UNIVERSITY OF BERGEN

Western Norway
University of
Applied Sciences

Department of Computing, Mathematics and Physics

MASTERS THESIS

Data Acquisition and Testing Software for a Proton Computed Tomography System

Author: Håkon Andreas Underdal

Supervisor: Håvard Helstrup

Co-supervisor: Johan Alme

June 3, 2019

There ain't no such thing as a free lunch

Unknown

Abstract

At the University of Bergen (UiB), a proton computed tomography(CT) prototype is under development. Proton CT (pCT) is an alternative to X-ray for planning proton radiation therapy. Currently, an X-ray is performed to prepare proton treatment. This requires a conversion of X-ray attenuation to the relative stopping power of protons which leads to a systematic error entailed by the translation process. A pCT system may reduce the error margin, hence minimizing the radiation dosage and increase the safety of particle therapy.

Developing a complete pCT readout and control system is a large and complicated task involving expertise from different engineering disciplines and other fields within natural science. This thesis will focus on the host software interfacing with the pCT readout unit (PRU) that is going to be implemented in the pCT machine. The host software will be used to manage the PRUs from a control room. An operator of the host software is able to evaluate the status of the system, such as the current state of the pixel detectors and readout units. This allows for configuration and calibration of the PRU and PRU peripherals.

This thesis will discuss the implementation details of the host software and how it may be developed further. The host software is run in a Linux environment and developed in C++ to accommodate future requirements in time of execution and management of large data sets. It has been tested continuously by acquiring and analyzing detector data where the pixel detector has been irradiated with low-intensity sources at UiB. The software has also been run in a radiation experiment at the German Cancer Research Center beam facility. The host software is a framework with multiple complied tests that can be executed to verify the operation of the pixel detector.

The thesis also includes an overview of existing work achieved by the pCT group at UiB and suggestions for improvements. Furthermore, it includes an introduction to particle therapy and details of the particle detector.

Acknowledgment

I want to thank my supervisors, professor *Håvard Helstrup* and associated professor *Johan Alme*. Their continuous feedback and advice through this year have been invaluable. I also want to give a special thanks to *Ola Slettevoll Grøttvik* for taking his time to answer my questions and providing exciting and challenging tasks along the way. I am also thankful for the pCT group at UiB allowing me to contribute to the project and for letting me partake in the beam experiment in Heidelberg.

A huge appreciation goes toward *Annar Eivindplass Hilde* whom I have worked closely with the past year. I will never forget our technical and philosophical discussions in the microelectronics lab alongside the quality banter. Additional appreciation goes out to my fellow students these past five years.

Lastly, I want to thank my family for believing in me when I did not. Without their continued support, I would not be where I am today.

H.A.U.

Contents

List of Figures

List of Tables

List of source codes

List of abbreviations

ALPIDE ALICE Pixel Detector

BC Bunch Crossing

COBS Consistent Overhead Byte Stuffing

CT Computed Tomography

DAC Digital-to-analog converter

FPGA Field-programmable Gate Array

GUI Graphical User Interface

HU Hounsfield scale

ITS Inner Tracking System

INL Integral Non-Linearity

MEB Multi Event Buffer

PRU Proton CT Readout Unit

RSP Relative Stopping Power

TCP Transmission Control Protocol

Chapter 1

Introduction

1.1 Motivation

Digital cameras use image sensors to capture light (photons) to create images. The image sensors are two-dimensional arrays containing pixels that register photons passing through. Depending on the energy of the photon a wavelength is given which corresponds to a color. There are mainly two types of image sensors; CCD and CMOS. In a CMOS sensor, a pixel will output a voltage if it is hit by light. An image is created by scanning the pixels either by one row at a time until the whole frame is complete, or by scanning odd lines then even lines and combine them into a complete frame. A higher number of pixels increases the resolution of the image, but also increase the amount of data transmitted of the sensor. The iPhone 8 has a seven million pixels image sensor. The device has to read all pixels, identify their position, combine into a frame, check for errors and add metadata to the image. Raw images usually consume vast amounts of memory so the device will need to compress the image to a certain degree without losing too much quality. This is a specialized process often done by a subsystem referred to as a readout system. The readout system is implemented to carry out the tasks above and forward the final result to other peripherals. Image sensors are used in many fields ranging from medical imaging to astronomy to capture different types of particles such as ions. Each case needs its own readout system to manage and transmit data.

The topic of this thesis is the development of a readout and control system for medical purposes, more specifically a pixel-based proton computed tomography system (CT) based on detector technology from the ALICE experiment at CERN. The focus will be on the host software interfacing with a readout unit making up a complete readout and control system.

Today, mostly high energy photons are used in cancer treatment. In conventional radiography, a patient is scanned with X-ray and a radiation dose is calculated based on the scan results. Photons work very well when malignant cells that are just below the patient's skin. A problem arises when tumors grow deeper into the body. Photons are a problem because of the damage it does to the tissue surrounding the target of treatment. Researches have in the recent years looked towards the use of protons and other charged particles. These methods benefit the patients because of the more precise dosage delivery to the diseased tissue. Increasing precision of dosage lowers the risk of damaging organs in the proximity of the radiation target. The political willingness in Norway to build particle centers is strong. In 2017 Jonas Gahr Støre promised one billion NOK to build particle therapy centers if elected [1]. The same year the Norwegian government released a statement promising to fund centers in Oslo and Bergen which is to be finished in 2023 and 2025 respectively[2].

The University of Bergen has been granted funds to develop a prototype of a computed tomography (CT) system replacing X-rays with protons. The prototype is going to use several layers of the CERN-developed monolithic active pixel sensors. These chips were developed as a part of the upgrade of the Inner Tracking System (ITS) for the ALICE detector [3]. The ALICE Pixel Detector (ALPIDE) is sensitive to charged particles. With adequate software, it is possible to reconstruct particle paths and calculate their residual energy.

1.2 Thesis Motivation

The primary objective of this thesis is to develop a software framework to acquire and analyze data from ALPIDE detectors. An existing framework has been developed at CERN for other readout systems. The software provides configuration, calibration, and testing of the ALPIDE chips. At the University of Bergen, a new readout board has been designed with new firmware, embedded software and communication protocols. The focus of this thesis is to port the existing software and extend its functionality to the new setup.

This work builds upon the software produced at CERN and around the existing work achieved by the pCT group in Bergen. THe documentation for the software was limited, so a significant amount of time was spent reading and understanding the different software modules. A considerable part of the software is made to test the electrical yield, modules and tracking particles in the ALPIDE detector with the ported framework. Tests were done in Bergen with a gamma source, and with protons, helium and carbon in Heidelberg.

A major challenge was to verify and validate the software when communicating with the readout-electronics. The author of this thesis has limited knowledge of microelectronics, and a considerable portion of the time was spent learning how different electronic components function on a lower level. That also includes time spent on understanding how the ALPIDE chip itself operates.

1.3 Outline

This thesis is split into the chapters below:

- **Chapter 2: Computed Tomography and Particle Therapy** This chapter aims to describe the conventional radiography and the advantages for proton CT. How proton beam curves differentiate from photons in energy deposition.
- **Chapter 3: ProtonCT UiB and ALPIDE Chip** This chapter aims to present the proton CT prototype and architecture and principle of operation of the ALPIDE chip.
- **Chapter 4: The pCT Readout Unit and Control System** This chapter describes the readout unit and its subsystems. The readout unit is the interface between the ALPIDE chips and the host software described in this chapter.
- **Chapter 5: Host Software** In chapter 4 the readout system is discussed. A host software has been developed to interface the readout unit and assist in data acquisition and analysis of detector data. This software is described in this chapter together with future development.
- **Chapter 6: Testing** The pixel detectors are thoroughly examined before employed through a series of tests. The tests are integrated into the host software described in chapter 5. In this chapter the tests is described with full procedure, classification and real results.
- **Chapter 7: Conclusions** This chapter discusses the result of this thesis. It goes through the design choices and presents recommendations for further work.
- **Bibliography**
- **Appendix**

Chapter 2

Computed Tomography and Particle Therapy

In conventional radiography, an x-ray is passed through an object and registered by a detector on the opposite side. Depending on the density and composition of the object, an amount of the radiation is absorbed, and a 2D image of the internals of the object is created. In a standard computed tomography (CT) system the same principle is applied with a 360 degree rotating X-ray source sampling along the way. The produced 2D images are stacked together to form a 3D image. This image may reveal abnormalities such as tumors and damaged organs within a patient.

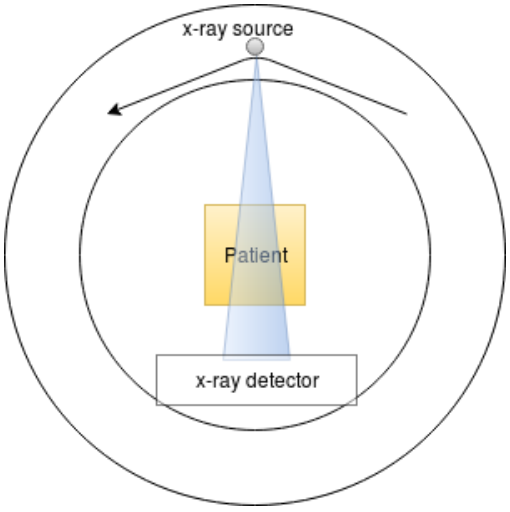


Figure 2.1: CT scan

2.1 Ionizing Radiation

In 2012 CT scan procedures contributed to roughly 50-80 % of the total population dose of x-ray imaging in the Nordic countries. In some countries, this exposure is higher than that from natural sources [4]. X-rays have enough energy to overcome the binding energy of electrons orbiting atoms and molecules. Therefore x-rays can knock electrons out of their orbits and create ions. Because of this x-ray are categorized as ionizing radiation. The most common scenario is the creation of hydroxyl which may interact with nearby DNA to cause strand breaks or base damage[5].

2.2 Particle Therapy Motivation

In standard radiotherapy, ionizing radiation is used to control the growth or kill malignant cells. There are several sources used in radiation therapy. Among them are photons, protons, helium, and carbon. To examine the differences, a plot of the ionizing radiation's energy loss while it traverses through matter is used. This plot is called a Bragg Curve. Figure 2.2 shows how protons and photons lose their energy while traveling through water. Photons have no mass and no charge and can easily penetrate any tissue. The photon curve shows however that most of the dosage is delivered early. This is not a problem if the diseased tissue is just below the patient's skin, but is complicated when it resides deeper into the body.

Protons do have mass and lose its speed faster contrary to photons when traveling through tissue. Protons deliver its energy much more precisely making it easier to irradiate accurately at any given depth. The peak of the curve is called a Bragg Peak. It occurs just before the particle goes to rest. Therefore the goal is to direct the beam so that the Bragg Peak occurs inside the diseased tissue.

2.3 Proton CT

The energy loss of protons is dependent on their initial energy, and the properties of the matter they are traveling through. Proton dosage levels are calculated from the relative stopping power (RSP). RSP is the retarding force on the particle as it interacts with materials. To obtain the RSP an x-ray CT scan is performed on the patient. The scan accumulates Hounsfield Units (HU), which is a number describing the relative inability of a particle to pass certain materials. Conversion of the HU obtains RSP which leads to a systematic range uncertainty of 2-3 % [7].

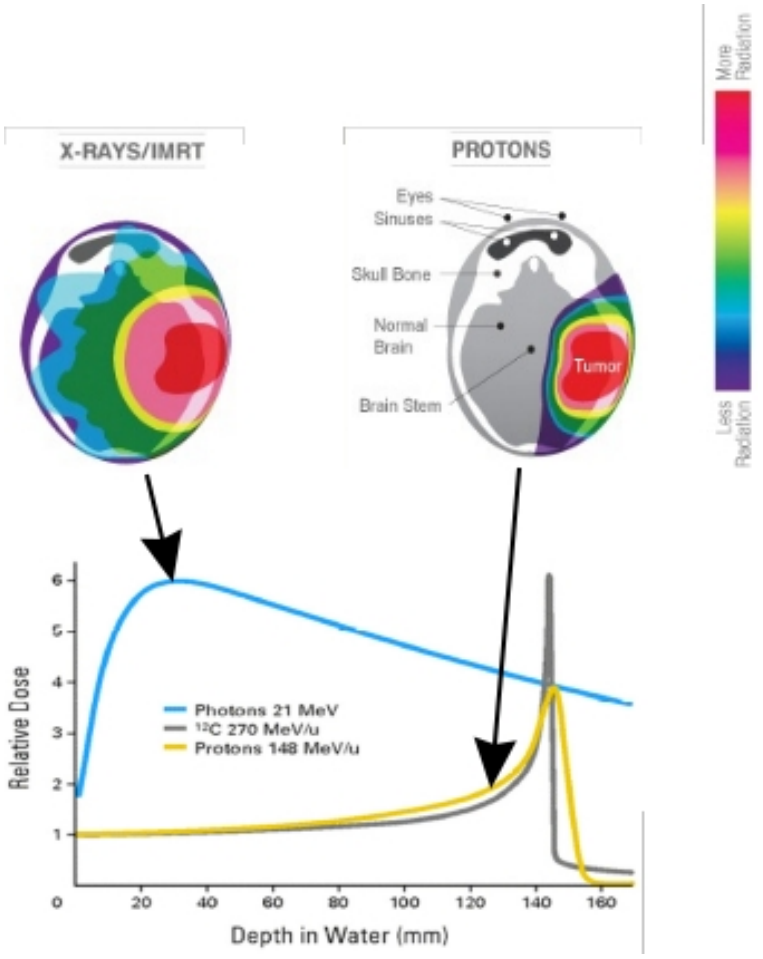


Figure 2.2: Proton energy curve and impact vs. photons [6]

In 2015 a scientific study showed that simulated proton CT had maximal range errors of 0.5 % compared to 7.4 % of single energy x-ray CT [8]. The motivation to develop a proton CT system is indeed to minimize the range error, hence making treatment safer for the patients.

In a proton CT scan, a proton beam is sent through the patient. The intensity of the proton needs to be high enough to travel through the patient being scanned. The measured residual energy together with an estimated path of the proton can calculate the RSP. With the tracking planes and distance to the treatment object, the proton's path can be measured and information about what and where the proton lose its energy.

2.4 Existing Proton CT Systems

Due to the increasing use of protons in radiation therapy, several pCT prototypes have been designed and even built. In order to track protons, a set of tracking planes are placed before and after the target of treatment with a measurement apparatus at the rear end. The apparatus is used to calculate residual energy. Together with the tracking planes, it is possible to estimate the path of the protons and determine the proton range for treatment usage. The arrangement of the tracking planes varies between the prototypes. The UiB pCT will only install tracking planes after the object of treatment.

In 2016 a prototype for a preclinical head scanner for pCT was suggested[9]. It uses tracking planes between the phantom. The first plane measures the direction and position of the incoming proton beam, while the latter measure the proton beam as it exits the phantom. A 5-stage scintillator detector is placed at the rear end. It will stop the protons and calculate the residual energy. While the tracking planes are capable of measuring multiple protons simultaneously, the scintillator detector has no lateral segmentation. Thus, the detector can only analyze one proton at a time. The system is still capable of completing a full scan in 10 minutes and measuring 1 million protons per second.

The projection measurement of protons produces vast amounts of data. A significant challenge is to offload, check the integrity and analyze this data in real-time. The system explained above estimates about 20 GB of data each scan. The computations are so extensive that an off-the-shelf microprocessor is often not a viable option. Instead, a custom ASIC or an FPGA is employed to offload events from the particle detectors.

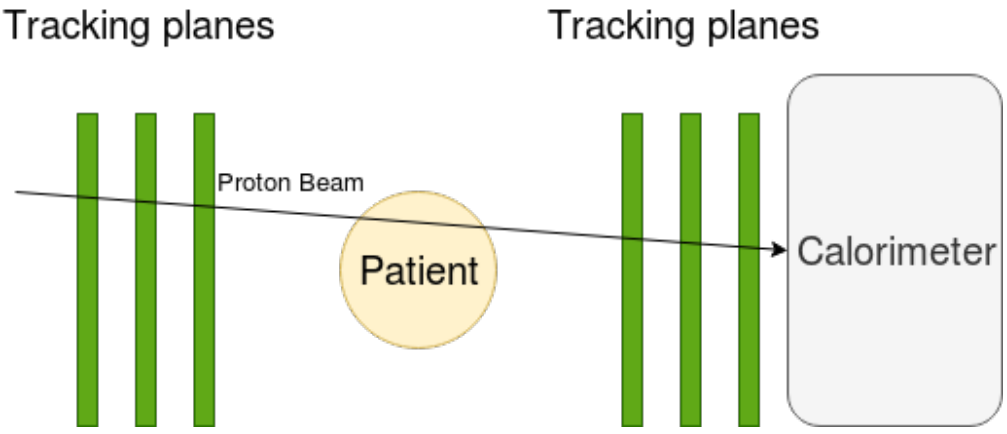


Figure 2.3: Example of a proton ct layout. A proton beam trajectory is measured before and after the patient. The calorimeter is used to measure the residual energy after the patient

Chapter 3

Proton CT UiB and ALPIDE chip

3.1 ALPIDE[1]

The ALPIDE chip was designed for Inner Tracking System of the ALICE experiment at CERN [11]. It is capable of registering charged particles with its 512 x 1024 sensitive pixel sensor matrix. The sensors are placed on a 180 nm CMOS Imaging process of TowerJazz. Each sensor contains a charge collection diode where a voltage will appear when charged particles irradiate the surrounding area.

3.1.1 Pixels

The pixel sensors have a sensing diode, a front-end amplifying and shaping stage, a discriminator and a digital section. This is illustrated in figure 3.1. A bias voltage can be used to tune the global threshold value. Any voltage that exceeds this value will cause the output from the Front-end low and activate the active-low input signal in the digital section. If active-low input is low while the STROBE signal is high, a digital one is latched into the Multi-event buffer (MEB). The value in the buffer is read out as a "hit", framed and transmitted of the chip. The STROBE signal is generated by TRIGGER which can be produced internally on the chip or externally via the control interface. The STROBE duration and gap between subsequent STROBE pulses are adjustable and can range from 25 ns to 1638.4 us.

[1] This chapter is mostly based on the ALPIDE Operations Manual [10]

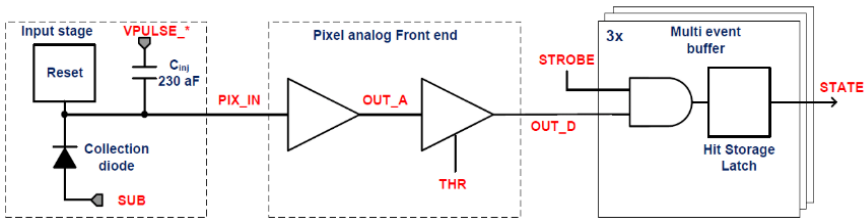


Figure 3.1: The ALPIDE pixel cell [10]

3.1.2 Pixel Indexing

The pixel sensor matrix is divided into 32 regions with each region containing 16 double columns as illustrated by figure 3.2. The pixels are arranged in the double columns with priority encoders in the middle effectively making a readout array of 512 priority encoders. The priority encoders determine how the pixels are indexed. Figure 3.3 shows how pixels are arranged in the double columns. With this arrangement, it is possible to find each pixel by row and column via region ID, double column and address given by the priority encoder.

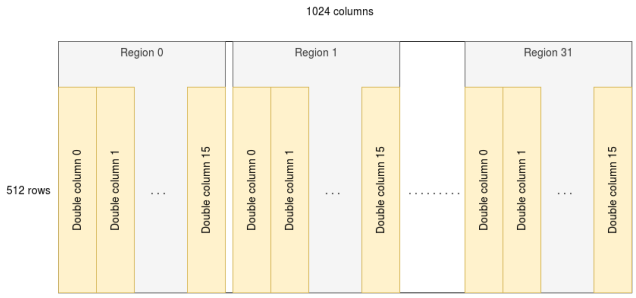


Figure 3.2: Region arrangement - Leftmost is 0 and rightmost is 31

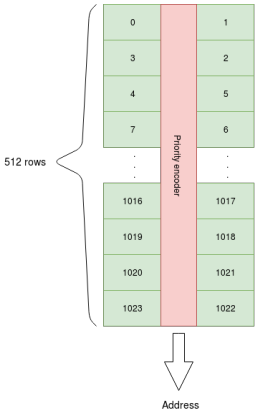


Figure 3.3: Pixel indexing by priority encoder inside a double column

3.1.3 Pixel Masking and Digital Pulsing

The in-pixels digital circuitry gives the ability to mask and set pixel states. Figure 3.4 shows the functional diagram of the pixels. The mask latch is activated with the CNFG_COLSEL and CNFG_ROWREGM_SEL signals. The column and row select signals offer the capability to mask pixels individually or by row. The PIXCNFG_DATA signal is then set high, effectively setting the output of the mask latch high. This output signal goes into the active low input of the last AND gate forcing the STATE low regardless of what is stored in the MEB.

The pulsing latch is activated with CNFG_ROWREGP_SEL and CNFG_COLSEL. It is important to note that CNFG_COLSEL is a shared signal for masking and pulsing, while they have their own row selection signal. The PIX_CNFG_DATA signal is latched into PULSE_EN and the output is driven into the MEB with the global DPULSE signal.

With this logic, it is possible to generate different pulsing patterns to validate the pixel logic.

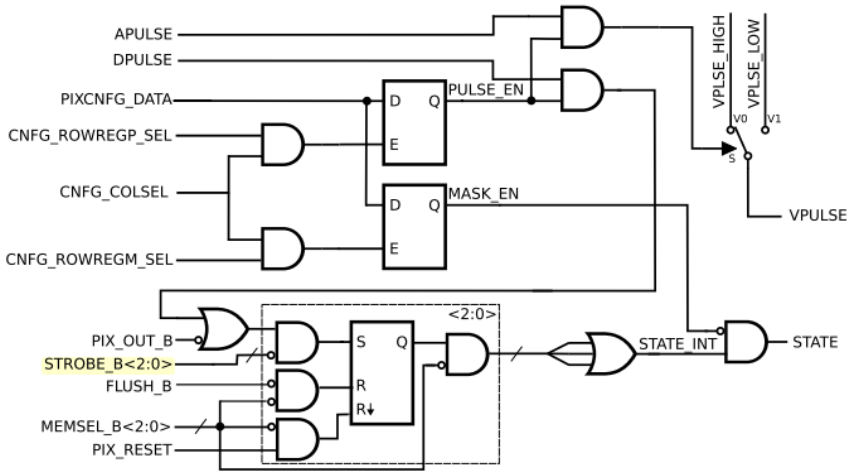


Figure 3.4: The ALPIDE pixel cell [10]

3.1.4 Data Transmission

The ALPIDE has two interfaces for transmitting data. One serial port and one parallel port with 8 bits. If the ALPIDE is configured in inner-barrel (IB) mode, the serial link port is used. The serial link speed is programmable and can be 1200 Mb/s, 600 Mb/s or 400 Mb/s. The data is 8b/10b encoded effectively giving a transfer speed of 960 Mb/s. The outer-barrel (OB) configuration is not used in UiB pCT project due to the relative high occupancy. In an OB configuration, the parallel port is used to connect the slave chips and master chip. The master chip acquires data from the slave chips and transmits their data with a bit rate of 400 Mb/s, or 320 Mb/s after encoding.

3.1.5 Framing and Triggering

One of the core functionalities of the ALPIDE is to store the state of the pixels and transmit that information to a readout system. A frame is a collection of pixels states within a specified time window. It can be viewed as a snapshot of the pixel detector grid. To generate a frame the chip has to be triggered.

Each pixel contains a three-hit storage multi-event buffer(MEB). This makes it possible to store three frames without the need to complete a data readout which prevents loss of data.

The ALPIDE offers a 12-bit counter for an internal time reference. It runs continuously to its maximum value of 3563 before it wraps around to zero. The maximum value matches the duration of one orbit of the ALPIDE system clock. Operating the chip with a clock synchronous to the chip's system clock provides a bunch cross (BC) counter. The BC counter can be reset by the BCRST command, and used to synchronize across chips. In the framing process, this counter is used to provide timestamps for frames. The BC counter is especially useful when employing multiple chips in an experiment. The counter gives the ability to track particles passing through several chips.

Frames are encoded and packed into a sequence of data words. All valid data words can be seen in table 3.1. The correct sequence is header word, region header, data short/long trailer word. The header word contains chip id and the bunch counter value. The region header tells in which regions of the pixel matrix the hits are located. Data short contains one hit with one address. Data long can store up to seven hits. It has a starting address, which is always a hit, and a 6-bit hitmap.

Table 3.1: Valid ALPIDE data words [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 EMPTRY 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

3.2 Proton CT UiB

A proton CT prototype is being designed at the UiB. It will use a single Digital tracking calorimeter (DTC) to track and measure the energy of protons. The DTC contains several detector layers and a calorimeter at the rear end. Multiple layers of pixels sensors compose the calorimeter with interleaving absorbing metal layers for energy degradation.

The detector layers in front of the calorimeter are ALPIDEs mounted on staves. The pixel sensors are configured in IB mode to utilize the high readout speed and allow the sensors to be mounted in the stave configuration. The staves are arranged in parallel creating a square. The square is referred to as a layer.

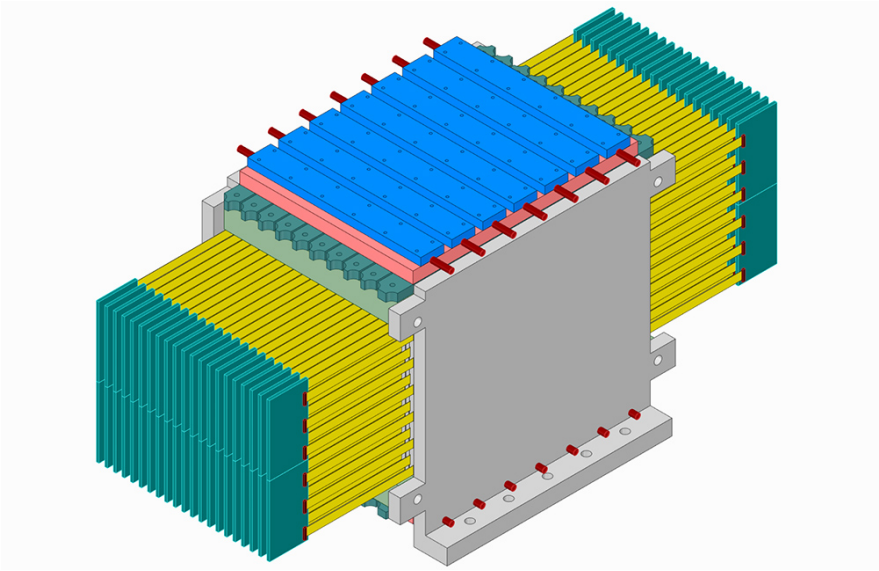


Figure 3.5: 3D model of the pCT UiB DTC

Figure 3.5 shows the design of the prototype. The blue bars on top aids in cooling the DTC. The yellow lines are PCB flex cables connect each stave to a carrier card, which again is connected to a readout board.

Chapter 4

The pCT Readout Unit and Control System[1]

The proton CT readout unit is located and maintained on a development board containing a Xilinx FPGA and other electronic interfaces to connect to ALPIDE chips and host system. The FPGA contains firmware modules to communicate with ALPIDE chips and a master control module to interface these modules and host system. The master control module implements communication between PRU master and host, the assistance of data readout and monitoring of chip and PRU currents and voltages.

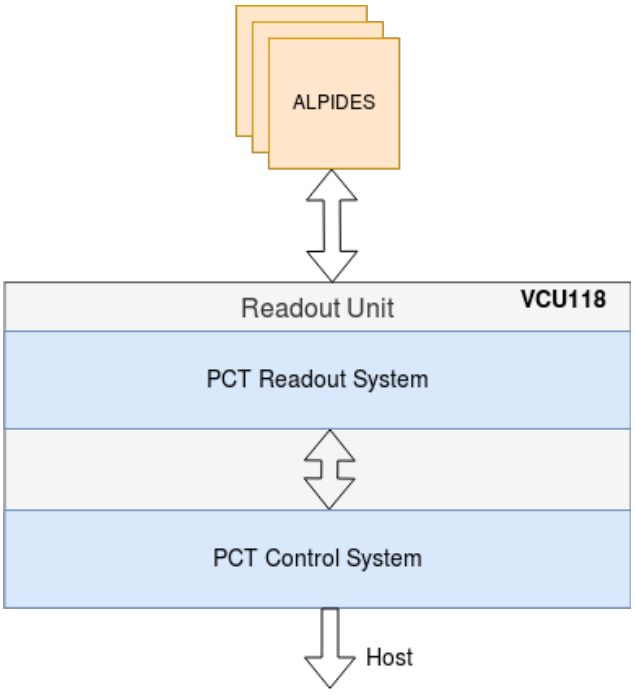


Figure 4.1: Simplified overview of the PRU

[1] This chapter is based on the work acheived by pCT group. Specifically the work of Karl Emil Bohne [12] and Ola Slettevoll Grøttvik [13].

4.1 Readout Unit

The readout unit (RU) is developed on a Xilinx VCU118 evaluation board containing an Virtex Ultrascale+ VUP9 FPGA. It has a Samtec Firefly interface which is used to transfer data between the RU and the ALPIDEs. It also has an ethernet port with tri-speed. It can be configured to operate in gigabit mode (1000 Mb/s) or fast ethernet mode (10/100 Mb/s).

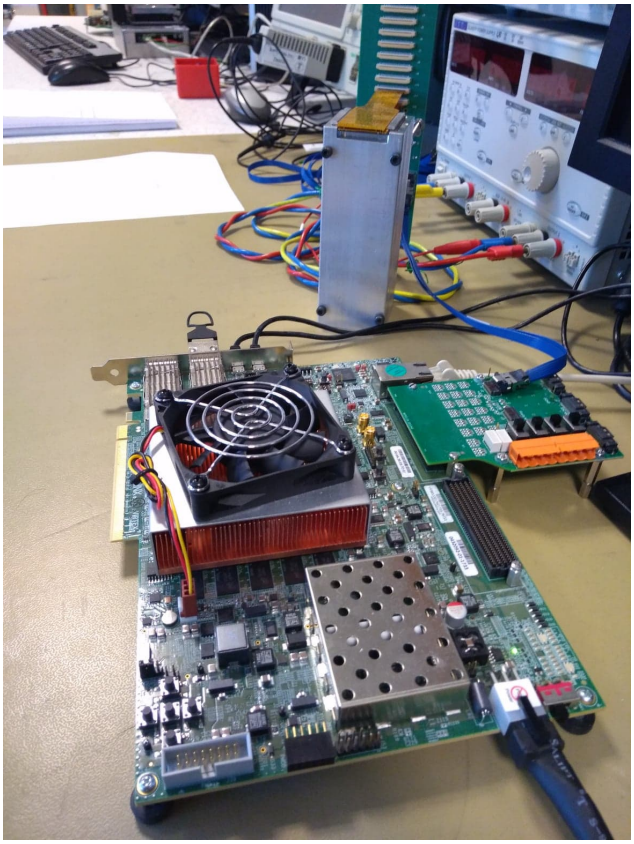


Figure 4.2: Readout unit connected to the mTower setup through a FMC and firefly cable

4.1.1 mTower

The mTower is a cuboid of metal with ALPIDEs stacked on top with a transmission card attached. The card connects the pixel detectors to the readout unit providing control and data readout from the PRU. The ALPIDEs are put together two by two and stacked on top of each other. Each detector pair is connected to the transmission card, but the PRU firmware provides individual channels to distinguish between them.

4.1.2 Firmware modules

The readout unit contains multiple modules to gather data from the ALPIDE and transmit it off the readout board. Figure 4.3 gives a simplified overview of the modules and how they are connected. The ALPIDE control module manages transactions over the ALPIDE slow control interface. The modules provide multiple registers to execute read and write operations on internal registers, perform commands, configure or access memories on the ALPIDE. Each stave has its own ALPIDE data module. The offload module retrieves readout data from all data modules and stores them in a FIFO buffer. The data is then streamed to a multi-channel DMA. The data is then transferred into a block of memory from the DMA. The DMA notifies the CPU that the transfer has been completed and the CPU may take action to transmit the data via TCP/IP[2].

A soft-core microprocessor is deployed to facilitate implementations of communication protocols and assistance of the readout process. The processor runs a real-time operating system (RTOS) to aid in software development and maintenance by providing a multithreaded environment and resource management system.

4.1.3 PRU Data Format

Readout data produced by the ALPIDE is processed by the PRU. Each ALPIDE frame is inserted into the PRU data format[3]. This format has four types (words). Header, data, trailer and empty frame. Each word is 16 bytes long and has the general format shown in table 4.1.

Table 4.1: General PRU Word format [14]

Name	WORD TYPE	RU	STAVE ID	CHIP ID	CONTENT
Length	2	6	4	4	112
Bits	127:126	125:120	119:116	115:112	111:0

A PRU event is an ALPIDE frame on the PRU data format. A PRU event consists of a header word, an arbitrary number of data words and a trailer word. The header word holds information about time and space for event reconstruction in hardware. The data word may hold up to 14 bytes of ALPIDE data. If there are more than 14 bytes of ALPIDE data, another data word will be generated. If less than 14 bytes, the rightmost bytes will be padded with 0XFF. The trailer word designates that the ALPIDE frame has been read out. It holds information about errors and number of bytes transmitted from the ALPIDE. The empty frame indicates that the ALPIDE

[2] A UART interface is available, but TCP/IP has been preferred because of higher throughput
[3] Read the PRU Data Format Specification [14] for more information

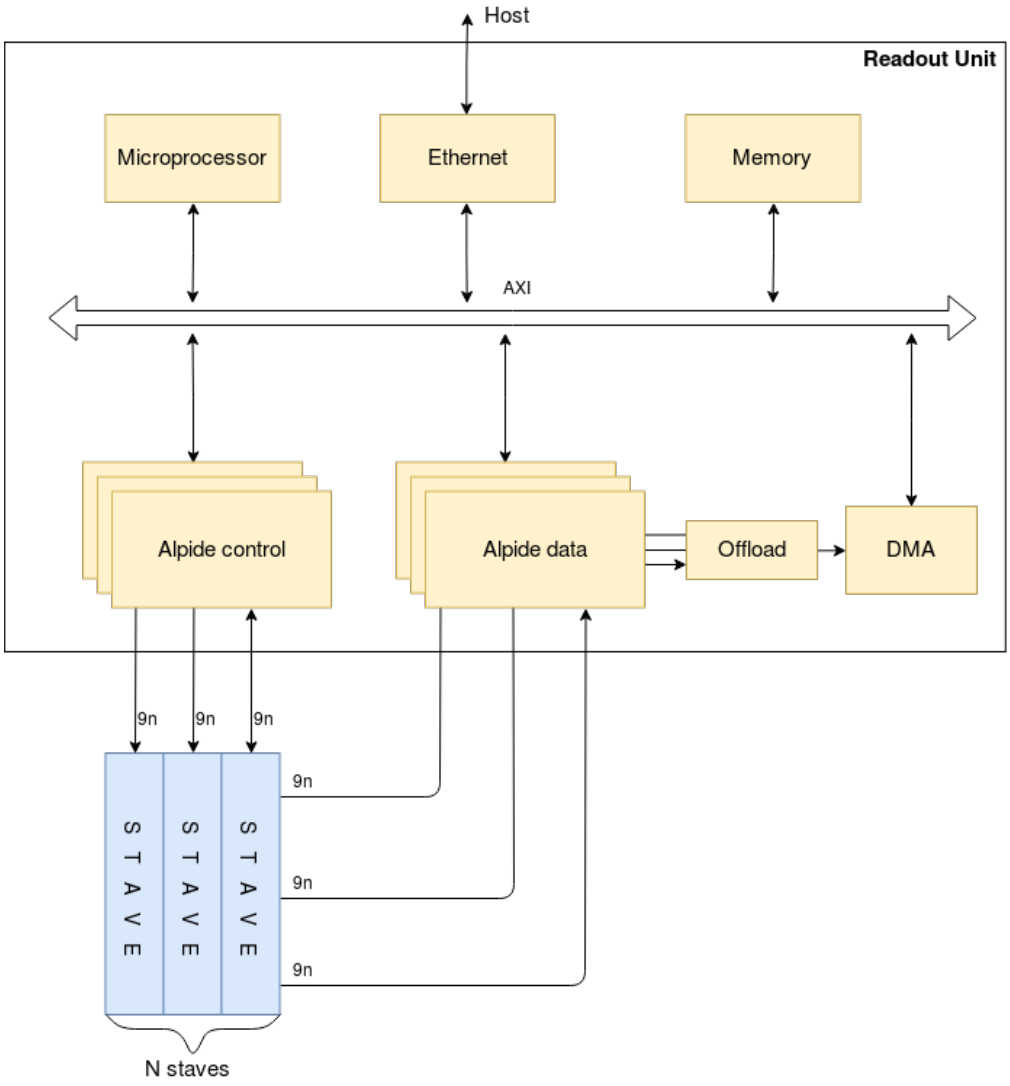


Figure 4.3: Readout unit showing central modules of the system

had no hits. The empty frame holds the same information as a header word, but also contain a bunch counter for the ALPIDE.

4.1.4 Message Format

A control format has been developed to communicate with the control system. The format is split into three sections, header, payload and trailer. The header is 3 bytes containing information about the length and command type of the packet. The length field is 2 bytes, which allows the packet to hold 1 to 65535 bytes of data. The payload contains the actual data in the packet. The trailer is 1 byte and holds a sequence number making it possible to associate a sent message with a received reply.

Table 4.2: General message format developed for communication with the embedded software from host [12].

Header(3 bytes)	Payload(1 - 65535 bytes)	Trailer (1 byte)

Data sent over unreliable interfaces use the Consistent Overhead Byte Stuffing (COBS) to encode data. It adds an overhead of one byte for every 254 bytes in addition to an ending byte. With this scheme, the receiver can recover from malformed packets [4].

The command type field is used to indicate that a message is for a PRU peripheral, one or multiple ALPIDEs.

4.1.5 Embedded Software

The embedded software delivers two communication links between the PRU and the host software; UART and TCP/IP. Through these access points, all PRU firmware modules are accessible for configuration and monitoring. The software is also able to monitor PRU modules and perform ALPIDE procedures such as ADC calibration and pixel masking without external input.

Figure 4.4 gives an overview of the most central threads running on the microprocessor. The TCP/IP and UART tasks forward packets from the host software to the control interface. The control interface decodes the payload in the packet and executes and formulates a reply to which is transmitted by the former tasks.

The reception of a special command spawns the offload-thread. To prevent multiple instances of this thread a flag is set to indicate that it is active. The thread proceeds by transferring data from the DMA to a series of buffers. An interrupt is emitted to signal that the transfer is complete. Buffers containing data is then sent to host.

The monitor task allows monitoring items given an address of a register, a threshold, and a callback function. The callback function will be called if the threshold value is exceeded. An example is the monitoring of voltages or currents of an ALPIDE DAC.

[4] Encoding and decoding of packets with COBS has not been implemented in the test software. This is because it was decided that all communication should be done over TCP/IP

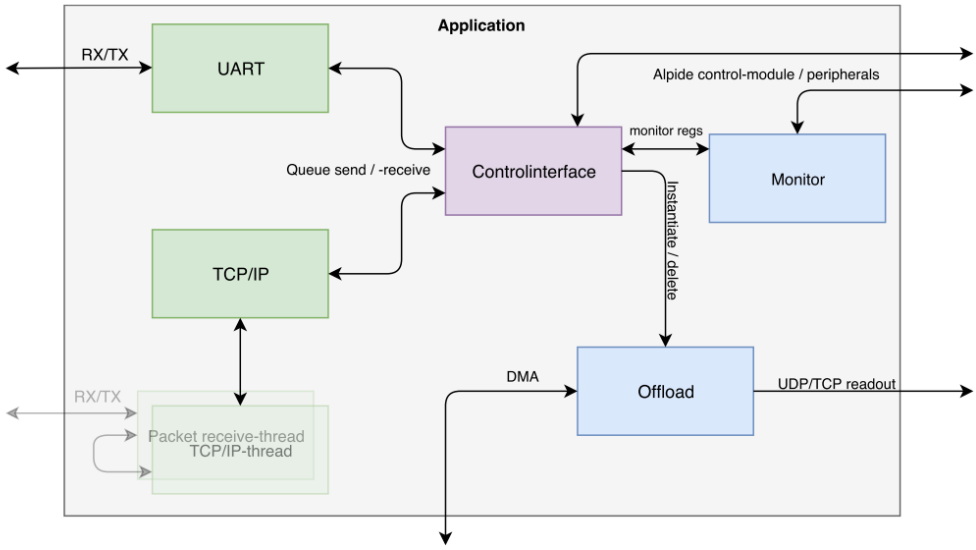


Figure 4.4: Central threads run in the microprocessor [12]

Chapter 5

Host software

The host software is a complete framework to configure, calibrate, monitor and test electrical yield of the ALPIDE chips. The software provides the user abstract classes and functions to integrate their own readout boards and develop new tests. It also contains developed ALPIDE tests that can be used by other readout systems with little alteration. In chapter 4 the pCT readout unit and control systems firmware and embedded communication and message protocols were discussed. This chapter will show how the host-side of the complete system functions and interface with the PRU.

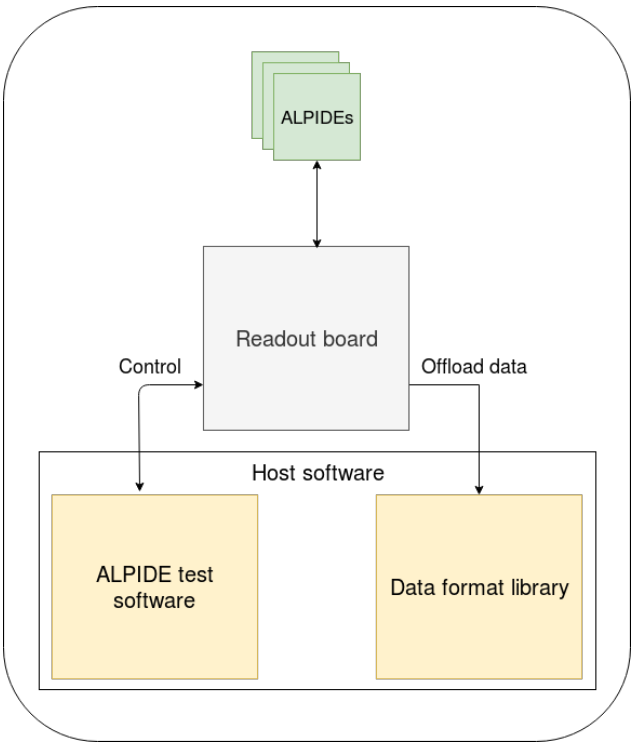


Figure 5.1: Overview of the main components in the host software

5.1 Overview

The host software consists of two components; The ALPIDE test software and the Data format library. The ALPIDE test software controls the communication with the PRU through readout board objects that contain the transport layer TCP/IP and protocols of higher abstraction to interface with the PRU. The software also provides objects for the ALPIDEs, scan settings and a configuration system to calibrate and configure pixels detectors and readout boards.

The Data format library has been designed from scratch. It manages readout-data arriving from the PRU by decoding PRU words, building events, decoding raw detector data, checking for errors and storing information on disk. The information is stored in the ROOT format which facilitates analysis by giving the ability to use the tools provided by the ROOT framework. It also contains a logger to debug the process of decoding PRU and ALPIDE data. The logger is configurable with verbosity levels, severity levels and the ability to report on specified events. A simplified block diagram of the two components can be seen in figure 5.1

5.1.1 Development Principles and Tools

The ALPIDE test software and Data format library were developed in two different repositories. They were segregated because they use different design patterns and styles. The test software is written in a mix of object-oriented and imperative C-style while the Data format library is written as an object-oriented application. When developing the software, the focus was on getting the system to work as soon as possible. Therefore an ad hoc approach was chosen, but with modularity in mind. Besides a fast development methodology, the code is documented using special syntax. The documentation can be parsed, and a manual can be compiled using Doxygen[15]. The software uses CMake[16] to set compiler flags, import libraries, and generate makefiles. At last, both software components make use of the C++ Boost library[17] for various tasks. This software library provides support for a wide range of applications with structures and procedures for tasks such as networking, multithreading, and image processing.

5.2 Software Structure

The tests developed by the host software use two threads for initiating readout and reception of readout-data generated from the readout system. These threads configure the readout board and ALPIDE, start the readout process and parse offload data into PRU events that are stored on disk. Readout-data containing errors are discarded, while remaining events are stored on disk.

All communication is done via TCP/IP. A writing operation is verified by instant read-back. The offload thread is terminated when a time limit is reached, or a pre-calculated number of events are received on the host-side.

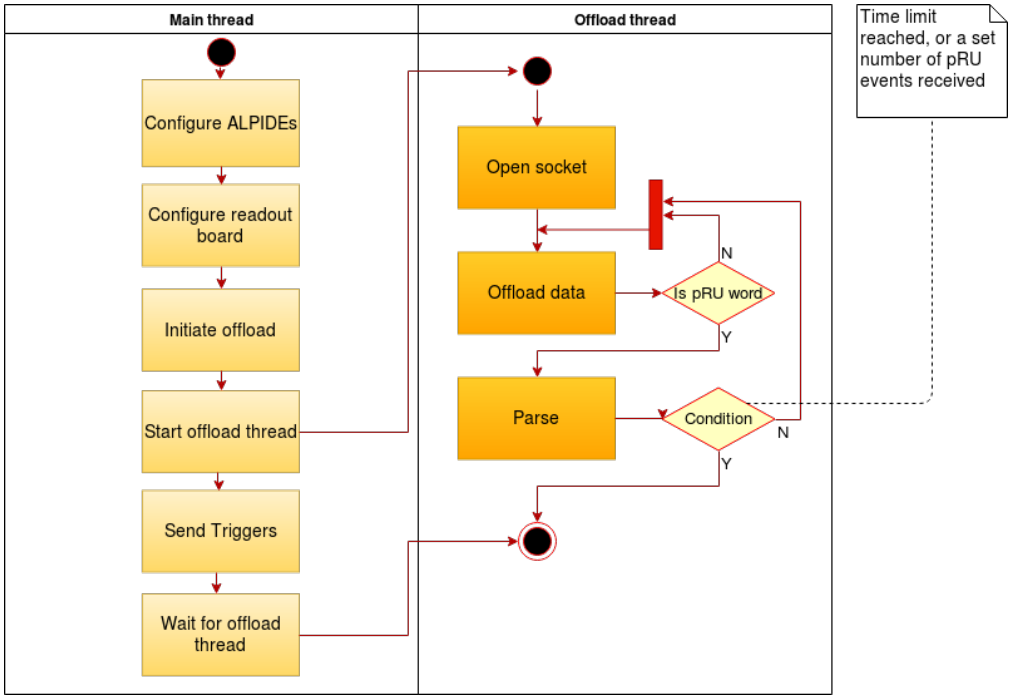


Figure 5.2: Test software application - Simplified flow chart of the two threads

The main thread will get a reference to the offload thread and "block itself" by calling the *join()* function on the offload thread. This function will block the calling thread until the other thread object has finished.

5.2.1 Data Readout

The data readout process starts by sending a special packet to the embedded software on the PRU. The embedded software will spawn an offload thread that will try to connect to the host software. On success, the host application will continue its process by resetting the BC counter of all ALPIDEs through the trigger manager firmware module on the PRU. As mentioned in section 3.1.5, the BC counter is used to synchronize across detector chips and provide a timestamp for ALPIDE frames which may be used to assist in analysis of readout-data. The host application then proceeds to set different trigger parameters such as the number of triggers and delay between each trigger. The trigger module also allows for *trigger trains*, which are a specific amount of triggers grouped together. The delay between trains may also be altered. To ensure that other configurations have time to complete, a pre command delay can be set to delay the PRU of send-

ing triggers to the ALPIDEs. In the end, a final pulse is sent to initiate triggering from the PRU.

Data offloading is handled in the Data format library. A connection is established with the readout board via TCP as a server waiting for an incoming connection. The readout board will spawn an offload-thread when it receives a special packet from the host system and begins transmitting data, as mentioned in the previous paragraph. The host software then creates a text file which is used to dump raw data received from the PRU. This file ensures that if the application is terminated preemptively, the offload data is stored and could be parsed and analyzed at a later point. The process continues by attempting to read 16 bytes from the socket, which is the exact length of a PRU word, effectively receiving one PRU word per read. It would be simple to configure the readout task to receive multiple data words per read as an option to increase the speed of the application. It could also reduce the probability of buffer overflows in the readout unit by offloading data faster. The computer running the host software has an enormous amount of space compared to the PRU, so freeing data buffers on the PRU should be a priority. If the incoming word contains all ones (0xFFFFFFFF) then it classified as a flush word. A flush word is a delimiter word indicating that the PRU offload stage needs to be flushed. The host software does not act on this information, and goes back and try to read more data. Before sending the data to the parser or storing on disk, the byte order of the offload data is reversed from little endian to big endian. The offload sequence will terminate when a time limit is reached, or a set number of PRU events has arrived. Figure 5.3 shows this process.

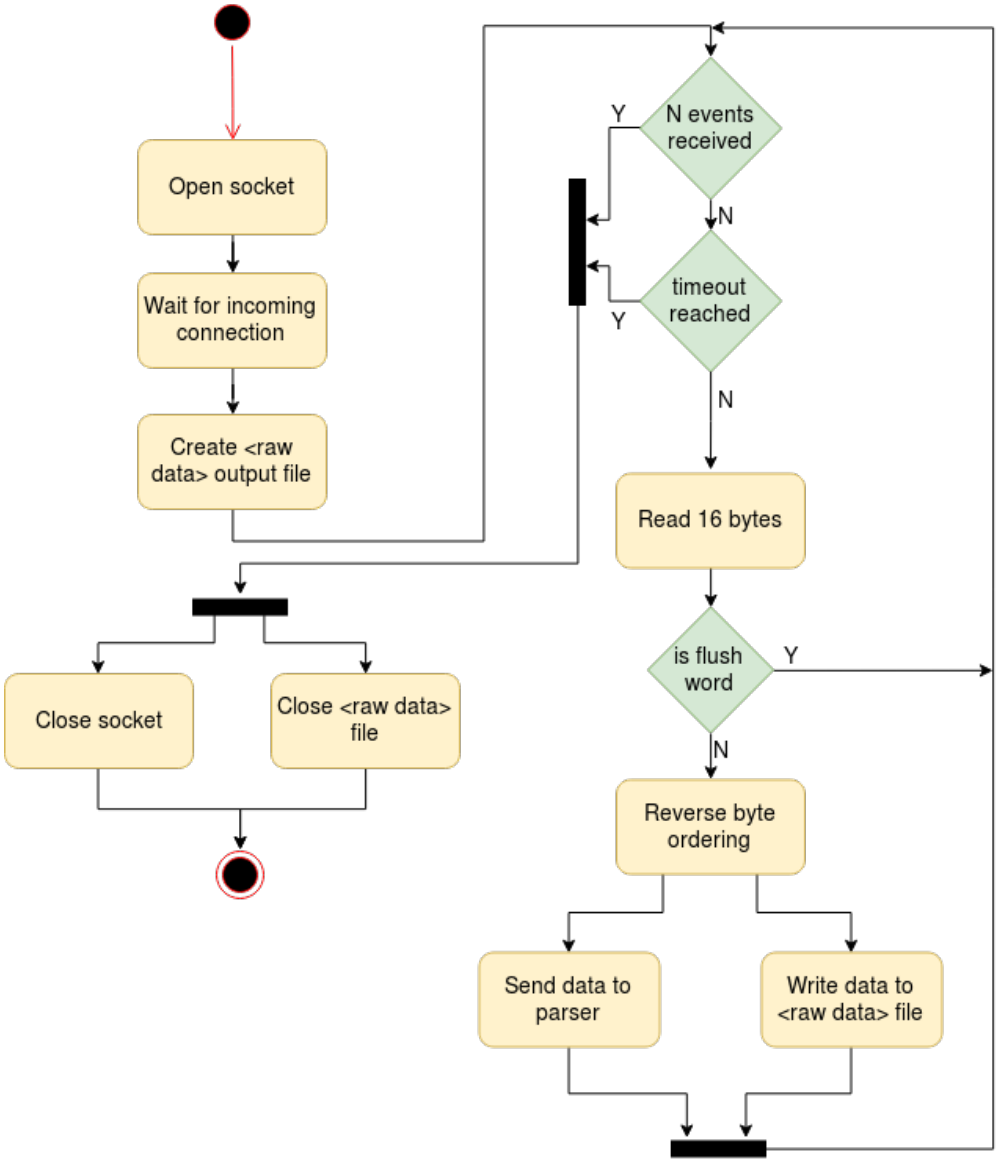


Figure 5.3: Offload-task flow chart

This task is from a technical view done within a functor[1]. A functor is an object acting as a function or function pointer. In C++ the object has overloaded the parenthesis operator (also called function operator). A functor was chosen to comply with the Boost library standard for letting the TCP run on its own thread.

[1]Functors are defined in mathematics as maps between categories and are entirely different from what is presented here. Functors in different programming languages are universally the same.

5.2.2 Parsing PRU Data

RU words received from the readout system are translated from a sequence of raw bytes to a sequence of characters representing each byte. The prefix "0x" is prepended to the string to emphasize that the string is hexadecimal. The historical reason for this conversion is that data was previously read from a text file. This string is sent to the parser for decoding and check of erroneous data.

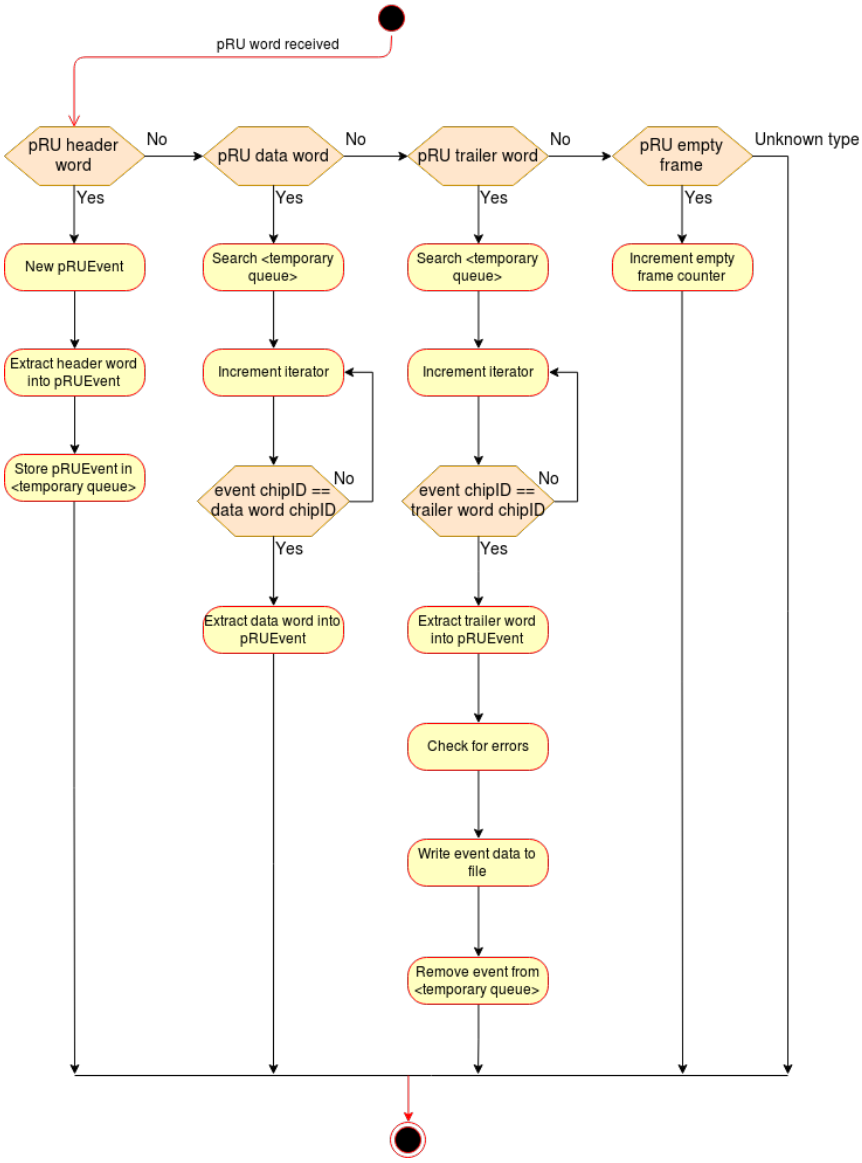


Figure 5.4: Flow chart of the parsing process

A PRU event consists of a header word, an arbitrary number of data words and a trailer word in that order. The main functionality of the parser is to group the words into an event and write information such as hits, errors and other data describing the event to file. The parsing pro-

cess is captured in Figure 5.4. When a PRU word arrives it is identified as a header word, data word, trailer word or empty frame. Unknown types are discarded and reported immediately. An arriving header word will create a new PRUEvent object and each field of the header word will be extracted into the newly created object. It will then be pushed into a temporary queue. When a data word arrives, the parser will try to match the word with an event in the temporary queue. They are matched through the chip ID which resides within the data word and event object. The data word is then extracted into the event. The trailer word completes the event. The parser will again try to match the word with an event via chip ID. Given a match, the trailer word is extracted into the event. The event is then checked for various errors. The trailer word itself contains error flags set by the readout system which are reported. It also contains a field saying how many bytes of data that lies within the PRU data words. The bytes are counted and compared to that field. The event is not discarded if an error is found. It is only reported. The event is then written to file. How and what is written to file is described in section 5.3. At the end, the event is removed from the temporary queue.

As mentioned above, the only way to associate PRU words to PRU events is through chip ID. This is prone to cause errors. It might happen that a trailer word does not reach the parser, and then a trailer word from a different event is inserted into the wrong event. An alternative way of connecting PRU words and events is described in section 7.4.1.

5.2.3 Tests structure

All tests (scans) are developed as individual applications and compiled into binaries which are executed sequentially. The source file of each test has to follow the specific naming convention *main_*.cpp*[2]. One example is *main_digitalscan.cpp* which examines the digital part of each pixel in the ALPIDE pixel matrix. All source files with this convention will be treated as an application and can be compiled with the Unix Make utility. To compile a test, the *make ** command is performed. Again an example would be *make digitalscan*. See Appendix D for more information on how to build and compile tests.

The source code of each test varies, but they follow the same pattern. It has a main function with global functions and variables. The readout boards and ALPIDE chips are global variables which are altered by the global functions. The scan itself is also a function called by the main function. This approach allows for quick changes, but the source code of the tests may become very large.

[2]Kleene star (*) means any Unicode sequence

5.2.4 Initialization of ALPIDE and Readout Board

The ALPIDE chips and readout boards are initialized with a set of classes, helper functions, and a configuration file. The software uses a configuration file to initialize the configuration objects for the readout board, ALPIDE, and tests. The configuration objects are then used to retrieve register values which are then written to the ALPIDE and readout board.

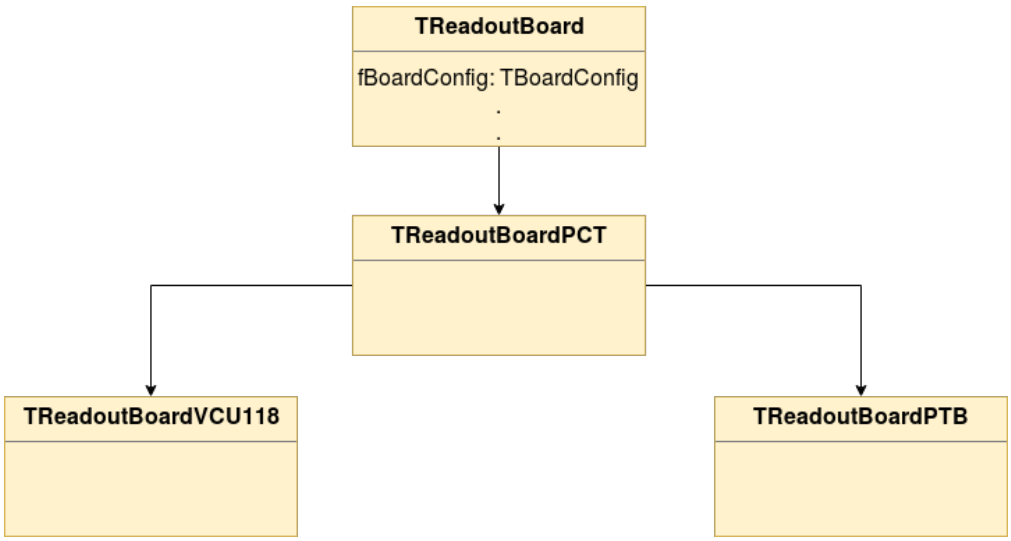


Figure 5.5: Hierarchy of readout boards

Figure 5.5 shows that each readout board contains a pointer to a board configuration object. A dynamic cast is performed to convert the pointer to the correct class. The readout board class for VCU118 should for instance point to a configuration file for VCU118. The ALPIDE object also contains a pointer to an ALPIDE configuration object.

The configuration process starts with a global configuration object, TConfig, as illustrated in figure 5.6. It parses the configuration file and creates base class configuration objects for all readout boards. The config file can contain multiple boards with their settings and has a field to activate them. This activation field allows board settings to be stored and used at a later point. When the test application has received the board types, it can create a specific board configuration object. The newly created object will fetch its values from TConfig by doing a dynamic cast and copy the values to itself. The application has now a config object for its intended readout board and can get the IP and port to connect to the RU. A readout board object is then created and it will try to establish a connection. A simple write verifies the connection by reading the value back and compare.

The configuration of ALPIDEs has a similar approach, but dynamic casting is not needed be-

cause it only exists one class for configuration ALPIDEs.

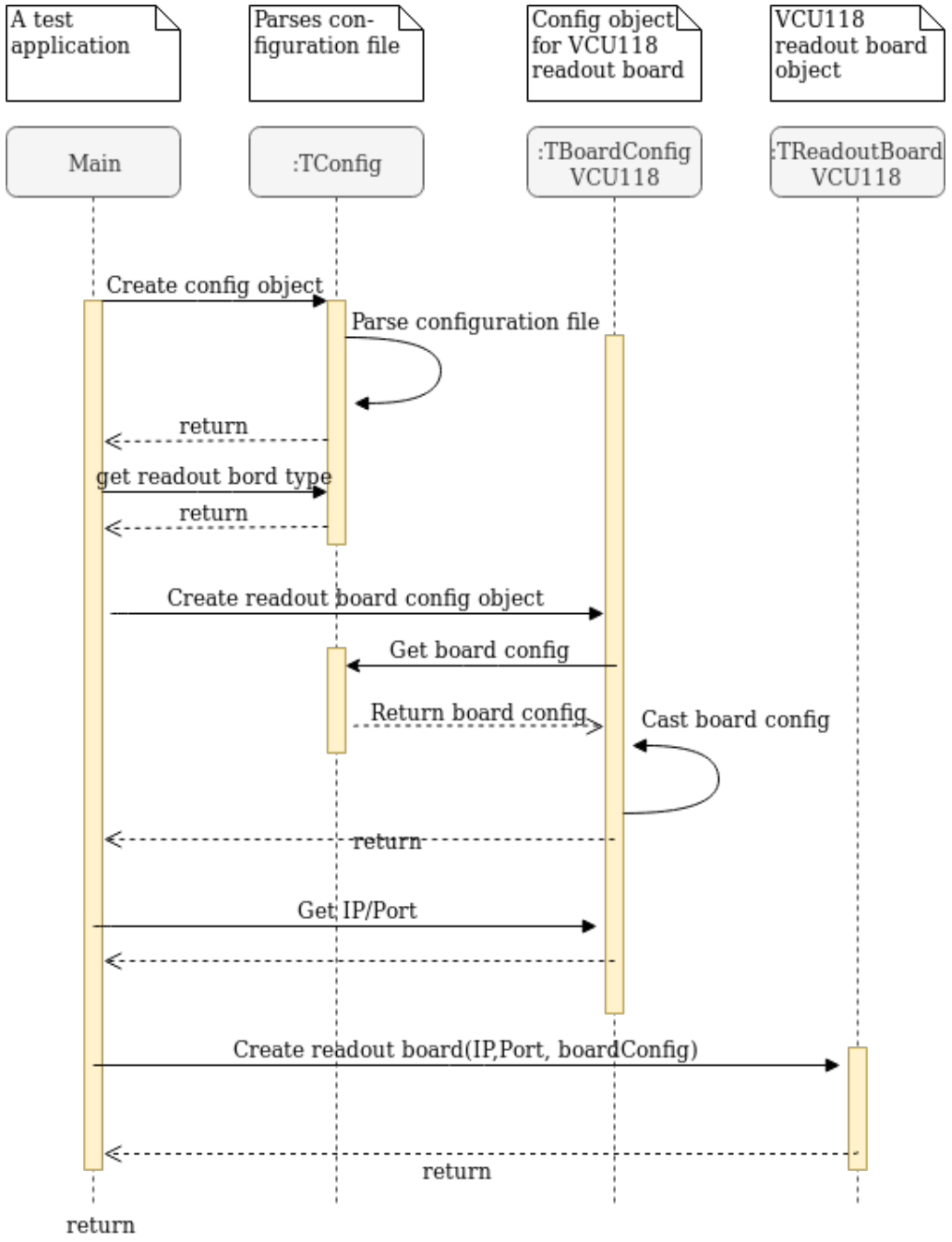


Figure 5.6: Process of setting up the VCU118 readout board with its corresponding configuration settings

5.3 Storing Data

PRU events are stored in a special format and saved on disk. The ROOT file format was chosen to distribute data to other researchers for further analysis efficiently. The encapsulated data is simple to read and manage in the well-known ROOT environment. ROOT also offers efficient compression algorithms to reduce file size. Besides being well-known it gives the user a visualization tool to facilitate analysis of data in multiple dimensions.

5.3.1 ROOT

ROOT is a framework produced at CERN for storage and analysis of large data sets [18]. The framework has a machine-independent binary format which is compressed using a vertical data storage technique. ROOT provides a rich set of statistical, mathematical, and algorithmic functions such as integration and minimization for analysis. It also implements histograms to do graphical analysis on data sets in one or multiple dimensions. The results can be stored in a range of formats such as JPEG, GIF or PDF.

5.3.2 TTree

The TTree is a data structure used to store information in memory and disk. Every tree branch has a modifiable amount of memory with 32 MB as default. Branches can be variables, objects or a CloneArray containing objects of the same type. It is important to note that ROOT requires the user to generate an individual cxx file and a library file for new classes and structs. This is done by supplying a link definition of the class or struct to ROOT.

5.3.3 Event_t

Event_t is a C++ class with only public member variables as shown in appendix A. It contains information about one event such as hit data, error flags, chip id and a time reference. The hit data is decoded region, address and double column from the ALPIDE words region, data short and data long. It is a pair of x and y giving a hit in the ALPIDE pixel matrix. The member variables in the structure have recognizable names to facilitate data extraction and analysis by people with knowledge of the basic principles of operation of the ALPIDE.

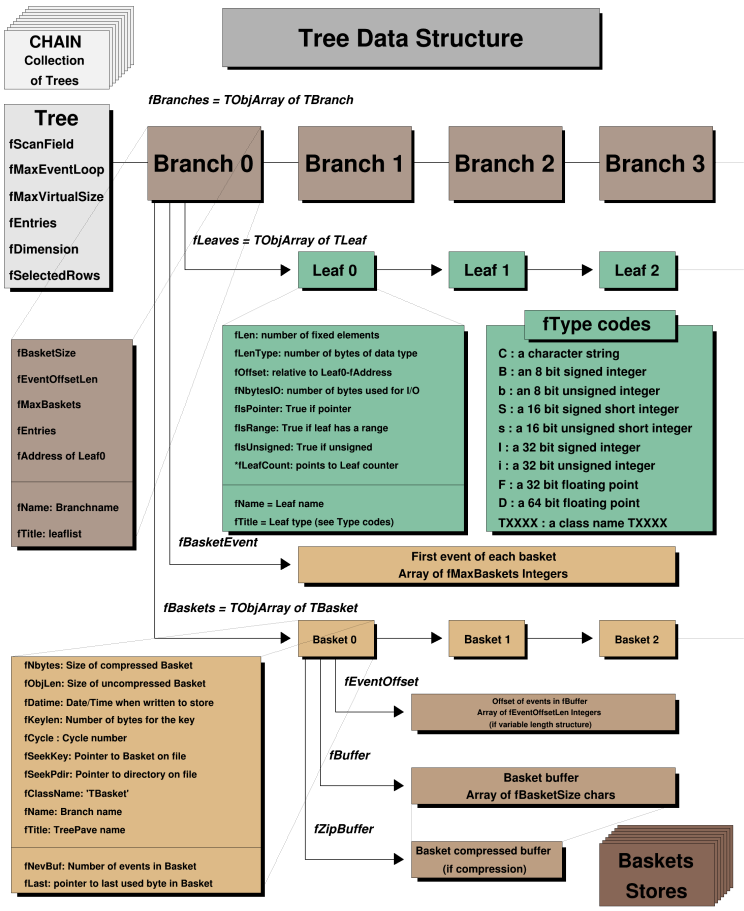


Figure 5.7: TTree data structure

5.3.4 Saving PRU events to Disk

PRU events with at least one header word, data word and trailer word are immediately extracted into an Event_t object. The new event structure is then added as a branch to a TTree object. The tree then fills its branch baskets with the newly added data. This happens in memory and will quickly consume a considerable amount of RAM. ROOT offers the ability to compress its objects in memory, but this reduces the execution speed of the program. Instead of compressing, the tree is written to file and deleted from memory after 1000 events. A new TTree object is then constructed and repeats the process. It is possible to write to file when the tree object has a certain amount of bytes instead of a specific number of events. In case of many events, a lot of files will be written to disk. They can easily be concatenated under or after the program has finished.

Each branch name includes event id from the PRU header word, thus making it clear that each branch representing precisely one PRU event. It is worth mentioning that the event ID field has a depth of 32 bits. For long runs that generate more than 4294967295 PRU events, the event

ID counter will wrap around and start from zero. This will annihilate the uniqueness of the ID given to each PRU event. This might not be a problem if other resources are used to differentiate between events such as timestamps.

5.4 Decoding ALPIDE data

Each PRU data word contains 14 bytes of raw ALPIDE data. This data is decoded by a set of static functions that extracts frame information. Decoding is a straightforward bit-fiddling process. The use of bitwise operators can acquire all fields, such as bunch counter and chip ID from the ALPIDE data word. The words and its contents are explained in section 3.1.5.

The process of extracting hits from the ALPIDE data words is a bit more sophisticated. Section 3.1.2 explains how the priority encoder indexes pixels. This section will demonstrate how the hits are obtained after the software receives them.

A hit is a pair (x,y) of column and row in the pixel matrix. From figure 3.3 in section 3.1.2 it is trivial that to find the row, it is only necessary to divide the address by two. The column is found by region and double column. By using the address, it can be determined if it is the left or right column in the double column.

Table 5.1 shows an example of a frame containing one hit. It has a header, region, data short and trailer word. To get the pixel position the region is acquired from the REGION HEADER. Next, the priority encoder and address are obtained from the DATA SHORT. From the example we get that region = 2, priority encoder = 8 and address = 21. With the functions in listing 5.1 we calculate the row and column and get the hit (81,10).

Table 5.1: Example sequence of ALPIDE data giving one hit in the pixel matrix

ALPIDE DATA WORDS	CHIP HEADER	REGION HEADER	DATA SHORT	CHIP TRAILER
Data	1010 0001 00000000	110 00010	01 1000 0000010101	1011 0000

Listing 5.1: Two functions to get column and row

```
1 int AddressToColumn(int ARegion, int ADoubleCol, int AAddress) {
2     int Column     = ARegion * 32 + ADoubleCol * 2;     // Double column
        before column
3     int LeftRight;
4     if (AAddress % 4 == 0 || AAddress % 4 == 3)
5       LeftRight = 0;
6     else
```

```
7         LeftRight = 1;
8       Column += LeftRight;
9       return Column;
10  }
11  int AddressToRow (int AAddress)
12  {
13      return AAddress / 2;
14  }
```

5.5 Staves

In chapter 3, the design of the pCT prototype was explained. The machine has pixel detectors mounted on staves that contains nine chips. ALPIDEs mounted on staves are identified by a jointly injective pair of identification numbers; chip ID and stave ID. The software reflects this configuration by supplying a TStave class and the associated configuration class TStaveConfig. The pair of classes provide the ability to execute commands on all chips. At this time the class does not take advantage of the broadcast commands implemented in the ALPIDE chips, but loops through the chips connected to the stave and transmit the same command.

Currently, the applications developed with the host software focus on individual pixel detectors, not in stave configuration. In the future, all chips will be mounted on staves, and the tests have to be altered to use stave objects to access the ALPIDE objects.

5.6 PRU Logger

A logger is implemented in the data format library to provide feedback on the parsing process of PRU words. The logger is defined as a single instance in a header file and distributed across the application. The object operates as a variadic function taking any argument compliant with the C++ standard library stringstream class. It may also take PRUEvent objects. The logger fulfills the mutual exclusion requirement and succeeds in a multi-threaded environment. Besides being a function of indefinite arity, the class also takes two template arguments; Severity type and verbosity level.

Severity type allows the user to set a message to either DEBUG, WARNING or ERROR. Verbosity

level gives the user the ability to set the verbosity level of the message to either LOW, HIGH or ALL. A threshold in a configuration file can be set to control which messages should be printed. Messages can be streamed to both console and a text file. The configuration file has two additional notable options. The first options is the ability to print a detailed description of a specified event. This is useful when parsing the data for a second time by getting a comprehensive record of the event in interest. The logger provides a continuous report of the parsing process. The second option lets the user set how often the report should be printed after a particular number of parsed events.

5.7 Configuration System

The configuration subsystem manages ALPIDEs, readout boards and scan settings with a configuration file and several classes. The file holds all individual and general ALPIDE, readout board, and scan settings. The subsystem had to be redeveloped to fulfill the following requirements:

- Store noisy pixels and mask them automatically in the configuration process
- Store broken ALPIDE memory regions and disable them automatically in the configuration process
- Store individual information about ALPIDEs and readout boards
- Facilitate the initialization process of ALPIDEs and readout boards
- Force user to place settings under sections making the configuration file more transparent

Appendix C shows an excerpt from the configuration file. It contains sections where each section has an arbitrary number of pairs that holds a key and a value. The key is represented on the lefthand side of the equal sign, and the value of the righthand side. The subsystem parses the configuration file and stores the section headers and pairs in a structure called a property tree[3]. Each section becomes a node, and all pairs are grouped into an ordered list of subnodes of its section node. The values can be retrieved by referring to the section header and the key as shown in listing 5.2. The get-function also has a template parameter <T> giving the returned value in this case type <bool>.

Listing 5.2: Retrieving values from the property tree

```
1 if(fIniSettings.get<bool>("board_vcu118.ENABLE")) {
```

[3]Both parsing and the property tree structure is part of the boost library

```
2       Init(TBoardType::boardVCU118);
3   }
```

For each enabled ALPIDE chip in the configuration file, the subsystem creates a TChipConfig object which is pushed to a vector. This object holds information such as noisy pixels, defect memory regions, and DAC settings for one ALPIDE. In the initialization process, the size of this vector is used to determine the number of TAlpide objects to create. The configuration objects are then pointed to by newly created ALPIDE objects.

5.8 Build System

To manage the source files, required libraries, and compilation of code, the build system CMake has been used for both the ALPIDE test software and data format library. CMake is a cross-platform toolset used for managing, testing, and packaging for software. CMake reads files called CMakeLists.txt which is written in its own language. CMake requires a CMakeLists.txt file at the top level of the project but can have an arbitrary number of such files downwards in the project hierarchy. The build system will generate makefiles which can be used to build the project as an application or library. From figure 5.8 it can be seen that each directory with source files has a CMakeLists.txt and one placed at the top. The top file typically contains compiler flags and version requirements, output directory and finding library paths and versions. The other is for linking source files and adding include directories (.h and .hpp files), except for the directory with the source file containing the entry point function of any C++ program; *main()*. This file will set the name of the executable. CMake does not download or direct the user to the required libraries as opposed to the build system Maven used in Java projects. All requirements have to be to acquired manually by the user. Besides creating makefiles with commands for building the project, it can also add custom commands such as cleaning project and automatic documentation generation.

Both repositories have robust "CMake scripts" which facilitates further development in terms of adding new sources and libraries. Source files added to the current directories in the data format library are linked automatically, and no modification of the CMakeLists.txt files are necessary.[4]

As mentioned the build system generates Makefiles with commands to build the project. The GNU tool Make is used to control the generation of executables with the makefiles.

[4] CMake allows adding source files automatically by searching for files recursively in the project hierarchy. This can lead to errors if new sources are added and the software is not rebuilt since CMake has no way of knowing if a new source file is added

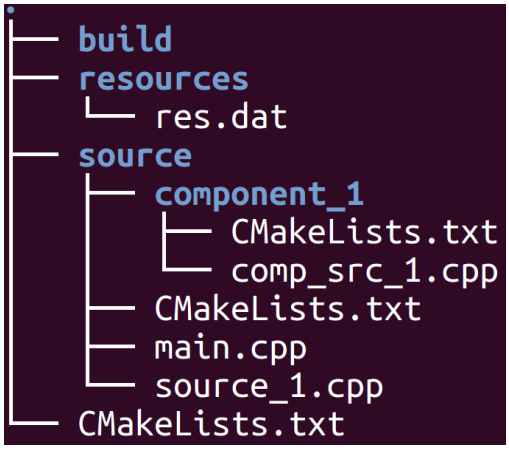


Figure 5.8: Simple example of a project structure with. Blue text indicates that is a folder, and white a file. CMakeLists.txt is put at the top of project and in each folder containing source files.

In regards to acquiring and compiling required packages (i.e. libraries) for the software, Conan.io is an alternative. It can be used to retrieve and build required packages automatically by applying a client-server architecture.

5.9 Future Development

A graphical user interface (GUI) will enhance the efficiency and ease of use for the underlying software functionality for users without in-depth knowledge of the host software. A GUI provides a visual interface to the underlying logical design of the program. It gives the user tools such as windows, buttons, and menus presented as logical symbols. The GUI will increase the usability by letting the user configure, calibrate and test ALPIDE and readout boards with a simpler interface.

There exist several frameworks to build GUI applications. The host software is written in C++ and has been developed and tested in Ubuntu and CentOS. It is highly unlikely that the programming language or development environment will change during the life span of the pCT project. Therefore the GUI framework should be compliant with C++ and Linux. The most prominent framework is Qt. It is cross-platform, written in C++, vast amount of features and it is open-source. Qt also ships a graphical development environment with a drag and drop interface to facilitate the development of GUIs. Another option is wxWidgets which also is a cross-platform library written in C++. It has much of the same functionality as Qt but only pro-

vides an API. Both frameworks are open-source and free of charge, but they differ in licensing. Any changes to the Qt source code has to be published unless a commercial license is bought. wxWidgets does not have this obligation.

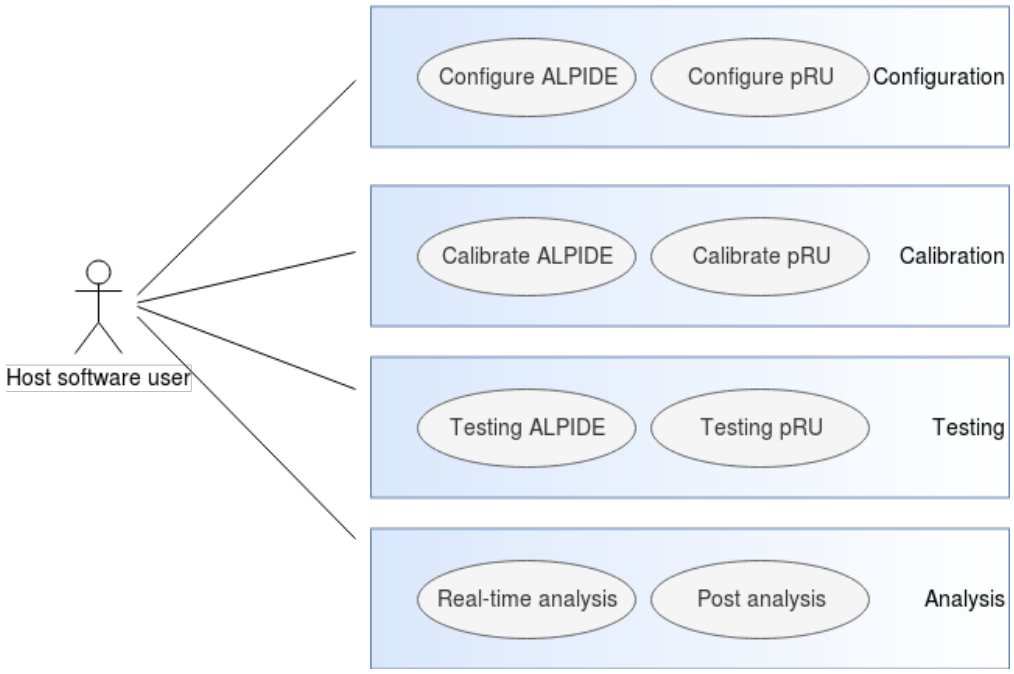


Figure 5.9: Use case diagram showing the most central operations of the host system

Designing a complete GUI lies outside the scope of this thesis, but this section will discuss the most important use cases of the host software and how a graphical interface can increase the ease of use. Figure 5.9 gives an overview of most central services in host software except the real-time analysis which is not implemented. Real-time plotting lets the user react on the incoming data without delay. This allows for an immediate alteration of test parameters instead of running the full test before getting a result. An example could be to track a charged particle through multiple layers with a specific angle of the beam. With real-time plotting, the real angle of the beam can be revealed and corrected almost instantaneously. Real-time analysis can be implemented in the GUI with the selected framework, or third-party software such as ROOT. This subsystem should provide the following abilities:

- Analysis in two and three dimensions
- Implement tools to create custom analysis (i.e. scripting)

This subsystem may fulfill additional roles, and the design should be flexible to accommodate for extensions. Besides designing a proper analysis window, the subsystem should not affect the

application to much in executing speed. Real-time plotting may be an exhausting process and reduce the efficiency of other tasks.

Chapter 6 describes all the individual tests and their classifications. Currently, all tests have to be run individually, and results have to be analyzed afterward. This will become a problem when the number of staves and chips increase. It will get challenging to get an overview of which staves and which chips have been tested. Giving the user a simple interface to select and run tests and display the results of the tests. Figure 5.10 gives an example of how the test results of the ALPIDEs may be grouped and presented. Each stave is classified by each of its nine connected ALPIDE chips. The user may inspect each of the stave's chips by clicking the arrow on the selected stave to get a list of ALPIDEs and their associated classification. Each pixel detector test could then be inspected by clicking the arrow on the wanted ALPIDE, and a list of all test and their classification may be inspected. If needed, another layer of abstraction can be added to group staves into layers to reflect the configuration of the proton CT prototype explained in section 3.2.

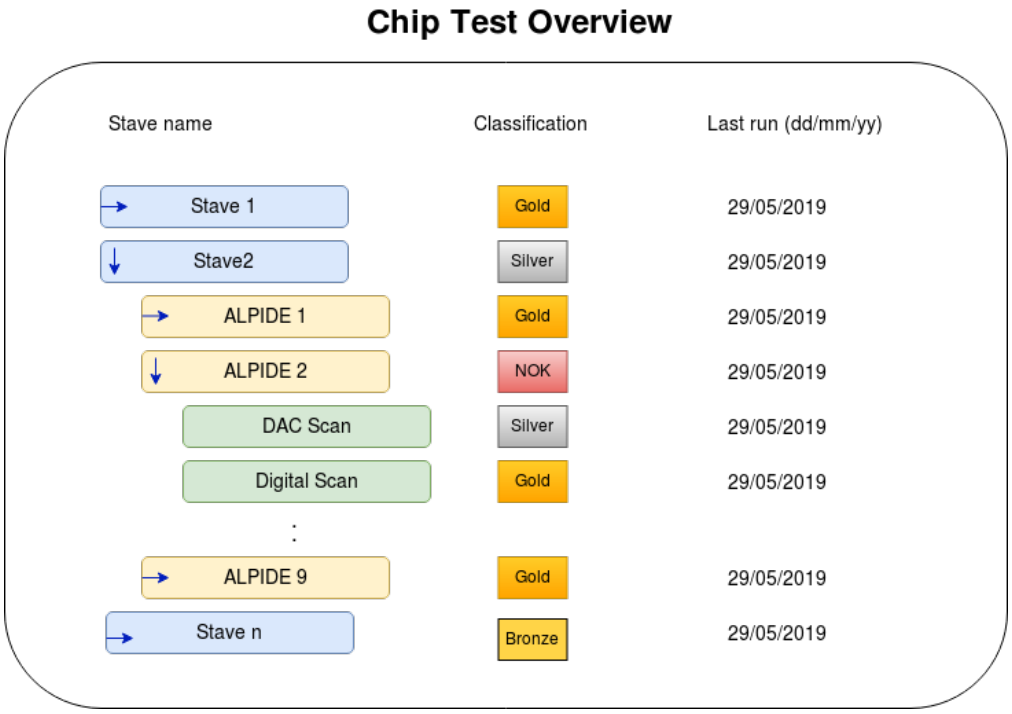


Figure 5.10: A proposal of how test results can be grouped and presented graphically. Arrow down indicates that the component has been selected an should show its attributes

Section 5.2.4 explains how ALPIDEs and readout board are instantiated and configured through classes, functions and a configuration file. Currently, the configuration file holds settings for individual devices but is a tedious and error-prone process to alter the settings manually. A

great addition would be to have menu to set both general and individual readout board settings.

Chapter 6

Testing

6.1 ALPIDE Testing

The production yield of the ALPIDE is verified by a series of tests by the host software. The chip has built-in registers to encapsulate and verify specific modules. Each test has its own set of classifications ranging from gold to not-okay(NOK) [1]. Gold being the desirable outcome. The test procedures was developed at CERN and ported to the new readout system discussed in chapter 4. Each test can be examined in the alpide-sw repository [2].

6.1.1 Digital Scan

The digital scan is performed to test the functionality of the digital part of each pixel. The test checks if it is possible to drive the pixel's state high or low by neglecting the front-end and access the pixel state register directly. The pixel's latch circuitry contains two registers, Mask and Pulse Enable register. With these two masking and pulsing patterns can be generated digitally. By forcing the Mask register high, the output of the pixel is set low regardless of Pulse Enable register value. If low, the output is selected by the Pulse Enable register. With the two registers, an arbitrary number of hits can be digitally injected into a pixel, readout and verified. The test procedure is given below:

[1] The classifications is based on the ALPIDE series mass testing and classifications [19]
[2] https://git.app.uib.no/pct/alpide-sw

1. Enable digital pulsing in FROMU configuration register 1
2. Mask all pixels
3. Unmask row 0 and set Pulse Enable register high for row 0
4. Trigger pixels and readout row 0
5. Mask all pixels
6. Repeat step 3 - 5 for all rows

Table 6.3 describe the four possible outcomes of the test. A malfunctioning pixel is defined as a pixel which outputs high if masked, or low if unmasked and Pulse Enable register is set to high.

Table 6.1: Classification of digital scan

Classification	Gold	Silver	Bronze	NOK
Malfunctioning pixels	< 50	< 2100	< 5243	otherwise

Figure 6.1 displays a full digital scan with 50 digital hits injected per pixel. The red color shows that all 50 hits were injected and read-out successfully for the whole pixel matrix. This particular test was classified as gold as it had 0 malfunctioning pixels.

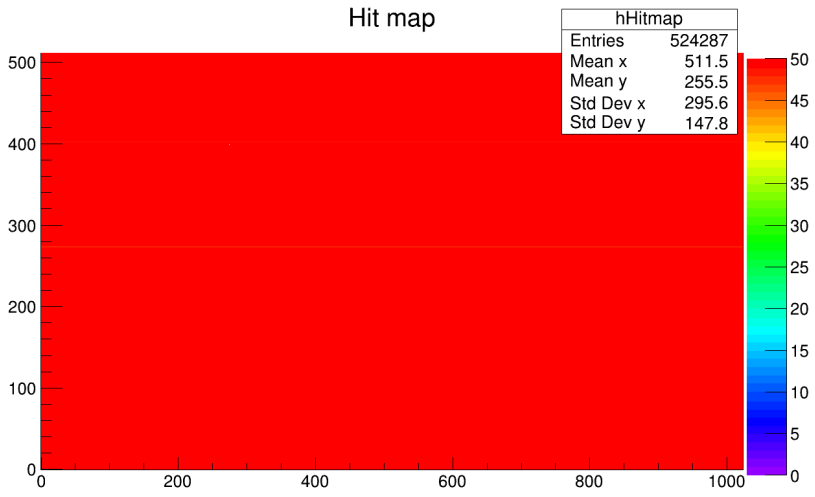


Figure 6.1: Digital scan showing proper functionality

6.1.2 Analogue Scan

Analogue scan tests the front-end (analogue) functionality of each pixel. The test injects two charges with different magnitude into a test capacitor in the pixel's front-end circuitry. The test capacitor drives the input signal of the pixel's analogue part simulating [3] a real hit generated by ionizing radiation. The two charges are set below and above the threshold value to find noisy, dead or normal pixels.

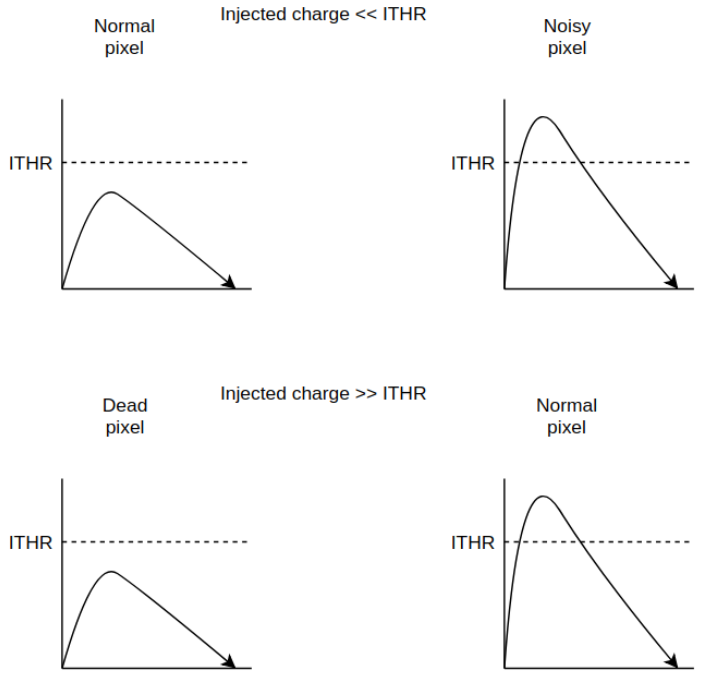


Figure 6.2: Dead and noisy pixel [Courtesy of Viljar Ekland]

Figure 6.2 shows how normal pixels should behave contra noisy or dead pixels. A noisy pixel is identified by injecting a charge well below the threshold value. If the pixel still produces a hit, then it is classified as noisy. To identify dead pixels, the charge is set significantly above the threshold value. If the charge does not produce a hit, it is classified as dead. As previously stated, the test injects two charges. If two hits are produced, the pixel is noisy, and zero hits indicate that is dead. A functional pixel will produce exactly one hit. Besides finding noisy and dead pixels, this scan can also be used to measure noise performance assessed by the fake hit rate. The test has the following procedure:

[3] A collection diode in the pixel's front-end circuitry gathers radiation and creates a charge. The test capacitor provides the ability to set whatever charge of interest.

1. Enable analog pulsing in FROMU configuration register 1 by setting bit 5 high
2. Mask all pixels
3. Unmask row 0 and set PULSE_EN high
4. Inject charge below threshold value
5. Trigger and readout row 0
6. Inject charge above threshold value
7. Trigger and readout row 0
8. Repeat step 2 - 7 for all rows

Table 6.2: Classification of analogue scan

Classification	Gold	Silver	Bronze	NOK
Malfunctioning pixels	< 50	< 2100	< 5243	> 5243

Malfunction pixels are pixel which are classified as either noisy or dead.

6.1.3 Threshold scan

Threshold scan determines the threshold charge value when a pixel registers a hit. Every pixel's threshold value may vary from the global threshold value set to all pixels. By injecting a set of charges using the analogue circuitry of the chip, an S-curve is plotted an measured.

(a) Hit rate given by charge

(b) Probability of firing by charge

Figure 6.3: Response of one pixel to 50 injections with 50 charge steps. (a)

Figure 6.3 shows an example of an S-curve of one pixel. Given a number of injections, each charge-step is plotted against the number of generated hits. By using the number of injections and generated hits, a probability of a pixel firing is given. The threshold value is determined where the probability reaches 50 %. If a 50 % probability does not exist, then the pixel's threshold cannot be determined. The scan has the following procedure:

1. Enable analogue pulsing
2. Mask all pixels
3. Unmask row 0 and set PULSE_EN high
4. Set charge value
5. Inject charge and readout row 0
6. Increment charge value and repeat previous step
7. Repeat step 2 - 6 for all rows

Table 6.3: Classification of digital scan

Classification	Gold	Silver	Bronze	NOK
Pixel with threshold (%)	< 99	< 95	< 80	otherwise

Figure 6.4 shows an example of a threshold scan. The color range is given by each pixel's threshold charge value multiplied by approximately 10 electrons. It is important to note that it is not

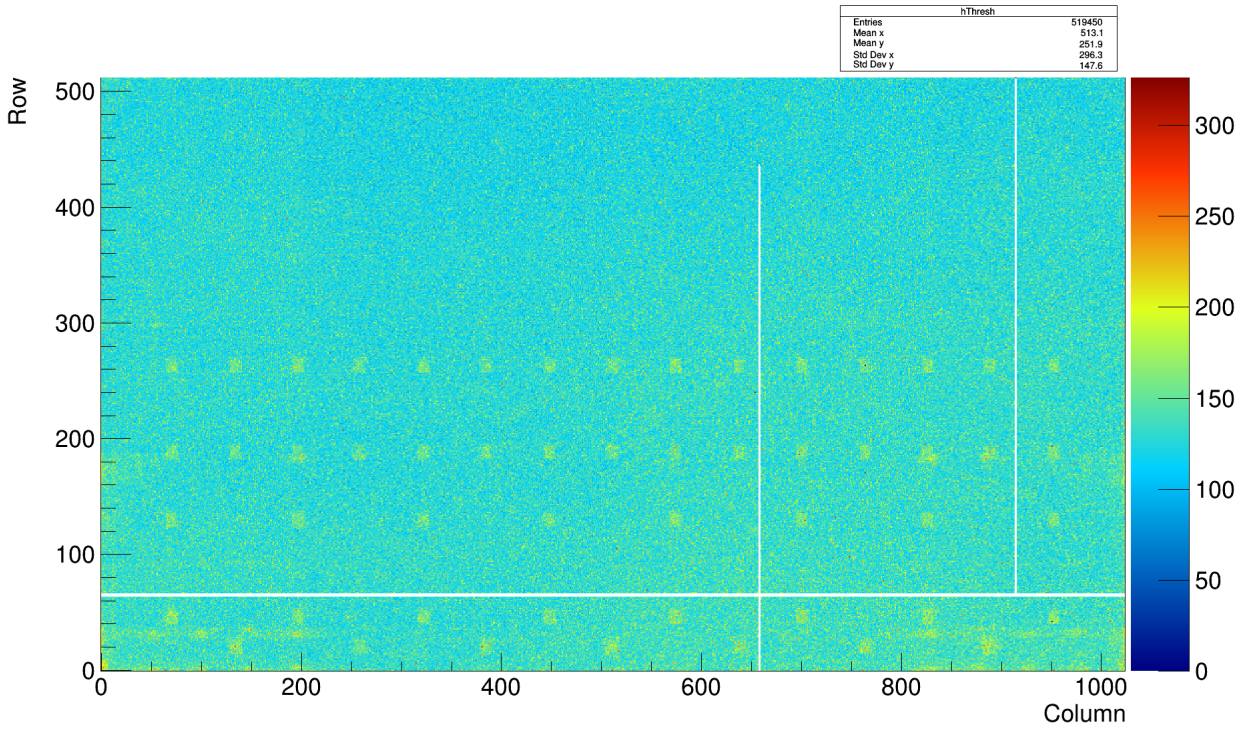
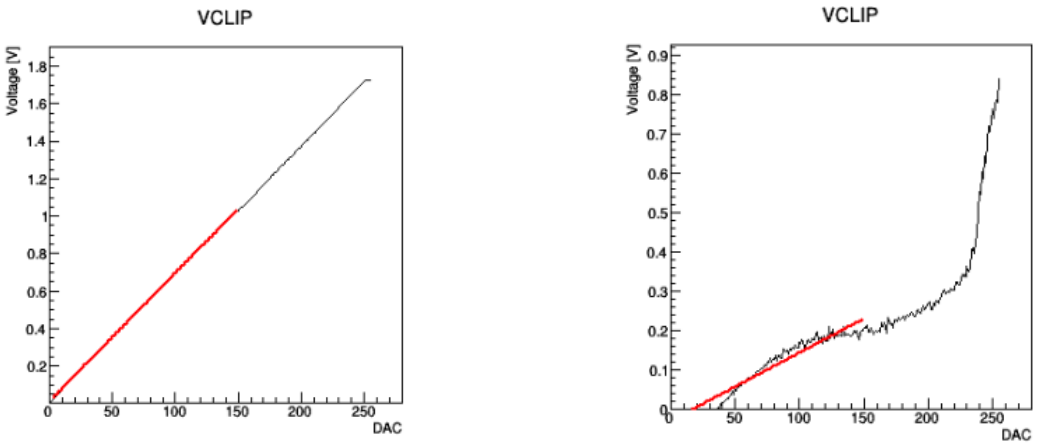


Figure 6.4: Threshold scan revealing the actual threshold of each pixel

possible to set a threshold for individual pixels, but rather a global one for all pixels and then observe each pixel's actual threshold value. In an ideal scenario, the whole image would have the same color, meaning that every pixel would have the same threshold, preferably the global threshold. However, it is expected that every pixel's threshold varies slightly from the global threshold, which also can be seen from the example. The vertical white lines reveal unresponsive columns. The reason for this phenomena is unknown. It might be dead, or that the top pixel is dead and has a rippling effect on the pixels below. The green circles are groups of pixels that lie on top of solder bumps. The solder bumps increase capacitive noise which is manifested by the image.

6.1.4 DAC Scan

The ALPIDE has multiple voltages and current digital-to-analog converters (DACs). The DACs are used to tune the biasing of the pixels front-ends. Each DAC has 8-bit code words ranging from 0 to 255. In this test, each codeword is written iteratively, and an analog value is read back. By plotting codewords against the analog values, a curve can be examined with integral non-linearity (INL). The test also use an additional range parameter to classify the DAC based on the minimum and maximum voltage or current output.



(a) Positive response

(b) Negative response

Figure 6.5: Scan of the VCLIP DAC of two different chips. (a) has an ideal straight curve, while (b) has a high deviation from the ideal straight curve

There are two methods to calculate INL: Best straight-line method, which determines the maximum vertical deviation from the ideal interpolating line. The other method, endpoint fit, is often selected because it gives information about the harmonic distortions of the DACs [20].

The INL can be calculated at any codeword by the summation of the differential nonlinearities (DNL) from the first codeword up to the given codeword. The DNL value is the vertical deviation from the expected analogue value to the actual analog value at a given codeword. The DNL may be a positive value, if the actual analog value is above the expected value, negative if below, or zero if equal. By this definition the following equation is obtained:

$$INL_c = \sum_{i=0}^{c} DNL_i$$

1. Write code word 0 to DAC
2. Read monitoring signal
3. Write code word 1 to DAC
4. Read monitoring signal
5. Write for all code word and read monitor signals
6. Repeat step 1 - 5 for all DACs

Figure 6.5 shows the result of two DAC scan from two different ALPIDEs. The ALPIDE manual does not supply the voltage increase from one codeword to next for each DAC. It only provides

a general equation voltage measurement.[4] By estimating that the transition step is 0.006757 voltage, the following results from table 6.4 are obtained.

Table 6.4: INL and range calculation of VCLIP DAC from figure 6.5

INL	**Range**	**Classification**	**Figure**
0.00852	1.718	Gold	6.5a
169.99	0.843	NOK	6.5b

Based on the the ALPIDE series mass testing and classifications [19], figure 6.5a was classified as Gold, while the 6.5b was classified as NOK based on INL and DAC range.

6.1.5 FIFO Scan

FIFO (memory) scan checks that all 32 memory regions are functional. Each region has 128 addresses with 24-bit depth. The test writes a series of patterns to all the addresses in each region and verify the operation by reading them back. The patterns are used to find stuck bits, shorts between adjacent bits and check the address decoder. The following patterns are written:

- 0x0
- 0x555555
- 0xffffff

The test has three classifications. Gold when no errors, bronze if stuck bits does not corrupt data integrity and NOK if multiple errors.

6.1.6 Register Scan

Register scan checks that all the registers in the ALPIDE are functional. This is done by writing a value to the register and then read it back. The test checks for stuck bits, shorts between bits and randomly chosen patterns for verification. Some of the registers are read-only, or pulsing registers. For the read-only registers, a default value is used to confirm the registers functionality. The following patterns are written and read back:

- 0x0000 and 0xfffff

[4] Full specification of the DAC units can be acquired by reaching out to the ALPIDE development team.

- 0x5555 and 0xaaaaa
- 0xDEAD

The test has two classifications. Gold when no errors and NOK if one or more errors occur.

6.2 Beam Test

In December 2018 a part of the pCT group at UiB, including this author, traveled to Germany with one PRU and six ALPIDE chips. The group had arranged a weekend to use the beam facility at the German cancer research center in Heidelberg. The main objective of the trip was to detect a Bragg peak inside the ALPIDE and observe the performance of the PRU under a real experiment. Previous radiation tests with this setup had been done with low-intensity sources, such as gamma and alpha radiation.

6.2.1 Goal

As mentioned above, the main objective of this experiment was to irradiate the ALPIDE with protons and detect a Bragg peak by analyzing the acquired data. An additional goal was to observe the performance of the readout unit and control software during high data output from the ALPIDE chips. At the UiB two chips and been irradiated simultaneously with a gamma source and data was offloaded successfully, but in this experiment, the chip occupancy would be much higher.

6.2.2 Setup

In this experiment, the mTower setup, PRU, and a laptop were placed in the beam room. The mTower had six mounted ALPIDE chips and was connected to the PRU via a Samtec firefly cable. The chips were laid out horizontally and stacked on top of each other as shown in figure 6.7. The PRU was connected to a computer via an ethernet cable. This laptop was controlled by another computer in the control room via a Secure Shell (SSH) connection. The electronic components in the test room were shielded from scattering particles by placing a Polymethylmethacrylate (PMMA) plastic cube in front of it.

All operations on the PRU and ALPIDE chips were executed from the computer in the control room. A bash script was made to be able to turn off and on the power supply connected to the

PRU. Each test run had different parameters for the PRU and the beam. Table 6.5 shows the settings that were altered during the experiment. A trigger train is simply a specified number of triggers. It enables a way to group triggers and set a time gap between each group. The beam had also different parameters during the experiment. A discussion on the different parameters and their implication on the test results are outside the scope of this thesis but is worth mentioning that the different sources such as carbon, helium, and protons were used. The intensity and energy were also altered during the experiment.

Table 6.5: Scan settings on the PRU

Parameters	Triggers	Trigger trains	Strobe duration

Table 6.6: Scan settings for beam

Parameters	Type	Intensity	Energy	Spills	Focus

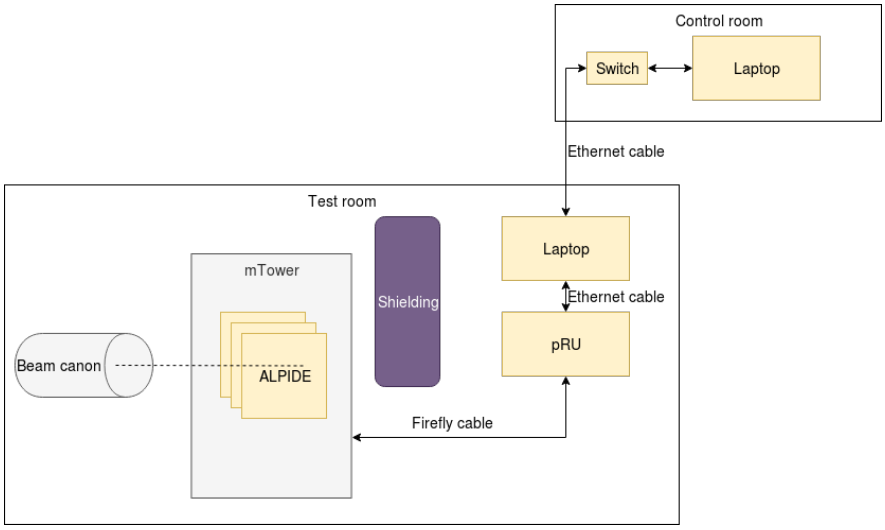


Figure 6.6: Showing the setup during the experiment in Heidelberg

6.2.3 Results and Conclusion

The irradiation tests showed that the readout system and control software works. Data was offloaded, successfully parsed, and presented visually. There were however bugs and occurrences of faults that had not been observed in previous tests or experiments. Trying to operate multiple ALPIDE chips concurrently was not accomplished. One chip would configure and operate accordingly, while the remaining chips would either not configure correctly or, the output data

was not possible to analyze. In one instance an ALPIDE frame seemed to consist of two different frames coming from two different chips. This was most likely due to a bug in the software. Even when limiting the experiment to one chip a considerable amount of data was lost due to buffer overflows in the PRU. It was however expected because of the low readout speed between the PRU and software. Another reason was that the PRU was set to abort the process of forming events if an ALPIDE frame had more than 35535 bytes. With the high intensity of the beam and extended strobe duration, it was likely to create large frames which were aborted during the experiment.

Table 6.7 shows four different scans performed. In all scans helium with the same intensity and energy was used together with the same strobe duration of 250 μs and 500 000 triggers. The table reveals a massive data loss during the scans. Buffer overflow indicates that the PRU had too much data and could not process this before new readout data arrived. Event dropped are aborted PRU events because of full buffers. As a result of the data loss, unexpected and erroneous data arrived in the software. For instance, a data word contained one byte with an ALPIDE trailer word. This yielded strange results when analyzing the data after the scan. As mentioned above, the experiment was a success, but more as a proof of concept. Both the PRU and the software showed insufficiency in both speed and managing unexpected input. The latter is especially aimed at the host software.

Table 6.7: Scan results - Showing information about the scan by the PRU retrived by softare

	Events	Event drop	Protocol error	Decode error	Busies	Buffer overflows
Scan 1	338284	44519	13	83	6	217790599
Scan 2	402290	74360	23	113	1	2097408878
Scan 3	380500	276151	26	148	5	4294967295
Scan 4	380054	280818	55	390	17	4294967295

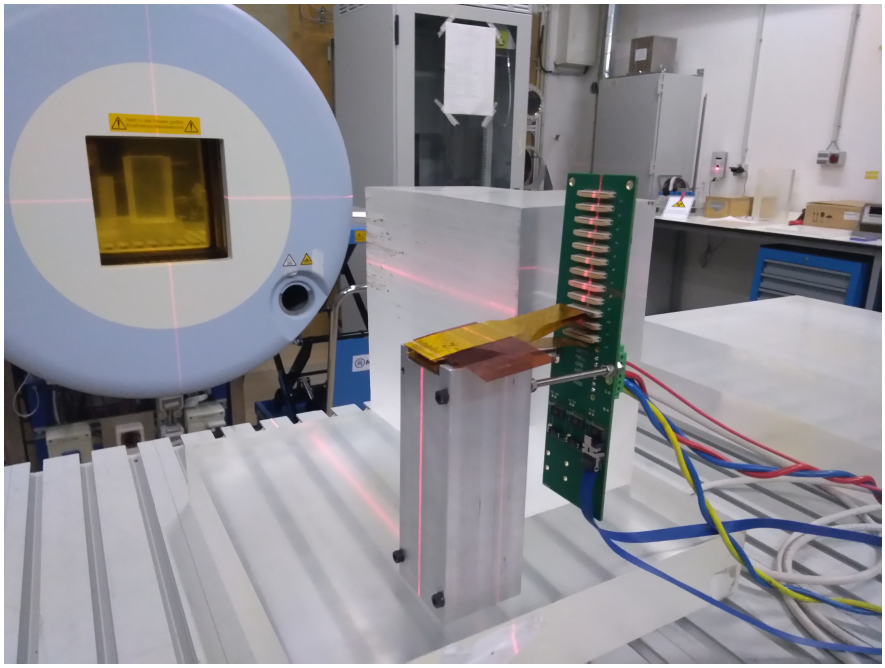


Figure 6.7: Beam test

Chapter 7

Conclusions, Discussion, and Recommendations for Further Work

7.1 Performance Evaluation

The results from chapter 6 shows that the system is working with the firmware and embedded software described in chapter 4. The system has shown that is capable of running over an extended period of time with the longest run being 17 hours. The applications showed no sign of memory leaks or other critical errors.

The software proved adequacy when processing and analyzing the acquired readout-data. However, the software showed its weakness when dealing with erroneous input data. In most cases, the errors were discovered and reported to the user, but the software lacks a robust way of handling such data. In some cases, the output plots yielded strange results ranging from slightly to completely different from what was expected. In the future, a decision on how to deal with certain errors has to be made.

A throughput test was developed to measure the maximum speed between the PRU and the host software. The test filled up the PRU's internal buffers with a sequence of natural numbers and then transmitted them to the host software. The results were measured to 2.3 MB/s which is adequate for testing one or two chips but becomes a problem when multiple staves are going to be tested and operated. The exact reason is unknown, but early conclusion points to a combination of a slow TCP/IP implementation and inefficient algorithmic design in the embedded software. The performance of the host software has not been a prioritized issue during this thesis. The main focus has been to verify the PRU and show that it ALPIDEs can be tested properly

with the new readout system. A test was done showing that the host software could process approximately 0.2 MB/s of PRU data which is well below what the readout unit can supply. There are still several optimizations that can be assessed. One of them is the parsing of readout-data. PRU words received from the readout unit is converted to strings and sent to the parser. The parser decodes the data words and converts the specified fields to primitive types. This is not currently necessary and is an expensive operation when the data sets become large. Another one is the efficiency of storing data on disk and memory. In the future, the data offloading will have much higher throughput and the software will need adjustments to comply with the new speed requirements.

7.2 Design Evaluation

Chapter 5 describes how the host system is built up by two separate components; ALPIDE test software and Data format library. This decision was made early to facilitate the development and testing of the two parts. The ALPIDE software is built as a framework, while the data format library was developed as an application, but diverged into a static library which got implemented into the host software. The result of this is two repositories with slightly different structures, principles, and conventions. Because both components are integral to the complete system and not standalone applications, they should have been developed as one in the same repository to maintain the same structure to simplify further development.

Chapter 5 further describes how the software is a framework for testing and analyzing readout-data from the chip detectors, which results in that each test is compiled into its own application. One of the drawbacks is that each test will generate its own report and on its own format which currently requires individual scripts in ROOT to analyze. A script has to be developed to automate the execution of the different tests. This increases the complexity and number of files to manage and is hard to maintain when a new test is added and the software gets bigger. The pCT project has a specific goal, and is not necessarily in need of a framework, but more an application. By creating an application instead of a framework, newly developed tests can be forced to follow a specific pattern which leads to tests having a universal outline. As of now, the software will need heavy alternation during the lifespan of the pCT project to comply with updated firmware and protocols. Creating a standalone application might be the way to go.

Another challenge is to interface an FPGA design that is under development where the firmware module exposes its registers directly to the host software. A modification of a register-address can break the procedures in the software. A solution is to let the firmware provide an API for

the software via the embedded system, so it does not have to care about changes of register addresses. The host software will, however, need to be updated when new functionality is added on the PRU. A subsystem can be made to identify the firmware version, and inform the user of available features. It can also be used for backward compatibility to use older firmware versions.

7.3 System Development Methodology Evaluation

In section 5.1.1, it was mentioned that an *ad hoc* approach was chosen to control the process of developing the host system. *Ad hoc* is often referred to as a methodology, even if it does not contain any systematic research strategy. Therefore, it is implied that the development process is *ad hoc* if no other methodology is chosen. This methodology is simple and provides a straightforward path from acquiring stakeholders requirements to implementation. It is often adequate for smaller systems with small teams of 1-2 developers. The problems arise if the requirements have not been expressed clearly, or changed during development. The developer then has to choose between starting over again or implement "hacks", which will reduce the cleanness of the code.

There exist several development frameworks. The agile methods assist developers to respond to unpredictable changes in requirements. A popular branch of agile methods is called Scrum. It is designed for teams consisting of three to nine developers. It breaks down work into tasks where each task is assigned a time window called sprints. The sprints are no longer than one month. After each sprint, the new functionality is presented and discussed with the stakeholders. The main difference between ad hoc and Scrum is the ability to react to unpredicted events. While ad hoc is more of a straight line, Scrum provides an incremental and iterative approach.

In this particular project, the ad hoc accommodated for quick updates and made it possible to experiment with multiple solutions rapidly. It did, however, induce challenges along the way, such as updating the software to comply with modifications in firmware modules, formats, and protocols. Implementing a solid methodology is time-consuming, and might have reduced the output of this thesis. On the other hand, it would have increased the lifetime of the host software during the life span of the pCT project by being more robust.

It is trivial to understand that software for medical equipment needs to be extremely robust. An appropriate development process has to be implemented together with other tools for managing the software. First-order predicate logic is often used to verify and validate important

software routines to ensures the highest quality. In the future, all aspects surrounding the development has to be examined.

7.4 Recommendations for Further Work

7.4.1 Add Event ID To PRU Trailer

Section 5.2.2 shows that the only way to connect PRU words to the correct event is through the chip ID field which resides in all PRU words. If a header word arrives twice from the same chip before a trailer word, then the trailer word is automatically connected to the event with the first header word. This might be incorrect and can corrupt data integrity. A solution is to add the same event id field which resides in the header word to the trailer word. This field will work as a unique identifier for each event instead of the chip ID field. The trailer word has 72 unused bits. A proposal is shown in table 7.1 adding the event ID field.

Table 7.1: Proposed trailer word

Name	WORD_TYPE	RU	STAVE	CHIPID	EVENT_ID	UNUSED	ERROR_FLAGS	EVENT_SIZE
Length	2	6	4	4	32	40	8	32
Bits	127:126	125:120	119:116	115:112	111:80	79:40	39:32	31:0
Value	0x2				0x0			

7.4.2 Reduce Number of Empty Frames

Empty frames are 128 bits and may come in large portions. One way to reduce the data load could be to reduce the transmissions of them. A solution is to create a counter in the firmware and only send an empty frame after an arbitrary number of frames. This way the data load is reduced by a proportional amount and the information about the number of empty frames are still kept.

Table 7.2: Proposed empty frame

Name	WORD_TYPE	RU	STAVE	CHIPID	RESERVED	COUNTER	UNUSED	BUNCH_CNT	SPILL_ID	TRIG_SOURCE	MODE	EVENT_ID	ABS_TIME
Length	2	6	4	4	1	10	10	8	16	2	1	32	32
Bits	127:126	125:120	119:116	115:112	111	110:101	100:91	90:83	82:67	66:65	64	63:32	31:0
Value	0x3						0x0						120 MHz Clock

Given a 10-bit counter the theoretical reduction of data is 128*1023 bits per empty frame. With an arbitrary counter depth (n), the following equation is obtained:

$$Reduction_n = 128 * \sum_{i=0}^{n-1} 1_i * 2^i, n \in \mathbb{N}$$

7.4.3 Tuning Data Compression

Section 5.3.4 mentions that the TTree data structure is compressed both in memory and on disk in real time. The ROOT framework provides the ability to set a compression level ranging from 0 to 100 of its structures. This feature allows lowering memory usage and file sizes by increasing the compression level, but this will also reduce the run-time speed of the program. Lowering the compression increase execution time, but also increase memory usage and file sizes. Determining the compression value depends on the hardware and the performance requirements of the application. As part of extending this software, the compromise between memory usage and run-time speed should be investigated.

7.4.4 Save and load chip information

In chapter 6, a series of tests are described. The tests are used to evaluate the current state of an ALPIDE chip. A great addition to the software would be to store this information and load it into the software at a later point. Currently, the software can load noisy pixels of chips from the configuration file and mask them. Declaring noisy pixels is a manual process. A better solution would be to let the analogue scan store a list of noisy pixels which can be loaded and handled at a later point. Other attributes are malfunctioning memory regions, varying pixel thresholds, and defect digital pixel circuitry. There are multiple ways to store this information. Two examples are object-serialization or storing a text file in XML, JSON or custom format.

Object serialization is a process where objects are written to file, buffer or sockets in binary or a human-readable format. This resource can be distributed and reconstructed (deserialized) by the same computer or elsewhere. Listing 7.1 gives an example of how scan results can be grouped in a structure. Serialization is part of Boost library and is simple to implement. It also only generates one file, instead of writing one text file for each scan result.

Listing 7.1: Proposal of structure holding tests results and fields identifying the chip

```
struct AlpideCircuitryInformation
{
    int chipID;
    int staveID;
```

```
    DigitalScanResults     * digital_info;
    AnalogueScanResults    * analogue_info;
    FIFOScanResults        * FIFO_info;
    RegisterScanResults    * register_info;
    ThresholdScanResults   * threshold_info;
    DACScanResults         * DAC_info;
};
```

With this feature, tests can retrieve and store information about ALPIDEs which can be used in the chip configuration process before an experiment.

7.4.5 Parser optimization

The parser discussed in section 5.2.2 showed to be rather slow. One of those reasons is the allocation and deallocation of PRUEvent objects. Each time a PRU header arrives the parser, an object is allocated on the heap. When the associated trailer word arrives, the event object is written to file and deleted from memory. Memory allocation and deallocation are costly operations. A reduction of these methods would greatly increase the executing speed of the application. A classical solution is to implement a buffer (memory) pool. The buffer will get a fixed size at the start of the program and delegated to PRUEvent objects as they are created. If more memory is needed the pool will be double its pool size. This will significantly decrease the allocation of memory and letting the memory be reused when a PRUEvent is complete. The drawback is that the buffer pool may become very large.

The readout task converts PRU words into hexadecimal as a string, which in turn is converted to a binary as a string. The binary string is then converted into different primitive types such as long int and int by the parser. This series of conversion is time-consuming and can be bypassed in many different ways. One of them is to let the parser receive PRU words as unsigned integers of size 128 and then use bit fiddling to extract fields from the word into the PRUEvent object. Appendix E gives a proposal of how a PRU word may be parsed when in a vector with 16 unsigned eight-bit elements. A test was created to compare this function to the current implementation. Both methods were set to parse 100000 test words with 1000 iterations, and a timer was set to measure the execution time. The result showed that the new function performed approximately *14,26* better. This was expected since it avoids string conversions. It is worth mentioning that the host system is compiled with the optimization flag -O0 which tells the compiler not to perform any optimizations. The tests also had zero optimizations, but if the flag -O2 is set, the new function performed approximately *42* times better compared to the current implementation.

Using the flag increase performance, but may induce unexpected behavior in a multithreaded environment if not careful.

7.5 Conclusion

This thesis has presented the full implementation of a software framework with developed applications to acquire and analysis detector data through the PRU. The software can be used to automatically configure and calibrate both ALPIDEs and readout boards. It provides a vast amount of features to develop new tests, and proper build system to facilitate further development and managing the software. The current and future challenges have been addressed and solutions have been presented. The peak was shown by the result of a real experiment, but it also confirms that the host software is a proof of concept and will undergo heavy alteration during the life span of the pCT project.

Bibliography

[1] Anja Strøen, Åse Marit Befring, and Marit Gjellan. Ap og Høyre kappes om kreftbehandling i valgkampinnspurten, 2017. URL https://www.nrk.no/norge/ap-og-hoyre-kappes-om-kreftbehandling-i-valgkampinnspurten-1.13680267.

[2] Regjeringen. Etablering av protonsentre i Bergen og Oslo. Technical report, Den norske regjering, 2017. URL https://www.regjeringen.no/no/aktuelt/etablering-av-protonsentre-i-bergen-og-oslo/id2579970/.

[3] B Abelev et al. Technical Design Report for the Upgrade of the ALICE Inner Tracking System. *Journal of Physics G: Nuclear and Particle Physics*, 41(8):087002, aug 2014. ISSN 0954-3899. doi: 10.1088/0954-3899/41/8/087002. URL http://stacks.iop.org/0954-3899/41/i=8/a=087002?key=crossref.f69e2a1127eed3a3b5681349a0966476.

[4] The Nordic Radiation Protection co-operation. STATEMENT CONCERNING THE INCREASED USE OF COMPUTED TOMOGRAPHY IN THE NORDIC COUNTRIES. Technical report, The Nordic Radiation Protection co-operation, 2012. URL https://www.dsa.no/dav/db58f19fef.pdf.

[5] David J Brenner and Eric J Hall. Computed Tomography — An Increasing Source of Radiation Exposure. *New England Journal of Medicine*, 357(22):2277–2284, 2007. doi: 10.1056/NEJMra072149. URL https://doi.org/10.1056/NEJMra072149.

[6] Ganesh Tambave. Characterization of Monolithic CMOS Pixel Sensor Chip with Ion Beams for Application in proton Computed Tomography.

[7] H.E.S. Pettersen, J. Alme, A. Biegun, A. van den Brink, M. Chaar, D. Fehlker, I. Meric, O.H. Odland, T. Peitzmann, E. Rocco, K. Ullaland, H. Wang, S. Yang, C. Zhang, and D. Röhrich. 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, jul 2017. ISSN 0168-9002. doi: 10.1016/J.NIMA.2017.02.007. URL https://www.sciencedirect.com/science/article/pii/S0168900217301882.

[8] David Hansen, Joao Seco, Thomas Sangild, S Ø Rensen, J Ø Rgen, Breede Baltzer Petersen, Joachim E Wildberger, Frank Verhaegen, and Guillaume Landry. *A simulation study on proton computed tomography (CT) stopping power accuracy using dual energy CT scans as benchmark*, volume 54. jul 2015. doi: 10.3109/0284186X.2015.1061212.

[9] H.F.-W. Sadrozinski, T. Geoghegan, E. Harvey, R.P. Johnson, T.E. Plautz, A. Zatserklyaniy, V. Bashkirov, R.F. Hurley, P. Piersimoni, R.W. Schulte, P. Karbasi, K.E. Schubert, B. Schultze, and V. Giacometti. Operation of the preclinical head scanner for proton CT. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 831:394–399, sep 2016. ISSN 0168-9002. doi: 10.1016/J.NIMA.2016.02.001. URL https://www.sciencedirect.com/science/article/pii/S0168900216001455.

[10] ALICE ITS ALPIDE. ALPIDE Operations Manual. 2016.

[11] M. Mager. ALPIDE, the Monolithic Active Pixel Sensor for the ALICE ITS upgrade. *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 824:434–438, jul 2016. ISSN 01689002. doi: 10.1016/j.nima.2015.09.057. URL https://www.sciencedirect.com/science/article/pii/S0168900215011122.

[12] Karl Emil Sandvik Bohne. Ethernet-Based Control System and Data Readout for a Proton Computed Tomography Prototype. jun 2018. URL http://hdl.handle.net/1956/18466.

[13] Ola Slettevoll Grøttvik. Design of High-Speed Digital Readout System for Use in Proton Computed Tomography. jun 2017. URL http://hdl.handle.net/1956/16041.

[14] Ola Grøttvik. pRU Data Format Specification. Technical report, 2018.

[15] Dimitri van Heesch. Doxygen. URL http://www.doxygen.nl/.

[16] Kitware. CMake. URL https://cmake.org/.

[17] The C++ Community. The Boost Library. URL https://www.boost.org/.

[18] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, Ph. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Gonzalez Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. Marcos Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel. ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 180(12):2499–2512, dec 2009. ISSN 0010-4655. doi: 10.1016/J.CPC.2009.08.005. URL https://www.sciencedirect.com/science/article/pii/S0010465509002550?via{%}3Dihubhttps://root.cern.ch/.

[19] Magnus Mager. ALPIDE series mass testing and classification. Technical report, ALICE ITS Collaboration. Received privately by email.

[20] Steve Arar. What Are the DNL and INL Specifications of a DAC? Non-Linearity in Digital-to-Analog Converters, 2019. URL https://www.allaboutcircuits.com/technical-articles/understanding-dnl-and-inl-specifications-of-a-digital-to-analog-converter/.

Appendix A

Event_t struct

```
class event_t : public TObject {
public:
 uint8_t ru_id;
 uint8_t stave_id;
 uint8_t chip_id;
 bool empty_frame;
 std::vector<std::pair<int,int>> xy;
 uint32_t event_id;
 uint32_t abs_time;
 uint8_t bunch_counter;
 uint8_t alpide_mode;
 uint8_t trigger_source;
 bool busy;
 uint16_t spill_id;
 bool error_busy_violation;
 bool error_decode_protocol;
 bool error_event;
 bool error_empty_region;
 bool error_double_busy_on;
 bool error_double_busy_off;
 bool error_buffer_overflow;
 bool error_max_size;
 bool error_max_wait;
 ClassDef(event_t, 1);
};
```

Appendix B

Software run environment

This section includes a list of libraries and environments the software has been tested with.

- Ubuntu 16.04 LTS or CentOS 7
- GCC 7.3
- C++11
- ROOT 6.14
- CMake 3.14
- Boost 1.58
- Doxygen 1.8.11

Appendix C

Configuration File

```
[scan_general]
NMASKSTAGES  = 200
PIXPERREGION  = 32
NINJ  = 1000
CHARGESTART = 0
CHARGESTOP = 50
CHARGESTEP = 1
TRIGGERTRAINS = 30

[board_vcu118]
ENABLE   = true
ADDRESS   = 192.168.1.10
TCPPORTNUMBER = 49153
PULSEDELAY  = 2500
TRIGGERDELAY = 40000
PULSETRIGGERDELAY = 20
PRECOMMANDDELAY  = 0
TRIGGERTRAINDELAY = 40000000

[chip_4]
ENABLE   = false
CHIPID   = 3
STAVE_ID  = 0
DISABLEDREGIONS_0_15 = 0x0
DISABLEDREGIONS_15_30 = 0x0
CHIPREADOUTMODULENUMBER = 2
NOISYPIXEL_0_ROW = 261
```

```
NOISYPIXEL_0_COL = 23
```

Appendix D

How to run tests

This section gives an complete tutorial on how to run the tests. All required libraries and tools are listed in chapter B. The sequence below assumes that make and git is installed and permission to access the repositories.

1. git clone https://git.app.uib.no/pct/data-format-sw.git
2. change to data format library project directory
3. mkdir build && cd $_
4. cmake ..
5. make EventDict
6. sudo make lib
7. git clone https://git.app.uib.no/pct/wp3.git
8. git clone https://git.app.uib.no/pct/alpide-sw.git
9. change to alpide sw project directory
10. mkdir build && cd $_
11. cmake ..
12. make digitalscan threshold analoguScan fifo dacscan

Appendix E

Improved PRU Parsing

```
uint32_t get_field(vector<uint8_t> &word, int pos, int depth) {
        uint32_t answer = 0;
        uint32_t vector_pos = (128 - pos)/8;
        uint8_t blocks = depth / 8 + 1;
        uint8_t ans_shift;
        volatile int depth_count = depth;
        int pos_count = pos;
        for(int i = 0; i < blocks; i++) {
                uint8_t temp_ans = word.at(vector_pos+i);
                uint8_t mask = ~0;
                uint8_t bit_pos = pos_count % 8;
                uint8_t mask_shifts_right = 0;
                if(depth_count >= 8) {
                        mask_shifts_right += 0;
                        ans_shift = 8;
                }
                else {
                        mask_shifts_right += bit_pos - depth_count + 1;
                        ans_shift = depth_count;
                }
                uint8_t mask_shifts_left = 7 - bit_pos;
                uint8_t tot_mask_shifts = mask_shifts_left +
                   mask_shifts_right;
                mask = mask >> (tot_mask_shifts);
                mask = mask << (bit_pos + 1 - (8 - tot_mask_shifts));
```

```
                temp_ans = (temp_ans & mask) >> mask_shifts_right;
                depth_count = depth_count - bit_pos-1;
                pos_count -= bit_pos+1;
                answer = answer << ans_shift;
                answer = answer ^ temp_ans;
        }
        return answer;
}
```

Appendix F

Calculating Integral Nonlinearity and Range of VCLIP DAC

```
using namespace std;

vector<double> DNL;
double range;

int calulateDNL(const double transition_step, const char* filename) {
    int codeword;
    double voltage;
    double expected_vol;

    double min = 0;
    double max = 0;

    FILE *fp = fopen(filename, "r");
    if(!fp) {
        cout << "Could not resolve file. Exiting..." << endl;
    }
    while(fscanf(fp, "%d %lf", &codeword, &voltage) == 2) {
        expected_vol = codeword * transition_step;
        DNL.push_back(voltage - expected_vol);

        if(voltage < min)
```

```
            min = voltage;
        if(voltage > max)
            max = voltage;
    }
    range = max - min;
}

double calculateINL() {
    double INL;
    for(double d : DNL) {
        INL += d;
    }
    return INL;
}
/*Only for VCLIP*/
int main(int argc, char **argv) {

    for(int i = 0; i < argc-1; i++) {
        calulateDNL(0.006757, argv[i+1]);
        cout << "INL: " << calculateINL() << " Range: " << range << endl;
        DNL.clear();
    }
    return 0;
}
```