

Ethernet-Based Control System and Data Readout for a Proton Computed Tomography Prototype

A thesis by

Karl Emil Sandvik Bohne

for the degree of

Master of Science in Physics



Department of Physics and Technology

University of Bergen

June 2018

Abstract

At the University of Bergen (UiB), work is underway to develop a proton based computed tomography prototype. Proton CT (pCT) is an alternative to photon-based imaging that shows great promise as a technology for use in proton treatment planning, while also delivering lower doses of harmful radiation than its X-ray based counterpart. Particle treatment planning is currently achieved by performing X-ray-based CT scans, of which the results are used to estimate a particle dose through a translation process. This introduces systemic errors due to the fundamentally different manners in which photons and particles interact with matter. pCT used for dosage planning purposes could eliminate the need for this conversion, allowing for treatment more precise and effective than what is currently possible.

The work needed to realize the complete pCT is extensive, and this thesis is primarily concerned with the control system for the multiple proton CT readout units (PRU) that will be used in the machine. This system will facilitate communication between a control room and the readout units, allowing an operator to determine the status of various system parameters such as power consumption, voltages and temperatures, program PRU peripherals according to a desired configuration, perform system initialization, trigger resets, etc. Such a system may also perform other tasks, such as automatic system-monitoring, or could provide assistance in the data-readout process.

This thesis discusses the requirements of such a system and how it might be realized, details its design, and describes in addition the full implementation of the required PRU field programmable gate array (FPGA) firmware on the current development board. Software for a soft-core processor running a lightweight OS and instantiated in the FPGA fabric is developed and tested successfully; providing serial- and Ethernet communication links via which a board can be controlled and monitored remotely, using a simple platform-independent API. Additionally, a DMA-based solution for data-readout is designed, implemented, and verified to be working by reading out actual detector data.

Other aspects of the system are also discussed, including ways of distributing a synchronized clock and trigger, power-monitoring, and future development. A primer on the workings of a proton CT in addition to particle-/photon matter interaction fundamentals is provided.

Acknowledgements

The work detailed in this thesis would not have been possible without the guidance provided by my two advisors, professor *Kjetil Ullaland* and associate professor *Johan Alme*. A special thanks goes to them, for providing me with invaluable advice and feedback along the way. I also owe *Ola Slettevoll Grøttvik* a great deal for introducing me to the project, answering my questions, for his feedback, and for keeping me occupied throughout the past year. I am also grateful to the pCT group for allowing me to contribute to the project.

Additional appreciation goes to the guys in room 312 for keeping the collective spirit of the group high. Although the office at times more closely resembled an internet-cafe/break-room hybrid than a place of study, these past two years would not have been the same without them. I would also like to thank the fantastic group of people with whom I first enrolled, five years ago, who has made this period not only tolerable, but at times even enjoyable.

I thank my family for their continued encouragement, and last but not least, my loving and supportive Julie, for being who she is, and for tolerating me these past seven-or-so years.

Contents

Abstract	iii
Acknowledgements	v
Acronyms	xv
Glossary	xvii
1 Introduction	1
1.1 Project Motivation and Goals	1
1.2 Thesis Structure	2
2 Computed Tomography	5
2.1 Ionizing Radiation	5
2.2 Interactions of Photons and Matter	6
2.3 Interactions of Particles and Matter	7
2.4 Particle Therapy and Proton CT Motivation	7
2.4.1 Proton CT	8
3 The UiB pCT and the ALPIDE Pixel Sensor	11
3.1 The ALPIDE Pixel Sensor	11
3.1.1 Basic Principles of Operation	12
3.1.2 Pixels	12
3.1.3 Data Transmission Unit	13
3.1.4 ALPIDE - Readout Unit Interface	13
3.1.5 Control Interface and Chip Addressing	14
3.2 The UiB pCT	15
3.3 Existing pCT Systems	15
3.4 Readout Electronics	16
3.4.1 Current Implementation	17
4 The pCT Control System	19
4.1 Features of a pCT Control System	20
4.1.1 RU - Host Interface	20
4.1.2 Board Initialization	21
4.1.3 Provision of House-Keeping Data	23
4.1.4 ALPIDE Monitoring	23

4.1.5	Additional Features and Data Readout	25
4.1.6	The AXI Master	25
4.2	Clock- & Trigger Distribution	26
4.3	The PRU Processor	27
4.3.1	Operating Systems	28
4.4	PRU Software Applications	30
4.5	A Summary of the Previous Sections	31
5	Firmware	33
5.1	Requirements	33
5.2	Implementation	35
5.2.1	Ethernet Subsystem	35
5.2.2	MicroBlaze Configuration	37
5.2.3	UART	37
5.2.4	Monitor Module	37
5.3	Readout of Detector-Data	38
5.3.1	Development-Stage Data Readout	38
5.3.2	Data-Readout in a Complete System	40
5.3.3	Other Considerations	43
6	Control Message Format and Protocol	45
6.1	Requirements	45
6.2	An Application-Level Protocol	46
6.2.1	Packet Format	46
6.2.2	Considerations for Unreliable Interfaces	47
6.2.3	COBS	47
6.2.4	Packet Fields	48
6.2.5	Message Replies	50
6.3	Addressing ALPIDEs via a Peripheral Command	50
6.4	Hardware Offloading of the CRC- and COBS Calculations	51
7	Software	53
7.1	Requirements	53
7.2	Overview	53
7.2.1	Development Principles	54
7.3	Software Structure	54
7.3.1	Control interface	55
7.3.2	Data-Readout	56
7.3.3	Monitoring	57
7.4	ALPIDE Control Module Driver	58
7.5	Data-Exchange Between Threads	58
7.6	Software Configuration	59
7.6.1	LwIP and FreeRTOS	59
7.7	Future development	60
8	System Testing	63
8.1	Host-Side Software	64
8.1.1	API	65
8.2	Testing	66

8.2.1	Testing of Communication	66
8.2.2	Test Bench for the Updated ALPIDE Data Module	67
8.2.3	Test of Data-Readout Solution	68
8.2.4	UDP Packet Loss	70
8.2.5	TCP	72
8.2.6	Testing of the Full Readout Chain	72
8.2.7	Testing of Self-Contained PRU Monitoring	74
9	Conclusion and Future Work	77
9.1	Performance Evaluation	77
9.2	Design Evaluation	78
9.3	Future Work	79
9.3.1	Porting of the Python Software	79
9.3.2	Porting of the Embedded Software	79
9.3.3	Extension of the Readout-System	79
9.3.4	Implementation of Higher-Level Control Software	79
9.4	Conclusion	80
A	Coding Style	81
B	Resource Usage	83
B.1	RAM	83
C	SPAD	85
C.1	Requests and Command Types	86
C.2	Replies	87
C.3	Examples	87
D	Python Framework	89
E	Various	93
E.1	ALPIDE Mask-Application	93
E.2	Documentation and Commenting	95
F	Repository Structure	97
	Bibliography	99

List of Figures

2.1	2D radiography showing source, target and detector setup.	5
2.2	Showing the dose delivered by a beam of photons, <i>modified</i> - and <i>native</i> protons as they pass through tissue.	6
2.3	Comparison of dosimetric planning with protons (top) to photons (bottom) [5].	7
2.4	Typical pCT layout, showing separate tracking planes and calorimeter and the path of a proton.	8
3.1	Block diagram showing the main components of an ALPIDE chip [10].	12
3.2	Showing how a voltage on the input to the analog section cause a hit to be stored in the pixel buffer if it surpasses a threshold while a strobe is applied [10].	13
3.3	Showing the format of ALPIDE broadcasts and uni-/multicast write operations. [10].	14
3.4	Showing the format of an ALPIDE read operation [10].	14
3.5	A model of the UiB pCT Digital Tracking Calorimeter (DTC), with ALPIDE chips in a horizontal stave configuration shown in dark blue [11].	15
3.6	The PRU, showing the most central modules of the currently planned design.	18
4.1	The pixel-matrix addressing scheme [10]. Writing 0xFFFF to the address formed by the region selector field set to 0b1111 and the row bit and both column bits set would for instance mask/clear all pixels, depending on the pixel configuration register.	21
4.2	An external sense resistor, current-sense amplifier and ADC allows for monitoring of chip currents and -voltages without occupying the AXI control module.	24
4.3	Illustrating the low likelihood of non-overlapping strobe-windows. . .	26
4.4	MicroBlaze architecture, showing the optional features grayed out [19].	27
4.5	Program flow in an RTOS-based system.	28
5.1	Simplified diagram showing modules that relate to the control system and development-stage data readout, as implemented on the FPGA on the VCU118 board.	34
5.2	The TCP/IP- / OSI stacks and overlap.	35

5.3	Blocks central to Ethernet-functionality, as implemented on the VCU118 FPGA.	36
5.4	Chips that are closer to center of a detector layer receive the majority of the hits.	38
5.6	An example AXI-stream transaction.	40
5.7	Offload system showing data offload modules, buffering stages, arbiter and UDP/TCP cores.	42
7.1	Central threads in the MicroBlaze application.	54
7.2	MicroBlaze application - UART-task flow-chart. The TCP task is similar, but is not required to verify encoding or CRCs.	55
7.3	MicroBlaze application - Data-readout task flow-chart.	56
7.4	Illustrating the OPC UA information model.	61
7.5	A possible control system for the pCT.	62
8.1	Testing setup showing VCU118 board, FPGA Mezzanine Card (FMC), and ALPIDE carrier.	63
8.2	Core host-side software elements. Blocks are implemented as objects with the exception of the Utilities-block.	64
8.3	Print-out of the serial output as the PRU is assigned an address via DHCP, and as it receives a connection.	65
8.4	UDP throughput as a function of PDU size on a 100 Mb link.	69
8.5	UDP throughput as a function of PDU size on a 1000 Mb link with Jumbo frames enabled.	70
8.6	Showing loss over UDP with datagrams sized at 8972 B.	71
8.7	A DMA transfer of ALPIDE data, as recorded by the internal logic analyzer.	72
8.8	A print-out of an ALPIDE data-stream, formatted as PRU words by the ALPIDE Data Module (ADM).	73
8.9	Showing a snippet of control-data read out while transfers are ongoing.	73
8.10	Showing a gap of eight cycles between assertion of tready on the DMA receive-side.	74
8.11	Intermittent assertion of tready on the DMA engine.	74
8.12	Debug-interface print-out of the embedded software reporting exceeded thresholds.	75
E.1	The pixel-matrix addressing scheme [10].	93
E.2	The pixel-matrix [10].	94
F.1	The MicroBlaze subsystem as it was implemented on the VCU118 platform.	98

List of Tables

3.1	Illustrating some basic specifications of previously developed pCTs [13].	16
3.2	ALPIDE readout-data format before processing on the PRU [10].	16
5.1	Xilinx DMA v7.1 figures at 100 MHz [33].	41
5.2	Control register	43
5.3	FIFO depth and -threshold register	43
5.4	bytes-offloaded counter register, 32msb	43
5.5	bytes-offloaded counter register, 32lsb	43
5.6	FIFO-overflow counter register	43
6.1	Application level protocol - Base packet format.	46
6.2	Application level protocol - Header and trailer, respectively.	46
6.3	Consistent overhead byte stuffing example.	47
6.4	A packet appended with a 16 bit CRC and byte-stuffed with COBS.	48
6.5	Payload - format for register writes- and reads, respectively.	48
6.6	Payload - ALPIDE register writes and reads, respectively.	49
6.7	Payload - ALPIDE broadcast opcodes.	49
6.8	Application level protocol - Reply packet.	50
7.1	Showing the format of an MQTT packet.	60
8.1	Test results with the updated data module.	67
8.2	PRU Headers.	73
B.1	Resource usage of components included in the MicroBlaze subsystem	83
C.1	Application level protocol - CMDTYPes	86
C.2	Payload - format for register writes- and reads respectively, and values of the WR/RD-field.	86
C.3	Payload - ALPIDE register writes, -reads, and opcodes, respectively.	86
C.4	Payload - The <i>special</i> command type	87
C.5	Reply-data ACKs/NACKs	87
C.6	Example write-request and response	87
C.7	Example read-request and response.	88
C.8	Example ALPIDE write-request and response, and contents of payload.	88
C.9	Example ALPIDE read-request and response.	88

Acronyms

ACM ALPIDE Control Module	LVDS Low Voltage Differential Signaling
ADC Analog to Digital Converter	LwIP Lightweight IP (software library)
ADM ALPIDE Data Module	MAC Media Access Control (-address)
ALICE A Large Ion Collider Experiment	MEB Multi Event Buffer
ASIC Application Specific Integrated Circuit	MQTT Message Queuing Telemetry Transport
AXI Advanced eXtensible Interface	MTU Maximum Transmission Unit
BRAM Block RAM	NIC Network Interface Controller
BSP Board Support Package	OB Outer Barrel
CERN European Organization for Nuclear Research	pCT Proton CT
CMS Compact Muon Solenoid	PDU Protocol Data Unit
COBS Consistent Overhead Byte Stuffing	PRU Proton CT Readout Unit
CRC Cyclic Redundancy Check	RISC Reduced Instruction Set Architecture
CT Computed Tomography	SCADA Supervisory Control and Data Acquisition
DMA Direct Memory Access	SCU System Control Unit
DTC Digital Tracking Calorimeter	SDK Software Development Kit
DTU Data Transmission Unit	TCP Transport Control Protocol
FIFO First-in, First-out (queue type)	UART Universal Asynchronous Receive Transmit
FMC FPGA Mezzanine Card	UDP User Datagram Protocol
FPGA Field Programmable Gate Array	UiB University of Bergen
IB Inner Barrel	
IP Internet Protocol / Intellectual property (-core)	
ISR Interrupt Service Routine	
ITS Inner Tracking System	

Glossary

8b10b encoding Encoding scheme that maps 8 bits of data into a 10 bit symbol in order to provide DC-balance and facilitate clock-recovery.

Datagram In the context of UDP: The PDU of the UDP protocol.

Jumbo frame An Ethernet frame with more than 1500 bytes of payload, up to a maximum of 9000.

Manchester encoding Encoding scheme where each bit is encoded as either a high- followed by a low value, or vice versa.

SNMP Network management-/monitoring protocol for collecting information on managed networked devices.

Struct In a C-context: an object-like data type with an arbitrary number of fields of arbitrary type.

Super-loop Large infinite loop in software containing an application or subroutine.

TCP/IP A collection of protocols that together make up the Internet Protocol Suite.

Introduction

The use of particle therapy in clinical medicine is growing; in 2016 alone, more than 170000 patients were treated with particles, with the majority of these receiving treatment using protons [1]. The dose a patient is to receive must be determined beforehand, and this is currently calculated based on the results of conventional X-ray CT scans. This introduces inaccuracies which reduce effectiveness, and can lead to long-term health issues due to accidental radiation of healthy tissue causing secondary cancers. With pCT, this translation process could be eliminated, leading to more effective treatment while also reducing unnecessary exposure to radiation.

In 2016, the University of Bergen received funds to facilitate development of a pCT prototype. The prototype will consist of several layers of CERN-developed pixel detectors, originally designed for use in the ALICE experiment. These are sensitive to impacting particles and photons, and will enable precise reconstruction of the tracks and residual energies of individual protons, which are both required in order to perform dose planning.

1.1 Project Motivation and Goals

Some work towards a pCT prototype has been completed: In 2016, the feasibility of using a digital tracking calorimeter for both tracking and residual-energy measurements of individual protons was verified, and in 2016 and -17, a block design for a prototype was laid out, parts of the electronics for the readout-units designed, and the detector-chips themselves selected and partly tested.

Several of these chips will interface to specially designed readout boards; the PRUs. Each board will manage several pixel detectors, and contain data processing modules that handle the data these produce, firmware that allows this data to be read out, and other auxillary modules. It is also likely to expand further as development progresses. A system is required that allows for the configuration, monitoring and control of these modules. Although considerable work remains before the pCT is realized, this thesis primarily discusses the requirements- as well as the implementation of such a control- and monitoring system. One of the goals was to design a

system that was sufficiently general for it to be of use in similar projects in the future, and that could easily be extended or built upon as the pCT project progresses.

Initial PRU development was done on a CERN-developed readout board, but has since migrated to a new Xilinx FPGA platform. No control interface existed for the new board, and neither did a method for readout of produced detector data. A subsystem that could perform this task was also needed so that elements of the design could be tested.

Finally, as the project is still in its relatively early stages, the details of the modules that will be included in the system are not fully decided. Alterations to the existing design that could streamline and simplify the development process could possibly be made, and this was also explored.

1.2 Thesis Structure

Chapter 2 - Computed Tomography and Particle/Photon - Matter Interactions

This chapter describes the workings of both conventional- and proton based CT. How particles and photons interact with matter, how they deposit energy as they do so and how this is used to form images is shown. Some of the promising aspects of pCT are discussed.

Chapter 3 - UiB pCT and the ALPIDE Pixel Sensor

This chapter describes the pCT in development at UiB, including a description of the pixel detectors used in the project, as well as the overall structure of the prototype, and how it differs from existing pCTs.

Chapter 4 - A pCT Control System

This chapter explains the need for a system that can be used to control the various components of the pCT. It defines the roles such a system is required to fill, any additional tasks it might perform, how these might change as the project develops, and outlines how such a system should be implemented.

Chapter 5 - Firmware

This chapter discusses the firmware necessary in order to provide the functionality described in chapter 4. With the exception of a module that requires external system components that are not yet available, this firmware was fully implemented as part of the work performed during this thesis, and this process is detailed. In addition, a DMA-based solution suitable for detector-data readout during the development stage was developed, and this is also described. Additional aspects of this process that will become relevant as the project evolves are discussed.

Chapter 6 - Development of an Application Level Protocol

This chapter develops a simple application-level protocol for transferring arbitrary data between two parties, which can be used to provide access to memory mapped modules on a device. It also defines optional additions to control the pixel detectors as well as the master-module responsible for carrying out the control related tasks. The former simplifies these operations while greatly reducing the overhead this typically involves.

Chapter 7 - Software

An embedded processor was implemented in the FPGA fabric as part of the design developed in chapter 5. Software for this CPU was written in order to provide communication links to a host, using the protocol developed in chapter 6. The processor also controls the DMA-readout process, and is capable of automatically monitoring on-board modules. This software is described in this chapter, and possible future development is outlined.

Chapter 8 - System Testing

This chapter details the testing performed of the developed system. The functionality and reliability of the communication links and the embedded software in general is verified, the modules present on the current system are tested, and so is the readout solution developed in chapter 5. Finally, the full readout chain is tested.

Chapter 9 - Discussion and Conclusion

This chapter discusses the results of this thesis, evaluates design choices made, and provides a discussion on future development of the pCT project.

Computed Tomography

Conventional computed tomography (CT) is based on radiography. In 2D radiography, high-energy X-rays are produced by a generator and passed through a target to be imaged. Part of the radiation is absorbed in the target, while the remainder passes through, impacting what is typically either a film or detector. The X-ray beam is attenuated to varying degrees by the different materials that make up the target, which is reflected by the image formed on the film or by the detector.

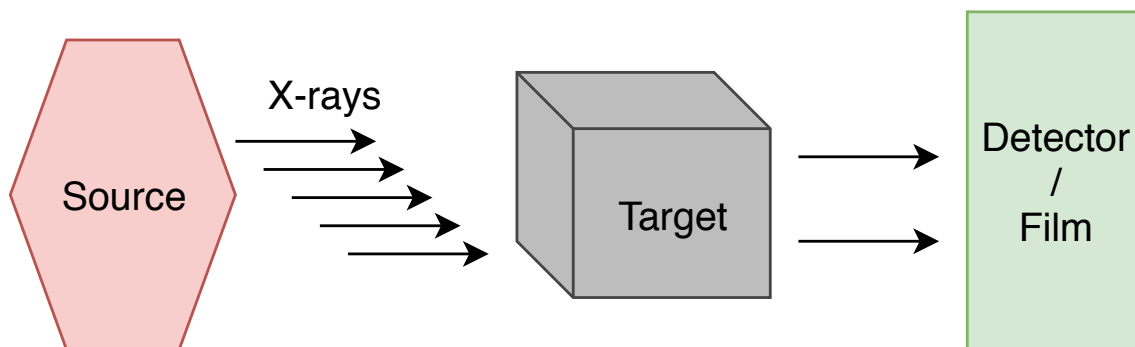


Figure 2.1: 2D radiography showing source, target and detector setup.

Computed tomography builds upon this principle by combining several such images taken from different angles. This is typically done by moving the target along one axis, around which the detector and source are simultaneously spun or vice-versa. The end result, after processing, is a series of cross-sectional images which can be processed further, forming a three-dimensional image of the volume. The technology is in wide use, and in 2007 in the US alone, more than 60 million such scans were performed [2]. In addition to diagnostic imaging, CT as already mentioned sees broad application as a technology to aid in radiation- or particle therapy treatment planning.

2.1 Ionizing Radiation

A criticism of the widespread use of CT for imaging purposes has been its adverse health effects due to the ionizing effects of X-rays [2]. This use has been linked to

irradiated patients experiencing a higher rate of cancer development than those not exposed, and younger patients seeing a further increased risk [3] [4]. These effects stem from the ionizing radiation produced by a CT, which in general is any form of radiation sufficient in energy to liberate electrons from the nuclei they orbit. This can occur both when the electrons are impacted by other subatomic particles, or through their interactions with photons. The effect of this when it occurs within tissue is cellular degradation through damage done to DNA and other cell structures; it is thus an undesired side-effect of imaging, but forms the basis of particle-/photon therapy.

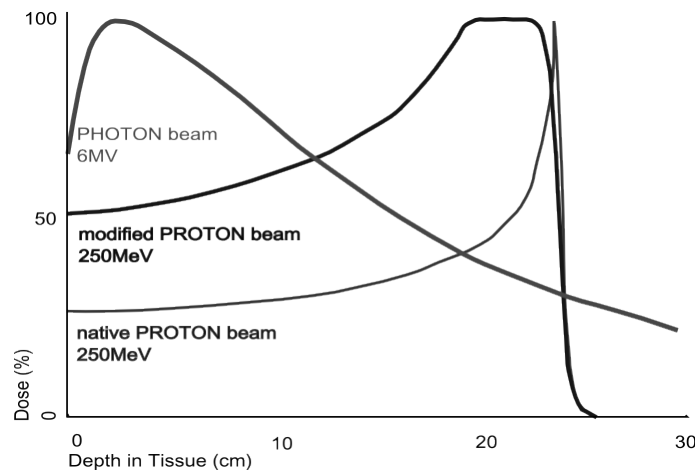


Figure 2.2: Showing the dose delivered by a beam of photons, *modified*- and *native* protons as they pass through tissue.

2.2 Interactions of Photons and Matter

As photons enter a physical medium, they interact with it via three central methods; they are absorbed by it through the photoelectric effect, scattered via Compton- and Rayleigh scattering, or cause pair production to occur.

Of these, primarily the first two are relevant in relation to CT as the latter typically occurs at energies higher than those used for imaging. In the photoelectric effect, a passing photon with sufficient energy¹ ejects orbiting electrons from their nuclei. Compton-scattering describes inelastic collisions between incoming photons and orbiting electrons², where some of the photon energy is transferred to the electron (which are in this case not ejected). The photons then scatter away at some angle.

The effects cause a beam of photons to deposit its energy almost linearly as it passes through a medium, and the affected photons are effectively removed from the beam.

¹As governed by $E_k = hf - \phi$, with ϕ being the *work function*, representing the binding energy of the electron and therefore the lowest energy a photon that ejects the electron may have.

²With the photon post-collision possessing energy equal to $E'_\gamma = \frac{E_\gamma}{1 + (E_\gamma/m_e c^2)(1 - \cos(\theta))}$

2.3 Interactions of Particles and Matter

Protons deliver their energy to surrounding matter via several mechanisms; through their interactions with electrons via the Coulomb force or atomic nuclei, Bremsstrahlung, or nuclear reactions. Of these, their interactions with electrons cause the majority of their energy loss. Below is the Bethe-formula, describing the loss of energy as charged particles pass through a medium:

$$-\frac{dE}{dx} = \frac{4\pi}{m_e c^2} \cdot \frac{nz^2}{\beta^2} \cdot \left(\frac{e^2}{4\pi\epsilon_0}\right)^2 \cdot \left(\ln\left(\frac{2m_e c^2 \beta^2}{I \cdot (1 - \beta^2)}\right) - \beta^2\right), \quad \beta = \frac{v}{c}$$

From this it can be seen that the energy loss of the particles *increases* as $\frac{1}{v^2}$. This attribute gives rise to the characteristic *Bragg-peak* of protons as shown in figure 2.2, displaying the Bragg-curve of two proton-beams.

2.4 Particle Therapy and Proton CT Motivation

The phenomenon of the Bragg peak is exploited in particle therapy to deliver ionizing radiation to a localized area. Typically the target is a tumor, which the radiation is intended to damage by ionizing the cells of which it consists.

The focused Bragg peak of particles can thus be used to deliver doses that are more precise than their photon counterparts. This is seen in figure 2.3, showing dosimetric planning with protons (top) and photons (bottom). Image A shows the expected dose deposition using a single lateral beam while image C shows the result if two opposing sources are used. The two bottom images show treatment with X-rays at different intensities.

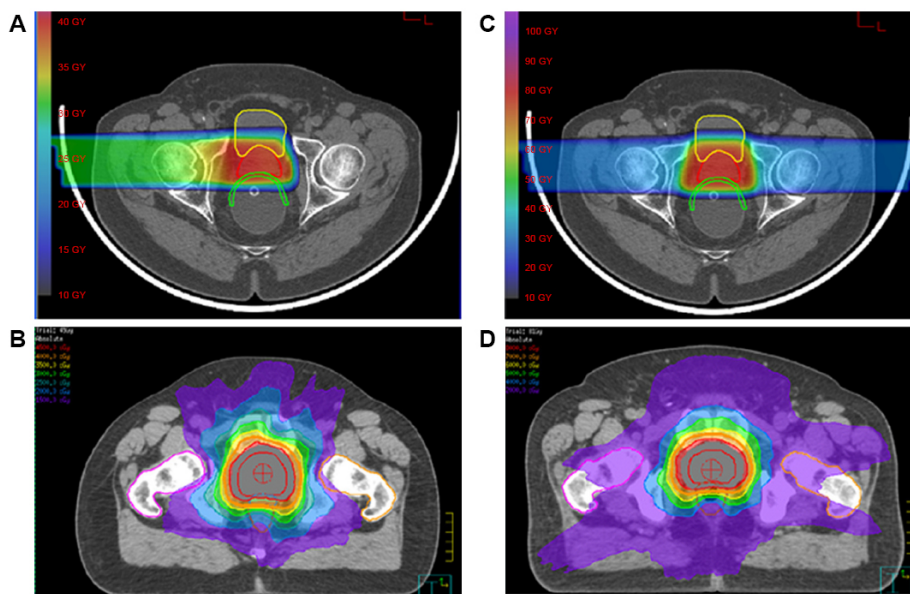


Figure 2.3: Comparison of dosimetric planning with protons (top) to photons (bottom) [5].

Typically, a proton beam used for this purpose consists of protons at varying energies. This causes their Bragg-peaks to appear at different depths in the tissue, allowing an entire tumor to be irradiated. This is the cause of the appearance of the *spread-out-bragg-peak* (SOBP) for the modified proton beam seen in figure 2.2.

2.4.1 Proton CT

A central difference that separates conventional CT from proton-based CT stems from the interactions described in the previous sections. Photons either pass through the target *completely*, or are absorbed, whereas particles collide on their way through. This causes them to scatter and exit the target at an angle different to the one at which they entered, and with their kinetic energies reduced. For proton imaging, *tracking planes* are therefore also required so that the proton tracks can be recorded, and these are typically placed behind as well as in front of the target. From these measurements, the most-likely-path (MLP) of the particles are estimated, while the residual energies are calculated by a separate calorimeter.

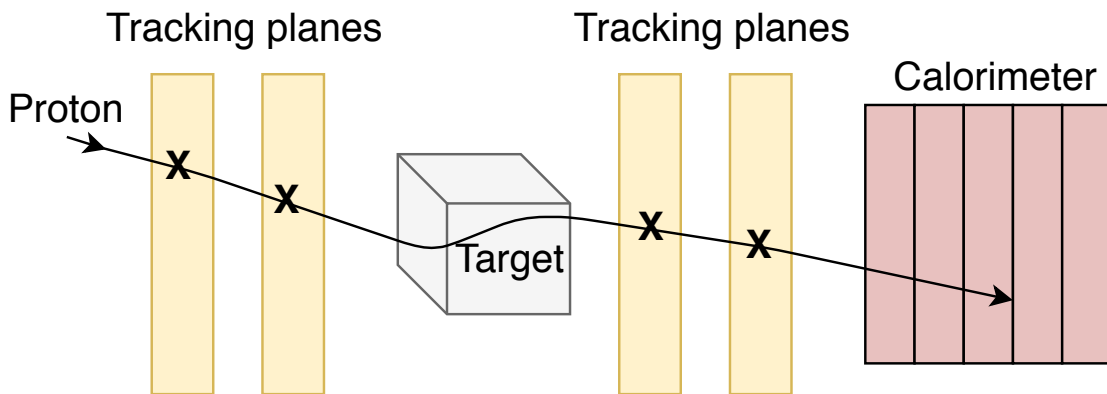


Figure 2.4: Typical pCT layout, showing separate tracking planes and calorimeter and the path of a proton.

Whereas particle therapy requires the protons to stop in the area of the tumor, Proton CT requires the protons to pass through so that their bragg peak appears in the residual-energy detector. pCT thus require beam energies in excess of that used for therapeutic purposes.

It was mentioned that a calculated particle dose is based on a translation process. This involves observing the attenuation of X-rays (represented by Hounsfield Units, or HU) as it passes through tissue, and converting this to proton relative stopping power, or RSP. This introduces range inaccuracies on the order of 2-3% [6]. These inaccuracies must be accounted for, and this is typically done by making the treatment robust so that the focused Bragg peak is not fully utilized.

A pCT as shown above eliminates the errors introduced by the HU - RSP conversion by providing a *direct link* between imaging and therapy, while minimizing damage to healthy tissue due to the Bragg peak and hence the majority of the deposited energy appearing external to the target to be imaged. Energy deposition is low with pCT; a head scan performed by a pCT showed only 1.39 mGy delivered [7],

while a head scan performed with the conventional type was in one case found to be approximately 57 mGy [8].

The UiB pCT and the ALPIDE Pixel Sensor

Accurate reconstructions of proton-trajectories as they pass through a medium in addition to measurements of their residual energies are needed when using a pCT for dosage planning purposes. The prototype under development at UiB will achieve this by layering square arrangements of CERN-developed monolithic pixel sensors that are sensitive to incidental photons and particles which exceed a set energy-threshold. This chip is described in some detail in this chapter¹, as is the general design of the UiB pCT as it currently stands.

The ALPIDEs used in the UiB pCT will serve *both* tracking- and energy deposition measurement purposes. In this respect the system will differ from many other designs, which typically use separate instruments to fill the two roles. However, it is important to note that the ALPIDE itself is not a calorimeter, and that energy deposition is calculated by observing the number of layers that are penetrated by a proton.

3.1 The ALPIDE Pixel Sensor

The ALPIDE is a particle detector originally designed at CERN for use in the ALICE experiment as part of the upgrade of the Inner Tracking System (ITS) [9]. It is capable of detecting particles and photons via a 512×1024 array of sensitive pixels, where each pixel consists of a sensing diode where a voltage appears as incidental high-energy photons and particles ionize its surrounding area. In addition, the chip contains an amplification-, shaping-, and discriminator stage as well as a digital section. This pixel-matrix is mounted on an ASIC that facilitates data-transfer, chip-control, and power-distribution.

¹The given descriptions are largely based on the ALPIDE Operations Manual [10].

3.1.1 Basic Principles of Operation

Figure 3.1 shows an ALPIDE block diagram. It displays the pixel array as separated into regions sixteen double-columns wide, where each white square in a double column contains a pixel. The vertical bars splitting the double columns contain priority encoders that control the order in which pixel data is read out. As hits are registered they are first buffered in RAM, and afterwards passed to the data transmission unit (DTU) (shown in gray), where they are framed, 8b10b-encoded, and transmitted on a high-speed serial link at a configurable speed.

The green blocks containing control-related functionality as well as the DTU show two sets of inputs. Which of these are used depend on the ALPIDE configuration.

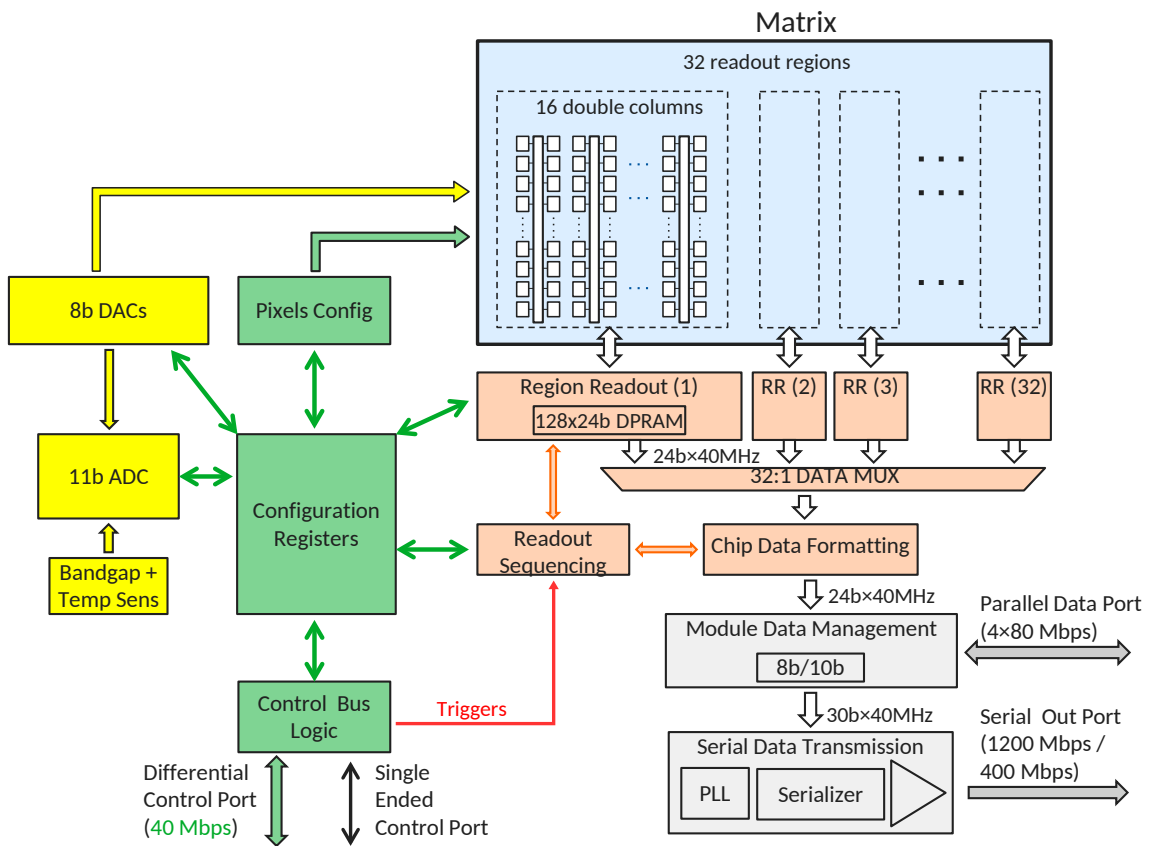


Figure 3.1: Block diagram showing the main components of an ALPIDE chip [10].

3.1.2 Pixels

Each pixel in the array contains a sensing diode placed at the input to an analog stage which discriminates on the voltage at this diode². A bias voltage can be adjusted to increase or decrease the discrimination threshold, and voltages that exceed it cause the active-low output to be applied to the digital section. If the input to this section is low while a strobe signal is high, it is stored in the multi event buffer

²For testing purposes, a signal can also be induced by charging a test-capacitor at this input, while a digital pulse can be applied to the digital section to directly set the pixel state register. Both can be used to force a "hit".

(MEB) and can be read out as a "hit". The generation of a strobe signal follows from the reception of a trigger, which, depending on chip configuration can either cause strobes to be generated continuously afterwards (*continuous mode*), or only once (*triggered mode*). The strobe-window duration is configurable, and can be set between 25 ns to 1638.4 μ s. Up to three hits may be stored in the MEB; if a window is asserted that will place a hit into a second (in the triggered mode) or third (in the continuous mode) buffer-slot, the chip produces a BUSY-signal, causing further triggers to be ignored until the MEB is below this threshold.

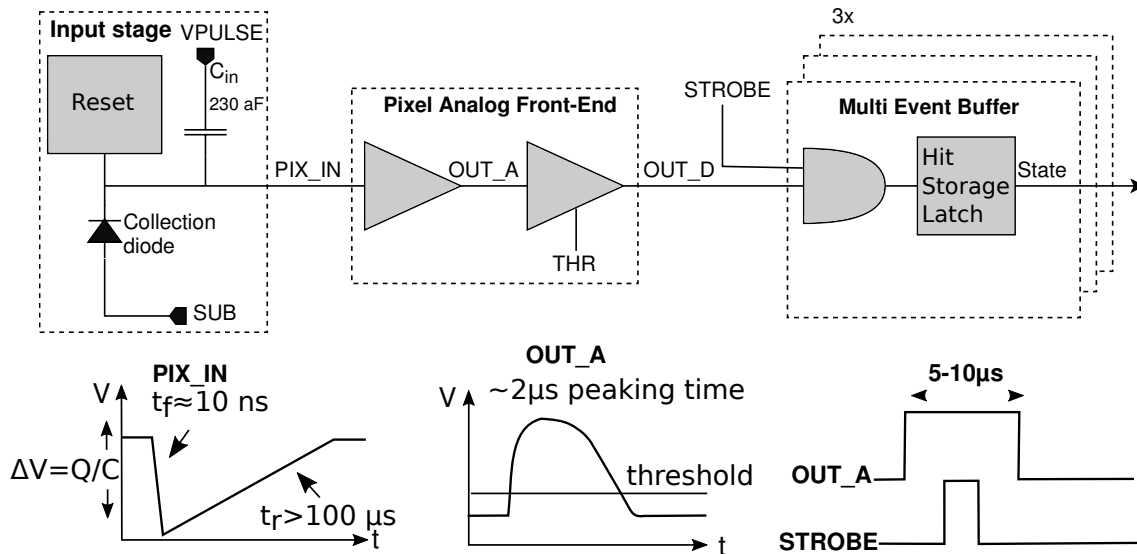


Figure 3.2: Showing how a voltage on the input to the analog section cause a hit to be stored in the pixel buffer if it surpasses a threshold while a strobe is applied [10].

3.1.3 Data Transmission Unit

The DTU provides a fast serial link for readout of pixel data. The ALPIDE can be used in two different configurations: *Outer Barrel (OB)*- and *Inner Barrel (IB)* mode, but is only used in the latter in the UiB pCT. In this case, each chip transmits 8b10b-encoded data via low voltage differential signaling (LVDS) at a max rate of 1.2 Gb/s. Without decoding, this corresponds to a 960 Mb/s data rate. Data rates of 600 Mb/s and 400 Mb/s are also possible in this mode.

3.1.4 ALPIDE - Readout Unit Interface

In the IB-configuration, nine ALPIDEs are mounted together on a *stave*. In this arrangement, the pixel detectors share a global differential 40.08 MHz clock which is multi-dropped to the chips. A differential control line (*slow control*) is also shared amongst the ALPIDEs that provides access to the 16 bit address-space of a chip, where control-commands are addressed either to a single chip via a chip ID system, or to an entire stave via *multicast*. Readout-data is sent off-chip via differential links unique to each ALPIDE at one of the three possible speeds.

3.1.5 Control Interface and Chip Addressing

One of the inputs on the ALPIDE is the seven-bit chip ID. Chips in the IB configuration shall have the three most significant bits of this input set to 0b000 to designate them as such, while the four least significant bits assign a unique identifier, which is then used when addressing it via slow-control.

Protocol

Transactions via slow-control are by default manchester encoded to facilitate AC-coupling, which can be toggled by writing to an ALPIDE register. Data sent via the differential control line consists of 10 bit wide characters; beginning and ending respectively with a low start- and stop-bit, and having a byte of payload between the two. Several operations consisting of one or more characters are supported, which are identified by an initial opcode. These operations are shown in figures 3.3 and 3.4. A *multicast* write is performed on all chips on a stave.

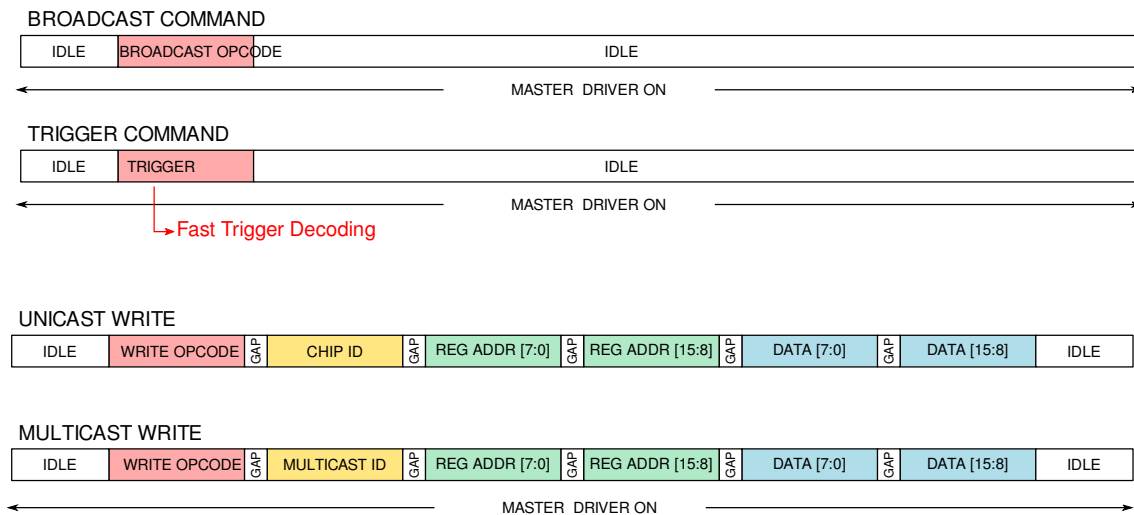


Figure 3.3: Showing the format of ALPIDE broadcasts and uni-/multicast write operations. [10].

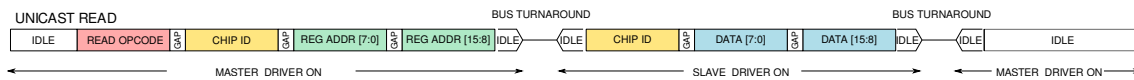


Figure 3.4: Showing the format of an ALPIDE read operation [10].

The *bus-turnaround phase* shown in figure 3.4 requires the master to stop driving the bus for fifty clock cycle in order to allow the chip to respond to the received request.

3.2 The UiB pCT

The UiB pCT will utilize the ALPIDEs both for tracking- and energy measurements, and only in their IB configuration as this allows for the highest data rate and for the chips to be mounted in the stave configuration. Several staves can then be mounted in parallel, forming a square arrangement in what is referred to as a *layer*.

A number of detector- and aluminum absorber layers that decelerate the protons will be sandwiched together and fixed in a support-structure that combined will make up the DTC; the detector structure enabling tracking- and energy deposition measurements of the protons.

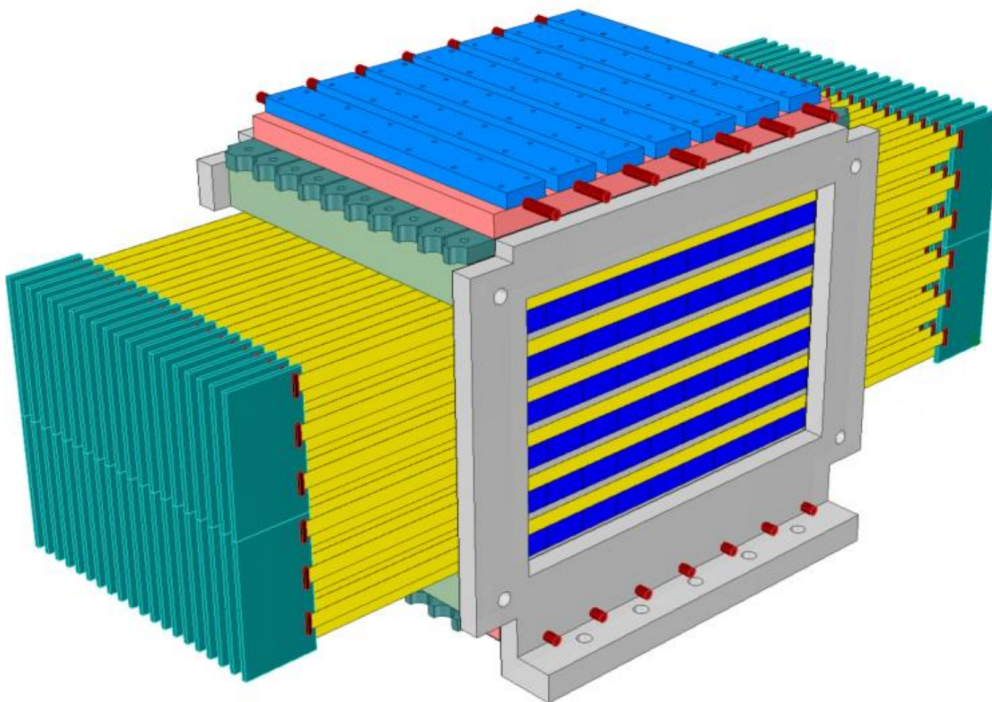


Figure 3.5: A model of the UiB pCT DTC, with ALPIDE chips in a horizontal stave configuration shown in dark blue [11].

The tracking layers will not be external to the DTC, but will instead be realized by excluding the absorbing layers between the two foremost detector layers. These will then perform the tracking, while the remaining layers act as the calorimeter. The implementation of the rear tracking planes is not yet decided, but these will likely also be ALPIDE-based. The light-blue bars on the red plate seen at the top of the structure facilitates cooling of the DTC.

3.3 Existing pCT Systems

Several prototype designs have been developed in the recent years due to the increasing use of particle therapy, with the majority of these being of the type with separate tracking instruments and calorimeters. One example is a head scanner using 200 MeV protons developed at the Loma Linda University Medical Center [12].

The machine is capable of measuring more than 1 million protons per second, enabling a scan to be performed in 7 minutes, making it one of the fastest currently available pCTs. A phantom³ is rotated between two silicon-strip tracking stages, and residual energy is measured via a final scintillator stage. Table 3.1 lists some additional systems and their basic specifications.

Table 3.1: Illustrating some basic specifications of previously developed pCTs [13].

Group	Position sensitive detector technology	Residual energy detector technology	Proton rate (Hz)
LLU/UCSC/NIU	x-y SiSDs	CsI calorimeters	15K
LLU/UCSC/NIU	x-y SiSDs	Plastic scintillator hybrid telescope	2M
PRIMA II	x-y SiSDs	YAG: Ce calorimeters	1M
INFN	x-y Sci-Fi	x-y Sci-Fi	1M
NIU/FNAL	x-y Sci-Fi	Plastic scintillator telescope	2M

Compared with existing systems, the very high readout speeds that are possible with the ALPIDEs will represent one of the prime advantages of the UiB prototype. This has the effect of greatly reducing the time needed to perform a full scan, making the UiB design very efficient. In addition, the fine granularity of the pixel arrays should provide high spatial resolution.

3.4 Readout Electronics

Readout-data produced by the ALPIDEs, as well as their slow-control signals, will interface to electronics located on the pCT Readout Unit (PRU), of which there will be several. Each board will contain a Xilinx FPGA that centrally will be used to process detector-data as it is streamed to the unit in the form shown in table 3.2.

Table 3.2: ALPIDE readout-data format before processing on the PRU [10].

Data Word	Length (bits)	Value (binary)
IDLE	8	1111_1111
CHIP HEADER	16	1010<chip_id[3:0]><BUNCH_COUNTER_FOR_FRAME[10:3]>
CHIP TRAILER	8	1011<readout_flags[3:0]>
CHIP EMPTY FRAME	16	1110<chip_id[3:0]><BUNCH_COUNTER_FOR_FRAME[10:3]>
REGION HEADER	8	110<region_id[4:0]>
DATA SHORT	16	01<encoder_id[3:0]><addr[9:0]>
DATA LONG	24	00<encoder_id[3:0]><addr[9:0]>_0_<hit_map[6:0]>
BUSY ON	8	1111_0001
BUSY OFF	8	1111_0000

Each ALPIDE will connect to its own ALPIDE data module, which handles a large part of this process. It performs 8b10b-decoding, filters out the IDLE-words produced by a chip when no data is ready for readout, packets data according to a custom format (*PRU words*), and monitors the busy-signal produced by a chip if it is unable to process additional hits. Several words will be buffered in block RAM (BRAM) FIFOs and afterwards forwarded to a data-readout stage where the data is streamed off-board; likely over Ethernet.

Communication between the ALPIDEs and PRUs as well as chip trigger-delivery

³An object used to evaluate and tune the performance of medical imaging devices.

will be facilitated by the CERN-developed⁴ ALPIDE Control Module (ACM) that interfaces the shared stave control line and handles the distribution and timing of the commands shown in figures 3.3 and 3.4. Operations on a chip are performed following a three-step process, where a chip-write is performed by writing the register-address and value to two of the module's registers, and triggering the execution of the operation by writing the chip ID and write-opcode to a third. A read requires only an address and the triggering write, with the value at the requested address available in a further control module register. With a layer size of 108 detector-chips, each PRU FPGA will require 12 of these blocks.

To control the PRU devices in general, and to provide a direct or indirect control link with a host, a master control module is required. This will also be responsible for the configuration of PRU modules on startup, automatic power- and temperature monitoring, and possibly other tasks. This module and its implementation is discussed thoroughly in the following chapters. There will also be a block that interfaces to the external voltage regulators that supply the ALPIDEs, allowing control of these and providing the host with monitoring data.

The ALPIDE trigger source will be a device external to the PRUs, and may also be responsible for distributing a synchronized clock to the readout units. The implementation of this module is not yet decided, but possible alternatives are discussed in chapter 4.

Ideally each unit will handle one ALPIDE layer although this will be dictated based on the amount of readout-data produced. Additionally, all modules on the PRU FPGAs will be compliant with the Advanced eXtensible Interface (AXI) standard, which simplifies connectivity as it is supported by all Xilinx IP, and also allows for automatic generation of interconnects between modules, and for these to automatically handle any clock-domain-crossings. A block diagram of the PRU FPGAs is shown in figure 3.6.

3.4.1 Current Implementation

While the PRUs that will be installed in the complete pCT prototype will consist of custom boards, current work is being done using a VCU118 development board on its Xilinx Virtex Ultrascale+ VU9P FPGA. Any implementations that were made as part of the work done in this thesis were made on this device. When work was started, the design included a single ACM and ADM, and a connection to an ALPIDE is provided via a custom FMC, interfacing to a FireFly-to-PCIe adapter, which in turn connects to the ALPIDE carrier board.

⁴Since modified by Ola Slettevoll Grøttvik (ola.grottvik@uib.no) to be AXI-compliant, whereas the original was a Wishbone-module.

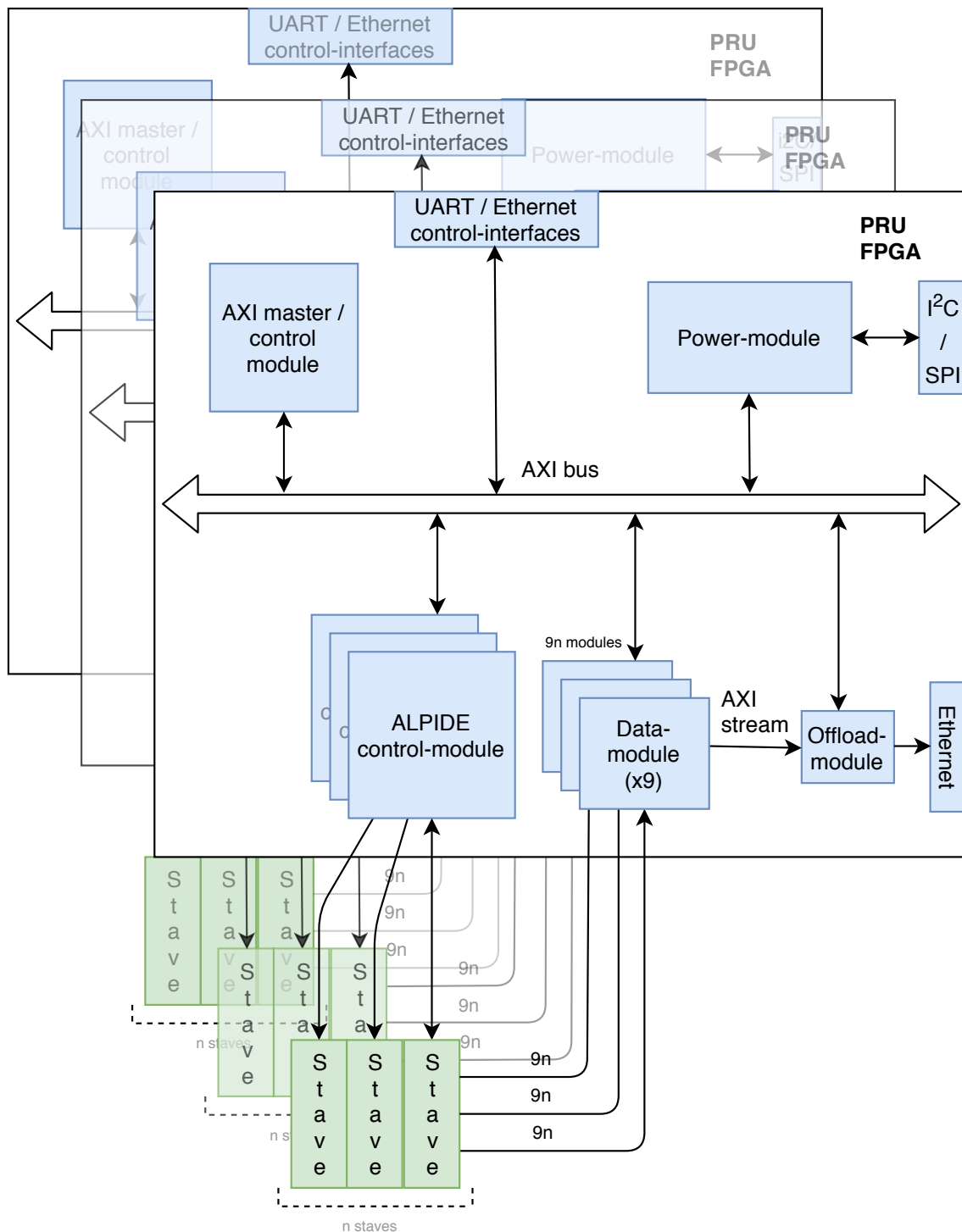


Figure 3.6: The PRU, showing the most central modules of the currently planned design.

The pCT Control System

In the complete pCT, each PRU will contain a multiple of several of the modules discussed in section 3.4, and a subsystem on the PRU that allows for complete control and monitoring of these is required. This involves in part the design and implementation of the AXI master control module located on the readout boards, any additional firmware this requires, as well as host-side software that interfaces to this module and allows for visualization and remote control. The master control module can be either embedded within the FPGA, or be an external unit. In any case, the system will be responsible for the following:

- Implementation of protocols for PRU AXI master - host (control room, testing setup, etc) communication.
- Facilitation of communication between the AXI master and all modules on the PRU.
- Initialization of the readout boards: on power-up, automatically configure devices present.
- Provision of assistance related to procedures such as data-readout.
- Provision of house-keeping data for monitoring and logging purposes.
- Application of chip-specific configurations for the ALPIDEs. Possibly streamed to the master-device or collected from on-board memory.
- Automatic monitoring of chip- and board currents and voltages.

The system may be required to fulfill additional roles, and a design must be flexible enough to accommodate this. In addition, the PRUs will interface to the control room either directly or through some other device, and making this interface simple is therefore an additional consideration. This will reduce the need for specialized hardware, and allow use of tried-and-tested protocols. This chapter further specifies the requirements of such a system, considers several options for its design, and proposes some alternatives.

The realization of a complete control system lies outside the scope of this thesis, but core elements are developed, implemented and tested, and are described in chapters 5, 6, 7, and 8. Future extensions and alterations are also discussed towards the end of their respective chapters. During the design process, emphasis was placed on developing a scalable-, self-sufficient-, and flexible system that could be used by itself in the development- and testing phase, and further built upon for use with the finished product.

4.1 Features of a pCT Control System

4.1.1 RU - Host Interface

For a host to communicate with the PRUs, data must be transmitted to the control-unit on the readout-boards over a suitable interface such as Ethernet, USB, RS-232 or others. RS-232 defines electrical signal characteristics such as voltage levels and timing, as well as the properties of related mechanical connectors and the circuitry [14]. Signals are transmitted over a single wire, limiting transmission distance and resulting in poor signal integrity. The high voltage swings limits performance, and there is no concept of addressing. RS-422 is a similar standard, but has a lower swing, operating between -6V and +6V, which improves signaling-rate. Differential signaling is used, which enhances noise immunity¹. Still, the upper data-rate of RS-422 is 10 Mb/s, and if, during the development stage, detector-data is to be read out over the same link as the control- and monitoring data, then this will not suffice. Again, there is no built-in concept of addressing, which would have to be developed. RS-485 is in many ways similar to RS-422, but can be multidropped to several devices. The 10 Mb/s bandwidth must however be shared amongst the connected devices.

USB can provide much better throughput, but is unsuitable for transmission over longer distances; although the standard does not explicitly define an upper limit to cable length, it defines the electrical characteristics it must meet, which puts an upper limit at around 1.2m. PCIe is an additional option, which offers performance up to 126 Gb/s if using third generation PCIe and 16x links. PCIe over cable is possible, and mounting the board in a specialized enclosure is therefore not necessary, although a PCIe card on the host side that interfaces with such a cable is. This introduces some complexity and cost. For control, there is no need for such performance, and sourcing and purchasing the cabling and interface card would be problematic for development. Range is limited if copper cabling is used, but Samtec for instance offers third generation PCIe over optical at ranges up to 300 m at 4x or 8x link-widths [15].

Ethernet is a family of protocols and standards that defines physical interconnects and cabling, the discrete transmission of data in the form of *Ethernet frames*, addressing schemes, and error checking. On top of Ethernet the TCP/IP protocol suite is often used, which handles aspects of addressing, lost frames (depending on

¹Noise that is picked up by one wire will be picked up by the second to an almost equal degree. As the "signal" is interpreted as the difference between the two, the noise is cancelled out.

transport protocol), routing, checksums and congestion handling. Generally, the lower levels consist of pre-existing firmware/software solutions while the user implements or uses a specific application-layer protocol. Compared to the alternatives listed above, Ethernet is more demanding in terms of implementation on the PRU side, and real-time control and monitoring, if this is wanted, is generally not possible. However, interfacing Ethernet requires only twisted pair cable and a network switch. Multiple applications may also transmit over a single Ethernet link through the use of sockets, allowing in this case both control- and readout-data to be transmitted through the same cable, during the early stages of the project.

For control, Ethernet is a suitable choice, providing speed, flexibility in terms of use (control and data-readout) and expansion, ease of setup, low cost, and offering existing communication protocols for all levels. A serial UART interface should also be available in order to provide a simple debugging channel for each board.

Relaying control-data to individual readout units via a *master PRU* and other indirect host - board interfaces are also possible, but it can be argued that a direct channel between each board and host provides both better flexibility as well as simplicity of implementation; The master PRU would mandate the design of a separate board, and separate interfaces between host - PRU and PRU - PRU would have to be developed.

4.1.2 Board Initialization

The PRU modules must be configured on start-up, or on reception of one or more commands instructing this to be performed. For an ALPIDE, initialization consists of a register programming sequence and the application of a pixel-mask that is applied in order to exclude noisy or otherwise defective pixels from readout. The AXI-master is responsible for carrying out these procedures, and could possibly do so by using chip-specific settings stored in on-board flash memory. However, local storage of configurations will likely not be necessary, which can be argued by first noting that for each ALPIDE there are only eighteen periphery control- and fifteen DAC registers that must be set, as well as one ADC control register that must be configured with chip-specific settings. Mask-application is a more comprehensive task, and the number of operations required to apply these will vary based on the number of defects in the chips. All 524288 pixels can in theory be masked on an individual basis (see section E.1 for a brief overview of this process), but the addressing scheme used to access the pixels is implemented in a manner that reduces the number of necessary write-operations. This allows masking of even a significant number of pixels in an efficient manner.

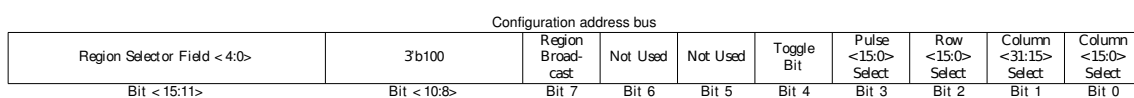


Figure 4.1: The pixel-matrix addressing scheme [10]. Writing 0xFFFF to the address formed by the region selector field set to 0b1111 and the row bit and both column bits set would for instance mask/clear all pixels, depending on the pixel configuration register.

In other words, the amount of data that must be written in order to configure a chip is not excessive. Configurations may also change over time, and if data is stored on-board, it would have to be updated on a per-board basis. It is likely that updating a central repository will be easier to manage. Instead, ALPIDE-specific settings could be kept in a database and applied from the host side. Ignoring any protocol-specific overhead, the amount of data needed to configure each chip can be estimated:

The width of the ALPIDE address space as well as its registers is 16 bits. In addition, a control value of 16 bits must also be written to the control module. The transaction is performed on the 32 bit AXI bus however, and therefore 24 B must be written per write-transaction. If a particularly bad detector requires, for instance, a third of its pixels to be masked, the amount of data that would need to be provided to the ACM would be approximately:

$$24 B \times 3 \text{ operations} \times \frac{N_{\text{pixels}}/16}{3} \approx 0.78 MB$$

30 % is in essence a non-functional detector, so this number is pessimistic. Additionally, the ALPIDE addressing scheme allows for several rows, columns or regions to be masked simultaneously, which might often be the case if pixels are malfunctioning. For instance, the ALPIDE chip that was used to test the systems developed in this thesis has one entire inoperative column.

If configuration data is stored on-board, the overhead in the above operation does not need to be stored. Masks could be represented as a group of 32 bit unsigned integers, with each bit of an integer indicating whether or not to mask a pixel. Each ALPIDE would then need 65.5 kB to store a complete mask, and an additional 136 B (approximately) to store the remaining registers. A layer consisting of 108 ALPIDEs would then require 7.09 Mb of data in total.

Board Configuration Times

Some estimations should be made regarding the time required to complete the procedures mentioned in section 4.1.2, as it could dictate whether configuration data should be stored on-board or streamed to each unit.

The majority of the modules on the AXI bus require little configuration, possessing only a few registers each. Likewise, setting the periphery control-, DAC- and the chip-specific ADC registers on the ALPIDEs is not an extensive procedure. As such it is the pixel masks that will require the largest amount of configuration data. An estimate on the time required to perform this procedure can be found by first noting that the ALPIDE write-operation equates to $1.8 \mu\text{s}/\text{pixel}^2$ if the module is operating on a 40 MHz clock. Masking the pixels of the hypothetical ALPIDE would then complete in about 0.019 s, or 2.12 s for a layer of size 108, again without regards to the master control module overhead.

If configuration data is streamed to each board by sending the AXI addresses and -data to write to the ACMs, this equates to 72 bytes per group of 16 pixels, excluding

² $3 \text{ writes} \times 3 \text{ operations} \times (((7+1) \text{ cycle})/\text{write})/40 \text{ MHz} \approx 1.8 \mu\text{s}$. 7 cycles are needed to initiate an AXI transaction, while 1 cycle is needed to transfer the data bits.

any protocol overhead, as stated. With 0.78 MB of data required in order to apply a mask, a layer of size 108 would require 84.9 MB of data to be streamed to the PRU in this case. On saturated 10 Mb, 100 Mb, or 1000 Mb links, this would take 67.2 s, 6.72 s, or 0.672 s, respectively, to transfer. These are all acceptable numbers, and more effective ways of transferring this data is possible, as the majority in the above estimations stem from overhead. In any event it is desirable to minimize time spent on the configuration stage, although gigabit transfer rates if using a soft core processor at a relatively low clock rate are unlikely.

4.1.3 Provision of House-Keeping Data

The ALPIDE is equipped with a 10-bit ADC, which after calibration provides values of chip temperature as well as all its voltages and currents. For monitoring-purposes, these must be read out, with the calibration procedure performed by either the master control module or the host. The conversion between raw ADC register values and corresponding units requires some floating point arithmetic, but the amount is small enough³ that it can be performed by the on-board master control module, even if this is a low-performance soft core CPU. The control-unit can either provide these values automatically, or as a response to a request.

4.1.4 ALPIDE Monitoring

Monitoring the pixel detectors is a two-part process, involving the ALPIDE built-in ADCs and their related registers, as well as the external regulators that provide the analog- and digital supply voltages to the chips on a stove-basis. The ADC values should be read at regular intervals by the AXI master (i.e without host/control-room instruction), with the reasoning being that the system should not rely on manual input in order to protect itself from damage. A scenario can be imagined where a disconnect occurs and temperature increases beyond a safe threshold, for instance.

Secondly, the voltages and currents supplied by the regulators must be monitored, which can be done either indirectly by reading the ALPIDE ADC registers, or by having a separate power-monitoring module on the PRU perform the task. An argument for the latter is that performing a full ADC-measurement is time consuming. With the ADC ramp speed setting set to its default value of 1 μ s/step this process takes 25 ms⁴. In CMOS transistors, and especially in those operating in high-radiation environments, latchup can present an issue [16]. The effect occurs when a low-impedance path is formed between the n-well-/p-well and substrate of transistors, appearing due to charge deposition caused by single event upsets. This causes conduction through the parasitic bipolar junction transistors present in CMOS circuits, and these currents can be significant, possibly damaging the affected part and in any case requiring a power-cycle to resolve.

The speed at which this occurs would require that if the master control module is to monitor for latchup directly, it would have to take frequent readings of the ALPIDE

³One FLOP is required to perform each conversion, equating to 4536 FLOPS for a layer of size 108.

⁴With the fastest ramp speed setting, this is reduced to 15 ms.

ADCs. Single-channel reads can be performed, and the time needed for the ADC to complete a single-channel measurement is in this case approximately 1.1 ms at the default ramp-speed setting. However, this would cause appreciable traffic on the AXI bus, and requires an ADC control register setting different than the one that allows for complete (all-channel) ADC measurement. In addition, although the ALPIDE is not latchup-immune [17], it is not likely to occur at a high rate, and the frequent polling might burden the master control module unnecessarily. To avoid the direct polling but still allowing the master control module to be informed on any events, the current can instead be sensed by an external ADC in combination with a sense-resistor and current-sense amplifier, as shown in figure 4.2.

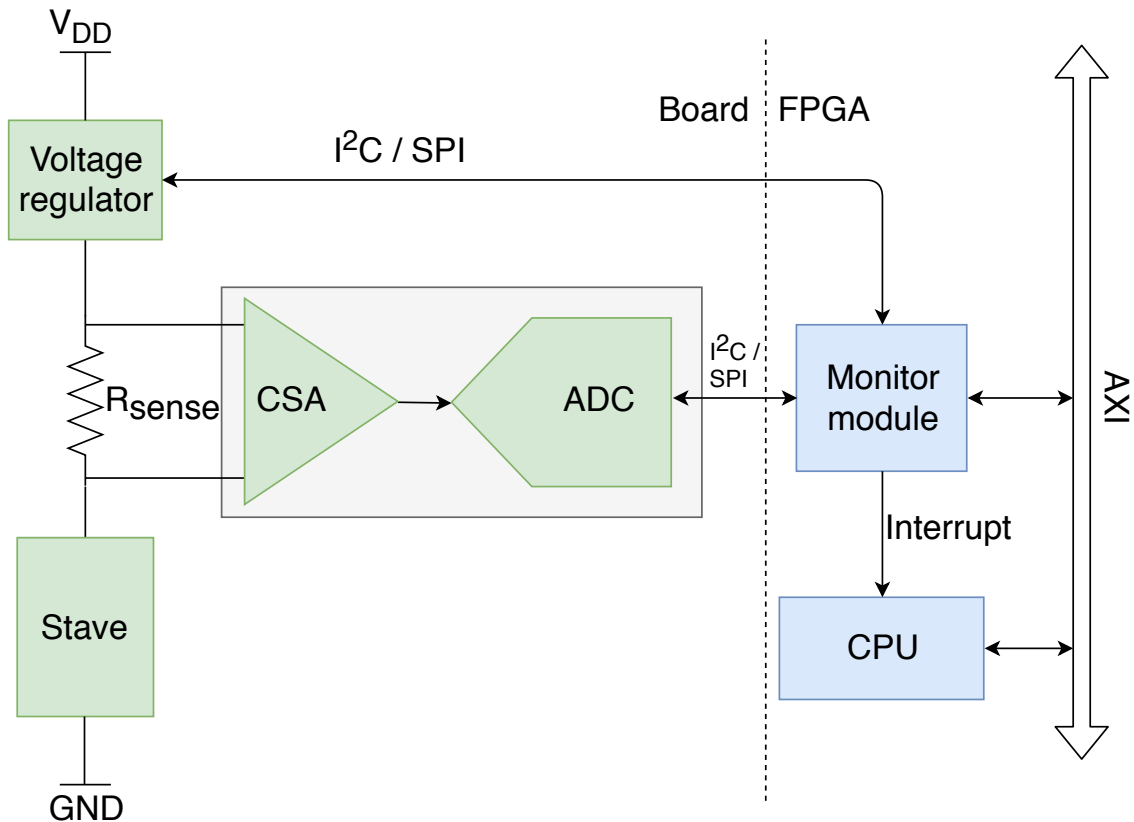


Figure 4.2: An external sense resistor, current-sense amplifier and ADC allows for monitoring of chip currents and -voltages without occupying the AXI control module.

The monitor module can either use the value provided by the ADC to instruct the voltage regulator to perform the power cycle if it detects a value that exceeds a configurable setting, or this can be done directly by the regulator as current limiting is a common feature. If the monitoring module is made AXI4-Lite compliant, the ADC measurements can be provided to the control module and hence a host by reading one of its registers. If interfaced directly to the FPGA, I²C might be preferable to SPI due to the smaller number of wires needed. In this case, for a layer consisting of 12 staves, only two pins will be required for the I²C lines in order to interface both ADCs (if these are also I²C-compatible) and regulators, although an external I²C multiplexer might be required in order to resolve any addressing conflicts. If the control module monitors latchup by polling registers of the ALPIDE ADCs, connections to the regulators are still required in order to perform the power cycle.

The readout of the ALPIDE ADC registers will place some requirements on the Ethernet bandwidth, but these are not great. High temporal resolution is not required, as real-time monitoring is provided by the on-board master control module and possibly the monitor module as described in this section. In that case, updates of these values on the host side does not need to occur at a particularly high frequency, and a rate of twice per second is likely sufficient. This requires that a *start measurement* opcode is first written to all ALPIDEs, which can be sent as a broadcast. If it is again assumed that data is acquired by providing the ACMs with addresses and data, readout of all 21 ADC-registers for a layer of 108 ALPIDEs will require approximately 870 kb/s of data to be sent to the PRUs, and 73 kb/s to be received by the host⁵.

4.1.5 Additional Features and Data Readout

During the development stage of the pCT project, it might be beneficial if the control system could assist in the data acquisition process, primarily through data-readout. This would involve collecting data from the ALPIDE data-modules and transmitting it to a host via the Ethernet link. The viability and implementation of such a solution and its applicability in later stages of the design is discussed further in chapter 5.

4.1.6 The AXI Master

It has been assumed that the AXI master on the PRU will be a processor (soft- or hard core), but implementing it in the FPGA fabric as a custom logic module is also possible. There are, however, reasons for using a CPU. One of these is that it facilitates the implementation of high-level communication protocols. TCP/IP is usually implemented in software, and others such as Modbus TCP, OPC UA, or Message Queuing Telemetry Transport (MQTT) are also available. TCP can be implemented in firmware, but due to the complexity of the protocol it is expensive in terms of resource use, and not many open source alternatives exist. A processor also provides a higher degree of flexibility; if functionality is added, this is trivial in software and done simply by adding more code. Appending parts to HDL state machines usually requires more effort.

An additional aspect is the necessary performance: it can be argued that the performance of the FPGA logic is not required, as running the communication stack, reading ADC-values, performing initialization procedures, and writing to- and reading from registers are not tasks that benefit greatly from parallelization. Finally, there will be many paths through the control-system logic. Although difficult to quantify, these types of operations are typically easier to implement as sequential instructions on a processor.

⁵ $108chips \times 21registers \times 192b \times 2Hz$ and $108chips \times 21registers \times 32b \times 2Hz$ for the send/receive cases, respectively.

4.2 Clock- & Trigger Distribution

A central aspect of the PRU-design is clock and ALPIDE-trigger distribution. To the ALPIDE control module, 40 MHz- and 160 MHz clocks are required. The AXI master and support modules discussed in this chapter also require a clock. While some skew between boards in the clocks supplied to firmware related to the control- and monitoring system is tolerable, it should be kept to a minimum for the ALPIDE control- and data module, and so should the inter-board skew in trigger arrival times. As mentioned in section 3.1.2, triggers cause the generation of the ALPIDE strobe-windows. Furthermore it is likely that the chips will operate in continuous mode, meaning that strobes are automatically generated by the ALPIDEs upon reception of the first trigger. Since data is tagged as belonging to a specific window, an event tagged by two layers, but tagged by the second as belonging to the previous strobe-window results in useless data as the events cannot be related. However, due to the approximately $2\ \mu\text{s}$ duration of the peaking time of the signal on the analog section output, some skew will be tolerable; the strobe windows are wide ($2\ \mu\text{s}$ - $1638.4\ \mu\text{s}$), and the gaps between them small (a few nanoseconds). The long peak time of the analog pulse and wide windows provide slack, in the sense that the probability of skew between boards in the nanosecond-range causing an event to be detected by one layer but not the other will be low.

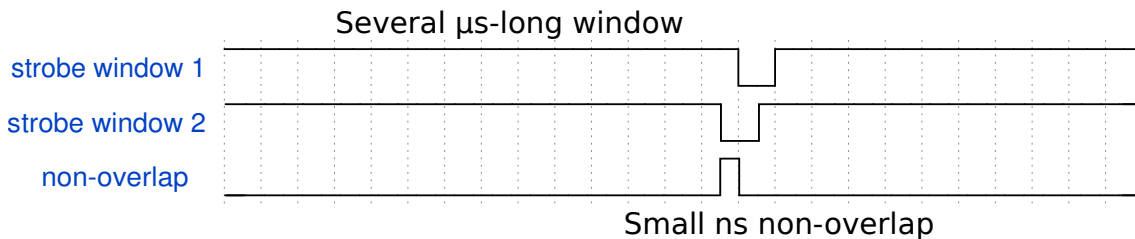


Figure 4.3: Illustrating the low likelihood of non-overlapping strobe-windows.

Distributing the clocks and triggers is the role of the system control unit. As these are the only roles of this module, its design is simplified. Signals can be transmitted over either optical- or coaxial cable, as both provide good noise immunity. As the SCU is only occupied with distributing the clocks and possibly triggers, there are commercial solutions that can perform this task. Several vendors offer cards with PCIe connectivity for programmability that can distribute a number of clocks⁶. If the distributor is developed as a custom board, this can be done by providing a single reference input clock which is then split using a clock divider. If the trigger is generated by a second device, this can be synchronized to the distributed clock.

To avoid the unknown latency that follows if the trigger is received over the Ethernet interface of the PRU, the signal must be received by a separate trigger-distributor on the FPGA. Several aspects cause this uncertainty; for one it is impossible to know where in the execution of a program the CPU will be on trigger-arrival, which will effect context-switch time. Secondly, Ethernet is not deterministic and hence triggers might arrive with a delay, and due to the possibility of packet-loss, they

⁶For instance these solutions offered by Pentek: <https://www.pentek.com/iocentral/IOCentral.cfm>

might not arrive at all⁷. Excluding the CPU, the bus might be in use on one PRU and idle on another, resulting in a delay. The local distributor module does not need to be a bus-master as the ALPIDE control modules features an asynchronous trigger-input in addition to the AXI-interface. During the very early development phase where no separate trigger-input is available however, it must be possible to transmit this signal via the Ethernet link.

Even if perfect synchronicity between the PRU CPUs could be assumed, unacceptable skew between trigger arrival time to the ACMs would occur if these are written via the bus, due to the time needed to perform an AXI write transaction. A delay of at least five cycles is introduced on each write operation, meaning that with a 100 MHz clock, a 70 ns second delay occurs between successive control modules (and hence two staves) receiving a trigger. From the discussion above, this might be tolerable if only a few staves are used. However, the delay between the first and last stave in a 12-stave system would in this case be at least 840 ns.

4.3 The PRU Processor

Several soft-core processors that can be embedded in FPGA fabric exist, including open-source- and vendor-independent alternatives. However, the PRUs will be built using Xilinx devices, and as such it is beneficial to select the MicroBlaze. The MicroBlaze is a 32 bit Reduced Instruction Set Architecture (RISC) soft-core processor with native AXI support. It can be configured to a great degree, trading performance for footprint⁸, and is compatible with several operating systems.

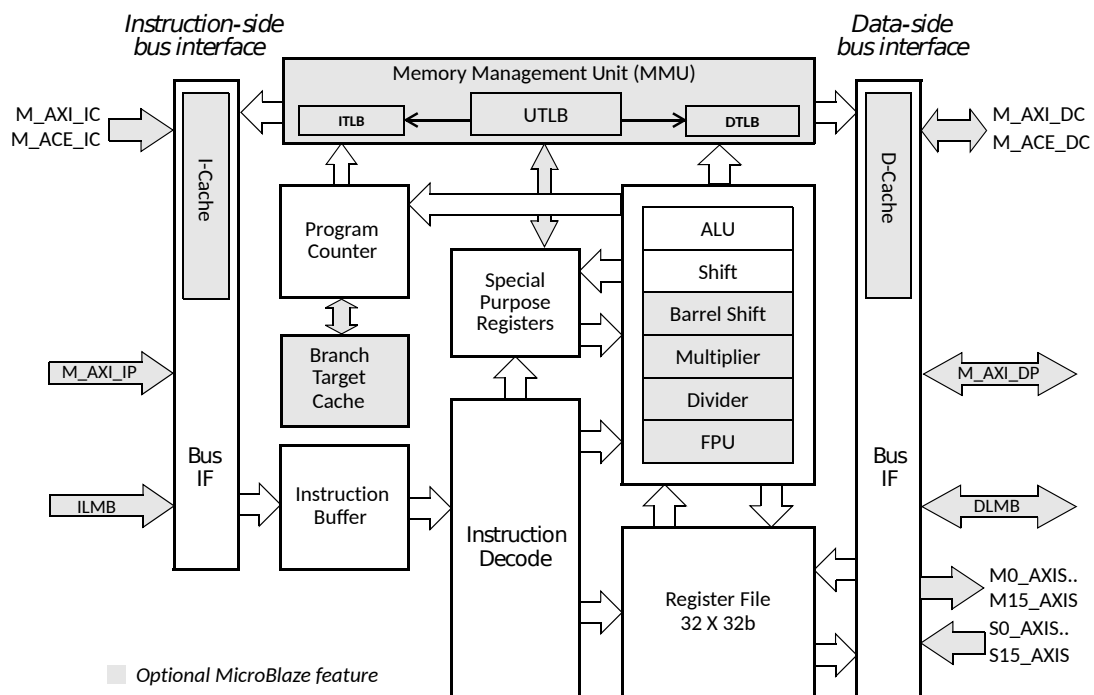


Figure 4.4: MicroBlaze architecture, showing the optional features grayed out [19].

⁷Or they arrive delayed due to some reliability mechanism of the transport level protocol used.

⁸On a Virtex Ultrascale+ platform, minimum area- and frequency optimized configurations consume respectively 556/6070 flip-flops and 242/5949 look-up-tables.

The demands on the processor will vary through the stages of design; if in the earlier stages it assists in the data readout process, it should be configured for maximum performance so it does not become a bottleneck. If these procedures are later delegated to other firmware on the PRUs, this is not required, and the additional features can be disabled in the event that FPGA resources become scarce.

4.3.1 Operating Systems

There is an option to run an operating system on the MicroBlaze, and several alternatives exist, either real- or soft real-time. An OS provides the capability of multi-threading and real-time handling of events, but can also lead to reductions in performance due to the overhead introduced by context switching, background tasks and daemons. In addition, higher requirements are placed on board-resources, and this should be weighed against the possible benefits.

In a system with no OS, a program is written as a single loop, and the flow is linear with any context-switching achieved through the use of interrupts. As *bare-metal* systems grow large, the time interval between successive executions of any one task grows. An RTOS-based system can place hard upper bounds on these intervals by spawning tasks to handle jobs; these can then be given a priority and set to perform work at fixed frequencies. In relation to the pCT control system this can be used to spawn a high-priority task that wakes to monitor selected PRU modules and ALPIDEs. Section 4.1.4 discussed monitoring of ALPIDEs and real-time capability could be helpful in this regard.

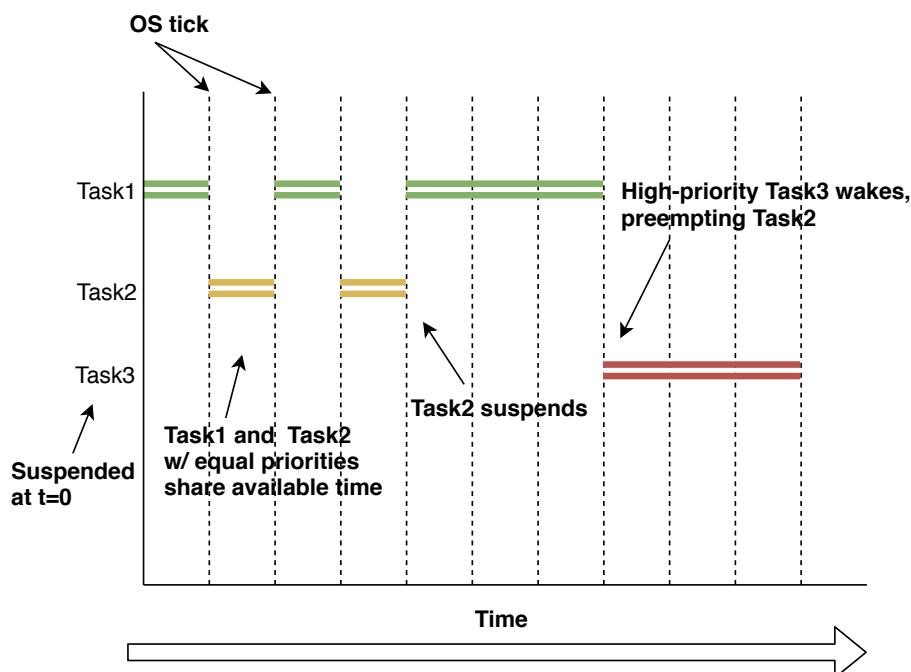


Figure 4.5: Program flow in an RTOS-based system.

Operating systems also facilitate event-driven programming; tasks that are idle and would otherwise waste CPU time can be suspended. Alternatively, if several running

tasks are assigned the same priority, they may each be given an equal amount of time. Different processes might also share access to common resources such as global variables through semaphores and mutexes, and communicate data through queues. Using an OS also simplifies expansion in the sense that if additional jobs are added to the system, a new task may simply be spawned, whereas added functionality might cause a task in a bare-metal system to be performed too infrequently.

Alternatives

One option is Linux. As an embedded operating system it sees broad use and the functionality it provides is extensive. The OS provides built-in networking-functionality, as well as a file-system, command line interface, and more. A further benefit of Linux stems from its popularity, which means that pre-developed software is easily available. However, the requirement that the embedded system is preferably not to use any external memory will be a challenge in this case. The VU9P chip on the VCU118 board features approximately 9.5 MB of BRAM [18], and this will not be sufficient by itself. In addition, the majority of this is to be used in the buffer stages between the ADM and data-readout stage(s). Xilinx offers the Petalinux tool-chain to aid in deployment of Linux distributions to Xilinx silicon.

Another alternative is FreeRTOS. It is a freely distributed, lightweight real-time operating system maintained as an Amazon Web Services (AWS) project. The kernel itself occupies between 6 kB-12 kB, whereas a Linux distribution may be several megabytes. The OS functions primarily as a way to enable multitasking and facilitates inter-task communication, and as such introduces little overhead. It includes an implementation of the TCP/IP stack, although this is not natively compatible with Xilinx IPs. The Lightweight IP (LwIP)-library can however be used with the OS, and features such as a CLI are available as optional add-ons. FreeRTOS is in addition truly real-time whereas Linux is not. This may or may not be useful, for instance depending on the power-monitoring scheme chosen, as described in section 4.1.4.

The performance penalty resulting from using an OS stems in part from context switching. With FreeRTOS, this represents the majority of the performance overhead. On an ARM Cortex-M3 CPU, it was found that with stack overflow protection-, trace features- and run-time stats turned off (as would be the case in a release-build), a switch time of 84 cycles was found. This will vary somewhat from CPU to CPU, but not by an extreme amount. A division instruction on the MicroBlaze consumes by comparison 29 cycles [19]. There are also other alternatives in this category such as VxWorks, QNX or RTEMS, but FreeRTOS is preferable due to its simplicity and its widespread use, meaning that there is a broad range of software available for it.

A further option is to not use any form of operating system, and run compiled C/C++ code directly on the processor. Any tasks run must be either in a continuous super-loop or be based on interrupts. For scalability-reasons, this is not considered a good alternative.

Of the three alternatives, FreeRTOS is a good option. One of the reasons for this

is that the scheduler allows for flexibility in terms of expansion; in an interrupt-based design, any additional functionality might mandate the restructuring of these routines. Another advantage that the OS brings is in making the application event driven; there is no need for a thread that is idle (for instance any thread that waits on data to appear on an input) to consume resources. An OS in general allows such threads to be suspended.

FreeRTOS is found to be a better alternative to Linux due to the complex feature set of the latter offering few advantages, while at the same time the various background threads and daemons could be detrimental to performance. A further advantage is that FreeRTOS provides actual real-time capabilities whereas Linux does not, which could be required for monitoring, where events needs to be handled quickly. This could in the FreeRTOS case be related to a high-priority task that is guaranteed to act within a set time period.

The additional memory required for a Linux distribution also represents a problem; the PRU should if possible not require external memory as this requires a large number I/O pins, and this could result in an issue as a large portion of the I/O is to be used to interface the ALPIDEs. This ideally leaves only the block RAM available as explained in the previous section.

4.4 PRU Software Applications

There are a few options for the type of application that would provide control of the PRU FPGAs, and the needs will likely change somewhat as the pCT project evolves. It is argued that especially during development, what is required is a flexible system that allows primarily for access to the device's address space, and thereby the modules on the boards. This is due to the system in this period undergoing frequent alterations, where modules are replaced or removed, addresses are altered, and test procedures are written that usually need to access these directly.

To accomplish this, a web server could for instance be implemented and used to display central parameters such as voltages and temperatures, running/not running, or similar statistics [20]. A user could type in addresses that they wish to poll or alter the value of, buttons could be added to run procedures, etc. This allows for visualization and basic use of a board, but is not particularly general; functionality would have to be hard-coded into the PRU CPU software. Additionally, HTTP transmits requests as ASCII-strings, which is not bandwidth-efficient; particularly on an embedded platform. Another option is to embed an MQTT client or OPC server on the units. This may work very well at a later stage, and is discussed further in chapter 7. These are however static structures that could complicate use especially during the early development stage when frequent changes are made to the design. In addition, it must be possible to control the processor via the UART interface. Chapter 6 further discusses the choice of protocol.

4.5 A Summary of the Previous Sections

From the discussions in in this chapter, it is clear that a system able to perform all required roles would be one where the PRUs feature soft-core microprocessors that communicate with a host directly. A processor allows for an implementation of the TCP/IP stack which provides a reliable control channel and support for many higher-level applications, and would be suitable for development of procedures that automatically configure PRU devices and perform monitoring. Device drivers can be written to control the ALPIDEs through the ACM, as well as for all remaining firmware on the PRUs. The processor can also implement routines that allow detector data to be read out via the Ethernet link during development. Configuration data can likely be streamed to the readout units instead of stored locally. The SCU is reduced to a unit that only distributes clocks and possibly triggers, simplifying design, and it was explained that some skew between boards will be tolerable.

Ethernet provides a simple PRU - host interface, which can consist only of a network switch and twisted-pair cabling. It also provides flexibility in the number of boards, as each PRU might simply be given a MAC address by which it can be identified. Reliability is also provided if the protocol that is used to transfer data to the PRUs is carried by TCP, as any data sent via TCP is guaranteed to reach its destination except in cases of physical disconnects or broken links, through sequence numbers and retransmissions. Ethernet generally does not permit real-time monitoring, but this will not be necessary, as it can be performed by the PRU CPU and the module(s) that monitor the voltage regulators.

An application-layer protocol that defines communication between a PRU and a host which can be carried by TCP over Ethernet and via the serial interface is required. This must provide access to all PRU modules, and allow for CPU-specific functionality to be performed. It was argued that what is required at the current stage of the project is primarily a way to directly access the memory space of the readout unit.

The following chapter details the requirements- and implementation of the PRU firmware required to perform the tasks detailed in this chapter; including support for data-readout. Chapter 6 describe the development of a simple protocol that provides access to the PRU modules and ALPIDEs in addition to the processor, and chapter 7 discusses the implementation of the embedded software that runs on the MicroBlaze CPU.

Firmware

The pCT control system, whether used in the development phase or for the complete prototype, is dependant on PRU firmware in order to implement the functionality described in chapter 4. Considerations that must be taken in this regard, as well as the implementation of these elements on the VCU118 platform, is detailed in this chapter. In chapter 4 it was also described how it would be beneficial if during the development stage the control system could assist in data readout. To achieve this, an extendable DMA-based solution was developed, and this is also described here. Considerations that will become relevant as the project progresses are discussed towards the end of the chapter. Chapter 7 describes the software on the embedded processor that controls the firmware discussed in this chapter.

5.1 Requirements

The PRU firmware related to the pCT control system must provide the communication channels, the processor, memories, and any necessary support modules.

Although very high performance is not required of the Ethernet link, some demands can be made based on the estimates made in section 4.1.2, where it was said that streaming individual pixel masks to a layer of 108 hypothetical ALPIDEs where each chip required 30% of its pixels masked would be completed in about 6.72s or 67.2s on saturated 100 Mb or 10 Mb links, respectively. Although both numbers are tolerable and that this pixel number was explained to be unreasonably high, it is preferable to minimize the time spent on configuration. When not in the configuration phase, the required bandwidth will be low, as described in section 4.1.4. For the early testing- and development phase, there is the additional consideration stemming from the single RJ45-connector and Ethernet PHY on the VCU118 board. This requires any Ethernet MAC used to be capable of gigabit-speeds as this port will be shared between readout-data and control.

Additional demands are placed on the CPU when it is involved in the data readout process, as it implements the User Datagram Protocol (UDP)- and TCP/IP protocols. During development the processor should therefore be optimized for per-

formance, whereas in a system where separate links carry control- and readout-data, this will be a lesser concern.

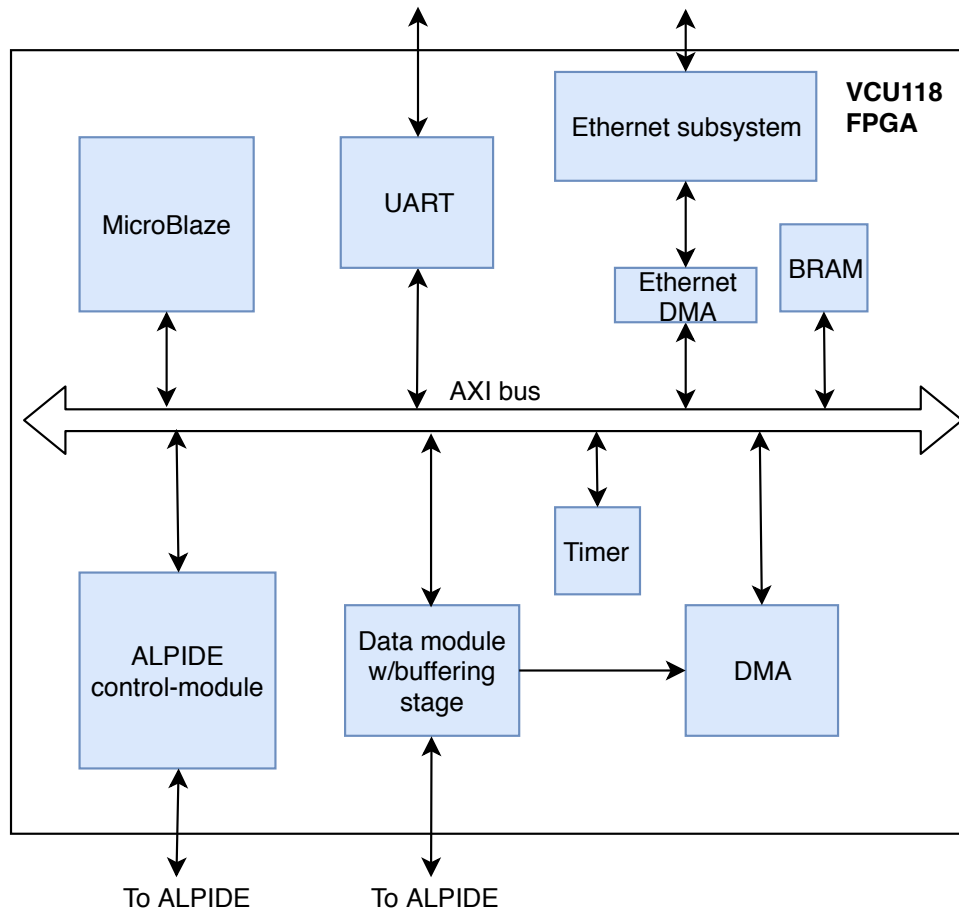


Figure 5.1: Simplified diagram showing modules that relate to the control system and development-stage data readout, as implemented on the FPGA on the VCU118 board.

A further requirement is that the use of block RAM be should be minimized, and external RAM preferably avoided. The BRAM is to be used for ALPIDE data buffer-stages, while external RAM consumes a large amount of I/O which is required for interfacing the chips. Although there is 5 GB of external DDR4 RAM on the development board, the system is built with this in mind. The lightweight OS aids in this regard, and during testing¹, acceptable performance was achieved while keeping the total memory usage below 1.2 MB.

The processor requires a small amount of block RAM in addition to an interrupt controller. Furthermore, any use of an operating system requires a timer in order to produce a *system tick*, and an AXI timer can fill this role. A debug module should be enabled as well as exceptions as this catches many silent faults that would otherwise go unnoticed. If the program and data are stored in external RAM, then the processor should in addition be equipped with instruction- and data caches of maximally allowable sizes. If however only block RAM is used, no caches are needed

¹See chapter 8.

since they provide no benefit in this case. No demands are placed on the serial UART interface.

5.2 Implementation

Firmware capable of satisfying the requirements above was implemented on the VCU118 platform, as part of the pre-existing design which consisted only of the single ADM, ACM, and the external ALPIDE that these interface to. Xilinx IP was used wherever possible as this simplifies compatibility with other components already present as well as future versions of Xilinx software.

While the solution for data-readout will eventually need replacement due to the amount of data produced by the full system as will be explained in section 5.3, the remainder of the modules contained in figure 5.1 and detailed in this section will be suitable for all stages of the design process.

5.2.1 Ethernet Subsystem

In order to handle reception and transmission of Ethernet frames, one or more modules that can handle the lower-two layers (*physical* and *data-link*) of the OSI-model is needed. While the physical layer is handled by an off-chip Ethernet PHY, everything above this layer must be realized by the FPGA. With regards to the control system, the internet- (IP) and transport (UDP/TCP) layers will be implemented in software running on the MicroBlaze, which leaves only parts of the Link-layer.

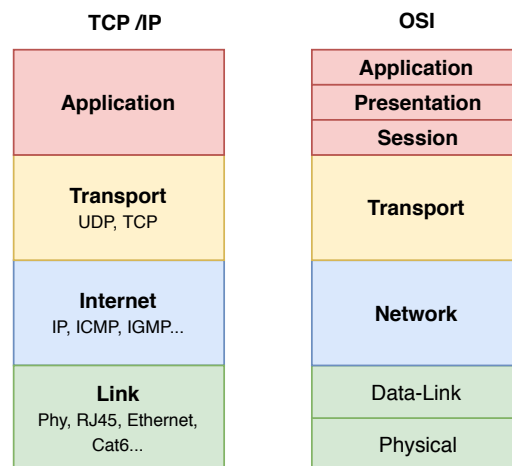


Figure 5.2: The TCP/IP- / OSI stacks and overlap.

Ethernet is common functionality on FPGAs, and Xilinx offers IP-cores that can fill this role. The license-free AXI Ethernet Lite can not be used during development as performance is limited at 100 Mb/s. The Tri-Mode Ethernet-MAC ("Tri" for 10/100/1000 Mb/s modes) however is an option. The MAC implements functionality defined by IEEE 802.3 [28], which specifies the link-layer and by extension the responsibilities of the MAC:

- Data encapsulation (transmit and receive)

- Framing (frame boundary delimitation, frame synchronization)
- Addressing (handling of source and destination [MAC-] addresses)
- Error detection (detection of physical medium transmission errors)
- Media Access Management
 - Medium allocation (collision avoidance)
 - Contention resolution (collision handling)

This core is by itself (together with the PHY) sufficient for speeds at either 10 Mb/s or 100 Mb/s. For 1 G and above, two additional layers are required: the Physical Coding- and the Physical medium Attachment Sublayer. This functionality is delivered via another Xilinx IP-core, and together with the MAC these make up the bulk of the *1G/2.5G Ethernet-Subsystem* [29], which was implemented as part of the design on the VCU118 platform.

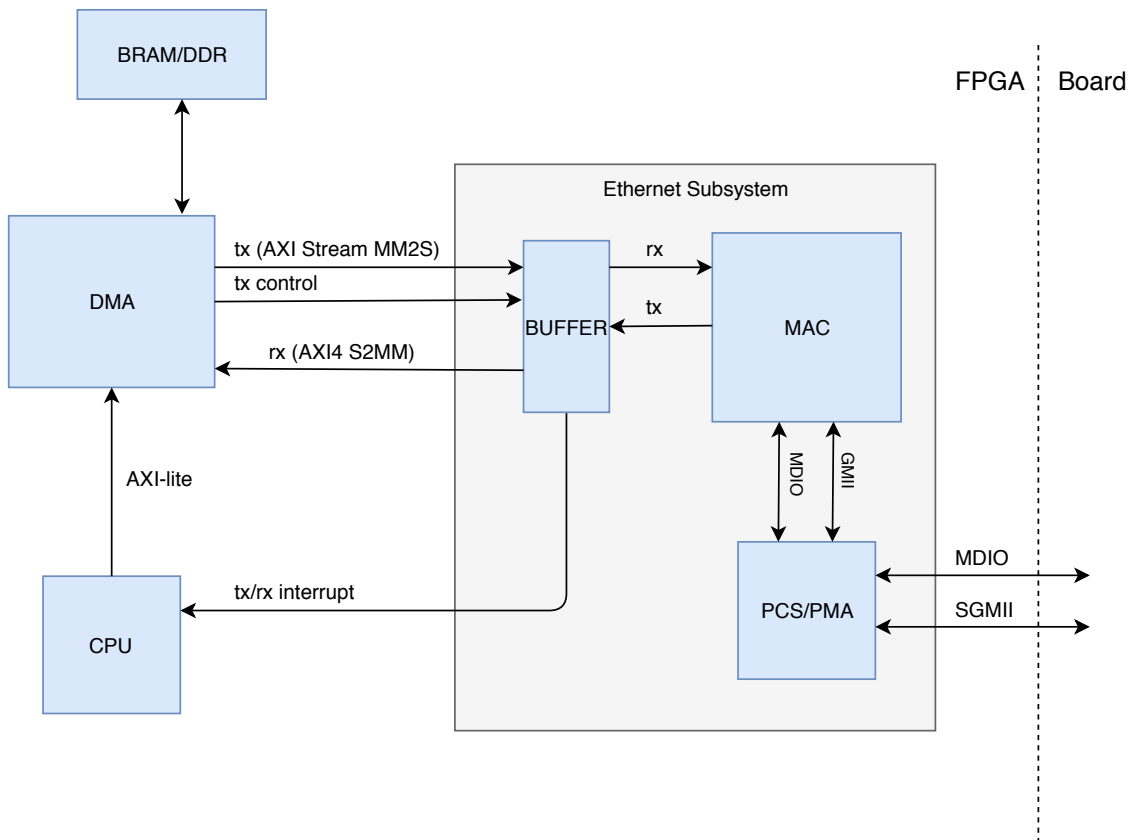


Figure 5.3: Blocks central to Ethernet-functionality, as implemented on the VCU118 FPGA.

The subsystem was in turn connected to an AXI DMA core, the interrupt controller for the MicroBlaze, and the external PHY as shown in figure 5.3. Furthermore, both RX- and TX-channels were configured for full firmware-offload of the TCP-, UDP-, and IPv4 checksums, as this significantly improves performance when using a software implementation of the TCP/IP stack.

This core will eventually provide insufficient performance for data-readout. At that

stage a transition must be made; likely to the UltraScale Integrated 100G Ethernet Subsystem in the case of the VCU118. Detector-data must in this case be read out via the SFP28 interfaces on the board.

5.2.2 MicroBlaze Configuration

The processor was configured for maximum performance to allow it to assist in data readout from the VCU118 board. This includes adding to it the barrel shifter, basic FPU, 32 bit integer multiplier and -divider, as well as the pattern comparator. As explained in section 5.1, these features might not be necessary if the processor is not involved in data readout. The MicroBlaze application requires only a small part of the internal BRAM, hence no caches are needed. An interrupt controller was also added in order to handle the interrupts generated by the Ethernet subsystem, UART, and the DMA-system to be described in section 5.3.

5.2.3 UART

The UART itself can be of any type, but should be compliant with the AXI-interface. The design implemented on the VCU118 uses the AXI UART16550 [30] as it allows for greater customization than some simpler options including configuration of the threshold at which the RX-interrupt fires; triggering when either 1, 2, 4, 8, or 14 bytes are present in the hardware receive-FIFO. Many UARTs will fire an interrupt as long as there is even a single byte in the receive-buffer, which wastes processing time spent in ISRs. It is also possible to configure the baud rate by accessing a firmware register, while the UART lite for instance requires re-synthesis for such a change.

5.2.4 Monitor Module

The monitor module discussed in section 4.1.4 was not implemented, as focus was placed on providing communication interfaces and functionality for data-readout. This can likely be a simple module, however. It should be AXI4-lite compliant, and must feature an external I²C interface. Registers must be added that allow for control of the regulators, and to provide access to the values read by the external ADCs. Functionality that allows the module to perform a power cycle via the regulators if it reads ADC values in excess of some threshold over a given time interval is also required, if this is not automatically performed by the regulators. Both the threshold and time interval should be configurable parameters of the module.

5.3 Readout of Detector-Data

The complete pCT system will consist of PRUs that each interface to 108 ALPIDEs² which at run-time each produce up to 1.2 Gb/s of data. This is reduced to 960 Mb/s after the data module removes the 8b10b-encoding, resulting in a combined worst-case output of 103 Gb/s per PRU. This places high demands on bandwidth, which are lessened by the fact that the beam is pencil-shaped. This has the effect of clustering hits around the center of a layer, leaving the outer detectors idle much of the time. In addition, *IDLE*-words are filtered away by the data modules as explained in section 3.4, and even if a chip is registering hits, this does not equate to a chip registering hits at full capacity.

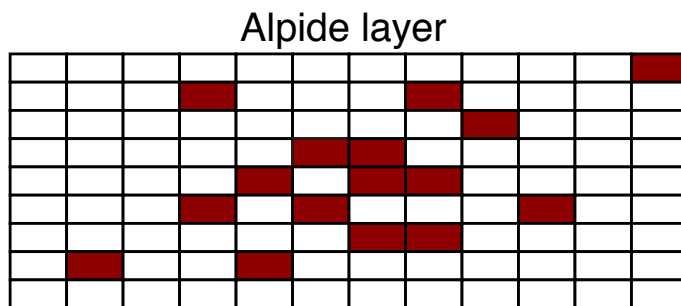


Figure 5.4: Chips that are closer to center of a detector layer receive the majority of the hits.

Combined, this will reduce the amount of data to be read out. Still, no simulations that predict the volume of data that will be produced, other than the worst-case scenario above, currently exist. Producing these simulations should be a top priority for the pCT team.

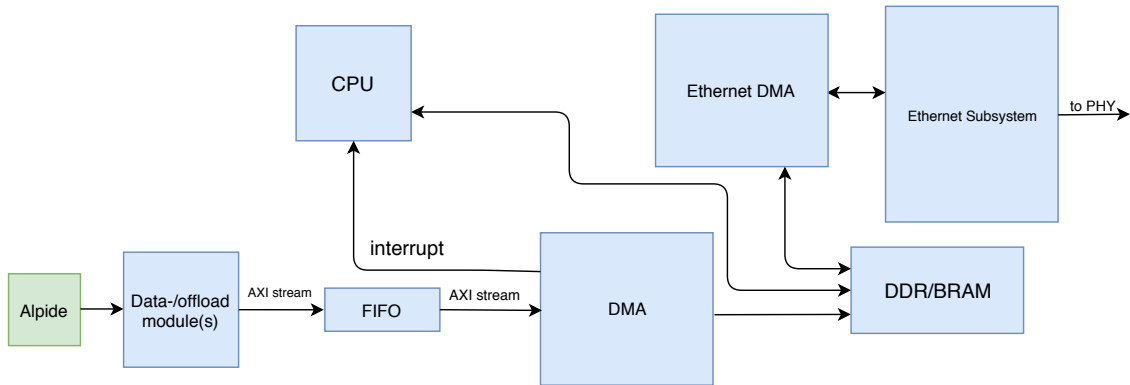
5.3.1 Development-Stage Data Readout

Design and verification of the full offloading-solution required by the complete pCT falls outside the scope of this thesis, but a subsystem capable of offloading the data produced by a single- or possibly a stave of ALPIDEs was implemented to aid the development-process towards the full design. The supported number of devices is partly dependant on the degree of ALPIDE threshold settings, and partly by the upper limit set on the TX-throughput by the MicroBlaze-, FreeRTOS-, and LwIP-system.

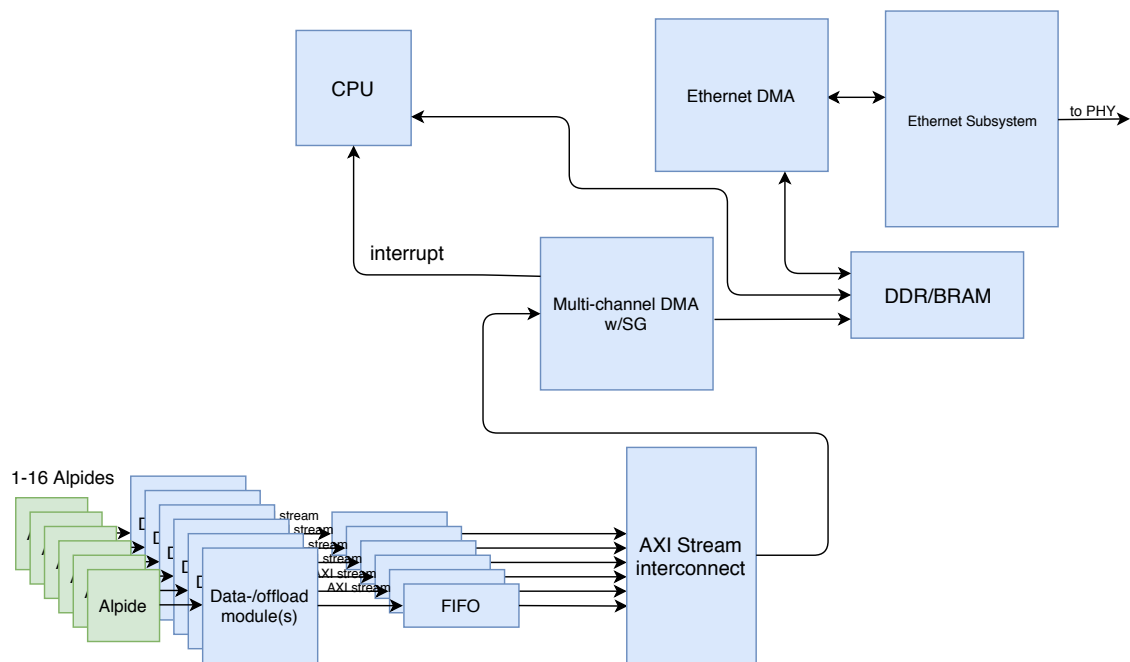
The subsystem involves first buffering data produced by an ADM with an AXI Stream FIFO, which also serves to bridge the clock-domains that separates the ALPIDE data module and the MicroBlaze Subsystem. An AXI DMA-block receives data from the FIFO, and places it into BRAM. On completion of a transfer, the processor is notified via an interrupt, after which it transmits the data via TCP or UDP. The system was implemented to accommodate one ALPIDE, but can be expanded further by operating the DMA-engine in multi-channeled, scatter-gather mode, connected to the FIFOs through an AXI Stream Interconnect. Up to 16

²Given that each PRU handles an entire layer.

channels are available if using the AXI DMA module. Alternatively, the streams could be aggregated to a single FIFO-stage by a priority encoder-like module, of the type discussed in section 5.3.2. Elements central to the simple- and extended subsystem are displayed in figures 5.5a and 5.5b. For testing purposes, a simple counter that adheres to the AXI Stream protocol was written and used as a data source.



(a) Development-stage offload-system.



(b) Extended development-stage offload-system.

The results of testing the solution shown in figure 5.5a, as described in chapter 8, showed that acceptable performance was possible also with the MicroBlaze, but that this was dependant on larger packet sizes being used. The data-offload module should therefore be made to buffer a number of frames, before passing it to the FIFO from which DMA streams it into RAM; ideally a number that places the total number of bytes close to the maximum transmission unit (MTU) of the network. A timeout-mechanism will also be required in order to prevent a situation where the offload module has a partially filled buffer below this threshold for an extended period of time. This could also be performed by the processor by aggregating data from multiple transfers, and sending it only when some threshold is exceeded. The

software overhead introduced however means that this is not recommended.

A transfer is set up by first starting the stream-to-memory-mapped channel, followed by writing to two registers the destination-address and desired transfer length in bytes, where this last write-operation starts the transfer. AXI Stream is comprised of eight signals, but most are optional and application specific. Typically, three signals control stream transactions: `tvalid`, `tready`, and `tlast`³. A master that has data available indicates this by asserting `tvalid`, `tready` is indicated by the slave to signify that it is ready to accept data, and `tlast` is indicated by the master on the final transfer of a packet. A master transfers no data unless `tready` is asserted, and is not allowed to wait for this signal to be asserted before asserting its `tvalid`. In figure 5.6, a master transmits a packet over the duration of ten clock cycles, which it indicates on the tenth cycle by asserting `tlast`; data is shifted out on to `tdata` only if both `tready` and `tvalid` were high on the previous rising edge of the clock.

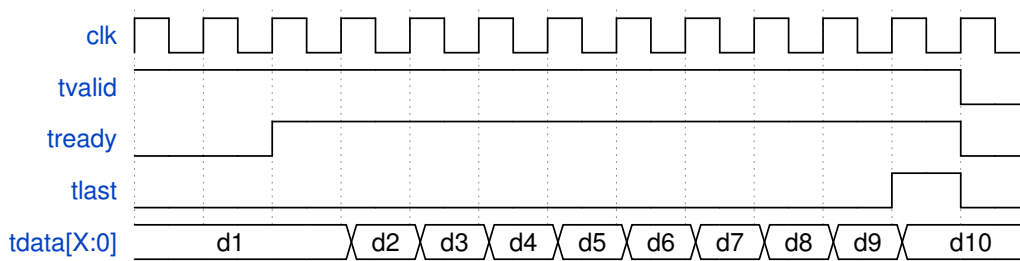


Figure 5.6: An example AXI-stream transaction.

Regardless of the set transfer-length, the AXI DMA module terminates a transfer upon the assertion of `tlast`, and thus less than the desired amount of data may be received. The actual amount can be determined by reading the contents of the transfer-length register after completion of the transfer. It is essential that a transfer-length greater than the largest possible packet that might be received is set, as undefined behavior can occur if the requested amount is transferred without `tlast` being asserted [33]. This also means that the stream-device on the receive side of the DMA module is obligated to generate this signal. Additionally, four pipeline registers on the stream-to-memory-mapped side will fill with data given that `tvalid` is set, even if a transfer is not initiated. If the stream-device contains stale data, this can be accounted for by flushing the stream-device before configuration of the module is performed.

5.3.2 Data-Readout in a Complete System

Due to the extensive bandwidth requirements, it is likely that the the data-readout procedure in the finished system should not involve a processor at all, except when used to configure the readout-subsystem itself.

The upper-bound of achievable throughput with a 100 MHz clocked MicroBlaze-system running LwIP in socket (multi-threaded) mode is listed in XAPP1026⁴ as

³In true streaming applications, this too is often not required.

⁴Newest version dated 2014.

approximately 70 Mb/s [32]. In chapter 8 it is shown that much higher performance is reachable with increased clock rates and use of *jumbo frames*. However, even if gigabit rates were achievable, this would still be insufficient for the number of ALPIDEs that will be included in the complete prototype. Using LwIP in *raw-mode* (single-threaded mode) greatly improves TX/RX-performance, with the system above achieving TX-throughput of 250 Mb/s; again it is expected that this number will in reality be higher. This however would exclude any multi threading on the processor, which would complicate software development, and is still far below what is required.

A Zynq platform (XC7Z020-CLG484-1) is listed as being able to attain much higher numbers due to its hard MAC and higher clock frequency, with 542 Mb/s and 949 Mb/s of bandwidth in threaded and single-threaded mode listed, respectively. Still, this will not be sufficient for the full system.

Eventually, any solution that utilizes DMA will be limited by the upper throughput of this block, but this limit is placed high: with a clock rate of 100 MHz and a bus-width of 32 bits, it lies at approximately 3.2 Gb/s in the stream-to-memory-mapped mode as shown in table 5.1. The block however supports higher clock rates, and the stream-width can be increased to 128 bits⁵. Virtex 7 devices are typically listed as comparable to Ultrascale devices in most Xilinx product guides, and for these, the upper supported clock rate is listed at 200Mhz-280Mhz, depending on the chip's speed grade. Higher data rates should therefore be possible.

Table 5.1: Xilinx DMA v7.1 figures at 100 MHz [33].

Channel	Clock Frequency (MHz)	Bytes Transferred	Throughput (MB/s)	Percent of Theoretical
MM2S	100	10000	399.04	99.76
S2MM	100	10000	298.59	74.64

Two of the limiting factors in a DMA based solution lies in transferring data in and out of memory, as well as the overhead introduced by software when setting up DMA descriptors and framing the data according to the network protocols. Another is the number of DMA modules that will be required for a full layer and the complexities that this brings; The AXI DMA module can interface up to sixteen AXI stream modules, and a 9x12 layer will thus require seven of these. Each must be set up with their own set of descriptors, and their completion interrupts handled. If this is handled by the processor, the software overhead will be significant.

For the complete prototype, a readout-system that excludes the CPU from this process should therefore be explored, as this will result in better performance and simpler implementation. Such a system could involve:

- AXI Stream data-FIFOs; one per data-module.

⁵This matches the width of the words produced by the ADM, and is done when testing the block as described in chapter 8.

- Arbiter(s) to aggregate the data streams.
- A larger FIFO with a programmable full signal.
- Dedicated UDP/TCP / IP-core(s) implemented in FPGA fabric.

The system would work by each data module first posting to a FIFO as it produces data. The arbiter monitors the output of the occupancy-counters belonging to the FIFOs, and selects the one with the highest value. Data is then passed through to a larger FIFO, which programmable-full signal can then be used to indicate that a transfer can be initiated by the UDP/TCP + IP core. More than one such subsystem might be necessary in order to satisfy the bandwidth demands. The reason that data is again buffered after the arbiter instead of streamed directly to the UDP/TCP core, is that it simplifies the arbiter itself. Without a second buffering stage, the arbiter must instead select the *ideal* combination of FIFOs on each clock cycle, in essence attempting to solve the Knapsack problem⁶, whereas with it, it is primarily a priority encoder. Figure 5.7 shows a block diagram of the components of such a subsystem.

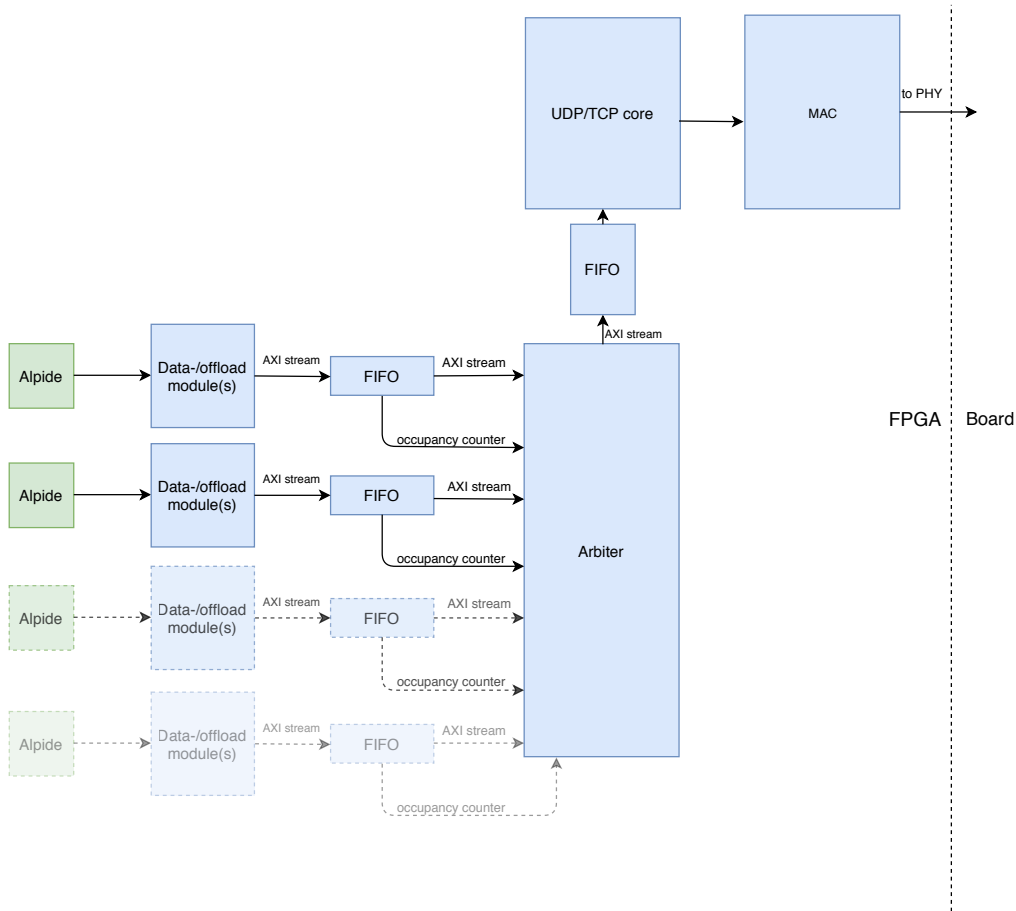


Figure 5.7: Offload system showing data offload modules, buffering stages, arbiter and UDP/TCP cores.

⁶Given a set of values S , find the subset S_u of S such that the sum of its elements is as close to, but not larger than, some target value K : $\max(\sum_{i=1}^{|S_u|} S_{u,i}) \leq K$

If the data modules handle timestamping and chip-/stave-identification, then the arbiter needs only to select the FIFOs to drain, and handle the AXI stream handshake between itself and the larger FIFO. Several alternatives for the UDP/TCP IP-core itself exists, and Xilinx offers multiple through third-party vendors, offering >10 G performance. For instance [34] or [35]. Alternatively, open-source options are available via OpenCores. The latter is suitable in the development-phase in order to verify the operation of the remainder of the chain.

Registers and Interface

To facilitate communication with the processor, the arbiter should be made AXI-Lite compliant, and feature status- and control-registers. As a minimum, the following should be included: Statistics-counters should also be implemented:

COUNTERS_RST	TARGET_SUM	RUN/HALT
--------------	------------	----------

Table 5.2: Control register

FIFO_DEPTH	FIFO_THRESHOLD
------------	----------------

Table 5.3: FIFO depth and -threshold register

BYTES_OFFLOADED_UPPER

Table 5.4: bytes-offloaded counter register, 32msb

BYTES_OFFLOADED_LOWER

Table 5.5: bytes-offloaded counter register, 32lsb

OVERFLOW_ERROR_COUNTER

Table 5.6: FIFO-overflow counter register

The overflow-counter is incremented any time the arbiter reads an occupancy counter equal to the FIFO depth.

5.3.3 Other Considerations

In applications where *any* lost data is *completely* unacceptable, UDP cannot be used as a transport-layer protocol due to its unreliability. This is likely the case for the pCT when it is at a stage where proton-tracks are to be reconstructed; if a hit penetrates for instance three layers, but datagram-loss results in it being reported only in two, an incorrect value on the residual energy of the proton that made the track will be produced. Loss on high-quality modern hardware can be made low, but is not insignificant and can never be guaranteed to be zero; see for instance [36]. In this case, there are two options if readout is done using Ethernet: Develop an application layer protocol for UDP that ensures reliability, or use TCP. To meet bandwidth requirements, this must be offloaded to firmware.

CERN has previously used UDP in the ALICE experiment, where the detection (but not recovery) of lost datagrams was made possible by prepending datagrams with sequence numbers [37], and currently deploy TCP in readout-builders⁷ used in the CMS, where 4×10 Gb/s links carry a modified implementation of it, removing some features considered redundant in order to improve efficiency [38]⁸. Of course, the data offloaded must be stored host-side, but this should not be an issue, and servers with the necessary performance are easily found. Lastly, it is again noted that TCP is a streaming protocol with no concept of "packets", and so the format of the ALPIDE data transmitted from the PRUs must therefore be either delimited, length-prefixed or constant in length. The current format uses the latter approach, with a word originating from a data module always consisting of 128 bits.

⁷Aggregates collected data to form full events.

⁸Removes for instance the TCP close- and listen mechanisms.

Control Message Format and Protocol

In chapter 4 the roles of the pCT control system was specified, and these often involve reading- or modifying values related to memory mapped devices located on the PRUs. For this to be possible, messages that instruct these actions to be performed must adhere to a specified format and protocol; when requests for register-values are received, a certain response should follow, etc. Malformed packets must also be handled; the board can be interfaced via a UART, which has no way of detecting and handling corrupt packets except for parity checks. Although one of the features of TCP is automatic error-detection of corrupt protocol data units (PDU), it will bring *any* data into the application layer, whether or not it conforms to a protocol. In addition, as both TCP and UART are streaming interfaces, packet boundaries must be indicated.

This chapter develops a simple application level protocol for communication between a host and a device that allows for transfer of arbitrary data, and uses it to transfer information regarding the memory space of a device. It is not dependant on use specifically with the UiB pCT or even an embedded processor, but does define special (optional) functionality for targeting the ALPIDE chips. It is suitable for use with all streaming interfaces, such as UART and TCP/IP over Ethernet. A few similar protocols were found, but these consisted either of proprietary hardware/software and were not available (See *AXI over Ethernet* [25]) or targeted Wishbone buses [26]. In addition, the need for communication over the serial interface had to be regarded.

6.1 Requirements

Packets must first of all contain the information necessary to specify their destination, payload, and the nature of the request (read, write, broadcast, etc). The serial interface mandates that data sent over it must feature a form of error detection so that corrupted packets can be detected and discarded. Error correction of malformed packets might not be necessary, but being capable of detection is a must. Simple parity-checks are not sufficient as errors are hidden if the number of bit-errors are even. A better approach is to append a cyclic redundancy check to a

packet, and to also include a mechanism that allows a receiver to re-synchronize if such bit errors occur. This is only applicable for the UART interface, as TCP/UDP PDUs include their own checksum.

Furthermore, although very high performance is not required for a control channel, a data format should have a high payload-to-framing ratio to maximize the useful bandwidth. It is also required that packets are allowed to vary in size, as it should be possible to embed several commands in a single packet. In addition, a sender should be made aware of whether or not a sent command was executed successfully. For generality, the protocol should be usable in any project that feature a remote processor or microcontroller with access to memory mapped peripherals.

6.2 An Application-Level Protocol

A control-data format is defined where packets of data are sectioned into three parts: a *header*, *payload*, and a *trailer*. The header specifies the type of command(s) contained within the payload and its length, the payload contains the actual data, and the trailer a sequence number that lets a sender associate a sent message to a received reply. The length field occupies 2 bytes, allowing for messages to contain between 1 and 65535 bytes of data. Buffer limitations (especially) on the board side will usually place a practical limitation lower than this. Furthermore, transmitting data over unreliable mediums such as serial interfaces mandate that packets must contain information to aid in locating the beginning of a packet; a receiver might connect while a message is partly sent, or data might be corrupted, causing a receiver to misinterpret received data. A lightweight byte-stuffing technique and a checksum is used to ensure these situations can be recovered from. The built-in reliability of TCP means that it requires neither of these.

6.2.1 Packet Format

The base format of a packet is shown in 6.1

HEADER(3B)	Payload(1B-65535B)	TRAILER(1B)
------------	---------------------------	-------------

Table 6.1: Application level protocol - Base packet format.

LEN(2B)	CMDTYP(1B)
---------	------------

SEQNUM(1B)

Table 6.2: Application level protocol - Header and trailer, respectively.

For reliable, streaming interfaces¹, sending data packaged as shown in table 6.1 is sufficient. Any type of data can in theory be contained in the payload. Values in all fields are sent in big-endian form.

¹Also works for packet-based transport-layer protocols, although the length-field is in this case redundant.

6.2.2 Considerations for Unreliable Interfaces

To resolve situations where a receiver and sender become de-synchronized, or to avoid those where a receiver would issue erroneous commands due to having received corrupted data, streaming data sent over unreliable interfaces must be delimited and preferably attached a checksum. In the case of the protocol discussed here, this is achieved by appending a 16 bit CRC of the CCITT type to the trailer of a packet, while delimiting is achieved by *byte stuffing*.

6.2.3 COBS

Data sent via unreliable interfaces must be properly delimited if the size of packets is to vary, and if a receiver is expected to recover from bit errors. Consistent Overhead Byte Stuffing (COBS) [27], is a simple encoding scheme used for this purpose, reserving use of the value 0x00 for use as a delimiter. It carries a fixed overhead of one byte per 254 bytes of data, plus one additional byte appended to the end of the message which might consist of one or more such blocks. The first COBS-byte functions as a *pointer* to the next byte in the message that in un-encoded form would be a 0x00-byte. This byte is encoded with a value specifying the relative position of the next encoded 0x00, and so-forth. If no such byte exists in the block, the first COBS-byte simply points to the end of the message, where an actual 0x00 should be located. The example below illustrates the technique; red bytes being the encoded components.

un-encoded	01 04 00 12 03 A1 00 00 CA
encoded	03 01 04 04 12 03 A1 01 02 CA 00

Table 6.3: Consistent overhead byte stuffing example.

Packets longer than 254 bytes can also be encoded; if a COBS byte has the value 0xFF, this implies that 254 data bytes follow, but that *no encoded zero is to be found at the end*, but instead a regular COBS byte. Larger packets can therefore be encoded with the use of one or more such max-length blocks, followed by a final block delimited by a zero. Arbitrarily sized packets can thus be encoded with COBS, with the upper limit in this case set by the length field.

Including both COBS as well as a length indicator is not redundant as it aids in detecting malformed packets. It also means that a receiver is not required to check every received byte for the delimiter-value, but can instead receive a larger chunk as specified by the length field.

For unreliable streaming interfaces, a packet as shown in table 6.1 is thus appended a checksum, COBS-encoded and delimited. This ensures reliability, and allows a receiver to quickly re-synchronize on the next received packet if an error occurs.

COBS1(1B)	HEADER(3B)	Payload(1B-65535B)	TRAILER(1B)
CRC(2B)	COBS2(1B)		

Table 6.4: A packet appended with a 16 bit CRC and byte-stuffed with COBS.

6.2.4 Packet Fields

This section provides an overview of the fields of a packet and their purpose. A thorough description as well as examples can be found in the appendix.

COBS1

The first COBS-byte which always appears at the start of a message, pointing to the position of the next encoded zero-byte in the block. If none exists, it points to an additional COBS byte if there is more than one block, and if not, to the final delimiting zero-byte.

CMDTYP

Indicates to a receiver how to interpret the payload that follows. Three types were defined to be used with the PRU, indicating data meant for one or several ALPIDEs, PRU peripheral, or if the packet is of the reply type.

LEN

Specifies the number of bytes of *payload*.

PAYLOAD

The payload of a packet contains its usable data. Arbitrary data can be contained in the field, requiring only a matching CMDTYP. For PRU purposes, three types were defined as specified by the command types mentioned above: ALPIDE data, via the ACM, peripheral data, or special commands directed at the CPU.

PAYLOAD - Memory-Mapped Module

Payload intended for memory-mapped modules follow a simple format, requiring a write-code and the 32 bit register value and -address in the case of a write, or a read-code and the address in case of a read:

WRCODE(1B)	REGVAL(4B)	REGADDR(4B)
RDCODE(1B)	REGADDR(4B)	

Table 6.5: Payload - format for register writes- and reads, respectively.

PAYLOAD - ALPIDE

A payload type that addresses ALPIDEs is defined. This has the effect of reducing the number of bytes necessary to access one or more ALPIDEs, while also allowing software on the PRU to handle the timing of writes performed to the ACM. Packets intended for one or more ALPIDEs follow the format as given by the ALPIDE user manual, with two modifications: a field designating the target stave (STAVEID) is added in addition to a NSNGL-field allowing up to 7 messages to be written to one chip without repeat of the opcode-, chip ID- and stave- fields. The value of this field corresponds to the number of messages that follow.

The use of the NSNGL field reduces the number of bytes needed to perform multiple writes or reads. This is useful for instance during configuration. In addition, one of the primary tasks of the PRU CPU is to transmit housekeeping data; this involves reading the ALPIDE ADC registers of which there are 22, and the NSNGL field allows these to be contained within three sub-message in a compact way.

OPCODE(1B)	CHIPID(1B)	STAVENSNGL(1B)	REGADDR(2B)
REGVAL(2B)			

OPCODE(1B)	CHIPID(1B)	STAVENSNGL(1B)	REGADDR(2B)
------------	------------	----------------	-------------

Table 6.6: Payload - ALPIDE register writes and reads, respectively.

OPCODE(1B)

Table 6.7: Payload - ALPIDE broadcast opcodes.

CRC16

Field containing the CRC16 as calculated by the sender. A sender calculates this using the non-encoded contents of a packet as input. The receiver performs the same calculation on the received packet after decoding the message, removing the COBS-bytes and excluding the received CRC itself, and validates that it matches the received value.

SEQNUM

Contains an 8 bit sequence number that can be used to relate a received reply to a sent command. A sender increments the number any time it transmits a message, while a receiver stores it and appends it to its reply.

COBS2

Unique identifier signaling the end of the message; always 0x00.

6.2.5 Message Replies

Packets ideally arrive intact and correctly formatted, after which data is written or read or some other operation performed. Alternatively, the framing could be erroneous, the sequence number out of order, the command type unknown, or contain some other error. A receiver handles this by sending a reply containing a status code specifying what (if any) type of error occurred, in addition to data related either to the error or the successful reception. A reply follows a format identical to that of a command, with its sequence number that of the command to which is being responded. Like commands, replies must be COBS encoded and appended with a CRC when required.

See section C in the appendix for a description of the defined status codes that can be contained within replies, as well as the general format in the case of responses to read-/write requests.

LEN(2B)	REPTYPE(1B)	Payload(1B-65535B)	RECVDSEQNUM(1B)
---------	-------------	---------------------------	-----------------

Table 6.8: Application level protocol - Reply packet.

6.3 Addressing ALPIDEs via a Peripheral Command

The ACMs are memory mapped components on the PRU, and hence the ALPIDEs can be manipulated through peripheral writes, i.e by packets with the CMDTYP field set to "peripheral". This however may result in timing issues, and requires more data to be sent over the Ethernet links.

Using the ALPIDE CMDTYP ensures the timing of writes performed to the ACM is kept. It was mentioned that three writes to this module are necessary to write to an ALPIDE, and two in order to perform a read. If the latter is attempted by writing to the ACM CTRL- and ADDR registers and immediately followed by a read of its READ_DATA register, invalid data will be found here due to the execution time of the ACM state machine. Software on the PRU CPU ensure that these types of errors do not occur.

Furthermore, the AXI bus on the PRU defines a 32 bit address space. The three writes to the control module entail three 32 bit addresses and corresponding values per ALPIDE-write; a total of 24 bytes. The same action performed using the format developed here requires only nine². For reads, the numbers are 16 and 5 in the respective cases.

²If several writes are performed by using the NSNGL field, the result is skewed even further. For seven messages written to the same ALPIDE, 31 bytes are needed when using the ALPIDE format, compared to 224.

6.4 Hardware Offloading of the CRC- and COBS Calculations

If required, calculation of the CRC can be offloaded to hardware where this is an option, as is often done for the checksum carried in a TCP/UDP PDU. Table-driven implementations of CRC calculations can be used however, and these are quite efficient as the calculation is reduced to a single XOR and table-lookup per byte. The XOR is a single-cycle instruction on many processors, including the MicroBlaze [19], and the table-lookup consists of collecting a single value from RAM. If the UART is operated at low baud rates, this is in any case a non-issue.

Likewise, one of the advantages of COBS is its low computational cost. In the worst case, a packet consists of a string of zeroes which must then all be encoded. In this case the algorithm must iterate over as many bytes as the packet is long, replacing each value as it goes. The overhead introduced by COBS is very low in general; on random data, the theoretical average is only 0.23% [27].

Software

In chapter 4, the roles of the pCT control system were defined, and the implementation of all necessary firmware in addition to a DMA-based solution for data-readout was detailed in chapter 5. In chapter 4 it was also argued that what is required at the current stage of development is a flexible system that can provide access to the address space of the device, and thereby the modules on the AXI bus, including ALPIDE(s); a simple protocol that could provide this functionality was developed in chapter 6. Software was written for the MicroBlaze processor in order to facilitate the above, which is described in this chapter. Complete code and additional documentation can be found in the pCT WP3 GitLab repository¹.

7.1 Requirements

The software deployed on the MicroBlaze processor is responsible for providing the communication links between a host and the PRUs. Through it, access to all PRU modules must be provided, so that these can be configured, controlled, and monitored. Functionality that can control the DMA transfer process is also required, and, additionally, it was stated that the processor should be capable of monitoring system-modules without external input. A driver for the ACM will also be beneficial, as this can handle the timing of the ACM state machine, and implement some of the more complex procedures that the ALPIDEs require, such as mask-application and ADC calibration. This would also simplify future development, should different software be deployed.

7.2 Overview

The embedded software implements a client/server-style architecture, where the PRU fills the role of a command server that accepts commands that adhere to the chapter 6 protocol. In chapter 4, FreeRTOS was chosen as a suitable OS for the embedded CPU, and the software is implemented as tasks running under this OS. Two tasks are dedicated to the serial- and Ethernet interface. These provide data to

¹<https://git.app.uib.no/pct/wp3>

a consumer task which interfaces to the greater PRU system, executing the received commands and returning any requested data. Another performs the automatic monitoring, while a final task controls the DMA subsystem developed in chapter 5. Networking is implemented by using the open-source LwIP TCP/IP stack. A block diagram of the components of the software as it was implemented on the MicroBlaze is shown in figure 7.1.

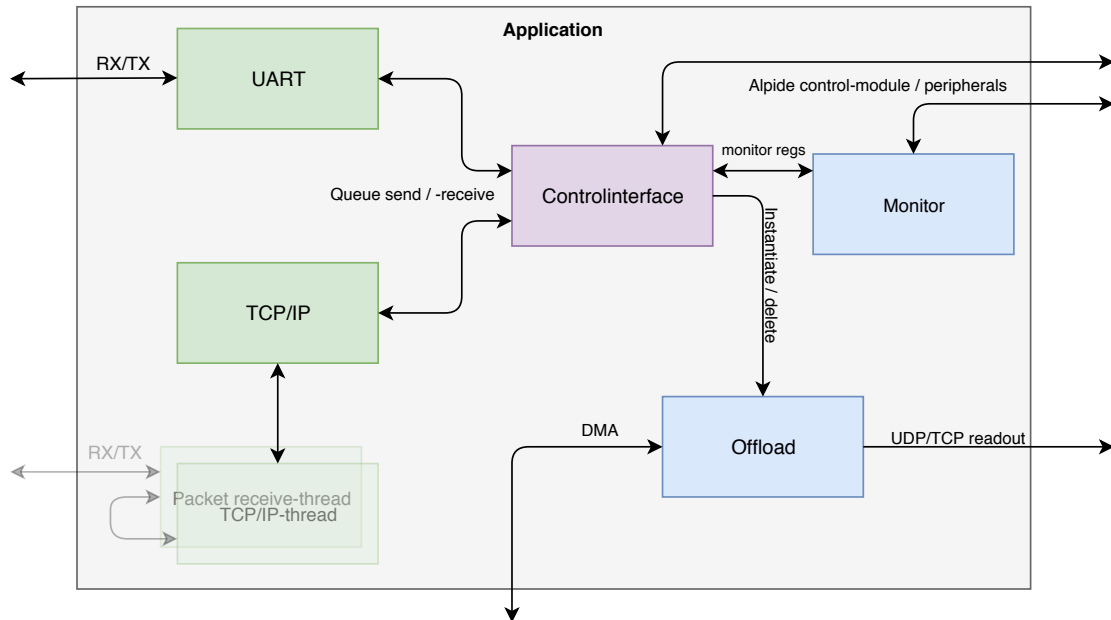


Figure 7.1: Central threads in the MicroBlaze application.

The software described here is developed primarily in the Xilinx Software Development Kit (SDK), and is written in C. At the time, the current version number of the OS was 9.0.1, and 1.4 in the case of LwIP. Control of the CPU from the host-side is performed via an API implemented in Python, which is described in chapter 8².

7.2.1 Development Principles

While developing the software, focus was placed on modularity so that the application could be easily expanded or used alongside other software. In addition, readability and proper documentation was prioritized, and complete Doxygen documentation was generated by commenting code in the Javadoc style. To keep memory use transparent, all FreeRTOS objects-, in addition to larger constructs such as receive-buffers, are allocated at compile-time. A modification of the OS distribution was required in order to do this, which is described in section 7.6.1. In an effort to make the application event-driven, subroutines were implemented using interrupts instead of polling, when possible.

7.3 Software Structure

The two threads responsible for reception and transmission of host - PRU control data are implemented as simple state machines. These verify the framing of received

²An overview of this API is also included in the appendix, section D.

packets, discard those that are erroneous, and forwards the remainder to a task that executes the commands contained within the packets. In the case of the thread handling the serial interface, received data is written to a software FIFO from an ISR any time an RX-interrupt is received, whereas the thread that receives and transmits packets via TCP/IP only calls its receive-function when a complete packet is not buffered, and always requests the maximum number of bytes that will fit into its buffer when it does so; one, several or no full packets might be received at a time in this case. Packets are delivered to the task that executes the commands via a queue. Figure 7.2 shows this process in the case of the UART-thread.

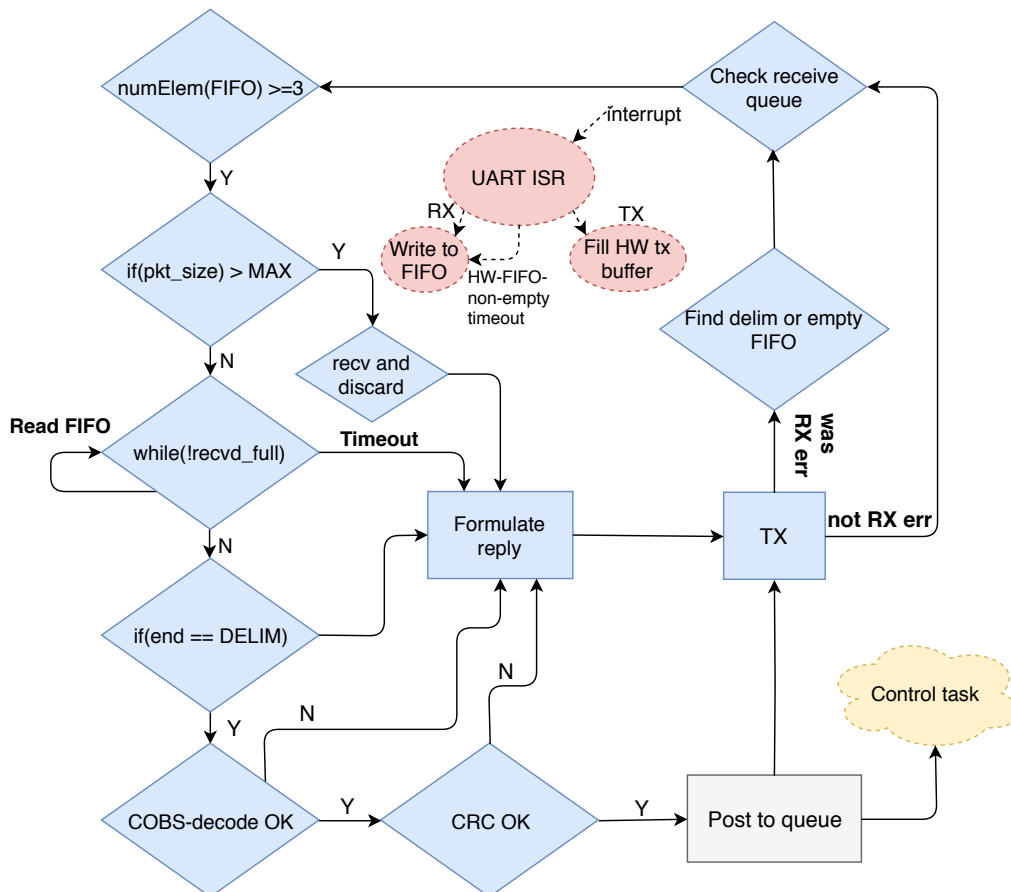


Figure 7.2: MicroBlaze application - UART-task flow-chart. The TCP task is similar, but is not required to verify encoding or CRCs.

7.3.1 Control interface

The control-interface task is responsible for processing the payload in the received packets, and executing the contained commands. The task controls only the payload, and not the framing. The CMD-field is checked for validity, and if valid, the task calls the function responsible for carrying out the specific command, which is either of the ALPIDE-, peripheral- or special command type. The packet is iterated over, with the number of messages in the packet indicated by the LEN-field. After each message, reply-data corresponding to the completed action is written to a FIFO, consisting either of a status-/error code or data. Finally, the contents of the FIFO is used to formulate a reply which is posted to the appropriate queue, determined

by the ID-field of the received message. Afterwards, the process repeats, with the task blocked and using no CPU resources as long as no data appears in its queue.

7.3.2 Data-Readout

The data-readout solution from chapter 5 is controlled by a task that is spawned and deleted upon reception of a command. It is kept in a tight loop in order to maximize performance. On creation, a flag is first set to indicate that it is active in order to prevent multiple instances being spawned. The task then opens- and binds to a socket, and optionally connects if configured for TCP. A DMA transfer is initiated, and the task is notified upon its completion through an interrupt; this interrupt fires after tlast is asserted and the last transfer is made on the outgoing side. This event also causes any other thread that might have been switched in during this period to yield to the data-readout task. A register on the DMA module is read to determine the number of bytes transferred which are then sent to a receiver. The task is not suspended while waiting for a transfer due to the latency that is incurred when waking it.

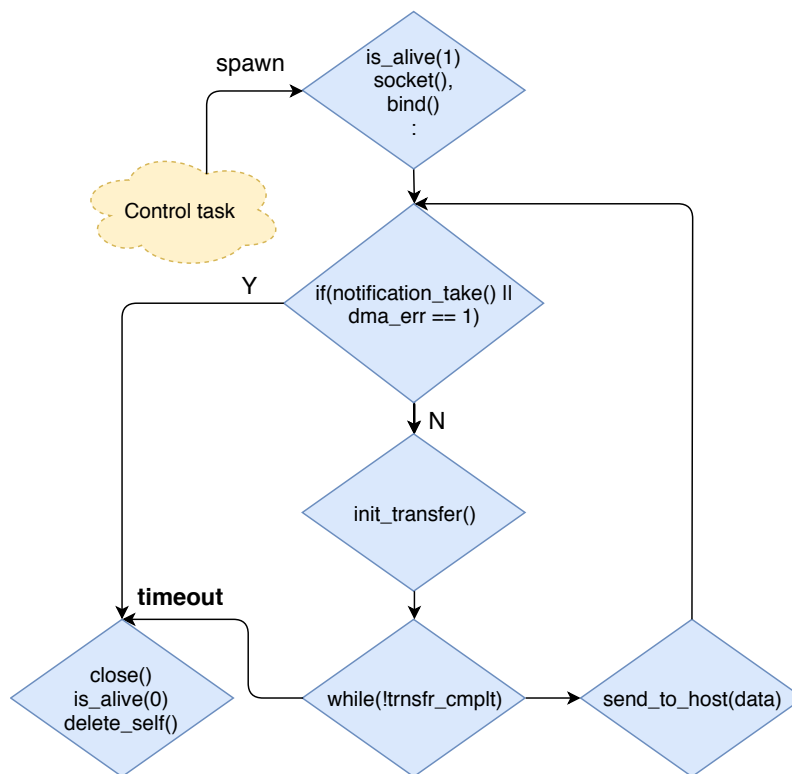


Figure 7.3: MicroBlaze application - Data-readout task flow-chart.

Instead of deleting the task directly from the control thread, a notification is posted to allow any ongoing transfers to finish, and for the socket to be properly closed. A timeout can occur if there is no data on the receive-side for an extended period of time, and a DMA error can be generated if the device that is being received from fails to generate the required tlast-signal, as described in section 5.3.1. The length of a timeout is configurable via a macro; this is a software timer as the module has no built in mechanism that provides this functionality. Figure 7.3 shows the task's behavior. As also stated in section 5.3.1, the ADM is made responsible for

providing the stream FIFO with adequately sized packets, although this could easily be implemented in software by aggregating smaller transfers until some threshold is exceeded, sending this and copying any remainder to the front of the DMA buffer and repeating the process.

7.3.3 Monitoring

The PRU should be capable of monitoring its devices without manual external input, as explained in section 4.1.4. This was achieved by implementing a high priority task that wakes at configurable intervals to perform measurements and act in case set thresholds are exceeded. Two data types are defined for this purpose, shown in listing 7.1.

Listing 7.1: Data types that facilitate monitoring of items.

```
typedef struct monitored_reg {
    u32 reg_addr; //!< The address to monitor
    u32 val_threshold; //!< Threshold set for the address
    /**
     * @brief Function to be called if value at regAddr exceeds
     *        valThreshold.
     * @param *ptrToSelf A pointer to the monitoredReg
     */
    void (*threshold_exceeded)(void *ptr_to_self);
} monitored_reg;

typedef struct monitored_alp {
    u8 chip_id; //!< ChipId
    u32 control_base_addr; //!< Base address of chip's control module
} monitored_alp;
```

A `monitored_reg` can thus be added to the list of monitored items by providing a register, threshold value and a callback. The ALPIDE type is simpler as any ADC value in excess of the set threshold on any ALPIDE is handled in the same manner. The thresholds for the chips are set in a separate header file. Monitored items and ALPIDEs are added before compilation. Being able to do this at runtime is not necessary, as a board is static in the sense that its modules do not change once it is programmed and running. The intervals at which the task performs work is configurable, as is the selection of ALPIDE ADC registers and thresholds at which these are limited.

As an example, listing 7.2 shows the adding of a monitored register with address `0x40000000` and a threshold value `0x0A`, that if exceeded causes the function `exampleFunc()` to be called, which simply prints the register value and sets it to `0x00`.

Listing 7.2: Arbitrary procedures can be called in response to an exceeded threshold.

```
static u8 add_all_monitored_items() {
    ...
    add_monitored_item(0x40000000, 0x0A, &example_func);
    ...
}

static void example_func(void* ptr_to_monitored_reg) {
    monitored_reg *item = ptr_to_monitored_reg;
    xil_printf("Register at address %x has value %x. Threshold %x\n",
              item->reg_addr, Xil_In32(item->reg_addr),
              item->val_threshold);
    Xil_Out32(item->reg_addr, 0x00);
}
```

Any type of functionality can in this manner be related to a monitored item. In addition to monitoring the ALPIDE ADC values, the task can for instance be set to monitor the phase alignment process of the ADM, resetting the module if errors occur within some given time frame as explained in section 4.11.1 of [39]. Due to the relatively long (minimum) execution time (>15 ms) of an ALPIDE ADC full-read operation, monitoring does not cause appreciable activity on the AXI bus.

7.4 ALPIDE Control Module Driver

A driver was developed to interface to the ALPIDE control module. This driver by extension allows execution of most ALPIDE functionality; writing-/reading registers, calibration of the on-chip ADC, application of chip-specific configurations, conversion of raw ADC-register values to corresponding (volts, amperes, celsius) values, etc. Functionality to apply the pixel masks is also provided. Some of this functionality was also implemented host-side through the Python API, but should future developments offload a greater degree of processing to the PRU CPU, this driver can be leveraged further. The driver also maintains timing of writes performed to the ACM in regards to its state machine.

7.5 Data-Exchange Between Threads

Tasks in multi-threaded applications must often exchange information. In FreeRTOS, this is done primarily via queues which hold data of arbitrary type, and where the data is passed by value. In the embedded software discussed in this chapter, this data consists of messages in the form of structs, and a few techniques for sending these between tasks were explored before deciding on a *buffer pool* solution, which are outlined here.

One possibility when passing data is for the sender to keep a struct variable. A pointer to the struct can then be copied onto the queue. This allows the queues to be compact, only occupying whatever the pointer-size is times its capacity, but results in issue due to the asynchronicity of the threads; messages might be posted faster than the receiver can handle. If the pointers point to locally declared variables, these might be overwritten before they are fully processed by the receive, which forces the queues to be of size one.

Another option is to copy the structs themselves on to the queues. This is simple to implement, since for the sender there is no risk of overwriting posted-, but not yet processed data, and for the receiver there is no need to de-allocate memory or otherwise indicate completion of processing. Copying the structs is however resource intensive: each struct consist of an array of bytes equal to the maximum size of a message, an ID-field, and a socket-descriptor; copying this information is time consuming. In addition, the queues must be large in order to accommodate several of the structs.

A further solution is dynamic allocation: A sender can dynamically allocate memory to fit the given message, and a pointer to this memory passed. On reception, a

receiver dereferences the pointer, processes the contents and frees the memory. This is a common way to handle data exchange in multi threaded applications, and allows queues to be small in size. However, memory leaks can occur if care is not taken, dynamic allocation is generally not fast, and it has the additional drawback that run-time memory usage is obfuscated. Finally, the queues are sized at compile-time; If in the worst case it is expected that they might be filled, there is no reason not to allocate a queue-sized number of messages at compile time.

The final approach is to allocate a *buffer pool* of structs at compile-time. Tasks may take pointers from a FIFO that points to structs in this pool when sending messages. After processing the message, the receiver again puts the pointer in the queue, allowing it to be used again. This allows the queues to be compact, requires no dynamic memory allocation and allows better determination of program memory usage at compile time.

7.6 Software Configuration

Several settings that alter aspects of the embedded software such as IPs, ports, baud rates and queue-/buffer sizes are defined as C macros and can be set in a separate header file.

7.6.1 LwIP and FreeRTOS

Both the networking stack and OS can be configured through their respective configuration headers, or via the Board Support Package (BSP).

FreeRTOS v9.0.0 added support for static allocation of objects such as tasks, queues, mutexes and semaphores, but this is not included in the current (v2018.4) Xilinx SDK distribution of the OS. As memory usage should be well defined, this was made possible through alterations to this distribution, as well as to the .tcl-scripts that builds the OS; a section in the embedded-software documentation in the pCT WP3 GitLab repository describes this process.

Furthermore, both the OS and LwIP use a heap implementation that allocates a block of memory at compile-time, and provide their own implementations of the malloc()- and calloc() functions. The heap defined in the linker script is hence only used when calls are made to the C standard-library malloc(). If this is never done³, this is wasted memory and should be set to its minimum value. Likewise, the stack defined in the linker script is only used when objects are allocated before the scheduler is started. This is only done in main(), and is therefore set to a low value of 512 bytes. The OS heap was set to 262 144 bytes and the tick-rate left at its standard frequency of 100 Hz. Tasks that block before their allotted time (10 ms in this case) is spent are immediately switched out, and higher switching speeds were found to be detrimental to performance; explained by the overhead introduced by the increased number of context-switches.

³This is the case for the application described here.

LwIP can be configured extensively, and this can make a large impact on its memory consumption. The Ethernet subsystem was set up to enable checksum-offloading, and this must also be enabled in software. In addition, the TCP window is increased to its maximum size of 65 535 bytes, the send-buffer to 16 384 bytes, and the maximum number of queued TCP segments to 512.

7.7 Future development

Supervisory control and data acquisition (SCADA) systems allow for monitoring and configuring of individual components that make up a system. On the host side, such systems should eventually be interfaced to the PRUs. SCADA systems are typically hierarchical, with some human-machine interface that displays data, a central computer or a group thereof that acts on the received data and sends commands to lower-level modules (typically PLCs in an industrial setting) which in turn communicate the commands to sensors, valves, relays etc. Ethernet is often used in the physical- up to and including the transport layer, while standardized protocols fill the application layer (Modbus TCP, OPC, SNMP, and MQTT are some examples).

While Modbus will not suffice due to the 16 bit-address limitation, some of the other alternatives may be suitable. MQTT lies on top of TCP/IP and is a protocol developed in order to service IoT devices over low-speed networks. It is based on a publish/subscribe-model where clients subscribe to *topics* that are then delivered by a broker. A topic can contain a hierarchy of information, each separated by a slash; the temperature as read by an ALPIDE for instance could be published as *boardW/staveX/chipY/tempZ*. In turn a PRU could subscribe to the topic *board-W/staveX/chipY/setReqZ* and alter register-values in this manner. Wildcards also allow several boards to subscribe to the same topic, or a board to subscribe to all available *temperature*-topics, for instance. Several options for embedded deployment of MQTT exist, with one of the larger being the Eclipse Paho project. This includes support for FreeRTOS, but with a different socket-layer that would have to be modified [21]. Additionally, recent additions to the AWS-FreeRTOS repository includes an MQTT client implementation and example⁴. MQTT is agnostic to the contents of the packet-payload, and as such primarily provides *transport* of data between parties, with the format and handling of this data having to be defined by the parties. Table 7.1 shows the format of an MQTT packet, where the fixed header indicates packet type and -length, and quality of service. The variable header defines the message- and client ID as well as the topic. The payload contains any data, if present.

Table 7.1: Showing the format of an MQTT packet.

Fixed header(2B)	Variable header(1B-4B)	Payload(0B-NB)
------------------	------------------------	----------------

Another alternative is OPC, which unlike MQTT is built on a client/server-type architecture. It encompasses a number of standards, and was initially developed to "*... abstract PLC specific protocols ... into a standardized interface ... [that] would convert generic OPC read/write requests into device-specific requests and vice-versa.*" [22].

⁴<https://github.com/aws/amazon-freertos>

The modern iteration of OPC is OPC UA which among other features include platform independence, and a focus on scalability and security. A strength of OPC UA is its information model, allowing the definition of abstract data types and concepts typically associated with object-oriented programming languages such as objects, inheritance and instance methods. In the context of a pCT control system this could be used to define a base type *board module*, having a name, address and read()- and write() instance methods. From this an *ALPIDE module* could inherit, defining functionality specific to it, etc.

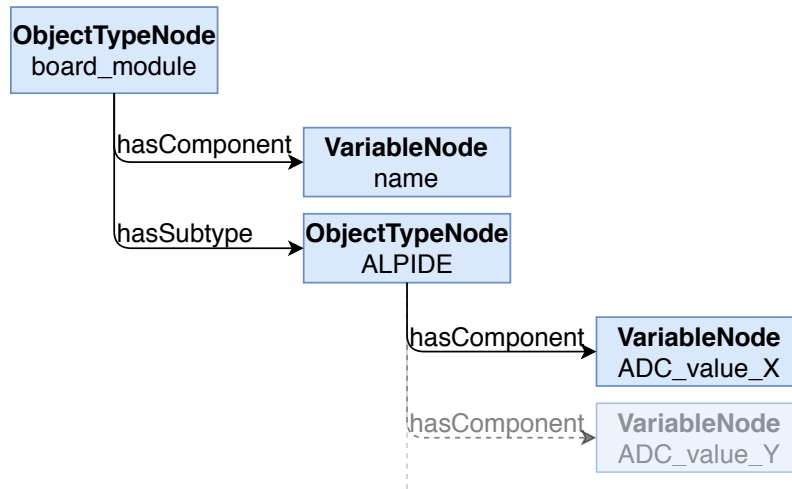


Figure 7.4: Illustrating the OPC UA information model.

This standardized data representation has the further advantage that it can be understood by *any* part of a system that implements OPC UA, something the undefined MQTT payload does not allow. Furthermore, OPC UA allows the content of VariableNodes as shown in figure 7.4 to be related to arbitrary data sources; one approach that is used in the CERN ATLAS experiment is to use a low-level protocol providing memory-mapped access to connect readout unit FPGAs to an OPC UA server on the host-side [23] [24], where a low-level interface and a software application collects register values and stores them in a buffer. The OPC UA server (or any other consumer) can then collect the values at their own pace. This allows for the protocol providing direct access to memory mapped FPGA modules to be used by itself, without other higher-level software. It also allows for such a lower level protocol to be used in any future projects where interfacing to an FPGA is necessary, but monitoring is not. On the other hand, the OPC UA server will provide good integration with commercial and open source SCADA solutions, and no OPC on the PRU side keeps memory usage low.

Open source servers/clients for both FreeRTOS, Linux and Windows exist; see for instance open62541⁵. The standard is complex and hence implementations of OPC UA are larger than MQTT. Speed is also likely to be lower, but for remote-monitoring, real-time is not required as stated in section 4.1.4. Additionally, the robust security features might be unnecessary for this project. A build of open62541 for FreeRTOS was however still less than 1 MB in size.

⁵<https://open62541.org/>

In industrial SCADA systems, often several OPC servers will exist. In the case of the much smaller pCT, a single server that serves the PRUs and a client implemented host side will suffice. In the future, the pCT might include other external modules such as motors, sensors, power supplies, etc. These might be serviced by other OPC UA servers that interface to the same client. As illustrated, the OPC server interfaces to the greater SCADA software. This will have to be chosen also. In doing so, the scale of the project should be kept in mind, i.e many of these systems are designed for use in large factories and facilities. Software such as *UaExpert* by *Unified Automation* can be used to provide a client that interfaces to the server.

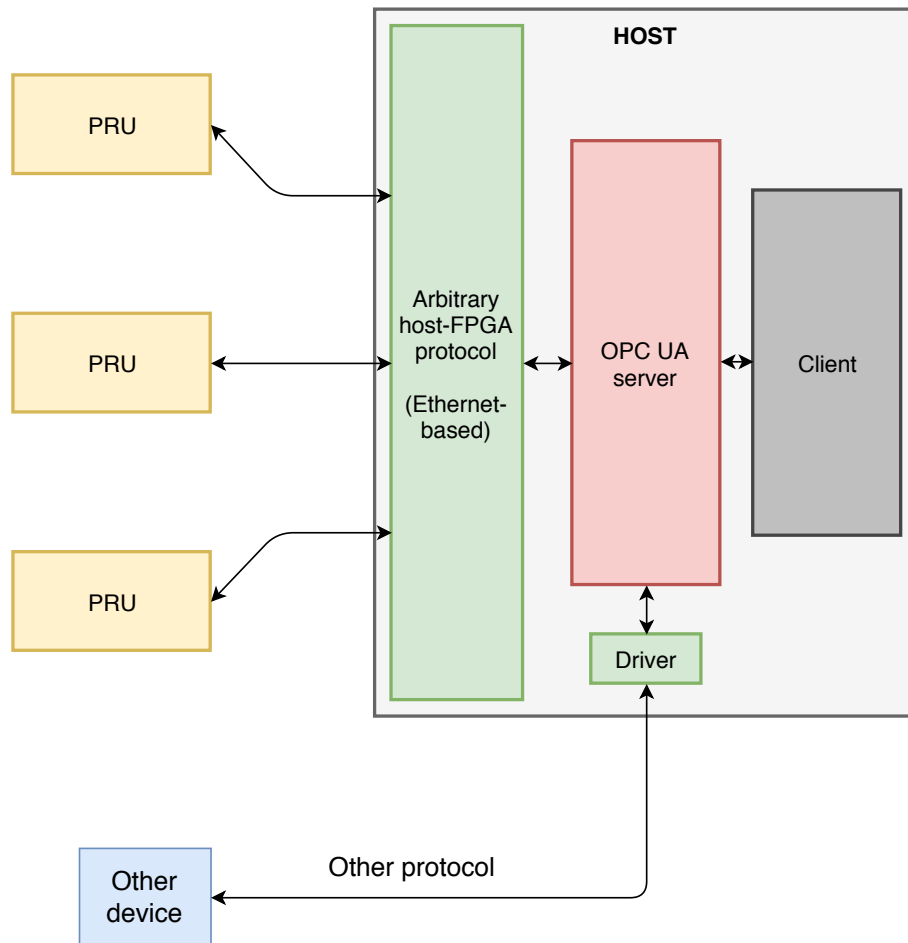


Figure 7.5: A possible control system for the pCT.

System Testing

Testing of the implemented firmware and software was performed on the test-setup at UiB, consisting of the VCU118 development board and the ALPIDE carrier, interfaced to the Virtex platform by an FMC with FireFly cabling.

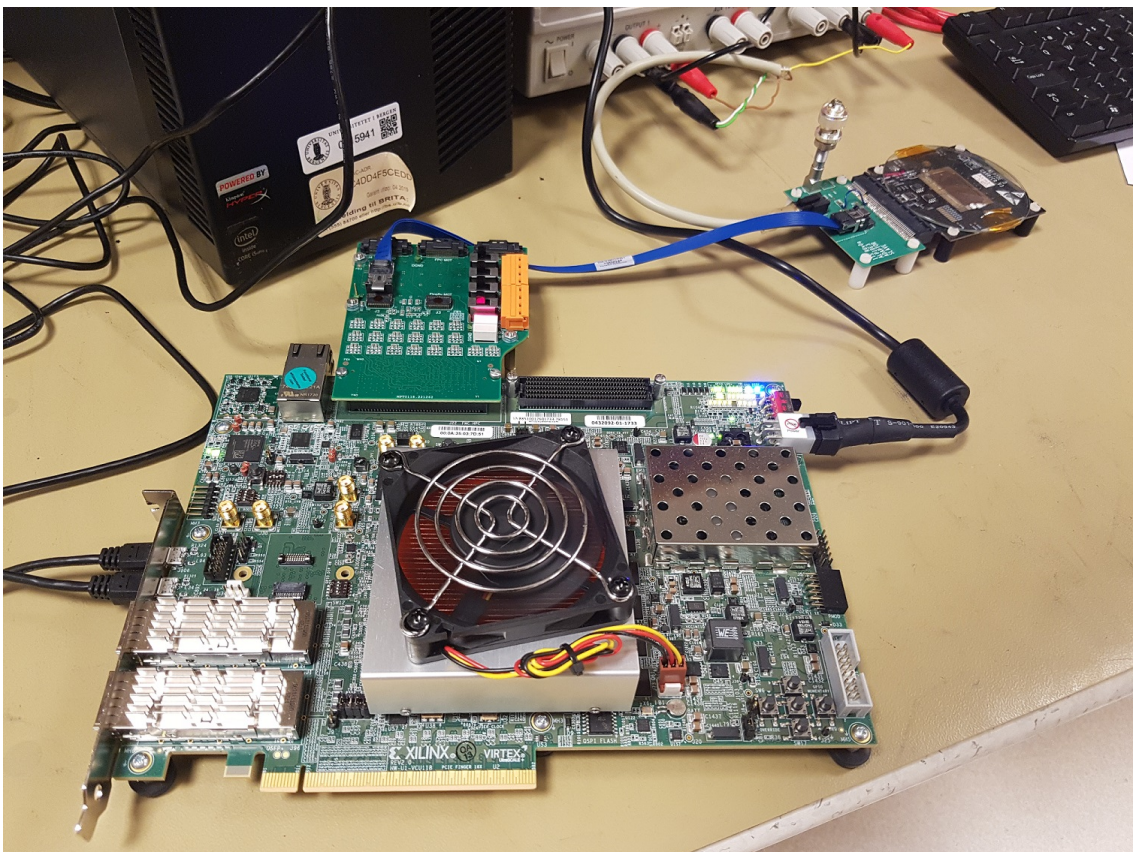


Figure 8.1: Testing setup showing VCU118 board, FMC, and ALPIDE carrier.

8.1 Host-Side Software

Software was developed for use on the host-side in order to enable communication with the embedded processor. This was written in Python¹ and provides an API that allows a host to communicate with any device that implements the protocol developed in chapter 6. In addition it defines some functionality specific to the VCU118 board that was used during testing. A base object defining a board or device implements base read-, write- and similar methods, from which objects that require special functionality can inherit.

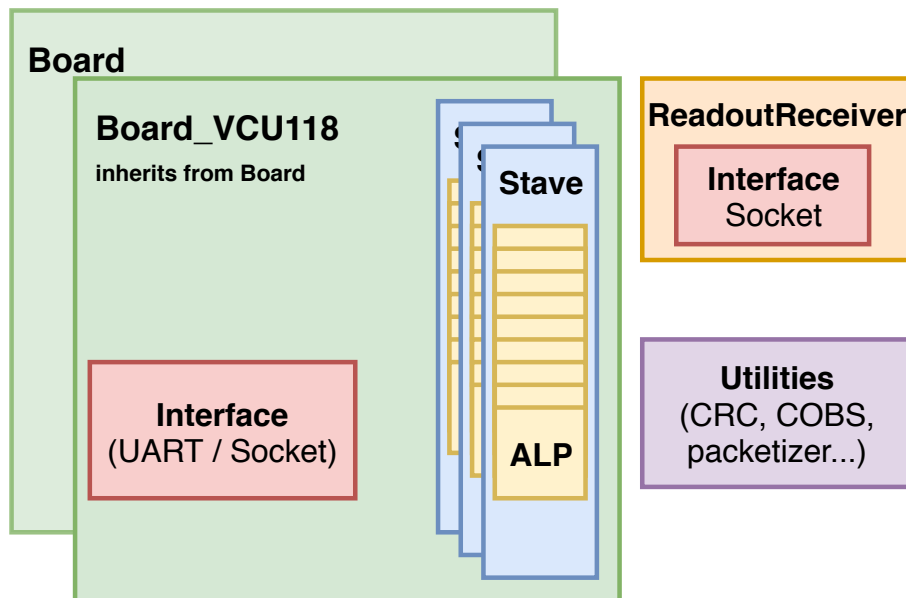


Figure 8.2: Core host-side software elements. Blocks are implemented as objects with the exception of the Utilities-block.

The use of inheritance was intended to simplify transitions to other boards or devices. The BoardVcu118 that was used for testing is an object with zero or more Stave objects which in turn consist of zero up to nine ALPIDE objects. The board also has an Interface-object, allowing for serial- or Ethernet communication.

Utilities for COBS encoding and -decoding, CRC calculation and packeting of data were also implemented, as well as a separate readout-data receiver that allows for testing of the offload solution from chapter 5. This is spawned as a separate thread by the board so that control can be performed while data is being read out.

¹A few reasons led to the choice of language: 1) it was requested by members of the pCT-team at UiB due to familiarity. 2) Python provides platform independence 3) Developing a working product is often faster in Python than other languages: often fewer lines of code are needed, when compared to *similar* languages such as Java.

8.1.1 API

All board objects are provided with an Interface-parameter which can be of either serial- or socket type. A BoardVcu118 also accepts an additional parameter "staves", that can be present if one or more staves are to be added to the board at instantiation. This can either be a single stave or a list of such objects. An ACM is sufficient and a *physical* stave must not necessarily be present.

Listing 8.1 shows a procedure that instantiates a BoardVcu118 object with a single stave with ID 0x00 and a single ALPIDE on the stave with chip- and PRU ID 0x00, as well as a socket interface on port 49153, the given IP, and of the SOCK_STREAM i.e TCP type. Communication is then possible, as demonstrated in listing 8.2.

Listing 8.1: Instantiation of a BoardVcu118 object.

```
my_board = BoardVcu118(Stave(0x00, [Alpide(0x00, 0x00,0x00)]),
                      MainSocket(49153, "192.168.1.195", "SOCK_STREAM))
```

Listing 8.2: Writing to- or reading from objects on a board.

```
#Writes to- and reads from a register on the board
my_board.write_reg(reg, val)
read_value = my_board.read_reg(reg)

#Writes a special command to the CPU
my_board.write_special(cmd)

#Performs a multicast write on a stave with ID s_ID
my_board.get_stave(s_ID).multicast_write(reg, addr)

#Writes to- and reads from an ALPIDE with the chip ID chip_ID,
#on a stave with ID s_ID
my_board.get_alpide(s_ID, chip_ID).write_reg(addr, val)
read_value = my_board.get_alpide(s_ID, chip_ID).read_reg(addr)
```

Multiple reads and writes can also be included in a single packet by using the `read_multiple_regs(list_of_regs)` and `write_multiple_regs(list_of_reg_val_tuples)` methods². The status of the VCU118 application can also be viewed via the serial interface, if enabled, as shown in figure 8.3. DHCP can be enabled/disabled in the BSP.

```
JART initialization OK
DHCP provided address:
IP : 192.168.1.195
Netmask : 255.255.255.0
Gateway : 192.168.1.1
Connected to 192.168.1.218 on port 57461.
```

Figure 8.3: Print-out of the serial output as the PRU is assigned an address via DHCP, and as it receives a connection.

²Ideally these methods should post to a separate thread that handles the interface and outgoing/incoming packets. This allows the interface thread accumulates data to send and receive, improving efficiency. As the software was eventually to be ported to C++, this change was not considered worthwhile.

8.2 Testing

Testing of the firmware, protocol, and software developed in chapters 5, 6, and 7 was done using the Python framework. As all testing was achieved by first communicating with the embedded processor, every aspect of the chapter 6 protocol had to first be verified as functioning correctly. In addition, the long-term stability of the embedded software had to be ensured.

The DMA-based data-readout solution discussed in chapter 5 and 7 was tested extensively. This was first done by providing the embedded software with data from the AXI stream counter. Tests were also done to find the maximum throughput that the system was capable of. UDP is unreliable and this was also demonstrated; the findings here prompting the alteration of the software so as to also support readout via TCP.

The developed communication framework allowed the data module to be further tested on the VCU118 board. Eventually this module was at a state where it was also possible to test the full readout chain.

8.2.1 Testing of Communication

The reliability of the protocol developed in chapter 6 had to be verified. This included asserting that oversized packets, those containing erroneous CRCs or COBS encoding, lacking- or having an incorrectly placed delimiter or similar were detected and recovered from. That payload was handled correctly also had to be ensured. Using the python framework, a test bench was set up where the results of each test was logged together with timestamps for any events that occurred. Testing was performed using both the serial and Ethernet interfaces, with the UART operated at baud rates between 9600 baud and 921600 baud. Not all tests are applicable for both interfaces, as data sent over TCP is not COBS encoded or appended a CRC.

The interval at which errors were injected, their placement in a packet and how a value was altered was chosen at random in order to increase coverage.

Listing 8.3: Typical text written at the end of a test. Example showing results of a test where misplaced delimiters were injected into messages sent via the serial interface.

```
10/05 09:33:57 INFO
#### COBS-DELIM TEST ####
Sent 166755 corrupted packets out of a total 1000000
Errors detected 166755
Errors missed: 0
Abnormal responses (not COBSERR or READACK): 0
```

Millions of read- and write operations were performed. In all cases, injected errors were detected with the receiver able to recover on the next packet due to the COBS encoding, requiring only to receive- or collect existing data from the FIFO until a delimiter is found, signifying the end of one packet and the beginning of another.

With TCP the approach differs in regards to recovery, as the reliability mechanisms

ensures that if *any* malformed data arrives to the application-layer then it is the sender who is choosing to send bad data, and this warrants closing the connection. This is only for framing, and invalid payload instead generates an error reply. The two communication channels are verified to work also when used simultaneously.

Testing of the ACM Driver and ALPIDE Communication

Chapter 5 mentioned that a driver was written for the ACM. This had to be verified as working correctly as correct ALPIDE communication was essential for the later tests. This was done through directed tests and also verified through the various test benches that were later ran and described in following sections.

8.2.2 Test Bench for the Updated ALPIDE Data Module

Previous work on the pCT project done by Ola Slettevoll Grøttvik³ involved the design of the ALPIDE data module, which processes and packets ALPIDE events according to a specified format. This had been tested on the CERN-developed readout board, but had since been modified. A test bench previously written for this purpose was rewritten and expanded. Built-in testing functionality of the pixel detectors are used, where equal pseudo-randomly generated test vectors are loaded into the ALPIDE and data module, which configures it to expect these on its inputs. The number of runs to be performed, length of cabling, current driver- and various other settings can be specified. All settings, vectors, counter values and words are logged to a file on completion. Errors are logged whenever a decoded word does not match any of the test vectors. If errors occur, the first occurrence is logged in a specific column.

Readout-data is transmitted to the VCU118 board via FireFly-cabling. On the previous board using the Kintex 7 FPGA it had been observed that increasing the length of this cable resulted in the phase-aligner⁴ in the ALPIDE data module being occasionally unable to locate valid delay-taps. Testing was repeated on the Virtex Ultrascale+ platform in part to see whether this was still the case.

Table 8.1: Test results with the updated data module.

Parameter	Test 1	Test 2	Test 3
Runs [N]	1800	1800	1800
Test Length [s]	54000	54000	54000
Cable length [m]	0.3	1.0	2
Correct words [N]	6.53×10^{12}	6.53×10^{12}	6.53×10^{12}
Incorrect words [N]	3.48×10^8	5.16×10^9	8.75×10^9
Runs with errors [N]	1	7	4
BER	5.32×10^{-5}	7.90×10^{-4}	1.34×10^{-3}

The large number of incorrect words detected when testing with the 0.3m is due to a chain of errors detected during a single run, which was not seen at any other

³ola.grottvik@uib.no

⁴A data module block that by iterating over available delay-tap settings searches for the amount of input delay to apply to an input data-stream in order to provide the best sampling point.

time. It is seen that increasing the cable length introduces errors. However, it was also noted by observing the log that errors occurred only in two forms: as part of a larger event where millions of errors were detected during the same run as described above, and in groups of 10-100. For instance, 3.48×10^8 errors are detected during a single run during the test with the 0.3 m cable. This is perhaps indicative of power supply fluctuations or some form of jitter, and not signal integrity issues introduced by use of the longer cable. A probable explanation for this is jitter occurring when an input-word is sampled; this may cause an incorrect sampling point to be selected. If this this point is not validated any further, it would cause all words that follow during that run to be sampled incorrectly, which is what is observed.

As this is likely a bug in the logic, and not indicative of the data module or transceivers not functioning correctly, the BER listed above is not representative of the errors introduced due to signal integrity issues.

On the 2 m cable, communication with the chip through the ALPIDE control module also becomes problematic. One of the sub-components of the module is an input serializer, which features a register to adjust the phase at which the received data is sampled. Setting this to a value of one is found to resolve the communication issues.

8.2.3 Test of Data-Readout Solution

The DMA based data-readout solution was tested. Initially, the ALPIDE data module had not been fully developed and so an actual ALPIDE could not be used as a data source. An AXI Stream counter was instead written and used as a replacement. Verifying the functionality of the solution and measurement of the average throughput was done by sending a set amount of counter data from the PRU to the host. Upon completion, it was verified that in any packet a previous value was always smaller than the next. Compilation was done with the -O3 optimization-flag set.

Listing 8.4: Python receive-function used to test throughput.

```
def measure_throughput(self, packet_size_to_get, bytes_to_get):
    """ Try to receive packet_size_to_get-sized datagrams from the
    socket, and continue to do so until a bytes_to_get number of bytes
    is received """

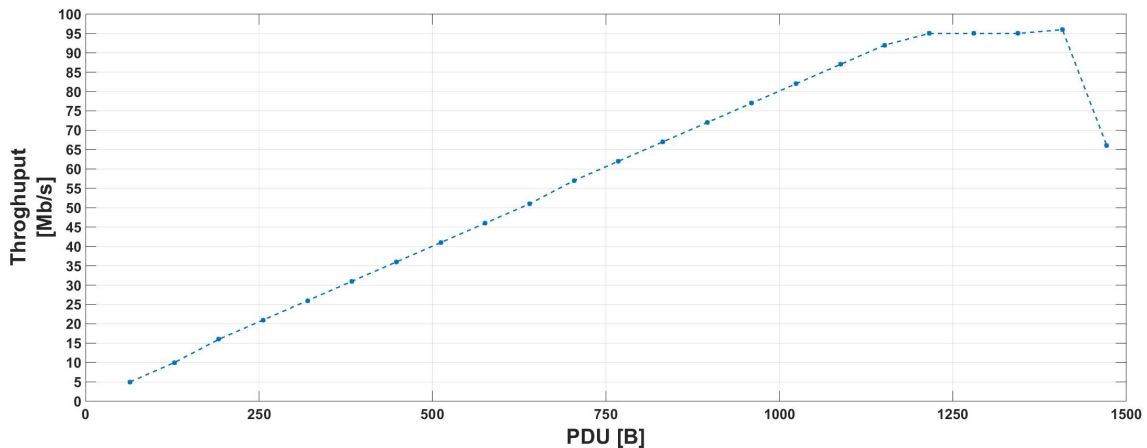
    received = 0
    bytes_recvd = 0

    t_start = time.time()
    while bytes_recvd < bytes_to_get:
        bytes_recvd = bytes_recvd +
            len(self.sock.recvfrom(packet_size_to_get)[0])
    t_end = time.time()

    print("Received {} bytes in {:.2f}s.\nApprox ~{:.0f}b/s
    or ~{:.0f}mb/s".format((bytes_recvd), t_end-t_start,
        ((bytes_recvd)/(t_end-t_start))*8,
        ((bytes_recvd)/(t_end-t_start))*8/(10**6)))
```

First tests were performed on a 100 Mb link. The assumption that sending packets close in size to the MTU would increase performance was verified, and throughput was found to vary substantially with PDU size. Approaching the standard Ethernet MTU of 1500 bytes resulted in saturation of the link, while packets in excess cause fragmentation and degraded performance.

Figure 8.4: UDP throughput as a function of PDU size on a 100 Mb link.



It is clear that software is the limiting factor until saturation occurs, with the overhead stemming from assembling the UDP datagram headers and setting up the DMA transfer. Transmission and data-moving is performed by firmware. Larger datagrams increase the data/header ratio which increases performance. It should be noted that the numbers seen in the figures are of delivered *payload*. In addition to the user data contained within the datagrams, a sent frame contains 20 bytes of IPv4 header data, 8 bytes of UDP header data, and 14 bytes of Ethernet framing data. The full throughput is therefore slightly higher if this is included.

Jumbo Frames

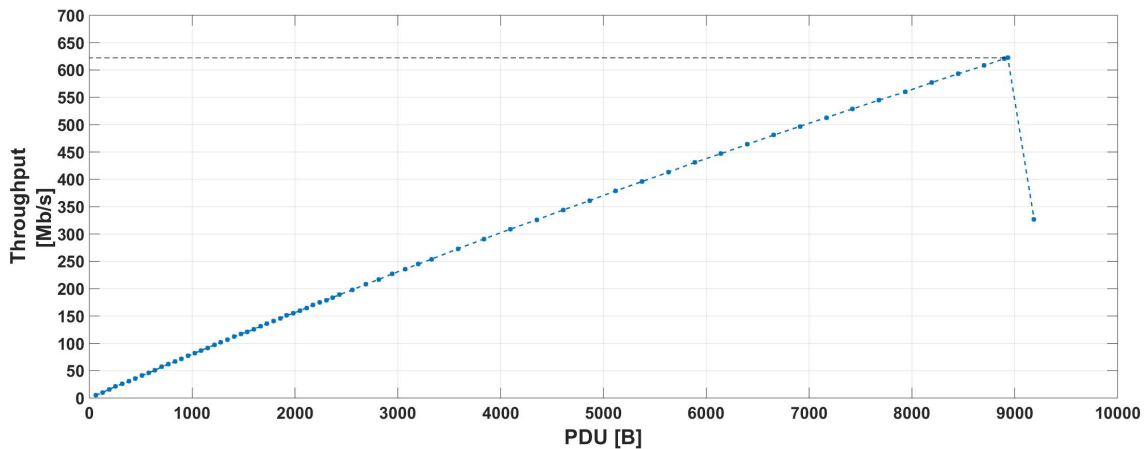
Most commercial ISPs support the standard Ethernet MTU of 1500 bytes, and therefore many networks that connect to the Internet are configured to match for compatibility reasons. It is however possible to go beyond using *Jumbo frames*, in which up to 9000 bytes of data can be sent. From the results on the 100 Mb/s link it was expected that this would increase performance. First it must be enabled host side:

```
$ sudo ip link set eno0 mtu 9000
```

Given a Linux system where *eno0* is the name of the network interface controller. Jumbo frames must also be enabled for LwIP; either in the *opt.h* file or via the BSP. A gigabit switch was used as a link, jumbo frames were enabled and the throughput logged. This improved throughput substantially, and although saturation this time was not reached, a maximum of 625 Mb/s was achieved before the larger MTU is exceeded and fragmentation again occurs.

During testing, the MicroBlaze was provided with a 150 MHz clock, and so as to not be limited by the counter was instructed to send random chunks of data at the tested PDU sizes.

Figure 8.5: UDP throughput as a function of PDU size on a 1000 Mb link with Jumbo frames enabled.



8.2.4 UDP Packet Loss

Measurements of UDP packet loss was performed to see whether there was dependence on the PDU size; the larger jumbo frames can for instance not be used if it results in unacceptable losses. The processor was set up to offload UDP datagrams that contained a sequence number that was incremented on each datagram sent. Since datagrams arrive either completely or not at all, it was sufficient to verify that on reception of a datagram, its length corresponded to the specified size, and that the value contained equalled the previous value received incremented by one. A loss is interpreted as the difference between the expected sequence number and the one received. Measurements were performed with a 1 m Cat6 cable; results might differ as the cable lengths increase and this should also be verified. Other aspects of the setup could alter the results also, for instance NIC drivers and the switch used.

A logger was set up containing first a header displaying the datagram size. A loss was logged with a timestamp, time elapsed since start of test, and the expected- and received sequence number from which the number of datagrams lost was derived.

Listing 8.5: An excerpt from a packet-loss log showing two events.

```

10/05 16:59:46 INFO  Running packet loss test with packet
                    size 8972B. Running time set to 21600s.
...
10/05 19:50:01 INFO  2018-05-10 19:50:01.33 (+10214.93 s):  90 / 97
10/05 20:40:01 INFO  2018-05-10 20:40:01.60 (+13215.19 s):  5 / 6
...
10/05 22:50:26 INFO  Received 1633149.603008MB. Lost 63 packets ,
                    which is 9.29e-08 of total

```

It was assumed that since the server-host link was direct, passing only through the switch, out-of-order delivery would not occur. This as it turned out was a potential source of error, although the likelihood of this occurring on a closed network is very low and was never recorded. Datagrams might also arrive at the receiving NIC, but fail checksum verification. These packets will never arrive to the application layer and hence also be reported as lost. To avoid overflow on the receiving side, the receive-buffer size was set to 10 MB.

It is not trivial to diagnose where a datagram is lost; no errors were reported in the PRU software, and it was verified that buffer-overflow at the receiver had not occurred. This leaves either the switch or a silent error on either the receiving- or sending end. It was noted that in the majority of cases, multiple datagrams were dropped any time loss occurred. This could indicate a buffer overflow at the switch level, and it should be worthwhile to repeat the test with a higher-quality switch. Alternatively the Ethernet MAC driver could contain defects. It is unlikely that the firmware is not able to handle the amount of data produced by LwIP. Events were few enough that it could easily be verified that datagrams were not counted as "lost" due to out-of-order delivery.

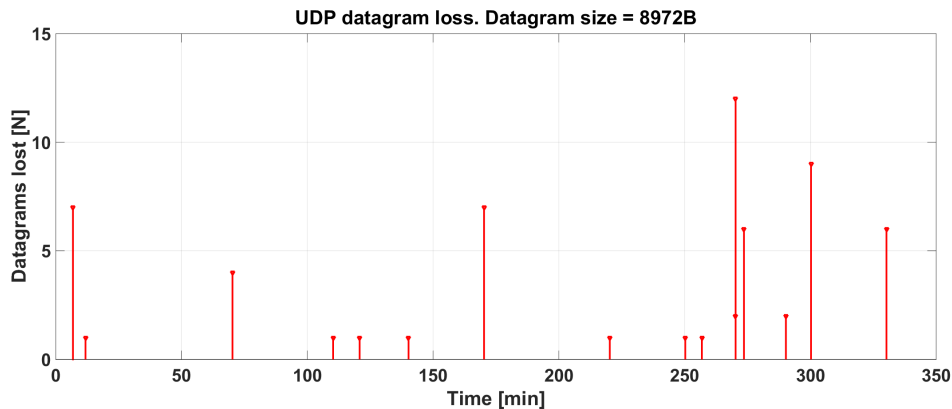


Figure 8.6: Showing loss over UDP with datagrams sized at 8972 B.

The plot shown in figure 8.6 shows the losses that were recorded when 1.63 TB of data was transmitted from the VCU118 board over the course of approximately 6 hours. Loss was not found to vary with time, and on each event, 3.7 datagrams were on average lost. In total, the loss represented 9.3×10^{-8} of the transmitted data. Mean throughput was 621 Mb/s with jumbo frames enabled and the datagrams sized at 8972 B⁵.

Although it is likely that datagrams are lost in the switch, and that results could improve with a higher-quality equipment in general, these results demonstrate the unreliability of UDP. Furthermore it is impossible for a receiver to know whether or not a frame has been lost, thus even on a high-quality system it cannot be determined whether a connection is loss free unless sequence numbers are used.

⁵28 bytes are needed for headers.

8.2.5 TCP

The embedded software was extended so as to also support detector-readout via TCP, which can be selected via a toggle-macro in the software options. It is currently planned that a Zynq platform will at some point during development be used for the readout process; it was not known whether the MicroBlaze could achieve performance using TCP comparable to that achieved with UDP, but in any event the code should be a drop-in for use with these devices.

Tuning for TCP

Several aspects of TCP make the protocol resource-intensive on the software side. Among these the TCP *window*, flow- and congestion control, and sequencing of segments to ensure data is reassembled in-order on the receiving side. The majority of these can be configured, and some were found to have significant impacts on throughput; in particular the window size. TCP ensures reliability partly by enforcing acknowledgment of data sent. The receive-window keeps track of the sent but unacknowledged data, and its current size is continuously tracked by a sender. If the window size reaches 0, a sender will stop transmission and wait for the receiver to acknowledge reception of data. Increasing the window size thus allows for fewer ACKs to be sent as more data can be sent until this is required. On the FPGA side, this was increased to its maximum (LwIP) size of 65 535 bytes.

Furthermore, the TCP send-buffer was set identical to the increased window-size, and the maximum TCP segment size was set to 9000 bytes to match the higher jumbo frame limitation. Even with these changes, it is not possible to approach the bandwidth possible with UDP, and a throughput of approximately 190 Mb/s is achieved with the soft core processor.

8.2.6 Testing of the Full Readout Chain

In May 2018, the ALPIDE data module had reached a state where testing of the full readout chain was possible. The stream counter used with the DMA engine was removed and replaced with the ADM, which included an intermediate FIFO buffer-stage. The intention here was not to extensively test the chip, but to verify that the readout chain was functional. Figure 8.7 shows a transfer being performed by the DMA module. In this case the ALPIDE had been configured with its *ITHR* current source set to a high value. This increases the pixel threshold, which decreases the number of hits⁶ that are recorded, resulting in the staggered tvalid signal.

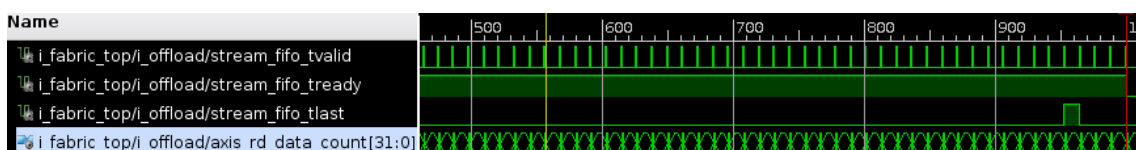


Figure 8.7: A DMA transfer of ALPIDE data, as recorded by the internal logic analyzer.

⁶In this case the hits are only noise.

It is seen that as `tlast` rises, `tready` remains high for four counts of `tvalid`; this is due to the filling of the four pipeline registers, as mentioned in section 5.3.1.

The data that is read out is captured by the `ReadoutReceiver` module described in section 8.1, and stored in a text file. A PRU word is as mentioned always 128 bits/16 bytes in size, and is either of the data-, tag-header-, or tag-trailer type. A data-word is always framed by a header and a trailer, and any number of data-words may be found between the two. A snippet of a print-out is seen in figure 8.8.

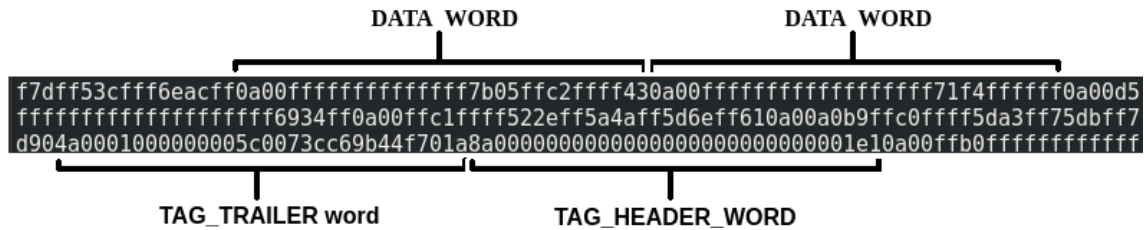


Figure 8.8: A print-out of an ALPIDE data-stream, formatted as PRU words by the ADM.

In the print-out, three word-types can be identified by their initial content⁷. The two data-words are part of a larger event that is not shown.

Table 8.2: PRU Headers.

Header name	WORD_TYPE[127:126]	RU[125:120]	STAVE[119:116]	CHIPID[115:112]
DATA_WORD	0x00	0x0A	0x00	0x00
TAG_HEADER_WORD	0x01	0x0A	0x00	0x00
TAG_TRAILER_WORD	0x02	0x0A	0x00	0x00

It is seen that the words stem from a readout unit with ID `0x0A`⁸, and an ALPIDE with chip ID `0x00` on stave `0x00`. The large number of hexadecimal 255 seen in the data words are padding for unused ALPIDE data fields.

As data is read out, the number of triggers sent in addition to some status information is periodically read out over the control interface, as shown in figure 8.9.

```
2018-05-25 20:50:01,372 INFO
Triggers sent: 874500
Triggers NOT sent: 0
Decode errors: 0
Num events: 874500
Num protocol errors: 0
Num event errors: 0
Num empty frames: 0
Time left: 14:38:09.849794
```

Figure 8.9: Showing a snippet of control-data read out while transfers are ongoing.

⁷The PRU word-specification can be found in the pCT WP3 repository.

⁸This was set as such in order to locate it easily in the print-out.

DMA issues

Some unexpected behavior was occasionally seen on transfers, although most could be explained or solved. One initial issue that was encountered was incorrect generation of `tlast`, where this was fixed at a high level on the receive-side of the DMA engine. This caused transfers to finish immediately, which limited throughput. This was fixed by generating `tlast` at the correct intervals. The DMA pipeline-registers initially also caused some confusion, as they are not mentioned in Xilinx documentation. These cause for instance `tready` to stay high indefinitely if connected to an empty FIFO⁹.

Another situation that was observed was intermittent assertion of `tready`. Although this was not verified, this could be due to arbitration on the AXI bus; some of the registers of the data module are periodically read out to check for transmission errors, and this could for instance be one source of conflict.

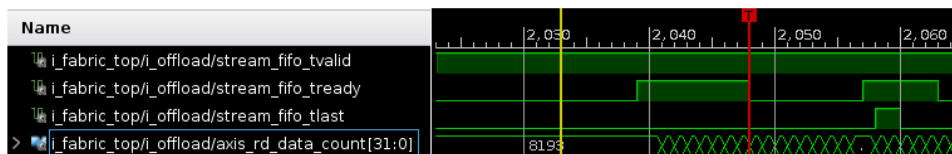


Figure 8.10: Showing a gap of eight cycles between assertion of `tready` on the DMA receive-side.

This is even clearer in figure 8.11, showing several smaller gaps in addition to a larger eight-cycle gap.

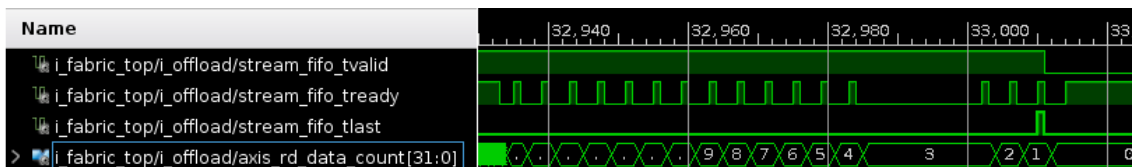


Figure 8.11: Intermittent assertion of `tready` on the DMA engine.

8.2.7 Testing of Self-Contained PRU Monitoring

Chapter 7 described the integration of an automatic monitoring task that can be enabled in the PRU software to monitor given addresses and chips without the need for user input. Testing the solution involved adding items to monitor and verifying that if their thresholds were exceeded, action was taken. Behavior in boundary cases also had to be controlled, such as that occurring when there is nothing to monitor, an attempt is made to add items beyond the pre-defined maximum capacity, and similar situations.

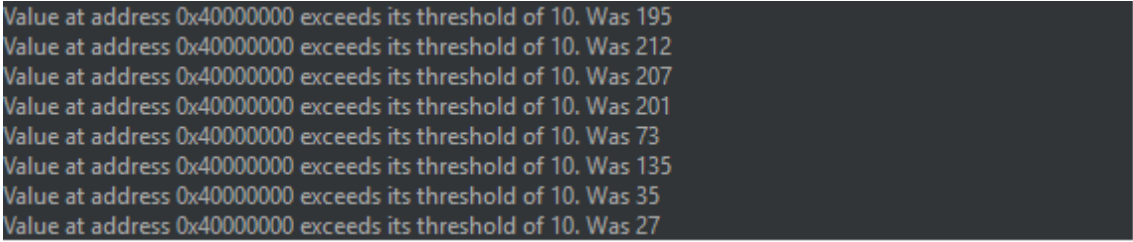
Although there are no temperature sensors or voltage regulators directly accessible from the FPGA fabric on the current VCU118 design, arbitrary items can be added

⁹Due to the registers in this case never filling

to the list of monitored items. Functionality could thus be verified by adding monitored registers and forcing the associated callback by writing to them a value that exceeded the set threshold.

A writable dummy-register was added, and a threshold for it set to 0x0A. The monitoring task was set to wake every 50ms to perform its readings. A callback was defined that simply printed the address and value found over the serial interface, and wrote the value of 0x00 back to the register. The value could then be asserted by a read operation.

Reading out the debug interface showed that the task was performing as it should:



```
Value at address 0x40000000 exceeds its threshold of 10. Was 195
Value at address 0x40000000 exceeds its threshold of 10. Was 212
Value at address 0x40000000 exceeds its threshold of 10. Was 207
Value at address 0x40000000 exceeds its threshold of 10. Was 201
Value at address 0x40000000 exceeds its threshold of 10. Was 73
Value at address 0x40000000 exceeds its threshold of 10. Was 135
Value at address 0x40000000 exceeds its threshold of 10. Was 35
Value at address 0x40000000 exceeds its threshold of 10. Was 27
```

Figure 8.12: Debug-interface print-out of the embedded software reporting exceeded thresholds.

Conclusion and Future Work

9.1 Performance Evaluation

The results of chapter 8 show that the remote-control system is working as intended. The extensions done to the protocol developed in chapter 6 in order to ensure operation when used over unreliable interfaces are also demonstrated to be functioning, with all injected errors detected by the embedded processor software. The system is also shown to work reliably during extended uptime, with multiple tests being run over the course of multiple days.

The ALPIDE data module is tested on the new Ultrascale+ platform. Using it and the ALPIDE's testing features, it is shown to be mostly functioning, although some errors are introduced both with the 1 m and 2 m cable. In addition, the ADM was seen to on some occasions select an incorrect sampling point which causes all further words to be incorrectly sampled until a reset is performed. The cause of this behavior should be investigated.

The DMA-based readout solution developed in chapters 5 and 7 was tested with the MicroBlaze, both using UDP and TCP. Readout speeds of approximately 622 Mb/s were attained using UDP with jumbo frames, with the processor provided with a 150 MHz clock; this is less than half of the stated maximum, and thus higher data rates should be possible. The unreliability of UDP was demonstrated, and low but non-zero loss in the range of 10^{-8} was recorded. UDP will therefore be acceptable for testing the chips, but when the prototype reaches a stage where data is used to reconstruct proton tracks, TCP must be used. Readout using TCP is tested also with the MicroBlaze, where a throughput of approximately 190 Mb/s is measured; the reduction in bandwidth most likely due to the increased software overhead. The read-out data is received and stored on a Linux machine, and is confirmed to follow the format of the specified PRU words; hence this part of the readout chain is also verified.

9.2 Design Evaluation

Chapter 4 described the system control unit to have been originally intended to be a separate board that would interface to the PRUs through a separate control link. It was argued that the readout boards may interface directly to a host, and that monitoring of all units on the PRU might be performed by the readout units themselves. This allows the SCU to be replaced with a unit responsible solely for clock (and possibly trigger-) distribution, which should allow for this to be a commercially available part, needing no custom design. It was also stated that due to the wide strobe-windows and long peak-time of the ALPIDE analog output signal, some skew between boards can likely be tolerated. It was also concluded that due to the effective pixel-addressing scheme on the ALPIDE, no local storage of configurations on the PRUs should be necessary, and that the required bandwidth for the control system was low in general.

Chapter 4 further described the requirements of the PRU soft-core processor, and the software that it should run. It was argued that what was primarily needed at the current stage of the project was a way to directly access the address space of the device, and as such a protocol that could be used both with the Ethernet- as well as the serial interface that would allow this was necessary.

The implementation of the firmware described in chapter 5 should, with the exception of the DMA-subsystem, need little to no alteration during the lifespan of the pCT project. The DMA-based solution for data-readout is shown to be working, and the performance results from chapter 8 implies that it should function well on the Zynq platform; few alterations should be required for this transition.

In chapter 6, a simple protocol for transferring arbitrary data over streaming interfaces was developed, which was used to provide access to the address space of the FPGA. Some time was spent on the details of this protocol, especially in regards to the implementation of the reliability-features for its use over unreliable interfaces. In the end, it is found to be working reliably, and when implemented produces no errors even after several days of uptime. At a later stage, MQTT or OPC UA might replace the use of this protocol over the Ethernet link, or it could be used in conjunction with an OPC UA server implemented on the host side as described in section 7.7. Over the serial interface, this type of access will remain useful.

Embedded software for the MicroBlaze CPU was developed, and was described in chapter 7. The software provides serial- and Ethernet communication links, controls the DMA-readout process, and enables the CPU to automatically monitor on-board devices. Two alternatives for higher-level software that could at a later stage be used with the system was detailed: MQTT and OPC UA. Both was stated as having their advantages, and although the publish/subscribe architecture of MQTT will likely offer higher performance than the client/server architecture of OPC UA, this should not be a concern as there should be no need for real-time monitoring from the host-side as explained in section 4.1.4.

9.3 Future Work

Much work remains before the complete pCT can be realized, but this section will focus primarily on those relevant to its control system.

9.3.1 Porting of the Python Software

The Python framework described in chapter 8 has already proven useful and can be reused for future projects. However, extensive developments were made to existing (host-side) ALPIDE testing software, developed by CERN and written in C++, over the course of this year. This has made modifying this software so as to be compatible with the system developed in this thesis, desirable. This should not be too difficult, as it is primarily the lower-level write- and read methods that needs to be modified. The C++ software is written in a similar fashion as the Python framework in the sense that specific functionality is restricted to individual *board-types*. Porting therefore means defining a *VCU118*-board and implementing its specific methods.

9.3.2 Porting of the Embedded Software

The embedded software should require little porting in order to be used on the Zynq platform, except from part of the functionality that relates to the specific UART used with the MicroBlaze.

9.3.3 Extension of the Readout-System

The DMA-based readout solution is tested, and found to be working. If this is to be used also for the testing of staves, it requires the extension described in section 5.3.1. Some work will be required to port this to the Zynq device, but this should be minimal.

9.3.4 Implementation of Higher-Level Control Software

MQTT and OPC UA was described as possible candidates for a higher-level pCT control- and monitoring system, and both where listed as having their advantages. Two concrete options were also mentioned, and these should be investigated further.

9.4 Conclusion

This thesis has detailed the requirements and full implementation of an embedded system that can be used to control, automatically configure- as well as monitor modules on- or external to an FPGA. To this end, a small-footprint soft-core processor system has been implemented that allows for remote access to such a device's memory space through serial- and Ethernet interfaces via a simple platform independent API. The use of a compact OS ensures modularity and therefore the scalability of the system, and depending on platform and configuration, no external memory is required as the small kernel-size and lightweight networking stack allows the software to fit entirely within approximately 1 MB of RAM. A DMA-based solution for data readout was specified, implemented and tested. The extension of both this system as well as the control-system in general as the pCT project progresses has been detailed.

Coding Style

The work discussed in the preceding chapters often consist of code written in various languages; primarily Python and C. Striving for consistency, some guidelines were followed:

Python

Any code written in Python adheres to PEP8 - Style guide to Python [41]. Some central elements include:

- Spaces are the preferred indentation method
- Python 3 disallows mixing the use of tabs and spaces for indentation.
- Line-width is 79 characters.
- PEP8 is flexible in terms of naming conventions, but:
 - Constants are ALL_CAPS_WITH_UNDERSCORES.
 - Variables are all_lowercase_with_underscores.
 - Class-methods and -functions are all_lowercase_with_underscores.
 - Classes are CamelCase with the first character also capitalized.

C

All code written in C follows the *Linux kernel coding style*¹ with the notable exceptions that indentation consists of four spaces instead of eight, and that line width is 100 characters. To keep memory consumption transparent, no dynamic memory allocation is used². To this end all queues and tasks are allocated at compile-time, and larger local variables such as receive-buffers are declared static.

¹Available at <https://www.kernel.org/doc/html/v4.10/process/coding-style.html>

²LwIP implements a `malloc()` that allocates chunks of memory from a statically allocated pool.

Resource Usage

Although it is unlikely that the complete pCT design will be size constrained if implemented on the current Virtex Ultrascale+ chip, an estimate of the resource-usage that the central cores used in the control subsystem mandate is given here should it become a concern. This is based on Xilinx' numbers, found in the product guides cited in the bibliography.

The subsystem, including processor, DMA modules, Ethernet cores etc in addition to an ALPIDE control- and data module consumed less than 2% of the chips resources.

Core/component	LUTs	FF
MicroBlaze (max. performance)	4124	226
AXI DMA	2020 ¹	3525 ²
1G/2.5G Ethernet Subsystem	3200 ³	5300 ⁴
AXI UART 16550	308	345

Table B.1: Resource usage of components included in the MicroBlaze subsystem

B.1 RAM

FreeRTOS and LwIP keeps memory usage down; built with the -O3 flag⁵, the embedded software consumes approximately 1 MB.

⁵See <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

SPAD

The *Serial Protocol for Arbitrary Data* (SPAD) is used to transfer information in any form between two hosts, by providing a *frame* into which this information can be placed. It is suitable for data-transfer over streaming interfaces, and provides special reliability-measures when the transfer is performed over one that is unreliable. The actual data is prepended by a two-byte long length indicator, and a byte-long command-type field used to indicate the *type* of data that follows. This type in addition to the structure of the following data is up to the user to implement. For reliable interfaces, a frame begins with the length- and command-type field, and ends with a sequence number that is zero on the first packet sent, and incremented by the sender on each packet until rolling over after 255; a receiver verifies the sequence number. Buffer sizes typically place a limit on this size lower than the maximum possible length, and a receiver silently drops any oversized packets. On unreliable interfaces, packets are in addition first appended a CRC and then COBS-encoded to ensure re-synchronization is possible if erroneous data is received.

On reception of a packet and execution of its contents, a reply is always returned containing either one or several ACK-type messages, or error-indicators - each one byte long. A message can contain several sub-messages and each will cause a ACK or NACK to be added to the reply. Packets that contain a number of correctly formatted sub-messages but some that are malformed will cause any good sub-messages to generate ACKs, and a final NACK for the first malformed sub-message after which the remainder is dropped. A collection of ACKs or NACKs generated by one request is sent as one reply.

C.1 Requests and Command Types

Three CMDTYPs were implemented and used; each of which correspond to a type of payload:

CMDTYP	Description
0x01	ALPIDE-destined request
0x02	Peripheral-destined request
0x03	Special command request
0x04	Reply

Table C.1: Application level protocol - CMDTYPes

WRCODE(1B)	REGVAL(4B)	REGADDR(4B)
RDCODE(1B)	REGADDR(4B)	
WR-/RDCODE	Description	
0xAA	read	
0xFF	write	

Table C.2: Payload - format for register writes- and reads respectively, and values of the WR/RD-field.

The number of these and hence length of the payload matches the value found in the beginning length field. For instance, a packet containing two write-commands would have a payload 18 bytes long, which would also be the value of the length field.

ALPIDE-payload:

OPCODE(1B)	CHIPID(1B)	STAVE(5b)	NSNGL(3b)	REGADDR(2B)
REGVAL(2B)				
OPCODE(1B)	CHIPID(1B)	STAVE(5b)	NSNGL(3b)	REGADDR(2B)
OPCODE(1B)				

Table C.3: Payload - ALPIDE register writes, -reads, and opcodes, respectively.

Again, any number of these sub-messages can be contained in the payload section so long as it is shorter in total than the buffer size set by the receiver.

Special command types.

Special cmd val	Description
0x01	Spawns the data-readout thread
0x02	Deletes the data-readout thread

Table C.4: Payload - The *special* command type

The special command is intended to instruct the processor to perform longer procedures.

C.2 Replies

Replies are formatted like requests, have the CMDTYP *reply* (0x04), and have in their payload either data if this was requested, one or several ACK-type bytes to indicate successful execution of a sub-message, one or more ACKs with a final descriptive error byte if an error was encountered, or a single error.

Data in payload	Description
0x01	Sub-message successfully executed
0x02	Peripheral read code
0x03	ALPIDE read code
0x04-0x1F	reserved
0x20	CRC check failed
0x21	A COBS-delimiter was not found
0x22	Erroneous COBS-encoding detected
0x23	Timeout following partial packet-reception
0x24	Invalid WR/RD code in peripheral packet
0x25	Unrecognized CMDTYP-field
0x26	Invalid ALPIDE opcode
0x27	Invalid special command

Table C.5: Reply-data ACKs/NACKs

C.3 Examples

The following packet instructs a board to write the value 0x45 to address 0x40000000, and shows the reply that is returned.

	len(2B)	CMDTYP(1B)	Payload(9B/1B)	Seqnum(1B)
Request	0x0009	0x02	0xFF4000000000000045	0
Response	0x0001	0x04	0x01	0

Table C.6: Example write-request and response

The following is a read request and the generated response for the value stored at address 0x40000000, following the write-operation.

	len(2B)	CMDTYP(1B)	Payload(5B/5B)	Seqnum(1B)
Request	0x0005	0x02	0xAA40000000	1
Response	0x0005	0x04	0x0200000045	1

Table C.7: Example read-request and response.

Writing the value 0x1FC to the command register (0x01) of the ALPIDE on stave 0x01 and with chip ID 0x01:

	len(2B)	CMDTYP(1B)	Payload(8B/1B)	Seqnum(1B)
Request	0x0007	0x01	0x9C0109000101FC	2
Response	0x0001	0x04	0x01	2

(WR)OPCODE	CHIPID	STAVE/NSNGL	ADDR	VAL
0x9C	0x01	0x09 (0b00001001)	0x0001	0x01FC

Table C.8: Example ALPIDE write-request and response, and contents of payload.

To read the value that was written:

	len(2B)	CMDTYP(1B)	Payload(5B/3B)	Seqnum(1B)
Request	0x0005	0x01	0x4E01090001	3
Response	0x0003	0x04	0x0301FC	3

Table C.9: Example ALPIDE read-request and response.

Python Framework

This section provides an overview of the Python framework and its API used in chapter 8. Some methods/functions and classes are left out for brevity.

board.py

```
class Board( object):
    def __init__(self, interface):
        """Initialize a board with a given (MainSerial or MainSocket) interface. """
    def write_reg(self, addr, value, get_ack=True):
        """ Write a 32 bit value to a 32 bit address. """
    def write_special(self, data):
        """ Writes a special command to the CPU. """
    def write_multiple_regs(self, reg_val_pairs):
        """ Packets multiple address-value pairs in a packet writes the result. """
    def read_reg(self, addr):
        """ Returns the value stored at the 32 bit address. """
    def read_multiple_regs(self, addr):
        """ Packets multiple addresses in a packet and returns the values. """
    def write_and_assert_peripheral(self, reg, val, bit_mask=0xFFFF, msg=None):
        """Write a 32 bit value to a 32 bit address and assert the value.
        read-only-bits can be masked with bit_mask.
        """
```

board_vcu118.py.py

```
class BoardVcu118(Board):
    def __init__(self, staves=None, interface=None):
        """Initialize a board with a given (MainSerial or MainSocket) interface
        and optional stave.
        """
    def get_stave(self, stave_id):
        """ Returns the stave with the corresponding stave_id if it exists. """
    def get_alpide(self, stave_id, chip_id):
        """ Returns the Alpide with given IDs if it exists. """
    def add_alpide(self, alpide, stave_to_add_to):
        """ Adds an Alpide to a stave on the board. """
    def start_offload(self, board_ip=None, port=None, send_cmd_only=True,
        use_tcp=False):
        """ Starts offloading of Alpide data from the PRU. """
```

```

def end_offload(self):
    """ Ends offloading of Alpid data from the PRU. """
def setup_test_vector(self, alpid, tv0, tv1, tv2):
    """ Loads matching set of test-vectors into ALPIDE and ADM. """
def reset_data_module(self, data_module_number):
    """ Resets the given ALPIDE data-module. """
def reset_bank(self, bank_num=None, global_rst=False):
    """ Resets the bank with ID bank_num. """

... various other methods for specific VCU118 functionality ...

```

alpid.py

```

class Alpid(object):
def __init__(self, chip_id, stave_id, pru_id, conf=None,
             half_b_val=None, discri_sign_val=None):
def read_reg(self, reg):
    """ Read an Alpid register. """
def write_reg(self, reg, val):
    """ Write to an Alpid register. """
def send_opcode(self, opcode):
    """ Send an opcode to the ALPIDE via its command register. """
def init_alpid(self, read_back=False):
    """ init. chip according to operations manual (defaults) """
def reset_alpid(self, read_back=True):
    """ Manually reset all chip registers. """
def write_and_assert_alpid(self, reg, val, bit_mask=0xFFFF):
    """ Writes value val to register reg of an ALPIDE and
        asserts the written value.
        Read-only-bits can be masked with bit_mask.
    """

... various other methods for pixel masking, periphery control register
    configurations, ADC, readout, etc ...

```

readoutdata_recvr.py

```

class ReadoutDataReceiver(object):
def __init__(self, port, addr, use_tcp=False, is_thread=False,
             queue_to_self=None, queue_to_board=None):
def receive_data_udp(self, receive_num_packets=None):
    """ Receives data from a board via UDP """
def receive_data_tcp(self, receive_num_packets=None):
    """ Receives data from a board via TCP. Also handles
        the listen + connect procedure.
    """
def bytes_to_int(self, to_convert):
    """Helper function to turn bytes received into integers. """
def measure_throughput(self, packet_size, bytes_to_get):
    """Measure the TX throughput of the sender. Log results to file. """
def measure_packet_loss(self, packet_size, secs_to_rcv=10**6):
    """ Receive sequenced datagrams from a sender. Log mismatches
        between expected and received datagrams. """

```

utility.py

```
def get_alpide_packet(interface , opcode=None, chipid=None, stave_ofst=None,
                    reg_val_pairs=None, skip_packaging=False):
    """ Returns a packet intended for an ALPIDE with given features """
def get_peripheral_packet(interface , reg_val_pairs=None):
    """ Returns a packet intended for a PRU peripheral. """
    """ Receives data from a board via UDP """
def get_special_packet(interface , data):
    """ Returns a packet containing one of the special commands """
```

Various

E.1 ALPIDE Mask-Application

Application of the ALPIDE pixel masks is a large part of the chip-configuration process, and the procedure of masking one or more pixels is briefly described here.

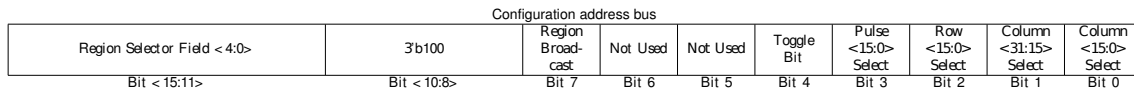


Figure E.1: The pixel-matrix addressing scheme [10].

The pixel matrix is divided into 32 regions of size 32×16 . When configuring a pixel for masking, the (global) *pixel configuration register* is first set; this is buffered and applied to all mask- and pulse registers, but not yet applied if the row- and column selection bits in figure E.1 are not set. A sub-section in a given region is then selected by first setting the region selector- and column selection fields to their desired values in the address field above, and writing to that address a value that selects one or several of these columns. To select a row, the same region selection field is applied with the row-bit set and the column-fields cleared, and a value corresponding to the desired row written to that address. The value that was written to the pixel configuration register is now latched to the selected pixels. The column- and row bits should then be cleared. Masking a single pixel hence requires four writes to the ACM:

- One write to the pixel configuration register.
- One write to select a column.
- One write to select a row.
 - Whatever was written to the pixel configuration register is now applied.
- One write to clear the column- and row select bits.

When applying a mask to the entire chip during configuration, the first write-operation that sets the pixel configuration register is only performed once. Furthermore, looping over all pixels is achieved by selecting first a selection of columns

corresponding to the bits to be masked in a given row, and then selecting that row. In this manner, 16 pixels are set at a time if the entire matrix is iterated over. If configuring all pixels of the matrix, then, the number of write-operations to be performed is:

- Five initial writes to first unmask all pixels, followed by setting the PIXCNFG_DATA bit.
- For all rows in each of the 32 regions:
 - One write to select a group of columns.
 - One write to select a row - this masks the pixels that are selected by the columns/row combination.
 - One write to clear the selection bits.
 - Repeat.

Multiple rows, regions or even the entire matrix can be configured simultaneously in this way.

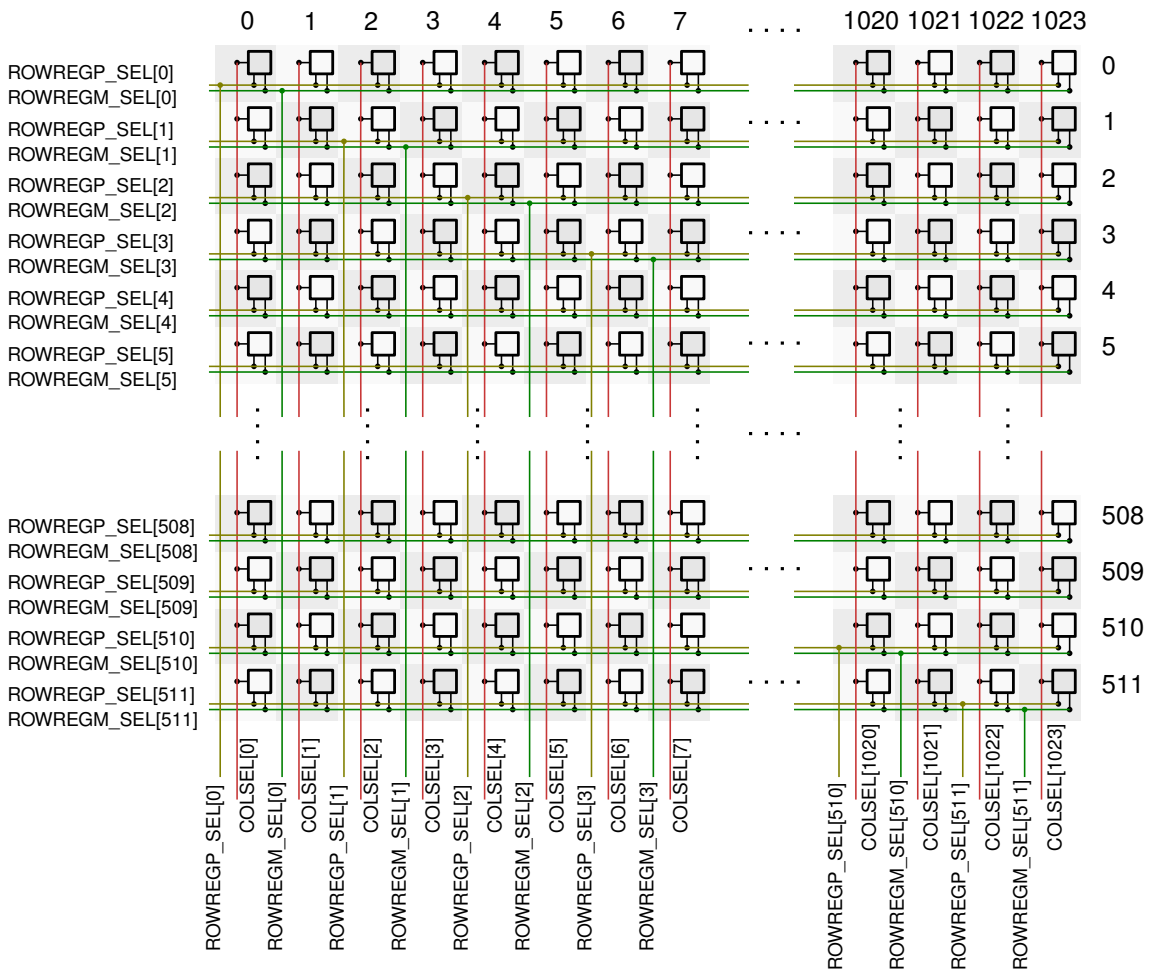


Figure E.2: The pixel-matrix [10].

E.2 Documentation and Commenting

C

As the code is part of the pCT work package 3 repository, emphasis was placed on proper documentation. In-code documentation follows the typical Javadoc syntax:

```
/**
 * @brief Read an ALPIDE register.
 *
 * @param ctrl_base_addr the base-address of the ALPIDE control module.
 * @param chip_id ID of the chip.
 * @param reg_addr address of register to read.
 *
 * @return the value stored in the register.
 */
u16 read_alp_reg(u32 ctrl_base_addr, u8 chip_id, u16 reg_addr);
```

Doxygen-generated documentation is also produced for all code, and is available on the official pCT work package 3 Gitlab repository¹. A document describing the embedded software in general is also found here.

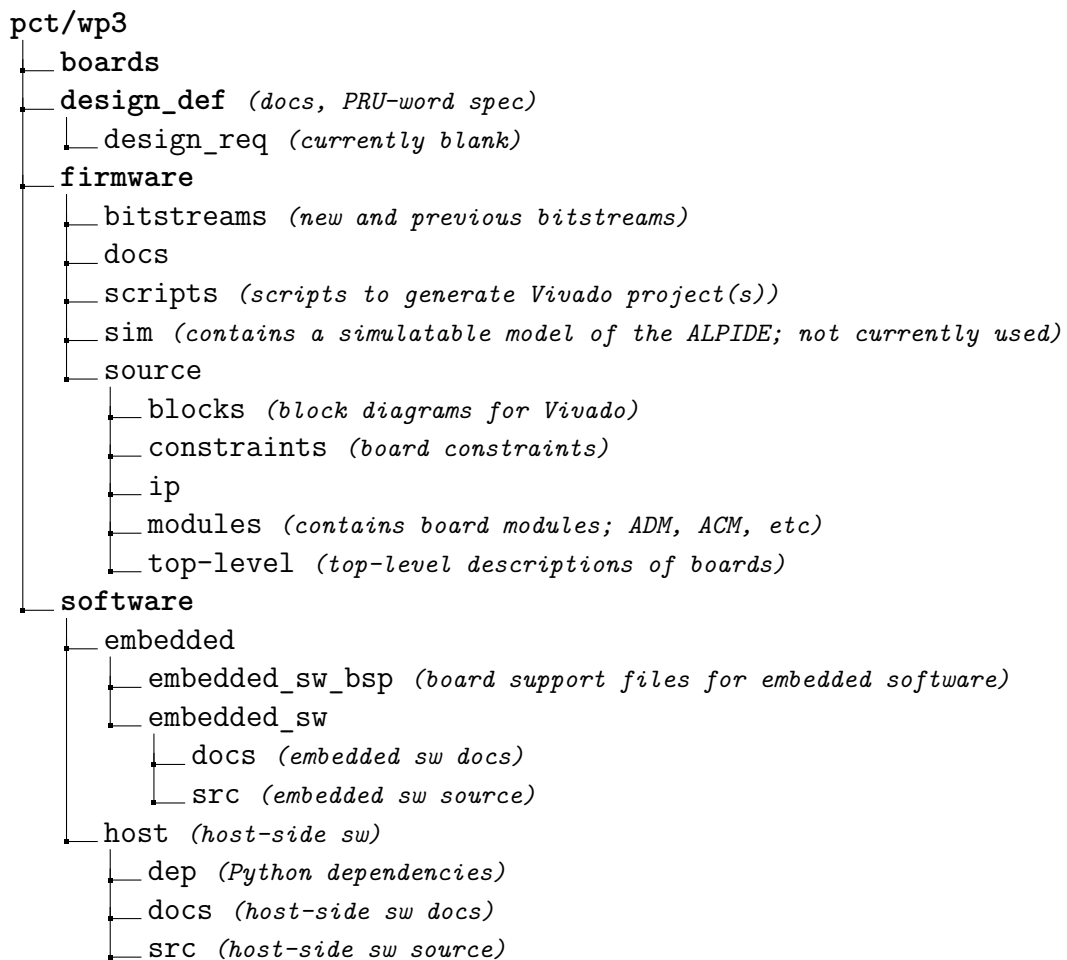
Documentation in general follows a "header files are for the user, source files for the developer"-mantra; source files are commented thoroughly, while the header files are commented more simply, with little details of e.g actual implementation.

¹<https://git.app.uib.no/pct/wp3>

Repository Structure

Version control with Git was used continuously in order to integrate the firmware and software developed as part of the work done in this thesis in to the pre-existing pCT Work-Package 3 repository. The structure of this repository is shown below.

<https://git.app.uib.no/pct/wp3>



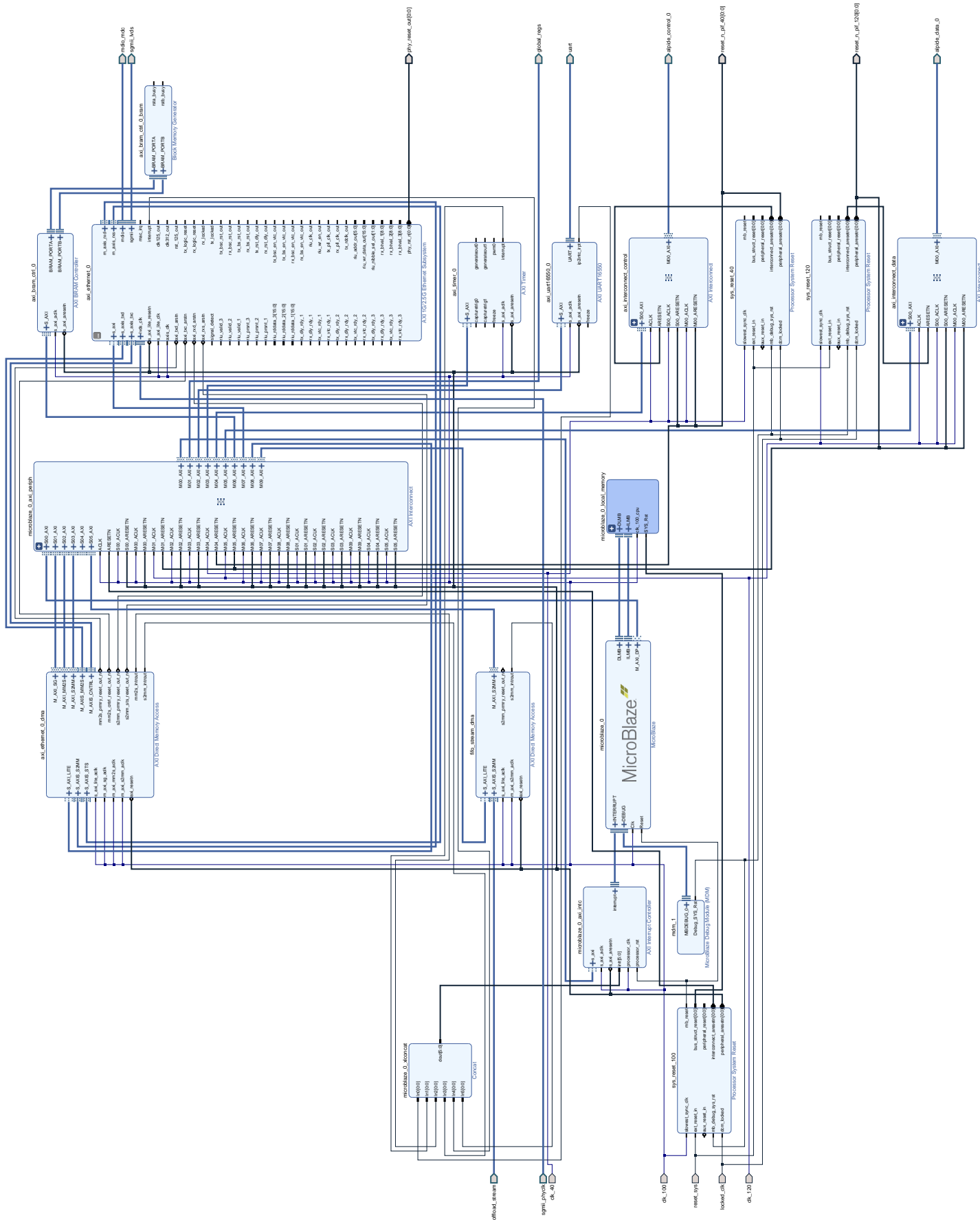


Figure F.1: The MicroBlaze subsystem as it was implemented on the VCU118 platform.

Bibliography

- [1] Particle Therapy Co-Operative Group "*Particle Therapy Patient Statistics (per end of 2016)*". Particle Therapy Co-Operative Group, 2017. [Online]. Available: https://www.ptcog.ch/archive/patient_statistics/Patientstatistics-updateDec2016.pdf
- [2] D. J. Brenner, E. J. Hall "*Computed Tomography — An Increasing Source of Radiation Exposure*". New England Journal of Medicine 357.22: 2277-2284, 2007. [Online]. Available: <https://www.nejm.org/doi/full/10.1056/NEJMra072149>
- [3] J. D. Mathews, et al. "*Cancer risk in 680 000 people exposed to computed tomography scans in childhood or adolescence: data linkage study of 11 million Australians*". Bmj 346: f2360, 2013. [Online]. Available: <https://www.bmj.com/content/bmj/346/bmj.f2360.full.pdf>
- [4] D. L. Miglioretti., et al. "*The Use of Computed Tomography in Pediatrics and the Associated Radiation Exposure and Estimated Cancer Risk*". JAMA pediatrics 167.8: 700-707, 2013
- [5] K. Yamoah, P. A. S. Johnstone "*Proton beam therapy: clinical utility and current status in prostate cancer*". OncoTargets and therapy 9: 5721, 2016
- [6] H. E. S. Pettersen, et al. "*Proton tracking in a high-granularity Digital Tracking Calorimeter for proton CT purposes*". Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 860: 51-61, 2017. [Online]. Available: <https://arxiv.org/ftp/arxiv/papers/1611/1611.02031.pdf>
- [7] R. P. Johnson, et al. "*Results from a Prototype Proton-CT Head Scanner*". Conference on the Application of Accelerators in Research and Industry, CAARI 2016, 30 October – 4 November 2016, Ft. Worth, TX, USA. [Online]. Available: <https://arxiv.org/pdf/1707.01580.pdf>
- [8] P. C. Shrimpton, M. C. Hillier, M. A. Lewis, M. Dunn "*Doses from computed tomography (CT) examinations in the UK-2003 review (Vol. 67)*". National Radiological Protection Board, March 2005
- [9] G. A. Rinella "*The ALPIDE pixel sensor chip for the upgrade of the ALICE Inner Tracking System*". Nuclear Instruments and Methods in Physics

- Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 824, pp. 434-438, 7. May 2016. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0168900216303825>
- [10] ALICE ITS ALPIDE development team *ALPIDE Operations Manual version 0.3*. CERN, 2016
- [11] H. Shafiee *Prototyping of a Tracking Calorimeter for Computed Tomography in Proton Therapy [Presentation]*. UiB pCT workshop, 07 March 2018
- [12] H. F. -W. Sadrozinski, et al. "Operation of the Preclinical Head Scanner for Proton CT". Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 831, pp. 394-399, 21. September 2016. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0168900216001455>
- [13] G. Poludniowski, N. M. Allinson, P. M. Evans "Proton radiography and tomography with application to proton therapy". The British journal of radiology, 88(1053), 20150134, 2015
- [14] Texas Instruments "Interface Circuits for TIA/EIA-232-F" Texas Instruments, September 2002. [Online]. Available: <http://www.ti.com/lit/an/s11a037a/s11a037a.pdf>
- [15] Samtec "PCIe Optical Half Cables Application Note". Samtec, January 2016. [Online]. Available: http://suddendocs.samtec.com/notesandwhitepapers/pcie_half_cable_app_note.pdf
- [16] E. Haseloff "Latch-Up, ESD, and Other Phenomena". Texas Instruments, Application Report, SLYA014A - May 2000. [Online]. Available: <http://www.ti.com/lit/an/slya014a/slya014a.pdf>
- [17] J. Szornel "Radiation Effect Studies on ALPIDE at 88" Cyclotron" [Presentation]. US LHC User's Meeting, November 2016. [Online]. Available: https://indico.cern.ch/event/561618/contributions/2354431/attachments/1365249/2067938/jms_alpide_rad.pdf
- [18] Xilinx "UltraScale+ FPGAs - Product Tables and Product Selection Guide". Xilinx, 2018. [Online]. Available: <https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf>
- [19] Xilinx "MicroBlaze Processor Reference Guide (UG984)". Xilinx, October 5, 2016 [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug984-vivado-microblaze-ref.pdf
- [20] A. Hanafi, M. Karim "Embedded Web Server for Real-time Remote Control and Monitoring of an FPGA-based On-Board Computer System". LISTA Laboratory – Faculty of Sciences Dhar el Mahraz University Sidi Mohammed Ben Abdellah, 2015. [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7106185>

- [21] Eclipse "*Embedded MQTT-SN C/C++ Client*". [Online]. Available: <https://www.eclipse.org/paho/clients/c/embedded-sn/>
- [22] OPC Foundation "*What is OPC?*". OPC Foundation. n.d. [Online]. Available: <https://opcfoundation.org/about/what-is-opc/>
- [23] C. V. Soare "*OPC UA IPbus server*". Atlas Central DCS, CERN Summer Student Report, September 2015. [Online]. Available: <https://cds.cern.ch/record/2055198/files/OPCUA-IPbus-Server-Cristian-Valeriu-Soare-report.pdf>
- [24] D. R. Hlaluku "*Tests with beam setup of the TileCal phase-II upgrade electronics*". High Energy Particle Physics Workshop 2017, 2017 J. Phys.: Conf. Ser. 889 012005
- [25] W. Kamp "*AXI over Ethernet; A Protocol for the Monitoring and Control of FPGA Clusters*". High Performance Computing Research Lab Auckland University of Technology, New Zealand. [Online]. Available: <https://ieeexplore.ieee.org/document/8280120/>
- [26] "*Etherbone*". [Online]. Available: <https://www.ohwr.org/projects/etherbone-core>
- [27] S. Cheshire, M. Baker "*Consistent overhead byte stuffing*". IEEE/ACM transactions on networking, vol.7, no. 2, April 1999. [Online]. Available: <http://www.stuartcheshire.org/papers/COBSforToN.pdf>
- [28] IEEE Computer Society "*IEEE Standard for Ethernet*". IEEE Std 802.3TM-2012 (Revision of IEEE Std 802.3-2008)
- [29] Xilinx "*AXI 1G/2.5G Ethernet Subsystem v7.0 Product Guide (PG138)*". Xilinx, April 5, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_1/pg138-axi-ethernet.pdf
- [30] Xilinx "*AXI UART 16550 v2.0 LogiCORE IP Product Guide (PG143)*". Xilinx, October 5, 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_uart16550/v2_0/pg143-axi-uart16550.pdf
- [31] Xilinx "*AXI UART Lite v2.0 LogiCORE IP Product Guide (PG142)*". Xilinx, April 5, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf
- [32] Xilinx "*LightWeight IP Application examples*". Xilinx, November 21, 2014. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf
- [33] Xilinx "*AXI DMA v7.1 LogiCORE IP Product Guide (PG021)*". Xilinx, October 4, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

-
- [34] Missing Link Electronics, Inc. *"10G/25G TCP/IP Stack"*. Missing Link Electronics, Inc. [Online]. Available: <https://www.xilinx.com/products/intellectual-property/1-4dbvjf.html>
- [35] Intilop Inc *"10G TCP and UDP Offload Engine. Full TCP UDP Offload"*. Intilop Inc. <https://www.xilinx.com/products/intellectual-property/1-4dbvjf.html>
- [36] M. J. Christensen, T. Richter *"Achieveing reliable UDP transmission at 10 Gb/s using BSD socket for data acquisition systems"*. [Online]. Available: <https://arxiv.org/pdf/1706.00333.pdf>
- [37] F. Costa et al. *"The new frontier of the DATA acquisition using 1 and 10 Gb/s Ethernet links"*. TIPP 2011 - Technology and Instrumentation for Particle Physics 2011
- [38] G. Bauer, et al. *"10 Gbps TCP/IP streams from the FPGA for HighEnergy Physics"*. 20th International Conference on Computing in High Energy and Nuclear Physics
- [39] O. S. Grøttvik *"Design of High-Speed Digital Readout System for Use in Proton Computed Tomography"*. UiB, Master thesis, June 2017. [Online]. Available: http://bora.uib.no/bitstream/handle/1956/16041/thesis_final.pdf?sequence=1&isAllowed=y
- [40] Xilinx *"AXI Interconnect v2.1 LogiCORE IP Product Guide (PG059)"*. Xilinx, December 20, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf
- [41] G. van Rossum, B. Warsaw, N. Coghlan *"PEP8 - Style Guide for Python"*. Python, August 1, 2013. [Online]. Available: <https://www.python.org/dev/peps/pep-0008/>