2785) \text{ ms}$	$= 2.8 \pm 0.4 \text{ ms}$
Expected transfer rate	$= \frac{31.33 \text{ kB}}{2.8 \text{ ms}}$	$= 10.9 \pm 1.6 \text{ MB/s}$
Required transfer rate		$= 10.36446 \pm 0.00011 \text{ MB/s}$
Bandwidth utilization		$= (95.1 \pm 1.4)\%$

In general, this scheme allows lossless data transfer, however the time margins are small. A disturbance on the computer may still result in occasional packet loss.

3.5.2 Sending commands

Reading memory faster can be achieved in two ways: by using Pipe In or adding an address incrementer in the FPGA firmware. Regardless of method chosen, sending commands at a higher rate requires replies to be read with a Pipe Out. For reasons explained shortly, this requires a major reorganization of the source code.

Reading replies

With the current implementation, sending a command is a stand alone process. It is implemented in a separate class, and two windows use this class to send commands independently. This works well when a reply can be read immediately after it is sent. When using a Pipe Out for replies, they are transferred in bulk and needs to be buffered in a FIFO. This complicates the process when two windows want to send commands independently.

Command replies can either be routed to the existing SCDP FIFO, or to a new FIFO. Utilizing the existing FIFO requires restructuring of how SCDPs are downloaded, but it has the advantage that it allows more commands to be buffered for sending. However, if the command replies are routed to a separate FIFO, the remaining RAM must be divided between two FIFOs, thus reduce the number of uploaded commands that can be buffered.

Router class

A router class could be implemented so that it lies in the background and continuously empty the FIFO. The DPU Emulator has to download command replies, possibly from two different windows intermingled, and route them to the correct window. This could be fixed by adding a router class, and let the windows communicate with a router instead of the DPU Emulator Interface directly. To make sure replies are routed to the correct window, and compared with the correct sent command, a serial number has to be added.

Including DAU address bits, and letting the serial number be 6 bits, a memory read or write command is 32 bits long. If the FIFO is reorganized as proposed, two blocks of 18 kb RAM is available.

$$\text{Number of buffered commands} = 2 \text{ blocks} \times \frac{18 \text{ kb/block}}{32 \text{ b/command}} = 1152 \text{ commands}$$

Routing command replies to the SCDP FIFO, 1152 commands can be buffered for sending. Alternatively, it can be split in two FIFOs of 576 commands each, one for commands buffered for sending and one for replies.

Sending commands

To satisfy timing constraints, Wires and Triggers for every command must be abandoned in favour of a new approach. To get the maximum performance, the FPGA has to be involved in the read out process.

Memory read commands can be modified with an extra argument specifying how many bytes to read, starting from the given address. This solves the particular problem of reading many registers, but it cannot be used for memory write commands. Replacing WireIns with a Pipe In is the most flexible solution. It requires more modifications, but it may be used for both read and write commands.

The command must be transmitted to the DAU, and the DPU Emulator Interface has to wait for the response. The six bit serial number added is not sent to the DAUs, but used in the DPU Emulator Interface. A command sent to a DAU is 24 bits, but an additional 5 bits are used by the xlink protocol. In total, 29 bits is transmitted at 18 Mb/s, which is the transfer rate of the data link. It is assumed that a maximum of 10 extra clock cycles at 36 MHz, or 5 at 18 MHz, is used to process each command, though this is not true for all commands.

Utilizing two FIFOs, the block size is reduced to 1 kB, and an approximate transfer rate of 700 kB/s and 800 kB/s for Pipe In and Pipe Out respectively. Assuming that the local performance is comparable, and that one TriggerIns and one Wire Out update is required, the number of commands per second can be approximated. The Trigger In is used to start sending the commands after they have been uploaded, and a Wire Out is used to signal that all commands are finished.

Time spent transferring data:

$$\begin{aligned}
\text{Bytes to transmit} &= 576 \text{ commands} \times 32 \text{ bits/command} &&= 2.25 \text{ kB} \\
\text{Duration of upload} &= \frac{2.25 \text{ kB}}{700 \text{ kB/s}} &&= 3.2 \text{ ms} \\
\text{Duration of download} &= \frac{2.25 \text{ kB}}{800 \text{ kB/s}} &&= 2.8 \text{ ms}
\end{aligned}$$

Time waiting for commands to process:

$$\begin{aligned} \text{One command} &= \frac{(29+5+29)\text{ bit/command}}{18\text{ Mbit/s}} &&= 3.5\ \mu\text{s} \\ \text{All commands} &= 3.5\ \mu\text{s/command} \times 576\text{ commands} &&= 2.0\text{ ms} \end{aligned}$$

Then the round trip time can be approximated as:

$$\begin{aligned} \text{Round trip time} &= \text{Upload commands} + \text{Processing commands} \\ &\quad + \text{Update WireOuts} + \text{Download commands} \\ &\quad + \text{Trigger In} \end{aligned}$$

Thus the approximation yields:

$$\begin{aligned} \text{Round trip time} &= (3.2 + 2.0 + 2.8 + 0.4994 + 1.25)\text{ ms} &&= 10\text{ ms} \\ \text{Commands per second} &= \frac{576\text{ commands}}{10\text{ ms}} = 58 \cdot 10^3\text{ commands/s} \end{aligned}$$

As a conclusion, preliminary calculations suggest that the proposed solution is well within the timing constraint of 5000 commands/s. There is no need for further optimizations by combining command replies with SCDPs so 1152 commands can be buffered for sending. There is, however, a little overhead when a single command is sent compared to the current solution. It is possible to allow the current and the proposed solution to coexist, and then choose which ever is most suited in a given situation.

The minimum block size of Pipes is 128 bytes with a transfer rate of 100 kB/s. The following calculation is an approximation when a single command is uploaded through a Pipe Out, where the processing time of a single command is neglected.

$$\begin{aligned} \text{Duration of transfer} &= \frac{128\text{ B}}{100\text{ kB/s}} &&= 1.25\text{ ms} \\ \text{Round trip time} &\approx (1.25 + 1.25 + 0.4994 + 1.25)\text{ ms} &&\approx 4.2\text{ ms} \\ \text{Commands per second} &\approx \frac{1\text{ command}}{4.2\text{ ms}} &&\approx 240\text{ commands/s} \end{aligned}$$

3.5.3 Increasing download performance

An experimental method where a Pipe Out was replaced by a Block-Throttled Pipe Out (BTPipe Out) was tested. When data arrived continuously, it worked as expected, and reduced the data loss to approximately 2%. However, when SCDPs stopped arriving, e.g. disabling test mode, it stopped responding.

BTPipe Out implements a FIFO with a threshold set to a user configured block size. FrontPanel only transfer blocks of data with this exact size, thus reducing the USB negotiation required and increasing the transfer rate [9]. The probability of getting an exact multiple of the block size is very slim, so when the SCDP source is stopped, it never fills the last block. This results in the FrontPanel API freezing while it waits for the FIFO to fill.

A solution presented by Opal Kelly is not using BTPipe Out unless it is guaranteed that the block will be filled [22]. In reality, this is no solution, as it requires a high level negotiation, for example using WireOuts.

4 Software and hardware co-testing

The Electronic Ground System Equipment is used to characterize and debug the detectors in development. This testing started while the EGSE was still in development. Some challenges arise when a program in development is used for debugging an electric system, because there is no frame of reference known to be correct.

Testing of the CZT and EGSE, and configuration of the read out chains in particular, were carried out in parallel. For the sake of clarity, testing of the EGSE and CZT is discussed separately. Chapter 4.1 discusses how the EGSE was tested. Throughout this process, bugs were corrected as they were found, and the test repeated once the bug was corrected. Chapter 4.2 discusses how the CZT was tested, as well as some of the errors that were disclosed throughout the tests.

4.1 EGSE Tests

It was known that the DAUs had some issues that were being worked with. However, the EGSE was not yet confirmed to be working so no conclusions of where the error lied could be drawn until the EGSE was reliable.

Configuring a CZT is a very intricate process of four steps. Three of the four steps have some pitfalls that must be tested.

1. An 858 bit string has to be composed for each XA-ASIC
   (a) Settings must be in the correct order
   (b) Groups of bits must be oriented correctly
   (c) Logical pixels must be mapped to the correct channel
2. Eight bit strings, each of 858 bits, is concatenated into a 6864 bit read out chain configuration
   (a) The eight strings must be in the correct order
   (b) Each string must be oriented correctly

3. The 6864 bit configuration string is divided into 858 bytes of data and uploaded to the XA configuration memory. This is achieved with memory write commands
   (a) Each byte must be uploaded to the correct memory location
   (b) A byte must be uploaded with correct endian
4. Configuration memory is shifted into a read out chain

All steps must be performed without errors for the configuration to be successful. In total, seven pitfalls exist (Sub-bullets a - c above) related to generation and configuration of a CZT. Several tests were applied, to check that all pitfalls were avoided.

4.1.1 Uploading configuration test

When an XA-ASIC is powered up, random data is contained within the configuration shift register. The first test was therefore to check that all pixels could be disabled. This is achieved by shifting inn a pattern where all pixels are deactivated, and then monitor that the trigger activity stops.

This test establishes that the XA configuration memory is written to, that a configuration memory is shifted into the read out chain and finally, that all channels are disabled as expected.

4.1.2 Bit sequence test

"DLT_mode" (Disable late trigger) and a pattern of channels was enabled when it was established that configuration was possible. The purpose of this test is to examine the bit sequence shifted into the daisy chain by probing with an oscilloscope. DLT is the first bit of the configuration register, and serves as a reference.

This test shows whether or not the bit sequence is shifted in reversed order, that the channel sequence is correct relative to DLT and that bytes are uploaded with the correct endian.

4.1.3 Testing groups of bits

A group of bits can either be uploaded as $B_1 \ldots B_N$ or $B_N \ldots B_1$, but it was unclear from the XA documentation which of them was correct. Each XA-ASIC has a coarse threshold setting that applies to all channels. Enabling one pixel

and adjusting the threshold, changes the activity. If the group of bits is uploaded correctly, the expected result of an increased threshold is reduced activity.

By observing the trigger activity while changing the threshold, the orientation of groups of bits is established.

4.1.4 Checking content of the configuration

Next, the first 33 and last 51 bits of the configuration register were manually checked by uploading test patterns to the XA configuration memory. Content of the memory was read back and compared with an expected string, as well as observed with an oscilloscope while it was shifted into the read out chain. The settings were also compared to that of the LabVIEW program, which showed that the configuration was equivalent.

Individual channel settings utilize the remaining 774 bits. These bits were not tested individually, but it was verified that the first 33 bits were followed by 129 channel disable bits, and that 129 test enable bits were next to the 51 last bits. Calculations also showed that there were 516 bits in between, thus a total of 858 bits. Appendix B.4 has a complete overview of the XA configuration register.

4.1.5 Verifying detector modules and pixel map

A detector module is composed of 256 pixels, each assigned a logical address (See appendixC). Logical pixels 1 to 128 are connected to XA-ASIC 1, and logical pixels 129 to 256 are connected to XA-ASIC 2, but there is no direct relationship between logical pixel and channel, thus a pixel map is required. An existing pixel map was used as a starting point for this test.

A lead plate with ”M X G S” cut out was placed above a read out chain with one letter directly above each detector module, starting with ’M’ above detector module 1 (See figure 4.1). Approximately 50 cm above the lead plate there was a radio active source.

Lead attenuates photons, so pixels directly beneath letters are more exposed to radiation. Acquiring data from this setup and loading it in the Image Window, the expected result is ”M X G S”, starting at detector module 1, through 4. Two inconsistencies were observed. The letters showed ”S G X M”, starting from detector module 1, and half of the image of each detector module was rotated 180° relative to the other half.

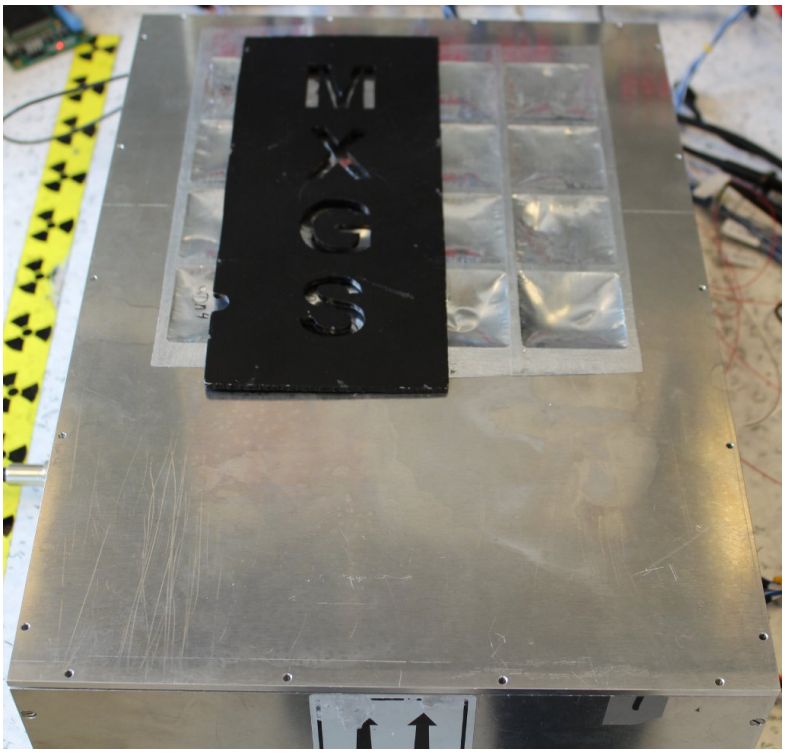


Figure 4.1: CZT DAU with MXGS lead plate on top. There is one letter above each detector module. Pixels beneath letters are more exposed to radiation.

The source code was scrutinized to check whether the existing mapping was implemented correctly, and it was concluded that the pixel mapping had to be rotated on the second XA-ASIC. The order of detector modules had accidentally been reversed, which was also corrected. Redoing the test, showed the satisfactory result of "M X G S" appearing as expected (See figure 4.2). Appendix C shows the corrected pixel map.

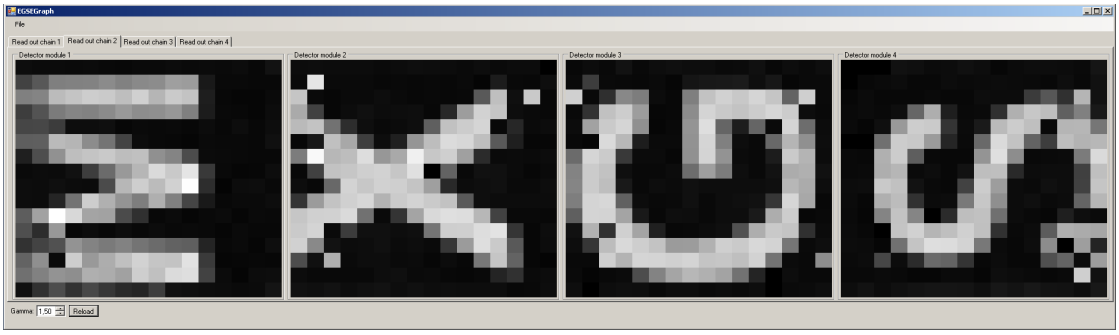


Figure 4.2: Image plotted from data received in this test setup

4.1.6 Summary

First it was verified that a read out chain can be configured (Chapter 4.1.1), thus step 4 is tested. The next test showed that all sub-strings of 858 bits are shifted into the read out chain with the correct orientation (Chapter 4.1.2). This test would have disclosed errors related to pitfalls 2.b and 3.b. Then, a test showed that groups of bits got shifted in with the correct orientation, thus checking pitfall 1.b (Chapter 4.1.3).

Chapter 4.1.4 discussed how settings, except channel specific settings, were tested. Because channel specific settings were left out, pitfall 1.a and 3.a were only partly tested. Finally, chapter 4.1.5 took an image and confirmed that pitfall 2.a was avoided. The image gives a good indication of the pixel map, but it is not confirmed from the basis of this test.

As a conclusion, there is no reason to believe errors are present in the configuration process, except from possibly small errors in the pixel map.

4.2 CZT Tests

Most of the tests were performed with one read out chain at a time, but all read out chains were subject to most of the tests. First the different tests are discussed, then a summary of the discoveries is given. The tests are not presented chronologically.

4.2.1 Configuration

A CZT can be commanded to check the configuration it shifts in. When the bits are shifted into the daisy chain, the old configuration is shifted out. A bit shifted out is XORed with the bit shifted in. If the configuration shifted in is the same as the configuration shifted out, the XOR should always yield 0. If one of the 6864 XORs is 1, the configuration is different and reported.

Configuration patterns of this test were shifted in twice. Because the first pattern shifted in is new, an error is expected the first time it is configured. The second configuration has the same pattern and no error is expected.

4.2.2 Trigger and multihit signals

When a hit is detected by an XA-ASIC, a trigger puls is sent. There is only one trigger output per XA-ASIC, a digital current signal. If two or more hits happen at the same time the two currents are added, thus the trigger output is proportional to the number of simultaneous hits.

This test is divided in two. The first test enables a single pixel to observe some properties of single hits. The second test enables several pixels to test the multihit behavior.

Trigger

The first test had a single pixel enabled. An oscilloscope probed the trigger signal, which also counted the trigger rate. A radio active source was used to increase the activity, and the expected result is an increased trigger rate measured with the oscilloscope.

The EGSE acquired data produced in this process, and from these data, a rough SCDP production rate was estimated. This gave a quantitative comparison of the oscilloscope activity and SCDP production rate, which gives an indication if the CZT produced SCDPs reliably. It was also checked for multihit packets, which should not happen for this test.

One of the configuration options of the XA-ASIC is trigger width, which is configured from the EGSE software. The trigger width was changed from the GUI, while it was measured with an oscilloscope.

Multihit

Location of particle events is put onto an address bus of the XA-ASIC. If two enabled pixels produce an event at the same time, two addresses are superimposed on the address bus. Because of this, it is important to report in the SCDP that it was a multihit, implying that the address is wrong. Two bits are reserved the multihit signal in a CZT SCDP, hence a maximum of three multihits are reported.

As discussed, the trigger output is proportional to the number of simultaneous hits. An oscilloscope probed the trigger output to see that multihit occurred. Acquired data from this test should have some packages with multihit, but there should not be more multihits than pixels enabled.

4.2.3 Address

A CZT address is composed of 14 bits but split into two parts. Seven bits are pixel address, and seven are XA-ASIC address. This test focused on the XA-ASIC address, which is configurable from the XA-ASIC Window.

A single pixel was enabled for this test, so multihits did not interfere. Data was acquired using the EGSE, and event address compared to the address on the bus.

4.2.4 Energy

Energy output from an XA-ASIC is an analog signal proportional to the deposited energy, and if multihits occur, all energies are added. This test probed the multihit and energy signals simultaneously. The purpose was to see that energy from two separate XA-ASICs was added correctly.

The second part of this test checked the ADC (Analog Digital Converted). A radioactive source with known energy spectrum served as a calibration source, and data was acquired with the EGSE. Figure 4.3 shows an energy plot of an Am-241 source taken by a CZT DAU after noise is removed.

4.2.5 Summary

Shifting in a new configuration indicated they were different and shifting in the same configuration indicated they were equal, as expected. Trigger and multihit tests showed that the first and fourth read out chain had some problems connected to these signals. The test also showed that trigger width changed when configured from XA-ASIC Window, but the measured width did not coincide with the configured width.

Gathered data showed a consistent right shift of the XA-ASIC address. SCDPs from test mode are read reliably, so the error is either related to wrong configuration, or related to the address bus going from the detector modules to the FPGA.

Multihit energy tests disclosed no inconsistent behavior. Pixel threshold was not configured at this time, so a spectrum plot of the gathered data showed a significant portion of noise. However, the energy peak and full width at half maximum (FWHM) gave a quantitative indication that the ADC operated properly.

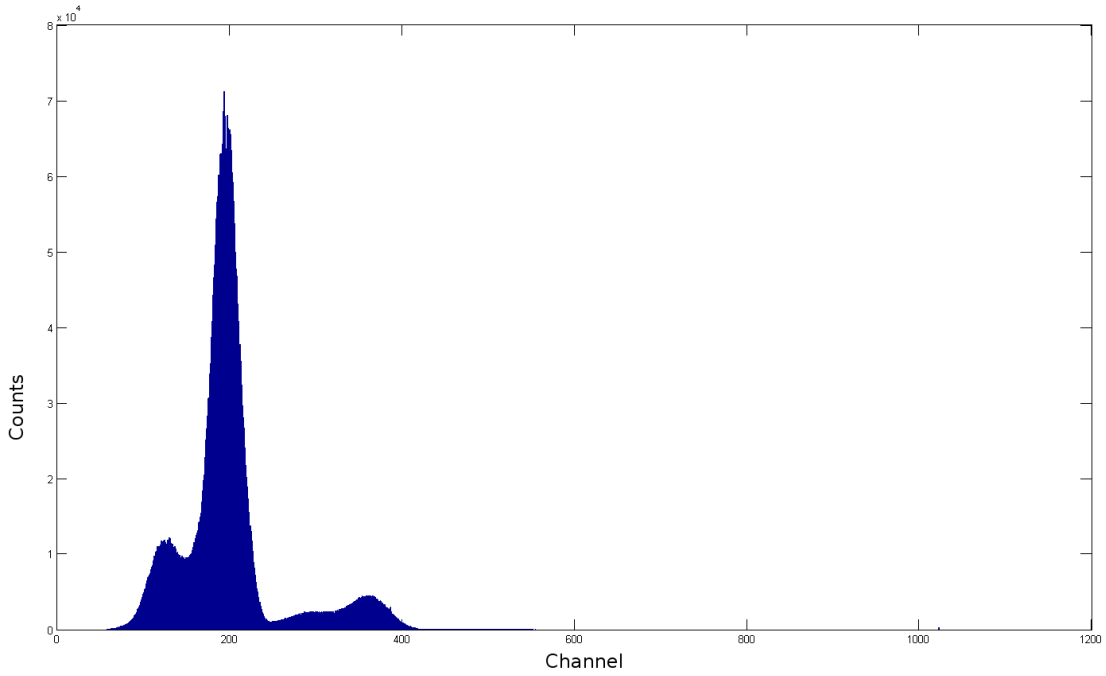


Figure 4.3: Energy plot of an Am-241 source taken by a CZT DAU

4.3 Conclusion

The EGSE test procedure exposed numerous software bugs that were fixed as they were found. With bugs corrected, there is confidence that the configuration process is correct, granted that the pixel map is correct. With this established, it was concluded that error related to trigger width and XA-ASIC address lay outside of the EGSE, and system engineers later resolved these issues.

5 Structural Thermal Module (STM) Test

This chapter discusses some tests an STM version of the MXGS was exposed to. Some tools were required to read out data from these tests. Chapter 5.2 discusses a set of hardware tools used and chapter 5.3 discusses the software used.

The objective of these tools was to read out temperatures during a thermal vacuum test. Initially, the thermal test was scheduled before a vibration test, and the developed hardware was not designed to withstand vibrations. The tests were rescheduled so the system was subject to vibration before thermal vacuum. The developed tools did not work as expected, which is discussed in chapter 5.4 and 5.5.

5.1 Introduction

The ASIM system will be launched into space and mounted onto the Columbus module of ISS. During its lifetime it is subjected to several stress factors. When it is launched, it must tolerate mechanical vibrations, and once it reaches space it is exposed to vacuum. Finally, it experiences sunrise and sunset several times a day while in orbit, which results in mechanical stress due to thermal expansion.

A full-scale dummy model of the MXGS, a Structural Thermal Module (STM), was exposed to similar stress factors. The test took place at the facilities of Instituto Nacional de Téchnica Aeroespacial (INTA), outside of Madrid.

My involvement in the STM test was related to reading out temperature data from the thermal cycling, which took place in January of 2013. This chapter is based on an internal document, [23].

5.2 Hardware

Overview

Hardware mounted in the STM is made specifically for these tests, but the electric and thermal footprint mimicked the flight version. Two separate measurement and read out systems are used. Several sensors are placed all over the MXGS STM and read using a proprietary system supplied by INTA. This system uses one wire for each sensor, thus the connectors limit the number of sensors the system can have.

Extra sensors

Extra temperature sensors are mounted inside a BGO crystal. Their purpose is to measure the temperature gradients in conditions close to what the flight module experiences. The electrical interface between the inside and outside of the vacuum chamber has a limited number of wires, so additional sensors must be digitized and transmitted through a serial interface to a computer. A specially built read out system, "BGO Sensor Board", is developed by UoB and placed inside BGO DAU 1. It has 24 channels made for Pt-100 elements, 8 channels made for accelerometers and 14 general purpose channels. The STM used 7 Pt-100 elements (See figure 5.1).

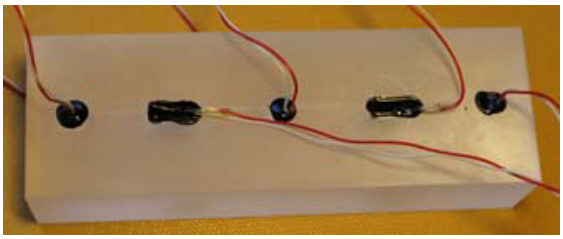


Figure 5.1: Placement of Pt-100 thermal sensors of the BGO STM. Two sensors are on the opposite side

BGO sensor board communication

In differential signalling the information lies in the voltage difference between two wires, and in single ended signalling the information is carried in a voltage relative to a common ground. Contrary to RS232, which is single ended, RS485 and USB are differential signals. If wires are physically close, for example in twisted pair, noise affects both wires equally, thus the voltage differences between them remain.

All cables are bundled together inside the STM, which can result in undesired crosstalk noise. The read out board supports RS232, RS485 and USB, which are selected with jumpers on the circuit board. Of the three, RS485 is preferable. USB, being a bus interface, is more difficult to utilize in computer programs, because there is a negotiation process of when communication can occur. As a result of the negotiation, it is also difficult to analyze data when probed with an oscilloscope.

RS232 to RS484 interface

The laptop that is used to read out temperature data is equipped with an RS232 serial port, thus an RS232 to RS485 interface is needed. The BGO sensor board includes an RS232 to RS485 interface, so a second BGO sensor board is assembled and used as interface. Only the necessary parts of the interface are mounted, and figure 5.2 shows a section of the BGO sensor board circuit where the unrelated parts are removed.

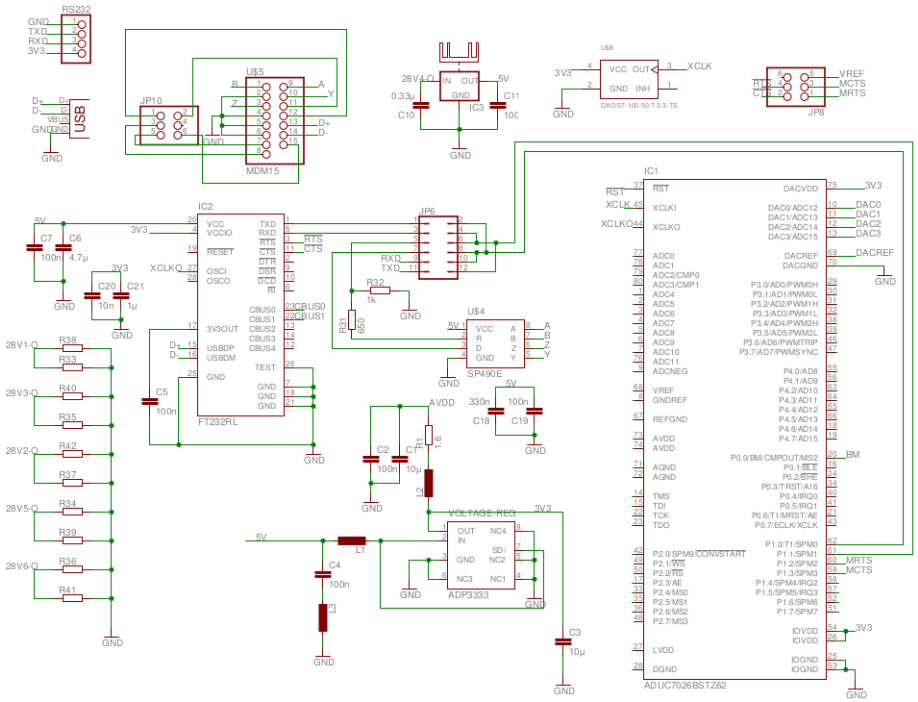


Figure 5.2: RS232 to RS485 interface circuit

A signal diagram of the electric circuit between computer and BGO is shown in figure 5.3. Following the signals, it is apparent that a crossover on JP6 on the BGO side is necessary.

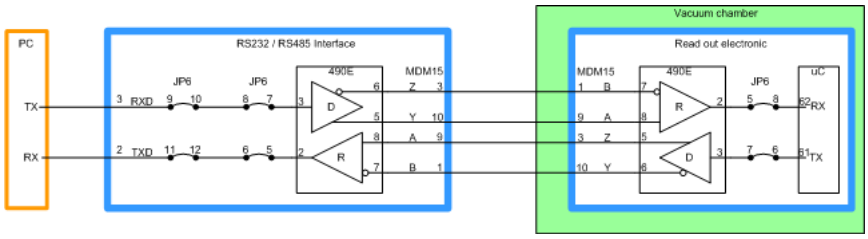


Figure 5.3: RS232 to RS485 signal diagram

5.3 Software

Along with the BGO sensor board, a program that acquires data, converts them to physical values, plots the values and writes them to disk, was made in LabVIEW. Part of the master thesis was to assist with commissioning the UoB made read out electronic for the thermal cycling. It was a requirement that while on the job, all parts of the read out system should be understood and possibly modified in the case of unforeseen events. With no previous experience in LabVIEW, a few days were spent porting the program into a C# program.

The software is made as a C# program, shown in figure 5.4. It has one text box to choose sample rate and one text box to choose the number of samples to mean over. It also has a window where temperatures are plotted, and all channels can be toggled on and off using buttons. The output files are compatible with the existing program.

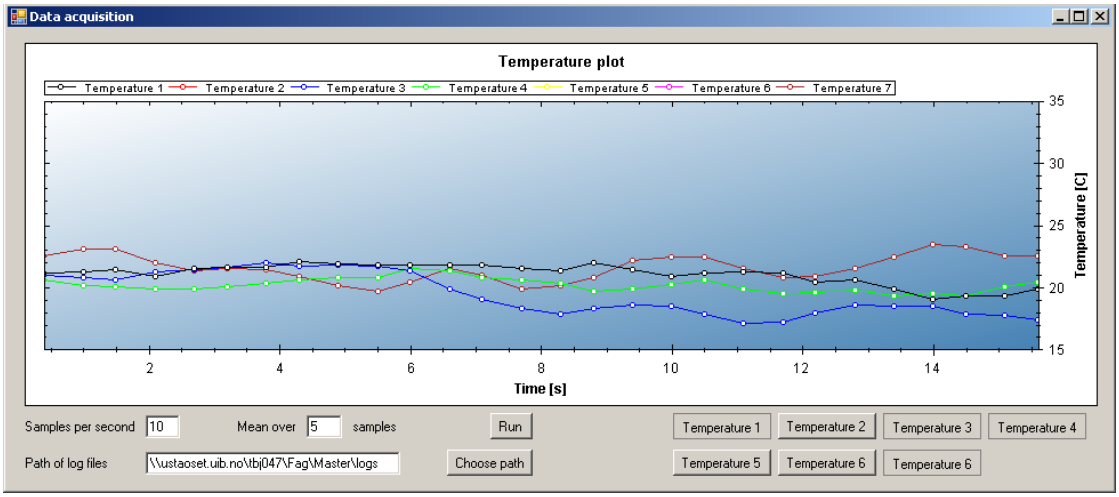


Figure 5.4: Program for reading out temperatures

5.4 Project Commissioning

The original plan was to do the thermal vacuum test before the vibration test. Because of this, the read out electronics was not designed to withstand the vibrations. Late in the process, the tests got rearranged so the vibration test was prior to thermal cycling. This may have contributed to why the system was unable to read temperature data.

Daily overview

Preparations for thermal cycling started January 2. 2013, and by this time the STM module had already been subject to vibration tests. The first two days were used to get the STM into the vacuum chamber, do some measurements of the system, and remove the air. Toward the end of the second day, some preliminary tests were performed where only the BGO related electronics were powered on. The system consumed much more power than estimated, and the test was aborted.

The following day, this matter was discussed with the team of engineers at INTA. It was concluded that it was not possible to power only the BGO the way it was attempted to, and the BGO was never subjected to the power applied the day before. The whole system was powered on, but it showed the dissatisfactory result of an inoperative system.

Diagnostics

Resistance of the BGO read out electronics was measured once again, and found to be in accordance with the DAU STM user manual. It was attempted to use the documentation and follow signals from the BGO all the way to the computer. However, there was no documentation of the internal connections available, so this strategy had to be abandoned. Associate Professor Kjetil Ullaland of UoB and System Engineer Jan Svoboda of Technical University of Denmark (DTU) were consulted in an attempt to locate the error.

A cross-over DSUB cable was made at the UoB, but the female counterpart at INTA did not exist. A new connector had to be soldered on site, but wire A was accidentally soldered to wire A, and so forth. In practice this means that transmitter is connected to transmitter and receiver is connected to receiver. This mistake was corrected, but still no data was received.

Svoboda discovered that the inverted and non-inverted differential cables were swapped. This was also corrected for, but the system was still inoperative.

The following weekend was a special weekend for the Spanish people, and no on site activity was scheduled until Tuesday. INTA, being a military facility, made it impossible to do any work until the staff returned. The return trip was not rescheduled, and unfortunately the problem remained unresolved.

5.5 In retrospect

The trip was of short notice, and there was little time for preparations before departure. Regrettably, the system was never tested with the laptop before hand. It is possible that the circuit was operating, but that the COM-port on the computer was faulty. The BGO board was not designed to withstand vibrations, so it is equally likely that the circuit was malfunctioning.

The microcontroller should be programmed to send a periodic keep-alive signal that could be used for diagnostics. It would be able to disclose whether the transmitter part of the circuit was operating and whether the differential pair was swapped. If the microcontroller had been programmed to respond when unknown commands were received, swapped differential pairs at the receiver could be disclosed as well.

In conclusion, two easy patches to the microcontroller would be enough to determine whether or not the circuit was usable.

6 Summary and Outlook

6.1 Summary

EGSE

The DPU Emulator is rewritten from scratch, with a high focus on scalability, and reusability for future projects. A priority was to get a working system as quickly as possible, before adding extra features. The first step was to get a working system that could be used for simple commanding, and downloading SCDPs. The new DPU Emulator has a text based scripting that allows for advanced commanding of the attached DAUs. All pixels of a CZT detector may be configured from a script, or through a separate XA-ASIC window. This window has a graphical interface for all settings related to all 32 XA-ASICs of a CZT DAU.

The performance of the system is extensively tested. It handles the maximum possible event rate of four DAUs for 3.9 ms and one DAU for 15.6 ms. Events typically comes in bursts and it is unrealistic that the maximum rate persist over extensive periods of time. However, it handles 70% of the maximum rate indefinitely, which is an acceptable performance. Using variable names instead of addresses gives a slight overhead in performance, and the current implementation sends approximately 260 commands per second.

From the benchmark tests it is apparent that the current architecture is not able to cope with the timing constraint of reading all binning data each second, thus some architectural changes to the DPU Emulator Interface are proposed. Calculations suggest that the proposed solution can send over 50000 commands per second, but with the cost of increased complexity.

A CZT test procedure is discussed. This procedure was used to eliminate bugs in the DPU Emulator software. When the correct EGSE behavior was established, the DPU Emulator was used to test, diagnose and debug the CZT detector.

STM Test

An STM version of the MXGS was subject to vibration tests and thermal vacuum. Extra temperature sensors were placed on the BGO to characterize the thermal gradient in conditions similar to the real environment. These sensors had to be digitized and transmitted serially because of a limited number of wires.

A separate system developed by the UoB digitized the temperature of seven Pt-100 elements. A computer program reads the temperatures, writes them to the harddisk and. It is also capable of plotting the temperatures directly in the user interface.

The thermal vacuum test was initially planned to be prior to the vibration tests, but later the tests were rearranged. At that time the system was finished and not qualified for vibrations. During the first week of January, the thermal vacuum test started. The system did not respond, and it is believed that the reason is because of the vibrations, or because of a faulty serial communication port on the laptop.

6.2 Conclusion

The first lines of EGSE code was written in mid August, and by mid September a proof of concept existed. It had the most basic functions, and it was able to command a DAU from a simple script language. Since then the program has been used in testing and debugging of the BGO and CZT detectors.

Compared to the old solution, the EGSE now has increased robustness, consistent behavior and better performance. A wide diversity of scripts may be written and executed, giving the test team increased productivity and flexibility.

The requirements for the EGSE was that it 1) used a Opal Kelly XEM3001 as interface between computer and DAUs, 2) is capable of high speed data acquisition and low speed commanding, 3) has script based commanding language, 4) is scalable, 5) is easy to modify and expand. If time allowed it, it should be adapted to future projects.

From the start it has been focused on scalability in the solution. In practice, this means that the source code is coded properly without tweaks and hacks. To make it easier to read and modify, the source code is split into several classes which makes it easier to get an overview of the code. Throughout the project, it

has also been of great importance to add describing comments continuously while coding.

As a conclusion, the EGSE satisfy the requirements from chapter 3. It is also possible to adapt it to other projects, but some modifications might be needed in the source code. The software replaced the LabVIEW implementation for simple tasks after a few weeks of development. It has been thoroughly tested, and used for several months. It is capable of handling four DAUs simultaneously, and proved to be reliable.

6.3 Outlook

The ASIM project is moving toward the final phase of development. The weeks and months following this project is spent on sorting out the few remaining issues, characterizing the detectors and building a pre flight module. Through out this process, the EGSE will be exhaustively used.

A Operation Manual

A.1 Installation Procedure

Dependencies:

1. Microsoft .NET Framework 4 or newer
2. Opal Kelly FrontPanel version 4.0.8

The software has not been tested with newer versions of .NET 4, but in general this should be backwards compatible. Opal Kelly changed their driver namespace, so newer versions of FrontPanel will not work.

Prior to plugging the Opal Kelly device into the USB slot, the Opal Kelly FrontPanel version 4.0.8 should be installed. After installation of the Opal Kelly driver, the device should be attached to the computer to complete the installation. Once the installation procedure is completed and the dependencies are met, the program may be started.

A.1.1 Hardware Connections

The EGSE equipment is designed to be operated with up to four DAUs at the same time. The DAUs should be connected to the DPU Emulator Interface using LVDS signalling. The FPGA cannot output an LVDS signal, so this has to be made electronically outside the FPGA (See figure A.1).

Attach the DPU Emulator Interface to a computer with a USB cable, and DAUs to the DPU Emulator Interface. Table B.1 and B.2, as well as figure B.1 and A.1, has the information needed to connect the DAUs.

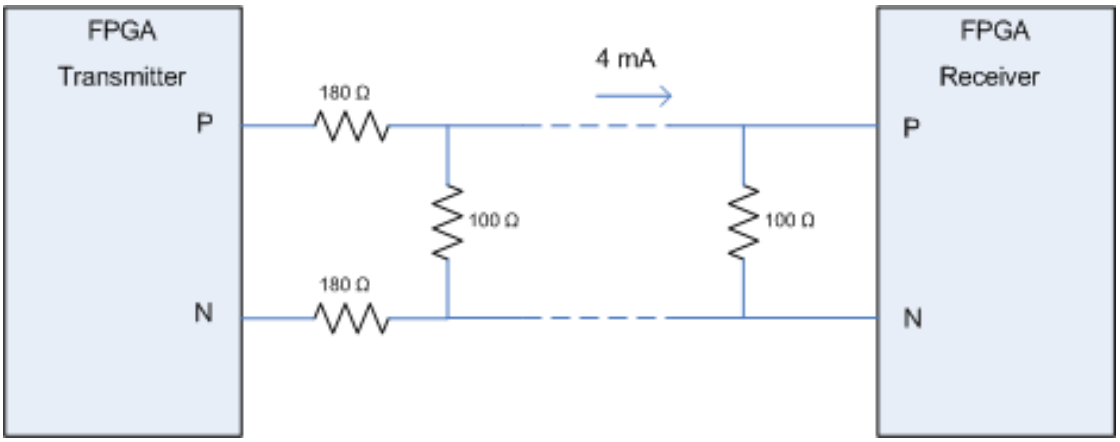


Figure A.1: Converting from differential output to a LVDS compatible signal.

A.1.2 First Time Start

Make sure to follow the installation procedure prior to starting the EGSE program. For the program to start successfully, the libFrontPanel-csharp.dll and libFrontPanel-pinv.dll must be present in the same folder as the EGSE.exe file. If they are present, the program will start and show the "Communication Window", which is the window used to communicate with the DAUs.

No files are loaded automatically the first time the program starts, but the software offers some automation during start-up. In the settings dialog, three XML-files can be selected to be loaded at start-up, as well as a firmware that is uploaded every time a connection to the DPU Emulator Interface is established.

A.1.3 Connecting to DPU Emulator Interface

The program attempts to connect to a DPU Emulator Interface at program launch. If no DPU Emulator Interface is attached at runtime, the connection must be established manually. This is achieved with the "Tools $\rightarrow$ connect to Opal Kelly"-menubar button. An information box informs whether the connection was successful or not.

The Opal Kelly XEM3001 uses an SRAM-based Xilinx FPGA. This means that the HDL-code is lost when the power is lost, i.e. when it is disconnected from the computer or when the computer is turned off or rebooted. If a firmware is configured in the settings dialog, the program attempts to upload this configuration once connection to the DPU Emulator Interface is established.

If the Firmware version in the statusbar shows firmware version 65535, no firmware is loaded into the FPGA. To upload a firmware, the "Tools → Program FPGA"-menubar button is used. Once clicked, an "Open File Dialog" opens, which is used to navigate to the desired .bit-file. When the bit-file is uploaded, the status bar should be updated to "Firmware version: 23". (Firmware version as of 01.06.13).

A.2 Windows

A.2.1 Configuration Window

A predefined configuration file is shipped with the program, mxgs.xml. To load this file, open the "File"-menu of the menu bar, and click the "Load" button.

In the register table of the Configuration Window, the names can be general or specific. The more specific the register name is the higher priority it has. In general, the register name can be of the following form:

[DAU Name/Type].[Module Name/Type].<Name>

[DAU/Type].<Name> is not a valid input. If DAU Name/Type is given, then Module Name/Type has to be supplied as well. Items in square brackets are optional and items in angle brackets are mandatory. DAU name takes priority over DAU type, and module name takes priority over a module type. As an example, BGO1.PMT_IF.CR0 takes priority over BGO.PMT_IF.CR0 because BGO1 is a specific DAU, while BGO refers to all BGO detectors.

In the shipped XML-file there is a set of default registers. Control registers CR0 to CR3 is assigned offset address 0 - 3, and status registers SR0 to SR3 is assigned offset address 4 - 7. A command referencing CR0 to CR3 or SR0 to SR3 results in a valid address regardless if the referenced module has these registers. This behavior is changed by removing the general registers, and manually add all registers of all modules.

The priority mentioned above applies to the bitfields table as well. DAU Name/-Type is mandatory for bitfields, and separated into its own column.

A.2.2 Communication Window

Scripting

The script functionality and its implementation is very simple. It has no syntax check and does not check that the variables are valid. Appendix B.3 has a quick reference of the available script commands.

The general syntax for writing commands to a DAU is:

mwc <DAU> <Module> <Register> [<Data>]
mrc <DAU> <Module> <Register>

DAU, module, register and data can all be given as a reference to a variable or as a numerical value. If numerical values are to be used, they can be given as decimal, binary or hexadecimal numbers. Binary and hexadecimal numbers start with "0b" or "0x" respectively.

If the data field is left empty, the program generates a default string. The following table is CR3 of a photomultiplying tube, and taken from the BGO documentation:

0x0003	CR3	ADE	7	Enable ADC	1
		SMP	6	Enable continuous sample mode	0
		TCE	5	Enable Tail cancellation	1

Table A.1: BGO, photomultiplying tube interface, control register 3

This table shows that the default bit string would be "10100000". If one want to override a value in the script, the data field can be e.g. ADE=0. The value can be given in decimal, hexadecimal or binary form. If more than one field is to be overridden, the data field takes a comma-separated list of bitfields to override.

As a final example, this is a valid data value that would send 01100000:

ADE=0b0,SMP=0x1

The script implements an easy for loop. A for loop can be used to send the same command sequence to several modules or DAUs, and the general syntax is:

for <variable> in <start> to <end>
  <code>

done

To use a loop variable, it should be referenced with a '$' sign before the variable name. The start number may be lower, higher or equal than the end number and if they are equal, a single iteration is executed. All code encapsulated from "for" to "done" are evaluated as stand-alone scripts, where all variables are replaced by the iteration value. As an example, the following code:

```
for i in 1 to 3
  for j in $i to 1
    echo $j
  done
done
```

Produce the following output to the status window:

```
1
2
1
3
2
1
```

The script also incorporates VHDL-style commenting. Text appearing after "--" is regarded as comments and filtered out. To display text to the status window, "echo" or "print" may be used, as in the example. In addition to printing to the status window, "print" also writes the text to the log file, if specified in the settings window.

To make the script sleep, the "sleep" command is used, where the argument is the number of milliseconds to sleep. The "wait" wait for the time synchronization pulse before it continues to process the script.

It is also possible to configure all pixels of the XA-ASIC Window through the script. The XA-ASIC Window must have been opened at least once prior to configuring pixels. It is not yet possible to upload the configuration, and the script only sets the pixel values. The general syntax is:

set <read out chain> <detector module> <pixel> <value>

The name convention in the script is broken compared to the design report, and read out chains are enumerated from 0 to 3. Furthermore, detector modules are

given relative to the read out chain, going from 0 to 3. The same applies to pixels, which are enumerated from 0 to 255. Per default, pixel correspond to channel number, but one can choose to address logical pixels by using a capital 'L' in front of the pixel address. Pixel value may be either a voltage offset (-7 mV to 7 mV), "on" or "off", or '+' or '-', where '+' and '-' increment or decrement by one millivolt respectively.

Retrieving Science Data

This explains how to retrieve science data, but it is assumed that the DAUs are already configured correctly and the DPU Emulator is connected to the DPU Emulator Interface.

To retrieve data, make sure the DPU Emulator Interface FIFO is enabled in the "Tools"-menu. If there is a checkmark next to the "FIFO Enabled"-button, the FIFO is enabled. It should be noted that this button is independent of the "Start Acquire" button in the Communication Window. Data is not relayed into the FIFO when the menu button is deactivated, thus no data is downloaded.

Asserting the "Start Acquire" button clears the FIFO before acquiring data. If there is data in the FIFO that should not be discarded, the FIFO must be manually dumped prior to asserting "Start Acquire". This is achieved by operating the "Tools $\rightarrow$ Dump FIFO"-menubar button.

When no more data is needed, the "Start Acquire" can be de-asserted to stop downloading science data packets. The program can also be configured to automatically stop downloading data. It can stop after it has downloaded a specific number of SCDPs or after a given duration.

Example scripts

This example shows how to send the same command to all read out chains of 4 DAUs, granted that CZT1 to CZT4 is defined in the "Detectors" table of Configuration Window.

```
for dau in 1 to 4 -- Loop over all DAUs
  for roc in 1 to 4 -- Loop over all read out chains


    -- Set clock division factor to A0
    mwc CZT$dau XA_CFG_$roc CR1 CLKDIV=0xA0
```

```
  done
done
```

This example disables all pixels of the entire CZT DAU. *for roc in 0 to 3 --*

```
Loop over all read out chains
  for dm in 0 to 3 -- Loop over all detector modules
    for pixel in 0 to 255 -- Loop over all pixels
      -- Disable pixel
      set $roc $dm L$pixel off
    done
  done
done
```

A.2.3 XA-ASIC Window

This window is used to configure a CZT DAU. It has one tab for each read out chain, and one groupbox for each detector module. A detector module groupbox has 256 pixels, and the configuration for two XA-ASICs. Hovering the mouse above a pixel, shows its channel address.

The settings of the window can be saved to, and loaded from, an XML file. In the settings dialog, an XML file can be entered to automatically load a configuration upon start. The value of the comments field in the bottom right corner is saved in the XML configuration file.

When the window first starts, a number in each pixel shows the channel address. Once the a button is operated, either left or right click, the channel address is replaced with the new value. This also happens when an XML-file is loaded.

All pixels can be toggled by clicking them. A right-click on a pixel opens a menu that is used to set the pixel specific threshold voltage.

To upload the configuration, the correct DAU must be chosen in the dropdown box in the bottom right corner, and the read out chains to configure must be checked. Once completed, click the "Configure"-button to start uploading the configuration.

If new CZT DAUs are added to the "Detectors" table of the Configuration Window, the XA-ASIC Window must be closed and opened for the dropdown box to update.

A.2.4 Image Window

This window can be used to plot data acquired from a CZT. When the data is loaded, it searches for the pixel with the most hits. All other pixels are then scaled relative to this pixel. A side effect of this is that there is always at least one pixel that is "noisy".

Pixels that are white have high activity, and pixels that are black have low activity. The window has a user configurable gamma correction factor. When the pixel activity is similar, this may be used to distinguish shades of gray. The image can not be exported as an image.

This window is not robust, and data files containing both CZT and BGO data should not be loaded into the window. It does not distinguish between DAUs, so if data is acquired from two different DAUs, the images are superimposed.

A.3 General operation of EGSE

Changing files

If there are any modifications to the tables in Configuration Window while running the program, the files are not saved automatically, and the program does not ask upon closing whether or not to save the changes before closing.

Adding and removing DAUs

To reduce noise, unused port on the DPU Emulator Interface is disabled by default. Once the DPU Emulator has uploaded the firmware, or once it is connecting to the DPU Emulator Interface, ports that are assigned a DAU address in the "Detectors" table of the Configuration Window is enabled. The DPU Emulator Interface has one LED reserved for each port, and if the port is enabled, the belonging LED is activated. When data is received from a DAU the corresponding LED blinks. The LED closest to JP2 is DAU 0, and the LED closest to JP3 is DAU number 3. If DAUs are added or removed from the table while the program is running, one must sometimes move around in the table before all ports are enabled.

Test mode

The DPU Emulator Interface can be used to test both long and short words. Test mode is enabled through the "Tools"-menu, where it has its own submenu. This menu has three buttons, "Internal test mode", "LnS" and "Burst", where LnS means "Long not Short". There is also a submenu, "External loopback", that has "Force short" and "Force long".

When internal test mode is enabled and LnS is disabled, short words are relayed back. This can be used to see if commands are sent and retrieved reliably. If internal test mode and LnS is enabled, the DPU Emulator Interface start to produce SCDPs at a rate corresponding to four saturated data links. The burst button produces SCDPs for approximately 1 - 2 ms.

The pattern of the generated SCDPs can be controlled with the script. By sending a command with internal test mode enabled, the pattern of the first 24 bits of a SCDP is set. These bits get the value of the sent command. The last 20 bits of the SCDPs is a packet serial number.

By default, the DPU Emulator Interface sets the Long-not-Short flag of the xlink transmitters automatically from the header bits. This behavior can be overridden from the "external loopback" menu, which is useful to provoke a transmission error.

FIFO

If the FIFO is disabled from the tools menu, the DPU Emulator Interface stop relaying packages into the FIFO. As a consequence, no data is received until the FIFO is activated. Content of the FIFO can be manually deleted using the "Tools $\rightarrow$ Reset FIFO"-menubar button, or manually downloaded using the "Tools $\rightarrow$ Dump FIFO"-menubar button.

Error counter

An error counter counts the error reported by the xlink receivers. It does not count the number of words that are discarded because of a full FIFO. This counter is reset using the "Tools $\rightarrow$ Reset error counter"-menubar button.

Reset firmware

All internal signals of the DPU Emulator Interface is reset when the "Tools → Reset"-menubar button is operated. This means that the content of the FIFO is deleted, that the error counter is reset, that all DAU ports are disabled and that the FIFO is disabled. The software automatically re-enables the DAU ports, but not the FIFO.

B Quick reference

B.1 Physical I/O

DAU	SIGNAL	POLARITY	FPGA PIN	JP2 PIN
DAU 0	Data out	P	10	8
		N	11	11
	Strobe out	P	12	12
		N	13	13
	Data in	P	15	14
		N	16	15
	Strobe in	P	18	16
		N	19	17
	1 MHz Clock	P	7	6
		N	20	18
DAU 1	Data out	P	34	30
		N	35	33
	Strobe out	P	36	34
		N	37	35
	Data in	P	39	36
		N	40	37
	Strobe in	P	42	38
		N	43	39
	1 MHz Clock	P	31	28
		N	44	40

Table B.1: This table shows the pin coding on the FPGA and the associated JP2.

DAU	SIGNAL	POLARITY	FPGA PIN	JP3 PIN
DAU 2	Data out	P	138	21
		N	139	18
	Strobe out	P	140	17
		N	141	16

Continues on next page...

...Continued

DAU	SIGNAL	POLARITY	FPGA PIN	JP3 PIN
DAU 2	Data in	P	143	15
		N	144	14
	Strobe in	P	146	13
		N	147	12
	1 MHz Clock	P	135	23
		N	148	11
DAU 3	Data out	P	114	43
		N	115	40
	Strobe out	P	116	39
		N	117	38
	Data in	P	119	37
		N	120	36
	Strobe in	P	122	35
		N	123	34
	1 MHz Clock	P	111	45
		N	124	33

Table B.2: This table shows the pin coding on the FPGA and the associated JP3.

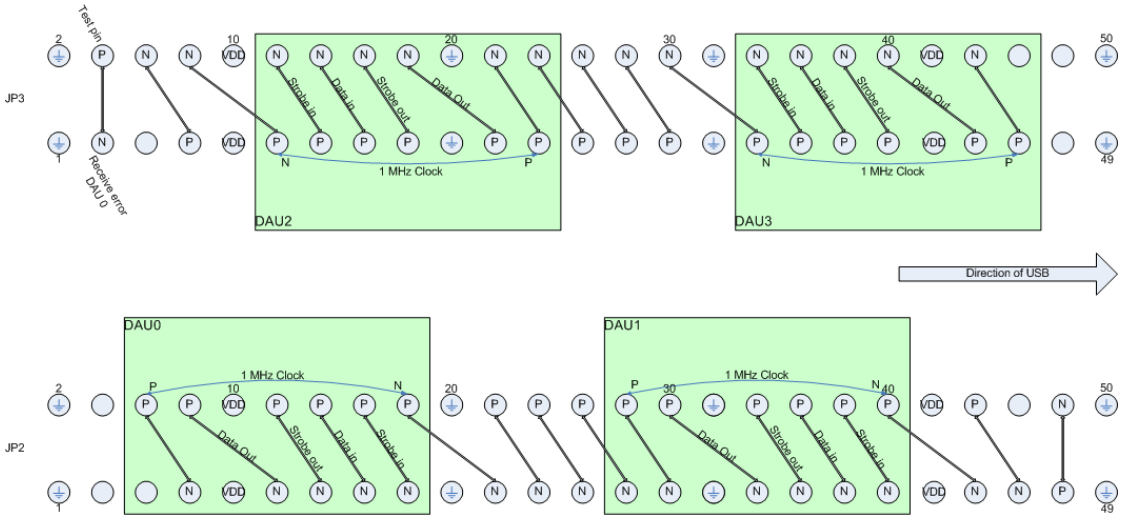


Figure B.1: Graphical representation of the pinning

B.2 Virutal I/O

Logic	Function	Addr	Bit	Description
General		*Wire Out*		
	Version	0x24	15:0	Firmware version
	TCP reg	0x23	7	TCP register
		Trigger In		
	Reset	0x40	2	System reset
	Reset TCP	0x40	9	Reset TCP register
Transmit		*Wire In*		
	Data	0x10	15:0	<2 flag bits><14 addr bits>
		0x11	15:0	<000000><2 DAU addr bits><8 data bits>
	Force LnS	0x13	7	If 0, length is detected from flag bits. If 1, LnS from test logic sets length
		Trigger In		
	Start	0x40	0	Start transmitting data to DAU addressed in 0x11
Receive		*Wire In*		
	Enable DAUs	0x13	6:3	Enable DAU 3 - 0 respectivley
		Wire Out		
	Error count	0x25	15:0	Number of errors received
		Trigger In		
	Reset errors	0x40	8	Reset error counter
Router		*Wire In*		
	Enable FIFO	0x13	2	Enable relaying of SCDPs
		Wire Out		
	Short word	0x20	15:0	Same as Wire In 0x10
		0x21	15:0	Same as Wire In 0x11
	Short ready	0x23	4	Short word ready to be read
	Stop	0x23	0	Router is stopped from relaying SCDPs to FIFO

Continues on next page...

...*Continued*

Logic	**Function**	**Addr**	**Bit**	**Description**
Router	Ack.	*Trigger In* 0x40	7	Acknowledge short word
FIFO	Threshold	*Wire In* 0x12	13:0	Set FIFO threshold. When above, router is stopped.
	Data	*Wire Out* 0x22	13:0	Short word ready to be read
	count			
	Above	0x23	1	FIFO is above threshold
	Empty	0x23	2	FIFO is empty
	Start	*Trigger In* 0x40	4	Reset FIFO (wipe content)
	Read	*Pipe Out* 0xA0		Read content of FIFO. 16 bits word width. 16 MSB sent first, and 16 LSB sent last. Byte order is little endian.
Test	Enable	*Wire In* 0x13	0	Enable test mode
	LnS	0x13	1	If test mode and LnS are 1, SCDPs are sent to FIFO. Format: <Flag bits ('00')> <Wire In 0x14> <data bits of 0x15><'00'> <serial number>
	Pattern	0x14	15:0	Same as Wire In 0x10
		0x15	15:0	Same as Wire In 0x11

Continues on next page...

...Continued

Logic	**Function**	**Addr**	**Bit**	**Description**
Test		*Wire Out*		
	LnS	0x23	5	Loopback of LnS
	Enable	0x23	6	Loopback of enable
		Trigger In		
	Load	0x40	6	Load test pattern. If enabled and LnS, pattern of SCDP is changed. If enabled and not LnS, a short word is sent.

Table B.3: Virtual input and output from the computer to the XEM3001

B.3 Script commands

Command	Description
mwc <DAU> <Module> <Register> [Data]	Sends a memory write command to a DAU
mrc <DAU> <Module> <Register>	Sends a memory read command to a DAU
mddp <DAU> <Module> <Register> [Data]	Sends a mddp packet to a DAU. This is only usefull in test mode or with an external loopback.
scdp <DAU> <Module> <Register> [Data]	Sends a scdp packet to a DAU. This is only usefull in test mode or with an external loopback.
sleep <Delay>	Puts the script to sleep. Argument is in milliseconds.
wait	Wait until the next TCP before proceeding.
echo <Text>	Prints the argument to the debug window.
print <Text>	Prints to both debug window and log file.
debug <Text>	Sends text to the debugger.
set <Read out chain> <Detector module> <Pixel> <Threshold>	Set threshold of pixel. Read out chain (ROC) and detector module (DM) is from 0 to 3, where DM is relative to ROC. Pixel is physical address, but L<Address> may be used to address logical pixels. Threshold may be -7 to +7 to set absolute value, "+" and "-" to increase or decrease by one and "on" and "off" to enable or disable.
for <var> in <start> to <end> <commands> done	Loops over all values in the given range. To reference the loop variable, use $<var>.

Table B.4: Quick reference of available script commands

B.4 XA-ASIC configuration shift register

Description	Number of bits
Dlt_Mode	1
Cc_enableb	1
Test_on	1
Thr_enb	1
Test_on2	1
S_outb	1
Reserved	2
B15-B7	9
Reserved	1
Mon_b4	1
Mon_b3	1
Mon_b2	1
Mon_b1	1
Mon_b0	1
Reserved	2
Mbias_b4	1
Mbias_b3	1
Mbias_b2	1
Mbias_b1	1
Mbias_b0	1
Reserved	3
Channel disable	129
Channel trim DACs	4x129 516
Test enable	129
Kota_hp	1
Bufb2_hp	1
Ibuf_hp	1
Prebi_hp1	1
Prebi_hp2	1
Rfast_en	1
Reserved	4
Bias DAC, str_bi	3
Bias DAC, Mbias	4
Bias DAC, TWB	3
Bias DAC, TDB bias	3

Continues on next page...

...Continued

Description	Number of bits
Bias DAC, Vthr global	4
Bias DAC, shabias	3
Bias DAC, ifs	3
Bias DAC, ifp	3
Bias DAC, resw_bi	3
Bias DAC, vthrbi	3
Bias DAC, vthrint_fine	3
Bias DAC, vthrint_coarse	6
Total:	**858**

Table B.5: Overview of the bits in the serial configuration register for XA1.82, listed sequentially from the regin pin [7].

C Pixel mapping

Figure C.1: Corrected pixel mapping of a CZT detector module

Abbreviation

ADC	Analog Digital Converter
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
ASIM	Atmosphere-Space Interaction Monitor
BATSE	Burst and Transient Source Experiment
BGO	Bismuth Germanate Oxide
CPU	Central Processing Using
CZT	Cadmium Zinc Telluride
DAC	Digital Analog Converter
DAU	Detector Assembly Unit
DLL	Dynamic Link Library
DM	Detector Module
DMA	Direct Memory Access
DPU	Data Processing Unit
DTU	Technical University of Denmark
EEPROM	Electrically Erasable Programmable Read-Only Memory
EGSE	Electronic Ground System Equipment
ESA	European Space Agency
FIFO	First-In First-Out
FPGA	Field Programmable Grid Array
GUI	Graphical User Interface
HDL	Hardware Description Language

HED	High Energy Detector
I/O	Input/Output
IDE	Integrated Development Environment
INTA	Instituto Nacional de Téchnica Aeroespacial
ISS	International Space Station
JP	Jumper
LED	Low Energy Detector
LSB	Least Significant Bits
LVDS	Low Voltage Differential Signal
MDI	Multiple Document Interface
MMIA	Modular Multi-spectral Imaging Array
MSB	Most Significant Bits
MXGS	Modular X-ray and Gamma-ray Sensor
NASA	National Aeronautics and Space Administration
PLL	Phase-Locked Loop
PMT	Photomultiplier Tube
RAM	Random Access Memory
RHESSI	Ramaty High Energy Solar Spectroscopic Imager
SCDP	Science Data Packet
STM	Structural Thermal Module
TCP	Time Correlation Pulse
TGF	Terrestrial Gamma-ray Flashes
UoB	University of Bergen
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
XML	Extensible Markup Language

Bibliography

[1] NASA. A Brief History of the Discovery of Cosmic Gamma-Ray Bursts, 1995. Visited: May 2013. `http://apod.nasa.gov/htmltest/jbonnell/www/grbhist.html`.

[2] T. Gjesteland. *Properties of terrestrial gamma ray flashes : modelling and analysis of BATSE and RHESSI data*. PhD thesis, University of Bergen, 2012.

[3] *MXGS Executive Report*, 1dB edition, 14 2011. Internal design document.

[4] *MXGS BGO DAU Design Report*, 3 edition, 4 2013. Internal design document.

[5] *MXGS CZT DAU Design Report*, 3 edition, 4 2013. Internal design document.

[6] E. Caroli, J.B. Stephen, G. Dicocco, L. Natalucci, and A. Spizzichino. Coded aperture imaging in x-ray and gamma-ray astronomy. *Space Sci. Rev.*, 45(3-4):349–403, 1987.

[7] Gamma Medica-Ideas. *XA1.82 DATASHEET*, V1R0 edition, 2012.

[8] IEEE Standard for Heterogeneous Interconnect (HIC) (Low-Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction). *IEEE Std 1355-1995*, pages i–, 1996.

[9] Opal Kelly Incorporated, 3442 SE Ironwood Ave, Hillsboro, OR 97123. *XEM3001v2 User's Manual*, 02 2007.

[10] N.H.E. Weste and D.M. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[11] M.D. McIlroy, E.N. Pinson, and B.A. Tague. UNIX Time-Sharing System. *Bell System Technical Journal*, 57(6):1902, 1978.

[12] E. Girczyc and S. Carlson. Increasing design quality and engineering productivity through design reuse. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 48–53, New York, NY, USA, 1993. ACM.

[13] E. Yourdon. *Object-Oriented Systems Design: An Integrated Approach*. Prentice Hall Professional Technical Reference, 1st edition, 1994.

[14] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[15] J.S. Busby. The problem with design reuse: an investigation into outcomes and antecedents. *Journal of Engineering Design*, 10(3):277–296, 1999.

[16] Microsoft Corporation. Overview of the .NET Framework, 2013. Visited: May 2013. http://msdn.microsoft.com/en-us/library/zw4w595w.aspx.

[17] B.W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *Software, IEEE*, 1(1):75–88, 1984.

[18] Microsoft Corporation. C# language specification, 2010. Visited: April 2013. http://download.microsoft.com/download/0/B/D/0BDA894F-2CCD-4C2C-B5A7-4EB1171962E5/CSharp%20Language%20Specification.htm.

[19] Opal Kelly Support. Opal Kelly Support, Accuracy of PLL, Unofficial, 2010. Visited: April 2013. http://forums.opalkelly.com/archive/index.php/t-974.html.

[20] J. Axelson. *USB Complete: The Developer's Guide*. Lakeview Research, 2009.

[21] Xilinx Inc., 2100 Logic Dr, San Jose, CA 95124. *Spartan-3 FPGA Family Data Sheet*, v3.0 edition, 10 2012.

[22] Opal Kelly Support. Block-Throttled Pipe Out Freezing solution, 2011. Visited: March 2013. http://forums.opalkelly.com/showthread.php?1032-Timeout-behavior.

[23] *MXGS DAU STM User Manual*, 1B edition, 02 2013. Internal design document.