# Design of a Configuration and Monitoring System for the Power Supply in the ProtonCT Project

Håvard Birkenes

Master's thesis in Physics

University of Bergen
Department of physics and technology

October, 2022

**Acknowledgements**

I want to thank my supervisor Johan Alme for invaluable guidance and feedback while writing this thesis. I also want to thank Matthias Richter for his input regarding software design, which gave me insight into a topic that was unfamiliar to me.

Being part of the ProtonCT project has been an exciting part of my studies. Being part of a larger project spanning several institutions was a fun and learning-rich experience, which I am grateful for. Working alongside Birger Olsen, Jakob Hauser, and Martin Eggen has been a delight and has taught me much about working as a team.

Thank you to my dear friends and peer students who have been with me for the last two years: Amalie, Bendik, Birger, Eirik, and Thomas. There was never a dull moment in the study hall, and there were always exciting conversations to partake in, whether academic or casual.

Finally, I want to thank my friends and family, who supported me during my studies. I am truly grateful for the fantastic people who have been with me and always believed in me. Thank you.

**Abstract**

The ProtonCT-project is a collaboration established at the University of Bergen and several institutions worldwide to create a prototype pCT-scanner. The pCT-scanner would be used to improve dosimetry plans for proton therapy by measuring the relative stopping power of the protons, instead of calculating it using a traditional CT-scan.

The prototype employs a Digital Tracking Calorimeter to measure the energy of the protons. This calorimeter is distinct in its design; it comprises 43 layers of pixel sensors. The pixel detectors were developed for the ALICE project by CERN. Each layer is comprised of 12 strings of ALPIDE chips, where a string is 9 ALPIDE chips mounted on a flex cable. These layers can track the trajectory and measure the energy of protons, which can be used to measure the relative stopping power.

The power delivery to the 43 layers of the calorimeter is still under development. Currently, a Monitoring Board is used to deliver the power to the strings and monitor their current consumption, voltage levels, and temperature. This thesis presents a control system solution for the power delivery system. This solution employs system design methodology to create a robust and modular design that can configure and monitor the power delivery to the layers. IPbus, a reliable communication protocol developed at CERN, is used to transmit data between the components of the power delivery system. Several levels of API abstraction have been designed to interface with IPbus and perform various functions, such as configuring the Monitoring Board and monitoring the measurements from the board.

Database systems have been developed to store data from the configuration and monitoring processes. A web interface has been deployed that gives the user an overview of the monitoring data. Lastly, GUIs have been developed to serve as the interface between the user, the databases and the Monitoring Board.

Tests and verification were performed of the power control system, and the results showed that the control system fulfilled the timing requirements for this project. Finally, a discussion is made on the structure of the control system. The discussion covers the control system and how it can be reused and expanded upon to cover the other systems of the pCT-project.

# Contents


**1 Introduction 1**
1.1 Background . . . 1
1.2 Bergen PCT project . . . 2
1.3 Power Control and Monitoring . . . 4
1.4 Objective of this thesis . . . 5
1.5 Structure of this thesis . . . 7

**2 Radiotherapy and Proton Therapy 8**
2.1 Radiotherapy . . . 8
2.1.1 History and General Use . . . 8
2.1.2 CT-scan . . . 10
2.2 Proton Therapy . . . 11

**3 Control System Design Methodology 17**
3.1 Overview . . . 17
3.2 SCADA systems . . . 17
3.3 SCADA software development . . . 18
3.3.1 Input and output signals . . . 18
3.3.2 Defining PPC operations . . . 19
3.3.3 Developing workstation software . . . 19
3.3.4 Software documentation . . . 19
3.4 Software design . . . 20
3.4.1 Object Oriented Programming . . . 20
3.4.2 Git and GitHub . . . 20

**4 The PCT Project 22**
4.1 Introduction . . . 22
4.2 Digital Tracking Calorimeter . . . 23
4.3 ALPIDE chips . . . 24
4.4 Cooling System . . . 25
4.5 Readout System . . . 26
4.6 Power Delivery System . . . 27

**5 Power Control System 29**
5.1 Control software . . . 30
5.2 IPbus . . . 32
5.2.1 Overview . . . 32
5.2.2 IPbus software . . . 33
5.3 MB Hub . . . 35
5.4 Monitoring Board . . . 38

**6 Configuration System 41**
6.1 Configuration API . . . 41
6.1.1 IPbus API . . . 41

6.1.2 Microcontroller API . . . . . 42
6.1.3 Configuration API . . . . . 42
6.2 Powering algorithm . . . . . 43
6.3 MongoDB Database . . . . . 44
6.4 Configuration GUI . . . . . 47
6.5 Configuration Timing . . . . . 51
6.5.1 Testing configuration timing . . . . . 52

**7 Monitoring System 53**
7.1 Monitoring API . . . . . 54
7.2 Monitoring GUI . . . . . 57
7.3 InfluxDB . . . . . 57
7.4 Grafana . . . . . 59
7.5 Characteristics of monitoring operation . . . . . 61
7.5.1 Transmission speed of monitoring process . . . . . 61
7.5.2 Storage Space . . . . . 62
7.5.3 Optimizing data acquisition . . . . . 63
7.6 Downsampling of InfluxDB . . . . . 64

**8 Testing and Verification 69**
8.1 Test Coverage . . . . . 69
8.2 Simulation . . . . . 70
8.2.1 Overview . . . . . 70
8.2.2 Features . . . . . 71
8.3 Testbench . . . . . 72
8.4 Test setup . . . . . 72
8.5 MB Hub Verification . . . . . 73
8.5.1 Reliability Verification . . . . . 73
8.5.2 MB Hub Speed Test . . . . . 73
8.6 USART Interface Test . . . . . 74
8.7 Baud rate error rate . . . . . 77

**9 Discussion 80**
9.1 Reusability . . . . . 80
9.1.1 Monitoring software . . . . . 80
9.1.2 Configuration software . . . . . 80
9.1.3 Register Package . . . . . 81
9.2 Error handling . . . . . 81
9.2.1 Control Software . . . . . 81
9.2.2 Microcontroller . . . . . 81
9.3 Configuration System . . . . . 82
9.4 USART communication . . . . . 83
9.5 Future work . . . . . 83
9.5.1 Docker Image . . . . . 83
9.5.2 Grafana . . . . . 84

9.5.3 System parameters . . . 84
9.5.4 Security measures . . . 84

**10 Outlook and Conclusion 85**
10.1 Summary . . . 85
10.2 Outlook . . . 85
10.3 Conclusion . . . 86

**A Software 87**
A.1 Microcontroller AdressMap . . . 87
A.2 FPGA register map . . . 93

**B Setting the Environment 94**
B.1 Python libraries . . . 94
B.1.1 Bitstream . . . 94
B.1.2 Qt5 Designer . . . 94
B.1.3 pyMongo . . . 94
B.1.4 InfluxDB . . . 94
B.1.5 Influx Python Client . . . 95

**C Typical errors 96**
C.1 Bit Error Rate and Masking . . . 96
C.2 InfluxDB Retention Policy . . . 96
C.3 Bitstream . . . 96
C.4 IPbus Address Values . . . 97
C.5 Grafana . . . 97
C.6 FPGA FIFOs . . . 98

# Acronyms

**ALICE** A Large Ion Collider Experiment.

**ALPIDE** ALICE Pixel Detector.

**API** Application Programming Interface.

**CERN** European Organization for Nuclear Research.

**CT** Computed Tomography.

**DCS** Detector Control System.

**DTC** Digital Tracking Calorimeter.

**FIFO** First In First Out.

**FPGA** Field Programmable Gate Arrays.

**GUI** Graphical User Interface.

**HDR** High Dose Rate.

**HMI** Human Machine Interface.

**HU** Hounsfield Unit.

**IMRT** Intensity-Modulated Radiation Therapy.

**iOT** internet Of Things.

**ITS** Inner Tracking System.

**LHC** Large Hadron Collider.

**LVDS** Low Voltage Differential Signaling.

**MAPS** Monolithic Active Pixel Sensor.

**MB** Monitoring Board.

**MSB** Most Significant Bit.

**MTBF** Mean Time Between Failure.

**OOP** Object Oriented Programming.

**PCS** Power Control System.

**pCT** Proton Computed Tomography.

**PET** Positron Emission Tomography.

**PPC** Programmable Process Controller.

**pRU** pCT Readout Unit.

**RnW** Read not Write.

**RSP** Relative Stopping Power.

**RT** Radiation Therapy.

**SCADA** Supervisory Control And Data Acquisition.

**TC** Transition Card.

**USART** Universal Synchronous Asynchronous Receiver Transmitter.

# 1 Introduction

## 1.1 Background

Cancer is the second most prevailing cause of death in the world, behind cardiovascular diseases. 14 million new cases of various cancers were reported worldwide in 2019, and as life expectancy increases, cancer rates are also expected to increase[1]. There are several treatments for cancers, among them is Radiation Therapy (RT), a treatment developed in the early 1900s using x-rays to irradiate harmful tumours. RT is an effective and non-invasive procedure, used to remove cancerous growth without surgical intervention. The main challenge of RT is delivering a lethal dose to the tumour without harming healthy tissue in the patient. Children are especially vulnerable to the long-term side effects of radiation therapy because their bodies are not fully developed yet.

**Figure 1.1:** Image of the first pediatric patient being treated for retinoblastoma using radiotherapy.[2]

To calculate the dose distribution, a Computed Tomography (CT)-scan is performed before radiation therapy. A CT-scan is done by sending X-rays through the patient, and detectors behind the patient measure the energy of the exiting photons. the absorption of the X-ray beams is proportional to the density of the tissue, meaning photons going through bone structures will lose most of their energy, while photons going through soft or no tissue will retain most of their energy. This property allows us to characterize the energy lost using the Hounsfield scale, a scale specifically designed for use in CT-scans. With the Hounsfield scale, it is possible to create an image of the patient's insides, which can be used for diagnostics and to calculate the dose distribution. However, even with these techniques, the patient will still receive a significant radiation dose in adjacent tissue. Using RT in high-risk areas, such as the brain or heart, can lead to severe damage to these organs.

Particle therapy is a new treatment developed in the mid-1900s as an alternative to RT.

Particle therapy was sought as an alternative because particles allow for concentrating the dose distribution, which avoids damaging healthy tissue. Particles release very little energy while moving through a medium; it is not until the particle completely stops that it effectively releases all its energy. This phenomenon is known as ”Bremsstrahlung”, and it allows us to deliver high doses of energy to a concentrated area while having relatively little entry dose and effectively zero exit dose. The moment when the particle releases all its energy is known as the ”Bragg-Peak”, and we can calibrate this ”peak” to target the tumour.

Knowing the Relative Stopping Power (RSP) of the medium the particles are sent through is essential to control the Bragg-Peak. A traditional CT-scanner can be used to calculate the RSP; this is done by measuring the body’s attenuation to radiation, measured in Hounsfield units, and converting the units into RSP. However, the conversion from Hounsfield-units to RSP is only an approximation, leading to higher uncertainty in the measurement. The stochastic nature of particles’ energy loss also leads to higher uncertainties, making it even more crucial to have precise measurements when performing particle treatment. Directly measuring the RSP with a particle CT-scan can circumvent the uncertainty stemming from converting from Hounsfield units. This has led to further research and development of particle-based CT-scan machines.

## 1.2 Bergen PCT project

Today the most common type of particle therapy is proton therapy; there are currently over 33 proton therapy centres in Europe[3]. Norway plans to open two proton therapy centres by the end of 2024, one in Oslo and one in Bergen. The Bergen pCT-project is a collaboration with the University of Bergen and several other institutions to build a Proton Computed Tomography (pCT)-scanner to be used in particle therapy. The pCT-scanner is realized with a Digital Tracking Calorimeter (DTC) to measure the energy levels and trajectory of the protons exiting a patient exposed to a proton beam, to measure the RSP. A model of the DTC is given in Figure 1.2.

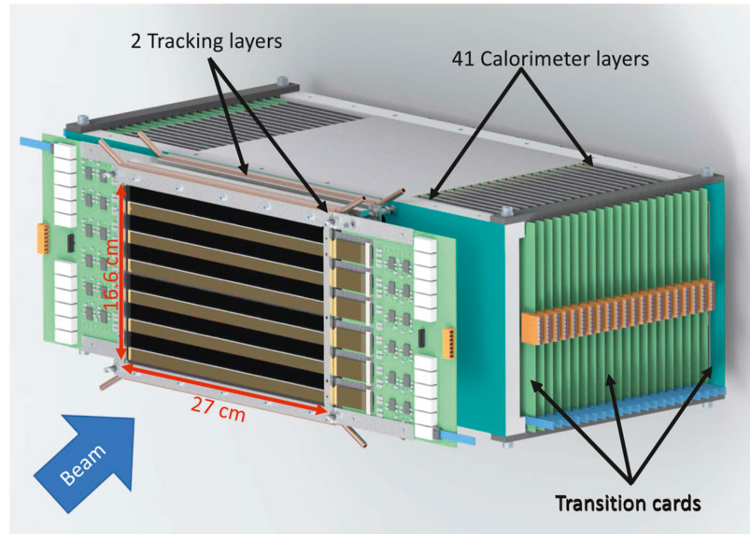


**Figure 1.2:** Image of the DTC to be used in the Bergen pCT project[4].

The DTC is developed using ALICE Pixel Detector (ALPIDE)-chips that was originally used for the Inner Tracking System (ITS) in the ALICE experiment at CERN. The DTC is comprised of 43 layers, and each layer is assembled out of 12 strings with nine ALPIDE-chips each. The protons enter the front of the detector, and the chips measure their energy as they travel through each layer. 2 of the 43 layers are designated tracking layers, calculating the protons' trajectory when exiting the patient. It is necessary to track the angle of exiting protons due to effects such as coulomb-scattering, which affects the energy and trajectory of the protons. A DTC is traditionally made with two tracking layers in the front and two layers in the rear of the DTC, but the pCT-project only has the rear trackers. *Single-sided* systems have advantages over *double-sided* systems, such as reducing the cost and complexity of the system. A simplified block diagram of the pCT system is shown in Figure 1.3.

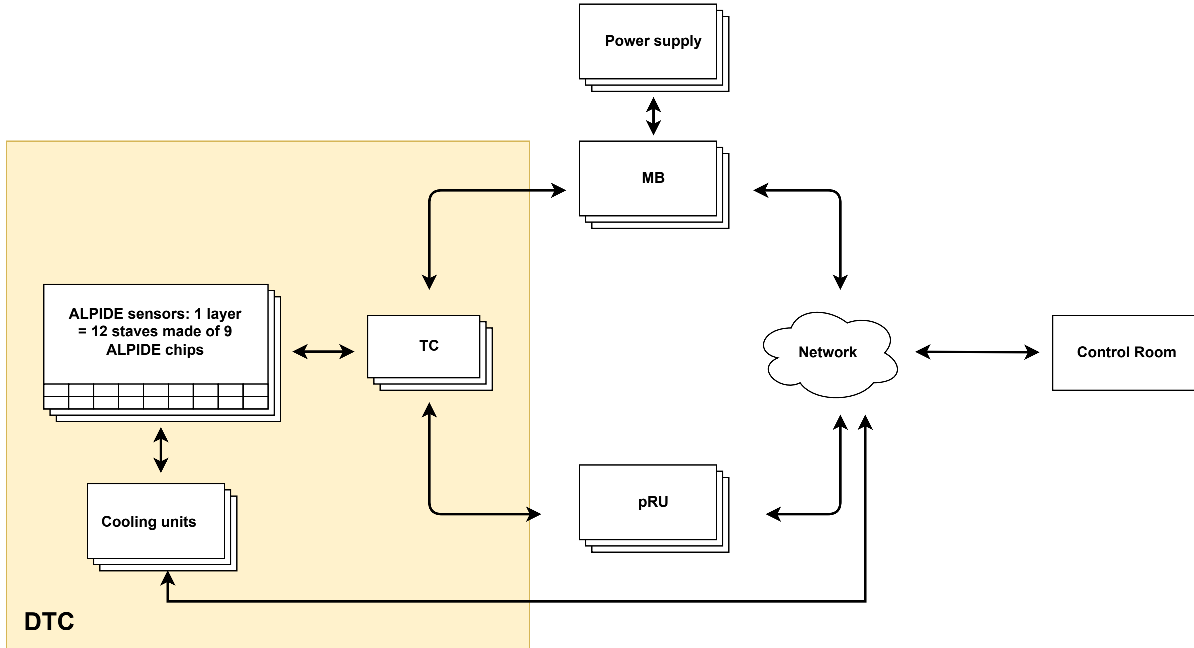


**Figure 1.3:** Simplified block diagram of the pCT system.

The figure shows the three systems encompassing the ALPIDE sensors, power delivery, readout, and cooling, all managed by the Control Room. The Monitoring Board (MB) and pCT Readout Unit (pRU) transfer power and perform readout of the strings, respectively. Cooling is located next to the detector layers and is directly connected to the Control Room. All three systems are still in development; this thesis will focus on the power delivery system.

A separate MB is responsible for delivering the power to the Transition Card (TC) and monitoring the temperature and current of the strings. Data transmission between the client and hardware is performed using IPbus, an established ethernet-based communication protocol developed for particle-physics experiments. The MB must deliver enough current and monitor the temperature and current draw of the strings of ALPIDE-chips. The plan is to have a power supply for each layer of the DTC and an MB for each layer to supply power and monitor the current draw.

The MB uses a microcontroller to monitor the temperature and current draw of the layer; this allows us to quickly turn off strings if the temperature or current exceeds safe ranges. For this purpose, the microcontroller has several configurable registers for threshold values and readings of analog current, digital current and temperature. The MB requires a configuration system for safely powering the strings without damaging the equipment. A client-side monitoring system is also needed to give the user accurate information about the strings during a run.

## 1.3 Power Control and Monitoring

The detector layer power system for the pCT requires both a configuration and a monitoring system. If the current consumption or temperature of the strings reaches a critical level,

then they are turned off as fast as possible by the microcontroller. However, there is also a need for a control and monitoring system that interfaces with the MB itself. The microcontroller must be configured through some process, and the system user needs a way to monitor the performance of the strings. It is common in the industry to use a standard for designing control systems called SCADA. Designing our control system using those standards is worthwhile to ensure the system is standardized and well-structured.

The configuration system must be able to configure the microcontroller with threshold values for current, voltage and temperature. The system also needs an algorithm for powering the strings one by one. Simultaneously turning on all strings will cause a large power spike, which can damage the equipment. The configuration system does not have a specific time constraint since the microcontroller takes care of time-critical events; however, the system should still not be too slow. The system should not be ”laggy”, meaning the computer should not lag when running the program, which is cumbersome for the user. Additionally, if the configuration process is slow, repeated troubleshooting becomes tedious, which should be avoided. Therefore the configuration should at most take 10 seconds to complete.

The monitoring system has similar requirements and no specific time constraint, but ideally it should be quick and not feel ”laggy” to the user. The monitoring also has a polling frequency, determining how often it polls data from the microcontroller. The monitoring system should have a sampling frequency of a minimum 1Hz to get a good sampling of the power delivery parameters. However, a frequency of about 2-4Hz would be ideal for our system.

## 1.4 Objective of this thesis

This thesis aims to design and build the configuration and monitoring system for the pCT power delivery to the detector layers. Many aspects of the systems must be assessed and considered to ensure they are suitable for the pCT-project. Both systems must be able to communicate with the microcontroller on the monitoring board. Therefore, an FPGA-solution is used with IPbus to receive and transmit data between client software and the microcontrollers. Figure 1.4 shows a block diagram of the power delivery system.

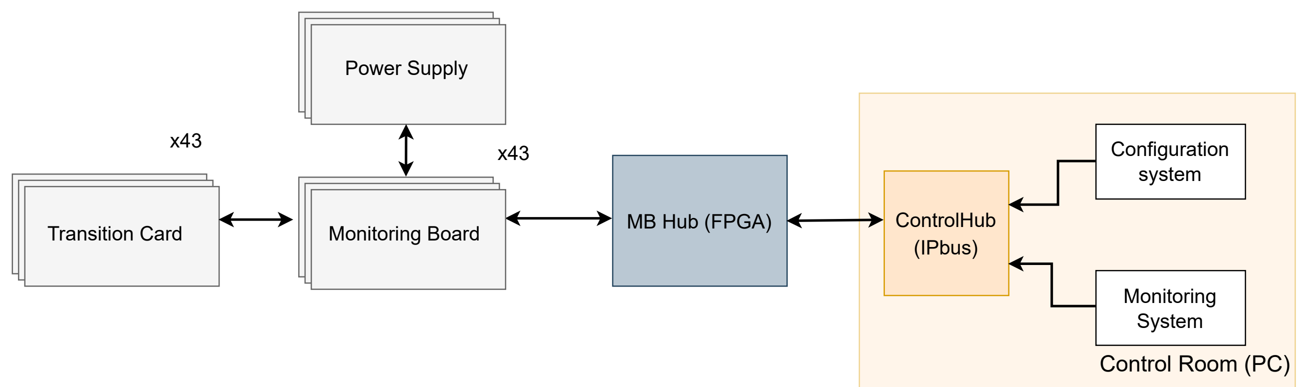


**Figure 1.4:** Block diagram of the power delivery system. The detector layers are directly connected to the TCs

The work will focus on creating the configuration and monitoring systems that will interface with the MBs and the MB Hub. This involves creating an Application Programming Interface (API) for the MBs, a Human Machine Interface (HMI) and databases for storing the data.

The work will use various software tools for communicating with IPbus and creating Graphical User Interface (GUI)s. The focus will be on creating a control system that is logically structured, modular, and generic. This will make the system easy to modify, expand upon, and reduce troubleshooting time, which is essential when working on an extensive control system. For example, the cooling and readout systems do not have a complete control system in place, but if the power control system is generic, then parts of that system can be reused for cooling and readout. The generic design would also benefit the pCT-project because it would be simpler to manage three similar systems than three isolated, custom systems.

The requirements for this control system are as follows:

- Configure all MBs in the time span of 10 seconds.
- Store configuration data in a database allowing for easy loading and saving configuration sets.
- A GUI that allows a user with little knowledge of the system to load configuration sets.
- A powering-on algorithm that turns on the strings one-by-one, leading to a slow ramp-up of power usage, to avoid creating a large power spike.
- Periodically read current draw and temperature from each layer and store the data in a time series database.
- Display the monitoring data from each layer in intuitive and easy-to-understand graphs.
- A low and high-level API to communicate with the IPbus module on the Field Programmable Gate Arrays (FPGA) and the microcontroller on the MB.
- Generic, modular Object Oriented Programming code that is easy to change and modify, so the system can be expanded upon and reused for similar projects in the future.
- The configuration process must not be "laggy" for the user and should not take more than a few seconds to complete.
- The polling frequency of the monitoring process must be at least 1Hz, but ideally it should be 2-4 Hz.
- The software design must follow software process standards and SCADA control system standards.

  The power system requires a low-level API that can perform basic communication with the MB, and this API becomes the base interface for higher level APIs. Furthermore, following software process standards ensures that the system is structured logically, which helps the system be modular and generic. This means the system can be easily modified and expanded upon in future, which is a must for a control system with a long shelf time. Additionally, following the model for designing SCADA systems serves as a guideline for creating a structured and well-documented control system.

## 1.5 Structure of this thesis

**Chapter 2 - Radiotherapy and Proton Therapy** *This chapter goes through the theory of radiation and particle therapy and discusses the challenges with those treatments. The focus will be on how particle therapy can improve cancer treatment from traditional radiotherapy and what must be considered when using particle therapy.*

**Chapter 3 - Control System Design Methodology** *This chapter covers design methodology for control systems and software design. The focus is on the Supervisory Control And Data Acquisition (SCADA) architecture, how it is constructed and the methodology behind creating such a system. Finally, a discussion is made on the methodology of software design and common ways to structure code.*

**Chapter 4 - The PCT project** *This chapter introduces the pCT-project, the background of the project, and gives an overview of the different components and systems that comprises the pCT-project. It will discuss the calorimeter and the design of the detector layers. A description of the control system for power delivery, readout, and cooling is also given.*

**Chapter 5 - Power Control System** *This chapter describes the Power Control System, how it is constructed and how it functions. It details the three parts that make up the Power Control System (PCS), which are the control software, the MB Hub and the MB. It additionally describes the IPbus communication protocol and how it is implemented in the control software.*

**Chapter 6 - Configuration System** *This chapter covers the design of the configuration system used in the power delivery system. It covers the design of the configuration API and the layers of abstraction used in the lower-level API for the PCS. It then covers the design requirements of the configuration system. It also discusses the theoretical configuration timing of the system, along with the measured timing performed on the prototype setup.*

**Chapter 7 - Monitoring System** *This chapter describes the solution for a monitoring system for the MB. The chapter tackles the characteristics of the system and what is needed for this project, along with the choice of time-series database used to store the data points. Features such as the specification of storing monitoring data and displaying it to the user using third party software is also discussed. Finally, the different types of data filtering are discussed, along with how Pandas dataframe can be used to customize the filtering of data points.*

**Chapter 8 - Testing and Verification** *The prototype setup of the PCS is introduced and various tests are described for testing the entire PCS chain. The results of those tests are discussed, and what they mean for the viability of the test setup. It will also go over simulation classes that were created to test API functionality with IPbus on software.*

**Chapter 9 - Discussion** *This chapter discusses the work done in this thesis, what is incomplete or missing, and discusses how the work done for the power control system can be applied and reused for the other cooling and readout systems in the pCT-project.*

**Chapter 10 - Outlook and Conclusion** *This chapter sums up the work done in this thesis and details the future work that must be performed to make the system operational.*

# 2 Radiotherapy and Proton Therapy

## 2.1 Radiotherapy

Cancer is an umbrella term for many diseases involving out-of-control cell growth in the body, the most common types being breast, lung, and prostate cancer[5]. RT as treatment started in the early 1900s, using X-rays to kill malignant tumours, and has become a standard procedure to remove or prevent the growth of malignant tumours. External RT uses an external radiation source to deliver the dose. In contrast, internal RT injects a radioactive element as a liquid into the body or near the malignant tumour. This thesis will focus on external RT methods.

### 2.1.1 History and General Use

RT was historically performed by using an external X-ray source and sending the radiation into the patient to treat growths or lesions from diseases such as lupus or skin cancer. Figure 2.1 shows a patient being treated with X-rays before the dangers of radiation therapy were known. Today, the procedure is performed in a similar manner, but a better understanding of radiation behaviour and dosage planning enables us to target cancers inside the patient and better treat life-threatening diseases. Imaging techniques such as CT-scans allow us to measure the radiodensity of the tissue in a patient, which can be used to create a dosage plan for RT. Not only does this allow us to target tumours deep inside the body, but it also allows for higher accuracy dose distribution, which minimizes the amount of radiation absorbed by healthy tissue.

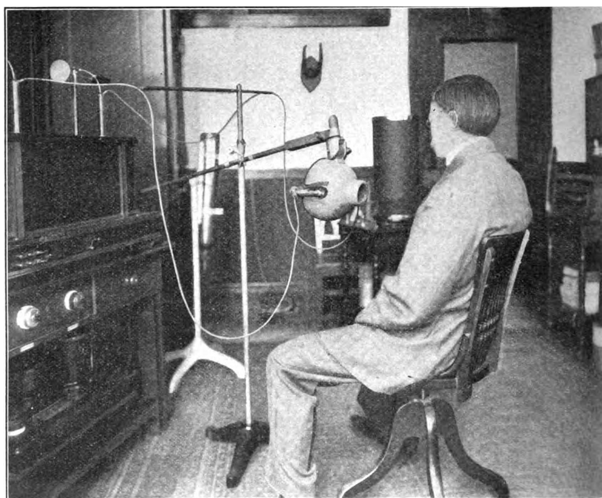

**Figure 2.1:** Image showing a patient being treated for tuberculosis in 1910, using X-rays[6].

Compared to chemotherapy and surgery, RT has the advantage of being a non-invasive procedure and can target specific areas in the body. This allows us to remove malignant tumours more effectively and is often used alongside surgery or chemotherapy to ensure that the entire tumour is removed. The downside of RT is that when the malignant tumour is irradiated, so is the healthy tissue around the tumour, which damages it and can lead to

more health issues. Using RT in tumours near critical organs, such as the heart or brain, is difficult and must be carefully calibrated to avoid damaging healthy tissue. Using RT in areas such as the head and neck has proven to cause many long-term side effects in patients, such as cognitive impairment and neurological damage. In rare instances, a new, local cancer may emerge[7]. Therefore, reducing the delivered dose to the tissue is crucial.

External delivery of radiation dose to a tumour is difficult because the healthy tissue in front of the tumour absorbs large amounts of the dose, which can damage the tissue. Alternative methods must therefore be applied to treat internal tumours in a patient. A solution is using Intensity-Modulated Radiation Therapy (IMRT), a method where multiple, weaker radiation sources are focused on the tumour. This effectively spreads the high entry dose throughout the body, sparing the healthy tissue while still giving the tumour a high dose. A study was done to compare the effectiveness of IMRT in comparison to the use of High Dose Rate (HDR) brachytherapy [8]. Figure 2.2 shows dose distribution of IMRT versus HDR.

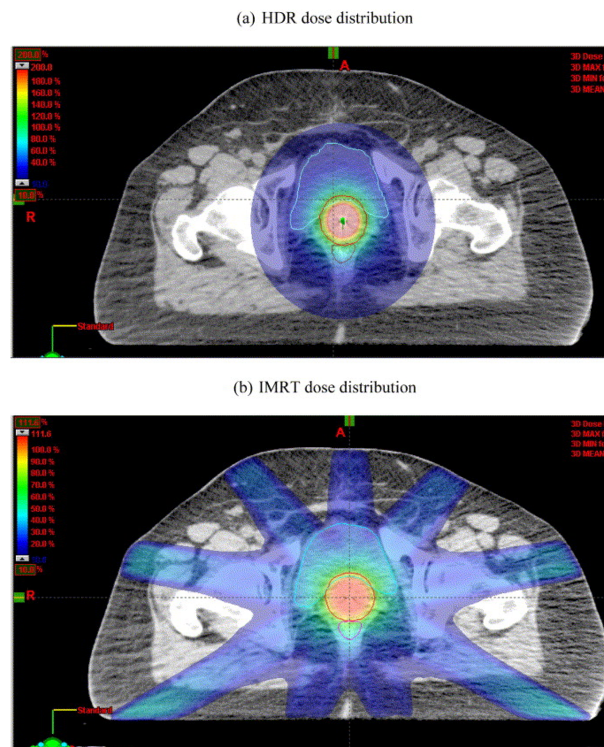


**Figure 2.2:** Image showing dose distribution targeting a tumour(red circle) using HDR(a) and IMRT(b)[8]

From the figure, the dose is spread throughout the body with IMRT, compared to HDR, where the dose is centred near the tumour. The study concluded that IMRT provided a substantial decrease in dose delivered to the organs-at-risk near the tumour, compared to HDR[8]. IMRT is a viable treatment for treating tumours, but a large dose is still delivered to healthy tissue near the tumour using this method. IMRT may not be a viable treatment option if the tumour is close to critical organs, such as the brain or heart.

### 2.1.2 CT-scan

CT-scans are performed before RT-treatment to calculate the body's radiodensity, which is the body's attenuation to radiation. These measurements are then used in dosage planning. It can also be used as a general imaging machine of a patient's insides, which can be used for diagnostics. A CT-scan uses X-rays and computer technology to image a "slice" of the patient's body; many of these slice pictures are taken and put together to form a slice-by-slice model of the entire body of the patient.

A CT-scan is normally realized with an X-ray tube that rotates around the patient that sends radiation through a slice of the patient. Sensors on the other side of the machine detect the photons as they leave the patient. Figure 2.3 shows a typical setup when performing a CT-scan.

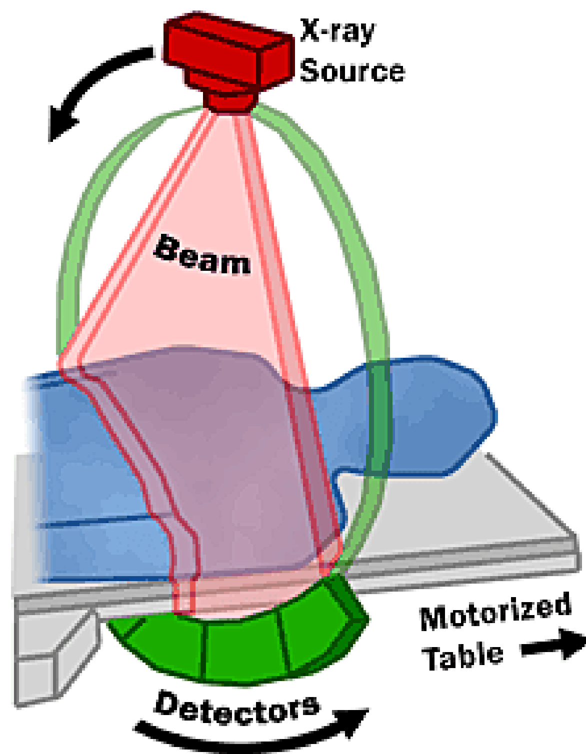


**Figure 2.3:** Image showing a typical setup for a CT-scanner[9]

The photons from the X-rays lose their energy as they move through a medium, in this case, the patient. The energy loss from the photons is proportional to the density of the tissue. The detectors on the back of the patient measure the remaining energy of the photons exiting the patient and can, in turn, calculate the radiodensity of the path the photon travelled through the body.

Radiodensity measurements used in CT-scans are measured in the Hounsfield-scale, which uses Hounsfield Unit (HU). HU, also called CT-unit, is a linear transformation of the absorption coefficient of the X-ray beam. 0 HU is arbitrarily set to be the energy lost as the X-ray travels through water, and -1000 HU is the energy lost when travelling through air.

Using this unit, we measure the relative energy absorption of different tissues in the body and use computer technology to reconstruct this information into a picture of the patient's insides. Figure 2.4 shows one slice from a CT-scan performed on the patient's lungs.

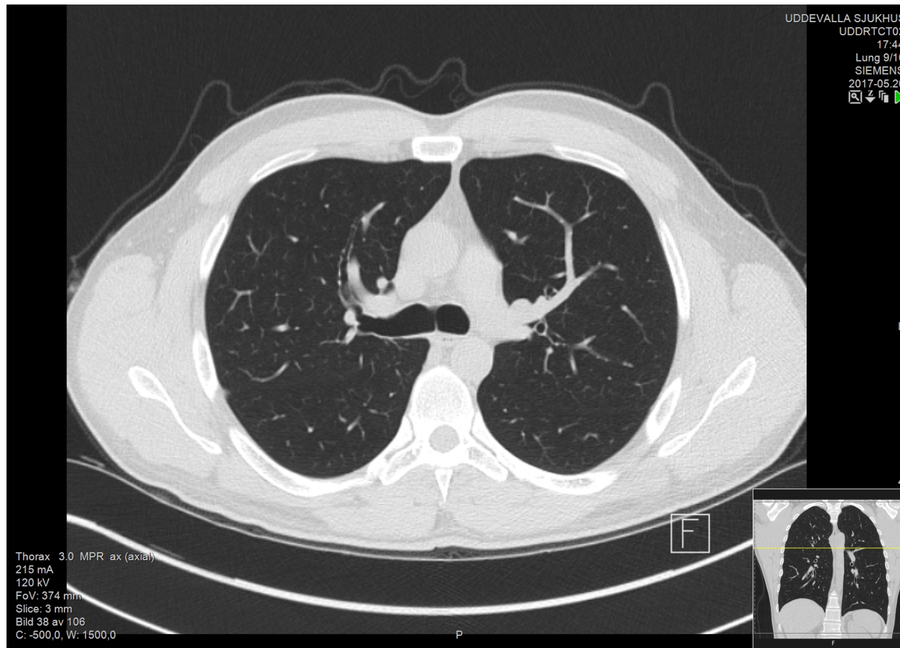


**Figure 2.4:** Image slice of a patient's lungs. Image down right of the image shows what slice of the lung is displayed. By Mikael Häggström, used with permission.

## 2.2 Proton Therapy

Particle therapy is a medical treatment similar to RT. However, high-energy particles are used instead of X-rays to irradiate the body. The most common one today is protons. The idea of particle therapy came about in the mid-1900s. Scientists discovered that, unlike photons, high-energy particles do not deposit their energy gradually as they move through a medium; instead, they sharply deposit most of it in one area. It was therefore theorized that proton therapy could be used as an alternative to RT to limit dose delivery to healthy tissue.

Figure 2.5 shows this sharp dose distribution of particles compared to the dose distribution of photons in water.

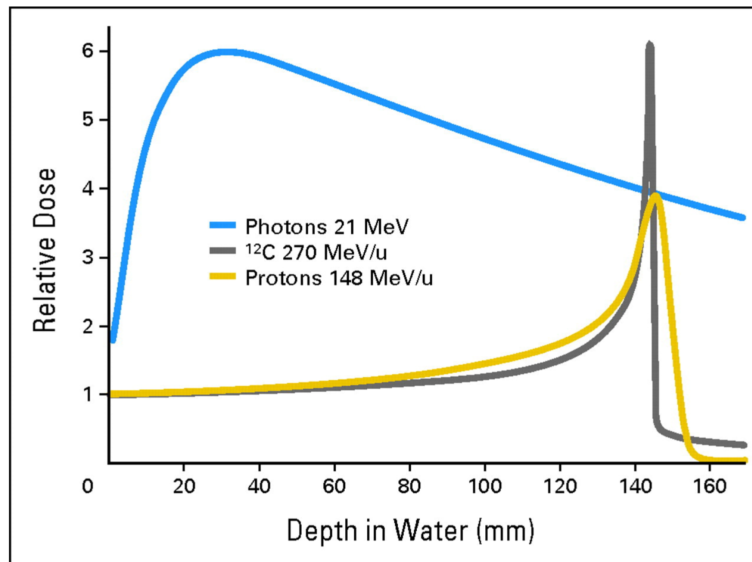


**Figure 2.5:** Graph displaying the relative dose given in water by photons, protons and carbon-12, as function of distance. The bragg peak of proton and carbon-12 is shown to the right[10].

From the figure, photons reach their max dose delivery early and slowly decrease as it moves in the water. On the other hand, protons have a relatively low dose delivery until it reaches the Bragg peak, where almost all energy is deposited. The reason protons deposit their energy is due to a phenomenon called "Bremsstrahlung", where the particle loses most of its energy as it slows down in the medium. To put it in another way, the energy loss of a particle in a medium is inversely proportional to the particle's velocity, creating the Bragg peak.

The effectiveness of proton therapy can be shown by comparing the dose delivered with conventional radiotherapy. Figure 2.6 compares the dose distribution of IMRT treatment and proton treatment of lung cancer in a patient.

One can observe from the figure that IMRT irradiates a large part of healthy tissue outside the tumour, as the entry and exit dose is high as predicted by Figure 2.5. On the other hand, the proton treatment has a lower entry dose, and most radiation is limited to the area around the tumour. Proton therapy allows us to irradiate the tumour without harming the healthy tissue in the body.

Proton therapy is a viable treatment option, but it comes with challenges that must be tackled before beginning treatment. There are political and economic challenges with proton therapy, including sizing the facility to house the particle accelerator. However, this thesis will focus on the physics challenge of proton therapy. Proton particles naturally behave very differently from massless photons. This means that photon dose and range calculations are insufficient for protons[12].

Range uncertainties can significantly impact proton therapy compared to RT. Figure 2.7 compares the dose distribution between radiation therapy and proton therapy and displays

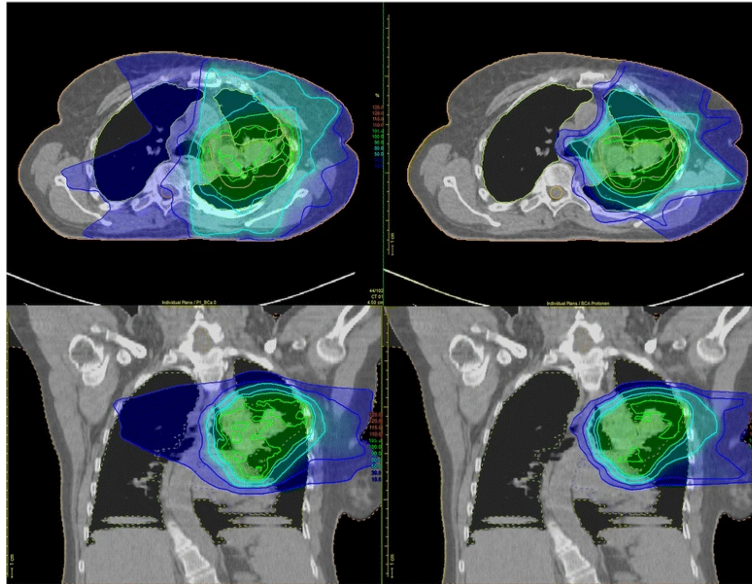

**Figure 2.6:** Image showing the dose distribution of IMRT treatment(left) and proton therapy(right) of a tumour in the lungs[11]. The upper image shows top-down image of the dose distribution, the lower image shows the same distribution from the side.

how range uncertainty can affect the dose distribution. The photons have a high entry dose, and the dose delivered is reduced as photons travel deeper into the tissue. The proton dose is delivered by tuning the Bragg Peak of the beam to cover a small part of the tumour. After delivering a high dose, the beam is tuned again, so the Bragg Peak covers another part of the tumour. This process is repeated until the entire tumour has been given a high dose. This creates a Spread Out Bragg Peak (SOBP) distribution, as shown in Figure 2.7.

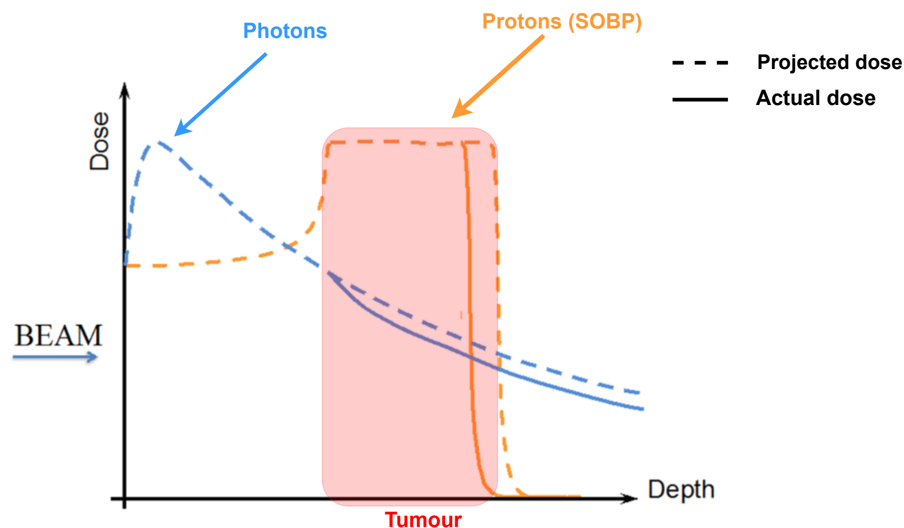


**Figure 2.7:** Image showing the effect of range uncertainties in photons(blue) and protons(orange)[12].

The dotted line is the projected dose to be delivered to the tumour, and the full line is the actual dose delivered due to range uncertainties in the RSP measurement. From the graph of the photons, we can see that, although there is a difference between the projected dose delivered and the actual dose, it does not impact the dose distribution in a significant way. This is because photons have a broad dose distribution; large parts of the tissue are irradiated regardless of the range uncertainty.

The deviation between the projected and actual dose is more detrimental for the protons. Because the dose delivery of the protons is much sharper, an undershoot in the dose delivery results in parts of the tumour not being covered by the Bragg Peak. Therefore almost no dose is delivered to it. Likewise, suppose there was an overshoot in the dose delivery. In that case, healthy tissue could potentially enter the Bragg Peak, resulting in high irradiation of the tissue, which could cause harm to the patient. This illustrates that range uncertainty has a much more significant impact on proton therapy than radiation therapy.

The range calculations for proton therapy are traditionally done by performing a CT-scan of the patient, and the Hounsfield Units from the scan is converted to RSP. RSP is a measurement of the stopping power of a particle moving through water, which can be used to estimate particle dose delivery. The HU measurement has an uncertainty to it. Although the uncertainty is not necessarily increased as one converts from HU to RSP, it will have a more significant impact on particle therapy as opposed to RT[12].

A solution to avoid uncertainty in the measurement is to measure the RSP of the tissue directly instead of converting it from HU. This can be done by performing a CT-scan with the same particle used for treatment, in this case, a proton CT-scan. By measuring RSP directly with a pCT-scan, it is expected to achieve a higher range precision than the conventional conversion method used with a normal CT-scan. A study was performed to test the accuracy of a pCT-scan versus a CT-scan. A fresh post-mortem porcine structure was used to compare the RSP measurement between a direct pCT-measurement, and the RSP calculated from the CT-scan[13].

First, a measurement of the accuracy of the pCT was done by performing a pCT-scan of eight vials containing different tissue, shown in Figure 2.8. The RSP value of these different tissues was measured beforehand, with a multi-layered ionization chamber, and the RSP values from that experiment are compared with the RSP from the pCT-scan.

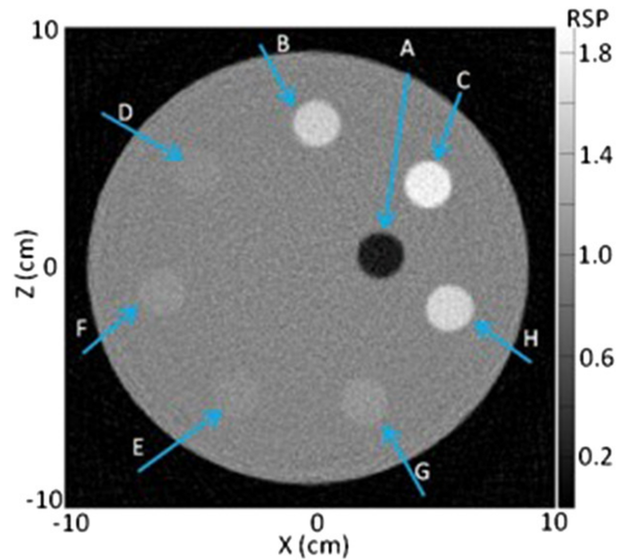


**Figure 2.8:** Image of the RSP test done on several vials containing porcine tissue[13]

The test revealed that there was up to 1% difference between the RSP measured with pCT and the results from the ionization chamber. The range uncertainty from converting from HU to RSP has been shown to be up to 3.5%[14]. This suggests that pCT yields higher accuracy measurements than a CT-scan.

Another test compared the calculated RSP data from a CT-scan, with the measured RSP from a pCT-scan of the pig's head. The result is shown in Figure 2.9.

The data reveals a relatively small difference between the calculated RSP and the measured RSP in soft tissue, with an average difference of only 2% between them. Structures such as teeth and bone show larger discrepancies between measured and calculated RSP; there is up to a 30% difference between them. The results from this test show that there is a significant advantage to measuring the RSP value directly using pCT in comparison to calculating the RSP values through converting Hounsfield units. From this, we can conclude that pCT shows potential to outperform traditional CT when measuring RSP in a patient.

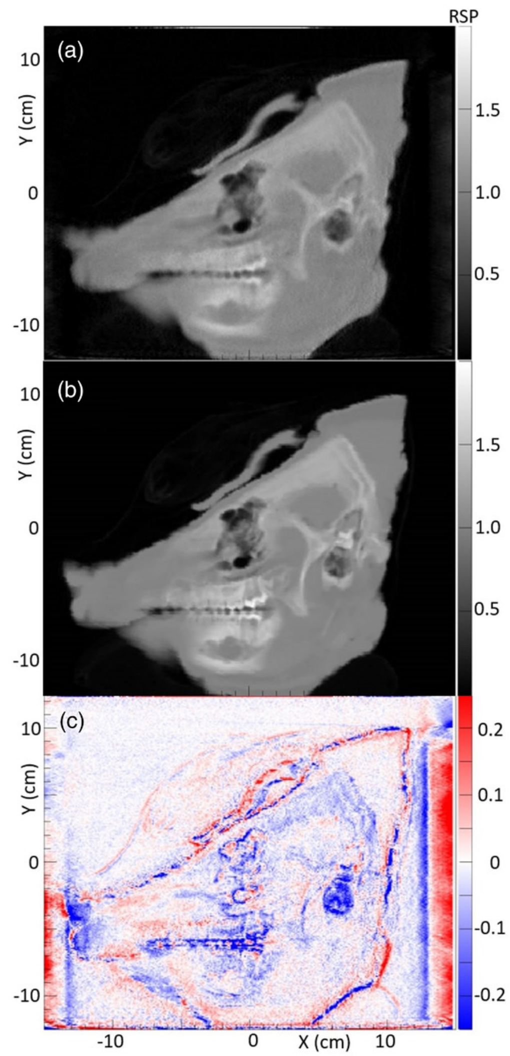


**Figure 2.9:** Image showcasing difference in measured RSP and calculated RSP of the porcine structure. (a) Measured. (b) Calculated. (c) difference between measured and calculated RSP [13].

# 3 Control System Design Methodology

*This chapter covers design methodology for control systems and software design. The focus is on the SCADA architecture, how it is constructed and the methodology behind creating such a system. Finally, a discussion is made on the methodology of software design and common ways to structure code.*

The readout units and the power supply boards require a software endpoint that controls the entire system. This entails creating a system that can set up and configure the various parts of the system, monitor the system, and report errors, i.e. we need to create a monitoring and configuration system.

## 3.1 Overview

Control systems have been a part of the industry for decades; it started in the 1960s when specialized minicomputers interfaced directly with devices in the plant or factory[15]. These systems were primarily used for automation but were highly specific and integrated into its system, using proprietary communication protocols and technology. Later, programmable controllers started to phase out minicomputers, and HMI evolved from terminals with keyboards to advanced GUIs. Ethernet eventually became the industry standard communication protocol. Finally, these technologies converged into a single system, nicknamed SCADA.

A control system is generally made of sensors, a controller to control and retrieve data from the sensors, a supervisory computer that manages the process for the sensors, and HMI software. The HMI software provides access to the data from the sensors to the user and, if needed, also allows the user to modify processes in the system.

## 3.2 SCADA systems

SCADA systems have five main functions: data collection, network data communication, data presentation, and remote monitoring and supervisory control[16]. Although today, many control systems are based on internet Of Things (iOT), which has similar architecture. However, iOT bases itself on utilizing cloud networking and processing for its communication infrastructure. Due to the lack of wireless communication in the pCT-project, the focus will be on how SCADA systems are developed, although many of the same principles apply to both.

A SCADA system typically is made out of five components:

- Field devices and signals
- Programmable Process Controller (PPC)
- Human Machine Interface (HMI) and workstation software
- Database servers
- Communication infrastructure

Field devices and signals encompass all sensors and the actuators that control them. The Programmable Process Controller (PPC) is responsible for automatically controlling the field devices, retrieving data from them and sending it to the HMI. HMI is a part of the software being executed in the workstation and is made of a user interface and database manager. The HMI serves as the interface for the user to set up configuration processes and monitor the system performance. For this purpose, a configuration database is required to set up the PPCs with correct values. A historical database is also needed to store monitoring data from field devices. A communication infrastructure connects the PPCs to the workstations running the HMI software. An image of a typical SCADA-system is shown in Figure 3.1.

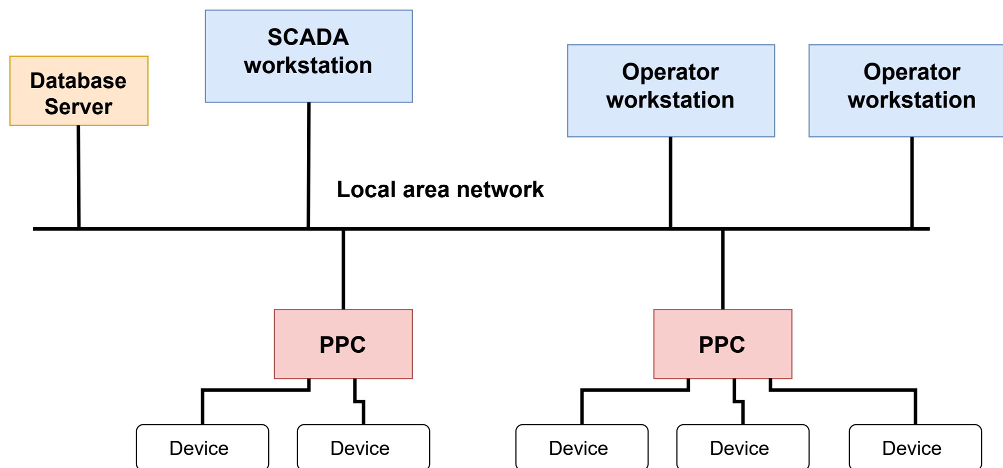


**Figure 3.1:** Block diagram of a typical SCADA system, based on diagram from[17].

The image shows field devices connected to PPCs that control them. The PPCs are connected to the SCADA HMI, or smaller operator HMIs through a local area network. The PPCs are usually connected to a Local area network, where they send and retrieve data to the HMIs. The HMI interfaces with the database server to process configuration and monitoring data, although some systems allow the PPC to directly access the database.

## 3.3 SCADA software development

The life cycle of a SCADA-system project is critical to understand to create a similar system. In addition, the methods involved in creating a typical SCADA-system ensures the software and its documentation become well organized and structured.

### 3.3.1 Input and output signals

First, the process areas and the field signals must be identified. The process area denotes a specific area responsible for one aspect of the operation. Each process area requires a PPC to manage signals and a network to connect to the rest of the system.

The field signals encompass all input and output signals in a system. Knowing all signals in a system will give us an estimate of the system's requirements and aid in the development process. The input and output signals determine the configuration and monitoring processes.

The input signals affect how the PPC will control the field devices. For example, these can be discrete values that determine if a valve is open or closed or analog signals that determine voltage thresholds in a circuit. The output signals are the values monitored by the PPC, polled by the HMI, and stored in a database. These are usually measurements from a sensor, such as a temperature, voltage, or flow.

### 3.3.2 Defining PPC operations

The functionality of the PPC should be defined and documented so the reader can understand how the monitor and control operations are performed. In addition, the documentation of the PPC aids in developing the workstation software, which must interface with it.

### 3.3.3 Developing workstation software

The software to be used in the workstations comprises two parts: the system's graphical display, which serves as the HMI, and the databases containing information of the PPCs. The graphical display must be designed so the user can quickly and efficiently survey and control the system. Security must also be considered in the design; certain functions that can change parameter values should ideally be only available to administrative users. This reduces the chance of equipment or instruments being damaged, or data being deleted by inexperienced users.

The database software must store configuration values for the PPCs and the monitoring data from them. There are usually two databases maintained for this purpose, one for configuration and one for monitoring. Communication software must be developed that regularly polls data from the PPC and inserts it into the monitoring database. Likewise, the configuration software must retrieve data from the configuration database and send the data to the PPC. The tag values used in the database should be meaningful and standardized, making it easy for users to get an overview of the data.

### 3.3.4 Software documentation

Another essential aspect of such a control system is having a consistent programming standard and a complete software documentation. SCADA and similar systems often have a long shelf time, and a well-documented system will make maintenance and work in the future more manageable. In general, starting with good software documentation and expanding upon it during the project development will lead to sufficient documentation of the entire system.

Good documentation of a SCADA-system includes:

- Description of database architecture, along with an explanation of the tag values.
- Description of the PPC logic, including functions and signals involved.
- An operational manual for the displays and functions available to the user.
- A reference of the displays, programs, and databases used in the application.

## 3.4 Software design

Software design can be a complex task when developing an extensive system. Therefore, specific measures should be taken to ensure that software development and maintenance are manageable. Using programming standards and conventions can drastically increase the readability and the ease of expanding the software. Additionally, using version control systems like Git and GitHub makes maintaining the code in a project with multiple people straightforward.

### 3.4.1 Object Oriented Programming

A consistent programming standard helps debug and expand the system's functionality. For example, Object Oriented Programming (OOP) is a prevalent programming paradigm used to structure programs and increase the code's reusability and maintainability.

OOP have four basic principles when creating classes in programs: Abstraction, Encapsulation, Inheritance, and Polymorphism. Abstraction is based on creating an object or class interface that allows an outside source to interact with the object without knowing the inner implementation of the class. This reduces the program complexity and serves as an intuitive segmenting of the software.

Encapsulation declares that only the class should be able to manipulate and interact with its private variables. This assures that the classes are not dependent on other class implementations, which increases the modularity of the software. Finally, Inheritance and Polymorphism are closely related. They describe how classes can be derived from other classes, "inheriting" their functions, and how the inherited functions can be redefined, altering the class's behaviour.

These principles guide the developer to create modular and structured software which is reusable, easy to maintain, and modify. Abstraction is the most important principle, and encapsulation relates to how to create a good abstraction of the system. These two principles will be used the most in the work of this thesis. Systems designed to manage large amounts of data acquisition, such as a Positron Emission Tomography (PET)-scan system, have in the past used OOP to design and manage large programs[18]. This gives more credence to use OOP in similar projects.

### 3.4.2 Git and GitHub

The use of the Git and GitHub tools is essential today to manage the project code and maintain it as a version control system. The version control system is essential for verifying the code. no matter what happens, a stable version of the software will always be available in previous versions on Git or GitHub. GitHub also enables a better workflow with co-developers or mentors, allowing easy access to the source code and giving feedback.

The GitHub repository used for this thesis contains a README file containing all practical information about the repository and the project's goals. This includes information about the configuration and monitoring process and a user guide to start the databases and GUIs. An example of the README file is given in Figure 3.2.

## Configuration and Monitoring

### Configuration

the microcontrollers have 8 registers that should be able to configure through the GUI:

| Register | Comment |
|---|---|
| DVDD threshold 1 | warning threshold |
| DVDD threshold 2 | error threshold |
| AVDD threshold 1 | Warning threshold |
| AVDD threshold 2 | error threshold |
| PWELL threshold 1 | Warning threshold |
| PWELL threshold 2 | error threshold |
| Temperature limit | thermistor value |
| Enable signal | each bit corresponds to one string |

*db_manager* class sets up and store configuration sets, *config_db.json* gives an example of how the different sets are stored in the database. The configuration API is a high level interface built on top of the microcontroller API. The hierarchy is as follows:

**Figure 3.2:** Example from the README file showing the overview of the configuration process.

# 4 The PCT Project

*This chapter covers the PCT-project that is currently under development at the Department of physics and technology in Bergen. It discusses the DTC being developed for the project, and how it tracks and measures the energy protons. Finally, a technical overview of the systems comprising the pCT-system.*

The Bergen pCT-project is a collaboration between the University of Bergen and several other institutions worldwide. The purpose is to build a pCT-scanner to be used in measuring RSP. The design is based on using a Digital Tracking Calorimeter to detect protons emitted from a particle accelerator. In particular, the focus is on creating a scanner for use on pediatric patients.

## 4.1 Introduction

The pCT-project comprises several control systems and sensors. The main instrument is a DTC that uses ALPIDE-chips to measure the energy of protons. The DTC is made out of 43 layers of ALPIDE sensors and there are three control systems that oversee the DTC. These systems are cooling, readout, and power delivery. The cooling system is connected directly to the DTC, while power delivery and readout perform their actions through a TC. All systems are connected to a network that allows them to communicate with the Control Room.

The pCT-system is still under development, Figure 4.1 shows the current implementation of the system.

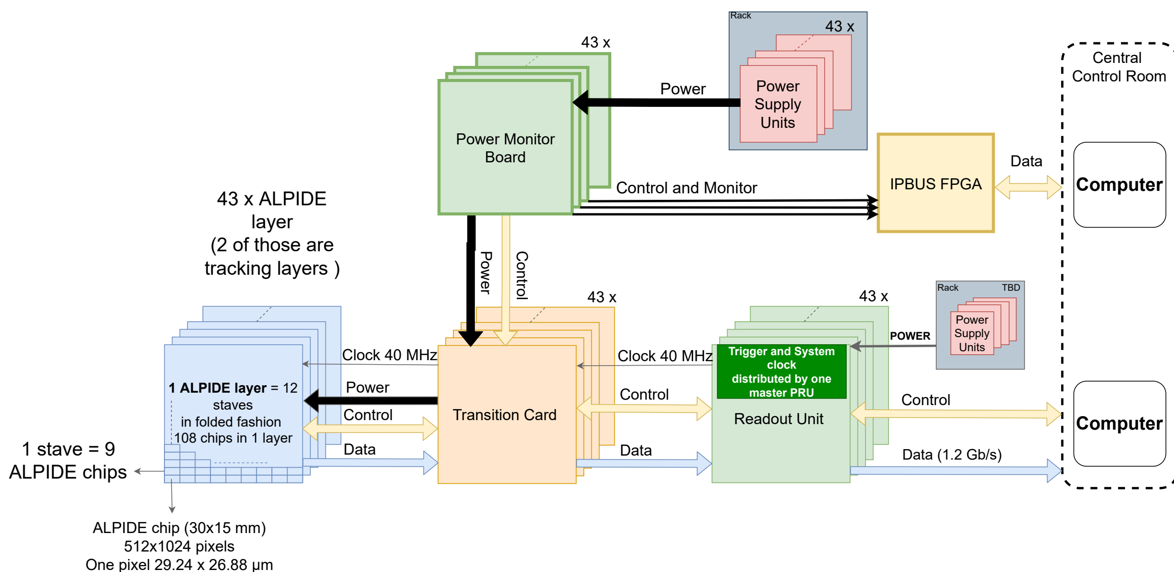


**Figure 4.1:** Block diagram of the current design of the pCT system, without the cooling attached. The upper part shows the power delivery and monitoring system, and lower half is the readout unit.

## 4.2 Digital Tracking Calorimeter

The DTC is the main instrument of the pCT-system; it combines proton tracking and energy measurement into one technology. It is made out of 43 layers of pixel detectors, where two layers in the rear are used to track the trajectory of the protons and the other 41 measures their energy. A model of it is given in Figure 1.2.

A DTC is usually realized with front and rear trackers, but only front trackers are used in this prototype. Studies have shown that single-sided proton imaging using Pencil Beam Scanning can lead to similar spatial resolution in the image as compared to double-sided imaging[19]. The single-sided tracker also comes with a few advantages, such as a lower material budget and less complexity in the rigging of the DTC.

The layers of DTC are functionally identical. However, the calorimeter layers are made with aluminium absorbers to fully contain the energy range from the proton beam, estimated to be 230 MeV. The tracking layer must have as small a material budget as possible to minimize the scattering of the protons, which can affect the accuracy of the scan. The tracking layers are used to track the angle of the protons that exit the patient; the two layers provide the linear path of the proton, which is used to derive the angle. Based on the angle, one can statistically estimate the most likely path the protons took as they moved through the patient. This statistic is crucial for reconstructing the CT image.

A layer of the DTC is realized using 12 strings of ALPIDE-chips. A layer comprises two half layers, and a half layer comprises two "slabs" of ALPIDE-chips. The "slabs" are three strings glued together; a picture of a half layer is shown in Figure 4.2.

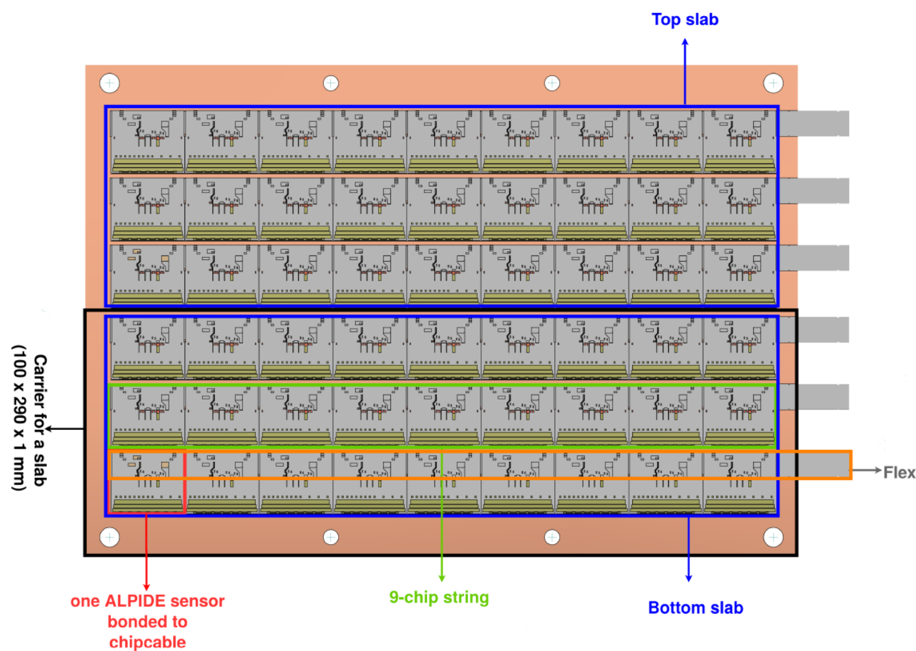


**Figure 4.2:** Image of half sensor layer, highlighting various components[20].

We can note from the half-layer design that the ALPIDE-chips only covers about half of

the layer; the rest of the space is dedicated to the flex cable. Therefore, two half-layers are stacked so that the ALPIDE-chips of one half-layer cover the flex cable of the other, ensuring full area coverage. This gives us a full layer of the DTC.

## 4.3 ALPIDE chips

The basic pixel detector sensor used for this project is the ALPIDE-chips that were designed for the upgrade of the ITS during the long shutdown of the Large Hadron Collider (LHC) during the 2019-2020 period. The chips are categorized as Monolithic Active Pixel Sensor (MAPS) with a span of 1.5 cm x 3 cm. The sensor is made of a matrix of 512 x 1024 pixels, which function as binary hit/no hit sensors. A detection threshold is set for the chip, suppressing all hits below the threshold level and functioning as a filter for non-relevant data, such as noise and radiation. A cross-section of the ALPIDE silicon is shown in Figure 4.3.

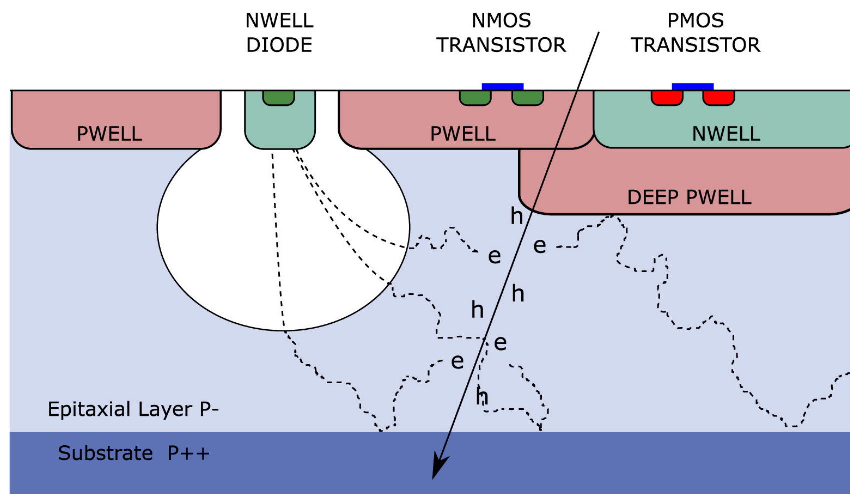


**Figure 4.3:** Cross section of an ALPIDE-chip, showing a particle entering the silicon and causing a hit to register.

The figure shows a particle entering the silicon and releasing electron-hole pairs from the epitaxial layer. The charge moves through the depletion region and if it reaches the set voltage threshold, a hit is registered in the pixel.

In the pCT project, nine ALPIDE-chips are bonded together on one flex cable, which is defined as one ”string”. An ALPIDE string is shown in Figure 4.4.

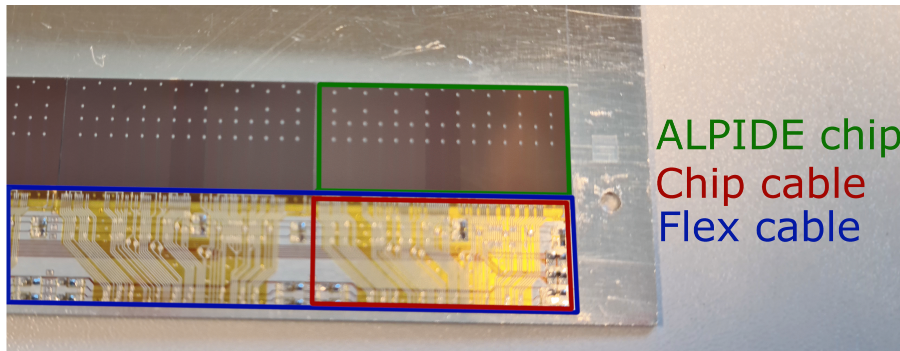


**Figure 4.4:** Image of parts of a string, highlighting the ALPIDE chip, chip cable, and flex cable.

The figure shows that half of the string is covered by the chip, and the other half is the flex cable used to mount the chips.

For this thesis, it is also relevant to look at the power consumption of the strings. The ALPIDE-chips do not all draw the same amount of current. It is therefore necessary to measure the current draw of the strings. A test was performed on a string at the University of Bergen, and it was recorded to draw 880 mA of digital current, and a maximum analog current of 200 mA is also expected. With these measurements, each string requires 1.9V and 1.1A to operate.

## 4.4 Cooling System

The cooling system consists of two parts, one for the front tracking layers and one for the calorimeter layers. The front tracking layers are liquid-cooled, while the calorimeter layers are cooled using five fans. Figure 4.5 shows lab setup of the cooling system.

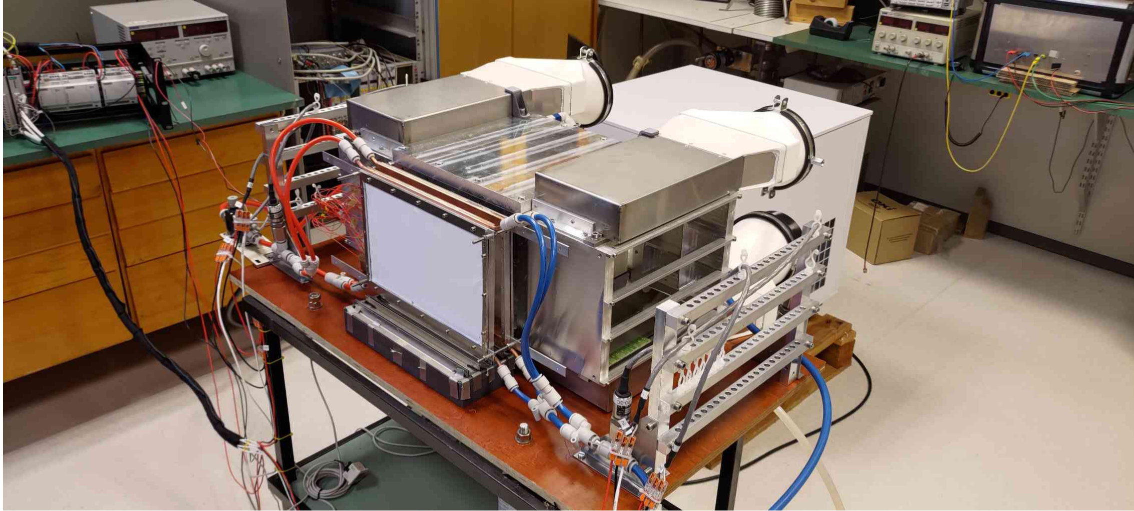

**Figure 4.5:** Image of the cooling system in the lab. The front trackers are liquid-cooled and are connected to two flow meters. The fans behind the box serve as the cooling for the calorimeter layers.

The cooling control is done through moxa cards, which are connected to the Control Room. The cooling system is monitored with four temperature sensors and two flow meters, and it can also monitor the RPM of the five fans. The control system for the cooling system is still under development; currently, it can only read out values from the sensors.

## 4.5 Readout System

The pCT readout system comprises 43 readout units, which act as the communication hub between the Control Room and the sensors. Each layer has a corresponding pRU, which controls the sensors as well as manages the data streams. The pRU board is realized using an Kintex Ultrascale KU085 FPGA, and the board is connected to the TC using 12 Samtec FireFly cables. A block diagram of the board is shown in Figure 4.6.

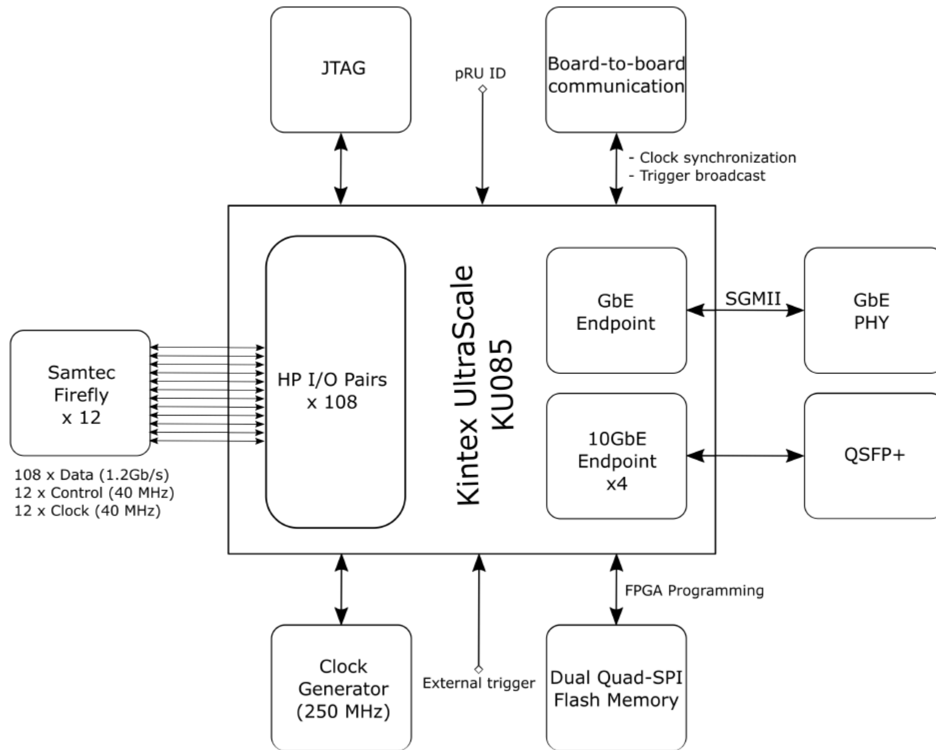


**Figure 4.6:** Simplified block diagram of the pRU board[21]

There is no monitoring system for the pRU, but a configuration system exists, which can store configuration values in a database and configure the strings. However, the configuration system requires a new revision to be able to integrate it into the full control system.

## 4.6 Power Delivery System

The power delivery system is responsible for powering the strings on the DTC and monitoring the strings' performance. Figure 4.7 outlines a detailed block diagram of the system.

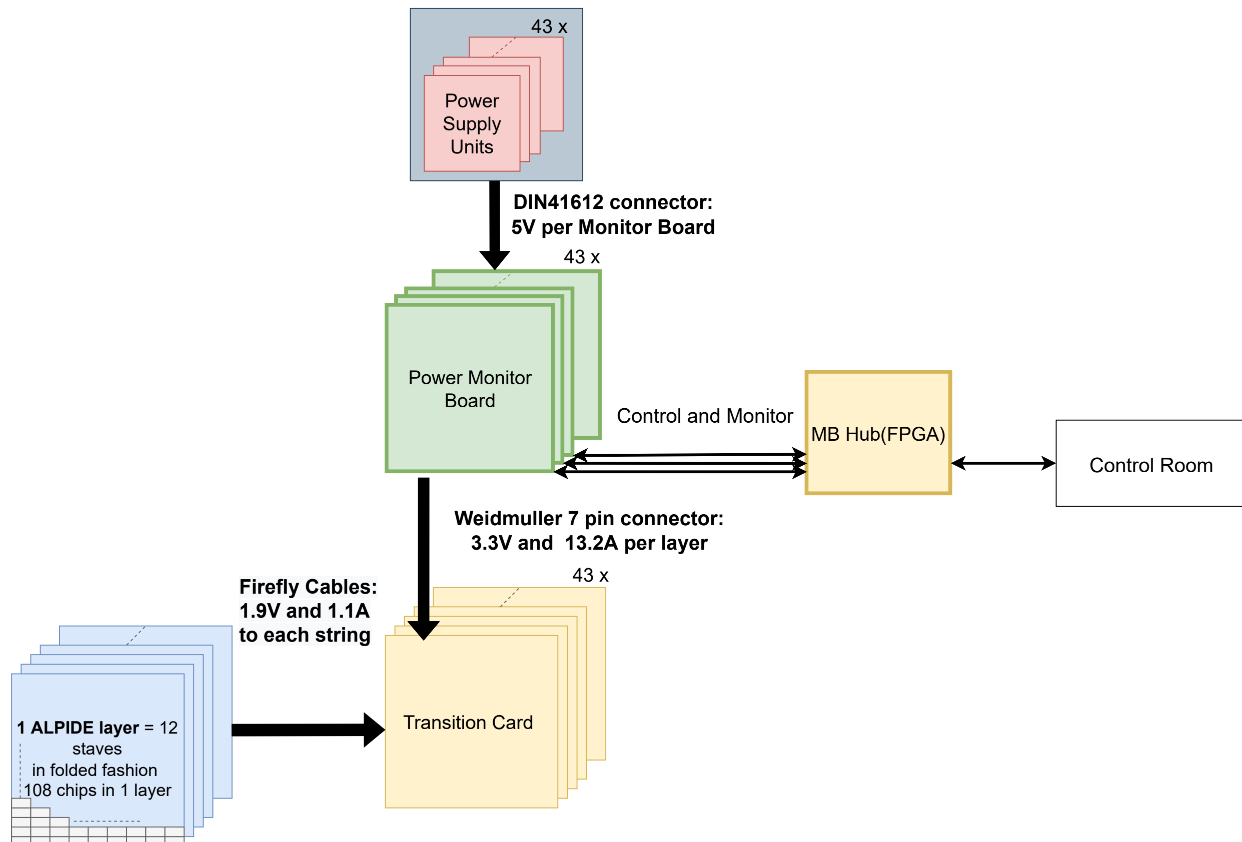


**Figure 4.7:** Detailed block diagram of the power delivery system.

Control of the system is done through the Control Room, which uses the MB Hub to interface with the MBs. The MB delivers power from the PSU to the TC. It also monitors the temperature and current usage of the strings. A string is turned off by the MB if it exceeds the current/temperature threshold. This ensures that the strings are turned off quickly so as to not damage the chips. The MBs deliver 3.3V to each TC, which uses linear voltage regulators to shift the level down to 1.9V before delivering it to the strings.

The control system of the power delivery is discussed in section 5, where the focus is on the communication chain between the Control Room and the MBs.

# 5 Power Control System

*This chapter describes the Power Control System, how it is constructed and how it functions. It details the three parts that make up the PCS, which are the control software, the MB Hub and the MB. It additionally describes the IPbus communication protocol and how it is implemented in the control software.*

The PCS is the name of the system that delivers power to the ALPIDE-chips and monitors their performance. The system is made of three parts, the control software on the computer, the MB Hub, which is implemented on an FPGA, and a Monitoring Board. The MB functions as the PPC of this control system, controlling the power to the strings and monitoring the strings' temperature and current consumption. The MB design and microcontroller software was developed by Birger Olsen. The MB Hub is a translation layer between the control software and MBs; it uses IPbus firmware to safely and reliably transmit data packets from software to MB. A single FPGA is used to send data to all 43 MBs, meaning it must have the architecture to process data packets for every board. Martin Eggen and Jakob Hauser developed the FPGA design. The control software comprises a monitoring and configuration system responsible for configuring the MBs through the FPGA and monitoring the strings' temperature and current consumption. The control software serves as the HMI and database system for the PCS. This control system on the software side will be discussed more in section 6 and section 7. A figure of the PCS is shown in Figure 5.1.

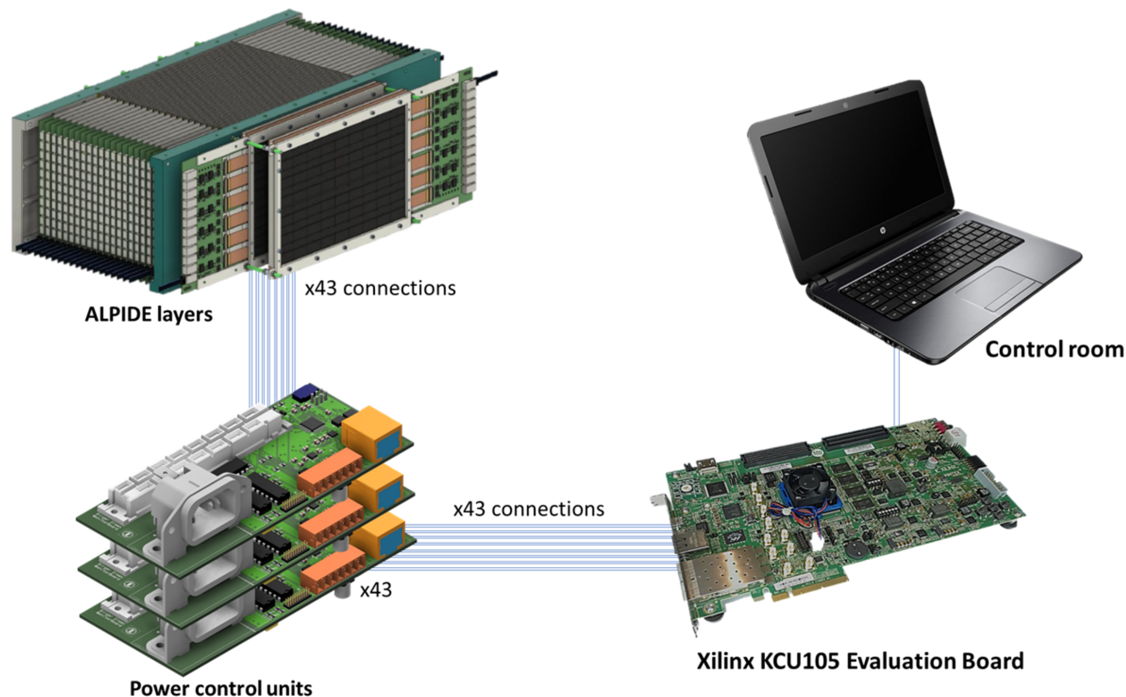


**Figure 5.1:** Overview of the PCS components. Note that the connection between the evaluation board and the MBs is through USART. The MBs delivers power to the sensors through a 7-pin connector[22].

The "Control Room" is the computer with the control software that can communicate with

the MB Hub. The "Control Room" will handle both the PCS and the readout of the ALPIDE-chips. We currently have a prototype implementation of the PCS, with a computer connected to a KCU105 evaluation board. The MB is still in development, so a Curiosity Nano development kit was used to verify the connection between the KCU105 and microcontroller software. An image of this setup is shown in Figure 5.2.

**Figure 5.2:** Image of the prototype of the PCS communication chain.

There is no connection between the microcontroller and the DTC yet, and only one "layer" is connected. However, it still allows us to do performance tests that will give us reasonable estimations of the speed and error rate of the final product. These tests are discussed more in section 8.

This chapter describes the different design approaches for each part of the PCS, focusing on the system's requirements and how it is implemented.

## 5.1 Control software

The control software comprises APIs, GUIs, a monitoring system, and a configuration system. The APIs and the GUIs together form the HMI software, which allows the user to interact with the control software and oversee the PCS. The configuration system performs the necessary steps to power on the strings quickly and safely, and it configures the microcontroller on the MB. The monitoring system retrieves the monitoring data from the microcontroller and stores the data in the databases on the client side. Finally, it displays the data to the user in an easy-to-understand manner. An overview of the PCS, emphasising the control software, is shown in Figure 5.3.

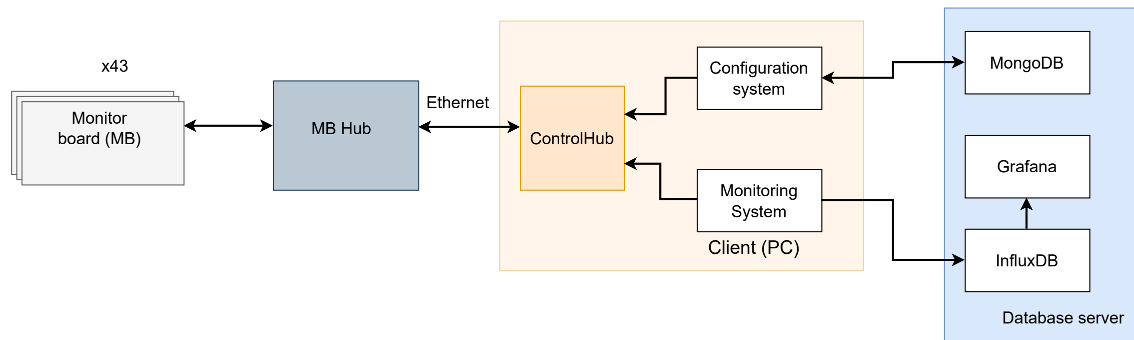


**Figure 5.3:** Overview of the PCS, emphasising the control software.

The configuration system and monitoring system are made of custom Python classes that use the IPbus API to communicate with the ControlHub, which in turn communicates with the MB Hub over an Ethernet cable. The figure also shows the databases the two systems use to store data; MongoDB stores configuration sets, and InfluxDB + Grafana stores monitoring data and displays it to the user. These two systems will be discussed further in section 6 and section 7.

The control software functions as the brain of the PCS. The MB Hub and the microcontroller on the MB are simple by design, and more complicated processes are meant to be performed by the software. Processes such as debugging for faulty strings or transmission tests are created on the software side. Testing and verification of the PCS is discussed in subsection 8.4.

Both the monitoring and configuration systems use GUIs to interface with the user, and a top-level GUI unifies both systems. An image of the top level GUI is shown in Figure 5.4.

The main hub GUI serves as the top level for the control software; this means the user can start the GUI and access all functionalities of the control software. This includes setting and creating configuration sets using the configuration GUI, starting and stopping the monitoring process, opening the Grafana web tool, and starting the configuration and powering algorithm.

The main hub GUI has buttons that will open up the configuring and monitoring GUIs, and it also has a button that will initiate the configuration process. Additionally, the window has a logging window to display the status and actions performed by the user.

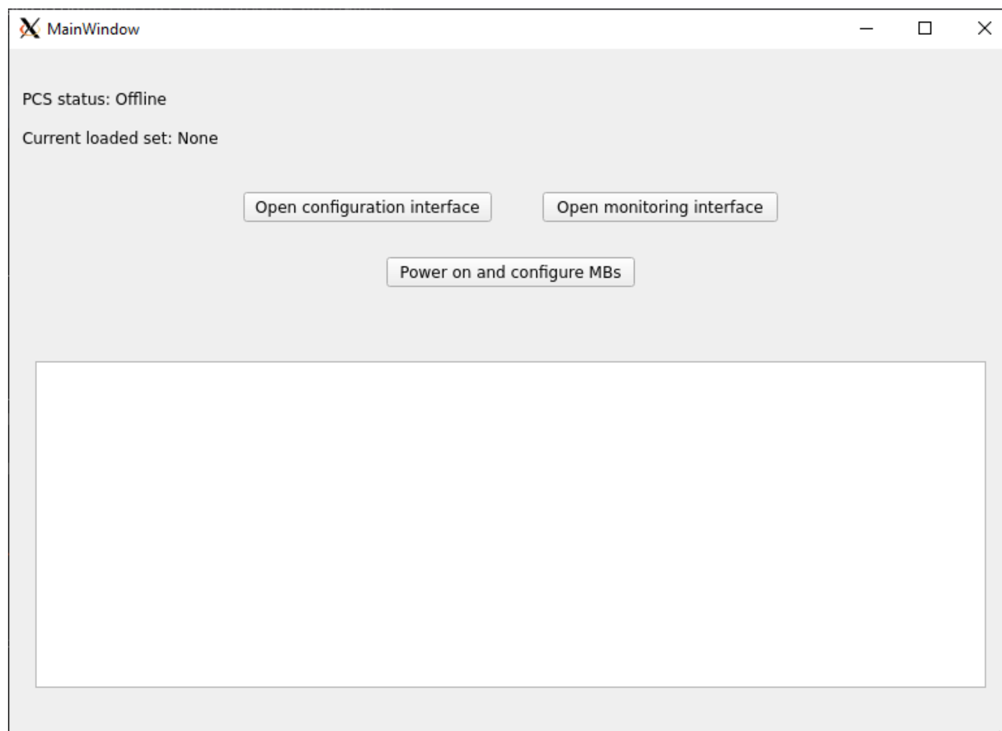


**Figure 5.4:** Image of the main hub GUI. The GUI has buttons for opening the monitoring and configuration interfaces, starting the configuration of the MBs, and a logging window for displaying messages.

## 5.2 IPbus

### 5.2.1 Overview

IPbus is a simple, packet-based control protocol for reading and modifying registers on an FPGA using an Ethernet connection. An Ethernet connection using IPbus protocol forms the network in the PCS. It is designed to be a high-performance control link for particle physics electronics; it is mainly used in the LHC experiments at CERN.

IPbus consists of software and firmware that together forms a control link from the computer to FPGA based hardware:

- IPbus firmware: IPbus module implemented in FPGA based hardware that handles reading and writing to registers.
- $\mu$HAL: C++ and Python API for interfacing with the IPbus firmware and perform read/write operations.
- ControlHub: Software application that serves as a hub for multiple $\mu$HAL interfaces.

The IPbus firmware is designed to be easy to implement, and there are example designs for several FPGA boards available which can be used. One of the reasons we chose to use the KCU105 board for this project was because it had an example design that could be repurposed for the pCT-project.

$\mu$HAL is the control software's interface to communicate with the IPbus module on the FPGA. This interface allows for issuing read and write requests and dispatching them to the FPGA registers. For PCS, we will use the $\mu$HAL Python library to interface IPbus with custom Python classes. The ControlHub software serves as a common access point for multiple $\mu$HAL entities on the software side. This is useful in larger systems where multiple systems use IPbus to transmit data. Figure 5.5 shows ControlHub used in a typical control system.

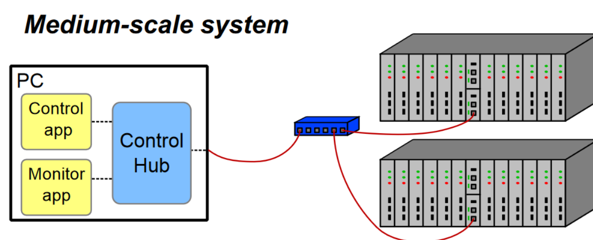


**Figure 5.5:** Example of a system using ControlHub to connect a control and monitor system to the hardware[23].

PCS requires a monitoring and configuration system, so it is natural to have a separate $\mu$HAL entity for each system and use ControlHub as a single access point for the IPbus module.

### 5.2.2 IPbus software

The IPbus software uses XML-files to store the address map of modules on the FPGA, as well as the register map of said modules. The address map of the FPGA design is listed in Appendix A.2. The address map contains the address value for each module, while the register map contains more detailed information about the registers. The individual module address maps are stored in separate XML-files. the XML-file for the com_modules is shown in Listing 5.1.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<node id="monitor_module">
    <node id="control" address="0x00000001" permission="w" description="not in
       use." parameters="reset=0x5"/>
    <node id="status" address="0x00000002" permission="r" description="not in
       use" parameters="reset=0x0"/>
    <node id="TX" address="0x00000003" permission="w" description="Writes data to
       corresponding MB" parameters="reset=0x0"/>
    <node id="RX" address="0x00000004" permission="r" description="Reads a 32-bit
       vector from the corresponding MB" parameters="reset=0x0"/>
    <node id="reset" address="0x00000005" permission="wr" description="Resets
       both TX and RX FIFO to clear them" parameters="reset=0x0"/>
</node>
```

**Listing 5.1:** XML code for the com_module used within IPbus software

The files contain information about each register, such as address value, Read not Write (RnW) access, and description. IPbus XML-files support additional functions, like masking bits or writing to a memory block, which is described in the IPbus user guide[24], but for this project such functionality is not necessary.

The $\mu$HAL Python library is vast, but for the end user, there are only three essential classes:

- ConnectionManager
- HwInterface
- Node

ConnectionManager contains the HwInterface object for each connected IPbus module and IP address information. The HwInterface class is used when communicating with the IPbus module. HwInterface contains all node classes for each module defined in the address map. The node class is analogous to the physical registers on the FPGA; they contain functions for reading and writing to the physical register the node represents. Listing 5.2 shows how one typically uses HwInterface to read and write from registers.

```
#Initialize HwInterface
manager = uhal.ConnectionManager("file://../bsp/pRU-connections.xml")
hw = manager.getDevice("demo_layer")

#Perform a write operation
hw.getNode("monitor_module_1.TX").write(0b010101010)
hw.dispatch()

#Perform a read operation
RX = hw.geNode("monitor_module_1.RX").read()
hw.dispatch()
```

**Listing 5.2:** Example of setting up the HwInterface in python and using it to perform read and write operations.

A read or write is performed by retrieving the node for a register and issuing a write or read from the node object. Note that the read or write functions only puts the request in a packet; it does not send it. It will only send the packet of requests when the dispatch function is called. This reveals a crucial aspect of IPbus, it can send data packets simultaneously. IPbus has a single-word latency of 250 $\mu$s when transferring data, which is significantly larger than other control protocols. However, by concatenating the data into larger packets before sending them, IPbus can increase its throughput up to a factor of 2000[23]. This means that IPbus performs optimally if we limit network dispatches as much as possible.

The conclusion is that we must design the control software around limiting dispatches. This allows us to fully utilize the IPbus throughput and ensure that our system is quick and responsive. Subsection 6.1 discusses how the API for the FPGA is designed to allow for sending multiple data packets in one transmission.

## 5.3 MB Hub

A communication hub between the control software and the MB is necessary to communicate efficiently with the MBs. A previous paper done for the pCT-project discussed different design architectures for the Detector Control System (DCS), which encompasses both chip readout and the PCS[21]. A microcontroller-based architecture was considered, but it was proven too large of a bottleneck in the system. A pure FPGA architecture was seen as the superior design for the system. One of the advantages of using an FPGA-architecture is the use of IPbus.

An FPGA must be chosen to be used in the PCS, and there are specific criteria the FPGA must meet for this project:

- It must have $2 \cdot 2 \cdot 43$ I/O pins, to communicate with all 43 layers using LVDS.
- The FPGA must be relatively new and still updated, so it will not become obsolete in the coming years.
- It must not be too expensive, although this is not a hard requirement, due to only needing one FPGA for the entire system.
- The FPGA should have an example design for IPbus available, this will cut the development time for the FPGA design by a large margin.

For this project, we chose to use the Xilinx Kintex UltraScale KCU105 to meet the demands for the number of I/O pins and have an example IPbus design available provided by the developers of IPbus. It was decided to use the evaluation board for the hardware implementation since only one FPGA is needed for this project.

The FPGA will be responsible for communicating with all the MBs. It needs to handle individual read and write for each layer and have global functions for broadcasting messages. It also handle errors in case something goes wrong on any of the MBs.

The FPGA design comprises 43 com_modules, each containing two registers for communicating with the MB, TX and RX. Each module would only be responsible for sending and receiving data through the TX and RX registers, no direct information of the MB is stored in the FPGA. This means less data is copied, reducing the chance of storing erroneous data. The modules also need a status register to track eventual errors from the MB. Martin Eggen and Jakob Hauser implemented this design. The com_module and global module address map are given in Appendix A.2.

A figure of their implementation is shown in Figure 5.6.

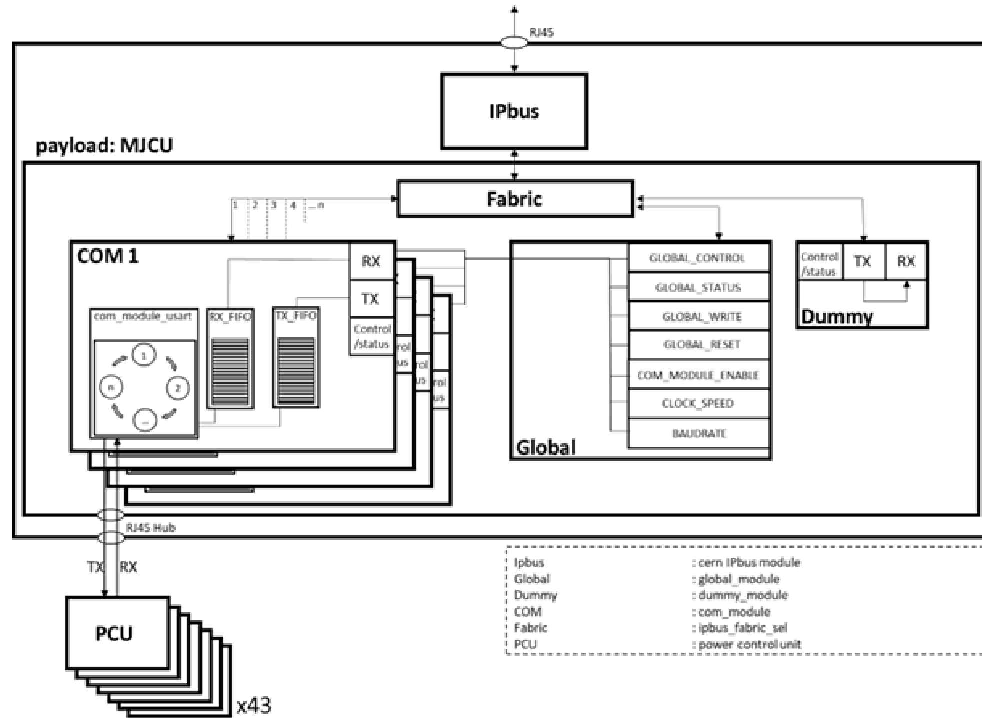


**Figure 5.6:** Block diagram of the MB Hub FPGA design. Note that "PCU" is meant to be the MB in the diagram[22]

This design has a "com_module" for each layer that handles write and read transactions to and from the microcontroller using a standard Universal Synchronous Asynchronous Receiver Transmitter (USART) protocol. In addition, these modules use a First In First Out (FIFO) to allow many data packets to be sent in quick succession without needing to wait for the transmission to finish.

The global module has several functions for configuring and transmitting data within the MB Hub. The global module has registers that define the baud rate used, which com_module is enabled, and a global reset of the com_modules. One of the crucial features of the module is the broadcast functionality. Writing instructions to all 43 com_modules is repetitive and takes more time to perform. The broadcast function solves this by simply writing one message to the broadcast TX register, which will write the same message to the TX register of each com_module. This feature is utilized in the monitoring processes to optimize the data acquisition. Finally, the global module contains an enable register for the com_modules on the FPGA. The enable register allows us to turn off com_modules that are not in use, which helps reduce the amount of redundant data retrieved by the control software.

Interfacing with the FPGA design requires us to know the operations performed by the FPGA when reading or writing to the microcontroller. For example, when a data packet is sent to the TX register of a com_module, the FPGA will send it down to the microcontroller and information from the transaction is always sent to the respective RX register. Figure 5.7 shows the data flow during a read and a write between the FPGA and microcontroller.

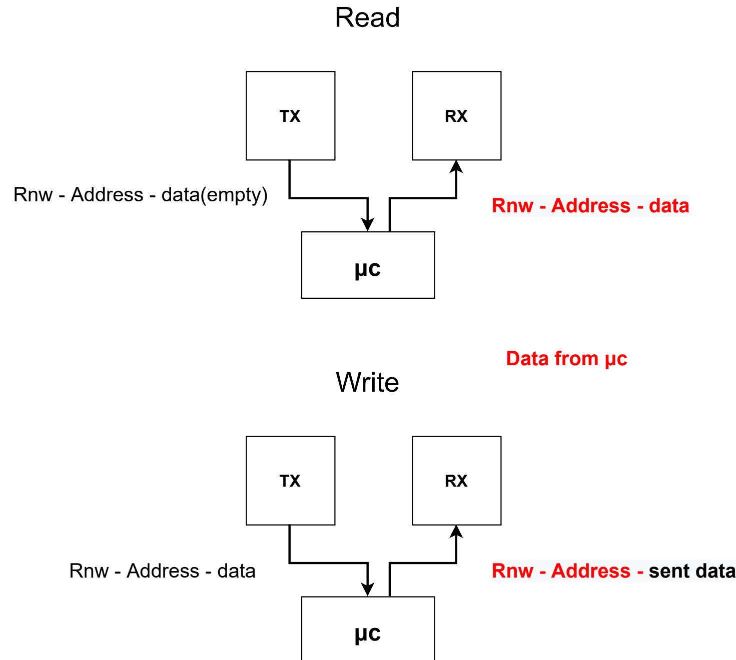


**Figure 5.7:** Block diagram data flow between FPGA and microcontroller showing handshake values in red.

When performing a read, the data is sent to the RX register, including which address was read, and RnW bit. When performing a write, handshake values from the microcontroller are inserted into the RX register, and the data is sent from the TX register. It is important to note here that the data value from the write operation originates from the FPGA, not the microcontroller.

All data written to the TX register of a com_module is sent down to the microcontroller through 2 Low Voltage Differential Signaling (LVDS) pins, using USART. The communication between is described in the timing diagrams of Figure 5.8 to Figure 5.10[25].

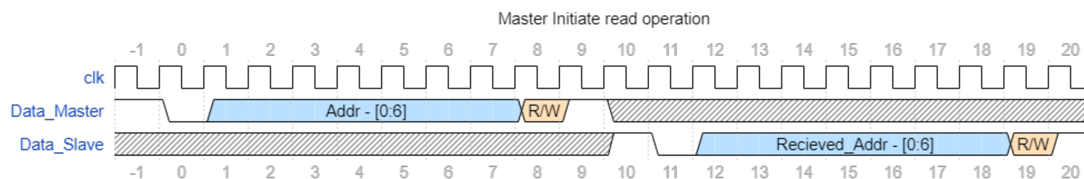


**Figure 5.8:** Timing diagram of Master initializing communication with slave using USART.

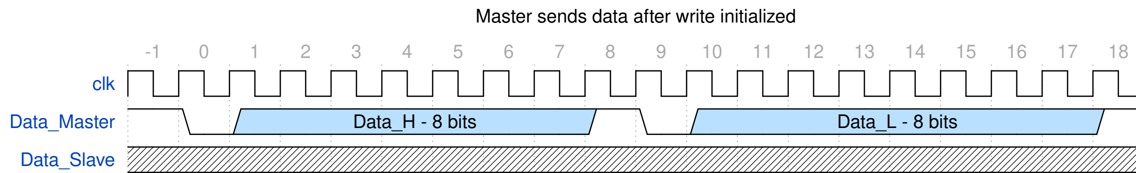


**Figure 5.9:** Timing diagram of Master sending 16 bit data through two packets.

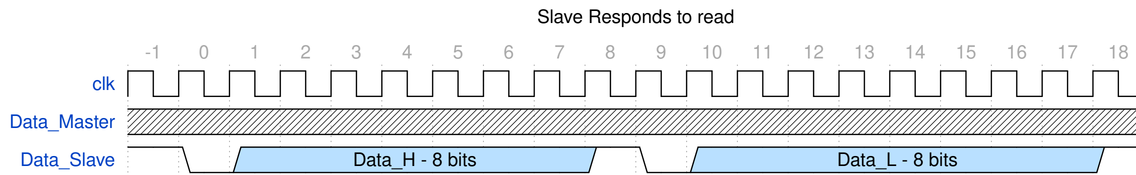


**Figure 5.10:** Timing diagram of Slave sending data after read request.

The master initiates an operation with a start bit and sends the address value and the RnW bit. Then, the slave performs a handshake by sending the received address back to the FPGA. The data is always sent with two packets of 8 bits, starting with the high bits and then the low bits. This is done due to the limitations of the microcontroller being 8-bit.

## 5.4 Monitoring Board

The MB is the equivalent of the PPC in this system, responsible for distributing power to the strings and directly monitoring their voltage levels, current consumption and temperature. The board and microcontroller software were developed by Birger Olsen at the same time as this thesis was written. This section focuses on the microcontroller software and its interaction with the MB Hub. The board delivers DVDD, AVDD, and PWELL current to the strings. Additionally, the board measures the current consumption, voltage levels and temperature of the strings. Figure 5.11 shows a simplified block schematic of the board.

An INA3221 chip measures the current consumption of the strings, and the temperature is measured through a PT1000 element. The system on the MB is controlled by an AVR128DB48 microcontroller. As mentioned in subsection 5.3, USART is currently used for the communication between the MB Hub and the microcontroller.

The register map for the microcontroller is given in Appendix A.1. The address map consists of threshold level registers, general purpose registers, control registers, and error handling registers. The registers on the microcontroller are, by default, 16-bit.

The data header used to communicate between the Control Room and microcontroller is 32 bits long and is made of RnW bit, address, and data to write. The data header is shown in Figure 5.12.

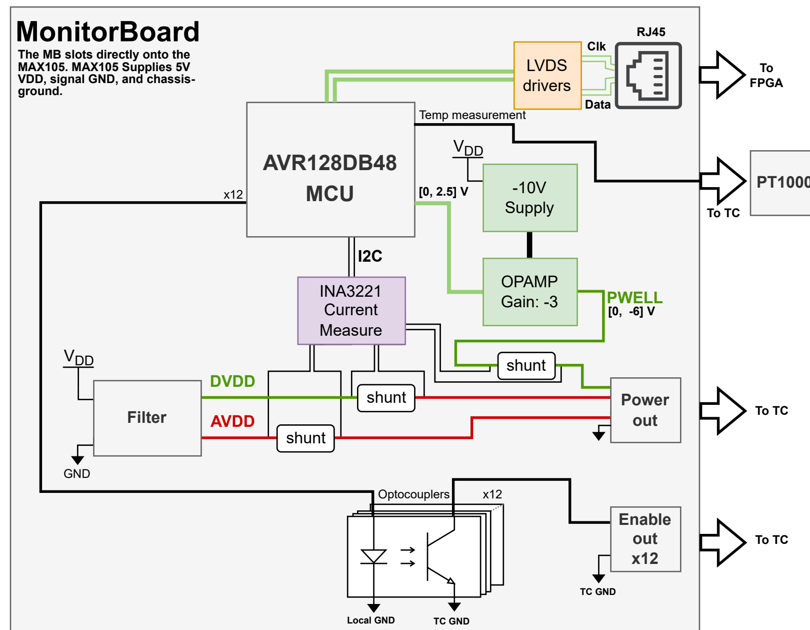


Figure 5.11: Simplified block schematic of the MB[25].

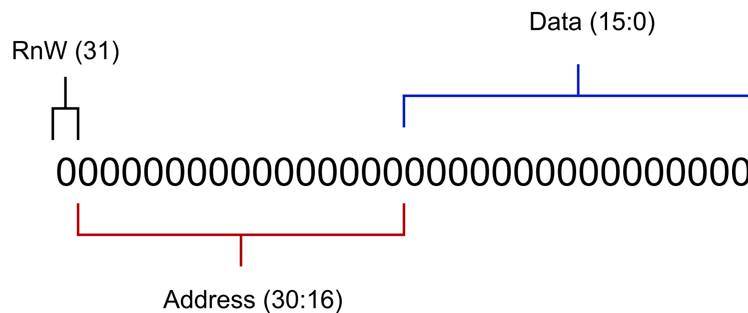


Figure 5.12: Data header for communicating with microcontroller.

The data bits only have a purpose during a write operation; the 16 least significant bits are not used during a read request.

The board measures DVDD, AVDD, PWELL, and temperature, and the microcontroller has threshold registers for each of these measurements. The microcontroller will turn off the strings if the measurements exceed these thresholds. Setting up these registers with appropriate values is a part of the configuration procedure, which is discussed more in section 6.

The microcontroller has three control registers; writing to the registers allows control software to execute custom functions built into the microcontroller. This would allow us to perform relatively complex functions without transmitting many data packets through the PCS. Functions such as checking for abnormal current consumption in the strings could be performed significantly faster with a custom function on the microcontroller. The control register feature is only in a conceptual state, and it has not been fully implemented in the current microcontroller design. CTRL1 register has functions for resetting register values

and scanning for strings with high current consumption; CTRL2 and CTRL3 are reserved for future use.

The microcontroller has two registers for managing errors, "error count" and "error message". These registers keep track of the number of errors that have occurred and the error codes. Each error code corresponds to an error message given in Table 5.1.

| Error name | Error code | Description |
|---|---|---|
| Temperature limit reached | 0x02 | The ADC value is reported to be above the temperature set by the TEMPERATURE_LIMIT register. |
| DVDD critical current | 0x01 | Critical current reached on DVDD line. |
| DVDD warning current | 0x03 | Warning current reached on DVDD line. |
| AVDD critical current | 0x04 | Critical current reached on AVDD line. |
| AVDD warning current | 0x05 | Warning current reached on AVDD line. |
| PWELL critical current | 0x06 | Critical current reached on PWELL line. |
| PWELL warning current | 0x07 | Warning current reached on PWELL line. |
| Write/Read denied | 0x08 | Tried to write or read a register that should not have been read or written. |
| String current error | 0x09 | Large current draws from the strings recorded. |
| Enable scan error | 0x0A | Large current draws from a single string recorded during the enable scan. |

**Table 5.1:** Error codes belonging to the microcontroller software.

The error message register can be reset by writing 0x00 to the error count register.

The correct procedure to read all errors would be first to read the number of errors from the error count register. The number of errors determines how many bytes of the error message register to read; when that is done, the procedure writes 0x00 to the error count register.

# 6 Configuration System

*This chapter covers the design of the configuration system used in the power delivery system. It covers the design of the configuration API and the layers of abstraction used in the lower-level API for the PCS. It then covers the design requirements of the configuration system. It also discusses the theoretical configuration timing of the system, along with the measured timing performed on the prototype setup.*

Seven registers on the microcontroller determine the threshold values for DVDD, AVDD, PWELL, and temperature. The configuration system must be able to configure the threshold registers quickly, and use the enable signals to power on the strings.

Each layer can have different nominal threshold values for voltage and current and therefore it is necessary to be able to change these values as we test the strings. A complete set of threshold values for all strings in every layer of the DTC will be referred to as a configuration set. The configuration sets must be stored in a database, and MongoDB was chosen for storing the sets; the database is discussed more in subsection 6.3.

All this leads to a requirement for a user-friendly configuration system that can configure the microcontrollers through IPbus, store configuration sets, and power on the DTC. Furthermore, these parts must be developed modular and generic to make the system scalable and easy to modify.

## 6.1 Configuration API

IPbus sends and retrieves information between the control software and the MB Hub. IPbus has a software library, $\mu$HAL, that provides end-user API for C++ and Python. For this project, Python is used to interface with uHAL and IPbus. This API can issue read and write commands to the IPbus module on the MB Hub and dispatch these commands in a single packet. Subsection 5.2 provides a detailed discussion of the IPbus software and how to implement it in Python. $\mu$HAL allows us to send and receive data, but this API is not sufficient for communicating with the microcontroller; there must be an additional interface that has functions for our FPGA design, and the microcontroller on the MB. Following the principles of OOP, it is natural to encapsulate the properties of the API into two interfaces, one for the MB Hub and one for the microcontroller on the MB. Additionally, the configuration system requires several high-level processes to operate, and therefore a separate configuration API is required.

### 6.1.1 IPbus API

The IPbus API contains functions for read and write commands using $\mu$HAL, broadcast functionality using the global module, and enabling/disabling the com_modules for individual layers. This interface Python class is named "ipbus low interface".

An essential part of the API is the "issue" functions. Subsection 5.2 discusses how IPbus benefits from packing data in larger payloads before dispatching them, which is the purpose of the "issue" functions. The "issue" functions loads the data request in a payload without

sending it to the MB Hub. This can optimize the communication chain between the control software and the microcontroller by limiting the number of network dispatches performed.

### 6.1.2 Microcontroller API

The API for the microcontroller must interface with the microcontroller and, by extension, the MB Hub. Therefore, the microcontroller API uses the IPbus API as a lower-level communication interface. The microcontroller API uses namedtuples to create the data header for communicating with the microcontroller.

Namedtuples is a container in Python with user-defined parameters. The address map of the microcontroller has been defined in a separate package called *mcu_addr*. An image of the namedtuple created for the microcontroller API is given in Listing 6.1.

```
register_addr = namedtuple('register_addr', ['name','addr_val', 'num_bits',
    'measurement_type', 'RW'])
```

**Listing 6.1:** Code showing the construction of the namedtuple in the "mcu_addr" package.

The namedtuple contain information such as name, address value, number of bits, measurement type, and read/write access of a register. The API retrieves information from the namedtuple and uses it to communicate with the microcontroller. This serves two purposes, first, the user does not need to know the address map of the microcontroller; they only enter the name of a namedtuple, and the API takes care of the rest. Second, the *mcu_addr* package serves as a common access point for the microcontroller address map. If a developer in the future needs to add a register, or change the address value of an existing register, then only one file must be edited, which is the *mcu_addr* package.

### 6.1.3 Configuration API

The configuration process requires several functions to load configuration sets from the database, verify handshake values from the MB Hub, and configure the relevant registers on the MB. A top-level API has been created for this purpose, and a diagram of the interface hierarchy for the configuration system is given in Figure 6.1.

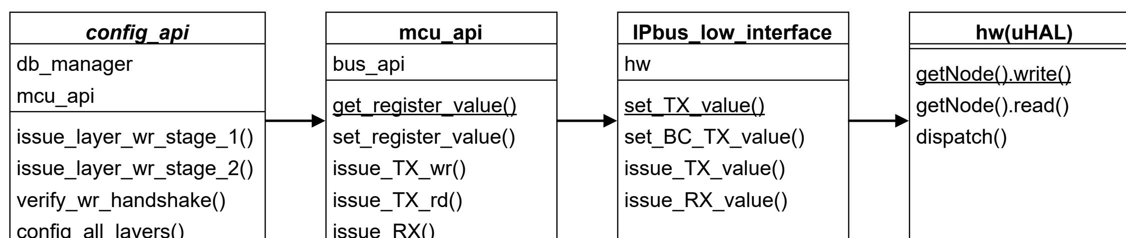


**Figure 6.1:** Interface hierarchy for the configuration system.

The "*config_api*" class contains a "db_manager" object and an "mcu_api" object; they handle database operations and communicate with the microcontroller, respectively. Setting up the configuration system only requires the user to instantiate a *config_api* object in a python

script. This class API has several functions for verifying handshakes, retrieving data from the database, and performing the ramp-up process. *config_api* has two functions for issuing configuration commands down to the DTC, stage 1 and stage 2. These two will be discussed more in subsection 6.2. The *config_online_layers*-function is the main function of the API that performs the entire configuration process.

## 6.2 Powering algorithm

There are several procedures and constraints for powering and configuring the strings of the DTC. Tasks such as enabling layers, sending configuration data and verifying handshakes must be performed every time the system is turned on. The calorimeter is made of $43 \cdot 12$ strings that all consume a significant amount of current. Several constraints must be considered while powering on such a system. Powering on all strings at once could cause a high inrush current that could cause a system malfunction if handled incorrectly. A solution to this is to develop a ramp-up algorithm that turns on each string one by one, gradually increasing the power level of the strings.

The algorithm must perform these tasks:

- Get the configuration values from the database.
- send write requests to every layer.
- verify handshake values from the write requests.
- Turn on each string and set threshold levels sequentially, and verify they do not get turned off by the microcontroller. This is to limit the current consumption of the strings while turning them on.

The configuration API contains functions that perform handshakes, and stages write commands to IPbus. The *config_online_layers*-function wraps these functions together to perform the configuration. The stage 1 and 2 functions are responsible for retrieving the values from the database and issuing the write requests for the specified layer. Stage 1 issues the write requests for the threshold 1 registers with static values; stage 2 handles the threshold 2 registers, and turns on the strings sequentially.

The reasoning behind having two stages is twofold, to prevent unnecessary write operations and to ensure more readable, concise code. Only four of the eight registers need to be changed when a new string is turned on, the three threshold values for AVDD, DVDD and PWELL, and the enable signal, which turns on the next string. If a single function issued all the write requests, we would have to configure twice as many registers each time a string is turned on. Stage 1 issues the write requests for the four static registers during the ramp-up process, while stage 2 issues requests to the four registers that change for each string. It is important to note that the global module is not used for the configuration process. The layers are assumed to have different threshold levels, so they are configured individually.

The algorithm will first use stage 1 to configure all layers, then use the stage 2 function in a loop, to gradually turn on the strings and set appropriate threshold levels. A flowchart configuring one layer is given in Figure 6.2.

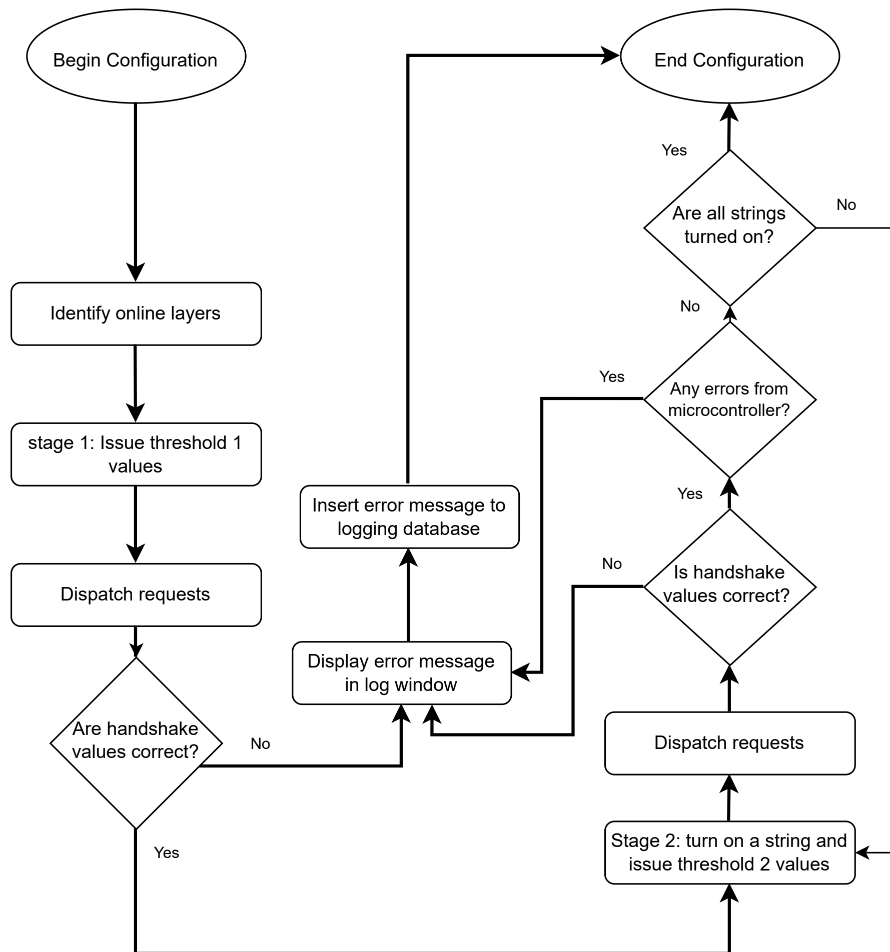


**Figure 6.2:** Flowchart of the configuration function of a single layer.

The function loops through all online layers during stage 1 and 2. The flowchart describes how one layer is configured during the function.

The left side of Figure 6.2 shows the flowchart of stage 1, where the warning threshold levels are set, along with the temperature limit. All the commands are issued prior to being dispatched as a single packet. Finally, the handshakes are verified. The right side of the figure shows stage 2, where it turns on the strings individually. The *error count* register from the microcontroller is checked at the end of the loop to assert whether any strings were turned off due to high currents.

The flowchart checks for errors during configuration, but the error handling afterwards has yet to be fully implemented. Ideally, the error messages should be displayed to the user and also be logged in a database, including a severity grade of the error.

## 6.3 MongoDB Database

The database chosen for the configuration is MongoDB. MongoDB was chosen because it is a popular, open-source, NoSQL database with Python API libraries. In addition, MongoDB is

capable of storing large amounts of data, which helps store many large configuration sets. The APIs access the database through the *db_manager* class, which interfaces with the PyMongo library. The configuration of the readout chips also implements MongoDB, and using the same database for both systems reduces the modular complexity of the pCT-system.

The hierarchy of a MongoDB database is shown in Figure 6.3.

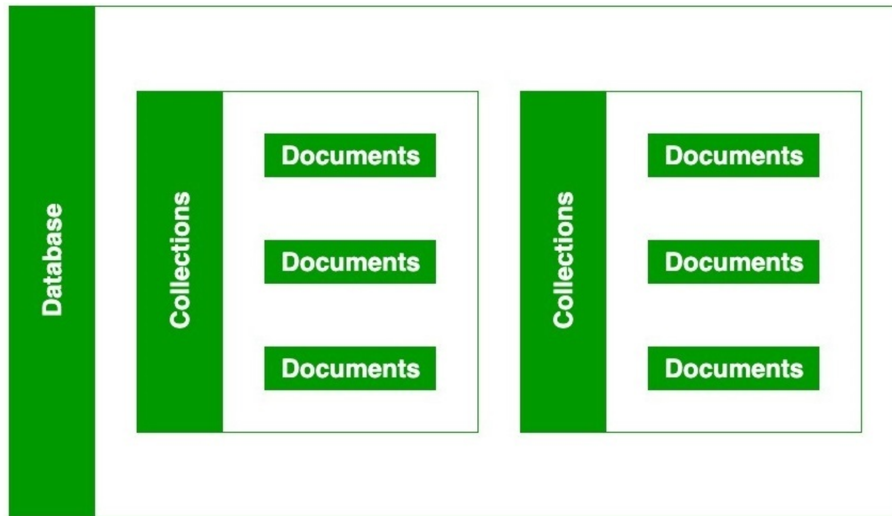


**Figure 6.3:** Hierarchy of a database in MongoDB.

Each database contains user-defined collections that each store their documents. For example, if we used one MongoDB database for both the readout and the power control, we would have one collection for each system.

MongoDB uses JSON as the template for its databases, JSON being a widely used text-based data format for storing and transferring data. A typical example of a MongoDB JSON document is shown below in Listing 6.2.

```
{
    "name": "John",
    "age": 22,
    "hobby": {
   "reading" : true,
   "gaming" : false,
   "sport" : "football"
    },
    "class" : ["JavaScript", "HTML", "CSS"]
}
```

**Listing 6.2:** Example document using JSON to store information.

A document contains fields, and each field contains a corresponding value. This value can be a number, string, or dictionary of subfields; the ”hobby” field in Listing 6.2 contains three fields with their values. For our purpose, the format of the initial configuration set was

simple, each field only needed to contain a single value, and each layer required its own set. Each configuration set must also contain a "last updated" timestamp for documentation, and it must have a field for enabling the com_modules on the FPGA. Two iterations of the database format were made; the first iteration of this data format is shown in Listing 6.3.

```
{
  "_id": {
    "$oid": "6295fd811cb9e58814cbd477"
  },
  "Config name": "Test",
  "Configuration values": [
    {
      "layer": 1,
      "Temperature limit": 111,
      "DVDD threshold 1": 500,
      "AVDD threshold 1": 600,
      "PWELL threshold 1": 700,
      "Enable signals": 4095,
      "DVDD threshold 2": 112,
      "AVDD threshold 2": 150,
      "PWELL threshold 2: 200
    },
```

**Listing 6.3:** First iteration of JSON format of a configuration set.

Each set has a unique name, and the configuration values are stored in dictionaries inside the "configuration values" field. This field contains a separate dictionary for each layer, containing the eight threshold values. Retrieving the information using Python is done by using a "get" command for the threshold values.

One of the challenges behind the ramp-up algorithm is that strings' current consumption can vary depending on the quality of the chips on the string. Some strings can draw double the current compared to others, and implementing this into the ramp-up process would only be possible if we knew each string's threshold values. The second iteration of the database contains individual threshold values for each string to alleviate this current consumption problem. Listing 6.4 shows an outline of this database implementation.

```
{
  "_id": {
    "$oid": "6295fd811cb9e58814cbd477"
  },
  "Config name": "Test",
  "Configuration values": [
    {
      "layer": 1,
      "Temperature limit": 111,
      "DVDD threshold 1": 500,
      "AVDD threshold 1": 600,
```

```
    "PWELL threshold 1": 700,
    "Enable signals": 4095,
    "DVDD threshold 2": {
        "String 1": 1,
        "String 2": 4,
        "String 3": 7,
        "String 4": 10,
        "String 5": 6,
        "String 6": 7,
        "String 7": 8,
        "String 8": 9,
        "String 9": 10,
        "String 10": 11,
        "String 11": 12,
        "String 12": 13
    },
```

**Listing 6.4:** Second iteration of JSON format of a configuration set, with for individual strings.

This new iteration is similar to the first, but all the threshold 2 levels for PWELL, DVDD, and AVDD are dictionary objects containing values for each string. The configuration API must sum these string values while performing the ramp-up process, leading to more logic on the software side, specifically for the configuration API.

Setting up the database format and creating new sets is performed with *db_manager*. This class contains functions for communicating with the Mongo database, creating new configuration sets and updating these sets with new values. All data formatting happens in this class, the Mongo database does not perform any special functions in manipulating the stored data.

## 6.4 Configuration GUI

The configuration API contains many functions for setting up the configuration process and managing the Mongo database. However, this class requires an HMI that allows for quickly setting up configuration data and is user-friendly. A person with no knowledge of the underlying process behind the configuration must be able to load a configuration set from the database and start the configuration of the strings. A logical answer to this problem is to implement a GUI for the configuration system. However, unlike the monitoring system, no commercial GUIs exist that can be applied to our configuration system, and thus a custom-made GUI must be implemented.

The primary feature the GUI requires is the ability to load configuration sets from the database and start the configuration with the push of a button. The GUI must also have an "advanced" tab that allows the user to create new configuration sets directly, insert them into the database, and modify existing sets.

Building the GUI and designing it is performed using Qt Designer. The Python library, PyQt5 is used to interface the GUI with our configuration API and database. QT Designer was used to design the GUI because it is easy to work with, and this tool has also been used previously to create GUIs for the readout electronics.

The class chart for the configuration GUI is given in Figure 6.4.

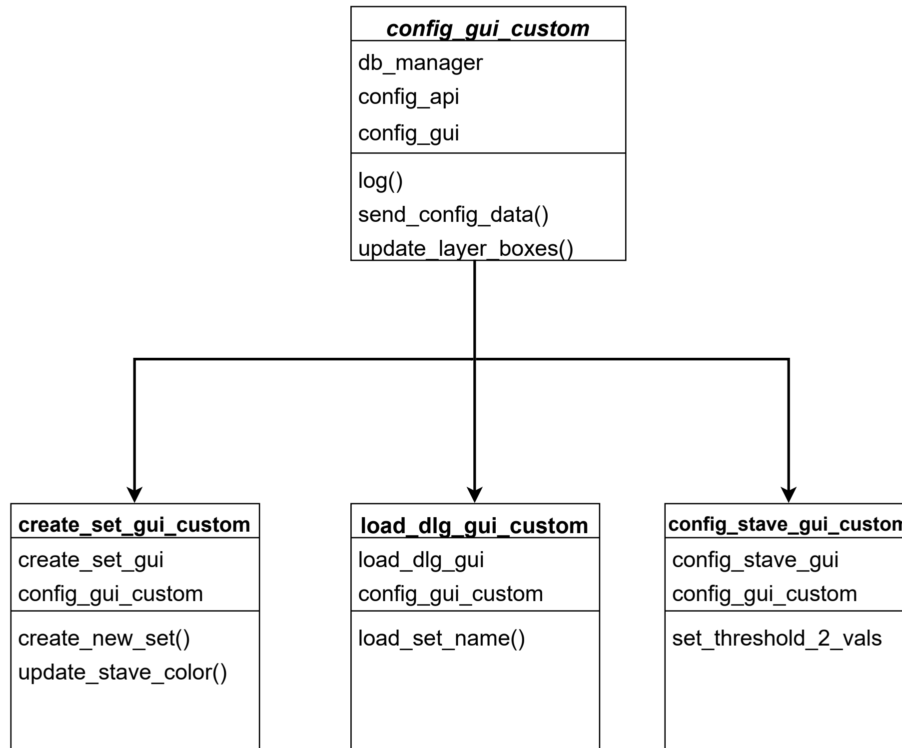


**Figure 6.4:** Class diagram of the configuration GUI.

The GUI is made out of 4 custom Python classes, *config_gui_custom* is the main window for the user. The other classes are the windows for creating new configuration sets, loading sets from the database, and setting the individual string threshold values. The *config_gui_custom* instantiates the configuration API and database manager and connects their functions to the GUI buttons.

The main window of the GUI is shown in Figure 6.5, displaying the ”basic” tab.

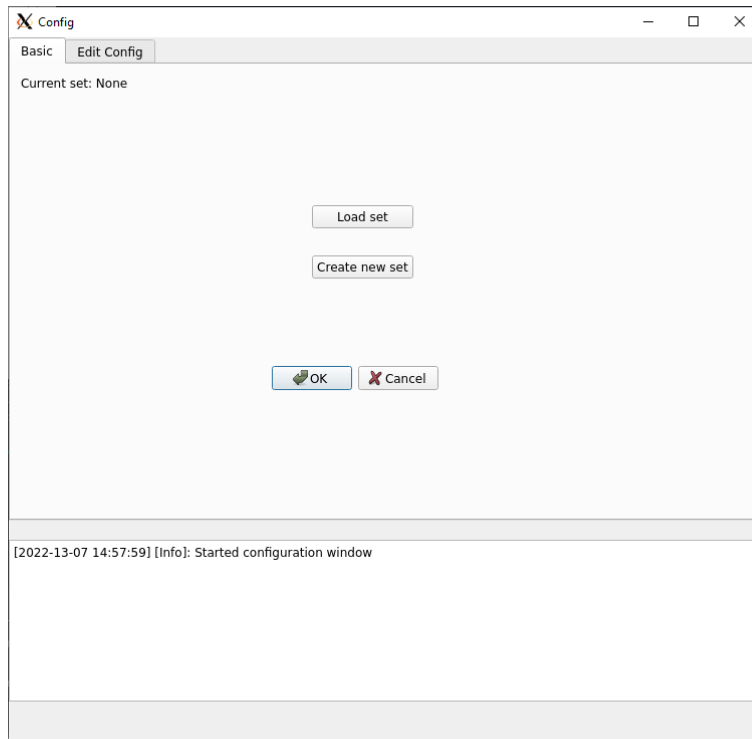


**Figure 6.5:** Image of the main window of the configuration GUI.

The window has buttons for loading a set from the database, creating a new set and inserting it into the database, and confirm/cancel dialogue buttons. The upper left of the window shows which configuration set is currently loaded. The window has a log window that informs the user of actions performed, errors, and confirmation messages. These log messages are currently only displayed in the GUI, since a logging database have yet to be implemented. A typical use case for the "basic" tab is loading a set from the database and then clicking the "start configuration" button on the main hub GUI to start the configuration process.

Clicking the "Load set" button will open up a dialogue window, showing every set in the database.

**Figure 6.6:** Image of load set window.

As shown in Figure 6.6, the window shows every set along with a timestamp of when it was last updated. Choosing a set to load is done by clicking on the set and pressing the "OK" button.

The "Edit Config" tab allows for both reading and modifying the threshold values for each set. It is shown in Figure 6.7.

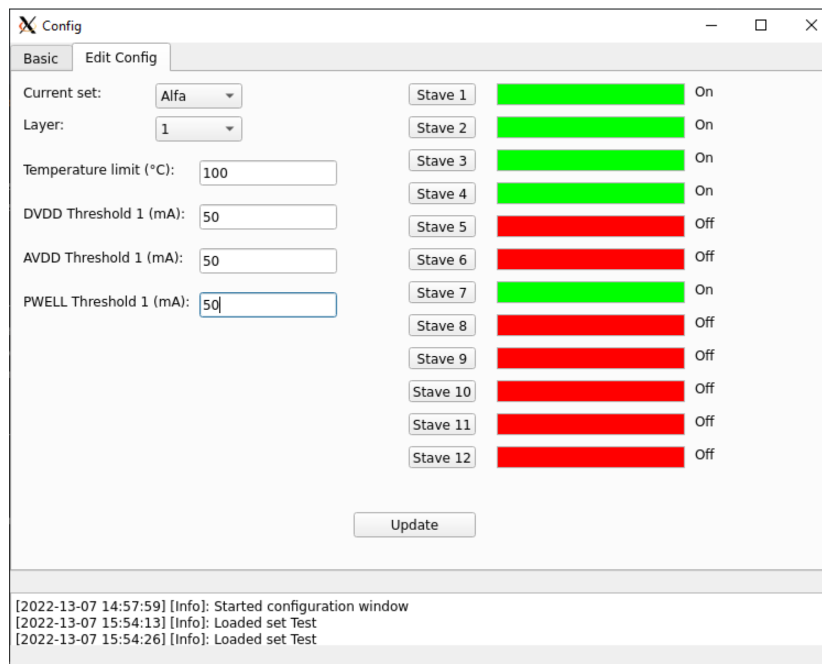


**Figure 6.7:** Image of the "Edit Config" tab.

The threshold values can be set for each layer, and enable signals for the strings can be changed interactively by clicking on their corresponding coloured box. The log window will alert the user when changes are made to a configuration set. This tab helps make small changes to existing sets but requires the user to know the structure of the configuration sets to edit them.

The "Create set" button opens a window similar to the "Edit Config" tab, allowing the user to create a new set. The user cannot set different values for each layer while creating it; instead, the "Edit Config" tab can be used afterwards to make edits.

## 6.5 Configuration Timing

Configuring the layers of the DTC requires sending multiple write requests through the PCS, which can take a significant time to perform. If the process is slow, it will feel sluggish for the user, impacting the user experience. Additionally, debugging the hardware becomes tedious and unrealistic if the configuration is slow; it is therefore vital to ensure the configuration is as fast as possible. The transmission speed of software, IPbus, MB Hub and microcontroller all affect the configuration process. The configuration should take at most 10 seconds to complete as a benchmark. Any longer and the system will feel sluggish and laggy to the user.

We will first cover the calculated delay of the configuration system. As mentioned in subsection 5.2, the transmission speed of performing one read/write with IPbus is 250 $\mu$s, but this can be improved per package by sending larger packets at once. The FPGA is an UltraScale high-speed board with an internal system clock of 300 MHz, meaning each instruction takes approximately 3 ns to perform, which we will assume is negligible compared to the other sources of delay in the PCS. The last chain in the PCS, the microcontroller on the MB can have a maximum baud rate of 1 Mbit/s, leading to 1 $\mu$s transmission speed per bit. Writing to the microcontroller requires sending 32 bits + 1 start bit; this results in a delay of:

$$\frac{n}{\mu_{baud}} = \frac{33}{921600} \approx 33\mu s \tag{6.1}$$

Where n is the number of bits being sent. The calculated delay of one write down to microcontroller becomes:

$$t_{write} + t_{\mu} = 250 + 33 = 288\mu s \tag{6.2}$$

The configuration of the layers is done in parallel; the FPGA can transmit data to all layers at once, so the configuration timing of one layer will approximately equal the configuration timing of so to calculate delay, we need only look at the configuration timing of a single layer. Eight registers must be written initially; for each string turned on, three threshold registers and the enable signal register must be written. This amounts to $8 + 4{\cdot}12 = 56$ write requests for a single layer. Confirming the handshake values must be done individually by software, and performing 56 write requests for 43 layers leads to reading an additional 56

values from each layer. The theoretical delay for an entire configuration process will then be:

$$2 \cdot (56 \cdot 43) \cdot t_{IPbus} + 56 \cdot t_{\mu} = 4816 \cdot 250 \mu s + 56 \cdot 33 \mu s \approx 1.2 s \quad (6.3)$$

It is important to note that the delay of the software is not considered in this calculation, only the delay through the PCS chain. This calculation also considers the worst-case scenario of the IPbus delay being 250 $\mu$s; we can expect the delay from the IPbus to be significantly shorter per data packet sent when sending large data packets at once. We can see from Equation 6.3 that the calculated delay amounts to 1.2 s, which is significantly faster than our maximum of 10s.

### 6.5.1 Testing configuration timing

The next step is to measure the actual delay of the PCS. We currently have a prototype layer of the PCS, which is a communication link from the software to the microcontroller. Only one layer is operational; therefore, tests performed will be incomplete and may not be completely accurate, but it will still give us a rough estimate of the expected delay in the final product. For comparison sake, we will first calculate the estimated configuration timing for one layer using Equation 6.3:

$$2 \cdot (56 \cdot 1) \cdot t_{IPbus} + 56 \cdot t_{\mu} = 112 \cdot 250 \mu s + 56 \cdot 33 \mu s \approx 30 ms \quad (6.4)$$

We can expect the measurements of the delay to be approximately 30 ms. The test configures the microcontroller 1000 times and calculates the average time and standard deviation. The time measurement is done with the Time library from Python, which includes the software delay of the system, and can lead to minor inaccuracies in the measurements.

The test setup had an average configuration time of 51ms($\sigma$ = 7ms), close to the calculated value but not wholly accurate. This can be explained by the software delay or other small processes in the transmission that were not accounted for in the calculations.

From the test, we can expect the configuration timing of all 43 layers to be slightly larger than the calculated timing from Equation 6.3.

# 7 Monitoring System

*This chapter describes the solution for a monitoring system for the MB. The chapter tackles the characteristics of the system and what is needed for this project, along with the choice of time-series database used to store the data points. Features such as the specification of storing monitoring data and displaying it to the user using third party software is also discussed. Finally, the different types of data filtering are discussed, along with how Pandas dataframe can be used to customize the filtering of data points.*

The microcontroller on the MB directly monitors the strings' current, temperature, and voltage levels. Most importantly, it is responsible for turning off the strings if they exceed these threshold values. The microcontroller is responsible for turning off the strings because it is a time-critical event; any delay in turning off the strings can damage the ALPIDE-sensors. However, the user still needs to know the status of the MB. Knowing the current draw and the strings' temperature is essential to effectively troubleshoot the DTC while performing tests. Therefore, it is necessary to create a monitoring system from the client side that can retrieve the monitoring data from the MB and display it to the user. It is logical to store the data in a time series database, which is a time-sensitive database that timestamps the data inserted. The data points from the database can be displayed using third-party software. However, not all data retrieved is valuable to the user. The data points should be downsampled to reduce data storage and strain on the database server.

InfluxDB was chosen because it is an open-source, popular time series database with a large online community. In addition, its compatibility with the graphing tool Grafana allows us to display the data with professional graphing tools. Influx and Grafana also feature intuitive web interfaces that are user-friendly and flexible. InfluxDB also supports the Telegraf plugin, which allows for seamless monitoring of database performance.

Timing of the client-side monitoring is also essential, even though there are no time-critical operations from the client. The user should be able to quickly assess the status of the MB with little delay to more accurately diagnose any faults during a run.

Figure 7.1 shows the overview of the PCS and highlights the monitoring system, along with its respective database.

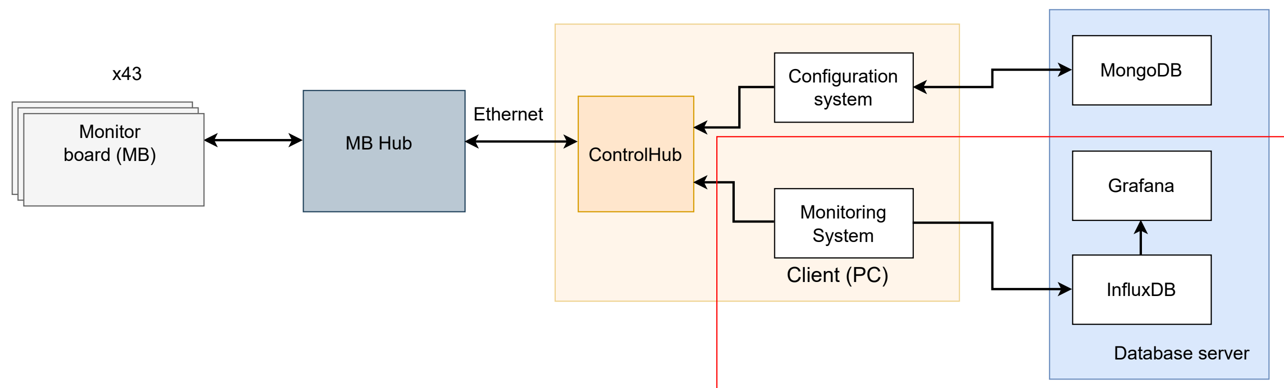


**Figure 7.1:** PCS overview, highlighting the monitoring system.

The monitoring system sends data to InfluxDB and Grafana interfaces with InfluxDB to display the data points.

## 7.1 Monitoring API

The monitoring system shown in Figure 7.1 is made of several classes responsible for retrieving data from the MBs, processing it, and sending it to the database. A dedicated monitoring API class is created for the system, which handles data polling and interfaces with the database. A class diagram of the hierarchy of the monitoring API is shown in Figure 7.2.

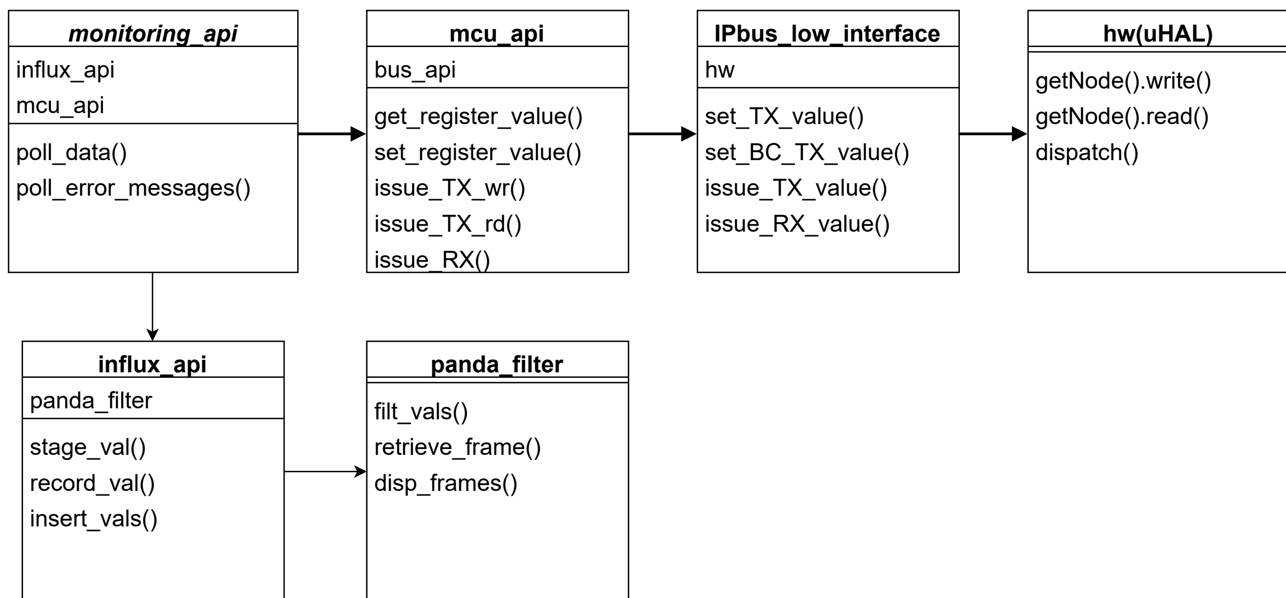


**Figure 7.2:** Interface hierarchy for the monitoring system.

The top level of the API interfaces with the microcontroller through the *mcu_api*. The database is interfaced using a custom class, *influx_api*, which interfaces with influxDB to store data in the database and filter the data using a pandaframe custom class. The primary function of the monitoring API class is to poll the data from the microcontroller and insert it into the Influx database.

Polling the data involves retrieving the measured voltage and current of DVDD, AVDD and PWELL and the measured temperature reading from the PT1000 element. In addition, the monitoring API must be able to handle errors from the MB. A detailed discussion of the error handling of the MB is given in subsection 5.4. The monitoring API must poll the error registers during the monitoring, which requires its own process. The polling function only polls data once and then returns error messages. This allows the function to return error messages to the monitoring GUI. Figure 7.3 shows the flowchart performed when the polling function is called.

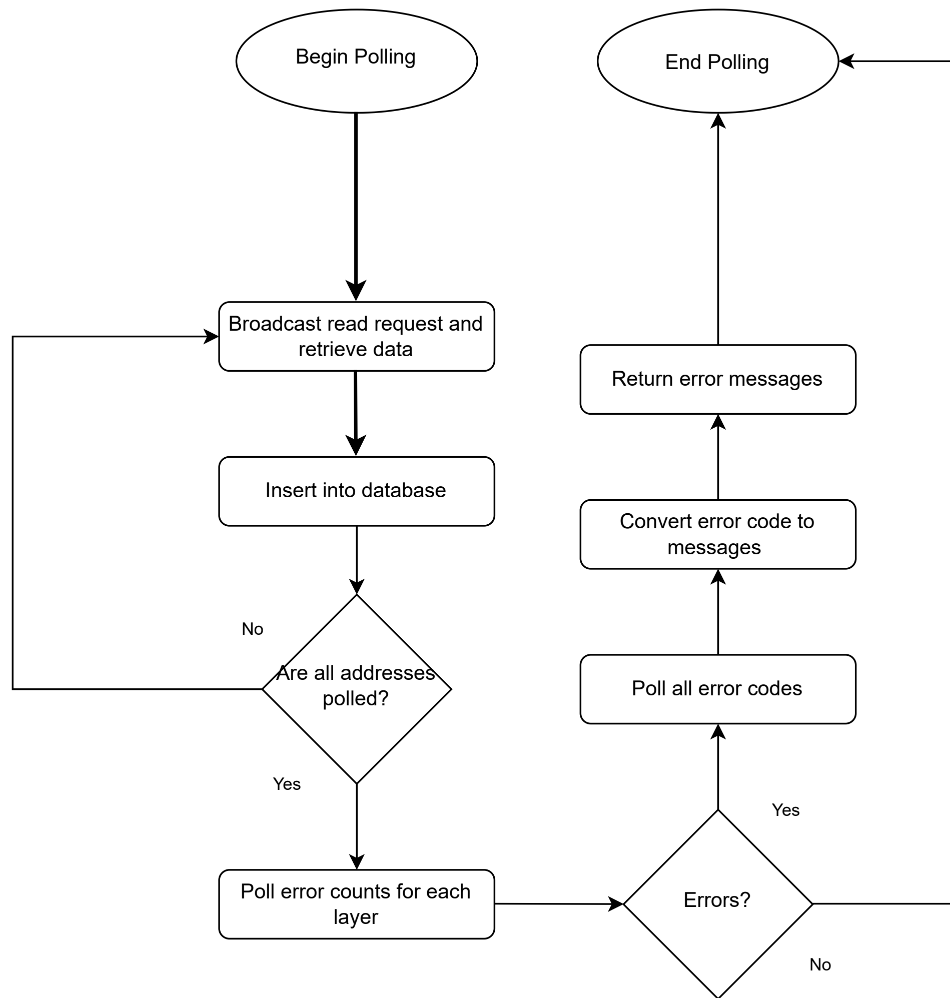


**Figure 7.3:** Flowchart of the polling function in the monitoring API.

Polling the data is simple, it uses the microcontroller API to broadcast and retrieve the data from all of the monitoring addresses. After inserting the data into the database, the function checks if any errors have occurred on the microcontroller. If yes, it will poll the error codes, convert them to messages and return them.

The polling function only polls the data once; the GUI which instantiates the monitoring API must continuously poll data using this function during the monitoring process. The flowchart of this procedure in the GUI class is given in Figure 7.4.

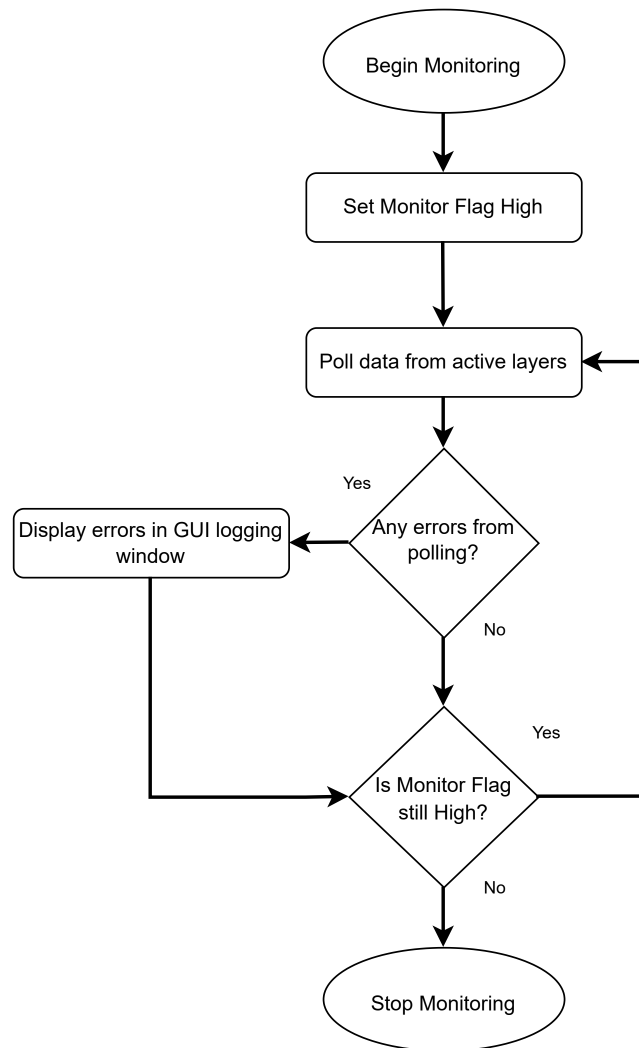


**Figure 7.4:** Flowchart of the top level monitoring process.

From the figure, a Boolean flag called "Monitor Flag" is set high and the process polls the data from the MB. If the polling function returns errors, they are then displayed in the GUI logging window. This monitoring process will continue until an outside function sets the Boolean flag low. Displaying the errors from the microcontroller in the GUI is a temporary solution for handling errors. The design decision has yet to be made on how to process the errors, for now they are only displayed in the logging window.

The monitoring API polls data based on the namedtuples stored in the *monitor_addr* list in the *mcu_addr* package. This means that one can freely change the monitoring addresses by adding or removing namedtuples in the list. This is useful if changes to the monitoring process are made, and also makes it easier to repurpose the API to other control systems in the pCT-project. In addition, each error message has a corresponding namedtuple which is used to convert error code to message, as well as indicating the severity of the error.

## 7.2 Monitoring GUI

The actual monitoring and overseeing of the polled data is done by Grafana which is discussed in subsection 7.4, but we still need a user interface for starting the monitoring process. For this purpose, a custom GUI for the monitoring system is designed using PyQt5 in the same manner as the configuration GUI discussed in subsection 6.4. An image of the monitoring GUI is shown in Figure 7.5

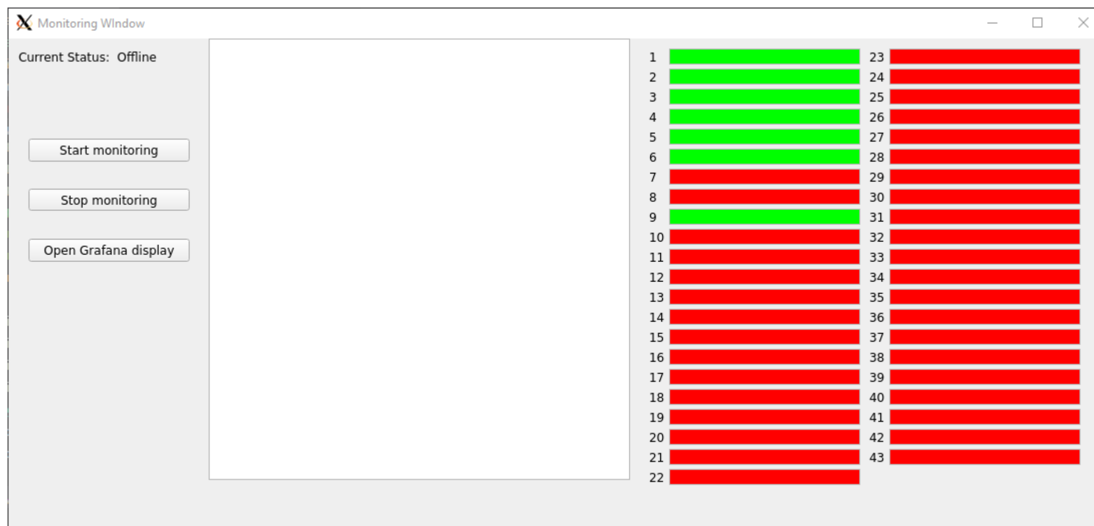


**Figure 7.5:** Image of the monitoring GUI, showing the buttons for managing the polling, as well as the log window.

The GUI is relatively simple with buttons to start and stop monitoring, as well as one for opening the Grafana display in the web browser. The log window notifies the user when the monitor process begins, as well as displaying any errors from the monitoring directly. It also graphically displays which layers are active on the right side, where green is on and red is off. This pattern only notifies if a com_module is turned off on the MB Hub, not the actual layers themselves.

## 7.3 InfluxDB

A database is required as storage for the monitored values from the MBs. A time series database stores data points along with a time stamp, allowing for plotting the data points as a function of time. There are many time series databases available on the market and they come with their own benefits. It was chosen to use InfluxDB for this project. InfluxDB is a popular, open-source, time series database featuring many plugins and APIs that can be integrated with the database. Among these plugins is Grafana, another popular web tool, used for visualizing data and plotting graphs with professional, easy to use interfaces. InfluxDB also features a web interface that can create query requests without knowing the syntax.

The InfluxDB web interface has a graphing tool, where queries can be set up and data can be displayed in various manners, such as graphs or histograms. An image of the tool is

displayed in Figure 7.6.

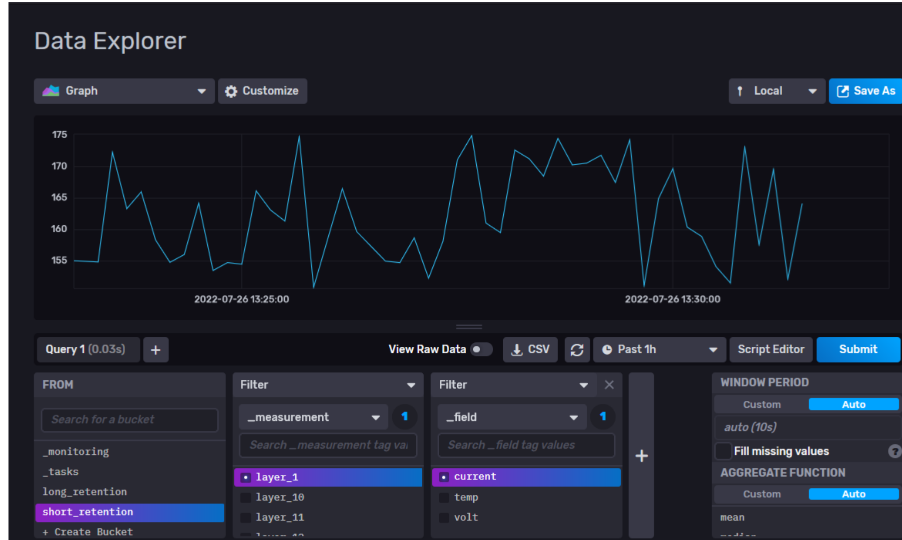


**Figure 7.6:** Image of the explore tab in the influx web interface, displaying arbitrary data inserted by the monitoring API.

The explore tab is useful for observing the data and viewing the raw data as it is inserted into the database. The lower part of Figure 7.6 shows the query tabs, which can be used to interactively create queries. The web interface also allows for creating dashboards, where many individual graphs can be displayed at once. The Telegraf plugin, which is mentioned later this section, is made of such a dashboard.

InfluxDB V2 is used for this project, the main difference between previous versions and V2 is the use of the new functional data scripting language specifically designed for InfluxDB, called Flux. Flux is designed to unify querying and processing data in the database, able to retrieve the data points from the database and automatically filter them. This query language is integrated into the web interfaces for both InfluxDB and Grafana, both having example queries that can be generated without needing to know the syntax of Flux. An example of a Flux query is shown in Listing 7.1.

```
from(bucket: "example-bucket")
    |> range(start: -1d)
    |> filter(fn: (r) => r._measurement == "example-measurement" and r._field ==
       "example-field")
    |> mean()
    |> yield(name: "_results")
```

**Listing 7.1:** Example of a flux query retrieving data and filtering it.

In the example, the "*from*" command chooses which source the data should be retrieved from, "*range*" specifies the time range to be filtered, and "*filter*" can filter the data based on tags or measurement type. The "*mean*" function calculates the average of the values, Flux has many functions for filtering the data points based on common filtering techniques.

The structure of the database must also be considered. InfluxDB stores data in "buckets", which is their name for a database with a retention policy. A retention policy is a given time interval for how long the database will store its data. All of this can be defined and modified in the InfluxDB web interface. Storing large amounts of data points for long periods of time is often resource heavy and unnecessary, it is therefore logical to store the data points in a bucket with a short retention policy. For longer periods of time the data can be filtered and sent to a different bucket with longer retention policy.

The filter used on the data point must reduce the size of the data set, while also minimizing the amount of information lost. Filtering the data points is discussed more in subsection 7.6.

InfluxDB also supports the Telegraf plugin, which allows for quick and easy monitoring of the database performance. This is useful for obtaining metrics on the database while the monitoring is performed. Figure 7.7 shows the Telegraf plugin window in the web interface.

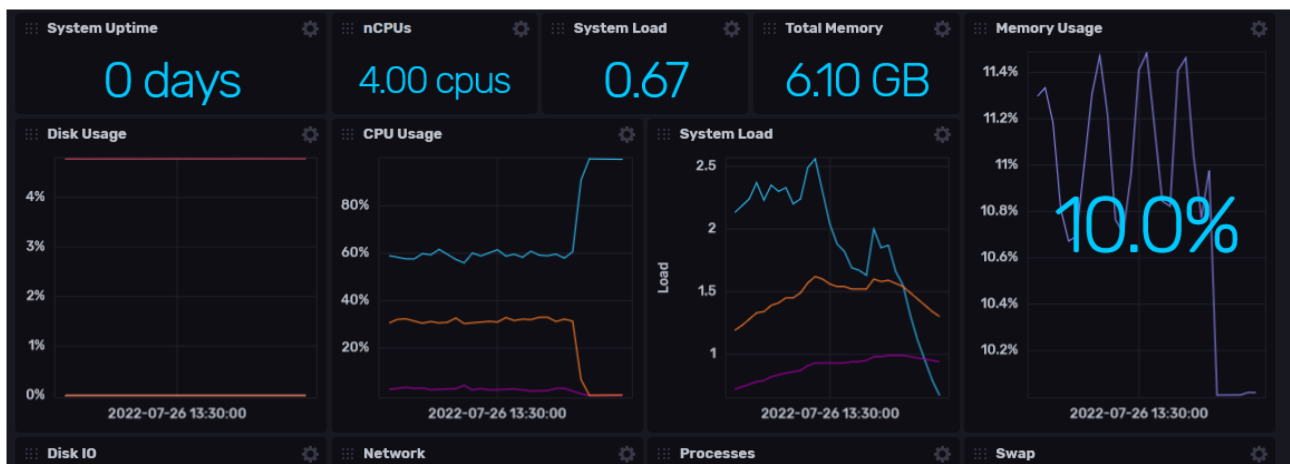


**Figure 7.7:** Image of Telegraf plugin displaying various metrics of the database.

From the figure, Telegraf monitors several parameters by default, including CPU usage, memory usage and system load. These measurements can be used to assess the performance of the database.

## 7.4 Grafana

Grafana is an open-source interactive web interface that can display data in several ways. It can provide graphs, histograms, and chart solutions for displaying data in a web application. Grafana was chosen as the graphing tool for this project due to its ability to display data clearly and concisely, create interactive plots, and be directly compatible with InfluxDB.

Grafana is hosted on a server, but unlike InfluxDB and MongoDB, we do not need to create a custom API to communicate with it. Connecting Grafana to influxDB and setting up the dashboard for the monitoring can all be performed in the Grafana web interface. After connecting the Influx database to Grafana, creating graphs for Grafana to plot is done by setting up a Flux query for the specified measurement.

For this project, we want a clear dashboard that shows off all the measurements being retrieved from the MBs, making it easy for the user to quickly spot abnormalities, such as high temperature. There are three different measurement types and 43 layers for each measurement that needs to be displayed. Plotting 43·3 graphs on a single dashboard would lead to an illegible and cluttered dashboard. The solution is to plot a histogram of each measurement, showing the last recorded value in the database. Grafana's interactive plots let us set a hyperlink in each histogram that points to the Grafana graph of the respective histogram's measurement. The histograms give the user an overview of all measurements and the ability to quickly assess each measurement individually with one click.

An overview dashboard was created in Grafana and an image of this dashboard is shown in Figure 7.8.

**Figure 7.8:** Image of the overview dashboard inside the Grafana web application.

The above figure shows the overview dashboard displaying arbitrary data that has been inserted into the Influx database. The dashboard is made out of three tabs, one for each measurement type, and clicking on a histogram will lead to a second dashboard that plots the data as a function of time for the measurement the user clicked on. The second dashboard is made of three plots, one for each measurement type. A custom variable is added to this dashboard that allows the user to change the layer from which the measurements are retrieved from. An image of the second dashboard is shown in Figure 7.9.

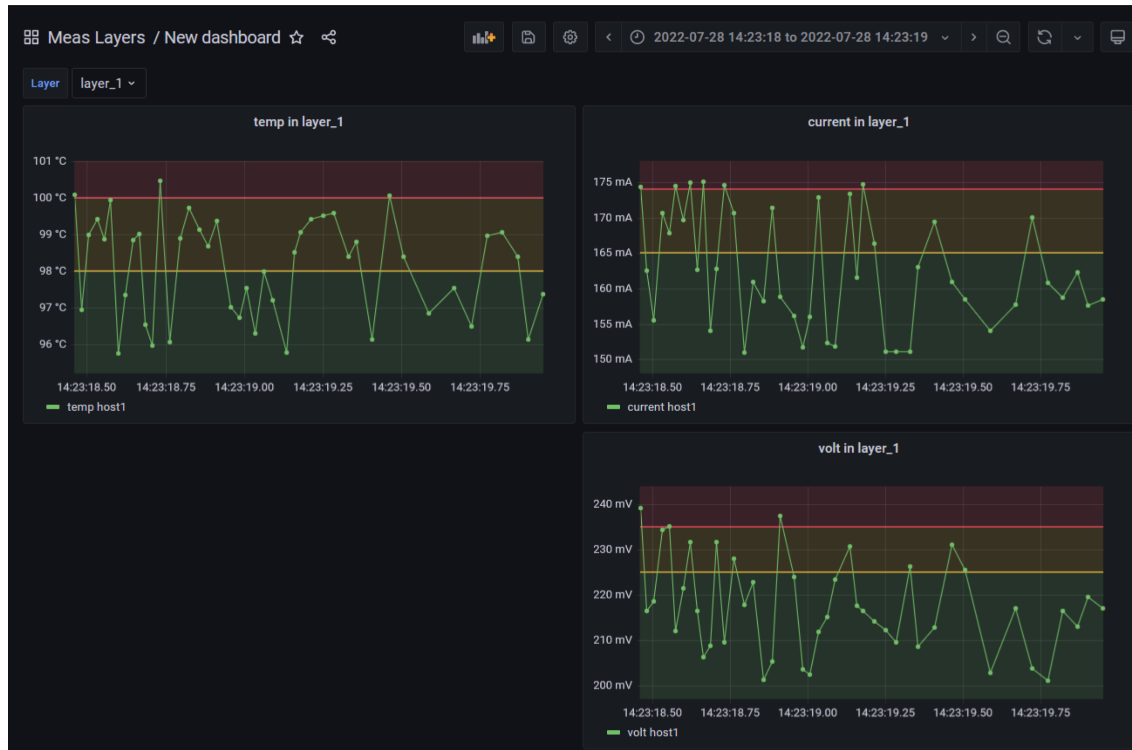


**Figure 7.9:** Image of the measurement dashboard inside the Grafana web application.

Using these two dashboards allows for easy and seamless monitoring of the data retrieved from the MBs. Both dashboards are exported to JSON-format and stored on the GitHub repository. Setting up the dashboards on a new Grafana server would only require uploading the JSON-document for each dashboard onto the Grafana web application.

## 7.5 Characteristics of monitoring operation

Attributes such as read and write speeds, storage space, and database performance are all important characteristics that need to be determined before we can begin designing and assess the system.

### 7.5.1 Transmission speed of monitoring process

During the monitoring operation, the temperature must be measured, as well as the current consumption, and voltage levels of DVDD, AVDD, and PWELL for all 43 layers, along with the error register. This leads to reading 8*43 registers every time we poll the MBs. There is no inherent reason to have a delay between the polling, so the data is polled as quickly as possible. The transmission speed of the data will then not only determine the speed of data acquisition, but also the amount of data points received per second, which determines how much storage space is needed and the sampling rate of the measurements.

For the monitoring system, a read request must be sent to all layers and then client software must retrieve the data from the RX register of every com_module. This translates to writing

to the global TX register and reading 43 registers from the com_modules. This action must be performed 8 times, to get all measurements. The time for polling all measurements from every layer once then becomes:

$$8 \cdot (t_{write} + t_{\mu} + 43 \cdot t_{read}) = 8 \cdot (250\mu s + 33\mu s + 43 \cdot 250\mu s) \approx 88ms \tag{7.1}$$

The read speed of IPbus is assumed to be 250 microseconds in this calculation. We can now assume that one poll of all measurements will approximately take 88 milliseconds. The amount of data points retrieved per second will then be:

$$\frac{1}{0.088} \cdot 8 \cdot 43 = 11.33 \cdot 8 \cdot 43 \approx 3897 \tag{7.2}$$

The monitoring system polls all measurements approximately 11 times every second, which leads to retrieving approximately 3897 data points every second. The sampling frequency is then 11 Hz, which is substantially better than our requirement of 1 Hz sampling frequency.

Furthermore, a test was performed to measure the timing of polling the data. This test used the prototype setup of the PCS discussed in subsection 8.4. The main limitation of this test is that the test setup only has one layer connected to the MB Hub, meaning we can only poll one layer during the test. We should therefore expect the measured polling time to be shorter than the calculated one. If we use Equation 7.1 to calculate the speed of polling one layer, the timing is found to be approximately 4.3 ms.

The test will poll all data points once and record the time spent polling them, it will then repeat this process 1000 times and find the average of the data. The results, along with comparison of the calculated values, is shown in Table 7.1.

| Calculated timing(43 layers) | Calculated timing(1 layer) | Measured timing(1 layer) |
|---|---|---|
| 88ms | 4.3ms | 8.7ms ($\sigma$ = 3.4ms) |

**Table 7.1:** Comparison of the polling timing of the monitoring API.

Polling the test setup took 8.7 ms with a standard deviation of 3.4 ms. Comparing the calculated timing of polling one layer with the measured timing, we can see that the calculated timing barely does not fall within one standard deviation of the measured timing. The discrepancy between the two values can be explained by the calculations not considering the software delay.

### 7.5.2 Storage Space

Storage space is a vital part of the database, and is determined by many factors. The storage space needed for this project is determined by the amount of data retrieved every second, the resolution of said data, the retention policy of the buckets in the database and how much space is needed to store the data points in the database.

The amount of data retrieved every second has already been calculated in Equation 7.2, amounting to 3897 data points per second. The type of data stored in the monitoring operation, excluding any log messages, is float numbers. The resolution of the float numbers is the amount of decimal points used for the data point, and therefore this must be decided beforehand. The temperature will not change erratically, an accuracy up to the first decimal point is therefore acceptable. The current and voltage is measured in mA and mV, respectively, already giving us a high level of precision. The voltage is measured over a shunt capacitor that can read voltages up to 250 mV and temperatures will obviously not exceed 1000 degrees, so 4 decimals is a reasonable resolution to store the data in, giving us one decimal point accuracy for both current and temperature. This translates to needing 3 bytes, or 12 bits to store the information in our database. In the InfluxDB hardware sizing guidelines, it is stated that a non-string value requires approximately 3 bytes of data to be stored[26]. This leads us to requiring 6 bytes of storage for a single data point. The amount of bytes of data stored each second will then be:

$$3897 * 4 = 15588 \tag{7.3}$$

Which becomes about 16 kilobytes per second. In this project, it is assumed the storage space will be approximately 500 GB. We can then calculate the highest retention policy without exceeding the storage limits. First, we calculate how long the monitoring process can run before running out of storage:

$$\frac{500 * 10^9}{15588} = 320759s \approx 8910 \ hours \tag{7.4}$$

Next, let us assume that the pCT system will be running for a maximum of four hours during one day. The maximum retention policy becomes:

$$\frac{8910}{4} = 2272 \ days \approx 6 \ years \tag{7.5}$$

In total, 500 GB of storage allows us to store data for a little over six years. This is a long time period for storing data, and this can be increased even more if we filter the data. A retention policy of six years might be excessive, it might not be relevant to store data for more than 1-2 years. This is a decision that has to be made by the people who will operate the system itself. In conclusion, the storage space itself does not play a major decision for how long we want to store the data.

### 7.5.3 Optimizing data acquisition

InfluxDB comes with a variety of options to optimize the data acquisition process. These different options can have a significant impact on the data acquisition and must be considered to ensure it is a good fit for our system.

InfluxDB features several write options for its Python client: synchronous, asynchronous and batching writes. The default option used is batching writes, which is considered optimal

for InfluxDB. Batching writes collects a specified amount of data points and when full, establishes a https connection with the database and sends the data. This process is very quick in comparison to synchronous writes, which has to establish a https connection for every data point.

The batching size can be customized by the user, but the optimal batch size for influxDB is 5000[27]. The drawback of the batching is that data will not be sent until the batch is full.

We can compare the speed of different batching sizes by inserting 43·1000 data points into the database and record the time taken. Table 7.2 shows off the time difference between 50, 500, and 5000 batch size.

| Write option | Write speed (43000 points) |
|---|---|
| Synchronous | 1151s |
| Batching(n = 50) | 10s |
| Batching(n = 500) | 4s |
| Batching(n = 5000) | 2s |

**Table 7.2:** Recorded speeds of inserting 43000 data paints into InfluxDB using different write options in the Influx Python API.

The results from the table show that the synchronous option is substantially slower than the batching options. The results also show that a higher batch size is faster, but there are diminishing returns past 500.

A large batch size could affect how quickly the values are inserted into the database, since the API will wait until the entire batch is filled up before sending. However, This not an issue if we use the recommended batch size of 5000. The monitoring API polls approximately 3897 values each second, meaning a batch of 5000 will be filled up every 1.3 second.

## 7.6 Downsampling of InfluxDB

Downsampling is the process of removing sample points in a signal and it is crucial to do for three reasons: to save disk space used for storing the data, to reduce network traffic, and to increase performance of your server. Recording data from a sensor inadvertently leads to redundant data points. If the measurement remains static over a greater period of time, recording high resolution data of essentially the same value is a waste of space. Additionally, the data is received from an instrument with an uncertainty to its measurements. Storing data points that are within the uncertainty range of the previous data point is also redundant. We want to downsample our data in such a way that we remove as much data as possible, while retaining the information.

There are many ways to downsample your data, there is an entirely separate field on the subject, but for this project there is no need to downsample with cutting edge technology. However, efforts should still be made to avoid storing too much redundant data. It is important for this monitoring system to record abnormal data points, i.e. spikes in the

measurement data, which can help us diagnose issues with the DTC. Using mean or similar filters results in too coarse of a filtering to be used in this project.

Creating a custom filter that fits our system is one solution, the question then becomes what type of filter fits the best. A filtering technique used by OSIsoft in their PI servers seems promising to base our filter on, due to its focus on filtering data from instruments[28]. The technique uses an exception and compression rule, that aims to reduce the amount of data points from an instrument while retaining the information of the graph. The exception filter is based on removing data that can be perceived as noise from the sensors. Figure 7.10 shows how this is implemented in OSIsoft's PI servers.

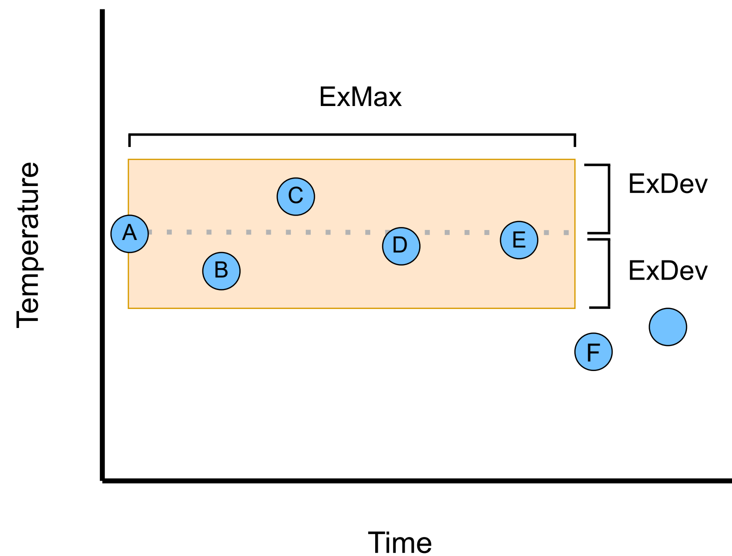


**Figure 7.10:** Image showcasing an example of the exception filter. Points in the deadband set by "ExMax" and "ExDev" is dropped by the filter. image based on slides from OSIsoft[29]

The filter starts with a "snapshot" value, which is point A in the figure. The filters sets two restrictions on the next points which are defined by the two parameters "ExMax" and "ExDev". "ExDev" defines the deadband of the filter, all points that appear in the deadband is considered noise and are removed from the set. If a point is recorded outside the deadband, then that point becomes the new "snapshot" value that defines a new deadband around it. The "ExMax" attribute determines the time span allowed between two points. If no new values exceeding the deadband is recorded for an "ExMax" amount of time, then the next data point is kept and set to be the new "snapshot" value, regardless of whether it was in the deadband of the previous snapshot. This is done to prevent excessive gaps between recorded data points. Lastly, when the data point exceeds the deadband, the previous point will be retained. This is done to retain the shape of the graph. Figure 7.11 demonstrates how not retaining the last point can affect the shape of a graph.

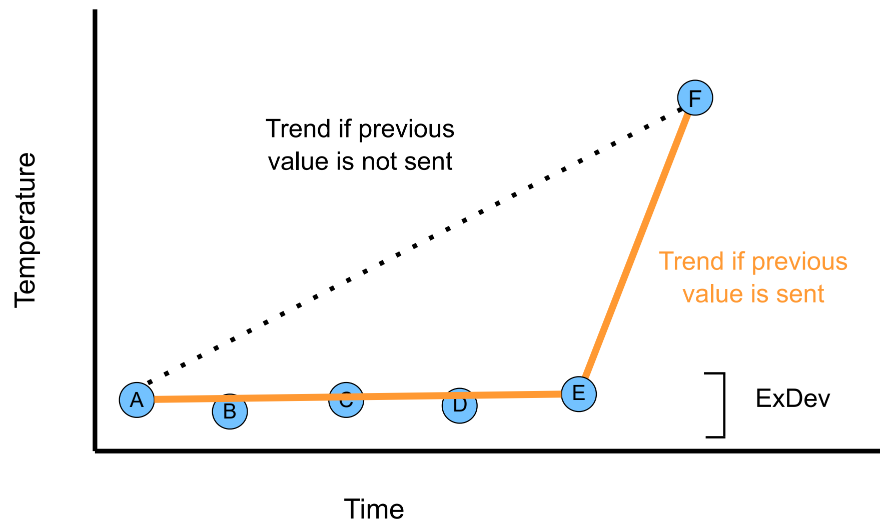


**Figure 7.11:** Graph showing the projected graph depending on if we retain the last point before the snapshot value. Based on slide from[28].

The graph displays the exception filtering method, where point B to E is not included in the set due to them being in the deadband, which gives us the dotted graph. The orange graph keeps point E in its set, since it was the previous value of point F. The dotted graph would give us the impression of a steady increase in the temperature from point A to point F, even though that is not the case. The orange graph keeps one extra value in its set, and in return, becomes a more accurate representation of the measurements.

Lastly we can mention the compression technique, which uses a "swinging door" algorithm to remove redundant data points. The process makes a line between two points and removes the next data points whose angle in relation to this line is not above a certain threshold[30]. This process is more complicated to implement than the exception test and it is currently seen as unnecessary for this project. The compression test can be revisited in the future if database performance becomes a problem.

Implementing custom filters for our data points means that we must be able to manipulate the data points in a clear fashion before sending them to the Influx database. Pandas dataframe is a two dimensional data structure with rows and axes for efficiently setting up tables of data and manipulating it. InfluxDB is also compatible with Pandas dataframes and can be inserted directly into the database. By inserting the data into Pandas dataframe, we can analyze the data and modify it as we wish, this allows us to efficiently create custom filters in Python. A custom class, *panda_filter*, is responsible for managing the andas dataframes and contain functions to format the inserted data. The class contains three Pandas frames, one for each measurement type. The dataframe consists of a column containing the actual data, and a column for the timestamp. Listing 7.2 shows an example of the structure of the Pandas dataframes.

```
    temp   _time
0    1      2022-10-24 13:21:31.373319
1    1      2022-10-24 13:21:31.375163
2    2      2022-10-24 13:21:31.375163
3    3      2022-10-24 13:21:31.375163
4    5      2022-10-24 13:21:31.375163

     volt   _time
0    2      2022-10-24 13:21:31.373319
1    3      2022-10-24 13:21:31.375163
2    4      2022-10-24 13:21:31.375163
3    6      2022-10-24 13:21:31.375163
4    7      2022-10-24 13:21:31.375163

     amp    _time
0    3      2022-10-24 13:21:31.373319
1    3      2022-10-24 13:21:31.375163
2    4      2022-10-24 13:21:31.375163
3    8      2022-10-24 13:21:31.375163
4    9      2022-10-24 13:21:31.375163
```

**Listing 7.2:** Listing showing the setup of the three dataframes used in the panda filter class.

The index is the leftmost value, followed by the measurement type column and lastly the timestamp column, which uses the "timedate" library format. The InfluxDB client supports Pandas dataframes, but the index must be set to the timestamp for it to be accepted by the client. The panda filter class automatically sets the timestamp to be the index when outputting the dataframes.

Another layer of downsampling is performed between the short retention and the long retention buckets in InfluxDB. The long retention bucket stores data for historical purposes and high resolution data is not needed, therefore a filter can be performed between the two buckets. The data flow for the monitor system is shown in Figure 7.12.

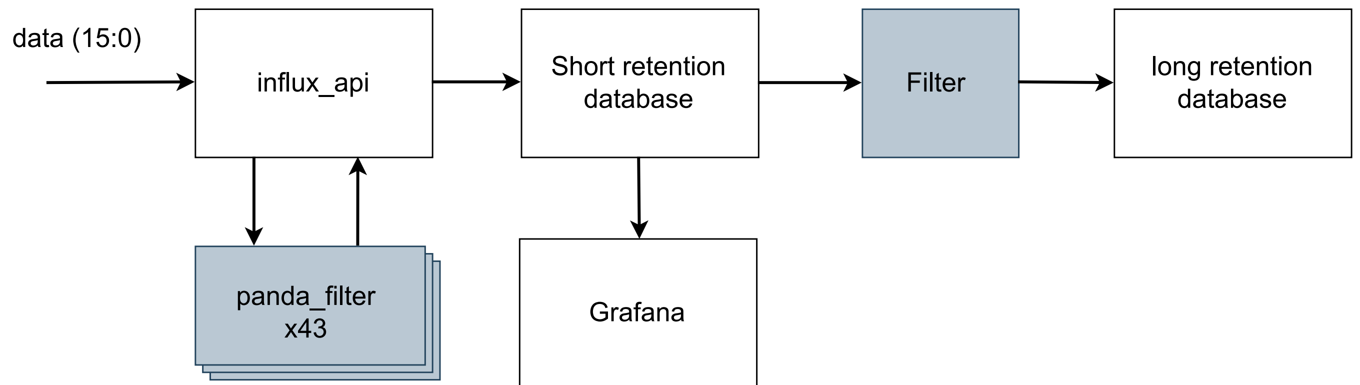


**Figure 7.12:** Block diagram of data being processed through the monitoring system.

The *influx_api* class instantiates 43 *panda_filter* objects that stores the data points and gives them a timestamp. When *influx_api* sends the data to the database, it retrieves the dataframes from the *panda_filter* and sends it to the short retention database. All data sent to the short retention database is also sent to Grafana to be displayed. The long retention database queries and filters data from the short retention database, to be stored for longer periods of time.

# 8 Testing and Verification

*This chapter describes the simulations created to test IPbus and control software and tests performed on the Power Control System prototype. Testing methods and test coverage is analyzed and discussed, and methods for error prevention are highlighted. Test scripts have been created to test and verify the various parts of the PCS and control software, and the results are discussed in this chapter.*

Testing and verification are among the most critical and potentially time-consuming parts of the design process. Therefore, it is essential to create tests for the system during the design process and ensure it is easy to test. This chapter describes the different tests performed on the control software, the MB Hub and the microcontroller of the MB.

## 8.1 Test Coverage

The PCS consists of the control software, the MB Hub, and the Monitoring Board. The tests will focus on the control software developed in this thesis and the communication link between the software and the MB. It is important to note that verification of the high-level APIs was not done due to the ALPIDE-sensors not being available in the test setup.

The tests and verification of the software design consist of:

- Simulation library for early testing of IPbus communication.
- Testbenches for low level APIs and database.
- GUI implementation which allows for direct testing of API functions.
- Performance test of InfluxDB.

The hardware design verification consists of:

- Reliability test of the MB Hub.
- Speed test of transmission speed of the MB hub.
- Reliability test of USART communication, including test for different baud rates.
- Speed test of configuration process and monitoring process.

A simulation library was implemented to test basic API functions without having access to the hardware. In addition, other software test scripts were created to verify the functionality of the APIs. One can also interactively test the functions using the main_hub GUI.

The hardware design comprises reliability and speed tests of the two communication links in the PCS. The speed tests also measure the performance of the configuration API and the monitoring API.

## 8.2 Simulation

The FPGA-design and the MB-design is still in development during this thesis and was initially unable to test and verify configuration and monitoring system components. A solution to this problem is to create a simulation that mimics the system's behaviour. The simulation aims to create a suitable testing environment for the GUI and APIs developed and allow for testing with IPbus. This involves the ability to read/write data from IPbus and process the data bit-wise. The data will determine if it is a read or write request and transmit the requested data to the virtual address register. The code written for the simulation will also be similar to the actual implementation, allowing the code to be reused.

The simulation is written in Python, using an OOP template, meaning each module of the PCS has its own defined class with functions and encapsulated variables. Due to it being a direct connection between the object class and its physical counterpart, expanding the simulation's functionality and adding new functions is intuitive and relatively easy to do.

### 8.2.1 Overview

The simulation is object-oriented and consists of 4 classes and test functions for each. The classes are based on the final FPGA-design considered in subsection 5.3. The MB Hub itself is represented by the *monitor sim manager* which instantiates *monitor module* objects and a *global module* object inside itself. The microcontroller's class is instantiated inside each monitor module class, mimicking how each module on the FPGA maintains its respective microcontroller. Finally, *monitor sim manager* is instantiated in the GUI class that is connected to the MongoDB API. This allows for testing the GUI and configuration of the microcontrollers in the simulation. A class diagram of the simulation is shown in 8.1.

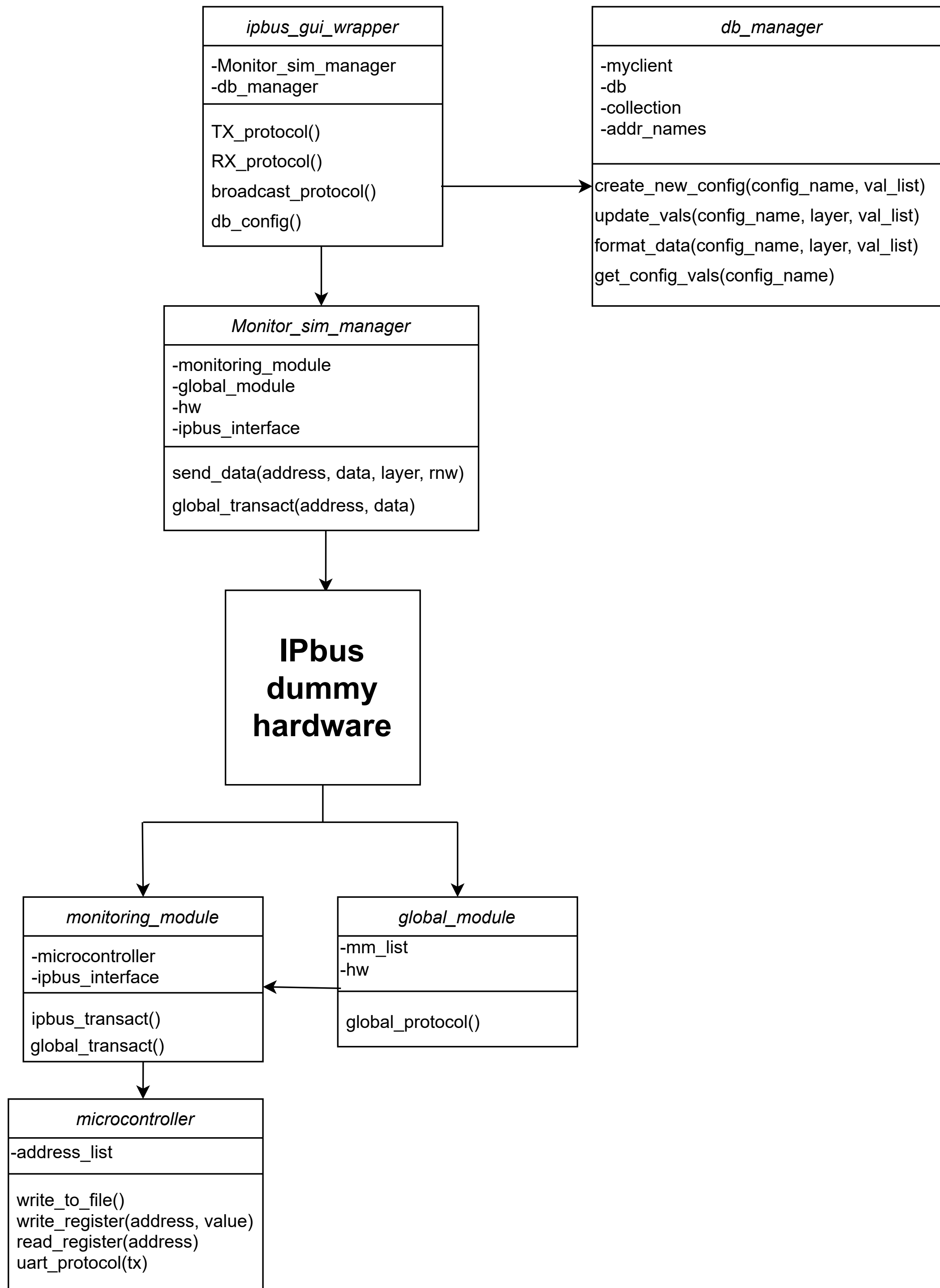


**Figure 8.1:** Class diagram of the simulation, including attributes and variables.

### 8.2.2 Features

The simulation uses IPbus dummy hardware to communicate between the monitor sim manager class and the monitor module class. The dummy hardware is hosted on the computer

localhost and has the same functionality as an IPbus module on an FPGA.

The MB Hub and microcontroller class can perform read and write operations to the microcontroller class's virtual registers. The microcontroller class decodes read/write bit, address and data from the TX value it is given, according to the data header discussed in subsection 5.4. A global module is also simulated, allowing for transmission to every microcontroller using the global transact function in the monitor sim manager.

Using the simulation is done by instantiating the monitor sim manager object in the code and using the send data function to send TX data. The function will return the data from the specified RX register in IPbus. In addition, the microcontroller class will create a text file with its addresses and contents, allowing the user to verify the data.

The simulation was created as a platform to perform tests on IPbus code without needing the actual hardware. However, halfway through the development cycle, the hardware became available, and the development of the simulation was deemed unnecessary. The simulation functionality is therefore outdated, but it can still be used to test simple read and write functions on software if needed.

## 8.3 Testbench

Testbenches have been made to verify the functionality of the APIs and classes comprising the control software. However, due to needing access to the layers themselves, functions testing the entire PCS chain has yet to be made, which means we do not have full coverage on the tests. Additionally, tests for the PCS-chain are unreliable due to the high bit error rate discussed in subsection 8.6.

Testing the entire system can be done by starting the main hub GUI and observing that the control software behaves as expected. When all parts of the DTC are available for testing, and the bit error rate is acceptable, a full-scale, automatic testbench should be developed that allows for the entire system to be quickly tested and verified. This would serve as a standard for the system and allow for quick verification when modifying or expanding existing functions. The testbench development should ideally be based on a testing framework, such as unit testing[31], to efficiently design tests for the control software.

## 8.4 Test setup

The development of the DTC is still ongoing. This implies that we do not have access to the ALPIDE layers or to the necessary hardware to set up all 43 monitor boards for the PCS. In addition, the design of the MB is still under development, and setting up the full PCS is therefore still not possible.

However, we can set up a prototype of the PCS without connection to the ALPIDE-sensors. This prototype is realized with a computer connected to the KCU105 evaluation board, and pins on the board are connected to the microcontroller development kit. An image of the prototype is shown in Figure 5.2.

The test setup can be used to test the MB Hub and, by extension, the microcontroller software. The setup also be used to verify the two communication links in the PCS, which is the IPbus link between control software and MB Hub, and the USART communication between the MB Hub and the microcontroller.

## 8.5 MB Hub Verification

The communication link between the control software and the MB Hub uses the IPbus communication protocol, and studies have shown it to be fast and reliable[23]. However, our communication link uses a custom FPGA design, so it must be tested to verify that it works as intended.

### 8.5.1 Reliability Verification

The IPbus suite has been extensively tested, on several different boards, with no errors recorded after transmitting 10 billion packets[23]. However, for completeness, our IPbus-link must be verified to ensure the correctness of our design. The test involves reading and writing random values to a register in the MB Hub and verifying that the value read was the one written to the register. However, we must be sure that the values are read from the actual register and not from a buffer inside the IPbus module. The test writes and reads values from three different registers to avoid this issue. The test reads and writes from the TX/RX registers in the dummy module, the baud rate register in the global module, and the enable register in the global module.

The test was run for 5 hours, and 10 million reads and writes were performed from each register. There were no errors reported from the communication. This suggests that the MB Hub design is reliable, and suitable for our use.

### 8.5.2 MB Hub Speed Test

The transmission speed between the control software and the MB Hub can significantly impact the speed of the PCS-chain; this property of the MB Hub must therefore be tested and verified. This communication link uses IPbus, and the IPbus user guide states that reading or writing a single word has a latency of approximately 250 $\mu$s[24]. This number might not reflect our setup, so we performed a transmission speed test. The test uses the Time library from Python to measure the write and read speed of the dummy module on the MB Hub.

The TX and RX register of the dummy module was written and read 1 000 000 times. On average, the single write latency was 0.4 ms($\sigma = 0.2$ ms), and the single read latency was also 0.4 ms($\sigma = 0.3$ ms). These measurements deviate from the assumed transmission delay of 250 $\mu$s. The software delay could be the cause of this deviation. The software is run on a laptop computer, which might not be as fast as a desktop computer.

With this new number for the transmission delay, we can redo the configuration and monitoring timing calculations performed in subsection 6.5 and subsection 7.5, respectively. Using Equation 6.4, but set the IPbus delay to be 400 $\mu$s, the delay of configuring one layer becomes:

$$112 \cdot 400 + 56 \cdot 33 = 46648\mu s \approx 46.6ms \tag{8.1}$$

A delay of 46.6 ms is closer to the measured timing of 51 ms and falls within the standard deviation of that test. There is a closer agreement with the calculations and the measurements if we consider the IPbus delay 400 $\mu$s. Similarly, we can calculate the new polling timing from the monitoring process. Using Equation 7.1 for one layer:

$$8 \cdot (400\mu s + 33\mu s + 1 \cdot 400\mu s) = 6664\mu s \approx 6.7ms \tag{8.2}$$

This new calculated timing for polling all data lies within the standard deviation of the measured timing of 8.7 ms. This again shows higher agreement with the calculation and the measurement if we use the measured IPbus delay.

## 8.6 USART Interface Test

The bit error rate that occurs while sending data from the MB Hub to the microcontroller must be verified and ensured to be within an acceptable rate. We can test this communication link by sending data from the control software to a TX register on the MB Hub, which starts the USART-transmission to the microcontroller. We know from previous tests that the communication link between the control software and the MB Hub has a negligible error rate; therefore we will assume all bit errors come from the communication link between the MB Hub and the microcontroller.

The test is done by performing a write to a microcontroller register; we perform 1000 read operations from the register. This gives one sample, and this test is done 100 000 times to give us a mean error rate per 1000 reads. Multiple bit patterns is used to test edge cases such as alternating bits, all 1's, all 0's or random bit patterns. Results of different bit patterns is given from Figure 8.2 to 8.5.

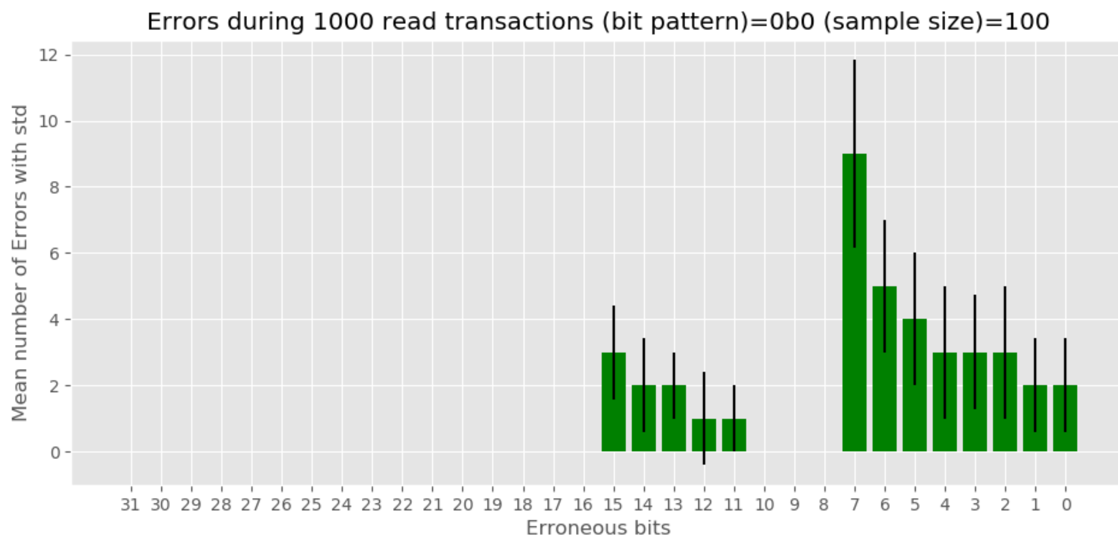


**Figure 8.2:** Histogram showing the mean bit error rate with all 0's bit pattern. Black line is the standard deviation.

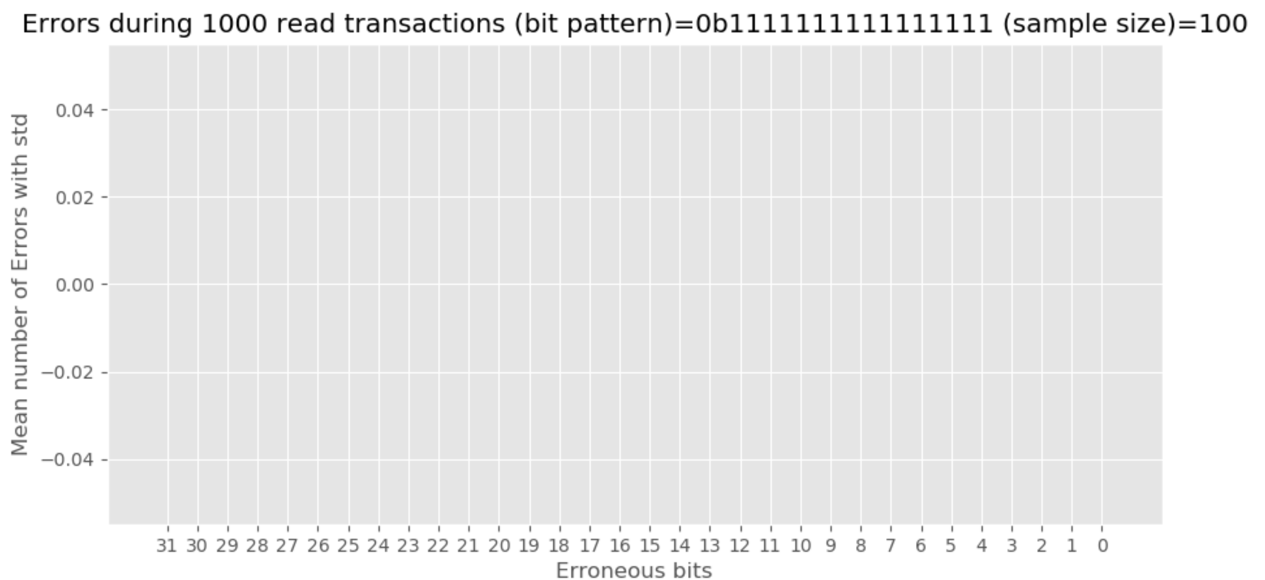


**Figure 8.3:** Histogram showing the mean bit error rate with all 1's bit pattern.

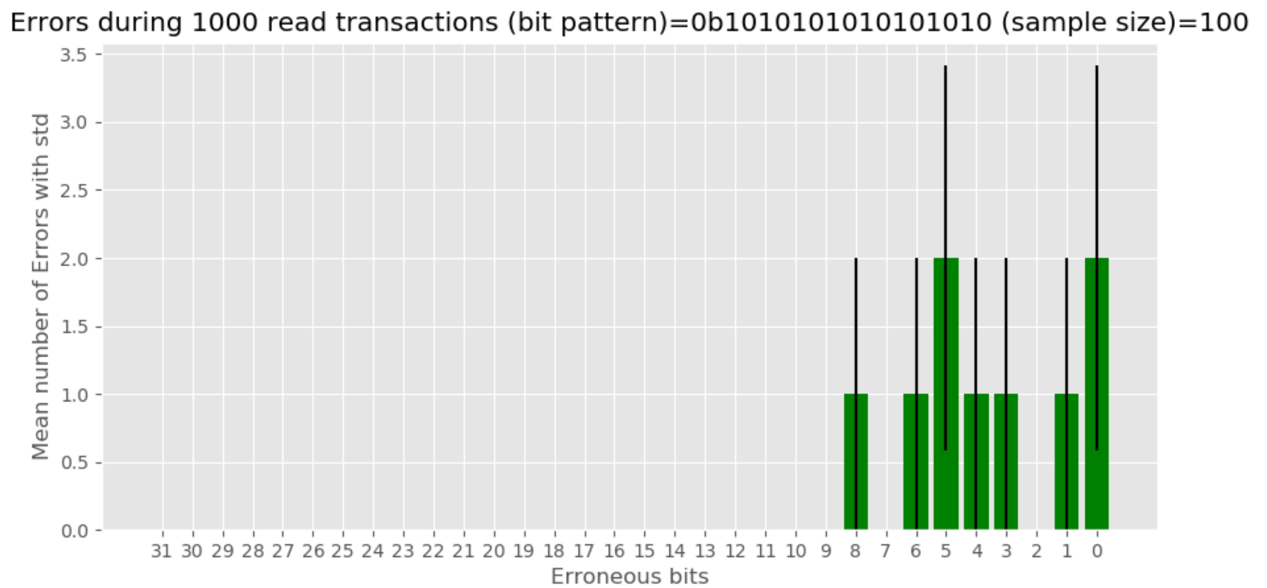


**Figure 8.4:** Histogram showing the mean bit error rate with alternating 1's bit pattern. Black line is the standard deviation.

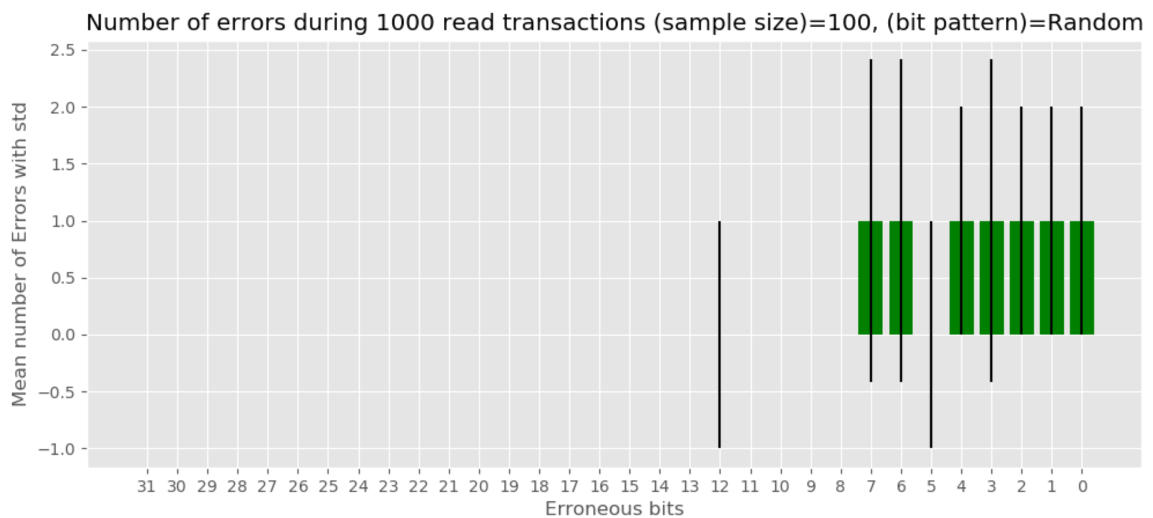


**Figure 8.5:** Histogram showing the mean bit error rate during a random bit read operation. Black line is the standard deviation.

The figures show that all but the "all 1's" test gave us errors. Alternating 1's and random bit pattern have a similar error rate, they match well if we consider the standard deviation of the error measurements. "All 0's" gave a significantly higher error rate than the other patterns.

The all 1's pattern is the outlier, with no errors during the test. This result, combined with the all 0's pattern having an abnormally high error rate, leads to the assumption that the

cause of the errors is misinterpreting 0's to be 1's. Judging from the bits, only the data bits(15:0) have errors, but other than that, no obvious pattern gives us more insight into the cause of the errors.

A system's reliability can be quantified by its Mean Time Between Failure (MTBF), which determines the time span between faults occurring in a system[32]. If we assume random values are being read, then from Figure 8.5, we can assume, on average, one error occurs for every 1000 USART transactions. From subsection 6.5, it was calculated that one transmission from control software to microcontroller took 288 $\mu$s; this means we can expect the MTBF of one USART link to be:

$$MTBF = 1000 \cdot 288 \mu s = 0.288 s \tag{8.3}$$

The entire PCS-system will have a USART-link for every layer, which means we have to consider the MTBF for every link to find the MTBF for the entire system. The formula for the system MTBF can be found by considering 43 subsystems in series[33]. If each subsystem has the same MTBF, then:

$$\frac{1}{43 \cdot \frac{1}{MBTF}} = \frac{1}{43 \cdot \frac{1}{0.288}} = 0.0067 s \tag{8.4}$$

The MTBF of the PCS-system will be approximately 7 ms, which is unacceptable for this project. The MTBF of the PCS should ideally be measured in months, not milliseconds. The conclusion is that the USART communication between the MB Hub and the microcontroller is unacceptable. The error rate must be significantly improved before it can be used in the PCS.

## 8.7 Baud rate error rate

Another aspect of the PCS-prototype that is tested involves using different baud rates for the transmission between the MB Hub and the microcontroller. If erroneous bits are caused by the sampling not hitting the falling edge of the clock, reducing the baud rate should improve the error rate, giving more time for sampling each bit. In the previous section, we observed an error rate of about 0.1% for the data bits. This test was performed with a baud rate of 921600. A test was performed to assess whether a lower baud rate improves the bit error rate. The test result is given in Figure 8.6.

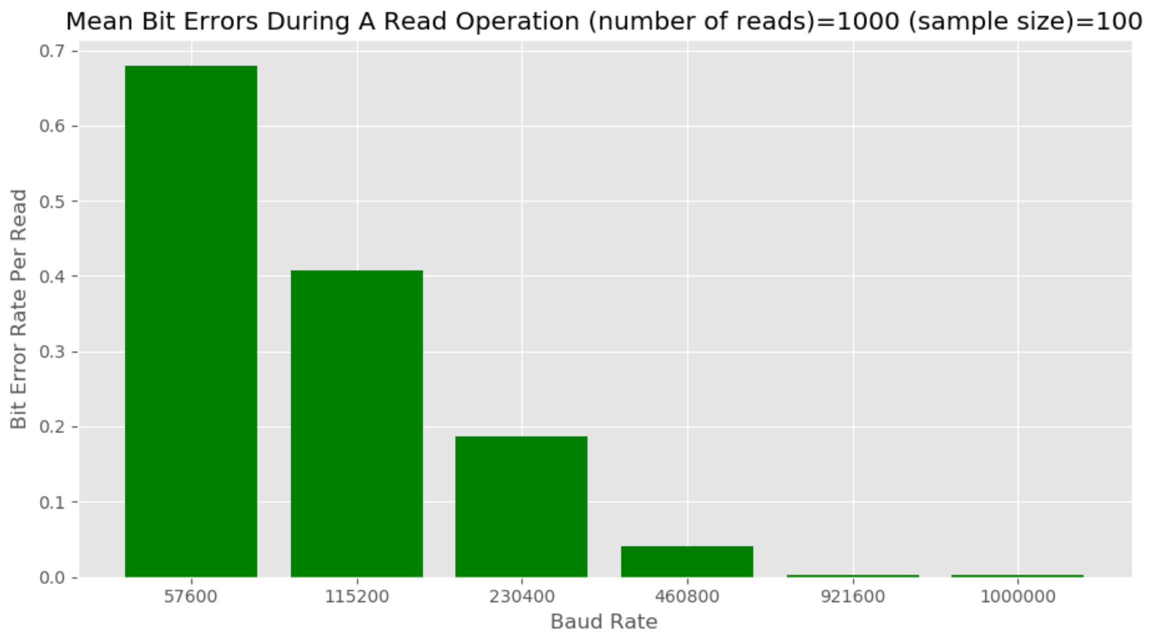


**Figure 8.6:** Histogram showing the amount of errors expected in one transaction for different baud rates.

The figure shows that the lower baud rates increase the error rate, not lower it. The error rate increased from 0.1% to a whopping 70% on average for the lowest baud rate tested. This unexpected result contradicts the behaviour of standard USARTs. The cause of this is not yet known; a possible explanation for the errors can be seen by observing the USART communication with an oscilloscope. Figure 8.7 shows the USART communication between the microcontroller and FPGA while performing a read operation.

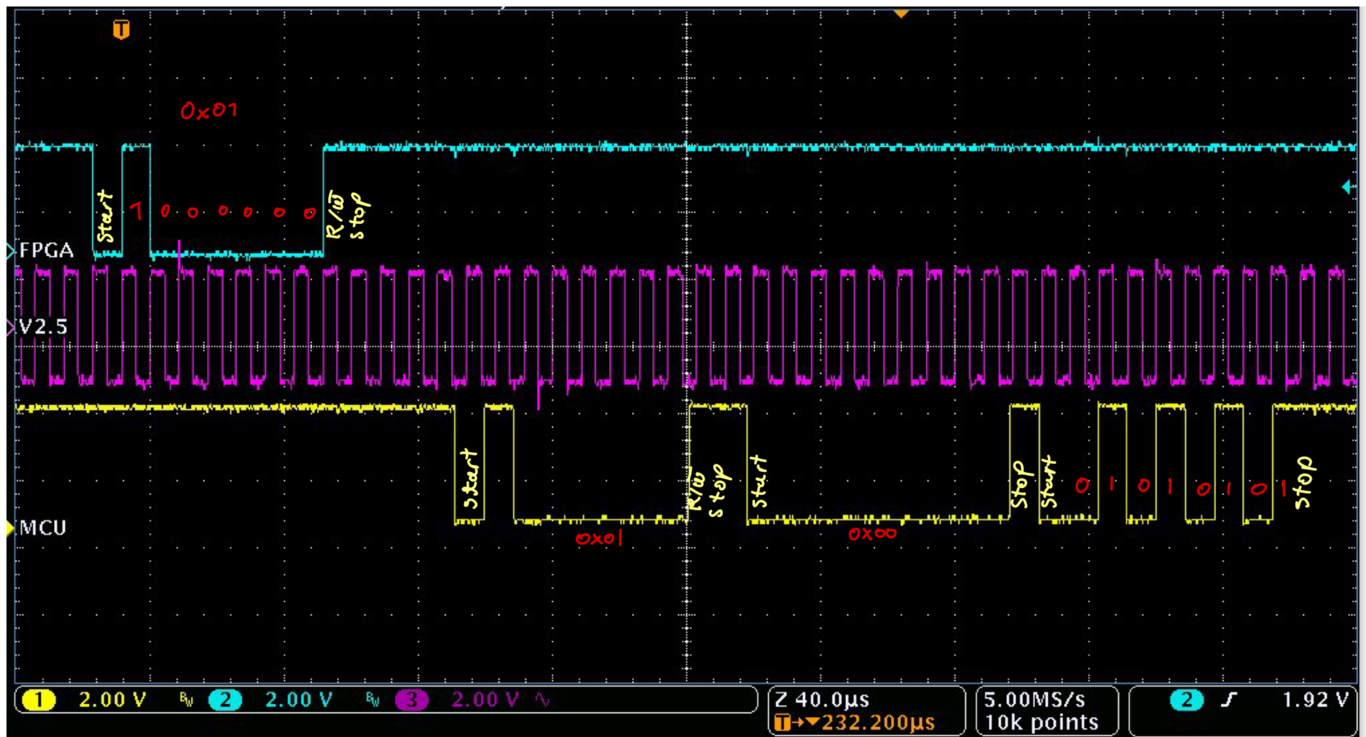


**Figure 8.7:** Oscilloscope image of the FPGA sending a read request to the microcontroller and receiving the data. Captured with a baud rate of 921600.

The figure shows the FPGA sending the read request; the microcontroller then performs a handshake by sending the read request back to the FPGA. After this, it starts transmitting the data inside the two registers associated with address 0x01, which in this case is 0x00 and 0xAA. Observe from the image that the timing of the microcontroller gets gradually skewed throughout the transmission. Based on the image, the last data bits appear to be sent from the microcontroller on the rising edge of the clock, not the falling edge. If a $\Delta$t skews the data transmission for each clock cycle, and this $\Delta$t is proportional to the clock speed, then a longer clock cycle, i.e. a lower baud rate, would create more skew and result in more errors.

The conclusion is that until the underlying issue is resolved, one should use high baud rates to avoid excessive errors. The microcontroller supports baud rates up to 1 000,000, and the standard baud rate closest to it is 921600. Higher baud rates also make the transmission speed faster, which is favourable for the PCS design.

# 9 Discussion

*This chapter describes the functionality of the control software created in this thesis and what is complete and missing. The chapter covers missing features and how the software can be expanded in the future. Finally, a discussion is made on the system's modularity and how we can repurpose it within the PCT project.*

## 9.1 Reusability

The configuration and monitoring systems developed in this thesis have been built using software design standards to make them generic and modular. In addition, the system is layered so that individual parts with specific functions can be reused in other systems. This section discusses how these parts can be reused.

### 9.1.1 Monitoring software

The monitoring software is comprised of *influx_api*, *configuration_api*, and *panda_filter*. The influx API class requires very little modification to be repurposed for other systems. The API has functions for inserting values into the database, and for filtering data points with the *panda_filter* class. Certain tags in the class must be redefined to match the new system it interfaces with, i.e. change "layer" to "flowmeter", for example.

The panda filter class uses Pandas dataframe to filter received data. It currently can only filter data based on the exception process described in subsection 7.6. However, additional functions can be created to perform other filters on the data.

The monitoring API does not require any direct modification to be repurposed to another system. However, it requires a lower level API to interface with, akin to the microcontroller API. In addition, the *monitor_addr* list must be edited to include the desired registers to poll.

### 9.1.2 Configuration software

The configuration software is comprised of *db_manager*, and *configuration_api*. The configuration API is specialized to configure, and power on the PCS; the configuration functions must therefore be reworked for other systems. Subsection 6.2 shows that the configuration process is in two stages, stage 1 configures general registers, and stage 2 performs the ramp-up algorithm. Stage 1 may be suitable to reuse since it only performs basic write operations. However, stage 1 also contains system-specific functions, such as handshake verification, that must be removed or modified.

The *db_manager* is the interface between the Python classes and MongoDB. It is relatively simple; its primary function is to create configuration sets for the PCS. The configuration sets store single values for most registers, but the threshold 2 registers have individual values for each string. This formatting must be revised or removed if we reuse the manager to create configuration sets for other systems in the future.

The lower level APIs that connect to the microcontroller, and the MB Hub uses IPbus to communicate, and other systems that use IPbus can base their design on these APIs. The readout system, in particular, uses IPbus for configuring the ALPIDE-sensors and could base its structure on the PCS configuration API. On the other hand, the cooling system reads data using Moxa modules, meaning the lower level API must be redefined. However, it could still utilize the same concepts, such as having multiple levels of API abstraction.

### 9.1.3 Register Package

The APIs developed for the PCS use namedtuples to process register information, which is described in Subsection 6.1. The namedtuples make it easier to repurpose the APIs to a different system. A developer can define the address map of a different module using the same namedtuple structure, then the APIs would only require small modifications to work with the new system.

## 9.2 Error handling

### 9.2.1 Control Software

The error handling from the control software is not finalized. By "errors," we refer to the microcontroller's error messages, which are discussed in subsection 5.4. Currently, only the monitoring system checks for errors during its polling function. The error handling design has yet to be decided, but this section will review potential implementations.

The configuration and monitoring systems require functions to retrieve error messages. This suggests that an "Error Manager" class should be developed, which contains general functions for handling the errors. Both the configuration and monitoring APIs could use this manager to handle errors, reducing redundant code in the APIs and increasing the software's modularity.

A design choice must be made on storing and displaying the microcontroller's errors. Ideally, the error messages should be stored in a logging database and displayed to the user, but that design decision has yet to be made. Currently, the error messages are not stored; they are only displayed in the monitoring GUI. InfluxDB could store the logging information, and separate tags could indicate each error's severity. Another option for handling logging is Grafana Loki, a log aggregation system that is optimized for Grafana.

### 9.2.2 Microcontroller

The microcontroller software supports custom control functions that can be performed by writing to the control registers on the microcontroller. Utilizing these custom functions could reduce the number of transmissions needed for error handling, increasing the speed of the error handling algorithm. For example, identifying faulty strings would be a cumbersome process for the control software; turning on the strings to test their current consumption individually would require multiple transmissions through the entire PCS-chain. A microcontroller function that automatically turns each string on, one by one, and then measures their current consumption would make the process significantly faster. The control software

would only need to call the function by writing to a register, retrieve the current values for each string, and compare them to the expected ones.

## 9.3 Configuration System

Currently, the broadcast function in the MB Hub is not used in the configuration process. This is due to the structure of the database; the database stores values for every layer. The broadcast functionality would require a separate function that checks if a register value is the same for every layer. If yes, broadcast the data; if not, it must configure the layers separately. This process could add more delay in the transmission in the form of software overhead.

It is therefore required to restructure the database to use the broadcast function effectively. A possible solution is using a "default" configuration set in the database. For example, we would have a single configuration set that is used to configure all layers; however, each layer has an exception list, which overrides the default set. The default set is merged with the exception values, creating the final configuration set. A figure of the possible implementation of a default configuration set is given in Figure 9.1.

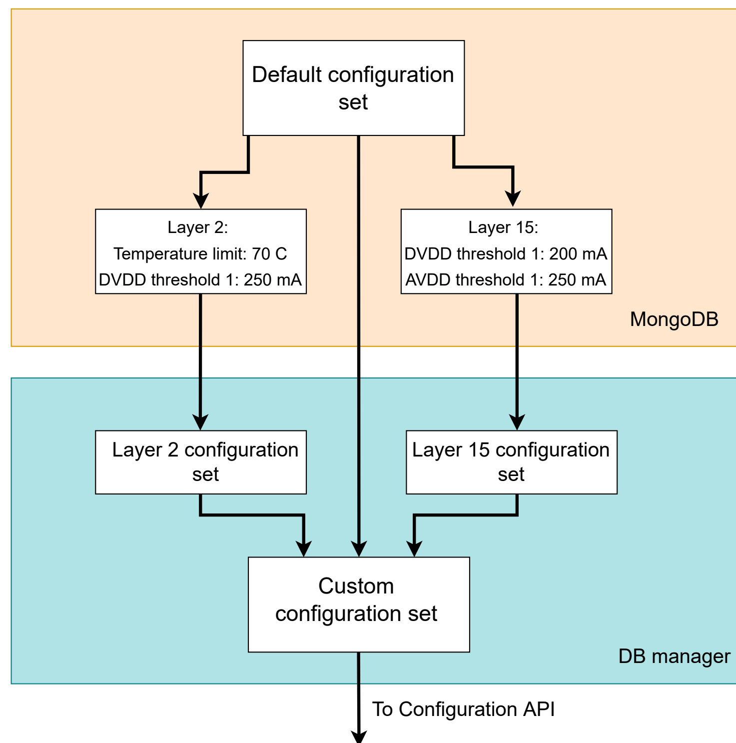


**Figure 9.1:** Block diagram showing the implementation of a "default" configuration set in MongoDB.

The figure shows the default configuration set in the MongoDB database, exception lists for two layers, and the DB manager class that retrieves the configuration set. DB manager will create a configuration set for each layer with an exception list and merge this configuration

set with the default one, which gives us a custom configuration set. This custom set is sent to the configuration API. Restructuring the database like this would also require changing the configuration API to utilize the broadcast function.

The calculations and results from subsection 6.5 show that the configuration timing is already adequate for this project, only spending a second to configure all layers. The "default" database presented in this section could be implemented in the future if timing becomes a problem.

## 9.4 USART communication

The communication link between the MB Hub and the microcontroller is outside the scope of this thesis. However, significant time has been spent on creating verification tests of the PCS, and by extension, the USART communication between the MB Hub and the microcontroller. This section covers the potential sources of the bit errors in the USART communication link.

A 17-hour test was done on the USART communication earlier in the development cycle, and no errors were observed. There were; however, errors observed at lower baud rates. The test setup was then briefly moved to an expo, and after returning, the error rate was observed to be much higher. The error rate was similar to the results shown in subsection 8.6, even at higher baud rates. These results suggest two sources of errors in the communication; the first is dependent on the baud rate of the USART, and the second error source stems from the inconsistent test environment.

The source of the baud rate error is discussed in subsection 8.7, where the conclusion was that a $\Delta$t skew in the transmission for every clock cycle caused the errors, which was exacerbated with lower baud rates.

The results from subsection 8.6 showed that the bits always flipped from 0 to 1, never the other way around. Previous tests of the prototype did not suffer from any errors in the bits at high baud rates, therefore the most probable cause of the errors is the inconsistent test environment, because the errors start occurring after moving the test setup.

## 9.5 Future work

This section covers the remaining work on the system that should be done before it can be incorporated into the pCT-project.

### 9.5.1 Docker Image

The control system uses several different packages and programs to function. The README file in the GitHub repository gives instructions on installing all dependencies and running the program. However, moving the system to a different computer is still cumbersome. Therefore, a Docker Image of the system should be created in future work. Docker is a program that allows for building an "Image", a virtual environment for your program containing all dependencies necessary to run it. By building a Docker Image of a program, moving it to other computers only requires downloading a single docker file and running it.

The main hub GUI, along with its lower level GUIs and APIs, can all be encapsulated in one Docker Image. However, the databases and Grafana will most likely be located on different servers; each should have its own Docker Image with its ports open to the GUI Image.

### 9.5.2 Grafana

The Grafana dashboard has four tabs for each measurement type: DVDD, AVDD, PWELL, and temperature. Each tab contains histograms for each layer, but it is currently impossible to display all tabs on one screen; this design may be cumbersome for the user. They must scroll through the tabs if they want to view multiple measurements simultaneously.

If the Grafana design is not satisfactory to the user in the future, a new dashboard design is warranted. A group of users should create this new design to ensure the dashboard is user-friendly.

### 9.5.3 System parameters

Currently, several system parameters are only defined as variables inside their respective classes. These parameters include ExDev, and ExMax, which affect the ”exception” filtering process, and the batch size of the Influx API. It may be inconvenient for the user to edit many different files to change these parameters. The solution would be to have a separate ”config” file containing parameter values that the user can easily configure. The ”config” file could be implemented in a simple text file or integrated with the configuration database. The relevant classes would then have to be changed to use the values from the ”config” file.

### 9.5.4 Security measures

Security measures should be applied to the control software to ensure unauthorized users are not allowed to delete data or perform actions that could harm the equipment. Two sides of the PCS should implement security measures: the database systems and the threshold values for the MB.

The configuration and monitoring databases will contain a vast amount of data, including monitoring data from several years and many configuration sets for the MBs. This data should not be able to be deleted easily by any user; ideally, only users with administrative privileges should have access to such functions.

The microcontroller on the MB automatically turns off the strings if the temperature or current thresholds are exceeded. However, if these thresholds are excessively high, the microcontroller will never turn off the strings, which can cause damage to the sensors. Therefore, there should be an upper limit on the threshold levels that are hard-coded into the control software, to prevent such accidents.

# 10 Outlook and Conclusion

*This chapter discusses the work that has been done in this thesis and the result of said work. A discussion is made on the future of the power delivery system and how it is related to the other systems of the pCT-project.*

## 10.1 Summary

The objective of this thesis was to create a control system for the power delivery of the pCT-project. The task involved creating a configuration and monitoring system for the power delivery, A Human to Machine Interface to the system in the form of GUIs, and a database system for storing configuration and monitoring data. Additionally, the system needed to be structured with clear levels of abstraction to make it easier to expand upon in the future.

It was decided to create several levels of abstraction of the API using Python classes. These APIs were interfaced with the MB Hub and the Monitoring Board, which comprises the Power Control system. A high-level API for the configuration process was created. This API incorporates a ramp-up algorithm to turn on the ALPIDE-sensors while configuring the Monitoring Board safely. MongoDB was chosen as the database used for the configuration system, and a Python class was designed to interface with the database and format the configuration sets. A GUI was created that allows the user to load configuration sets from the database and create them interactively using the GUI. The timing of the configuration process was calculated and measured to be satisfactory for the project.

A monitoring API was created to poll data and insert it into the database. InfluxDB was chosen for storing data with timestamps, and Grafana was used to display the data to the user. A filtering class was created that allowed for filtering data points based on OSIsoft's exception filter. A simple GUI was also created to set up the monitoring process.

A simulation was created to verify the early designs of the control software, and further tests were performed on the Power Control System prototype to verify the communication link between the control software and the Monitoring Board. However, the tests showed that the reliability of the Power Control System needed to be revised, and a prototype redesign was warranted.

Lastly, a discussion was made on the modularity of the control software and how it may be reused and expanded upon in the future.

## 10.2 Outlook

The control software presented in this thesis is working as intended and is well documented, which serves as a foundation for further development in the future. The system verification must still be performed with actual hardware; ideally, a testbench in the future should be created for the control software, allowing quick software verification. Handling errors from the microcontroller must be designed and implemented, ideally with a logging database, and this system must be integrated with configuration and monitoring processes.

A discussion has been made on the reusability of the control software in the context of other pCT-systems. Using a similar structure and code for the other systems is promising due to the generic control software. In particular, software relating to the databases is not directly tied to PCS and therefore has the potential to be reused in other configuration and monitoring systems. However, the low-level APIs must be redefined for the system it shall interface.

There is still room for optimization for the control system, but tests on the system have shown it more than fast enough for this project. The optimization may become more relevant when we include readout and cooling in the process.

## 10.3 Conclusion

This thesis has presented a control software that can be used for the power delivery system of the pCT-project. This control software consists of several levels of APIs, GUIs, databases, web interfaces, and test functions for the Power Control System. Tests concluded that the control software fulfilled the requirements of the pCT-project.

# A Software

## A.1 Microcontroller AdressMap

## B.1 Adressmap

| Register Name | Adress | Number of bits | Unit | Access | Default value | Description |
|---|---|---|---|---|---|---|
| FW_VERSION | 0x00 | 8 | - | R | - | Current firmware version. |
| ADC_VALUE | 0x02 | 12 | V/V | R/$\overline{W}$ | - | 12-bit ADC value. |
| PT100_READING | 0x04 | 8 | °C | R | - | ADC value converted to degrees celsius. |
| TEMPERATURE_LIMIT | 0x06 | 8 | °C | R/$\overline{W}$ | 100°C | On measuring a temperature above TEMPERATURE_LIMIT the enable signals will be set low. Error 0x01. |
| DAC_VALUE | 0x08 | 10 | V/V | R/$\overline{W}$ | 0x00 | DAC voltage output from the MCU, input from 0x00 to 0x400 creates an output of 0 V to 2.5 V. NOTE: Read about the PWELL generation before changing. DOUBLE NOTE: The DAC utilises the 10 most significant bits! |
| PWELL_VOLTAGE_MCU | 0x0A | 13 | mV | $\overline{W}$ | 0x00 | The desired PWELL voltage in millivolt. Writing a value to this registers triggers the MCU to create the complementary voltage on the PWELL line. |
| DVDD_CURRENT_THRESHOLD1 | 0x0C | 14 | mV | R/$\overline{W}$ | 0x00 | INA3221 DVDD critical threshold. If the DVDD line exceed this current draw the enable signals are set low. Error 0x02 |
| DVDD_CURRENT_THRESHOLD2 | 0x0E | 14 | mA | R | 0x00 | DVDD warning threshold. Error 0x03 |
| DVDD_VOLTAGE | 0x10 | 13 | mV | R | - | DVDD shunt resistor measured voltage. |
| DVDD_CURRENT | 0x12 | 13 | mA | R | - | DVDD shunt resistor measured current. |
| AVDD_CURRENT_THRESHOLD1 | 0x14 | 14 | mA | R/$\overline{W}$ | 0x00 | INA3221 AVDD critical threshold. If the AVDD line exceed this current draw the enable signals are set low. Error 0x04 |
| AVDD_CURRENT_THRESHOLD2 | 0x16 | 14 | mA | R/$\overline{W}$ | 0x00 | INA3221 AVDD warning threshold. Error 0x05 |
| AVDD_VOLTAGE | 0x18 | 13 | mV | R | - | AVDD Voltage |
| AVDD_CURRENT | 0x1A | 13 | mA | R | - | AVDD CURRENT |
| PWELL_CURRENT_THRESHOLD1 | 0x1C | 14 | mA | R/$\overline{W}$ | 0x00 | INA3221 PWELL critical threshold. If the PWELL line exceed this current draw the enable signals are set low. Error 0x06 |
| PWELL_CURRENT_THRESHOLD2 | 0x1E | 14 | mA | R/$\overline{W}$ | 0x00 | INA3221 AVDD warning threshold. Error 0x07 |
| PWELL_VOLTAGE_INA3221 | 0x20 | 14 | mV | R | - | PWELL voltage measured by the INA3221 |
| PWELL_CURRENT | 0x22 | 14 | mV | R | - | PWELL Current measured by the INA3221 |
| ENABLE_SIGNALS | 0x24 | 12 | - | R/$\overline{W}$ | 0x00 | Each bit represents an enable line controlling the power to a string. String 0 is tied to LSB. |
| STRING_DVDD_CURRENT_VALUE[n] | 0x26+[2n] | 13·12 | mA | R | 0x00 | The DVDD current values for each string after the scan flag has been asserted. In total 12 register with 13 bytes each. Each string register takes two bytes, and the address for string n is offset by 2n bytes. |

| Register Name | Adress | Number of bits | Unit | Access | Default value | Description |
|---|---|---|---|---|---|---|
| STRING_AVDD_CURRENT_VALUE[n] | 0x3E+[2n] | 13·12 | mA | R | 0x00 | The AVDD current values for each string after the scan flag has been asserted. In total 12 register with 13 bytes each. Each string register takes two bytes, and the address for string n is offset by 2n bytes. |
| STRING_PWELL_CURRENT_VALUE[n] | 0x56+[2n] | 13·12 | mA | R | 0x00 | The PWELL current values for each string after the scan flag has been asserted. In total 12 register with 13 bytes each. Each string register takes two bytes, and the address for string n is offset by 2n bytes. |

| Register Name | Adress | Number of bits | Unit | Access | Default value | Description |
|---|---|---|---|---|---|---|
| CTRL1 | 0x6E | 8 | - | R/$\overline{\text{W}}$ | - | Control register A |
| CTRL2 | 0x70 | 8 | - | R/$\overline{\text{W}}$ | - | Control register B |
| CTRL3 | 0x72 | 8 | - | R/$\overline{\text{W}}$ | - | Control register C |
| ERROR_COUNT | 0x74 | 8 | - | R/$\overline{\text{W}}$ | 0x00 | Amount of errors since last clear. Is cleared by writing 0x00 to this register. |
| ERROR_MSG | 0x76 | 128 | - | R | 0x00 | Error messages stored in sequence, each byte is an error message. The most reccent error is placed in LSB. Automatically cleared by clearing ERROR_COUNT |

## B.2 Error Codes

| Error name | Error code | Description |
|---|---|---|
| Temperature limit reached | 0x01 | The ADC value is reported to be above the temperature set by the TEMPERATURE_LIMIT register. |
| DVDD critical current | 0x02 | Critical current reached on DVDD line. |
| DVDD warning current | 0x03 | Warning current reached on DVDD line. |
| AVDD critical current | 0x04 | Critical current reached on AVDD line. |
| AVDD warning current | 0x05 | Warning current reached on AVDD line. |
| PWELL critical current | 0x06 | Critical current reached on PWELL line. |
| PWELL warning current | 0x07 | Warning current reached on PWELL line. |
| Write/Read denied | 0x08 | Tried to write or read a register that should not have been read or written. |
| String current error | 0x09 | Large current draws from the strings recorded. |
| Enable scan error | 0x0A | Large current draws from a single string recorded during the enable scan. |

## B.3 Control Registers

| ResRegs | | | | ErrorRes | SoftStart | EnScan | EnOff |
|---|---|---|---|---|---|---|---|

Figure B.1: CTRL1 register.

**Bit 7 - ResRegs:** Reset Registers
Resets all registers back to its default values(check the registermap for default values).

**Bit 6 - :**

**Bit 5 - :**

**Bit 4 - :**

**Bit 3 - ErrorRes:** Reset Errormessages
Clears the error message registers completely and resets the ERROR_COUNT pointer to 0.

**Bit 2 - SoftStart:** Soft Startup
Soft startup initialization. On asserting the bit the MCU turns off all enable signals and writes the current and voltage values to the INA modules. The MCU loops through the strings one by one and saves the current draw. On detection of large current draws from a string error 0x09 is written to the error register.

**Bit 1 - EnScan:** Enable Scan
Performs a scan of all the strings while logging the current values in the STRING_CURRENT_VALUEn register. The values are not reported back automatically. On completion the enable signals are tied low(off).

**Bit 0 - EnOff:** Enable Off
Turns off all enable lines. A bit faster than writing to the ENABLE_SIGNALS register as it is stored in a single byte, also ensures that all values are 0.

## A.2 FPGA register map

| Name | Register address | Width | Access | Type | Default Value | Description |
|---|---|---|---|---|---|---|
| GLOBAL_MODULE (module address 0x00) | | | | | | |
| GLOBAL_CONTROL | 0x01 | 32 | W | STAT | 0x00 | Not in use |
| GLOBAL_STATUS | 0x02 | 32 | R | STAT | 0x00 | Not in use |
| GLOBAL_WRITE | 0x03 | 32 | W | STAT | 0x00 | Writes data to every com_module in the system.<br>[31] R/$\overline{W}$ bit towards PCU<br>[30..24] Not in use<br>[23..16] PCU reg address<br>[12..0] Data to PCU |
| GLOBAL_RESET | 0x05 | 32 | W | CMD | 0x00 | Resets all modules except the global_module |
| COM_ENABLE_0 | 0x08 | 32 | R/W | STAT | 0xFFFFFFFF | These 32 bits enable com_modules[1→32]<br>'1' = ENABLED<br>Bit[n] controls com _module_[1 + n ] |
| COM_ENABLE_1 | 0x09 | 32 | R/W | STAT | 0xFFFFFFFF | These 32 bits enable com_modules[33→64]<br>'1' = ENABLED<br>Bit[n] controls com _module_[33 + n ] |
| SYSTEM CLOCK SPEED | 0x06 | 32 | R/W | STAT | 0x1D905C0 | The speed of the clock used by the com_modules.<br>This is used to calculate the baud rate towards the PCUs.<br>Default: 31Mhz (clock from IPbus) |
| PSU BAUD RATE | 0x07 | 32 | R/W | STAT | 0x1C200 | The baud rate to use towards the PCUs |
| COM_MODULE (module address 0x01 – 0x40) | | | | | | |
| CONTROL | 0x01 | 32 | W | STAT | 0x00 | Not in use |
| STATUS | 0x02 | 32 | R | STAT | 0x00 | Not in use |
| WRITE | 0x03 | 32 | W | STAT | 0x00 | Writes data to the corresponding PCU.<br>[31] R/$\overline{W}$ bit towards PCU<br>[30..24] Not in use<br>[23..16] PCU reg address<br>[12..0] Data to PCU |
| READ | 0x04 | 32 | R | STAT | 0x00 | Reads a 32-bit vector from the corresponding PCU.<br>[31] R/$\overline{W}$ bit towards PCU<br>[30..24] Not in use<br>[23..16] PCU reg address<br>[12..0] Data to PCU |
| RESET_FIFO | 0x05 | 32 | W | CMD | 0x00 | Resets both TX and RX FIFO to clear them. |
| DUMMY_MODULE (module address 0x43) | | | | | | |
| WRITE | 0x03 | 32 | W | STAT | 0x00 | Write anything to store a new value |
| READ | 0x04 | 32 | R | STAT | 0xDEADBEEF | Read back the stored value |

# B Setting the Environment

The control software is made of many libraries and programs that work together, this section will describe how to install and set up all dependencies involved in the PCS.

## B.1 Python libraries

The Python programs uses several libraries for communication and to process data. Most of these can be installed using pip, but some require more setup. The following libraries are required:

### B.1.1 Bitstream

Bitstream, used to perform bitwise operations on data from FPGA. Can be installed with:

```
pip install bitstream
```

### B.1.2 Qt5 Designer

Qt5 designer, GUI program to design custom GUIs, installed with:

```
sudo apt-get install qttools5-dev-tools
sudo apt-get install qttools5-dev
```

the program can then be opened with the command:

```
designer
```

The program creates .ui files which can be converted to python files using the command:

```
pyuic5 -o example_gui.py example_gui.ui
```

First parameter after -o is the name of the python file to be created and the second parameter is the name of the .ui file to be converted.

### B.1.3 pyMongo

pyMongo is the API between the python programs and the Mongo database. Can be installed with:

```
pip install pymongo
```

### B.1.4 InfluxDB

InfluxDB is a time series database used for the monitoring system. They provide several different local versions depending on the operating system. They can be found here:

```
https://portal.influxdata.com/downloads/
```

For this project, the linux binaries option was used to install the database:

```
wget https://dl.influxdata.com/influxdb/releases/influxdb2-2.4.0-linux-amd64.tar.gz
tar xvfz influxdb2-2.4.0-linux-amd64.tar.gz
```

The influx database can then be started with the command:

```
influxd
```

Influx is by default on port 8086 on localhost.

### B.1.5 Influx Python Client

The InfluxDB client for Python. API for writing and reading data to the Influx database using python. Can be installed with:

```
pip install influxdb-client
```

the Python client will need an API token to connect to the database, this can be retrieved from the Influx web interface, in the data section, under python client library. This page also contains examples of various operations one can perform with the Influx python API.

# C Typical errors

This section will cover common mistakes and pitfalls while using the software described in this thesis. Some may appear obvious, but it is documented here for the sake of completeness.

## C.1 Bit Error Rate and Masking

The repository contains 3 tests for testing bit error rate in the transmission from control software to microcontroller, and back. These tests use the microcontroller API class when transmitting data, and the API masks the data received, only keeping the data bits. One must disable the masking done in the microcontroller API before performing these tests or the address and RnW bits will always turn out erroneous in the tests.

## C.2 InfluxDB Retention Policy

The Influx web interface allows for setting the retention policy of the buckets in the database. The retention policy determines how long the database should bookkeep the data points from the monitoring process. There is also an option to not have a retention on the database, meaning it will store data points indefinitely. Storing the data points indefinitely can lead to a storage space problem, where one cannot star the server due to memory limits. There is no easy, immediate fix to this, I was not able to find a way to delete the data points, the "easiest" solution found was to re-install InfluxDB. There likely exist command lines that can fix this problem, but I could not find one while trying to fix this issue.

To avoid this issue altogether, always make sure that there is a set retention policy for every bucket in InfluxDB. The retention policies can be manually set through the Influx web interface.

## C.3 Bitstream

"Bitstream" is a Python library used for performing bit wise operations on the data sent and retrieved in the PCS-chain. It is primarily used to go through each bit of a bitstream, and it works by defining a bitstream object with a set bit-length and value. For whatever reason, if the bit-length is exactly high enough for the value inserted, the object will complain that the value is too high for the given bit length, i.e. it will return an error if the bit length is 3 and the inserted value is 5(0b101). Solution to this was to set the bit-length to be one more than what is needed, i.e. if a 32 bit vector must be analyzed, the length of the vector is set to 33. The consequence of the extra bit length is that each Bitstream will have a leading zero.

Bitstream works by having a pointer that starts at the Most Significant Bit (MSB), and when reading one bit, the pointer moves to the next bit in the stream. The leading zero in the Bitstream can therefore be circumvented by performing an initial read to move the pointer past the leading zero before starting the bitwise operations.

## C.4 IPbus Address Values

The IPbus address values are set in the XML-files, which determine the address value for the modules, as well as the addresses inside said modules. An unusual bug was discovered while testing the IPbus system using a dummy module. The first was that performing write requests to module 10 would also perform the same operation to module 1. Writing to module 12 would also write to module 2, and so on.

The cause for this was found to be the address values to each module. Module 1 had address 0x01, module 2 had 0x02, and so on, until module 10, which had address 0x10. I forgot while making the address values that these were hexadecimal values, not decimals, so I skipped straight to 0x10. In theory this should not have an effect on anything in the IPbus system, all that is important is that the address values are unique, but in this instance, changing module 10 to have address 0x0A fixed the problem. The underlying cause of this issue was never discovered, so efforts should be made to ensure that address values are sequential and consistent.

The cause of the bug was not determined, but the fault could have been in the use of the dummy module, which served as a hardware simulator.

## C.5 Grafana

Setting up graphs and queries in Grafana can be time consuming and performing operations on all 43 displays on the dashboard is very repetitive and inefficient. A more efficient way of modifying many graphs in the dashboard at once is not to directly use the Grafana dashboard. Instead, download the JSON-format of the dashboard, this can be accessed by going into the settings menu on a specific dashboard. Then go to "JSON Model" and copy the the code there. Put the code in a text editor, such as Visual Studio Code, and manipulate the dashboard from there. Changing variable names or query requests of multiple graphs can be done relatively easy using this method.

When done editing the file, go to the Dashboard menu on the Grafana web interface and click import:

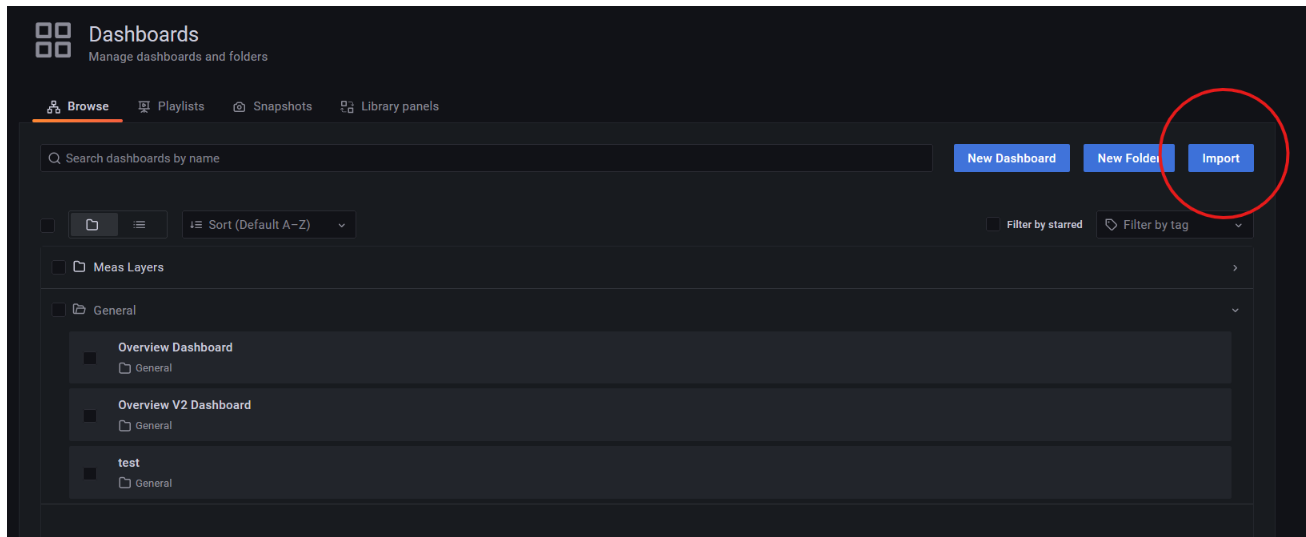


**Figure C.1:** Grafana dashboard highlighting the import button.

From there, enter your edited JSON-file and create the new dashboard, note that it will require a different ID value than the dashboard it was copied from.

## C.6 FPGA FIFOs

Strange behaviour was detected in the FPGA design whenever an attempt was made to read from an already empty FIFO. This behaviour added seemingly random values into the FIFOs while performing read and write requests. This bug should be fixed in the latest version of the bit-file for the FPGA, but to ensure consistency in the tests of the PCS-chain, a check is done in the tests to ensure that we are not reading an empty FIFO. The tests check if received data is equal to zero, if it is, then the FIFO was read too soon. This may happen if the FPGA does not get a response from the microcontroller in time or if the baud rate is too low compared to the speed of the read/write requests from the control software.

Also important to note, if the FIFO was read too soon, the rest of the transmissions will fall out of sync and will not match the expected result, rendering the test invalid. That is also why it is important to perform a reset of the modules for each test performed, to ensure that all FIFOs are empty when starting a new test.

# References


[1] David A Jaffray and Mary K Gospodarowicz. "Radiation Therapy for Cancer". en. In: *Cancer: Disease Control Priorities, Third Edition (Volume 3)*. Washington (DC): The International Bank for Reconstruction and Development / The World Bank, Nov. 2015.

[2] Stanford University Unknown Photographer. *Linear Accelerator to Treat Retinoblastoma 1957*. Visited on: 30.09.22. URL: https://visualsonline.cancer.gov/details.cfm?imageid=1924.

[3] Particle Therapy Co-Operative Group. *Particle therapy facilities in clinical operation*. Visited on: 22.10.22. URL: https://www.ptcog.ch/index.php/facilities-in-operation-restricted.

[4] J et al. Alme. "A High-Granularity Digital Tracking Calorimeter Optimized for Proton CT." In: *Front. Phys* (2020). DOI: 10.3389/fphy.2020.568243. URL: https://www.frontiersin.org/articles/10.3389/fphy.2020.568243/full.

[5] *Estimated number of new cases in 2020, worldwide, both sexes, all ages (excl. NMSC)*. Visited on: 23.08.22. URL: https://gco.iarc.fr/today/online-analysis-table.

[6] H.W. Young. *Popular Electricity and the World's Advance*. v. 2. Popular Electricity Publishing Company, 1909. URL: https://books.google.no/books?id=ow7OAAAAMAAJ.

[7] Brook Itzhak. "Late side effects of radiation treatment for head and neck cancer". In: *Radiat Oncol J* 38.2 (2020), pp. 84–92. DOI: 10.3857/roj.2020.00213. eprint: http://www.e-roj.org/journal/view.php?number=1453. URL: http://www.e-roj.org/journal/view.php?number=1453.

[8] Bulent Aydogan et al. "A dosimetric analysis of intensity-modulated radiation therapy (IMRT) as an alternative to adjuvant high-dose-rate (HDR) brachytherapy in early endometrial cancer patients". In: *International Journal of Radiation Oncology*Biology*Physics* 65.1 (2006), pp. 266–273. ISSN: 0360-3016. DOI: https://doi.org/10.1016/j.ijrobp.2005.12.049. URL: https://www.sciencedirect.com/science/article/pii/S0360301606000861.

[9] *Computed Tomography (CT)*. Visited on: 24.08.22. URL: https://www.fda.gov/radiation-emitting-products/medical-x-ray-imaging/computed-tomography-ct.

[10] Daniela Schulz-Ertner and Hirohiko Tsujii. "Particle Radiation Therapy Using Proton and Heavier Ion Beams". In: *Journal of Clinical Oncology* 25.8 (2007). PMID: 17350944, pp. 953–964. DOI: 10.1200/JCO.2006.09.7816. eprint: https://doi.org/10.1200/JCO.2006.09.7816. URL: https://doi.org/10.1200/JCO.2006.09.7816.

[11] Sebastian Zschaeck et al. "PRONTOX – proton therapy to reduce acute normal tissue toxicity in locally advanced non-small-cell lung carcinomas (NSCLC): study protocol for a randomised controlled trial". In: *Trials* 17.1 (Nov. 2016), p. 543. ISSN: 1745-6215. DOI: 10.1186/s13063-016-1679-4. URL: https://doi.org/10.1186/s13063-016-1679-4.

[12] Harald Paganetti. *Proton Beam Therapy*. 2399-2891. IOP Publishing, 2017. ISBN: 978-0-7503-1370-4. DOI: 10.1088/978-0-7503-1370-4. URL: https://dx.doi.org/10.1088/978-0-7503-1370-4.

[13] Don F. DeJongh et al. "A comparison of proton stopping power measured with proton CT and x-ray CT in fresh postmortem porcine structures". In: *Medical Physics* 48.12 (2021), pp. 7998–8009. DOI: https://doi.org/10.1002/mp.15334. eprint: https://aapm.onlinelibrary.wiley.com/doi/pdf/10.1002/mp.15334. URL: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1002/mp.15334.

[14] Harald Paganetti. "Range uncertainties in proton therapy and the role of Monte Carlo simulations". In: *Physics in Medicine and Biology* 57.11 (May 2012), R99–R117. DOI: 10.1088/0031-9155/57/11/r99. URL: https://doi.org/10.1088/0031-9155/57/11/r99.

[15] Rebecca Henderson and Richard G. Newell. *Accelerating Energy Innovation: Insights from Multiple Sectors*. University of Chicago Press, 2011.

[16] Atefeh Zare and M. Tariq Iqbal. "Low-Cost ESP32, Raspberry Pi, Node-Red, and MQTT Protocol Based SCADA System". In: *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*. 2020, pp. 1–5. DOI: 10.1109/IEMTRONICS51293.2020.9216412.

[17] Stuart G. McCrady. *Designing SCADA Application Software*. Elsevier Inc., 2013.

[18] Zhang Lijun et al. "An Integrated Software for the Performance Study of PET Detector with Neural Network Position Estimators". In: vol. 1. Jan. 2011, pp. 311–313. DOI: 10.1109/WCSE.2010.147.

[19] N Krah et al. "A comprehensive theoretical comparison of proton imaging set-ups in terms of spatial resolution". In: *Physics in Medicine, Biology* 63.13 (July 2018), p. 135013. DOI: 10.1088/1361-6560/aaca1f. URL: https://doi.org/10.1088/1361-6560/aaca1f.

[20] Tea Bodova. "High-Speed Signal and Power Distribution of a Digital Tracking Calorimeter for Proton Computed Tomography". In: *University of Bergen* (2020).

[21] Ola Slettevoll Grøttvik. "Design and Implementation of a High-Speed Readout and Control System for a Digital Tracking Calorimeter for proton CT". In: *University of Bergen* (2020).

[22] Jakob Hauser and Martin Eggen. "FPGA solution for communication between IPbus and a power control unit using a custom USART". In: *Høgskulen på Vestlandet* (2022).

[23] C. Ghabrous Larrea et al. "IPbus: a flexible Ethernet-based control system for xTCA hardware". In: *Journal of Instrumentation* 10.02 (Feb. 2015), pp. C02019–C02019. DOI: 10.1088/1748-0221/10/02/c02019. URL: https://doi.org/10.1088/1748-0221/10/02/c02019.

[24] *uHAL quick tutorial*. Visited on: 30.09.22. URL: https://ipbus.web.cern.ch/doc/user/html/software/uhalQuickTutorial.html.

[25] Birger Olsen. "Power and Monitor Solution for the Proton Computed Tomography Project". In: *University of Bergen* (2022).

[26] *Hardware sizing guidelines*. Visited on: 16.10.22. URL: https://docs.influxdata.com/influxdb/v1.8/guides/hardware_sizing/.

[27] *Optimize writes to InfluxDB*. Visited on: 15.09.22. URL: https://docs.influxdata.com/influxdb/v2.4/write-data/best-practices/optimize-writes/#batch-writes.

[28] *Exception reporting attributes*. Visited on: 12.10.22. URL: https://docs.osisoft.com/bundle/pi-universal-interface-uniint-framework/page/exception-reporting-attributes.html.

[29] *Exception deviation*. Visited on: 15.09.22. URL: https://docs.osisoft.com/bundle/pi-server/page/exception-deviation.html.

[30] *Compression testing*. Visited on: 22.10.22. URL: https://docs.osisoft.com/bundle/pi-server/page/compression-testing.html.

[31] Python Software Foundation. *Unit testing framework*. Visited on: 22.10.22. URL: https://docs.python.org/3/library/unittest.html.

[32] Alan S. Morris and Reza Langari. "Chapter 12 - Measurement Reliability and Safety Systems". In: *Measurement and Instrumentation*. Ed. by Alan S. Morris and Reza Langari. Boston: Butterworth-Heinemann, 2012, pp. 291–316. ISBN: 978-0-12-381960-4. DOI: https://doi.org/10.1016/B978-0-12-381960-4.00012-7. URL: https://www.sciencedirect.com/science/article/pii/B9780123819604000127.

[33] Don L. Lin. *Reliability Characteristics for Two Subsystems in Series or Parallel*. Aurora Consulting Engineering LLC, 2017.