An Automated DRS4-based Data Acquisition System for Characterizing Scintillators Coupled to Silicon Photomultipliers

Authors:
Thomas G. B. Mavropoulos
Sigve Bjørhovde Heggedal

Supervisors:
Ilker Meric
Stein-Arild Tjugum

Bachelor's thesis in Automation and Robotics at

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Spring 2022

Western Norway University of Applied Sciences

Preface

This bachelor's report is written by Thomas Bjørhovde Mavropoulos and Sigve Bjørhovde Heggedal, at Western Norway University of Applied Sciences (HVL) campus Bergen. The project was presented to us in the late autumn of 2021, and we started working on it early January 2022.

Our project came to us through our internal supervisor in this Bachelor's thesis, Ilker Meric, who obtained it from the company Roxar. Stein-Arild Tjugum, representing Roxar, has been our external supervisor. Our project is the successor of a former Bachelor's project, that was also carried out by students at HVL in cooperation with Roxar.

This is a project that has introduced us to radiation physics on a higher level than we had previously been exposed to throughout our education. Although the physics is fundamental for the inner workings of the detector, our objective has primarily demanded the use of disciplines that we were more familiar with beforehand. That includes subjects such as electric circuits, instrumentation and programming, which have all been part of our curriculum at HVL. In addition to this we had some prior experience in the realm of 3D-modelling and -printing, which has proven to be instrumental in completing our task.

We would like to thank our supervisors Ilker Meric of HVL and Stein-Arild Tjugum of Roxar, for the opportunity to work on this project, as well as their counsel and direction throughout. We would also like to thank HVL for access to the premises and materials necessary to complete this project.

Abstract

INTRODUCTION. Silicon photomultipliers (SiPM) are a technology that opens the door to new innovation. The SiPM has advantages over older technologies, such as photomultiplier tubes. However, it is necessary to determine the feasibility of adopting such technologies into systems designed to operate in a wide variety of environments for prolonged time. This requires reliable data acquisition regarding their performance.

AIM. We have aimed to develop an automated, modular and easy-to-use data acquisition system, that would assist Roxar in exploring the performance of silicon photomultiplier modules in conjunction with scintillators and radioactive sources. Reproducibility at all levels was emphasized.

METHODS. A number of constraints were balanced throughout the design process to meet the project's requirements. We detail these constraints and design choices as they pertain to the Domino Ring Sampler 4 Evaluation Board's (DRS4) software, the enclosure housing the silicon photomultiplier module, scintillator and other components, as well as the security measures taken to protect the silicon photomultiplier.

RESULTS. We have produced a lightweight and extensible system. It is capable of collecting relevant data over time, storing it in a format commonly used in data analysis, and displaying the data in useful graphs. This process is completed automatically after first being initialized. The enclosure design ensures the repeatability of experiments and safety of the sensitive silicon photomultiplier.

CONCLUSION. The resultant system does indeed meet the project's requirements. The overall design is such that it is reliable and easy to use for automation, delivers reproducible data and supports others developing it further.

Keywords: Gamma Detection, Scintillation, Photomultiplier, Domino Ring Sampler

Sammendrag

INTRODUKSJON. Silisium fotomultiplikatorer (SiPM) er en teknologi som åpner døren til ny innovasjon. SiPM har fordeler fremfor eldre teknologier som fotomultiplikatorrør. Imidlertid er det nødvendig å utforske gjennomførbarheten av å ta i bruk slike teknologier i systemer designet for å operere i en lang rekke miljøer over lengre tid. Det krever pålitelig datainnsamling for å kunne si noe om ytelsen deres.

MÅL. Vi har hatt som mål å utvikle et automatisert, modulært og brukervennlig datainnsamlingssystem for å hjelpe Roxar med å utforske ytelsen til silisiumfotomultiplikatormoduler opp mot ulike scintillatorer og radioaktive kilder. Reproduserbarhet på alle nivåer er spesielt vektlagt.

METODER. Vi har gjennom hele designprosessen balansert ulike hensyn for å best mulig møte prosjektkravene. Vi beskriver hvilke hensyn og designvalg vi har tatt med tanke på bruk av Domino Ring Sampler 4 Evaluation Board (DRS4) sin programvare, kabinettet som huser silisiumfotomultiplikatormodul, scintillator og andre komponenter. Vi gjør også rede for sikkerhetstiltakene som er tatt for å beskytte silisiumfotomultiplikatoren.

RESULTAT. Resultatet av dette prosjektet er et smidig og utvidbart system som både kan samle inn relevante data over tid, lagre dataen i et format som er populært for dataanalyse, og vise disse dataene som grafer. Denne prosessen fullføres automatisk etter initialisering. Kabinettdesignet sikrer repeterbarheten av eksperimenter og sikkerheten til den sensitive silisiumfotomultiplikatoren.

KONKLUSJON. Det resulterende systemet oppfyller prosjektkravene. Det overordnede designet er slik at det er enkelt og pålitelig å bruke til automatisering, leverer repeterbare data og åpner for videreutvikling av andre.

Contents

1 Introduction

In this thesis we aim to automate the process of data acquisition, as well as evaluate the operational parameters, of silicon photomultiplier (SiPM) modules used with scintillators for gamma detection.

1.1 Contracting entity

The company Roxar Emerson, previously Roxar Flow Measurement, concern themselves with the development of products and solutions for the oil and gas industry. Among the type of automation solutions the oil and gas industry requires is reliable flow measurement in their piping. To this end, Roxar has developed several products, such as their Multiphase Flow Meter.

Roxar is interested in exploring the viability of emerging technologies, such as silicon photomultipliers, and the application of such technologies. The hope is that this could lead to the development of new products with gains in precision and lower costs of manufacture.

1.2 Project requirements

We have been asked to develop an automated solution to assist with the exploration of scintillating materials in conjunction with silicon photomultiplier components. The end goal is an accessible system able to accurately gather relevant data from these materials and components, that can deliver reproducible results, and is able to present this data in intelligible and extensible forms.

The materials and components are required to be interchangeable and the system must be able to collect data regarding their stability over time and temperature sensitivity. To this end, we have been supplied with and directed to use the Domino Ring Sampler IV evaluation board, hereafter abbreviated to DRS4, for data sampling. We have also been supplied with a silicon photomultiplier kit and scintillator. These three components make up a gamma detector.

For this project, the ability for others to be able to replicate our work and reproduce our results is emphasized. Thus, particular care during the design process is expected,

and exhaustive documentation of our process and results will need to be presented.

To summarize, the system must:

- Detect gamma radiation
- Record relevant data
- Use the DRS4 for data gathering
- Function with different scintillators and photomultipliers

1.3 Scintillation Gamma Detectors

Scintillation Gamma Detectors are systems able to detect gamma radiation. This is accomplished through the process of converting the gamma radiation to light particles. The light is then detected via a suitably sensitive light sensor, which, in turn, supplies a small voltage to a device capable of processing that signal, such as an oscilloscope. An overview of such a system is shown in Figure 1.1.

In this section, we will elaborate on these component's functions, as well as relate pertinent details about the the specific components used in this project.

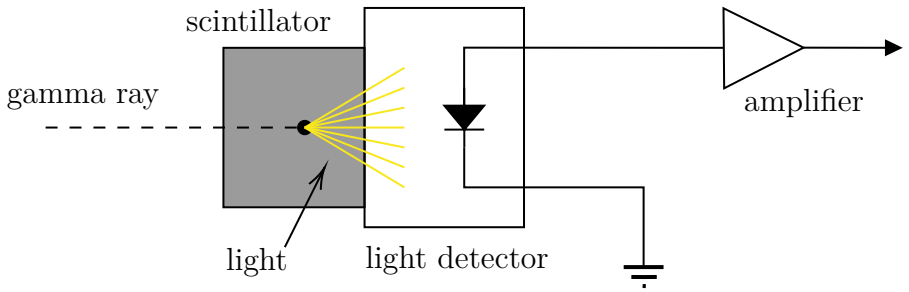


Figure 1.1: Sketch demonstrating the principle of gamma ray scintillation detection

1.3.1 Scintillation

Scintillation in this case means the flashes of visible or near-visible light that some materials emit when struck by certain energetic particles, such as gamma rays. Various materials that emit ionizing radiation can be utilized for the purpose of delivering a steady flow of gamma particles. The gamma radiation, which is on the high-energy end of the electromagnetic spectrum, can in turn be transformed into electromagnetic waves with a longer wavelength (and thereby lower energy) [3, 6].

Materials that emit light upon absorption of gamma rays are known as scintillators. Scintillators can generally be separated in organic and inorganic materials. Organic scintillators are aromatic hydrocarbon compounds, such as Naphthalene ($C_{10}H_8$), Antracene ($C_{14}H_{10}$) and Stilbene ($C_{14}H_{12}$). Inorganic scintillators range from crystals of salts, such as sodium iodide (NaI), caesium iodide (CsI) and barium fluoride (BaF_2), to plastic ma-

Figure 1.2: A scintillator - the LYSO crystal provided us for this project

terials such as polyvinyl toluene (PVT), where the latter is doped with a small amount of an organic scintillator [3].

For this project, we have been supplied with a Lutetium–yttrium oxyorthosilicate (LYSO) crystal by HVL. This is an inorganic scintillator developed in an attempt to combine the desirable characteristics of other compounds [12]. LYSO provides high light output and an excellent energy resolution.

1.3.2 Silicon Photomultipliers

Figure 1.3: The Ketek Silicon photomultiplier evaluation kit. © Ketek

The fundamental application of a Silicon Photomultiplier, hereafter referred to as SiPM, is to detect light. The SiPM's usefulness lies in its ability to detect electromagnetic radiation even at low levels and with high precision. The lower ranges at which a SiPM can detect light was previously only achieved through the use of photomultiplier tubes, which is an older technology with its own limitations [25]. It can directly detect electromagnetic radiation ranging from near ultraviolet (UV) to near infrared (IR), corresponding well with the range of the electromagnetic spectrum known as visible light

[2].

The SiPM can be considered a successor of the Geiger-Mode avalanche photo-diode (GM-APD). Simply put, the GM-APD utilizes photoelectric effect to detect photons. SiPMs are constructed out of several GM-APDs connected in parallel. This application enables counting of photons, as several photons can be detected simultaneously. This in turn leads to a higher level of precision and reliability [2].

During the course of this project, we will be working with a SiPM module supplied to us by HVL. It is part of a silicon photomultiplier evaluation kit by the company Ketek [24]. The kit includes a socket module for the SiPM to be inserted into, as well as a bias source and a preamplifer. These are there to ensure a clean and readable signal from the sensor. The whole of the kit will be incorporated into the project.

1.3.3 The Domino Ring Sampler IV

The Domino Ring Sampler 4 (DRS4) is an Integrated Circuit (IC), colloquially referred to as a "chip", developed by Dr. Stefan Ritt at the Paul Scherrer Institute in Switzerland. It was developed for the purpose of taking extremely precise measurements of time for experiments Dr. Ritt was conducting [11]. For the purpose of allowing other interested parties to easily evaluate the chip's capabilities, the DRS4 Evaluation Board, hereafter abbreviated to DRS4 rather than just the chip, was created. It comes with a series of features, but for the purposes of this project it might be thought of as a small, programmable and precise oscilloscope.

An oscilloscope is an electronic instrument that samples voltages and reads them out as waveforms. Oscilloscopes made for laboratory work are usually quite large compared to the DRS4 and their source code is usually not open for modification [15]. The three pictures in Figure 1.4 serve as a visual aid to underline the difference in dimensions.

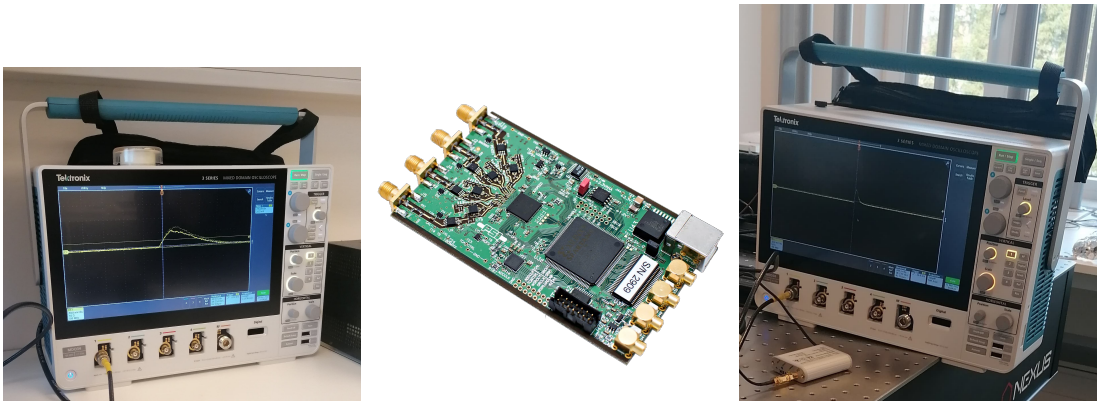


Figure 1.4: A Tektronix oscilloscope, a DRS4 and both of them together.

The DRS4 is also supplied with open source software (OSS) [26]. Open source signifies that the software's source code is made available and is released under a license that permits users to change and build upon said code [14]. The software supplied are

a graphical user interface (GUI) oscilloscope application, called DRS Oscilloscope, and a command line interface application, DRS command line interface. Both of these applications are "interactive" in that the user is required to first run the programs and thereafter supply a set of commands.

The applications can be run on Windows systems, GNU/Linux systems and Mac systems. This is due to a cross-platform library, wxWidgets, used by the library that handles most of the graphical interface disparities between systems [27].

For this project, we will familiarize ourselves with this software, its capabilities and limitations, and attempt to build upon it to meet the project requirements. From the outset there are three areas that require improvement: the need for interactivity in an automated process, the inability to acquire data over a given time and the inability of combining new sensor data, such as temperature, in data acquisition.

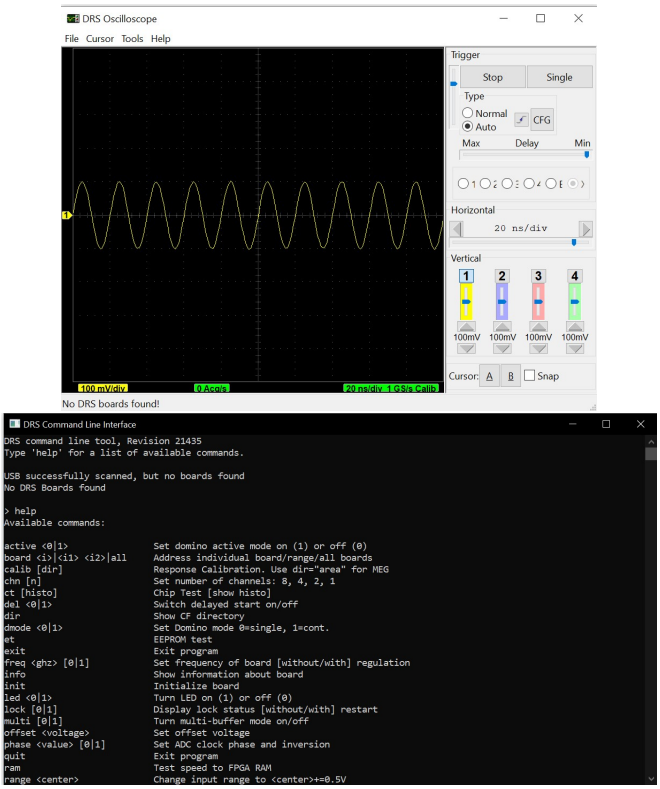


Figure 1.5: Screenshots of the DRS4's oscilloscope and CLI applications

2 Methodology

This chapter details the design choices that are the sum of our finalized product. We have chosen to separate this into two parts. One part pertaining to the physical system and one detailing that which concerns the digital inner workings — hardware and software.

A general overview of the elements that comprise the system can be found in Figure 2.1. Here the different components are separated and their individual functions, as well as the flow of data, are shown. The project concerns itself with all the steps depicted here. Proper housing for the radioactive source, scintillator and SiPM, safety measures for controlling power input to the SiPM, data gathering from the DRS4, control of the DRS4, and data analysis through the computer.

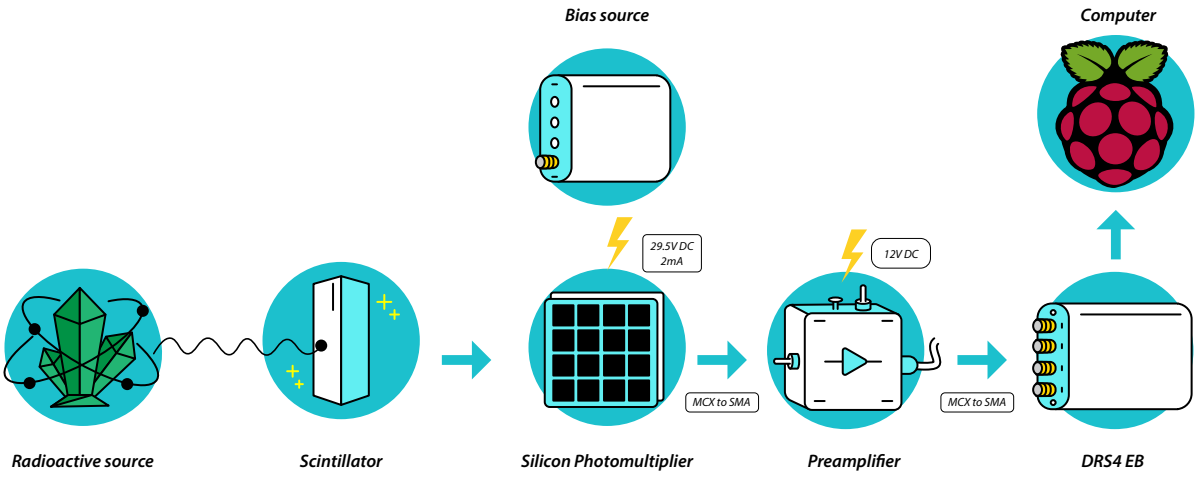


Figure 2.1: A general overview of the components the system is made up of and how they relate to each other in terms of their input/output flow

2.1 Hardware

As per Figure 2.1, this project concerns itself with several hardware components. Certainly there is hardware to read out and to compute data, but the project also concerns itself with other types of physical components. For the project to be accessible, and for

the SiPM modules and scintillators to function, there are several constraints that need to be met.

In this section we will detail the design choices we have made that comprise the physical components of this project.

2.1.1 The computer

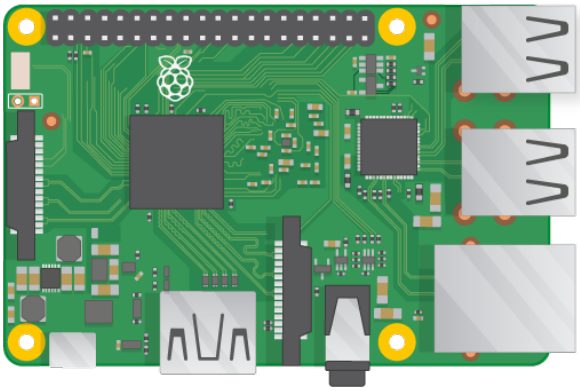


Figure 2.2: The Raspberry Pi 3 single-board computer. © Raspberry Pi Foundation

The necessity of a computer for this project is due to the data gathering and processing that is required. As we have seen from previous projects, the processing power to quickly sample and write data to file can not be met with simpler electronics, such as micro controller units (MCUs) like the popular Arduino [10]. Instead, we require a device capable of interpreting a wider range of programming and interfacing with various devices at high speeds. One such crucial device is the DRS4, which requires a USB connection with appropriate drivers and libraries.

To meet these requirements, we have chosen to base our project on the single-board computer, Raspberry Pi. This small computer has become popular due to its ease-of-use, small form factor, computing power and useful connectors, and has achieved this at a relatively low price point [18]. During the decade since its release, the Raspberry Pi has seen widespread adoption and, crucially, has shown itself capable of reliably operating over long stretches of time in industrial applications [19].

Before choosing the Raspberry Pi for this project, we looked at how it might impact the projects usability and longevity. In terms of usability, the computer's small size allows it to be used in a laboratory without taking up too much space - the Raspberry Pi is only 85.6mm × 56.5mm [16]. Adding to this is the fact that it can easily be controlled without monitor, keyboard or mouse — A so-called "headless setup". Instead, the Raspberry Pi can be securely controlled through wired (ethernet) and wireless (WiFi) network connection, as well as through serial console (UART) via the Secure Shell (SSH) protocol. This allows for flexibility and easy control of the device regardless of what security concerns any company's or institution's IT department might have regarding adding new devices to their networks.

In terms of longevity, the Raspberry Pi is not only capable of running the GNU/Linux operating system, but is, in fact, closely integrated with it. GNU/Linux is one of the most widely used operating systems and is found on smartphones, personal computers, servers and embedded systems. It is free, open source, extensible and reliable. The Raspberry Pi's officially supported operating system, Raspberry Pi OS, is a Debian-based distribution [20]. Debian is a GNU/Linux distribution commonly used in enterprise environments as it is known for it's long-term support, stability and security [21].

The Raspberry Pi is also notable for its available and easy to use GPIO (General-purpose input/output) pinout [17]. The GPIO pinout enables users to work with low-level electronics in a way which you would not find in a typical computer. It is through the GPIO pins one is able to establish a serial connection, and for this project we also use the GPIO for reading temperature data and supplying voltage to a relay switch.

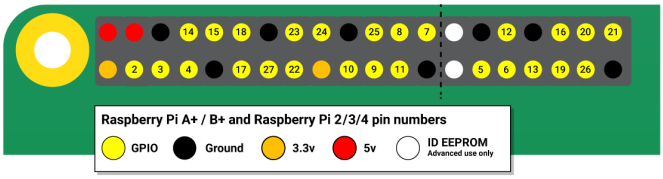


Figure 2.3: The Raspberry Pi's GPIO pinout scheme. © Raspberry Pi Foundation

Finally, the Raspberry Pi has exchangeable storage capabilities. Its primary storage is a mircoSD flash card, which can range from a few to several hundreds gigabytes of space. It is also able to support further storage space through external USB hard drives. Regardless of what types or amounts of data one wishes to acquire or for how long they are to be amassed, there are good storage options with the Raspberry Pi.

2.1.2 The PT1000 and MAX31865 breakout board

As per the project requirements, it is necessary to acquire relevant data to ascertain the functioning of the components in use. In particular, we are looking at the noise induced due to the SiPM's thermal sensitivity. Thus, the SiPM modules are meant to be tested at varying temperatures over time.

There are numerous ways of measuring temperature, but the PT100 and PT1000 sensors are commonly preferred for industrial applications [22]. The PT100 and PT1000 sensors are a type of resistance temperature detector (RTD). They are known for their high accuracy and repeatability, and the PT1000 provides even more accurate results for small changes in temperature. Still, it should be noted that there is a delay in response time that needs to be accounted for before choosing this type of sensor [23]. For the purposes of this project we consider this delay to be negligible. It would, however, be prudent to have an understanding of the temperature data being provided for this project. That is to say, we will only be measuring the ambient temperature inside of the

enclosure built for this project, and there is a delay of up to 60 seconds associated with larger shifts in temperature due to the construction of the PT1000. As per Figure 2.4, the PT1000 is housed in a protective and insulating casing. The delay is due to changes in ambient temperature needing to propagate through to the casing's core before being registered.

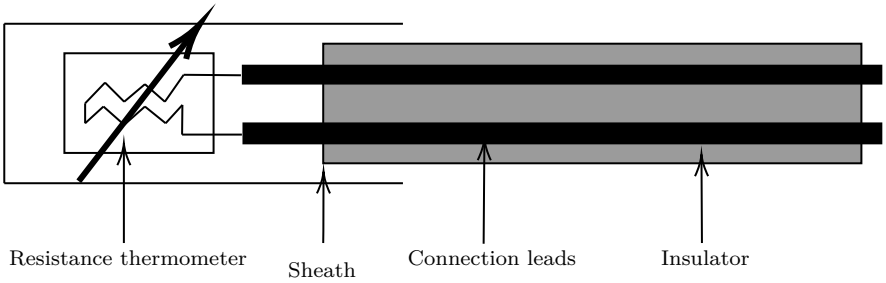


Figure 2.4: Overview depicting the inner construction of a RTD sensor, such as the PT1000.

In addition, when using a PT1000 sensor, it is not simply a matter of connecting the sensor to a computer and reading out an analog value. A circuit is required to allow us to measure the PT1000's change in resistance and convert it from analog to digital output. These are common and can readily be made, but for the purposes of ease of use and easy replication, we have decided to use a ready-made solution. The MAX31865 breakout board by Adafruit provides easy temperature output and requires little previous knowledge of electronics [1]. It is a flexible solution in that it is agnostic in regards to what type of wiring scheme one's particular PT1000 employ. The MAX31865 also comes with an analog-to-digital converter (ADC), which allows us to access the PT1000's analog output from the Raspberry Pi. The Raspberry Pi does not itself have any analog GPIO pins. Adafruit also provide a library for configuring the PT1000 and reading the temperature data [4].

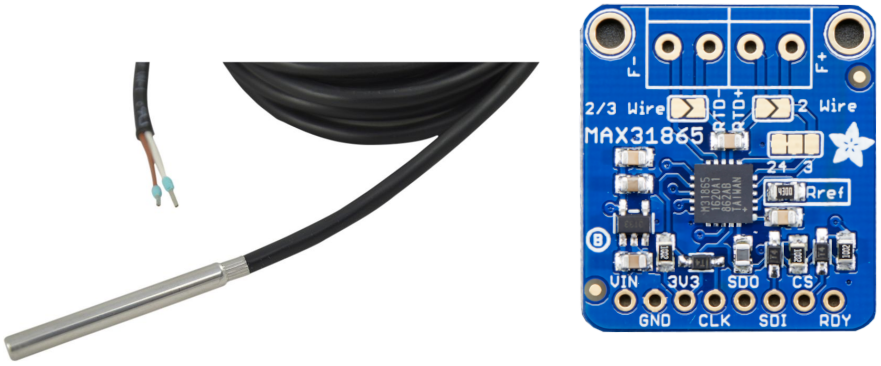


Figure 2.5: A PT1000 sensor and a MAX31865 breakout board. © Adafruit

2.1.3 Housing sensitive components

Early on we were made aware of the care with which the equipment needed to be handled. A previous Bachelor's project had been unfortunate enough to irreparably damage their SiPM module by supplying it with too much current [10]. There is also the concern that data acquisition might not only be affected by light contamination, but that continuous light could itself damage the SiPM — de-calibrating the device, lowering its sensitivity and affecting its electrical features [13].

Committed to learning from past mistakes, we looked closely at the previous group's enclosure and considered in which ways it could be improved upon. Their enclosure solution was a plastic box, shown in Figure 2.6, that was large enough to fit much of the system, and not only the scintillation gamma detector components themselves. The bias source and preamplifier were also placed inside of the enclosure. We surmised this was solely due to short length of the cables running from the SiPM module. This resulted in one having to pull the plug from the wall socket to power off the bias Source before lifting the lid, or risk exposing the SiPM to continuous light while powered.

In addition, there was the matter of the box's light-tightness. Holes had been drilled into the side of the box to provide the cables access to the outside. These had electrical tape placed over them to hinder light exposure.

Finally, the SiPM module itself was placed facing up on a cardboard box with cable holes in the bottom. The scintillator would be balanced on top of the SiPM and the radioactive source would presumably be positioned close by. The lack of appropriate holders for the components, or even defined positions, introduces the possibility of unwanted variance between experiments. Also, even a slight bump to the box might make the scintillator fall off the SiPM, which would halt an experiment in place. There should be no air gap between the SiPM and scintillator.

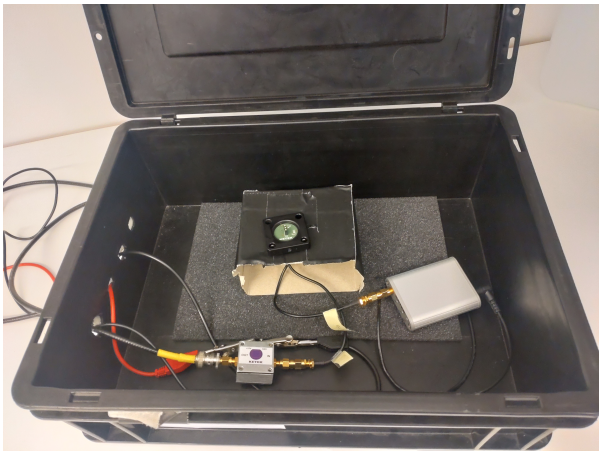


Figure 2.6: The enclosure utilized by the previous Bachelor's group.

Taking all of this into account, it became apparent that it would be beneficial to fashion a new enclosure. After all, this Bachelor's hinges upon very sensitive equipment's ability to produce reliable and repeatable results. To that end we believe there are two

key factors that must steer the design process for this project to be viable. First, it is important that we establish safety measures to avoid accidental damage to the equipment. Secondly, we must limit the chances of any experiment's results being altered or spoiled due to environmental variables.

Towards the goal of establishing safety measures for protecting the SiPM, we wanted power to be supplied to the bias source only when certain conditions were met. To accomplish this we decided to explore two avenues. One idea was to integrate a micro switch into the enclosure that would act as a "lid sensor". This sensor would signal whether the lid on the enclosure was present or not, with the intent of disallowing the supply of voltage to the SiPM whenever the lid was removed or inadequately positioned. The other idea was to use a phototransistor inside of the enclosure. This would allow our digital system to measure the enclosure's internal illumination. If the phototransistor was able to measure any light inside the box, it would signify that there was more light present than the SiPM was meant to absorb and so we would cut power.

As we wanted to make sure that a powered SiPM would never be exposed to light, we opted for an analog solution where the power supply was cut off whenever the enclosure was open. This would best be accomplished with a relay to control the supply voltage to the bias source. The idea is simple: whenever the lid is off, the bias source should be off as well. We would simply intersect one of the leads from the cable supplying the bias source, and connect either end to the connectors of the relay's "Normally Open" switch. In addition to the relay itself, we would design a simple circuit containing a flyback diode in series with a resistor. The purpose of the flyback diode is to protect the connected Raspberry Pi from over-voltage spikes when switching off the relay. The circuit is shown in Figure 2.7.

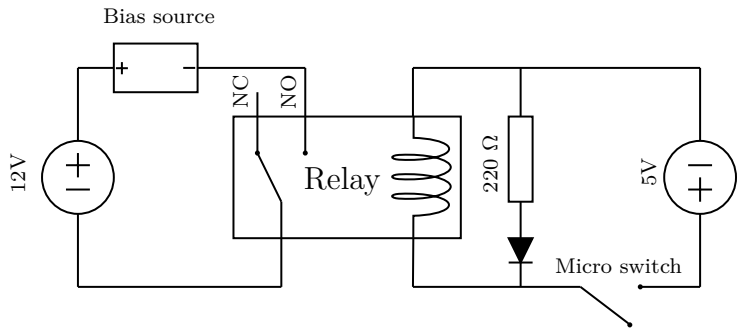


Figure 2.7: Circuit for operating the Bias Source via a relay.

The physical enclosure itself would have to satisfy the second key factor: providing an environment that supports repeatability for experiments. In addition to slots and holders designed for the PT1000 RTD, phototransistor and micro switch, a modular system with holders and standardised placement of components would need to be created. A way for cables to enter the box without causing light pollution would also be required.

We moved through several designs until we eventually ended up with an enclosure we would iteratively improve on. Cables would enter into the box from behind the SiPM,

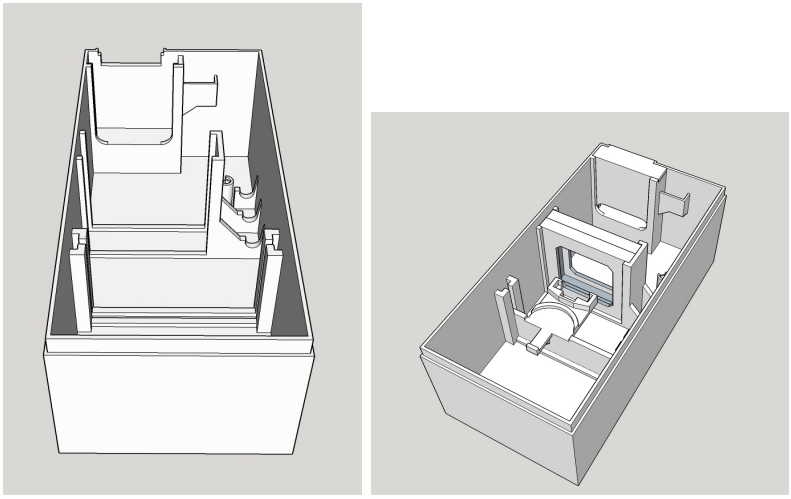


Figure 2.8: 3D models of the enclosure with and without ”loose parts” respectively

limiting light exposure in this way. An area with an insertable cover would be fashioned to further mitigate the risk of light pollution. In the same vein, the SiPM would be housed inside an insertable holder with fasteners on the sides of the box. The same would hold true for a scintillator and gamma source holder. Controlling all dimensions of the box, we would be able to ensure that there was always contact between the SiPM and scintillator.

In Figure 2.9 you will find our final enclosure designs: A model of a holder for the LYSO scintillator and a 1 coin-shaped gamma source, the cable cover (highlighted in green) allows cables to pass through while blocking light, the LYSO scintillator's contact with the SiPM is facilitated by their holders.

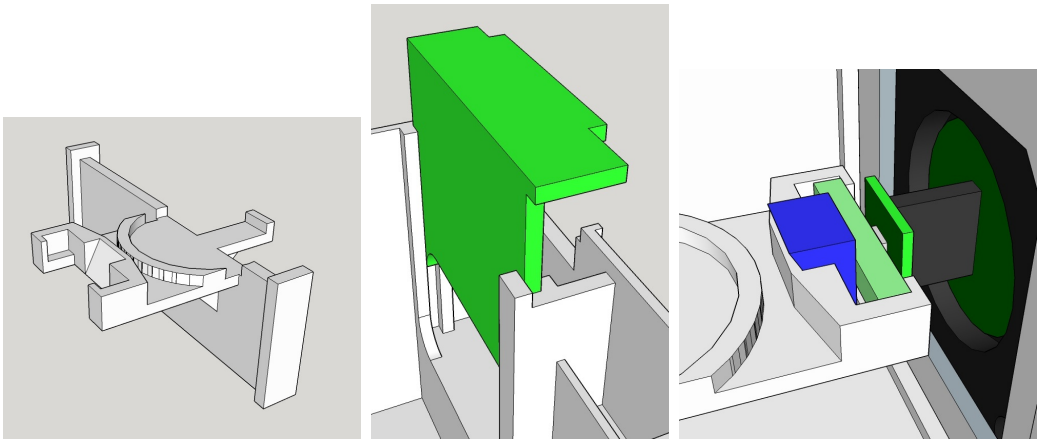


Figure 2.9: Scintillator and gamma source holder, the cable cover, and scintillator in contact with SiPM

2.2 Software

Software is the collection of programs used by a computer to produce a given result. When devising a system such as the one we have created for this project, there are many design choices that need to be made regarding the software.

There are many types of software categories, such as System Software, Programming Software, Driver Software and Application Software. These categories can be hierarchical in the sense that a opting software in one category will probably result in constraints in regards to what is available in other categories. There are external constraints that need to be met. For this project, we have endeavoured to build this system while keeping in mind stability, longevity and reproducibility — reproducibility here referring to the ease of setting up the system by oneself. We have also looked at compatibility with the software required to use the DRS4.

In this section, we will detail the design choices we have made that comprise the software components of this project.

2.2.1 Operating system

When deciding on an operating system (OS), we looked to balance several factors. We had to ask ourselves what type of system we envisioned, what type of hardware would be available to us and in what way our choice would affect the usability, stability and reproducibility of our system. We have built this system with the GNU/Linux operating system in mind. There were mainly two considerations in choosing this operating system — the computer and the DRS4.

We have previously accounted for our choice of computer, the Raspberry Pi. This computer is capable of running numerous operating systems (OS), such as Windows 10 IoT Core and different flavours of GNU/Linux. There are some differences between these systems, chief amongst them being that GNU/Linux is flexible and performant even on systems with fewer resources. Windows has traditionally been held to be less so due to the Desktop Environment (DE) that is a key feature of that system. Windows, however, is more widely used at all levels of computer literacy.

Building upon the foundation of the DRS4 was a requirement for this project. The DRS4's application software is able to run on Windows systems, GNU/Linux and OSX [8]. However, we encountered several obstacles when attempting to run the DRS4 software. In Windows, we found that the instructions for installation of driver software were outdated. While we were able to run the application software, the software and the device were never able to speak to each other because of these driver issues. Several attempts were made in mitigating this without result, and we also looked for help in the DRS4's forums. There we found that others had encountered the same problem, but that the authors of the software did not have a solution [7].

When attempting to install the software on a GNU/Linux system, we discovered that the installation instructions were similarly lacking. When installing the DRS4's software in GNU/Linux, compilation of the source code is required. As part of this

process, a specific version of an outside library for the graphical interface, wxWidgets 2.8.12, is required [9]. That version is 11 years old, and, probably, so are the installation instructions. We did eventually manage to successfully compile and make use of the DRS4 in GNU/Linux, and we were confident in our ability to build our own software on that foundation. However, the problems with the installation path prompted us to make certain design choices with regards to reproducibility.

2.2.2 Evaluating the DRS4 software

As mentioned previously, at the outset of this project there were three immediately apparent areas of concern.

First, the application software supplied with the DRS4 was built for interactivity. That is to say, the software must first be run and then given a series of commands to perform its tasks. In other words, its construction did not lend itself to automatic data acquisition.

Secondly, the software is unable to acquire data over a specified amount of time. Data acquisition is accomplished by defining a number of "events" that one wishes to capture. The events here signify a voltage detected above some set trigger threshold. While the expected amount of triggers over time might be somewhat predictable, this project seeks to map out the conditions in which SiPM modules start to provide inaccurate and unpredictable results. Thus, having to specify a number of events, rather than a duration for data acquisition is insufficient.

And finally, the application software was not built to incorporate other sensors and their data into the acquisition process. As noted before, while a SiPM is sensitive to photoelectrons, it is no less sensitive to thermally generated electrons [12]. We are interested in the degree to which temperature and prolonged use affect the noise-to-signal ratio of SiPM modules and must therefore include some measurement pertaining to the temperature coinciding with a trigger event.

Upon further inspection of the application software we discovered additional concerns for our consideration. As briefly mentioned in the previous section, the installation of the application software in GNU/Linux requires an old and unsupported version of the wxWidgets library. While we did find an installation path, this obstacle suggested to us that it might be imprudent to build further on that shaky foundation. In addition, we had considered that, for the purposes of automation, a graphical user interface would not be required. As we had decided to use GNU/Linux as the operating system, neither did we require the cross-platform capability the library afforded us. Therefore we surmised that the wxWidgts library introduced too much complexity to the project. We therefore decided that we would not try to extend the functionality of the existing application software. Instead, we would write our own program, only reusing the DRS4 library and its functions for communication with the DRS4.

Since the DRS4 library is written in C/C++, we chose C++ as the language we would write our own code in. For creating graphs we were asked to use Python, which is a popular language in academic circles due to its easy readability and powerful libraries

related to data analysis, specifically the libraries pandas and seaborn.

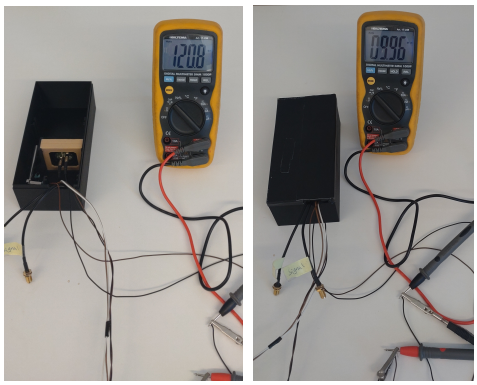
3 Results

In this chapter we present the results of our work during this project. As in the previous chapter, we present hardware and software separately.

3.1 The final enclosure

When building the enclosure, we took special care that it would ensure the repeatability of experiments, and that it would protect the SiPM from damage.

Due to the sensitivity of the SiPM, we needed to verify that our enclosure was not prone to light contamination. As mentioned previously, this pertains both to the repeatability of experiments and the integrity of the SiPM. To discover the degree of light-tightness our enclosure provided, we first used a flashlight inside of the enclosure in an attempt to visually verify any bleed-through. This simple test showed that there was no visible light passing through the enclosure's walls given a wall thickness of at least 2.25mm while using non-translucent black filament.



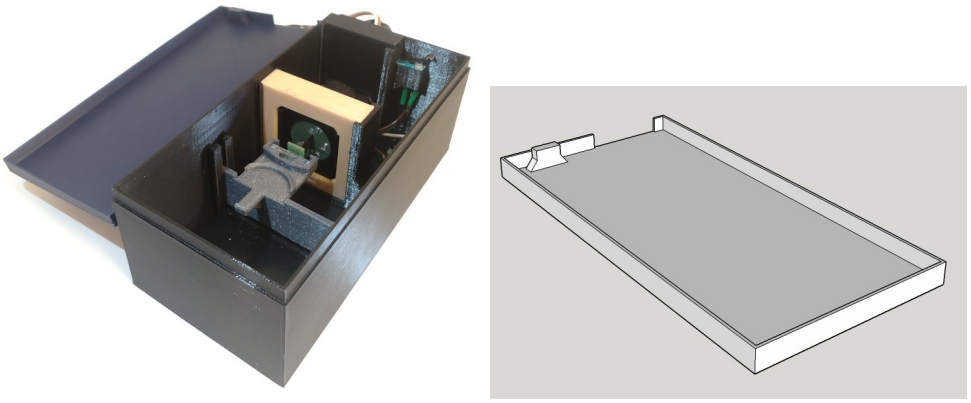
(a) Open enclosure, phototransistor in direct light (121 kΩ).

(b) Closed enclosure, no light contamination (9.96 MΩ).

Figure 3.1: Verification of light-tightness using phototransistor.

To further establish the degree of light-tightness, we measured the light inside the enclosure at different levels of light obstruction; without the lid, with the lid, and with the lid and cable cover. To this end we employed a phototransistor. With this method, as per Figure 3.1, we were able to show that light-tightness was achieved when both the lid and cable cover present.

Furthermore, the enclosure protects the SiPM from taking any damage because of light exposure by utilising a simple switch. The switch tells a relay whether the SiPM is to be supplied with power. As shown in Figure 3.2, the switch is depressed when the lid is placed upon the box. The lid is fashioned in such a way that it can only be placed in one orientation, ensuring that the lid is always correctly placed above the switch. The analog nature of this arrangement means a near-instantaneous power down of the SiPM, if the lid were to be moved askew or removed.



(a) Micro switch seen in back (b) The lid with extrusion for switch

Figure 3.2: Enclosure design ensuring SiPM is never exposed to light when powered

Then there is the matter of providing an environment for the other components that is conducive to good measurements. The PT1000 RTD snaps into place inside the box at a designated location. The SiPM is fastened inside a module which slides into place at an allotted position. This is directly adjacent to where a holder for the scintillator and gamma source is placed. The gamma source snaps into place and is held securely. Through experiments we have run, we have gotten results that are consistent with what one would expect from a design that ensures that the scintillator and SiPM are securely touching throughout the duration of stationary experiments.

To conclude, we have now been able to run the planned experiments using this enclosure. The improvements from the previous design allowed us to fit it inside a climate chamber, avoiding having the preamplifier and bias source there as well. More on the results of the experiment will follow in the next section.

3.2 Running the software

```
Usage: drs4client [OPTION]
Command-line application for timed data acquisition with DRS4 EB.

-f, --sampling-frequency Set sampling frequency. Default: 5 [GHz]
-i, --input-range        input range, ex. input 0 means -0.5 to 0.5. Default: 0
-l, --trigger-level      Set trigger level in volts. Default: 0.02 [V]
-d, --trigger-delay      Set trigger delay in ns. Default: 250 [ns]
-n, --numbered           Run for given number of events. Default: 0
-t, --timed              Run for given amount of seconds. Default: 0
-w, --ignore-wave        Do not save waveforms, only time and temp. Default: False
-h, --help               Shows this help message

        Examples:
        drs4client -t259200      Records events for 72 hours.
        drs4client -n50 -w       Records 50 events without waveform data.
```

Figure 3.3: The application software drs4client's help screen

The software is run and initiated with a set of parameters defining its operation. This is a common way of running command line applications which lends itself to automating tasks.

As part of the project we wanted to run two experiments. First we recorded timestamped events and coinciding temperature measurements — excluding waveform data — in room temperature over a period of 72 hours with a set of parameters (as shown below.) Then we repeated the same experiment, only now utilising a climate chamber to cycle the temperature at given intervals. To initiate the first experiment, we had to input the following command:

```
sudo ./drs4client --timed 259200 --ignore-wave --trigger-level 0.015
```

When later repeating the experiment under other conditions, that exact command is once again used. The program would then output the running parameters and indicate its progress. The output after 72 hours would then display:

```
Found DRS4 evaluation board, serial #2964, firmware revision 30000
Sampling frequency set to 5 GHz, input range 0 v, trigger level 0.015 v, trigger delay 250
    ns.
Time-based capture set to 259200 seconds. Capturing waveforms turned off.
Starting timed capture...
Running for 259200 seconds
Total elapsed time: 259200s
```

In the background the software produces files with event data. As mentioned, it may include or exclude waveform data. The benefit of excluding waveform data is that it requires less space.

The Comma-separated values (CSV) format is a common and widely used data exchange format [5]. The raw data from the software's data acquisition is stored as a text file, in a CSV-like format. The text file produced by the software is essentially lines of event data, including the event index, timestamp and temperature. Every line is preceded by a CSV structure containing the waveform data, if that information is stored. These waveforms can then easily be extracted and used for further analysis.

```
...
Event #344322, Temp: 39.838, Time: 86384515.0 ms
Event #344323, Temp: 39.838, Time: 86384769.0 ms
Event #344324, Temp: 39.872, Time: 86385006.0 ms
...
```

Figure 3.4: Example of file excluding waveform data

For the purposes of this project, however, when further processing the recorded data we strip out any waveform data as it is not used. Instead, the file is parsed, solely on the basis of the data found in Figure 3.4, into a pure CSV file. The time in milliseconds is converted into a timedelta format and the data is then further handled, resampling every 30 minutes of data and finding the mean temperature, as well as other possible calculations, such as finding the standard deviation and the number of recorded events every 30 minutes.

```
ms,temp_mean,temp_std,temp_count
0 days 00:00:00.002000,5.643623544543732,0.13925902931290338,7386
0 days 00:30:00.002000,5.2772295290102385,0.06979964283333595,7325
0 days 01:00:00.002000,5.137910683468018,0.025488690660789844,7301
0 days 01:30:00.002000,5.087723410682411,0.015091708748774501,7283
...
```

Figure 3.5: Example of resulting CSV file

This is then further used to plot and display three graphs, each displaying the relationship between temperature and time, the number of events every 30 minutes and time, and the temperature and the number of events every 30 minutes.

In the experiment conducted at room temperature, the equipment was left to run in a lab where the temperature only fluctuated slightly. During the second experiment, we set the climate chamber to continuously cycle the temperature throughout the experiment's 72 hours. From the starting point of 5°C, we incremented the temperature by 5°C every second hour. At 50°C we would then decrement the temperature in the same way, until again reaching 5°C.

By visual examination we find that the results of our two experiments indicate a relation between shifts in temperature and the performance of the SiPM. Figure 3.6 contains three graphs which represent the data acquired while running the experiment at room temperature. Here we see a lack of clear trends in relation to the change of temperature.

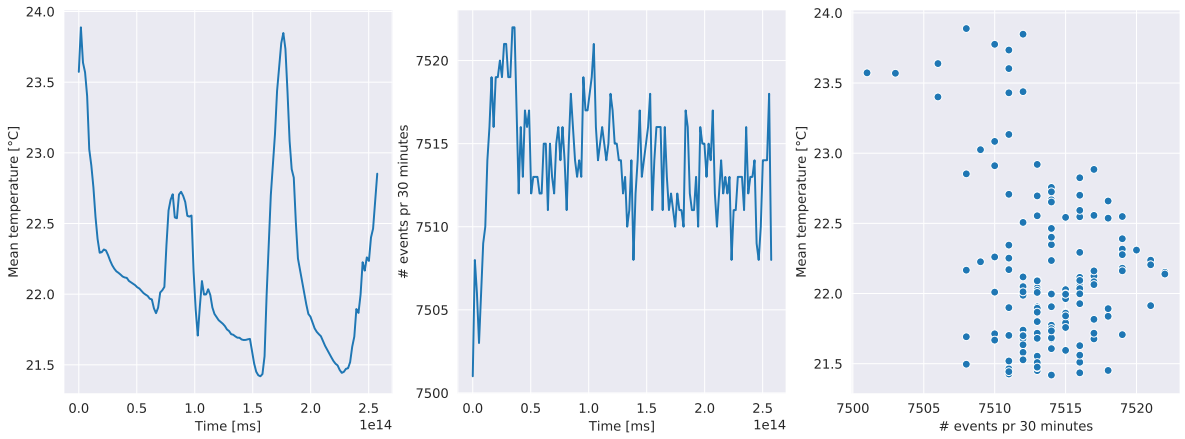


Figure 3.6: Three graphs representing data recorded at room temperature

This is clearly contrasted with the data recorded in the climate chamber. As shown in Figure 3.7, we see a trend emerge, that the number of events fluctuates with the changes in temperature. This seems to be in accordance with the hypothesis that a thermal sensitivity should give rise to dark pulses and noise [12].

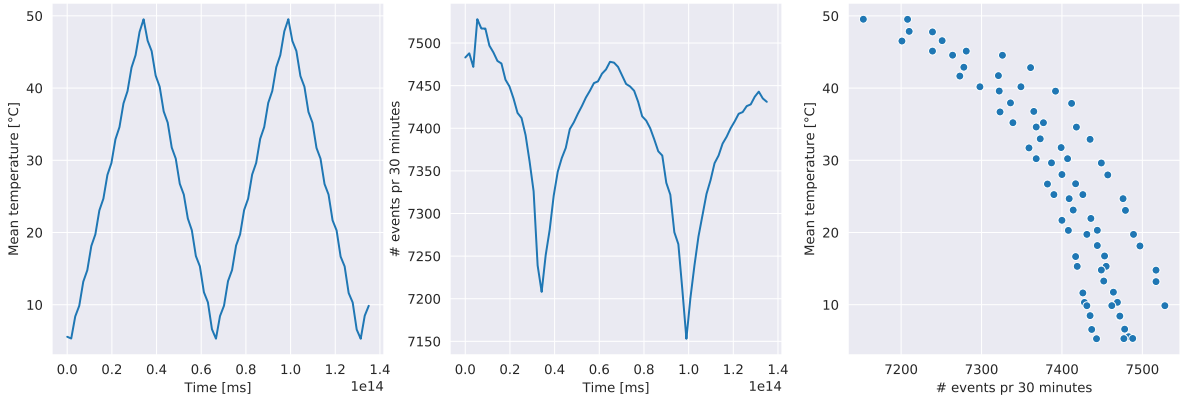


Figure 3.7: Three graphs representing data recorded in a climate chamber

4
Discussion

This project was set out to develop an automated data acquisition system based on the DRS4 capabilities. The system should be reproducible and extensible, and the results of experiments conducted through the system should be repeatable. Furthermore, the system should be able to conduct said experiments over long periods of time while gathering relevant data. To accomplish this we have had to engage with a wide variety of technologies and tie them together into one cohesive system. As shown, we have approached this from two sides; hardware and software.

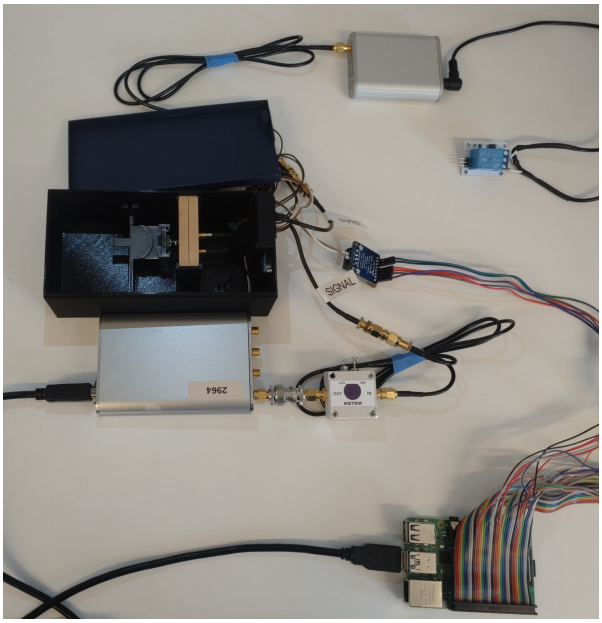


Figure 4.1: A final overview of the components the system is made up of.

Our solution to the project's requirements is an amalgamation of physical components and software. The software is open-source and runs on GNU/Linux. When we first started looking into the DRS4's software we discovered several problems in terms of drivers, libraries and code compilation. Three whole days were spent verifying that the code could be compiled, so that we had a foundation to build upon. Also, the amount of complex yet uncommented code in the DRS4 library meant that an inordinate amount

of time was devoted to grasping the programs flow. Therefore, with the intent of being more accessible to others, our software development process made a point of closely evaluating any drivers, libraries and code before including it in our software.

This is why we have kept the libraries used in our C++ code to a minimum. Only standard libraries and libraries required by the DRS4 library has been included. This makes it more robust over time and easier for others to dig into, changing small things or developing new features. Also, the GNU/Linux operating system is, of course, free and is not going anywhere. This same philosophy was taken into account when choosing the MAX31865 breakout board to read temperature data. It was chosen because it is easy to set up and use, lowering the bar for others to reproduce this system on their own.

As the enclosures 3D models are also available, this too is easily reproduced with a 3D printer, or extended with appropriate software. We feel that 3D printing provides an excellent opportunity for prototyping. However, as we learned, some issues may arise. Our models were designed in Google Sketchup, a free and intuitive software for creating 3D models. However, we found that these models were easily corrupted when exported to STL files, which is required for printing. We managed to resolve this by loading these files into Autodesk Fusion 360 and "repairing" them there through an automatic tool. Still, it was a unnecessarily time-consuming process and so we would recommend using other programs for 3D modeling in the future.

One of the most important aspect of our enclosure is the known geometrical relationship between components, as any change in the distance between scintillator and gamma source could have a significant effect on the measured results. This is important in order to ensure repeatability and reproducibility. The scintillator holder is designed to fit a 3x3x20 mm LYSO crystal, and allows the scintillator to be placed in contact with the SiPM. However, the enclosure is sufficiently dimensioned to allow utilization of larger scintillators and the holder design should be easy to adapt to this end. In the same spirit, the module holding the SiPM was implemented in order to facilitate the use of alternative photomultipliers that might be advantageous to run experiments on in the future. The module consists of two identical frames that the SiPM is "sandwiched" between. The module in turn fits into a slot in the enclosure. This makes it possible to use photomultipliers of both larger and smaller size than the SiPM used in this project.

Then there is also the matter of choosing a suitable filament for printing the enclosure. As previously mentioned, we recommend using a non-translucent black PLA filament to ensure light-tightness. Nevertheless, one iteration of the printing process ended up with an enclosure printed in white PLA, because of black or dark PLA was out of stock at the printing facility. We can clearly see from Figure 4.2 that light passes through the plastic. In other words, although off-brand filaments have served us well, the type of filament used does matter quite a bit. In addition to color, resilience towards heat emerged as a relevant property of the filament. PLA is considered to handle temperatures of up to 60°C before it softens, and that was enough for our experiments utilizing the climate chamber. However, it might be advantageous to consider more heat resilient materials to allow for data collection experiments at higher temperatures.

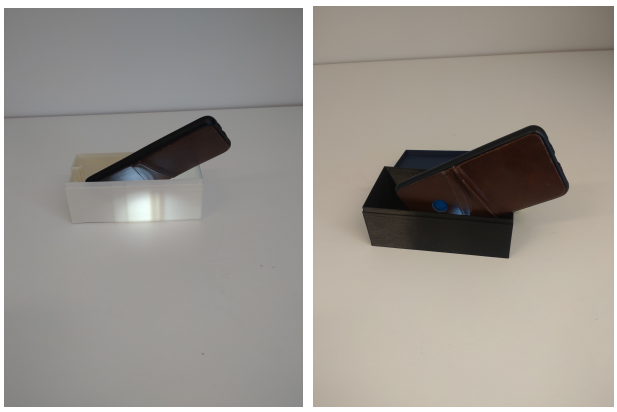


Figure 4.2: Light-tightness test of white and black PLA respectively.

Some of the enclosure's design was changed in the end. The phototransistor had originally been planned to be permanently integrated into the enclosure. However, we ended up not implementing it in the final solution. We deemed it an unnecessary addition to the enclosure as we found the micro switch and lid to sufficiently mitigate the risk of unwanted light exposure to the SiPM while powered. Still, as shown previously in Figure 3.1, the phototransistor proved itself useful for validating the enclosure's light-tightness.

Finally, we discovered that our self-made relay circuit might still be unsafe for the Raspberry Pi, despite our efforts to counter the risk of overvoltage with the flyback diode and resistor. Therefore we acquired a ready-made relay module designed for the Raspberry Pi to replace our self-made circuit. The Raspberry Pi supplies a constant voltage of 5V to both the "V+" and "Signal" pins on the module, so the only factor controlling the relay is the lid sensor. Initially we were interested in controlling the relay through our software on the Pi, but we found that it wouldn't provide any real benefit to do so. Operating the relay with the micro switch directly is the fastest possible way to disengage the relay, and thereby cut the supply voltage to the bias source. For this reason we opted to avoid solutions where the switch's signal goes into the Pi, which in turn controls the relay.

5 Conclusion

The study set out to develop a system capable of detecting gamma radiation and recording relevant data with the use of the DRS4. It should also be able to accommodate different scintillators and photomultipliers.

Now looking back at what we have produced, we see that our system is indeed capable of detecting gamma radiation through scintillation. Through our experimental testing at Roxar, we have also shown that we are able to acquire relevant data for further analysis. Furthermore, while we never had access to other components than the ones provided to us at the outset of this project, the enclosure was built to be modular. In other words, it is fashioned in such a way as to readily accommodate different scintillators and photomultipliers.

We also believe that we have created a system which is easily accessible and able to be developed further. Future projects should have little trouble building on this foundation to accomplish more advanced experiments and data analysis. For example, one might look at the SiPM's energy spectrum by extending the software's functionality, enabling the user to gradually adjust the trigger level over time.

Bibliography

[1] *Adafruit PT100 RTD Temperature Sensor Amplifier - MAX31865*. URL: https://www.adafruit.com/product/3328. (accessed: 24.05.2022).

[2] advansid.com. "Introduction to SiPMs". In: (2013). Last accessed 03.02.2022. URL: https://advansid.com/attachment/get/up_89_1411030571.pdf.

[3] Lucio Cerrito. *Radiation and Detectors: Introduction to the Physics of Radiation and Detection Devices*. Last accessed 03.02.2022. 2017. URL: https://link.springer.com/chapter/10.1007/978-3-319-53181-6_9.

[4] *CircuitPython Installation of MAX31865 Library*. URL: https://learn.adafruit.com/adafruit-max31865-rtd-pt100-amplifier/python-circuitpython. (accessed: 24.05.2022).

[5] *Comma-separated values*. URL: https://en.wikipedia.org/wiki/Comma-separated_values#Specification. (accessed: 27.05.2022).

[6] Veronica Danielsen. "Scintillasjon (fysikk)". In: (2020). Last accessed 03.02.2022. URL: https://snl.no/scintillasjon_-_fysikk.

[7] *DRS4 Discussion Forum - no board found*. URL: https://elog.psi.ch/elogs/DRS4+Forum/745. (accessed: 26.05.2022).

[8] *DRS4 Evaluation Board User's Manual*. URL: https://www.psi.ch/sites/default/files/2020-02/manual_rev51.pdf. (accessed: 25.05.2022).

[9] *DRS4 RPI Installation instructions*. URL: https://bitbucket.org/ritt/drs4eb/src/master/raspi.txt. (accessed: 25.05.2022).

[10] Athavan Sivakumaran Endre Standnes Jakob Svensholt. "Testing av Gamma Scintillasjonsdetektor. (Norwegian) [On the testing of gamma scintillation detection]". In: *HVL Open* (2019). DOI: https://hvlopen.brage.unit.no/hvlopen-xmlui/handle/11250/2602579.

[11] Laura Hennemann. "Measuring the simultaneity". In: (2015). Last accessed 25.05.2022. URL: https://www.psi.ch/en/media/our-research/measuring-the-simultaneity.

[12] Glenn F. Knoll. *Radiation Detection and Measurement*. Last accessed 26.05.2022. 2010. URL: https://www.wiley.com/en-sg/Radiation+Detection+and+Measurement,+4th+Edition-p-9780470131480.

[13] A. Nagai et al. "SiPM behaviour under continuous light". In: (2019). Last accessed 27.05.2022. URL: https://www.researchgate.net/publication/336208250_SiPM_behaviour_under_continuous_light.

[14] *Open-source software*. URL: https://en.wikipedia.org/wiki/Open-source_software. (accessed: 26.05.2022).

[15] *Oscilloscope*. URL: https://en.wikipedia.org/wiki/Oscilloscope. (accessed: 25.05.2022).

[16] *Raspberry Pi 4 Model B product brief*. URL: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf. (accessed: 24.05.2022).

[17] *Raspberry Pi GPIO 40 pin header*. URL: https://www.raspberrypi.com/documentation/computers/os.html#gpio-and-the-40-pin-header. (accessed: 24.05.2022).

[18] *Raspberry Pi in Industrial Automation*. URL: https://en.wikipedia.org/wiki/Raspberry_Pi. (accessed: 24.05.2022).

[19] *Raspberry Pi in Industrial Automation*. URL: https://en.wikipedia.org/wiki/Raspberry_Pi#Industrial_automation. (accessed: 24.05.2022).
[20] *Raspberry Pi OS Documentation*. URL: https://www.raspberrypi.com/documentation/computers/os.html. (accessed: 24.05.2022).
[21] *Reasons to use Debian*. URL: https://www.debian.org/intro/why_debian.en.html. (accessed: 24.05.2022).
[22] *Resistance thermometer*. URL: https://en.wikipedia.org/wiki/Resistance_thermometer. (accessed: 24.05.2022).
[23] *Resistance thermometer: Advantages and limitations*. URL: https://en.wikipedia.org/wiki/Resistance_thermometer#Advantages_and_limitations. (accessed: 24.05.2022).
[24] *Silicon photomultipliers evaluation kit*. URL: https://www.ketek.net/wp-content/uploads/2017/01/KETEK-PEVAL-KIT-MCX-QuickStartGuide.pdf. (accessed: 27.05.2022).
[25] *Silicon photomultipliers: theory and practice*. URL: https://www.youtube.com/watch?v=BH768QRfzFA. (accessed: 27.05.2022).
[26] *Software Download for the DRS4 Evaluation Board*. URL: https://www.psi.ch/en/drs/software-download. (accessed: 26.05.2022).
[27] *wxWidgets Cross-Platform GUI Library*. URL: https://www.wxwidgets.org/. (accessed: 26.05.2022).

A Appendix

The project's source code and 3D models can be found at https://github.com/tmavro/DRS4-client-HVL. Instructions for installation can be found there as well.

A.1 Installation

Compiling and running the software requires installation of libusb-1.0. It is also recommended that you use a screen multiplexer, such as tmux, to enable running drs4client in the background. On debian-based systems, such as Raspberry Pi OS, this can easily be done in terminal:

```
sudo apt update && sudo apt -y upgrade && sudo apt-get install libusb-1.0.0-dev tmux
```

To capture temperature data with the MAX31865 breakout board, you will need some Python libraries:

```
sudo pip3 install --upgrade setuptools adafruit-python-shell

wget
    https://raw.githubusercontent.com/adafruit/Raspberry-Pi-Installer-Scripts/master/raspi-b
    -O- | sed '124d' - > raspi-blinka.py

sudo python3 raspi-blinka.py

sudo pip3 install adafruit-circuitpython-max31865
```

From there, simply clone this repository and run make.

```
git clone https://github.com/tmavro/DRS4-client-HVL

cd DRS4-client-HVL

make
```

At this point your system will require a restart.

```
sudo reboot
```

The software should be run within tmux. It is advisable that you familiarize yourselves with the basics of tmux's operation. In addition, drs4client requires root privileges to access the DRS4 EB's USB connection and run.

```
tmux

sudo ./drs4client --help
```

A.2 Source code

Listing A.1: drs4client.cpp

```
1  /* Authors: Thomas B Mavropoulos and Sigve B Heggedal */
2
3  /*  ------
4       Libs
5      ------ */
6
7  //DRS-related
8  #include <unistd.h>
9  #include <ctype.h>
10 #include <sys/ioctl.h>
11 #include <errno.h>
12 #include "strlcpy.h"
13 #include "DRS.h"
14
15 #include <stdio.h>      /* fprint */
16 #include <string.h>
17 #include <stdlib.h>
18 #include <iostream>
19
20 #include <iterator>
21 #include <math.h>
22 #include <getopt.h>     /* getopt API */
23 #include <chrono>       /* Time-related */
24 #include <array>
25
26 /*  -------------------
27      Define directives
28     ------------------- */
29 #define DIR_SEPARATOR '/'
30 #define PROGRAM_NAME "drs4client"
31
32 /*  ------------------
33      Global variables
34     ------------------ */
35
```

```
36 // Argument options
37 int argFreq = 5;         // Sampling frequency
38 int argRange = 0;        // Input range, -0.5 - 0.5 [V]
39 int argTrigLvl = 0.02;   // Trigger level [V]
40 int argTrigDelay = 250; // Trigger delay [ns]
41 int argNumb = 0;         // Numbered run
42 int argTime = 0;         // Timed run
43 bool argIgnore = false; // Do not save waveforms
44 //char* argFile = "data.txt"; //TODO Filename handling
45
46 // Long argument options
47 static struct option const long_options[] =
48 {
49     {"sampling-frequency", optional_argument, NULL, 'f'},
50     {"input-range", optional_argument, NULL, 'i'},
51     {"trigger-level", optional_argument, NULL, 'l'},
52     {"trigger-delay", optional_argument, NULL, 'd'},
53     {"numbered", optional_argument, NULL, 'n'},
54     {"timed", optional_argument, NULL, 't'},
55     {"save-to", optional_argument, NULL, 's'},
56     {"ignore-wave", no_argument, NULL, 'w'},
57     {"help", no_argument, NULL, 'h'},
58     {NULL, 0, NULL, 0}
59 };
60
61 /* ---------
62     Methods
63    --------- */
64 void runBoardTime(DRSBoard* &b, std::chrono::seconds sekund, FILE* &f,
       int &c); // Runs board for given amount of time
65 void runNumbEvents(DRSBoard* &b, int events, FILE* &f,
       std::chrono::steady_clock::time_point starttime); // Runs board for
       given amount of events
66 int initScan(DRS* &drs, DRSBoard* &b); // Checks for board and
       initializes it
67 void showUsage(int status); // Shows help screen with options
68 void processArgs(int argc, char **argv); // Handles cli arguments
69 void printArgs(); // Prints the given cli arguments
70
71 using namespace std;
72
73 int main(int argc, char **argv){
74     processArgs(argc, argv); // Processing arguments from cli
75
76     // Initializing board
77     DRS *drs; //Boards
78     DRSBoard *b; //Board
79     drs = new DRS();
80     if (initScan(drs, b) == EXIT_FAILURE) return EXIT_FAILURE; //
           Initialize board, exits if no boards found.
81
82     printArgs();
```

```
83
84      // Run board for a given time or number of events
85      if (argTime > 0 && argNumb == 0) {
86      // Opens/creates file to save waveform
87      FILE* f;
88      f = fopen("data.txt", "w");
89      if (f == NULL) {
90          perror("Error writing to file");
91          delete drs;
92          return EXIT_FAILURE;
93      }
94          // Runs for amount of time in seconds
95          std::chrono::seconds sekunder(argTime); //TODO Implement more
                complex time handling, not just seconds
96          int countEvents = 1; // Counts number of events
97
98          cout << "Starting timed capture...\n";
99          auto start = std::chrono::steady_clock::now();
100         runBoardTime(b, sekunder, f, countEvents);
101         auto end = std::chrono::steady_clock::now();
102         std::chrono::duration<double> elapsed_seconds = end - start;
103         cout << "\nTotal elapsed time: " << elapsed_seconds.count() <<
                "s\n";
104
105         fclose(f);
106         delete drs;
107
108         //TODO Python plot magic here, probably?
109
110         return EXIT_SUCCESS;
111     }
112     else if (argNumb > 0 && argTime == 0){
113         FILE* f; // For storing waveforms
114         f = fopen("data.txt", "w");
115         if (f == NULL) {
116             perror("ERROR: Cannot open file \"data.txt\"");
117             delete drs;
118             return EXIT_FAILURE;
119         }
120
121         // Time counter
122         std::chrono::steady_clock::time_point starttime =
                std::chrono::steady_clock::now();
123         // repeat argNumb times
124         for (int j=0 ; j<argNumb; j++) {
125             runNumbEvents(b, j, f, starttime);
126         }
127
128         cout << endl;
129
130         fclose(f);
131         delete drs;
```

```
132
133         return EXIT_SUCCESS;
134     }
135     else {
136         cout << ("User must choose to either run based on an ammount of
                time or number of events. Exiting.\n");
137         delete drs;
138         return(EXIT_FAILURE);
139     }
140 }
141
142 // Returns temperature passed from python script
143 string measureTemp(){
144     array<char, 16> buffer;
145     string temp;
146     FILE* pipe = popen("python3 readtemp.py 2>&1", "r");
147     if (!pipe)
148     {
149         return "Temp error";
150     }
151     while (fgets(buffer.data(), 16, pipe) != NULL) {
152         temp += buffer.data();
153     }
154     pclose(pipe);
155
156     return temp;
157 }
158
159 // parses cli argument input to correct time format
160 void parseTimeArgumet() { //TODO Implement
161 }
162
163 void printArgs(){
164     printf("Sampling frequency set to %f GHz, input range %f V, trigger
            level %f V, trigger delay %f ns. \n",
165     argFreq, argRange, argTrigLvl, argTrigDelay);
166     if (argTime > 0) printf("Time-based capture set to %d seconds. ",
            argTime);
167     if (argNumb > 0) printf("Event-based capture set to %d events. ",
            argNumb);
168     if (argIgnore) {
169         printf("Capturing waveforms turned off.\n");
170     }
171     else {
172         printf("Capturing waveforms turned on.\n");
173     }
174 }
175
176 // Simply prints help screen to console
177 void showUsage(int status){
178     if (status != EXIT_SUCCESS) {
179         cout << "Show Usage failure: Nothing to see here.";
```

```
180         exit(status);
181     }
182     else {
183     printf(("Usage: %s [OPTION] \n"), PROGRAM_NAME); //... [FILE]...\n"),
            PROGRAM_NAME);
184     fputs(("Command-line application for timed data acqusition with DRS4
            EB.\n"), stdout);
185 
186     fputs(("\
187     \n\
188     -f, --sampling-frequency Set sampling frequency. Default: 5 [GHz]\n\
189     -i, --input-range        input range, ex. input 0 means -0.5 to 0.5.
            Default: 0\n\
190     -l, --trigger-level      Set trigger level in volts. Default: 0.02
            [V]\n\
191     -d, --trigger-delay      Set trigger delay in ns. Default: 250 [ns]
            \n\
192     -n, --numbered           Run for given number of events. Default: 0\n\
193     -t, --timed              Run for given amount of seconds. Default:
            0\n\
194     -w, --ignore-wave        Do not save waveforms, only time and temp.
            Default: False \n\
195     -h, --help               Shows this help message\n\
196     "), stdout);
197     printf(("\
198             \n\
199             Examples:\n\
200             %s -t259200      Records events for 72 hours.\n\
201             %s -n50 -w       Records 50 events without waveform data.\n\
202             "), PROGRAM_NAME, PROGRAM_NAME);
203     }
204     cout << endl;
205     exit(status);
206 }
207 
208 // Parse command line options
209 void processArgs(int argc, char **argv){
210 
211     int c;
212     while ((c = getopt_long (argc, argv, "f::i::l::d::n::t::wh",
            long_options, NULL)) != -1)
213     {
214         switch (c)
215         {
216         case 'f': //Set frequency
217             argFreq = atoi(optarg);
218             break;
219 
220         case 'i': //Input range
221             argRange = atoi(optarg);
222             break;
223 
```

```
224         case 'l': //Trigger level
225             argTrigLvl = atoi(optarg);
226             break;
227 
228         case 'd': //Trigger delay
229             argTrigDelay = atoi(optarg);
230             break;
231 
232         case 'n': //Numbered run
233             argNumb = atoi(optarg);
234             break;
235 
236         case 't': //Timed run
237             argTime = atoi(optarg);
238             break;
239 
240         /*case 's': //Save-to
241             printf("Option s has arg: %s\n", optarg ? optarg : "(none)");
242             //argFile = optarg; //TODO Fix this, maybe make a separate
                    method to handle properly
243             break;*/
244 
245         case 'w': //Ignore waveform
246             argIgnore = true;
247             break;
248 
249         default:
250             showUsage(EXIT_SUCCESS);
251         }
252     }
253 }
254 
255 // Board collects data for given amount of events
256 void runNumbEvents(DRSBoard* &b, int events, FILE* &f,
        std::chrono::steady_clock::time_point starttime){
257     float time_array[8][1024]; // Waveform data, time axis
258     float wave_array[8][1024]; // Waveform data, amplitude
259 
260     //start board (activate domino wave)
261     b->StartDomino();
262 
263     //Wait for trigger.
264     fflush(stdout);
265     while (b->IsBusy());
266 
267     //read all waveforms
268     b->TransferWaves(0, 8);
269 
270     //read time (X) array of first channel in ns
271     b->GetTime(0, 0, b->GetTriggerCell(0), time_array[0]);
272 
273     //decode waveform (Y) array of first channel in mV
```

```
274     b->GetWave(0, 0, wave_array[0]);
275 
276     double millisec =
            std::chrono::duration_cast<std::chrono::milliseconds>
277 (std::chrono::steady_clock::now() - starttime).count();
278 
279     //Save data
280     fprintf(f, "Event #%d, Temp:%.6s, Time: %.1f ms\n", events+1,
            measureTemp().c_str(), millisec);
281     if (!argIgnore) {
282         fprintf(f, "t[ns],u[mV]\n");
283         for (int i = 0; i < 1024; i++){
284             fprintf(f, "%.3f,%.1f\n", time_array[0][i], wave_array[0][i]);
285         }
286     }
287     //print some progress indication
288     printf("\rEvent #%d read successfully", events);
289 }
290 
291 // Board collects data for given amount of time
292 void runBoardTime(DRSBoard* &b, std::chrono::seconds sekund, FILE* &f,
        int &counter) {
293 
294     float time_array[8][1024]; // Waveform data, time axis
295     float wave_array[8][1024]; // Waveform data, amplitude
296 
297     std::chrono::steady_clock::time_point starttime =
            std::chrono::steady_clock::now();
298     std::chrono::time_point<std::chrono::system_clock> begin =
            std::chrono::system_clock::now();
299     std::chrono::time_point<std::chrono::system_clock> end =
            std::chrono::system_clock::now() + sekund; // Timed end-point
300 
301     while (std::chrono::system_clock::now() < end) { // While current
            time is still less than defined time end-point
302 
303         // Start board (activate domino wave)
304         b->StartDomino();
305 
306         // Wait for trigger. (This is where the small drift between given
                run time and actual run time is introduced.)
307         fflush(stdout);
308         while (b->IsBusy());
309 
310         // read all waveforms
311         b->TransferWaves(0, 8);
312 
313         // read time (X) array of first channel in ns
314         b->GetTime(0, 0, b->GetTriggerCell(0), time_array[0]);
315 
316         // decode waveform (Y) array of first channel in mV
317         b->GetWave(0, 0, wave_array[0]);
```

```
318
319         double millisec =
               std::chrono::duration_cast<std::chrono::milliseconds>
320 (std::chrono::steady_clock::now() - starttime).count();
321
322         //Save data
323         fprintf(f, "Event #%d, Temp: %.6s, Time: %.1f ms\n", counter,
               measureTemp().c_str(), millisec);
324         if (!argIgnore) {
325             fprintf(f, "t[ns],u[mV]\n");
326             for (int i = 0; i < 1024; i++){
327                 fprintf(f, "%.3f,%.1f\n", time_array[0][i],
                       wave_array[0][i]);
328             }
329         }
330         // print progress update
331         std::chrono::duration<double> elapsed =
               std::chrono::system_clock::now() - begin;
332         printf("\rRunning for %.0f seconds", elapsed.count());
333
334     counter++;
335     }
336 }
337
338 // Does a scan for boards and initializes with user-defined parameters
       (or defaults.) Always picks first board as we only support one board.
       Returns 0 if no board found.
339 int initScan(DRS* &drs, DRSBoard* &b){
340     // show any found board(s)
341     for (int i=0 ; i<drs->GetNumberOfBoards() ; i++) {
342         b = drs->GetBoard(i);
343         printf("Found DRS4 evaluation board, serial #%d, firmware
               revision %d\n",
344             b->GetBoardSerialNumber(), b->GetFirmwareVersion());
345     }
346
347     // exit if no board found
348     if (drs->GetNumberOfBoards() == 0) {
349         printf("No DRS4 evaluation board found\n");
350         return EXIT_FAILURE;
351     }
352
353     // continue working with first board only
354     b = drs->GetBoard(0);
355
356     //initialize board
357     b->Init();
358
359     // set sampling frequency
360     b->SetFrequency(argFreq, true);
361
362     // enable transparent mode needed for analog trigger
```

```
363     b->SetTranspMode(1);
364
365     //set input range to given value, +/- 0.5. Ex. 0 gives a range from
            -0.5 to 0.5
366     b->SetInputRange(argRange);
367
368     // The following lines enables hardware trigger on CH1 at given
            voltage positive edge
369     if (b->GetBoardType() >= 8) {         // Evaluaiton Board V4&5
370         b->EnableTrigger(1, 0);           // enable hardware trigger
371         b->SetTriggerSource(1<<0);        // set CH1 as source
372     } else if (b->GetBoardType() == 7) { // Evaluation Board V3
373         b->EnableTrigger(0, 1);           // lemo off, analog trigger on
374         b->SetTriggerSource(0);           // use CH1 as source
375     }
376     b->SetTriggerLevel(argTrigLvl);       // Trigger level. Defualt: 0.05 V
377     b->SetTriggerPolarity(false);         // positive edge
378
379     // set trigger delay. Default: zero ns trigger delay
380     b->SetTriggerDelayNs(argTrigDelay);
381     return EXIT_SUCCESS;
382 }
```

Listing A.2: Temperature readout

```
1  # SPDX-FileCopyrightText: 2021 ladyada for Adafruit Industries
2  # SPDX-License-Identifier: MIT
3
4  # https://github.com/adafruit/Adafruit_CircuitPython_MAX31865
5
6  import time
7  import board
8  import digitalio
9  import adafruit_max31865
10
11 # Create sensor object, communicating over the board's default SPI bus
12 spi = board.SPI()
13 cs = digitalio.DigitalInOut(board.D5)  # Chip select of the MAX31865
       board.
14 sensor = adafruit_max31865.MAX31865(spi, cs)
15
16 # Read temperature.
17 temp = sensor.temperature
18 # Print the value.
19 print(temp)
```

Listing A.3: Graph plotter

```
1 import csv
2 import re
3 import pandas as pd
4 import seaborn as sb
```

```
5  import matplotlib.pyplot as plt
6
7  # Create CSV file and writer
8  f = open('data.csv', 'w', newline='')
9  writer = csv.writer(f)
10
11 # Regex matches for temp and time
12 regex = re.compile(": (\d*\.\d*),.*: (\d*)")
13
14 with open("data.txt") as data:
15     # Go through each line of data.txt
16     for line in data:
17         # See if it matches with regex
18         for match in regex.finditer(line):
19             # Write a new row with regex matches
20             writer.writerow([match.group(1),match.group(2)])
21
22 #Close writer and CSV file, data.csv
23 f.close()
24
25 # Read data.csv, defining column names and setting index column set to
       'ms'.
26 data = pd.read_csv('data.csv', names=['temp', 'ms', 'count'],
       index_col='ms')
27
28 # Rewrite time column to panda timedelta milliseconds
29 data.index = pd.to_timedelta(data.index, unit='ms')
30
31 # Resample to calculate mean temperature and number of events every 30
       minutes
32 resamp = data.resample('30min').agg({'temp': ['mean', 'count']})
33
34 # Flattens multiindex columns and fixes names
35 resamp.columns = resamp.columns.map('_'.join)
36
37 print(resamp)
38
39 # Save the resampled data as CSV
40 resamp.to_csv('result.csv')
41
42 # Create plots
43 sb.set_style("darkgrid")
44
45 fig, ax =plt.subplots(1,3)
46
47 res = sb.lineplot(x='ms', y='temp_mean', data=resamp, ax=ax[0], ci='sd')
48
49 res.set(xlabel='Time [ms]', ylabel='Mean temperature [°C]')
50
51 res2 = sb.lineplot(x='ms', y='temp_count', data=resamp, ax=ax[1], ci='sd')
52
53 res2.set(xlabel='Time [ms]', ylabel='# events pr 30 minutes')
```

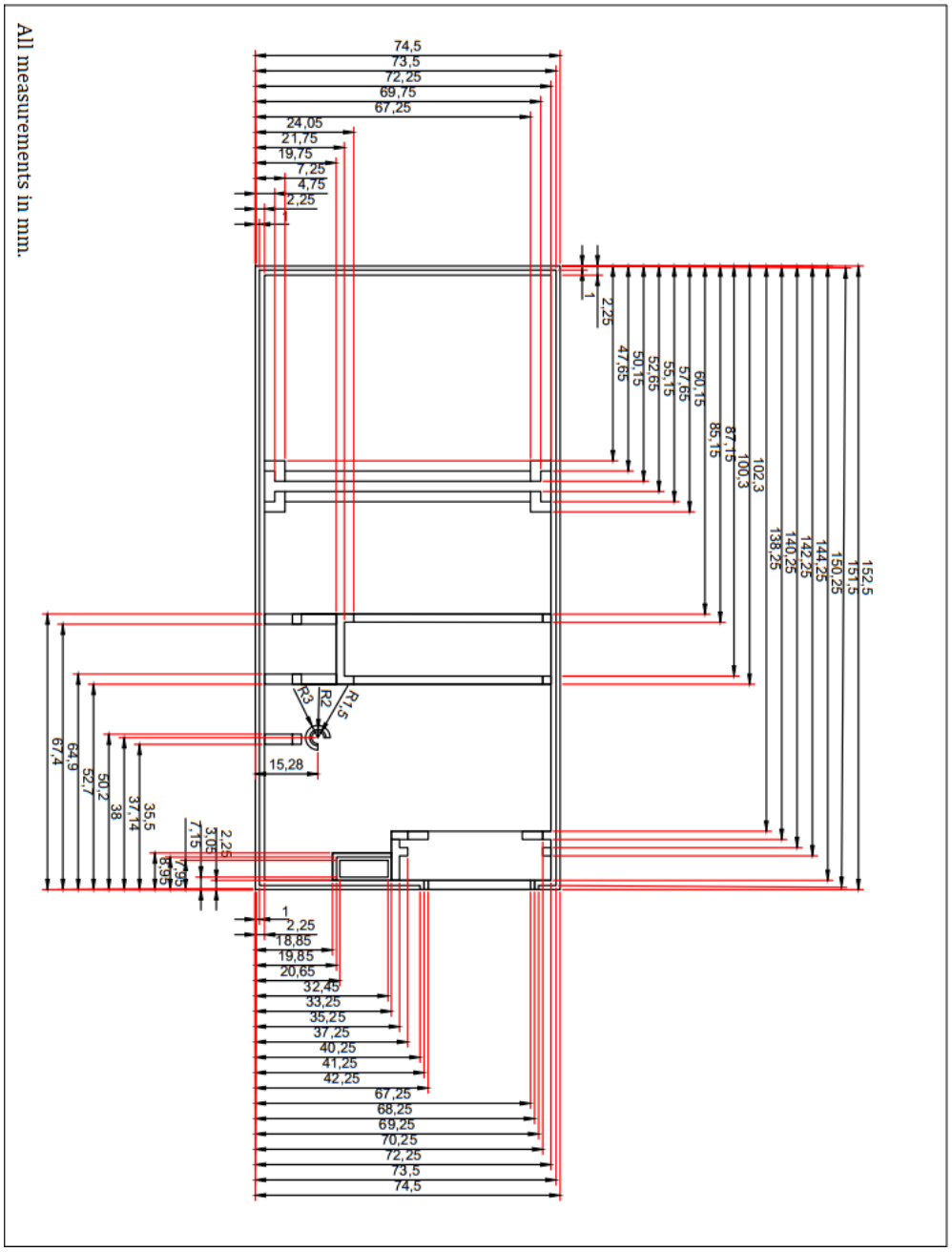
```
54
55 res3 = sb.scatterplot(x='temp_count', y='temp_mean', data=resamp,
       ax=ax[2])
56
57 res3.set(xlabel='# events pr 30 minutes', ylabel='Mean temperature [°C]')
58
59 plt.show()
60
61 fig.savefig("result.pdf", bbox_inches='tight')
```

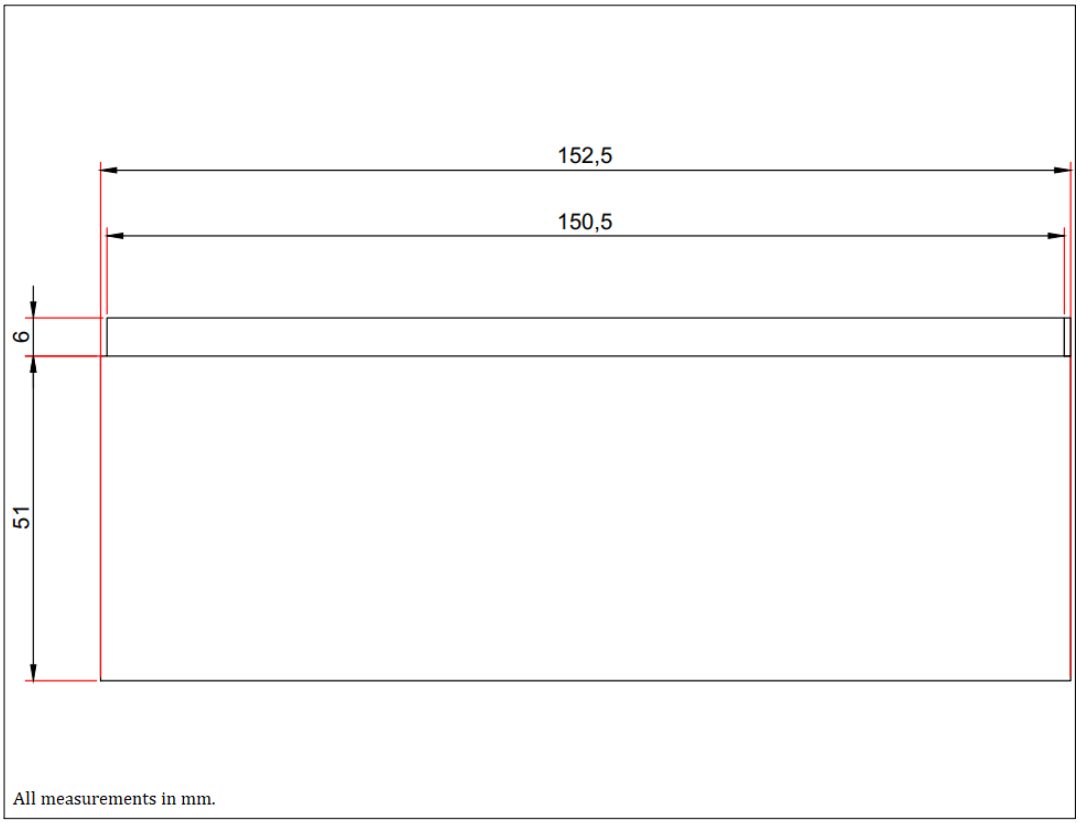
Listing A.4: Climate chamber

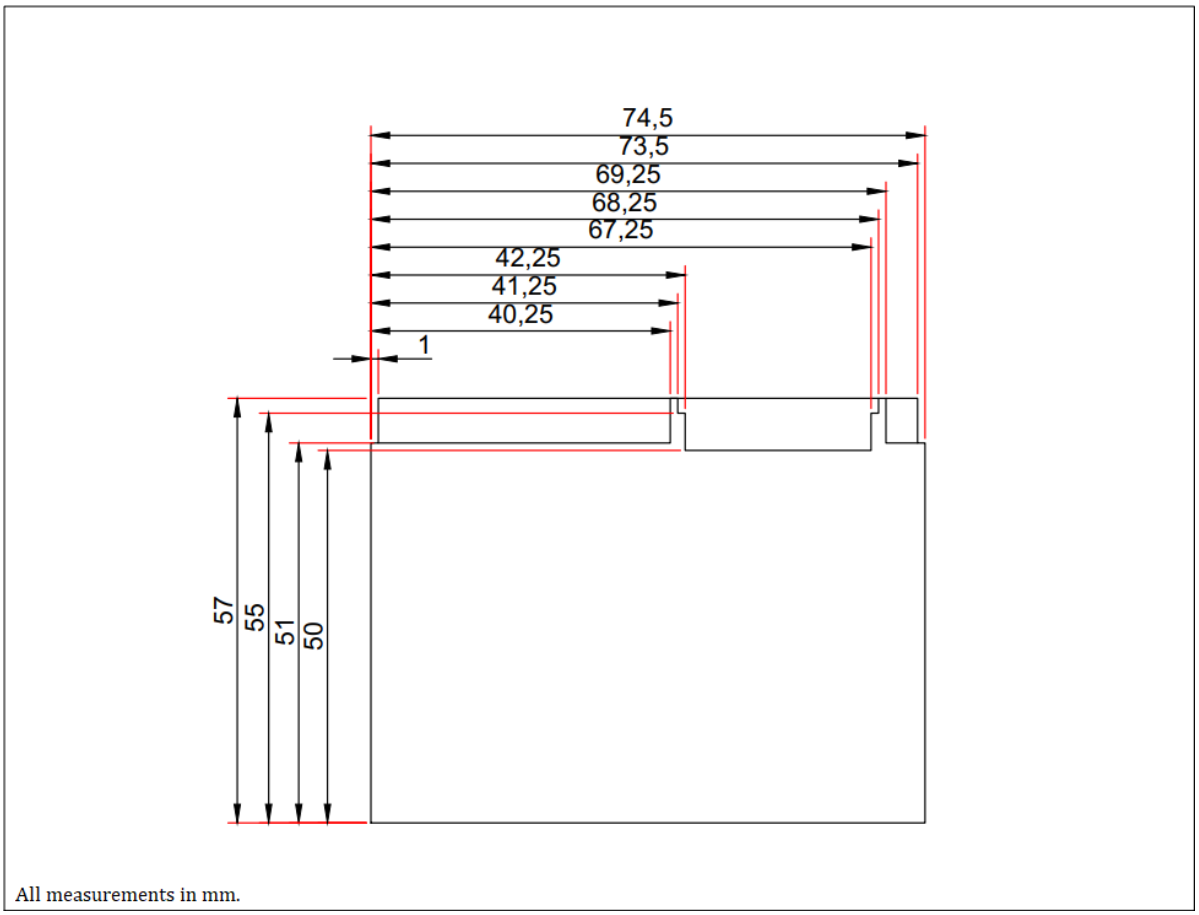
```
1  import datetime as dt
2  import serial.tools.list_ports
3  import cts_connection_serial as connect
4
5  ports = list(serial.tools.list_ports.comports())
6  for p in ports:
7      print(p)
8      if 'MOXA USB Serial Port' in p.description:
9          port = p
10         print('Connecting...')
11
12 cts = connect.Connection(port=port.device, baud=19200, timeout=0.25)
13 response = cts.read_set_temp_and_actual()
14 print('Set temp:', response[0], '  Actual temp:', response[1])
15
16
17 k = 0
18
19 temp_list1=[5, 10, 15, 20, 25, 30, 35, 40, 45]
20 temp_list2=[50, 45, 40, 35, 30, 25, 20, 15, 10]
21
22 hourly = dt.datetime.now()
23 minute = dt.datetime.now()
24
25 f = open("bachelorbrostemp.txt", "a")
26 f.write("date,set_temp,acc_temp\n")
27 f.close()
28
29 while True:
30     while k < len(temp_list1):
31         if dt.datetime.now() > minute:
32             minute = dt.datetime.now() + dt.timedelta(minutes=1)
33             f = open("bachelorbrostemp.txt", "a")
34             f.close()
35
36         if dt.datetime.now() > hourly:
37             f = open("bachelorbrostemp.txt", "a")
38             hourly = dt.datetime.now() + dt.timedelta(minutes=120)
39             cts.set_temp(temp_list1[k])
40             print("Setting temperature to", temp_list1[k], "degrees C
```

```
                at", dt.datetime.now())
41              f.write(str(dt.datetime.now()))
42              f.write(",")
43              f.write(str(cts.read_set_temp_and_actual()[0]))
44              f.write(",")
45              f.write(str(cts.read_set_temp_and_actual()[1]))
46              f.write("\n")
47              k += 1
48              f.close()
49      k = 0
50
51      while k < len(temp_list2):
52          if dt.datetime.now() > minute:
53              minute = dt.datetime.now() + dt.timedelta(minutes=1)
54              f = open("bachelorbrostemp.txt", "a")
55              f.close()
56
57          if dt.datetime.now() > hourly:
58              f = open("bachelorbrostemp.txt", "a")
59              hourly = dt.datetime.now() + dt.timedelta(minutes=120)
60              cts.set_temp(temp_list2[k])
61              print("Setting temperature to", temp_list2[k], "degrees C
                    at", dt.datetime.now())
62              f.write(str(dt.datetime.now()))
63              f.write(",")
64              f.write(str(cts.read_set_temp_and_actual()[0]))
65              f.write(",")
66              f.write(str(cts.read_set_temp_and_actual()[1]))
67              f.write("\n")
68              k += 1
69              f.close()
70      k=0
```

A.3 Technical drawings




152,5
150,5
6
51
All measurements in mm.


74,5
73,5
69,25
68,25
67,25
42,25
41,25
40,25
1
57
55
51
50
All measurements in mm.

A.4 Bill of materials

Component	Manufacturer	Model information	Cost in NOK
Micro computer	Raspberry Pi	Raspberry Pi 4	636,48
Power supply cable	Raspberry Pi	Raspberry Pi 5V supply	105,36
Temperature sensor	E+E Elektronik	PT1000 – 2 wire	158,00
Phototransistor	Osram Opto Semiconductors	900nm – 15mA – 35V	5,58
Rubber foam	Eurostat	100x100 mm	229,00
GPIO Socket Connector	Raspberry Pi	40 Pin GPIO Socket Connector	47,85
USB-to-Serial TTL cable	FTDI	USB-TTL-cable for RPI	220,44
PT1000 – Amplifier	Adafruit	PT1000 Amplifier w/ADC	182,33
Micro switch	Panasonic	Ultraminiature Micro Switch	32,20
Pin header	Phoenix Contact	Straight – 3,5mm – 20 poles	13,56
Memory card	SanDisk	Extreme 128GB	350,71
Relay	RND Components	PCB - effect relay	10,43
SMA cables	Nedis	Male to female 1m / 3m	302,00
		TOTAL	2293,94