Gezamenlijke opleiding UHasselt en KU Leuven

2020•2021
Faculteit Industriële Ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Software development for the operations and external communications of the OSCAR-QUBE project

PROMOTOR :
Prof. dr. ir. Ronald THOELEN
Prof. dr. Milos NESLADEK

BEGELEIDER :
De heer Jaroslav HRUBY

Sam Bammens

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

2020•2021

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Software development for the operations and external communications of the OSCAR-QUBE project

PROMOTOR :

Prof. dr. ir. Ronald THOELEN

Prof. dr. Milos NESLADEK

BEGELEIDER :

De heer Jaroslav HRUBY

Sam Bammens

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

KU LEUVEN

Acknowledgement

In this chapter I would like to express my gratitude towards everyone who helped in the realization of this master's thesis. First of all I would like to thank my supervisor Ir. Jaroslav Hruby for support and guidance. Furthermore I would like to thank the OSCAR-QUBE team for all of the effort and support they brought to the table. Finally I would like to thank the people from "Orbit Your Thesis!" for the opportunity and guidance they have provided the team with.

Contents

Acknowledgement 1
Table of figures 5
Table of tables 7
Table of abbreviations 9
Abstract 11
Abstract in Dutch 13
Chapter 1: Introduction 15
1.1 Aim of the thesis 16
1.2 Related work 16
1.3 Outline of the thesis 16
Chapter 2: Literature study 17
2.1 Magnetometers 17
2.2 NV-centers and Magnetic Resonance 18
2.3 Mission Control Software 22
2.4 Communication Protocols and packet structures 23
Chapter 3: Materials and Methods 27
3.1 OSCAR-QUBE magnetometer 27
3.2 Mission Control Software 28
3.3 Utilized Protocols 29
3.4 Graphical User Interface (GUI) 30
3.5 Wireshark 32
3.6 Nucleo-F746ZG board 32
Chapter 4: Experimental 35
4.1 Installing Yamcs 36
4.2 Packet configuration 37
4.2.1 Telecommanding packets 37
4.2.2 Telemetry packets 38
4.3 User Interface 40
4.3.1 PyQt5 and Designer 40
4.3.2 Layout 41
4.3.3 Yamcs-client API 43
4.3.4 Threading 44
4.3.5 TM data visualization 47
4.3.6 Pulsed visualization 48

4.3.7 Graph Zeroing ... 49
4.3.8 Exporting ... 49
Chapter 5: Results ... 51
5.1 Back end ... 51
5.2 Front end ... 53
5.2.1 Control tab results ... 53
5.2.2 Monitor tab results ... 57
5.2.3 Peak and Pulsed tab results ... 59
5.2.4 Threads ... 60
5.2.5 Functionality ... 61
Chapter 6: Evaluation ... 65
6.1 Reliability ... 65
6.2 Ease of use ... 65
6.3 Future work ... 65
Chapter 7: Conclusion ... 67
Literature ... 68
Appendix A ... 69
Appendix B ... 71

Table of figures

Figure 1: Magnetic sensors with their dynamic range and field of application [1] 17
Figure 2: NV-center in a diamond structure [4] 18
Figure 3: The energy level scheme of an electron within an NV-center during ODMR [4] 19
Figure 4: The concept of PDMR based on ODMR [7] 21
Figure 5: Mission control dashboard [8] 22
Figure 6: Comparison of the OSI and TCP/IP models [10] 23
Figure 7: Protocols with their associated layer [11] 24
Figure 8: CCSDS packet structure[13] 25
Figure 9: Schematic representation of the internal and external communication of the embedded system 28
Figure 10: Schematic representation of interfacing between the GUI and the embedded system when using the Display runner from Yamcs Studio 31
Figure 11: Schematic representation of interfacing between the GUI and the embedded system when using a python script as GUI 32
Figure 12: A Nucleo-F746ZG board [18] 33
Figure 13: General overview of connection of UHB and QUBE 35
Figure 14: Communication and IP configuration 36
Figure 15: MDB configuration 37
Figure 16: The correlation between the three stages within the XTCE structure 38
Figure 17: Packet differentiation process based on APID 40
Figure 18: General layout Control tab 41
Figure 19: General layout controls 42
Figure 20: General layout Monitor tab 42
Figure 21: General layout Peak and Pulsed tab 43
Figure 22: Action button telecommanding flow 44
Figure 23: Radio button telecommanding flow 44
Figure 24: Threading scheme 45
Figure 25: Process of Ping thread 45
Figure 26: Process flow interface thread 46
Figure 27: Convert QWidget to PyQtGraph 47
Figure 28: Visualization process of monitor parameters 48
Figure 29: Unit step function 48
Figure 30: Visualization process of pulsed data 49

Figure 31: Zeroing procedure ... 49
Figure 32: Layout export group ... 50
Figure 33: Final ParameterTypeSet definition ... 51
Figure 34: Header container definition ... 52
Figure 35: Debug packet structure definition ... 52
Figure 36: Control tab ... 53
Figure 37: Detail of system group ... 54
Figure 38: Commanding box of laser group ... 54
Figure 39: Laser strength selector group ... 54
Figure 40: Detail of microwave group ... 55
Figure 41: Detail of FPGA group ... 55
Figure 42: Detail of Debug group ... 56
Figure 43: Detail of ODMR readout graph ... 56
Figure 44: Detail of ODMR average graph + clear button ... 57
Figure 45: Monitor tab with zeroing activated on all plots ... 57
Figure 46: Monitor tab with zeroing disabled on all plots ... 58
Figure 47: Detail of export group ... 58
Figure 48: Peak and Pulsed tab ... 59
Figure 49: Detail of peak detection half ... 59
Figure 50: Detail of pulsed operation half ... 60
Figure 51: ODMR readout display in sweep mode ... 61
Figure 52: ODMR average over time display ... 62
Figure 53: Reference subsystem displays in their zeroed state ... 62
Figure 54: Stepping through pulse steps ... 63

Table of tables

Table 1: Crucial differences between TCP and UDP [12] 24
Table 2: Comparison between TGSS and Yamcs 29
Table 3: Fabrication of sendStringCmd datatype 38
Table 4: TC "Commands" tab spreadsheet configuration 38
Table 5: Parametertypes and the appropriate configurations to support TM of the QUBE 39
Table 6: Defined arrays and their corresponding parametertypes 39
Table 7: APID with corresponding packet type 39

Table of abbreviations

AC	Alternating Current
ASCII	American Standard Code for Information Interexchange
ADC	Analog Digital Converter
APID	Application ID
CCSDS	Consultative Committee for Space Data Systems
ESA	European Space Agency
GB	Gigabyte
GUI	Graphical User Interface
ICF	Ice Cube Facility
IP	Internet Protocol
ISS	International Space Station
lwIP	Lightweight IP
LED	Light Emitting Diode
MCS	Mission Control Software
MDB	Mission Database
MEMS	Microelectromechanical system
NASA	National Aeronautics and Space Administration
NV-centers	Nitrogen Vacancy centers
ODMR	Optical Detection of Magnetic Resonance
OSI	Open Systems Interconnection
PDMR	Photocurrent Detection of Magnetic Resonance
PUS	Packet Utilization Service
S/N	Signal to Noise
SQUID	Superconducting Quantum Interference Device
TC	Telecommand
TCP	Transmission Control Protocol
TGSS	Terma Ground Segment Suite
TM	Telemetry
UDP	User Datagram Protocol
UHB	User Home Base
VPN	Virtual Private Network
XML	Extensible Markup Language
XTCE	XML Telemetric and Command Exchange

Abstract

The measurement process automation and system health monitoring is essential in remote sensing applications. The multiparametric systems, such as diamond quantum magnetic field sensors require constant monitoring to ensure reliability of the acquired data. As this system will be located onboard the International Space Station (ISS), there is a constant stream of incoming data to acquire and process, as well as the need to schedule and handle automated measurement routines, and to resolve the situation during the loss of signal (LOS), and provide users with a clear GUI. This thesis aims to develop an automated control system capable of receiving telemetry, sending telecommands and performing scheduled tasks for the experiment onboard the ISS.

A user home base (UHB) was created to interface with the embedded system during the mission. To meet the requirements, a mission control software (MCS) was selected and configured. After this, a GUI was created using PyQt5 which interfaces with the MCS to operate the QUBE. Yamcs was selected as the MCS and after the networking and packet structure configuration it was able communicate with the embedded system. After which the control layout was developed and the supporting Python script for the system automation was written. The UHB was thoroughly tested on reliability and user experience during its development and the interface test campaign of the system. It was found it fulfilled all the requirements with substantial margins, meaning the system will be used during the mission.

Abstract in Dutch

Metingen uitvoeren, automatisatie en de gezondheid van het systeem monitoren is essentieel voor afstandsbediende meetapplicaties. Een multiparametrisch systeem, zoals diamant gebaseerde kwantum magnetometers, hebben permanente monitoring nodig om de betrouwbaarheid van de verworven data te verzekeren. Omdat deze magnetometer gelokaliseerd zal zijn aan boord van het International Space Station (ISS) is er een constante stroom van verworven data om te ontvangen en te verwerken. Ook is er een nood aan geplande en geautomatiseerde meetroutines. Daarbij zal het systeem om moeten kunnen gaan met situaties waarbij de connectie met de magnetometer wegvalt. Verder is het belangrijk dat er een overzichtelijke GUI wordt aangeleverd voor de operator van het systeem. Deze thesis heeft als doel een automatisch controle systeem te ontwikkelen dat in staat is om telemetrie te ontvangen, telecommands te verzenden en geplande taken uit te voeren voor het experiment dat zich aan boord van het ISS bevindt.

Een user home base (UHB) is gemaakt om te interageren met de magnetometer tijdens zijn missie. Om de vereisten te behalen werd een mission control software (MCS) geselecteerd en geconfigureerd. Vervolgens werd er een GUI gecreëerd met PyQt5 dat interageert met de MCS om de QUBE te kunnen besturen. Yamcs werd gekozen als MCS en na de configuratie, van het netwerk en de pakket structuur, kon het systeem met de magnetometer communiceren. Hierna werd de lay-out, met het ondersteunende Python script voor de automatisatie, van de UHB gemaakt. De UHB werd grondig getest op betrouwbaarheid en gebruiksvriendelijkheid tijdens zijn ontwikkeling en vervolgens tijdens de interface test campagne van het algemeen systeem. Het systeem slaagde voor deze testen met ruime marge, wat betekent dat de UHB gebruikt gaat worden tijdens de missie van de magnetometer.

Chapter 1: Introduction

Nowadays there are a lot of devices on the market to monitor the magnetic field. Each device has their specific dynamic range, sensitivity and accuracy, meaning each system has their specific requirements and limitations. However, the OSCAR-QUBE team created a diamond-based quantum magnetometer that aims to revolutionize this market. This is achieved by using readout methods created and studied by the Quantum Photonics research group where the project is located. The aim of the project is to fabricate a system that is superior in all three categories when compared to the other available technologies by implementing these readout methods.

The system created by the team was selected by the European Space Agency (ESA) to fly onboard the International Space Station (ISS). Once activated, it will monitor the magnetic field of the earth autonomously for ten months. Scientific data will be collected during the mission. Meanwhile, the team shall monitor the system and study its behavior while pushing the boundaries of it in real world applications. However, the team wants to be able to command the experiment from the ground while the project is running on the ISS. This commanding should be able to be performed by scientists that have no technological background on the system. In the meantime, there has to be a system that is capable of receiving the captured data directly from space. This data will have to be saved in order to postprocess the captured information. Furthermore, certain parameters within these telemetry packets will have to be visualized.

In order to fulfil these requirements, a user home base (UHB) has to be created. This is a computer assigned and configured to communicate with the system onboard the ISS. On this computer a mission control software (MCS) will be running. This software is both responsible for receiving the telemetry data gathered by the cube as well as sending the telecommands to reconfigure the embedded system. As an addition, most MCS automatically store the telemetry (and telecommand) packets as well as provide basic visualization too1.1

1.1 Aim of the thesis

This thesis aims to develop a system that is capable of the following requirements:

- The UHB should be able to handle incoming TM. This means the system can receive, identify, read and store incoming telemetry packets.

- Provide commanding capabilities. Since the QUBE can change its configuration using TC, it is crucial the system is able to execute these tasks.

- Provide a GUI which the operator can use to interface with the QUBE. This GUI should be intuitive and provide a clear overview of the system as a whole. Furthermore, a person with little to no technological background should be able to operate the QUBE using the GUI.

1.2 Related work

Since this is a one-of-a-kind project there is no closely related work. However, other small space projects were studied in order to extract useful pieces of information. Furthermore, other non-space related projects and concepts were investigated as they could be applied or used within the scope of the thesis.

1.3 Outline of the thesis

Chapter 2 will explain the scientific and technological backgrounds of concepts which are used during the making of the thesis. This will provide more insight in the operation of the embedded system and the design choices which were made. Chapter 3 will give an overview of all the materials and methods which were used during the development of the UHB. In chapter 4 the actual steps taken in order to achieve the current UHB can be found. Chapter 5 will show the final results. Lastly, in chapter 6 the results will be evaluated, and the conclusion is made in chapter 7.

Chapter 2: Literature study

2.1 Magnetometers

Monitoring the magnetic field is crucial in a broad range of fields. Therefore, there are various different methods that exist to execute this task. Each method has their specific requirements and will have their specific properties, meaning the preferred magnetometer for a particular task will depend on the requirements of this task and the properties of the sensor. However, the two main factors considered when selecting a type of magnetometer are dynamic range and the cost of the system.

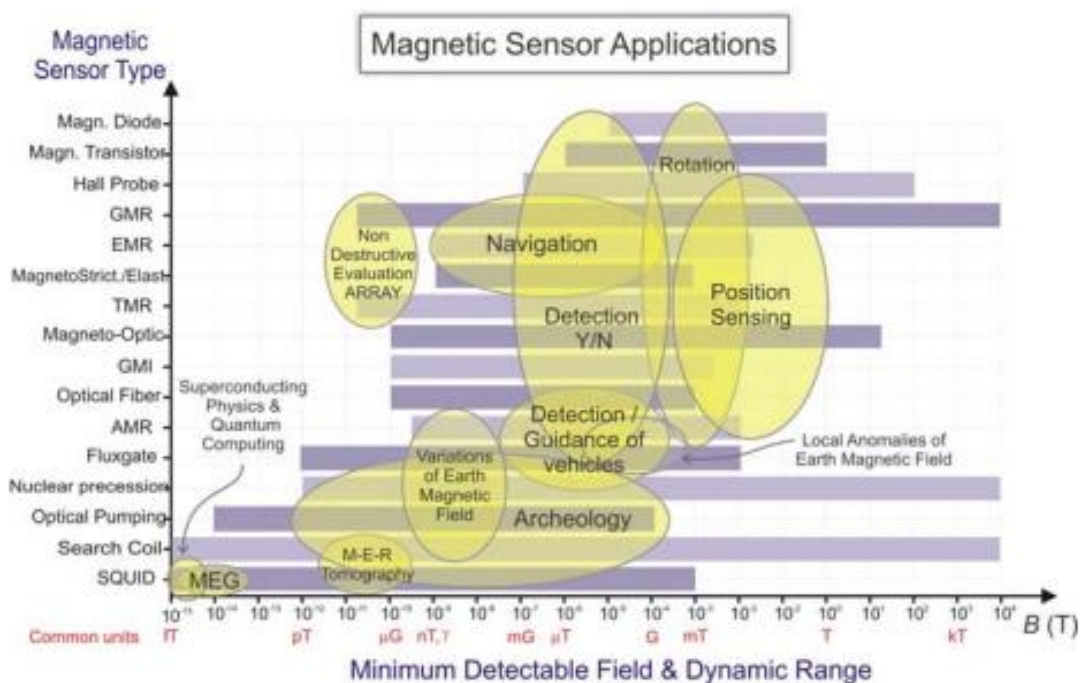

Figure 1: Magnetic sensors with their dynamic range and field of application [1]

As can be seen in Figure 1, the dynamic ranges of these technologies are spread out over the spectrum. The systems available with a high dynamic range are usually complex, costly or of a substantial formfactor. Therefore, if an application demands for the monitoring of the magnetic field over a larger dynamic range, multiple types of sensors are required to solve this efficiently. However, this means more costs and complexity will be added to the application.

Furthermore, the vectors of the magnetic field might be desired for certain applications. However, not all of the readout methods support this feature. Hence this will also have to be taken into consideration when selecting a magnetometer. Common sensors capable of measuring the vectorized magnetic field are [1]:

- fluxgates
- hall effect sensors
- Superconducting Quantum Interference Devices (SQUID)

However, each of these systems have specific benefits and drawbacks. The fluxgate for example is rather simple but lacks extreme sensitivity. SQUIDs on the other hand can reach this sensitivity but are complex systems which are mostly used in labs and kept away from noise. [2][3]

2.2 NV-centers and Magnetic Resonance

To combat the problems that occur when using regular vectorized magnetometers, new readout methods are developed which might solve these problems. These systems utilize diamonds infused with nitrogen vacancy (NV) centers and use its isolated electronic spin system to monitor the magnetic field. This enables the system to detect weak magnetic fields and provides a high spatial resolution (in the sub-nm range) while remaining highly sensitive. Currently these systems are relatively big in size when compared to the other sensors, but a lot of resources are put to the miniaturization of these systems. So far these efforts prove to be successful. [4][5]

At the core of these systems is a diamond infused with NV-centers. These NV-centers are the actual sensing part of the system, and they are formed when nitrogen atoms are trapped next to a vacancy within the diamond's structure (Figure 2). These vacancies naturally occur as defects in the diamond lattices. To date two different types of NV-centers have been known, namely the neutral NV-centers (NV^0) and the negatively charged centers (NV^-). Only the latter type proves to be useful in the detection of the magnetic field due the fact that a triplet spin ground level can be initialized, thus further mentions of NV-centers reference to the NV^--centers. [4]

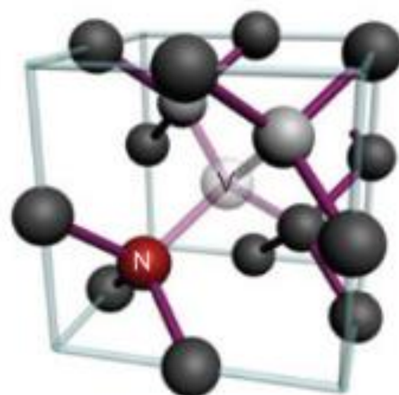

Figure 2: NV-center in a diamond structure [4]

ODMR is a readout method which utilizes the properties of these NV-centers to optically detect the strength of the vectors of the magnetic field. In order to achieve this, a green laser (532 nm) is shone onto the diamond. This will excite the electrons from the NV-centers within the diamond to a higher energy state. The excited state is temporal and after some period, the electron will relax back to its stable state. In this process the electron releases the excess amount of energy in the form of red light (637 nm). The intensity of the emitted light is stable, unless the system is under the influence of a microwave field.

Exposing this system to a microwave field will alter the behavior of the emitted red-light intensity. This is due to the fact that the NV-centers have a single resonant frequency within the microwave domain. Under the influence of the resonant frequency, a dip in the red-light intensity will occur while the rest of the spectrum remains stable as is shown in Figure 3. This phenomenon is called a dark transition. These transitions emit less light because the electron will first go through the metastable state before returning to the ground state and therefore emit a smaller amount of energy.

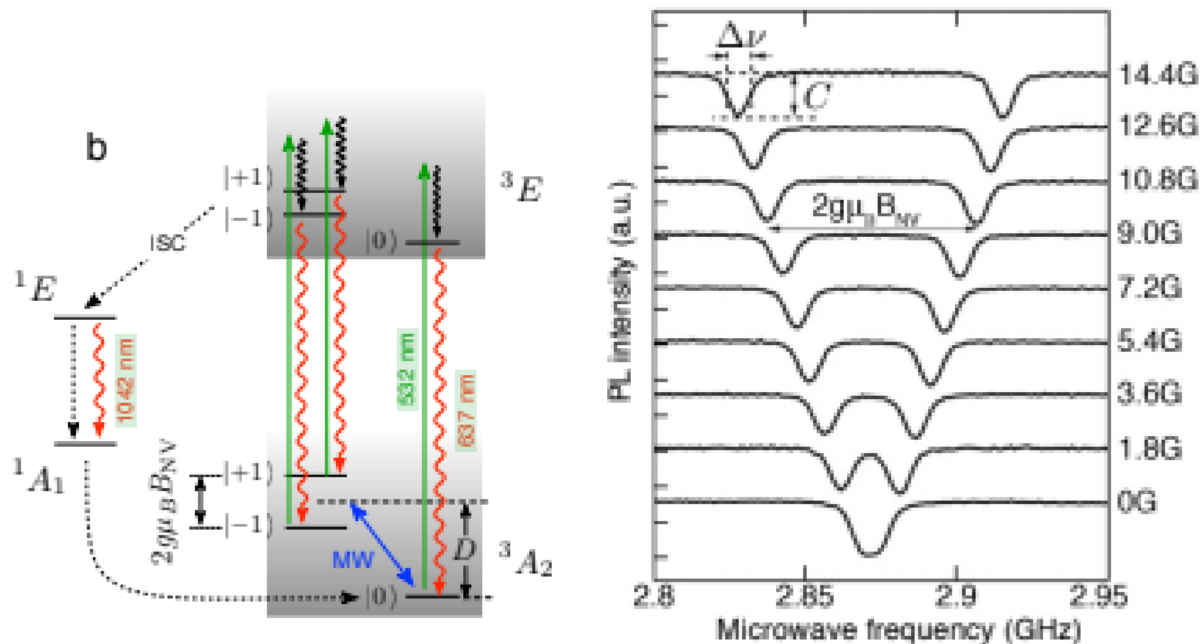

Figure 3: The energy level scheme of an electron within an NV-center during ODMR [4]

Exposing this system to a magnetic field will cause the regular ground state to split into two separate ground states. This phenomenon is called Zeeman splitting. These new ground states are further apart from each other based on the strength of the magnetic field. However, these two new ground states remain symmetric around the original ground level. This event will produce two different resonant frequencies in the microwave domain, meaning the original peak from the resonant frequency also splits into two separate peaks. Since these resonant frequencies are based on the state of the ground levels, they will remain symmetric around the original resonant frequency. Now, the strength of the magnetic field can be calculated based on the separation between those peaks, since the resonant frequencies will shift further apart if the magnetic field is stronger. Furthermore, upon determining the symmetry point in the frequency spectrum, only

one of the two peaks' position is necessary to calculate the strength of the magnetic field. [4] [5] [6]

However, NV-centers have a tetrahedral shape since they are located within a diamond's crystallographic structure. Meaning there are four orientations bound to a single NV-center. Therefore, there are four systems producing their own ODMR signal according to the strength of the field corresponding to their orientation. Because of this, eight peaks can be detected when using NV-centers. Nevertheless, all of the peaks will remain symmetric around a single frequency within the microwave spectrum. Using the information of the strength of the field on every orientation of the NV-center, the magnetic field can be monitored three dimensionally.

The ODMR principle is rather simple, however technically it creates various inefficiencies. Mainly due to the fact that the emission of red light has to be monitored. Therefore, a suiting optical sensor has to be employed to detect this signal, adding complexity to the system and possibly limiting performances. Since the sensitivity and accuracy of the readout are directly related to performance of the sensor. As an addition, the signal is more likely to get interference from other (light) sources, thus decreasing the system's performance and usability. To combat these issues, another readout method was created. This readout method is called the Photocurrent Detection of Magnetic Resonance (PDMR). This method utilizes the same principles as the ODMR-method, however it bases its readout system on the generation of photocurrents. These currents are created when the laser excites the electrons to their conduction band, which is of a higher energy state than the excited energy level used in ODMR. Upon applying a bias voltage to this system, the generated photocurrent can be read out by various sensors such as Analog Digital Converters (ADC) which generally perform better than photosensors. This causes the system to be more sensitive and accurate while reducing the changes of noise. Additionally, the PDMR readout method also aids the miniaturization process of the magnetometer.[4][6]

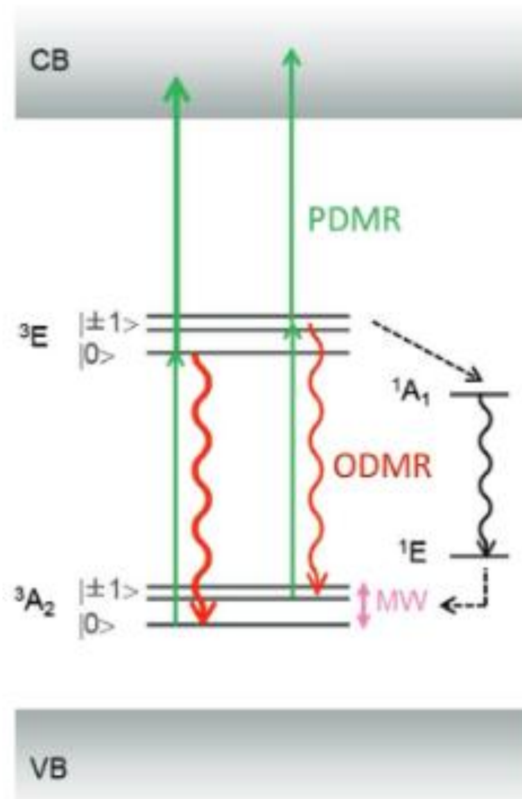

Figure 4: The concept of PDMR based on ODMR [7]

So far only the constant wave mechanisms have been explained. Here, the critical systems such as the laser, microwave and readout subsystems are constantly active. This will produce a sensitive and accurate system that is able to monitor the magnetic field three dimensionally. However, this can be further improved by utilizing pulsed schemes. When using pulsed schemes, the afore mentioned subsystems are triggered at a specific time for a specific duration. The benefit of using pulse schemes varies. Most pulse sequences are aimed toward increasing the contrast, sensitivity and/or improving the S/N ratio. Additionally, some pulse schemes might come with extra benefits, such as the Ramsey pulse scheme. This sequence improves the sensitivity of the measurements in the same way as Ramsey magnetometry. However, using the pulsed ODMR version, substantially less high microwave fields are necessary to achieve these improvements. Another well-known pulse scheme is the Hahn echo sequence. This scheme is mainly used to restrict sensing to AC signals. [6][7]

2.3 Mission Control Software

MCS are commonly used within the army, space sector and aviation sector. This software is responsible for the coordination and execution of a mission. It runs on a computer which is configured to receive incoming data and send commands to operate the machine executing the mission. Although MCS are used in a variety of applications, the core of the software remains the same and most packages offer the same general features. These features are:

- identifying different packet types and structures of incoming telemetry
- formatting of incoming data packets
- visualization of the data
- issuing commands
- data storage

More advanced MCS can also support alarms, automated procedures or have built in planning tools. Next to these feature the MCS aims to provide an overview of the mission environment and its status. Based on this information, the operator will validate the progression of the mission and use it to make rest of the planning.

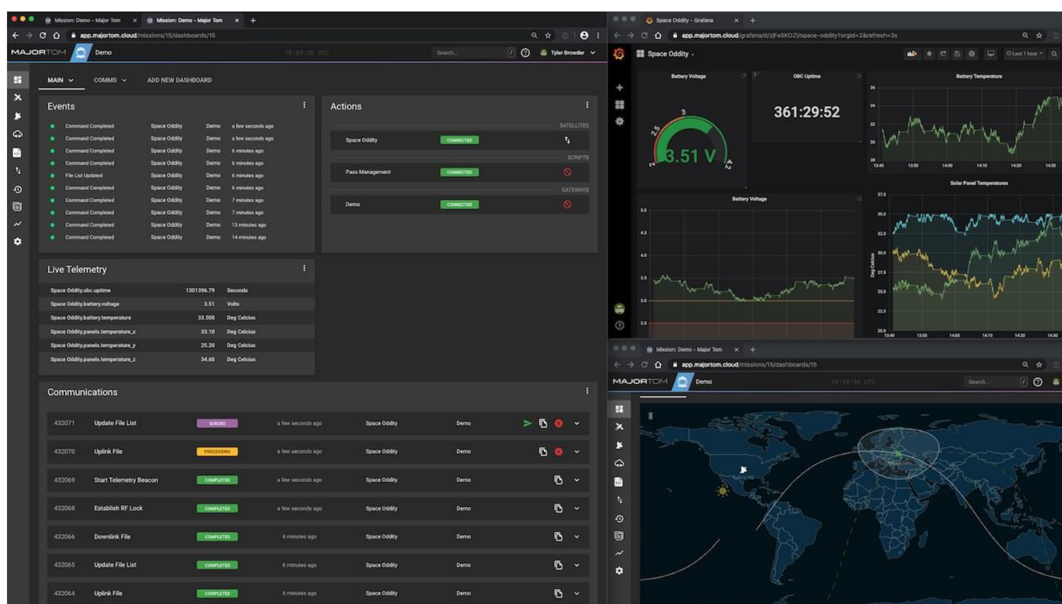

Figure 5: Mission control dashboard [8]

Within the space sector, these systems are mainly used to control missions of rockets, satellites or various other systems in space. Since most instances have their own requirements and standards within their missions apart from the general rules, they utilize different MCS. However, sometimes an instance chooses to create a custom MSC, such as Open MCT, created by NASA. But other instances prefer software from a private company such as the Terma Ground Suite Segment (TGSS). Nonetheless, most instances execute various types of mission. Each with their specific requirements. Therefore, multiple MCS can be used within the same instance. If none of the existing MCS adhere to the demands of a mission, the space entity can choose to create custom software.

2.4 Communication Protocols and packet structures

In order to enable communication within a network of different entities, some form of standardization is required. Rules will be defined to guarantee that each client is able to communicate with or over the network. These standard rules are called protocols and can be compared with human language. If someone is speaking the same language as another person, communication can be achieved, and information can be exchanged. If these people try to interact using different languages, the other person will not be able to understand the messages and no information will be exchanged. When the analogy is applied to computer networks it can be seen why it is crucial for each client to communicate using a protocol the destined client is able to translate. As an addition, computer networks need specific information about the intended destination to deliver the message correctly. This information is added to the packet, thus it is part of the protocol. Meaning the protocol does not only have to be supported by the intended client but by the network as well.

The most well-known network is the internet. This network has multiple abstract layers. Each of which has their specific contribution to successfully communicate a message to the desired destination. The seven-layer OSI model was the first standard model for network communications, becoming an international standard in 1984. However, the modern internet is based on the TCP/IP model which consists of five layers, which is simpler compared to the OSI model. [9]

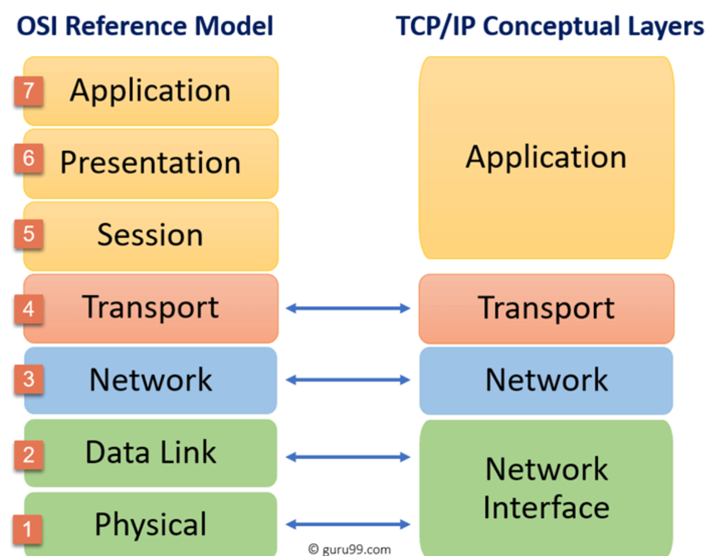

Figure 6: Comparison of the OSI and TCP/IP models [10]

As can be seen in Figure 6 there are only four layers on the TCP/IP side. This is due to the fact that occasionally the “Physical” and “Data link” layers are fused together and represented by the Network Interface layer. This layer is responsible for the transmission of the packets between two devices on the same network. The layer helps to define how data has to be sent using the network and will determine how the bits have to be transferred in the hardware responsible for the communication. The protocol associated with this layer is the ethernet protocol. [10]

Above the network interface layer there is the "Network" layer which is responsible for internetworking, meaning the "Network" layer will ensure the packet is transported correctly from source to destination. Common protocols from this layer are the IP, ICMP and ARP protocols. All of these protocols ensure that the network is constantly updated about the status of the clients and aids in the proper routing of the packets. [10][11]

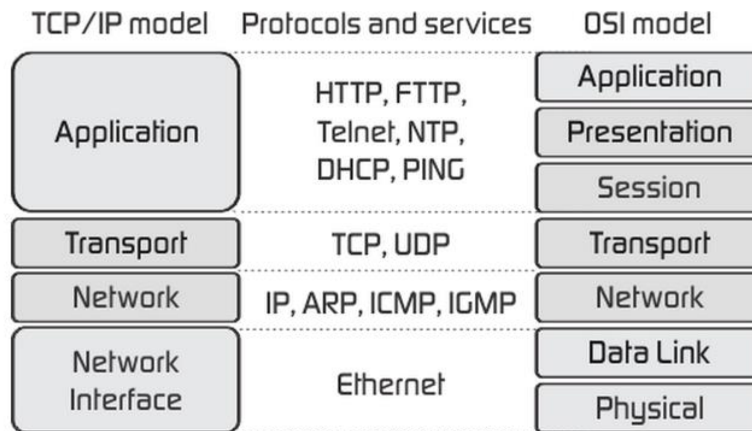

Figure 7: Protocols with their associated layer [11]

On top of this "Network" layer, the "Transport" layer can be found. This layer is responsible for controlling the flow, sequencing and error checking of the packets. Two protocols are part of this layer, namely the TCP and UDP protocols. The crucial differences between TCP and UDP are listed below in Table 1. [11]

Table 1: Crucial differences between TCP and UDP [12]

TCP	UDP
Live connection is required to transmit data	No live connection is required to transmit data
Requires handshakes to setup a connection	No handshakes required (connectionless protocol)
TCP rearranges data packets in the specific order	UDP protocols have no fixed order. Every packet is seen as independent
Does error checking and will resend a packet if the original packet returned an error message	Does error checking but disregards a faulty packet
TCP is reliable as it guarantees delivery of data to the destination router.	The delivery of data to the destination can't be guaranteed.
Is slower since it will wait for the acknowledgement of the receiver to send the next data.	UDP is faster due to the fire-and-forget mechanism

Lastly, the "Application" layer enables applications to access the network. A wide variety of protocols belong to this layer. For example, common protocols such as the HTTP protocol, the FTP protocol, NTP protocol etc. However, instances can create their own protocol for this layer since

the information within is bound to the application and not to the network. This means that different applications using the same protocols can both communicate with each other and exchange information. [11]

A standardized application protocol used within the space sector is the CCSDS protocol. A CCSDS packet exists out of three main components, as can be seen in Figure 8: a primary header, a secondary header and a data section. [13]

The primary header contains the general info, which is relevant to the whole system, therefore the structure of this header is static. These headers contain information such as: data length, packet ID, process ID, secondary header flags etc. The data within the primary header can be categorized into two categories. Firstly, the category which gives the system more information about the rest of the packet. A process ID, for example, falls under this category since it is used to identify the process or subsystem which is targeted to receive the rest of the packet's information. The other category, is in place to verify if the original data within the packet is still intact or if there were bitflips during communication procedures. [13][14]

Field:	Primary Header			Secondary Header TM			Data Section					
	-			-			Auxiliary header			Datapoints		
Byte:	0	...	5	6	...	11	12	...	X	X+1	...	Y

Figure 8: CCSDS packet structure[13]

The secondary header, if present, has a more dynamic structure, since it is bound to a specific process. Therefore, every process can have a custom secondary header structure. Within this header, it is specified which specific part of the process will have to be altered by the data portion of the packet.[13][14]

The last section, the data section, is completely customizable, thus there are no rules attached to the structure of this part. It is highly dynamic and it contains the data the system wants to communicate with the other side. Since there are no rules regarding the data section, it is possible to further structure this element by adding auxiliary headers and data parts.[13]

Chapter 3: Materials and Methods

3.1 OSCAR-QUBE magnetometer

The OSCAR-QUBE magnetometer was developed during the making of this thesis. The magnetometer manufactured by the OSCAR-QUBE team is a quantum magnetometer based on diamonds infused with NV-centers. This system employs the ODMR and PDMR readout methods to monitor the magnetic field. It was created as an iteration on the system created by the OSCAR-QLITE team. The current iteration of the sensor aims to further improve the performance and useability of the system by deploying the PDRM readout method next to the ODRM system. As an addition a lot of efforts are made to improve the ODMR's performance. Furthermore, the system was selected by ESA to fly onboard the ISS for four months. Therefore, the system is developed around this mission, meaning it was built to fit the requirements laid upon the team by ESA.

At the core of the system is an STM32F76Z microcontroller. This chip runs a FREERTOS which is responsible for both the external communications as for managing each individual subsystem internally. Since the microcontroller is seen as the control subsystem, the other systems in the magnetometer are:

- the laser subsystem
- the microwave subsystem
- the ODMR readout subsystem
- the PDMR readout subsystem
- the reference subsystems (magnetometer, thermometer, accelerometer and gyroscope)

These subsystems can also be seen in Figure 9.

The OSCAR-QUBE has built-in internal storage built in the form of a 16GB SD-card. Here the gathered data from the system will be stored if the live connection with the UHB is lost. This live detection is detected by pinging the QUBE, meaning if the system does not receive a TC for ten consecutive seconds, the QUBE will think the live connection is lost and start saving the generated data on the SD-card. If the connection gets restored, the system will dump the stored data while continuing to perform the measurements.

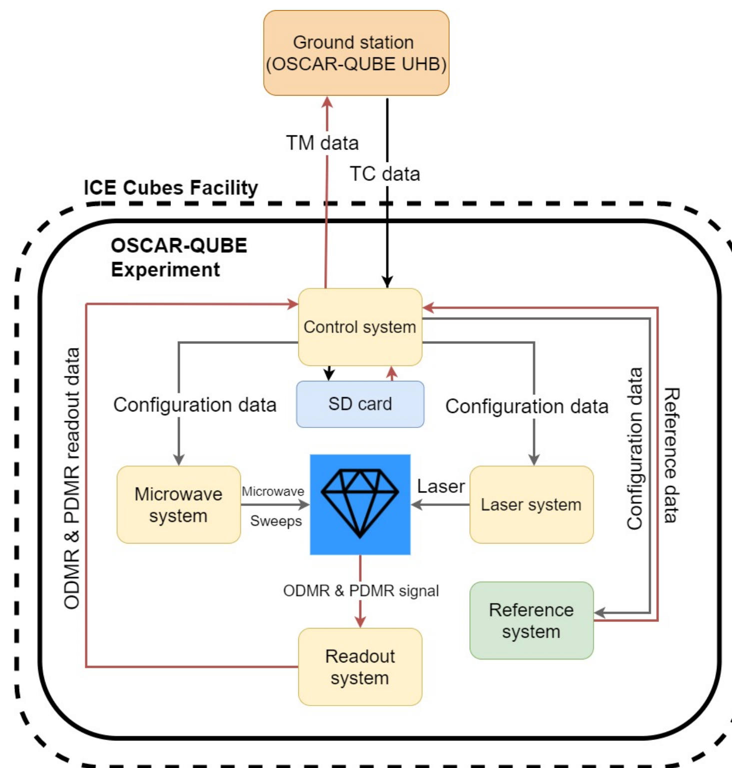

Figure 9: Schematic representation of the internal and external communication of the embedded system

The external communication with the QUBE utilizes an ethernet connection. Since it will be active within the Ice Cubes Facility (ICF) during its mission, only communication over ethernet is available to interface with the system. The lightweight IP (lwIP) stack was used to easily create the ethernet connection. It is configured to use a static IP address and use specific ports to send the TM and receive TC.

The diamond sample in the QUBE performed poorly for PDRM readouts, therefore the PDRM readout systems were disregarded during further development of the system. After which, the team fully focused on further optimizing and tailoring the system to fit the ODMR readouts. The TM packets were also restructured accordingly.

3.2 Mission Control Software

In order to create a UHB that would fit the requirements of the project, two existing mission control software were compared, while keeping into account the option of creating our own mission control software specifically written for this project. These software were Terma Ground Segment Suite (TGSS) and Yamcs. TGSS is an advanced mission control software created by Terma. It can be used for all phases of operations and is highly configurable. Terma also supports multiple types of pre-existing standards and protocols. The software seemed suited for the project, but could be overly complicated since it was designed to operate (multiple) satellites at once. Another downside to Terma was the requirement of a license in order to use the mission control software.

Yamcs on the other hand, is an open-source mission control software provided by Space Applications. This is the same company that will host the embedded system on the ISS. Therefore, the software is already more tailored to the requirements of the project. It did not support as many protocols as Terma, but it covered the most prominent protocols in the space industry such as CCSDS and PUS. Furthermore, it allows the user to define and use their own custom protocols. A downside to the Yamcs mission control software is the lack of quality of the documentation, but this was disregarded since a direct communication channel was available with Space Applications.

Table 2: Comparison between TGSS and Yamcs

Terma Ground Suite Segment	Yamcs
Advanced mission control software	Simple mission control software
License required	No license required
Proper documentation	Direct line of help to the developers
Integrated visualization	Visualization through Yamcs Studio
Modern and extensive visualization tools	• Yamcs: Graph of value over time • Yams Studio: Basic visualization tools
Focusses on operating (multiple) satellite(s)	Aims to support a single and basic external system.

Although Yamcs did not support a GUI other than a basic web interface, it did allow the UHB to use Yamcs Studio. This program allows the user to interface with the Yamcs mission control software and create custom GUI's which are called displays. Lastly, these two options were compared to creating a custom mission control software. This would enable the team to create a UHB without restrictions, but proved to be complicated as a proper mission control software has various different segments which have to be built into the system. Meaning the base of the system would be a bare bones representation of the previously mentioned mission control software. Out of these three options, Yamcs was selected due to the basic operation of the system, which leads to an easier learning curve of the software, and the direct connection with Space Applications. This connection could be useful if any impediments would occur while configuring or operating the system.

3.3 Utilized Protocols

The transport layer protocol selected to send the custom communication protocol over the internet with is the UDP protocol. Since the project is aimed to achieve a high data rate, UDP would be the best option. This is mainly due to the fact that UDP utilizes a 'fire and forget' methodology, meaning it will send the data and move on. The TCP protocol would be slower as it will send a packet and wait for an acknowledgement from the receiver upon its arrival. Since the project focusses mainly on the speed of data instead of the potential packet loss, UDP proved to be the preferred choice. As an addition the system will communicate using a dedicated VPN tunnel, meaning the packet loss will be insubstantial. Furthermore, utilizing the UDP protocol will prove to be more resilient to the effects during LOS where the connection cannot be guaranteed.

Following the selection of the transport layer protocol, the Consultative Committee for Space Data Systems (CCSDS) and the Packet Utilization Service version C (PUS C) protocols were investigated to serve as the application layer protocol, since these are widely used in the space industry. After studying both protocols, it was learned that PUS C was based on the CCSDS protocol. Yet both protocols were extensive and contained irrelevant data to the project, therefore it was chosen to create a custom protocol. This custom protocol would base its structure on the CCSDS procedures. However, it only utilizes the concepts that would be useful for the project, thus leading to a more efficient and smaller overall packet, resulting in a system that requires less bandwidth.

In order to create this protocol within the mission database (MDB) of Yamcs, the two methods of defining custom protocols were studied. The first method is the Spreadsheet loader. Using this method, a spreadsheet using a fixed layout and configuration is used to fabricate containers which hold either the telecommand (TC) or telemetry (TM) packet structures. The advantage of this method is that it is an accessible method of configuring the system, while the spreadsheet format leads to a clear overview of all the elements. The other method supported by Yamcs is the XML Telemetric and Command Exchange (XTCE) structure. Using this method, the TC and TM packet structures are defined following specific structure using XML. Both methods were studied and tested. Although proper documentation of XTCE was bulky and hard to find, XTCE was selected to configure the system. It proved to be substantially more efficient when compared to the spreadsheet method. The inefficiency of latter method came from the fact that it communicated using strings instead of an array of bytes. As an addition to this, the XTCE method enabled the user to configure the system and protocols with more detail since the method is a more low level version compared to the spreadsheet method.

3.4 Graphical User Interface (GUI)

The web interface provided by Yamcs was insufficient. It did not provide a proper overview and visualization while sending telecommands was an inefficient process. As an addition, constant pinging had to be provided by an external program, since the Yamcs interface did not have the capability to execute automated tasks. Therefore, a GUI had to be created meeting these requirements.

The first option to create this GUI was to use Yamcs Studio. This is a program provided by Space Applications that enables the user to create custom GUIs, called displays. Yamcs Studio has two sides: the display builder and the display runner. As the name suggests is one side to create and configure the displays. Custom JavaScript and Python scripts can be written and run within these displays to achieve the intended result. Here the layout could be created using the widgets provided by the program. The display runner is used to run these displays. At first this seemed the best option. It could easily connect to the Yamcs instance and was directly compatible with the system as is shown in Figure 10. Since it had a direct link with the Yamcs instance, configuring the displays was a very fluent process. However, while trying to implement multithreading some

problems occurred. When creating these threads using JavaScript, Yamcs Studio did not want to compile the code and gave an error message.

Therefore, the Python capabilities of Yamcs Studio were investigated. This would also prove to be helpful when integrating more complex algorithms into the system since some of these were already (partly) made by team members as tools during the development process of the system. Unfortunately, Yamcs studio operated using Jython instead of Python, making the process overly complex.

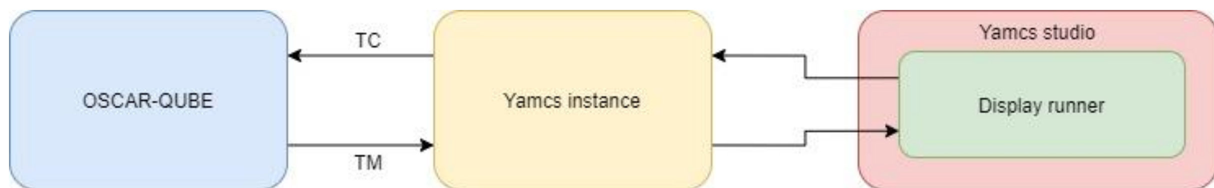

Figure 10: Schematic representation of interfacing between the GUI and the embedded system when using the Display runner from Yamcs Studio

Thus, the Yamcs API was investigated as an alternative. This API works effortlessly and interfaces to the Yamcs instance in a seemingly similar manner as Yamcs Studio as can be seen in Figure 11. However, the API only provides the ability to grab data from the instance or send telecommands. To create the layout and visualization capabilities, PyQt5 was selected. This is a Python library that is a translation of the Qt library which is written in C++. It enables the user to create responsive layouts and windows to interface with systems. PyQt allows the user to either manually write the code to define the layout or use the Qt Designer which is an external program. Using this program, a layout can be created using a drag and drop method. Here widgets can be dragged on their desired location and their size can be adjusted. Other settings can be altered in the settings window of the designer and even further modified in the code running behind the layout. After creating a layout using the designer, it will generate a .ui file. This file can be read by a Python program using the PyQt library making the widgets in the layout accessible to the rest of the script. Therefore, this method is an efficient way of creating a GUI. As an addition, PyQt comes with its own threading system, signal/slot mechanism to communicate between objects, and can be used cross-platform. [15]

Because of the high customizability and ease of development this method was selected to create the GUI of the UHB. Meaning a Python script would run the Yamcs API to interface with the Yamcs instance, perform basic conversions and formatting calculations and visualize the information in the appropriate PyQt widget. In the meantime, within the PyQt layout, several input widgets are used to trigger methods of the Python script which will properly format and send the TC to the Yamcs instance and thus to the QUBE.

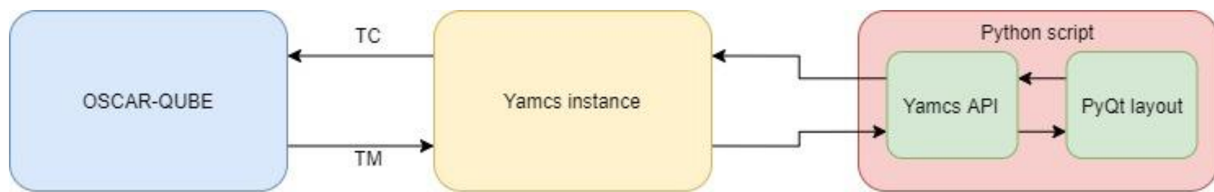

Figure 11: Schematic representation of interfacing between the GUI and the embedded system when using a python script as GUI

3.5 Wireshark

Wireshark is a network protocol analyzer. It enables the user to see all of the data packets with their details that are transferred over the network the computer running Wireshark is connected to. Various different types of communication channels are supported, such as Ethernet, Wi-Fi, Bluetooth etc. The software can give an overview of all the packets that are being sent in real-time, but also offers filtering features. Furthermore, a packet can be inspected in detail using Wireshark. Information about the specific packet such as destination, source, protocol, content etc. are displayed and can be found using the inspect functionalities. [16]

During the development of the UHB, Wireshark was used to validate the communication channels. At first it was being used to validate if the system's properties, such as IP address and ports, were configured correctly. Once these settings were set, it was used to create the TM link. It proved to be a helpful tool to confirm the packets were actually being received. Furthermore, this was used to check if the structure of the packets were altered upon being sent. When configuring the TC, Wireshark was used as a validation tool as well. However, the software was most useful during the initial stages of the communication channels and for inspecting the TM packets.

3.6 Nucleo-F746ZG board

A Nucleo board, seen in Figure 12, is a development board used to easily create prototypes of systems that utilize an STM32 microcontroller. The Nucleo- F746ZG board employs the STM32F746ZG chip as the brain of the board. It makes the STM32 accessible with all kinds of peripherals such as Arduino connector pins, integrated clocks, LEDs, buttons, ethernet port etc. Additionally, the Nucleo boards come with an ST-Link debugger/programmer. Therefore, separate probes are rendered unnecessary. Using the ST-Link capabilities, the system allows the user to easily debug the chip. Furthermore, it provides an integrated serial connection which can be used to communicate with the STM32 microcontroller. By adjusting the jumper configuration the integrated ST-Link of the Nucleo board can also support external systems and function as one of the ST-Links' probes itself. [17]

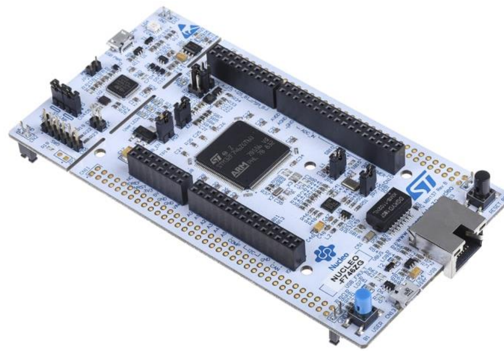

Figure 12: A Nucleo-F746ZG board [18]

The board was used to function as a dummy QUBE. It was connected directly over ethernet to the computer running the Yamcs instance. It was configured to have a static IP address, thus making the IP configuration of the computer simpler. Using this setup, the Nucleo board was able to send UDP packets over the ethernet port. The content of these packets could be tailored to the desired contents in order to test the connection and the responses of Yamcs on different kinds of inputs. This proved to be useful during the exploration part of the packet structures. Later the board was used to test the visualization and commanding aspect by sending data from the board to the computer to visualize. Commanding was validated by triggering an LED with a specific TC command.

Chapter 4: Experimental

Since the UHB is responsible for the communication with the QUBE which is onboard the ISS (Figure 13), it has to fulfill requirements of multiple different types. Therefore the UHB was designed to be split into two main segments. The first segment is the mission control software. After some considerations, Yamcs was selected to execute this role and function as the backbone of the UHB. It is designed to receive telemetry and send telecommands while storing the incoming packets and formatting the incoming raw values. These values can then be accessed by the second segment of the UHB which is the GUI. This GUI was chosen to be created as a Python script and is the main tool for the operator to interact with the system. Therefore it has to provide all the capabilities necessary to operate the QUBE while remaining intuitive for the operators. In this chapter the setup, creation and/or configuration of each part is explained in detail.

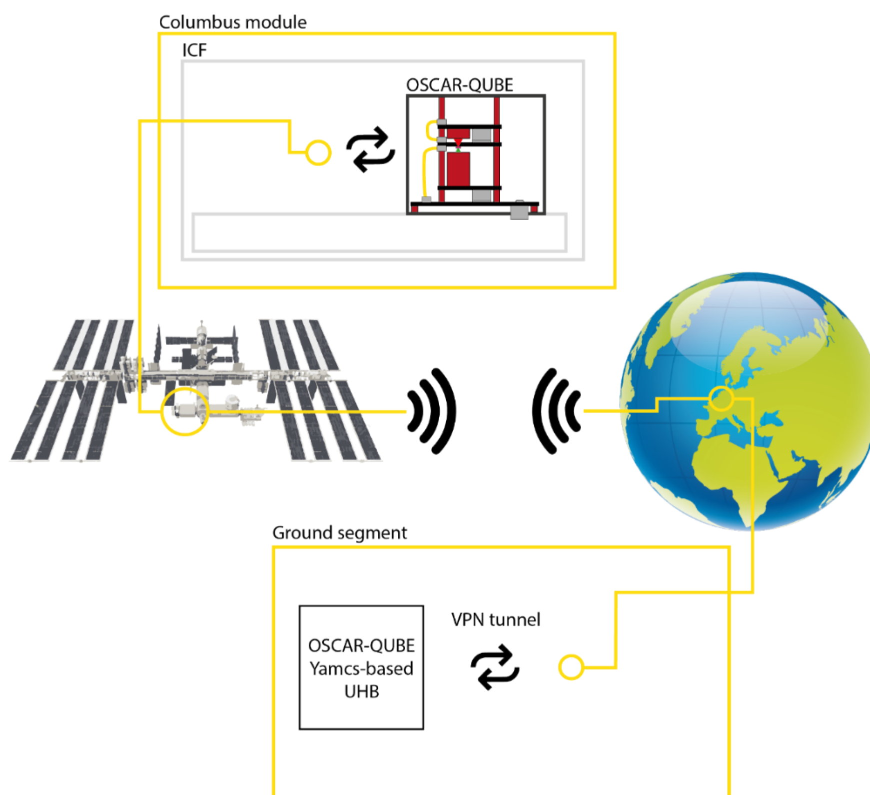

Figure 13: General overview of connection of UHB and QUBE

4.1 Installing Yamcs

The base of the system is the Yamcs mission control software. This software can be obtained by following the "Getting started" manual on the Yamcs website. The prerequisites mentioned here are:

- Java 11
- Maven
- Linux x64 or macOS

However, the last requirement seemed outdated since it was possible to run Yamcs on Windows 10 as well. For development purposes, Yamcs was also installed on a device running Ubuntu 18.04 LTS. However, in both cases the same steps were followed to install the software:

- install Java 11 openJDK
- install maven
- add maven to path
- clone the Yamcs repository from GitHub

Now Yamcs was installed on the computer with a default setup which could be started by executing the following command in Yamcs' base map:

mvn yamcs:run

After a familiarization process with the both the software and interfaces, the following steps were taken in order to receive and format incoming TM packets:

- Instance renamed to OscarQube in the yamcs.yaml file
- Communication settings reconfigured to UDP communication in the yamcs.OscarQube.yaml file (Figure 14)

```
#config the tm providers started by the TmProviderAdapter
dataLinks:
# Cube TM
- class: org.yamcs.tctm.UdpTmDataLink
  port: 10016
  name: cube_tm
  spec: cube-testharness
  stream: tm_realtime
  packetPreprocessorClassName: com.spaceapplications.icecubes.MyPacketPreprocessor

# Cube TM dump
- name: cube_tm_dump
  class: com.spaceapplications.icecubes.FilePoolingIcuTmDatalink
  incomingDir: /storage/yamcs-incoming/cube/tm
  stream: tm_dump


# Cube TC uplinkers
- name: cube_tc
  class: org.yamcs.tctm.UdpTcDataLink
  port:
  host:
  spec: cube-testharness
  stream: tc_realtime
```

Figure 14: Communication and IP configuration

- IP settings changed to match the QUBE's settings
- Add spreadsheet loader method to MDB in de mdb.yaml file (Figure 15)

```
cube-mdb:
    - type: "xtce"
      spec: "mdb/xtce.xml"
    - type: "sheet"
      spec: "mdb/icu-tmtc-testharness.xls"
```

Figure 15: MDB configuration

Using these settings, Yamcs was able to receive TM packets from the QUBE. However, on the setup running on Windows, the firewall had to be disabled first as it blocked the packets.

4.2 Packet configuration

As Yamcs was properly configured to set up a communication link with the QUBE, the packet structures were created. This was critical since Yamcs required a standardized way of reading, interpreting and assigning incoming data to specific variables. Additionally, TC also required specific structuring in order to operate the commanding aspect of the system.

4.2.1 Telecommanding packets

The QUBE's commands are of the string type. Therefore, only one telecommand was created that could send any given string to the embedded system. For this, the spreadsheet loader was utilized as it provided the best overview of the structures. First the datatype had to be fabricated, this was done on the "Datatypes" tab as seen in Table 3.

Table 3: Fabrication of sendStringCmd datatype

Type name	Eng type	Raw type	Encoding	Eng unit	Calibration
sendStringCmd	string	TerminatedString(0x0D)			

Now the datatype could be used to structure the TC packet on the "Commands" tab. This was achieved with the configuration seen in Table 4.

Table 4: TC "Commands" tab spreadsheet configuration

Command name	Argument assignment	Argument name	Data type	Default value
TC_CODE_SEND_STRING	packet_type=0xF6	Send String	sendStringCmd	0

4.2.2 Telemetry packets

XTCE was selected as the preferred method to structure the packets. This is due to the fact that the structure of the TM data is more complex when compared to the TC packets. Additionally, these datatypes required more custom tailoring then the datatype used for the commanding. XTCE requires three steps to be taken before a packet is composed as seen in Figure 16.

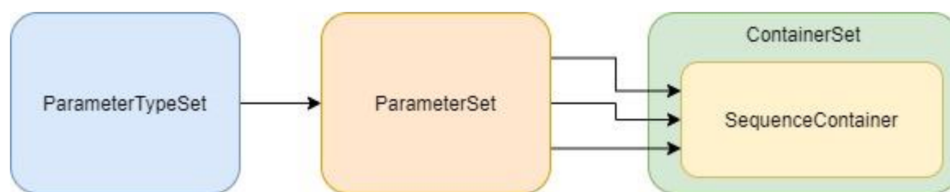

Figure 16: The correlation between the three stages within the XTCE structure

First the datatypes had to be defined in the ParametertypeSet. These could then be used to create parameters in the ParameterSet. In the ContainerSet, a SequenceContainer can be composed out of these parameters to structure the packets. Thus, each sequence container represented a different packet type.

In the ParameterTypeSet the types seen in Table 5 are defined. As can be seen, the byte order of the integer types had to be inverted. This was due to the fact that the system utilized the little-endian method to send these values to the UHB.

Table 5: Parametertypes and the appropriate configurations to support TM of the QUBE

Parametertype	encoding	Byte order
byte	Unsigned	/
uint16_t	Unsigned	Least significant byte first
int16_t	Twos complement	Least significant byte first
uint32_t	Unsigned	Least significant byte first
string	Termination character = '/n'	Unaltered

Additionally, the arrays had to be defined in the ParameterTypeSet. Since the QUBE did send different kinds of arrays, each had to be defined with their specific parametertype. Here it was opted to define every array parameter as their own datatype to mitigate the risks of causing mix ups.

Table 6: Defined arrays and their corresponding parametertypes

Array	Parametertype within the array
Measurement data	uint16_t
MW registers	uint32_t
Laser temperatures	uint16_t
Accelerometer and gyroscope settings	byte
Magnetometer settings	byte

With all of the required parametertypes defined, the parameters themselves were created in the ParameterSet. After this, two sequence containers, which describe the packet structure, were composed. These containers are:

- the measurement container, containing all the measurement and config data of the QUBE
- the debug container, which consisted only out of a single string

In order to differentiate between these two packets, a header packet was defined containing an APID. This header would be at the start of every sequence container and will aid Yamcs into selecting the correct sequence container to format the rest of the data with. The three different packet types and their corresponding APIDs are shown in Table 7.

Table 7: APID with corresponding packet type

APID	Packet type
0	Measurement + config data
1	Debug messages
Not applicable	Telecommands

The ID would serve as the first value read by the system. Based on this value, Yamcs will be able to identify the packet type and what sequence container it has to deploy to extract the rest of the data out of the of the packet. The full process is displayed in Figure 17.

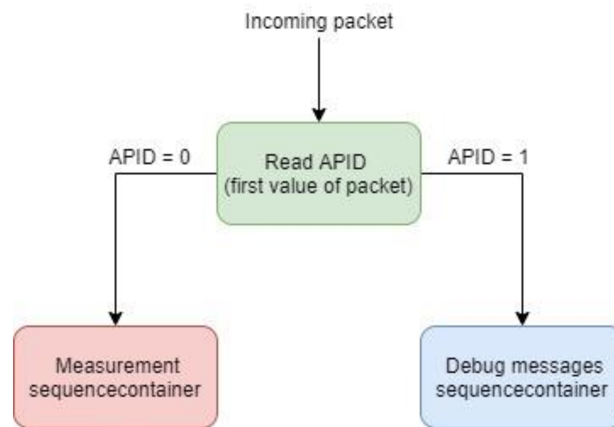

Figure 17: Packet differentiation process based on APID

Lastly padding bytes were added to the measurement sequence container as the QUBE aligned the structs when the data is sent the UHB.

4.3 User Interface

With Yamcs configured and able to receive data from the QUBE while commanding it through the web interface of Yamcs, a different more user-friendly option was explored to interface with the embedded system. Most of the people operating the system and performing experiment were scientists with little to no technological background. Therefore, a GUI was created which enabled the user to easily and efficiently command the QUBE while providing a clear overview of the incoming data.

4.3.1 PyQt5 and Designer

At the base of the GUI PyQt5 is located. This Python library was installed using the following pip command:

pip install PyQt5

To aid the designing process of the layout, the PyQt Designer software was installed as well. The following command were used to get the software:

sudo apt-get install qttools5-dev-tools
sudo apt-get intstall qttools-dev

4.3.2 Layout

The layout of the GUI was split into three tabs, each with their specific purpose. These tabs are:

- Control tab
- Monitor tab
- Peak & Pulsed tab

Control tab

The Control tab is responsible for:

- the general controls of the QUBE
- visualization of the live ODMR data
- visualization of the ODMR average over time
- visualization of the debug responses
- visualization of the link information

The layout of these items can be seen in Figure 18.

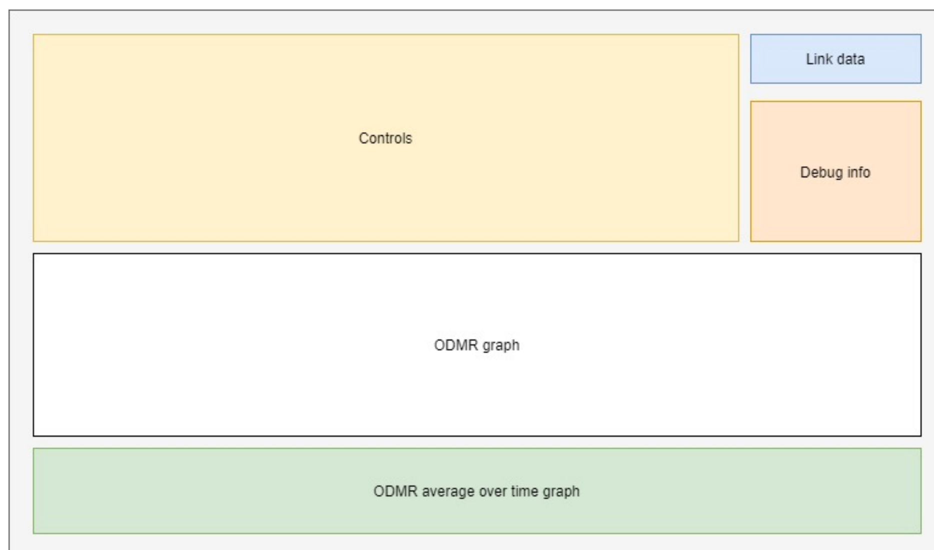

Figure 18: General layout Control tab

The control section was subdivided into:

- system controls
- laser controls
- microwave controls
- FPGA controls

These control groups follow the layout seen in Figure 19.

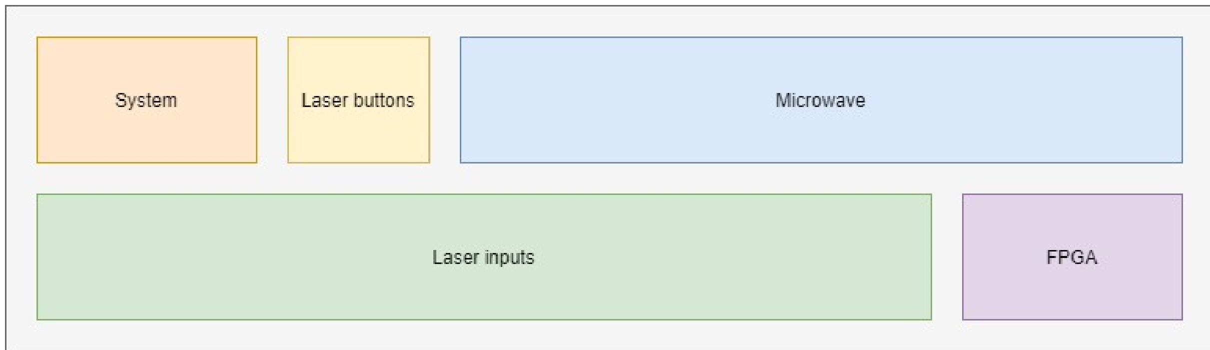

Figure 19: General layout controls

Monitor tab

On the Monitor tab the environment monitors and export capabilities can be found. Here the parameters gathered by the reference sensors will be displayed. The parameters which are displayed are:

- laser temperature
- board temperature
- X-, Y-, Z-axis of the magnetometer
- X-, Y-, Z-axis of the gyroscope
- X-, Y-, Z-axis of the accelerometer

The general layout of this tab is shown in Figure 20Figure 20.Figure 20

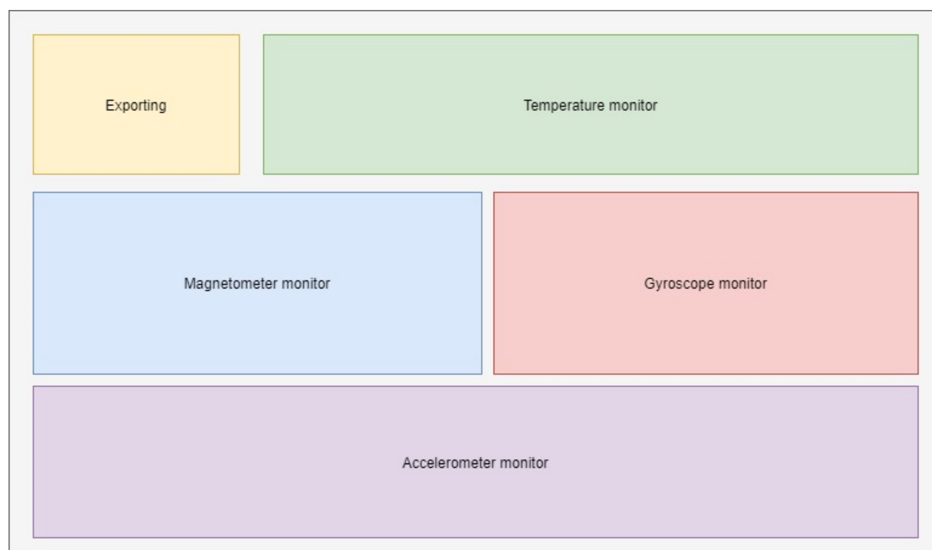

Figure 20: General layout Monitor tab

Peak & Pulsed

The Peak & Pulsed tab is reserved for the configuration of the Peak and Pulsed commanding. These commands are not located on the "Control" tab since they are more complex and require extra support to properly integrate in the GUI. The tab is horizontally split into a peak commanding part and a pulsed commanding part as can be seen in Figure 21.

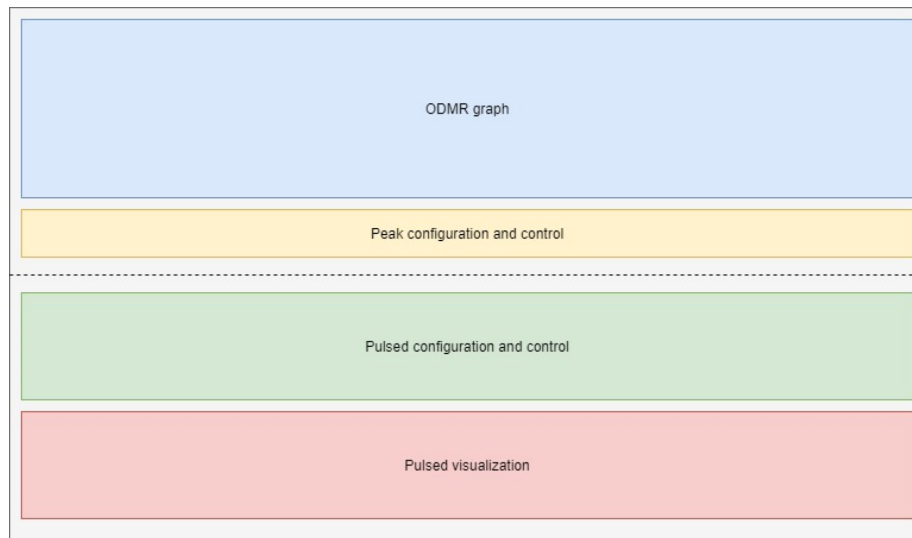

Figure 21: General layout Peak and Pulsed tab

The top part, the peak part, holds an ODMR graph which is the same as the ODMR graph found on the Control tab. It is used to find the parameters necessary to configure the peak configuration. Since constantly switching tabs would be detrimental to the operator's user experience, it was chosen to add the ODMR graph to this tab as well.

4.3.3 Yamcs-client API

To interface with the Yamcs instance, which functions as the MSC, the Yamcs-client API of Python is used in the script. The command to install the library is:

pip install –upgrade yamcs-client

Using this library in the Python script of the GUI, the credentials of the Yamcs instance have to be given to enable the connection between the Yamcs-client API and the Yamcs instance. Upon its initialization the processor and archive are taken and saved as variables so they can be used later by the rest of the script.

Commanding

All of the commanding in the GUI is performed by clicking the corresponding button. These buttons internally will trigger the appropriate responses which send the correct telecommand. There are two types of buttons within the layout of the UHB:

- Action buttons

This type of button can be split into two different types of their own. The first type does not require input data and therefore the system can directly send the telecommand. The second type requires an input. Hence this input will first require to be read and formatted by the algorithm before this can be sent. These processes are visualized in Figure 22

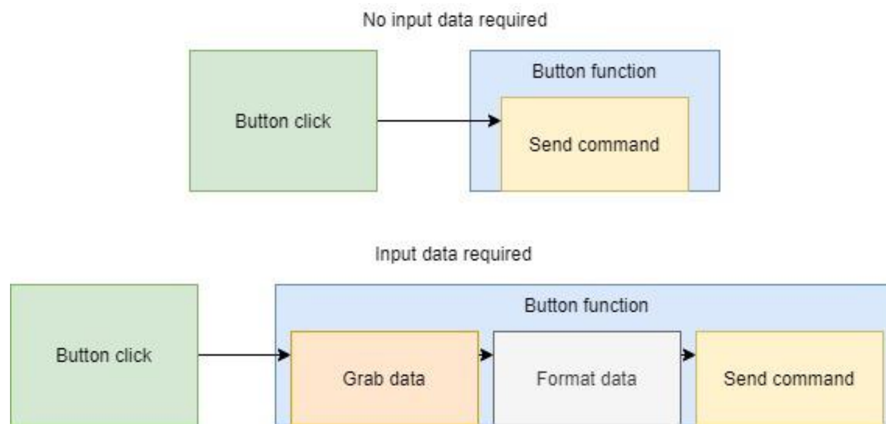

Figure 22: Action button telecommanding flow

- Radio-buttons

Radio-buttons are bound to the other radio-buttons in the group. These are used for commanding the modes since only one can be active at once. Furthermore, each button holds a "command" variable which can be used to easily send the appropriate command specific to each radio-button. The processes concerning these buttons are shown in Figure 23

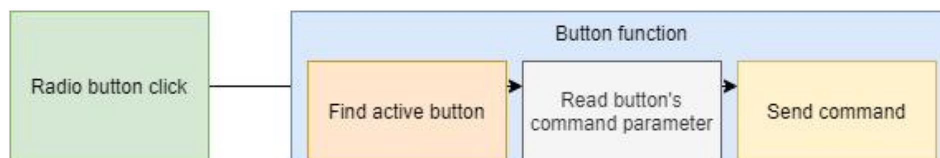

Figure 23: Radio button telecommanding flow

Since all of the telecommands are of the same command type, namely the "Send string" command, all of the functions eventually end up triggering the same method. This method is responsible for interfacing with Yamcs using the client API. Furthermore, all of the commands sent by this function (except for the ping command) will be printed on the GUI's console. This way a history of the commands is kept. Additionally, this can be used by the operator to validate the formatted command string.

4.3.4 Threading

The GUI uses multithreading to perform all of its tasks. This is mainly due to the fact that the main thread, called the GUI thread which is run by PyQt, will render the application "unresponding" if it is not able to read all of the input widgets within a spefic timeframe. Therefore this thread should only be used to interface with the widgets of the layout. The thread is able to support minor calculations, however more substantial processes should be executed somewhere else.

The GUI utilizes four threads, which are:

- the GUI thread
- the Iinterface thread
- the TC thread
- the Ping thread

How these threads correlate to eachother can be seen in Figure 24.

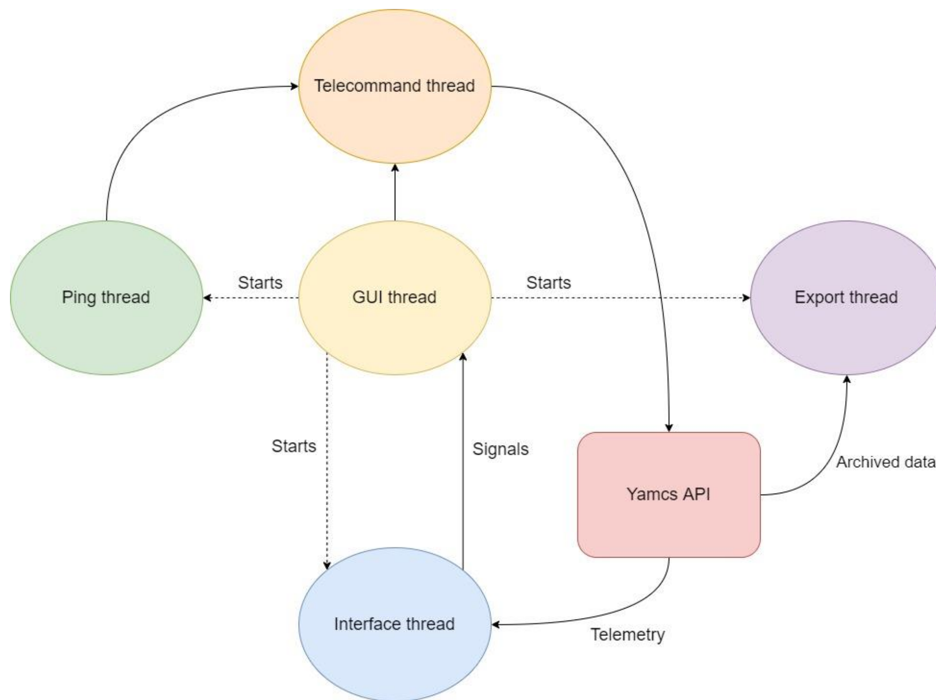

Figure 24: Threading scheme

Ping thread

The ping thread is responsible for the constant pinging of the QUBE. It is crucial that this task gets executed in time and unrelated to other processes. This is due to the fact that the system will decide if it should save its data on the SD-card or send the data directly to the UHB based on these pings. Using a thread therefore meets all these requirements. Additionally, if the GUI thread becomes unresponsive the Ping thread will continue operating, thus the QUBE will not lose the constant pinging from the UHB in this case. The process executed by the thread is rather simple as can be seen in Figure 25.

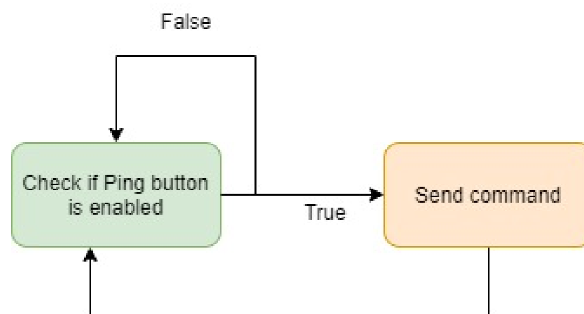

Figure 25: Process of Ping thread

This process runs once every four seconds. Due to the fact that the QUBE will only start saving data to the SD-card if there was no ping detected within a timeframe of 10 seconds, losing one ping packet does not pose a problem.

Telecommand thread

Unlike the other threads, this one does not loop. It is responsible of sending the TCs and functions as a buffer between the Yamcs client API and the GUI thread. If the client API takes longer than expected it will not freeze the GUI thread since it is running independently. Once the command is sent the thread will stop its execution automatically.

Interface thread

This thread constantly uses the Yamcs client API to get the incoming TM. Upon receiving a new packet, the packet will be checked to see if it was sent live or came from the internal SD-card of the QUBE. After which all of the desired parameters are extracted. These parameters are:

- time of receival of the packet
- ODMR data
- board temperature
- laser temperature
- X-, Y-, Z-axis data of the magnetometer
- X-, Y-, Z-axis data of the gyroscope
- X-, Y-, Z-axis data of the accelerometer

Furthermore, the thread computes the average of the ODMR array and utilizes the Yamcs client API to calculate the packet speed at which the UHB receives the TM.

Upon extracting the data from the packet, it is formatted and sent to the GUI thread using pyqtSignals. Here these signals trigger a method bound to their specific signal, which will make sure the value gets displayed. The complete process executed by the Interface thread can be seen in Figure 26.

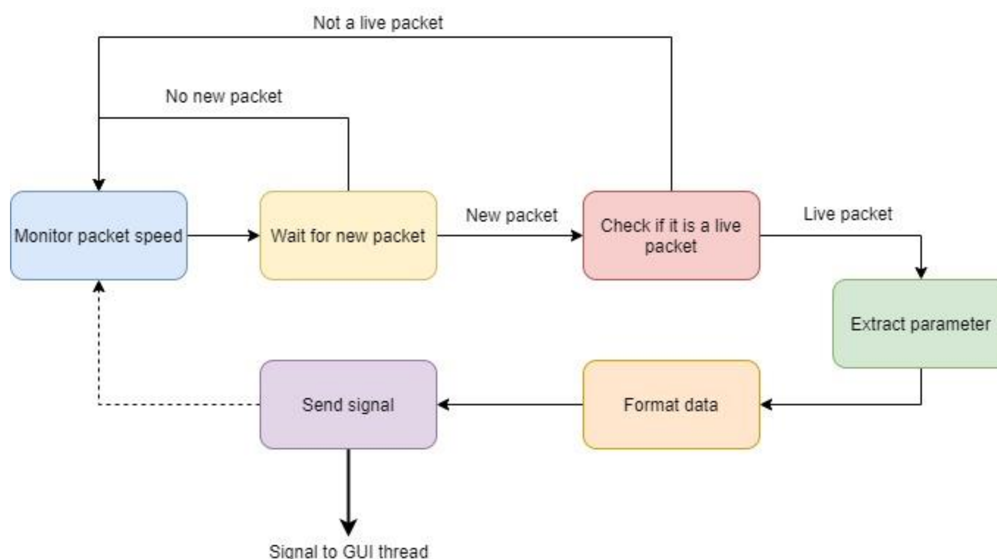

Figure 26: Process flow interface thread

4.3.5 TM data visualization

Most of the TM data is visualized using a PyQtGraph widget. This widget is not installed by default, therefore it has to be downloaded separately. This was achieved by using the following command:

pip install pyqtgraph

To use the PyQtGraph in the Designer a blank widget has to be promoted to a PyQtGraph. This can easily be done following the next steps:

- Drag a QWidget in the layout
- Right click and select "Promote to …"
- In the pop-up screen specify the class name and header file as shown in Figure 27.

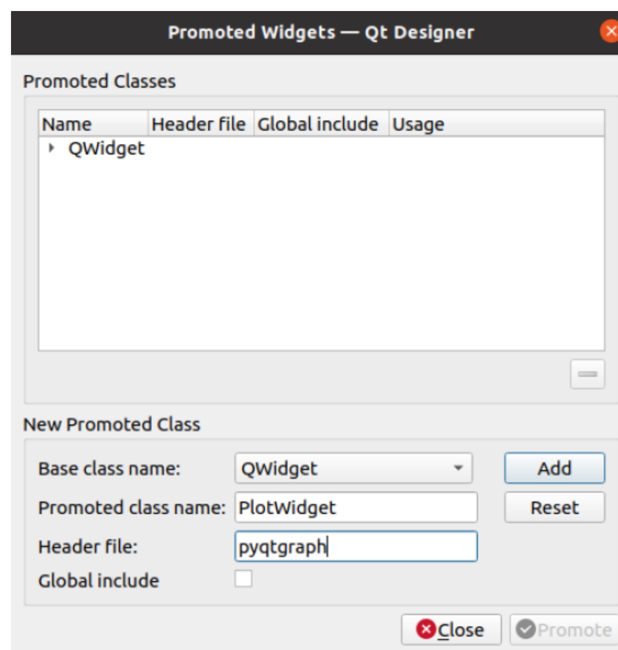

Figure 27: Convert QWidget to PyQtGraph

- Now the widget can be promoted

After these steps the widget can be used as a graph in the Python script. Upon initialization of the GUI an empty array will be plotted on the graphs to initialize the widget and grab the dataline of each graph. This data-line is necessary to update the plot later in the process. Once the Interface thread starts running the signals will be received by the GUI thread and trigger the methods that will update the corresponding graphs. Here two types can be differentiated:

- the graphs that show the evolution over time
- the graphs that show an array of live data

The graphs displaying live data will visualize an array containing the data for the whole graph with each update. To visualize the parameters over time a more complex approach was taken. Each parameter falling under this category will be appended to an array. However, to prevent loss of performance over time, these arrays will eventually stop at a specific length. The process of this can be seen in Figure 28.

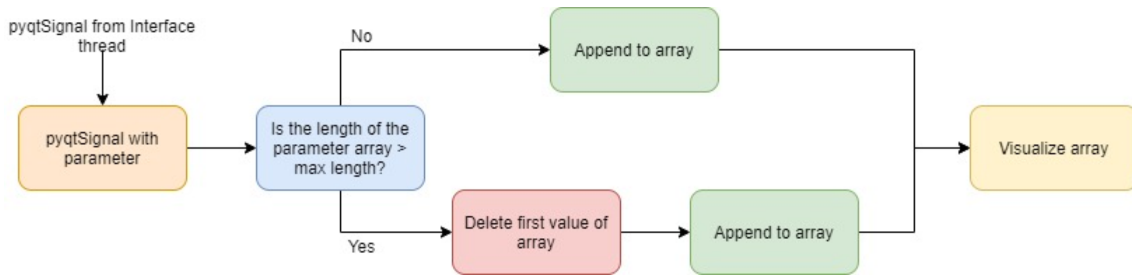

Figure 28: Visualization process of monitor parameters

Since these plots show the datetime on their X-axis the same process applies to this as well. Upon receiving the timestamp of the packet, it will be added to array. When another parameter wants to visualize their data, they grab the timestamp array as their X-axis. However, some arrays visualize less datapoints at once. In this case, only the last n points of the timestamp array will be used, where n is the length of the parameter array.

4.3.6 Pulsed visualization

The pulsed visualization plot is the only PyQtGraph that does not follow the same processes as the other graphs. This is mainly due to the fact that this plot does not display live data, but visualizes the operators current configuration disregarding the fact if this is active on the QUBE or not. Therefore, the operator can validate his configuration by going through all of the 100 pulsed steps.

This is achieved by reading the two rows of the table and compare the values of each column with each other. These two rows represent the first two steps of the 100-step pulsed operation. To visualize the other steps the difference is taken between the values in each column. The current setting is then calculated following the next formula:

*Current value = first row value + difference * selected step*

The visualization arrays are fabricated based on a custom unit step function. This function uses the delay and width parameters of each subsystem as a base for their calculations. The step function will only output 1 after the specified delay and for the given length of the width. All of the other values are set to 0 as can be seen in Figure 29.

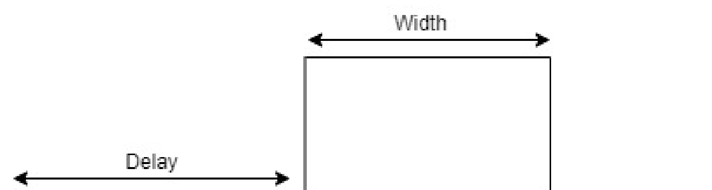

Figure 29: Unit step function

The complete process of visualization is shown in Figure 30.

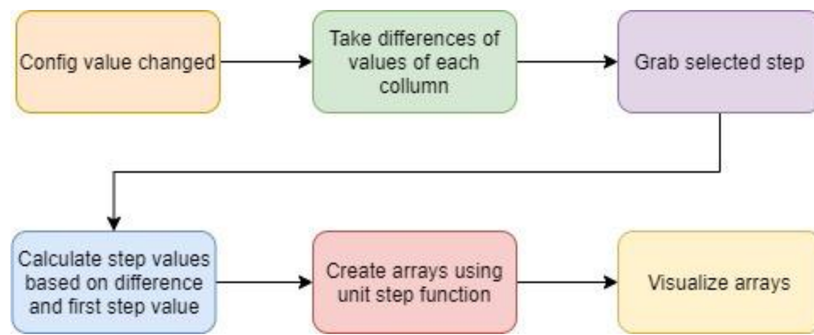

Figure 30: Visualization process of pulsed data

4.3.7 Graph Zeroing

Since the base values of every axis is substantially spread out on the graphs not a lot of detail can be seen in the data. Therefore, a zeroing capability was added for the graphs of the magnetometer, gyroscope, and magnetometer. With this capability, all of the base values are reduced to zero. Thus, the range of the Y-axis is more condensed. This leads to more visual details in the data of every axis. The procedure is shown in Figure 31.

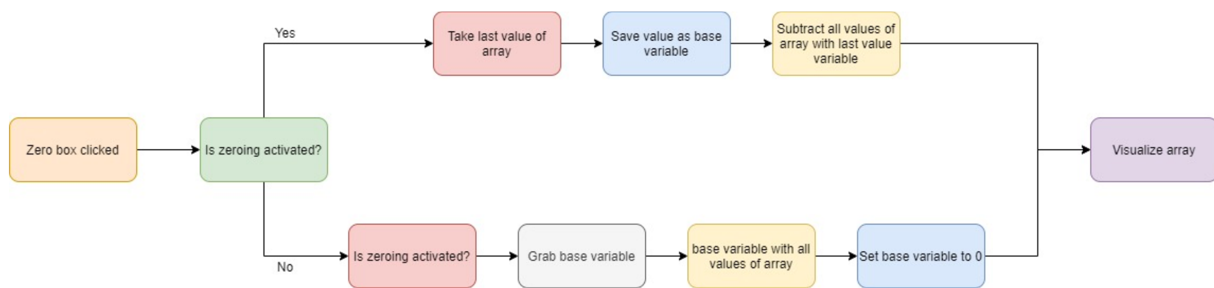

Figure 31: Zeroing procedure

The base values are then used to adjust new values coming in live from the QUBE. Therefore, the array does not continuously need to loop through this procedure. This will render the program more efficiently, which will benefit the performance of the GUI thread in which these procedures are located.

4.3.8 Exporting

Since is critical for the scientific goals of the project to analyze the data, it is essential that the UHB is capable of exporting the data for further analysis. To solve this problem, the Yamcs client API was utilized. Using the archive element of the API, stored data can be accessed. Additionally, the intended packet type was set to measurement data in order to automatically filter out debug packets. Furthermore, the timeframe of the exported packets can be set.

The layout, seen in Figure 32, of the export group was designed in such a way that exporting large timeframes was as easy as performing small and quick measurements.

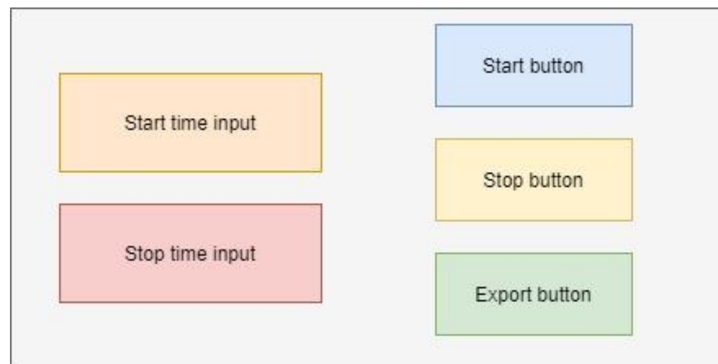

Figure 32: Layout export group

The time inputs are datetime input boxes which allow an easy way of manually selecting the datetimes. On the other hand, the operator can choose to use the start/stop button. Upon clicking the button, the appropriate datetime input box will be set to the current time. This allows to easily set the timeframe of the measurement while performing measurements. Upon clicking the export button, the export thread will start.

Chapter 5: Results

5.1 Back end

The final UHB was run on a Windows PC. Here the MCS was Yamcs, which was configured to communicate with the QUBE over ethernet using UDP packets. The TC packet structure was defined using the spreadsheet method while the TM packet structures were created using the XTCE method. Here the custom datatypes and arrays were defined in the "ParameterTypeSet", which can be seen in Figure 33.

```xml
<ParameterTypeSet>
    <IntegerParameterType name ="byte" signed="false">
        <UnitSet />
        <IntegerDataEncoding  encoding="unsigned" sizeInBits="8" />
    </IntegerParameterType>

    <IntegerParameterType name="uint16_t" signed="false">
        <UnitSet/>
        <IntegerDataEncoding encoding="unsigned" sizeInBits="16" byteOrder="leastSignificantByteFirst"/>
    </IntegerParameterType>

    <IntegerParameterType name="int16_t" signed="true">
        <UnitSet/>
        <IntegerDataEncoding  encoding="twosComplement" sizeInBits="16" byteOrder="leastSignificantByteFirst"/>
    </IntegerParameterType>

    <IntegerParameterType name="uint32_t" signed="false">
        <UnitSet/>
        <IntegerDataEncoding  encoding="unsigned" sizeInBits="32" byteOrder="leastSignificantByteFirst"/>
    </IntegerParameterType>

    <StringParameterType name="string">
        <UnitSet />
        <StringDataEncoding>
            <SizeInBits>
                <TerminationChar>0A</TerminationChar>
            </SizeInBits>
        </StringDataEncoding>
    </StringParameterType>

    <ArrayParameterType name="array" arrayTypeRef="uint16_t" numberOfDimensions="1"/>
    <ArrayParameterType name="MW_reg_array" arrayTypeRef="uint32_t" numberOfDimensions="1"/>
    <ArrayParameterType name="Laser_temp_array" arrayTypeRef="uint16_t" numberOfDimensions="1"/>

    <ArrayParameterType name="AccelGyroSettings_array" arrayTypeRef="byte" numberOfDimensions="1"/>
    <ArrayParameterType name="MagnetoSettings_array" arrayTypeRef="byte" numberOfDimensions="1"/>

</ParameterTypeSet>
```

Figure 33: Final ParameterTypeSet definition

Additionally, the byte order of the integer values were defined here as well. Furthermore, a string datatype was created to support incoming strings. These are used by the QUBE as debug messages. Here the end character is defined as "0A" which represents the hexadecimal value of 0x0A. This value stands for the "new line" character or "\n". Once Yamcs detects this value for a parameter using this parametertype, it is at the end of the parameter.

In the ParameterTypeSet the arrays are defined as well. All of the arrays are one dimensional and their corresponding datatypes are assigned at this points as well.

Using these parametertypes, the parameters themselves were defined. The final definitions can be seen in detail in Appendix A. These parameters were then used to compose the packet structures. Only two packet structures were created:

- a debug packet
- a measurement packet

However, to differentiate between the packets, a header was made in the form of an abstract Sequencecontainer as is shown in Figure 34 and only consisted out of an APID parameter. It functions as a packet identifier as this header is used in the other sequence container as the base container. For this the restriction criteria were utilized as a way of assigning packet structures to specific APID values.

```
<SequenceContainer name="Header" abstract="true">
    <EntryList>
        <ParameterRefEntry parameterRef="apid"/>
    </EntryList>
</SequenceContainer>
```

Figure 34: Header container definition

The simplest sequence container, which is seen in Figure 35, was the debug packet's container. This only consists of the header and a string datatype. This was due to the fact that the QUBE will send these packets as a response to being commanded. Therefore, not much complexity was necessary in this packet.

```
<SequenceContainer name="Debug">
        <EntryList>
            <ParameterRefEntry parameterRef="Debug_String" />
        </EntryList>
        <BaseContainer containerRef="Header">
                <RestrictionCriteria>
                    <ComparisonList>
                        <Comparison parameterRef="apid" value="1"/>
                    </ComparisonList>
                </RestrictionCriteria>
            </BaseContainer>
    </SequenceContainer>
```

Figure 35: Debug packet structure definition

The measurement container was a more substantial packet as can be seen in Appendix B. However, the general structure is as follows:

<Application ID><Measurement data><Reference data><Laser info><Microwave info><Qube info>

Both "Laser info" and "Microwave info" contained certain configuration settings of the subsystems which would be useful for postprocessing of the data. Furthermore, "Qube info" holds more information specific to the operations and state of the general system.

5.2 Front end

The frontend of the UHB is started by executing the UHB.py python script. This script will launch the PyQt layout based on the .ui file and initiate all the processes and variables required for operation of the GUI. The layout created for the UHB consist out of three tabs:

- the Control tab
- the Monitor tab
- the Peaks and Pulses tab

These tabs provide a clear overview and controls of the tabs' main purpose.

5.2.1 Control tab results

The first tab's, the "Control" tab houses the main controls of the system, a can be seen in Figure 36. Here, the main configurations can be altered and essential information is displayed. This tab is subdivided into groups, each group stands for a specific task or subsystem.

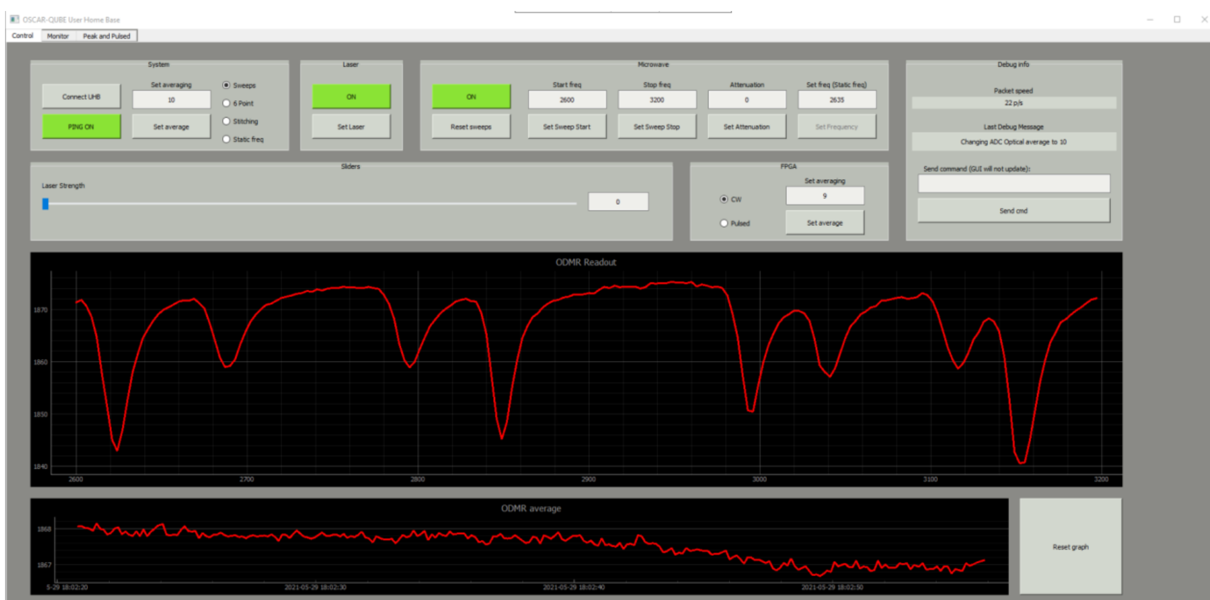

Figure 36: Control tab

System group

The first group, in the top left corner, is the System group. The tasks it is able to perform are:

- connecting the GUI with the Yamcs instance
- toggle pinging of the QUBE
- controlling the averaging of the data before the QUBE sends the packet
- selecting the measurement operational modes

The system group can be seen in detail on Figure 37.

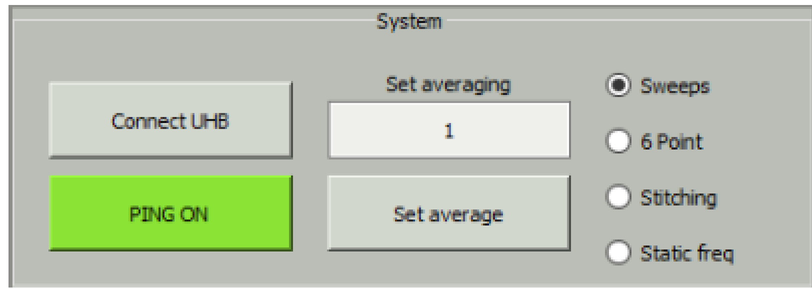

Figure 37: Detail of system group

Upon connecting the GUI to the UHB, it will automatically match the layout to the settings of the QUBE based on the information coming from the TM. Therefore, no mismatch will exist between the displayed data and the actual settings on the embedded system, thus mitigating the risks of causing confusion and potential errors.

Laser group

The laser group is split into two different group boxes as this fit the layout much better. The first box (Figure 38) is a smaller box containing the on/off button and houses the button to set the laser strength.

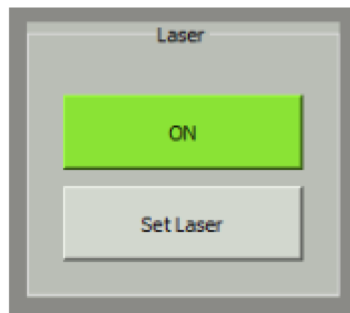

Figure 38: Commanding box of laser group

As all of the colored buttons, the state shown in text is the current state. Upon toggling the buttons to the OFF state, both the text and the background color will change. These toggle buttons display an OFF state by showing a red background as an addition to the text.

The other box containing a part of the laser group is called the "sliders" group. Here a slider can be found with an input box next to it, as can be seen in Figure 39.

Figure 39: Laser strength selector group

Both widgets are linked with each other, meaning if the value of one is altered, the other widget will copy this value. By linking the input box to the slider, the boundaries of the input value are automatically enforced.

Microwave group

The microwave settings can be adjusted with the controls in this group. The group contains the following commanding possibilities:

- toggle Microwave on and off
- set start frequency of the microwave sweeps
- set stop frequency of the microwave sweeps
- set attenuation of the microwave generator
- reset sweep start and stop frequencies to default values
- set static frequency (static frequency operational mode)

As can be seen in Figure 40, the "Set Frequency" button is disabled. This is due to the fact that the system is not set to the Static Frequency operational mode. Once the mode is activated in the system group, the button will be enabled and the command can be sent.

Figure 40: Detail of microwave group

FPGA group

The FPGA group (Figure 41) seems rather simple, however it is critical for the pulsed operation of the QUBE. This is due to the fact that the group houses the radio buttons to control this type of operational modes. Additionally, the averaging for these modes can be controlled as well since this setting is independent from the averaging of the "System" group.

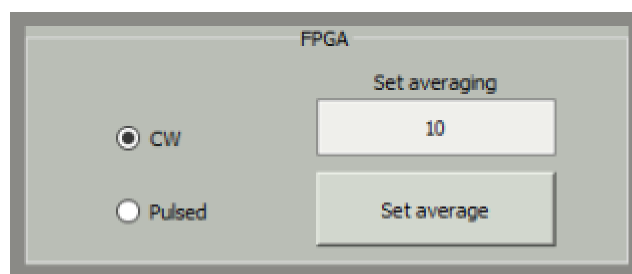

Figure 41: Detail of FPGA group

Debug group

This group (Figure 42) displays both the live packet speed and the last debug message received. However, the GUI filters out the responses from ping commands as they do not provide any relevant information. Therefore, only the last debug message stemming from an actual telecommand is shown as a way of confirming the TC's proper arrival.

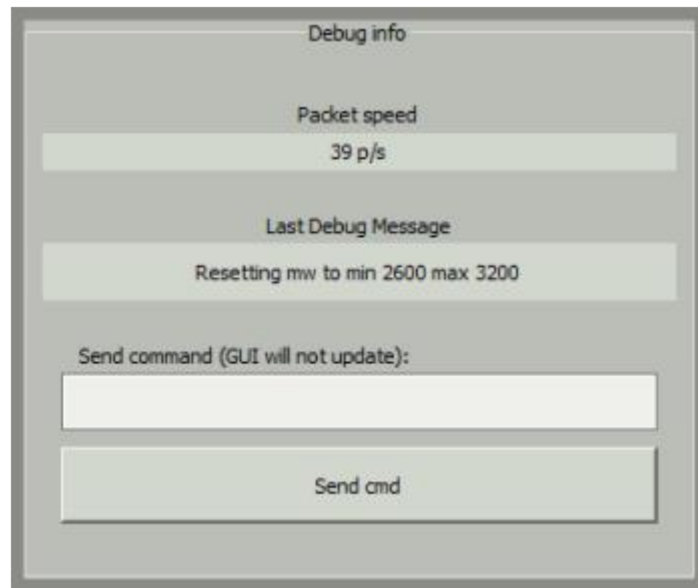

Figure 42: Detail of Debug group

Additionally, manual commands can be sent using the "Send command" input box and the corresponding "Send cmd" button. However, it is mainly reserved for admin level commands. These commands have a significant impact on the system and are not to be used lightly, therefore only a person knowing the exact format of these commands is able to send it using this attribute. The other commands which are integrated in the GUI can be sent as well using this feature, but the layout will not update accordingly. Thus, it is preferred to only use the (radio)buttons to send telecommands to the QUBE.

ODMR readout graph

The ODMR readout graph, seen in Figure 43, displays the live formatted ODMR data collected by the system. This graph is essential to validate if the system is in the correct configuration. On the X-axis the frequency range is displayed. Upon sending the "Set start frequency" or "Set stop frequency" commands using the buttons in the "Microwave" group, the frequency array is recalculated as the system will always send 200 ODMR datapoints to the UHB. Therefore, to properly be able to visually interpret the data, the X-axis is adjusted.

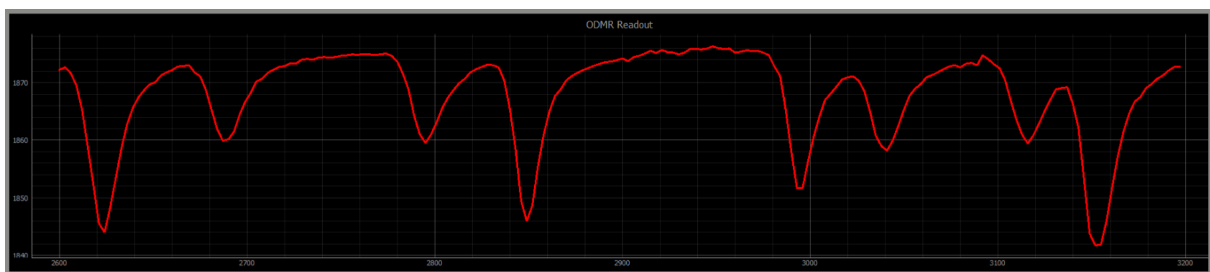

Figure 43: Detail of ODMR readout graph

ODMR average graph

Below the ODRM readout graph is the ODMR average plot. This graph, shown in Figure 44, displays the average value of the ODMR datapoints over time. Upon receiving a new ODMR measurement array, the average is calculated and added to the graph with the corresponding timestamp. This graph is mainly used to monitor the operation of the QUBE and validate settings, but can also be used scientifically.

Next to the graph, a button is placed. This button will clear the plot and reinitialize the arrays. Therefore, if any configuration changes are executed the transition can be seen in the plot. Nevertheless, if the operator would like to study the average within the new configuration, he can reset the graph to directly see only the data regarding the new settings in detail.

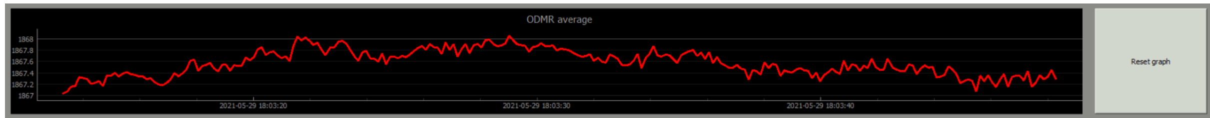

Figure 44: Detail of ODMR average graph + clear button

5.2.2 Monitor tab results

The monitor tab, seen in Figure 45, houses the export functionality and all of the plots concerning the monitoring of the environment of QUBE. It displays:

- laser temperature
- board temperature
- X-, Y-, Z-axis of the magnetometer
- X-, Y-, Z-axis of the gyroscope
- X-, Y-, Z-axis of the accelerometer

Each readout type is grouped and is shown in the same graph. Therefore, only five plots are found on the tab. These plots are:

- the temperature graph
- the magnetometer graph
- the gyroscope graph
- the accelerometer graph

Since there are only five plots, each graph could take up more space, leading to having a better overview and more detail in the graphs.

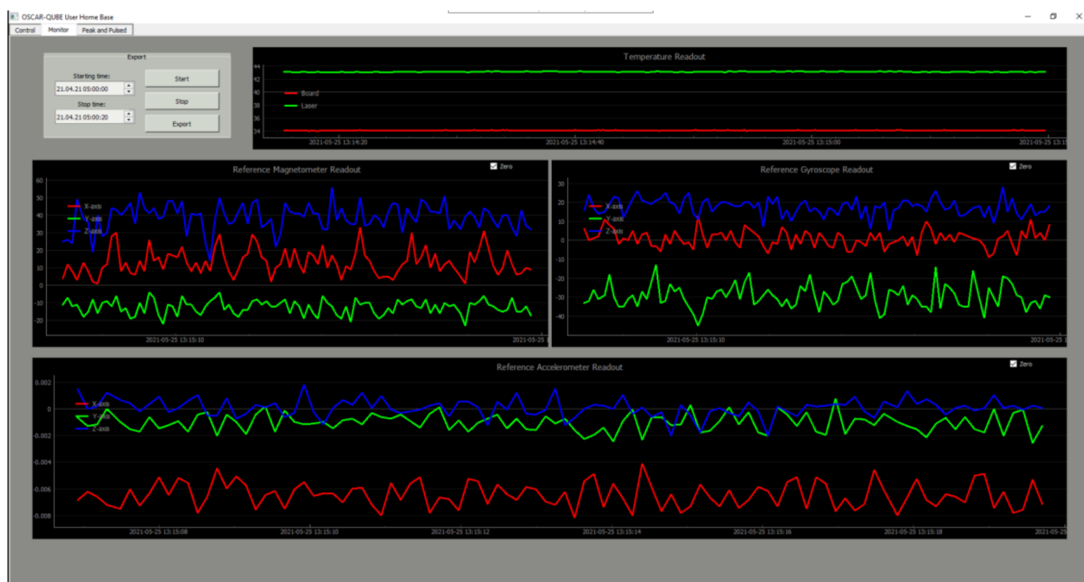

Figure 45: Monitor tab with zeroing activated on all plots

Each plot shows the timestamp of its values on the X-axis. However, not all graphs cover the same period. The temperature plot displays the last 500 points, while the other graphs only show 100 points. This is due to the fact that the changes in temperature appear slower over time then the others. The other plots put more emphasis on displaying abrupt changes to the QUBE's environment. Furthermore, the magnetometer, gyroscope and accelerometer plots each have their own zeroing feature. This was not created for the temperature plots as the detail seen in this graph was sufficient for its purpose.

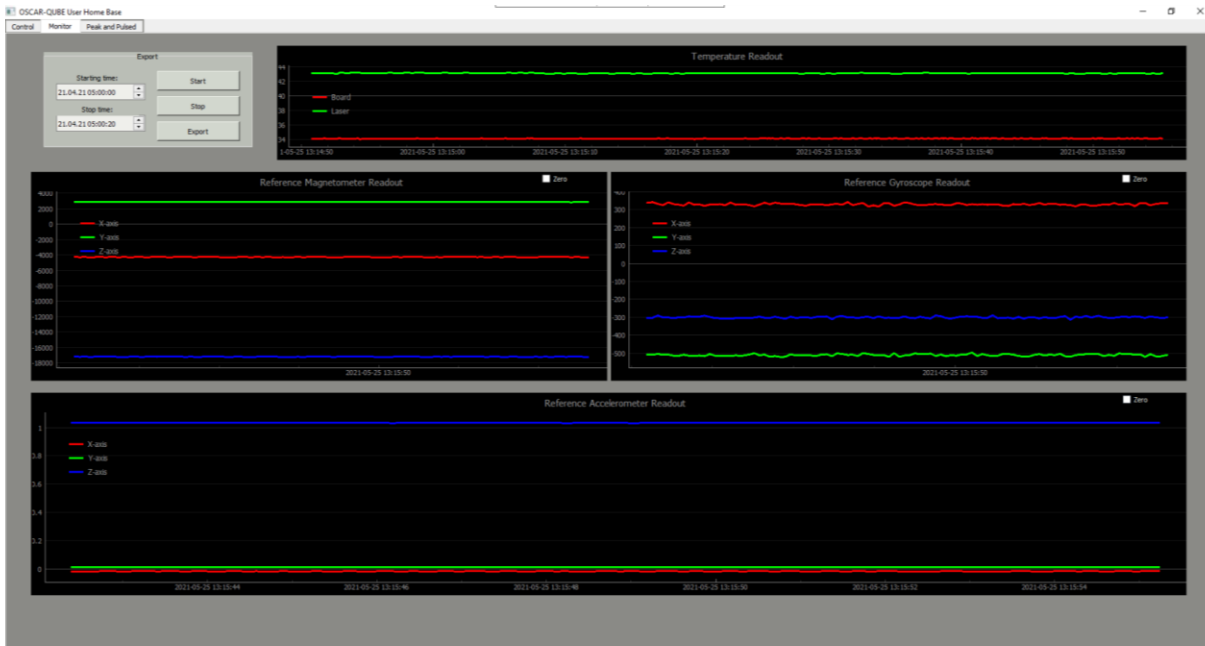

Figure 46: Monitor tab with zeroing disabled on all plots

Export group

This group, seen in Figure 47, enables the operator to export the packets collected between the start and stop time. This time can be set manually by typing it into the designated date-time box. However, when performing shorter experiments, the start and stop buttons can also be utilized. These buttons will set the corresponding date-time box to the time of clicking the button. This enables the scientists to easily define a start and stop time when manually performing experiments. Exporting this data will produce two different formats. The first is the pickle format which is mainly used to read the data with other Python based scripts. The second format is a csv format, in which each (type) of data will be stored in a separate csv file. This data will mainly be used in post processing algorithms which are not Python based, such as MATLAB.

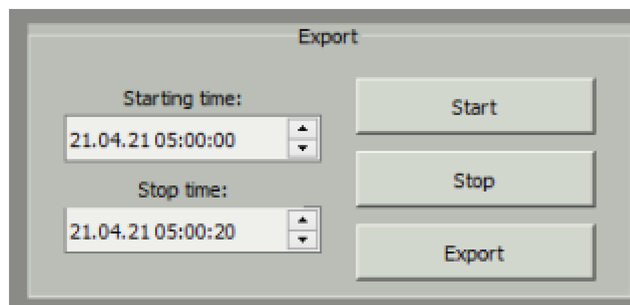

Figure 47: Detail of export group

5.2.3 Peak and Pulsed tab results

The last tab (Figure 48) enables the operator to configure the advanced operation modes. These operation modes are the peak detection operational mode and the pulsed operational mode.

Figure 48: Peak and Pulsed tab

Peak detection

The upper half of the tab, shown in Figure 49, is assigned to the peak detection. Here another "ODMR readout" plot is displayed because the intended configuration parameters can be derived from this graph. Under the graph, a table consisting of one row contains the configuration data. Next to this table, the button is placed to send the formatted pulsed command to the cube.

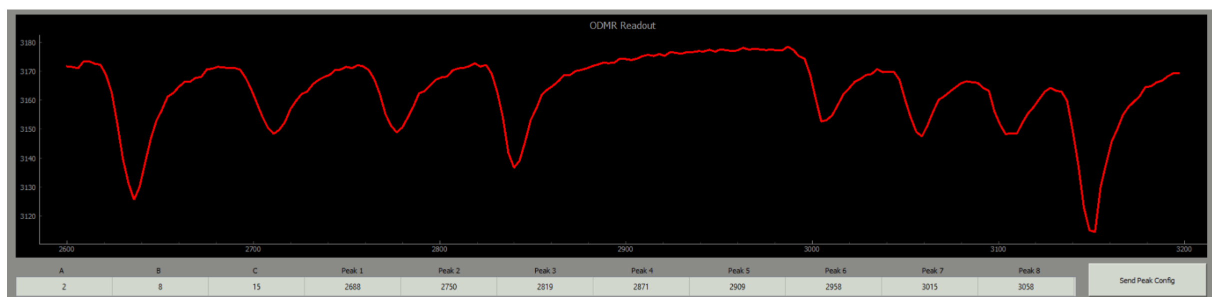

Figure 49: Detail of peak detection half

Pulsed operation

The bottom half of the tab, as seen in Figure 50, is devoted to the configuration of the pulsed operational mode. On the top there is a table consisting out of two rows. The first row of parameters stands for the first step of the pulsed mode, the second row represents the second step. Based on these two rows the next 98 steps are calculated and can be displayed in the graph below the table. The operator can step through these steps using three different methods:

- manually input the desired value into the input box
- step through using the arrow buttons on the right side of the input box
- use the slider to slide through all of the steps

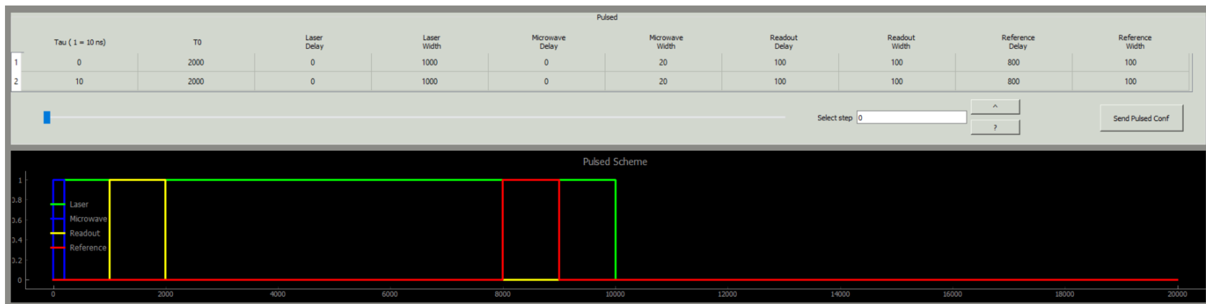

Figure 50: Detail of pulsed operation half

On each step the graph below will update and visualize the pulsed scheme the system onboard the ISS will perform on this specific iteration. Lastly, on the bottom right of the "Pulsed" group, a button is placed to send the two commands necessary to correctly configure the cube to operate as intended. Upon clicking this button, the GUI will first validate if the system is set in constant wave (CW) mode. If this is not the case, it will first change the QUBE's operation mode to this mode. This is due to the fact that the embedded system might freeze upon receiving these commands and when the system is in CW mode, there is a watchdog running in order to make sure the system keeps operating. If the system would freeze while in CW mode, the watchdog will detect this and automatically restart the system.

5.2.4 Threads

All of the threads perform well. The GUI thread is not suffering from any unresponsiveness, even when the most demanding tasks are being executed. The telecommanding and export threads form a proper buffer here. Furthermore, the interface thread performs nicely. Except for the fact that when the system is receiving data coming from the onboard SD-card of the QUBE, the graphs seem to lag. However, this is not the case. The perceived update speed of the monitoring widgets is decreased to roughly half of its update speed when receiving only live packets. This is due to the fact that the UHB has to filter out the packets coming from the SD-card in order to visualize the live data.

The ping threads operation is outstanding. During the development of the QUBE, the GUI thread froze on a couple of occasions. Nevertheless, the UHB kept sending pings at a constant rate to the embedded system. This proves the ping thread is reliable and it will keep pinging the QUBE regardless of the other tasks to be performed by the UHB.

5.2.5 Functionality

The UHB has a wide variety of functionalities to ensure the operator is able to execute tasks, interpret data etc. This chapter will show these results more into detail based on exported TM data which was gathered by the QUBE. This data was exported for postprocessing using the export feature of the GUI described in chapter 4.3.8.

Telemetry visualization

One of the main abilities the UHB should have is to receive, format, store and visualize incoming TM from the QUBE. The QUBE's operation is autonomous and therefore will just gather the data automatically and send it to the UHB. Upon receiving this data, Yamcs (of which configuration can be seen chapter 5.1) will read the incoming packet and assign each piece of the packet to the corresponding value. Once this step is completed, the data is automatically stored in Yamcs' database.

To then visualize this data the UHB utilizes a Python script. One of the key parameters of the TM which has to be shown, is the ODMR readout graph, which is described under chapter 5.2. This ah yeahdata displays the live ODMR readout data coming from the QUBE and can be seen in Figure 51. The graph shown here has plotted all of the graphs within the captured timeframe in the same plot. However in reality, the GUI will only plot one of these graphs at once.

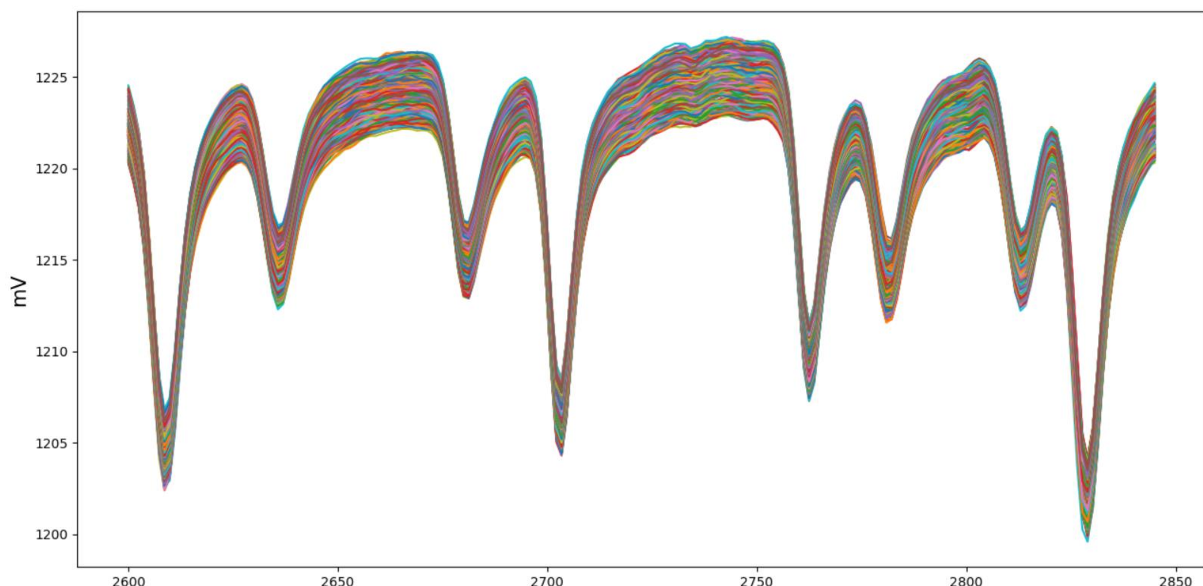

Figure 51: ODMR readout display in sweep mode

Furthermore, the average of each ODMR readout array (Figure 52) is calculated and displayed on the same tab below the ODMR readout graph. This data can be used during scientific experiments or when searching for the optimal configuration settings of the QUBE.

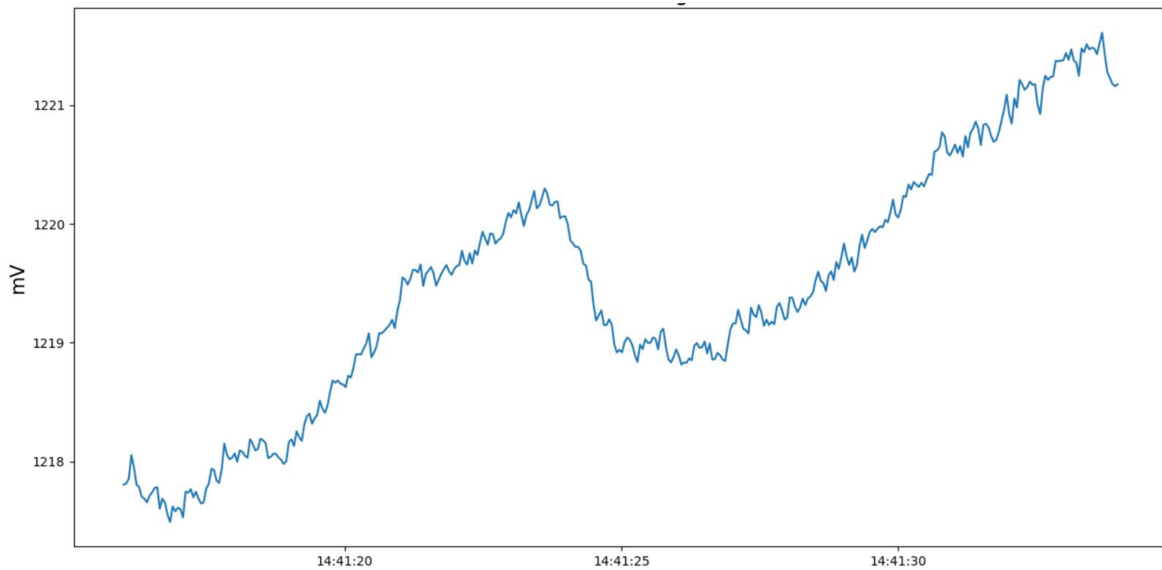

Figure 52: ODMR average over time display

Additionally, on the Monitor tab, described in chapter 5.2.2, other parameters that can help to understand the environment of the QUBE are visualized over time as well. These visualizations are demonstrated in Figure 53 in their "zeroed" form. However, as can be read in chapter 4.3.7 only the accelerometer, magnetometer and gyroscope data have this feature, meaning the temperature plot is not zeroed.

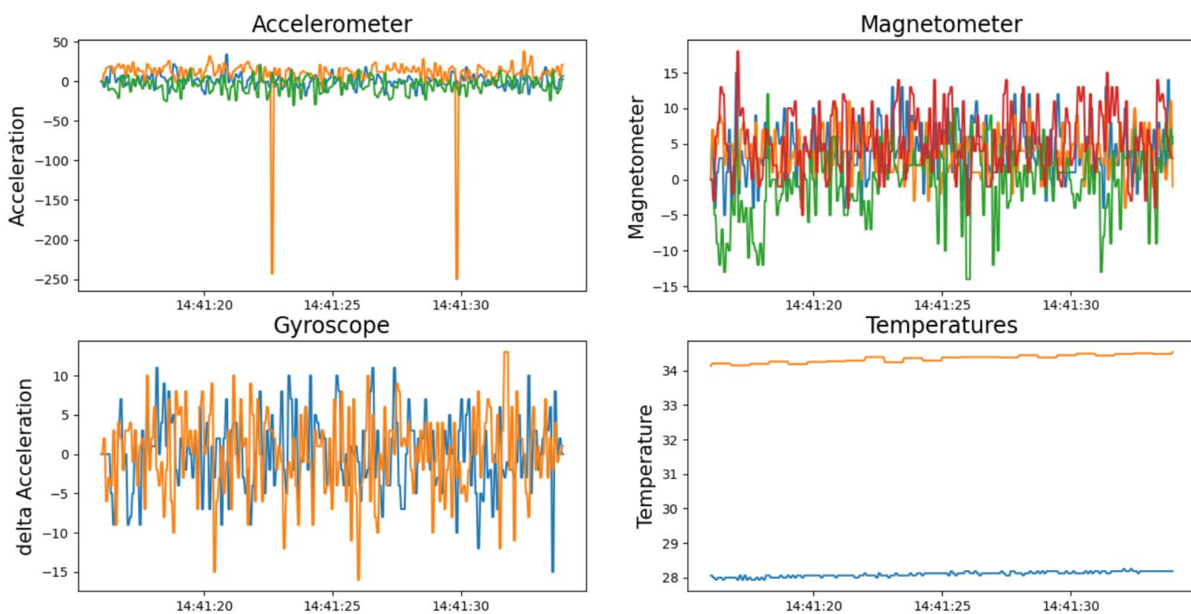

Figure 53: Reference subsystem displays in their zeroed state

Pulse visualization

The visualization of the pulsed operation, described in chapter 5.2.3, aids in properly configuring this operational mode. For this to work, the two rows of the corresponding table have to be filled in to fit the intended configuration. Based on these values the GUI calculates the pulse scheme which will be executed on the QUBE when utilizing this configuration. Since the embedded system executes 100 iterations of the pulsed operation to get a result, the GUI enables the user to step through each step. This is demonstrated in **Fout! Verwijzingsbron niet gevonden.**.

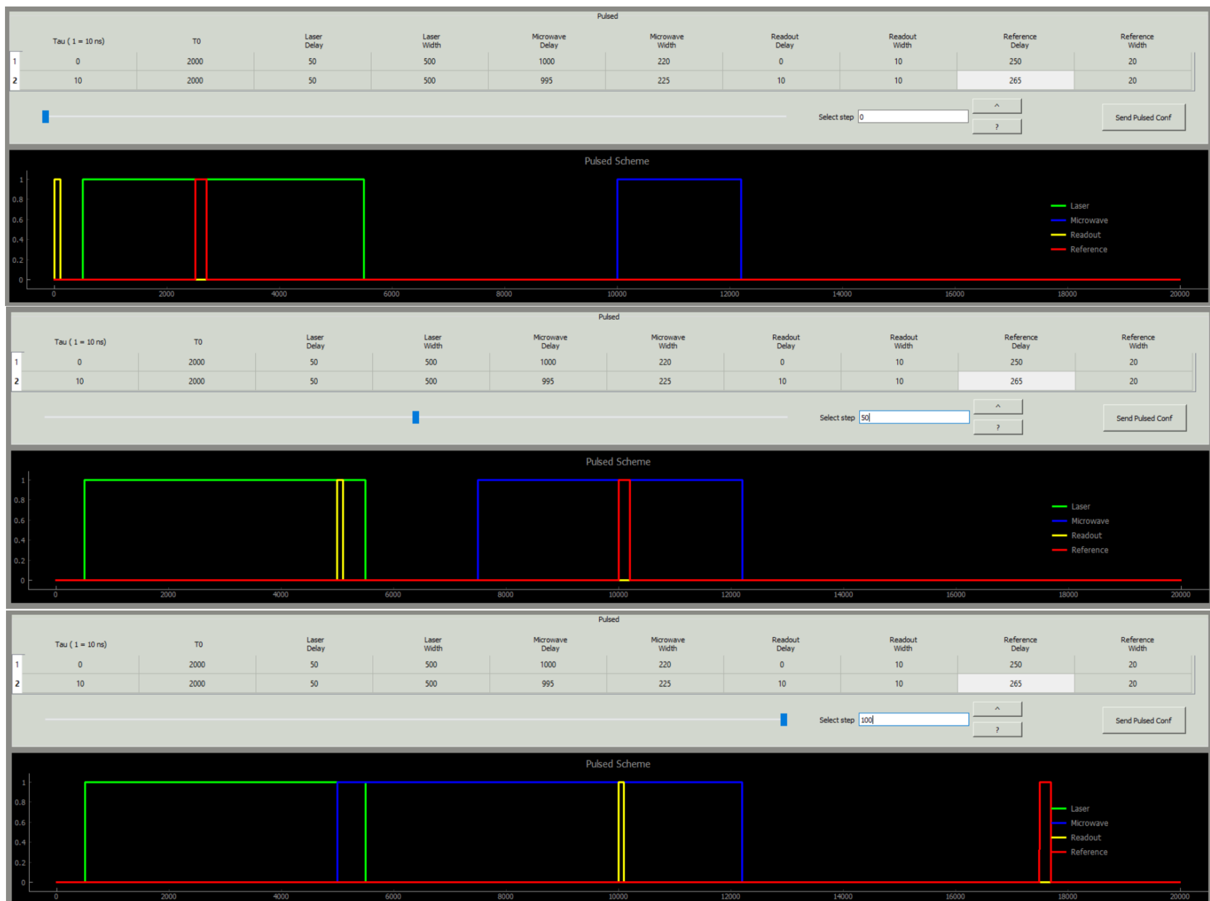

Figure 54: Stepping through pulse steps

Here the operator stepped through the configuration from step 0 to step 100. It can be seen that the readout pulse (yellow) moves forward. This should be the case as the delay in the table for step two is higher than step one. This also is the case for the reference pulse (red), but this pulse moves substantially more as there is a bigger difference between the delay in the first two steps.

Not only the delay can be adjusted. The pulse width can be altered as well over the iterations. The microwave pulse combines this pulse width altering with the difference in delay configuration to expand its pulse forward.

Chapter 6: Evaluation

The evaluation of the UHB was performed during the development of the QUBE. The UHB was used both for testing purposes and numerous measurements were performed using the system. Furthermore, the UHB was utilized during the interface test at Space Applications which validated the proper configuration of the VPN. In the meantime, simulation runs were performed with the QUBE using the actual networking system the project will use during its mission. Additionally, multiple tests were performed in order to see how the QUBE and UHB would react during times where the connection would be non-optimal or even lost. The boundaries were tested with these tests going substantially further then the nominal conditions the system will be exposed to.

6.1 Reliability

The tests performed at Space Applications proved that the system is reliable, even during non-optimal conditions. During these extreme conditions, both TM and TC packets might get lost, but losing TM packets does not pose any threat to the operations of the UHB. However, if a TC packet does not arrive at the QUBE, the state shown in the GUI might get mismatched when compared to the actual state of the embedded system. The debug responses in the debug group can be used to combat this. Since the QUBE will always respond to a TC, its arrival can be validated using this feature. Leading to an extra fail safe to confirm the TC properly arrived. If a packet did not arrive, the command can be sent again.

6.2 Ease of use

The ease of use was mainly evaluated by the scientific department of the team, as they had to perform most of the measurements during the development. Since they were able to give their preferences before the creation of the GUI, it was made with this in mind. Furthermore, during the development of the UHB, this could be further adjusted to their liking. As a result, the GUI provides a nice and intuitive interface with the embedded system.

6.3 Future work

In the future several points will be added. These points are:

- highlighting LOS in the GUI
- exporting over a serial connection
- automatic operational mode configuration
- advanced visualizations of the data

The LOS has to be highlighted in the GUI because no TC will arrive at the QUBE during these periods. Thus, to mitigate the risks of mismatching the displays with the actual state of the embedded system, a visual cue will be created. This will base itself on the calculated packet speed and can be cross validated using the data received from Space Applications. Furthermore, TC can be blocked dur LOS to ensure no mismatch will be created by the operator.

The export over serial connection will be implemented since the VPN blocks all of the internet connections unrelated to the experiment. Therefore, a serial connection bridge shall be created between the UHB and a Raspberry PI. The UHB shall run a python script which sends all of the incoming packets to the PI using a USB cable. This Raspberry will then receive these commands and distribute them to other servers by sending the serial data over the internet. On these servers, the data can be stored, processed, visualized etc.

In the future the system will be expanded with an automation feature in which the incoming data will be analyzed directly by the GUI and the parameter for certain operational modes, such as the six-point method, will be extracted from this data. Based on these parameters, the system will automatically reconfigure the QUBE to fit these values and switch to the corresponding operational mode. The GUI shall return to the first operational mode once the incoming data proves the current settings are outdated. Upon returning to the base operational mode, the incoming data shall be investigated once again, in order to extract the new parameters that would be most suited for the other operational mode. This will continue until the feature is disabled. The method would be useful since it will automatically find the best configuration for some of the most advanced operational modes. Furthermore, these modes are scientifically more valuable and this feature will ensure an automatic operation of them, meaning they can be used more often and do not require regular supervision and validation.

Advanced visualization of the data will also be added to the system. However, it holds the lowest priority out of all the other tasks. This is due to the fact that this feature is not mission critical or would bring significant benefits to the project.

Chapter 7: Conclusion

The main goal of the thesis was to develop a system which could interface with the OSCAR-QUBE magnetometer over ethernet. Furthermore, operators from the scientific department of the team, should be able to operate the experiment with little technological knowledge. A UHB was created with Yamcs as the MCS. The software was configured to be able to communicate over ethernet using UDP packets. Additionally, three custom packet structures were created and defined in the MDB. Two TM structured were defined using the XTCE structure. The TC was created using the spreadsheet method. Yamcs is able to differentiate between the packets based on the APID.

To allow operators to interface with Yamcs and control the QUBE, a GUI was created using PyQt5 and the corresponding Python script. Here a clear overview is given on three different tabs. Using the layout, the GUI can be commanded by using buttons. All of the inputs, such as the input boxes, buttons, radio-buttons and slider, of the layout are tailored for a better user experience based on the feedback given by several team members. Furthermore, the GUI contains all of the features necessary to properly operate the embedded system. Additionally, the GUI is responsible for the pinging of the QUBE, which it performs perfectly. The UHB was thoroughly tested on reliability and user experience during its development and the interface test campaign of the system.

During the interface test, the requirements of system stability were simulated with a data loss of 5% and the system recovered automatically from simulated LOS periods. Furthermore, the UHB is able to receive the TM data coming from the SD-card of the QUBE while maintaining a solid connection to receive live commands during the stress-test. This confirmed the system is robust and will reliably support the student project OSCAR-QUBE during its ten month mission onboard the ISS.

Literature

[1] M. Díaz-Michelena, "Small magnetic sensors for space applications," *Sensors*, vol. 9, no. 4, pp. 2271–2288, 2009, doi: 10.3390/s90402271.

[2] M. Buchner, K. Höfler, B. Henne, V. Ney, and A. Ney, "Tutorial: Basic principles, limits of detection, and pitfalls of highly sensitive SQUID magnetometry for nanomagnetism and spintronics ARTICLES YOU MAY BE INTERESTED IN," *J. Appl. Phys*, vol. 124, p. 161101, 2018, doi: 10.1063/1.5045299.

[3] A. Cerman, A. Kuna, P. Ripka, and J. M. G. Merayo, "Digitalization of highly precise fluxgate magnetometers," *Sensors Actuators A*, vol. 121, pp. 421–429, 2005, doi: 10.1016/j.sna.2005.03.053.

[4] L. Rondin, J. P. Tetienne, T. Hingant, J. F. Roch, P. Maletinsky, and V. Jacques, "Magnetometry with nitrogen-vacancy defects in diamond," *Reports Prog. Phys.*, vol. 77, no. 5, 2014, doi: 10.1088/0034-4885/77/5/056503.

[5] F. M. Stürner *et al.*, "Compact integrated magnetometer based on nitrogen-vacancy centres in diamond," *Diam. Relat. Mater.*, vol. 93, no. January, pp. 59–65, 2019, doi: 10.1016/j.diamond.2019.01.008.

[6] J. F. Barry *et al.*, "Sensitivity optimization for NV-diamond magnetometry," *Rev. Mod. Phys.*, vol. 92, no. 1, 2020, doi: 10.1103/RevModPhys.92.015004.

[7] M. Gulka *et al.*, "Pulsed Photoelectric Coherent Manipulation and Detection of N-V Center Spins in Diamond," *Phys. Rev. Appl.*, vol. 7, no. 4, 2017, doi: 10.1103/PhysRevApplied.7.044032.

[8] "Major Tom - Satellite Mission Control." https://azuremarketplace.microsoft.com/en-us/marketplace/apps/kuboscorporation1595969685746.majortomsaas?tab=overview (accessed May 27, 2021).

[9] "What is OSI Model | 7 Layers Explained | Imperva." https://www.imperva.com/learn/application-security/osi-model/ (accessed May 07, 2021).

[10] "TCP/IP Model: Layers & Protocol | What is TCP IP Stack?" https://www.guru99.com/tcp-ip-model.html (accessed May 07, 2021).

[11] "TCP/IP model vs OSI model |." https://fiberbit.com.tw/tcpip-model-vs-osi-model/ (accessed May 08, 2021).

[12] "TCP vs UDP: What's the Difference?" https://www.guru99.com/tcp-vs-udp-understanding-the-difference.html (accessed May 08, 2021).

[13] "PicSat." https://picsat.obspm.fr/communication/ccsds-packets?locale=en (accessed May 07, 2021).

[14] "CCSDS.org - The Consultative Committee for Space Data Systems (CCSDS)." https://public.ccsds.org/default.aspx (accessed May 07, 2021).

[15] "Riverbank Computing | Introduction." https://riverbankcomputing.com/software/pyqt (accessed May 12, 2021).

[16] "Wireshark · Go Deep." https://www.wireshark.org/ (accessed May 10, 2021).

[17] "STM32 Nucleo Boards." https://www.st.com/en/evaluation-tools/stm32-nucleo-boards.html#overview (accessed May 11, 2021).

[18] "NUCLEO-F746ZG STMicroelectronics | STMicroelectronics STM32 Nucleo-144 MCU Development Board NUCLEO-F746ZG | 917-3772 | RS Components." https://bh.rsdelivers.com/product/stmicroelectronics/nucleo-f746zg/stmicroelectronics-stm32-nucleo-144-mcu/9173772 (accessed May 11, 2021).

Appendix A


```
<ParameterSet>

    <Parameter name="apid" parameterTypeRef="uint16_t"/>
    <Parameter name="Debug_String" parameterTypeRef="string"/>

    <Parameter name="AccelX" parameterTypeRef="int16_t"/>
    <Parameter name="AccelY" parameterTypeRef="int16_t"/>
    <Parameter name="AccelZ" parameterTypeRef="int16_t"/>

    <Parameter name="GyroX" parameterTypeRef="int16_t"/>
    <Parameter name="GyroY" parameterTypeRef="int16_t"/>
    <Parameter name="GyroZ" parameterTypeRef="int16_t"/>

    <Parameter name="MagnetoX" parameterTypeRef="int16_t"/>
    <Parameter name="MagnetoY" parameterTypeRef="int16_t"/>
    <Parameter name="MagnetoZ" parameterTypeRef="int16_t"/>

    <Parameter name="Temp" parameterTypeRef="int16_t"/>

    <Parameter name="DataODMR_Array" parameterTypeRef="array"/>
    <Parameter name="DataPDMR_Array" parameterTypeRef="array"/>

    <Parameter name="Filler_Array" parameterTypeRef="array"/>

    <Parameter name="Freq_Array" parameterTypeRef="array"/>

    <Parameter name="MW_minFreq" parameterTypeRef="uint16_t"/>
    <Parameter name="MW_maxFreq" parameterTypeRef="uint16_t"/>
    <Parameter name="MW_attenuation" parameterTypeRef="byte"/>
    <Parameter name="MW_steps" parameterTypeRef="byte"/>
    <Parameter name="MW_registers" parameterTypeRef="MW_reg_array"/>

    <Parameter name="Laser_temps" parameterTypeRef="Laser_temp_array"/>
    <Parameter name="Laser_status" parameterTypeRef="byte"/>
    <Parameter name="Laser_error" parameterTypeRef="byte"/>
    <Parameter name="Potentio_value" parameterTypeRef="uint16_t"/>

    <Parameter name="Tick_count" parameterTypeRef="uint32_t"/>
    <Parameter name="Block_index" parameterTypeRef="uint32_t"/>
```

```
    <Parameter name="MW_broad_steps" parameterTypeRef="byte"/>

    <Parameter name="Freq_skip" parameterTypeRef="uint16_t"/>
    <Parameter name="Broad_sweep_steps" parameterTypeRef="byte"/>

    <Parameter name="Op_mode" parameterTypeRef="byte"/>
    <Parameter name="FPGA_mode" parameterTypeRef="byte"/>
    <Parameter name="Measure_flag" parameterTypeRef="byte"/>
    <Parameter name="LastPingTick" parameterTypeRef="uint32_t"/>
    <Parameter name="HasConnection" parameterTypeRef="byte"/>
    <Parameter name="ADC_settings" parameterTypeRef="byte"/>

    <Parameter name="AccelGyroSettings" parameterTypeRef="AccelGyroSettings_array"/>
    <Parameter name="MagnetoSettings" parameterTypeRef="MagnetoSettings_array"/>

    <Parameter name="DevicesInit" parameterTypeRef="byte"/>

    <Parameter name="Average_Optical" parameterTypeRef="uint16_t"/>
    <Parameter name="Average_FPGA" parameterTypeRef="uint16_t"/>

    <Parameter name="Padding" parameterTypeRef="byte"/>

</ParameterSet>
```

Appendix B


```
<SequenceContainer name="Measurements">
	<EntryList>
		<ArrayParameterRefEntry parameterRef="DataODMR_Array">
			<DimensionList >
				<Dimension>
					<StartingIndex>
						<FixedValue>0</FixedValue>
					</StartingIndex>
					<EndingIndex>
						<FixedValue>199</FixedValue>
					</EndingIndex>
				</Dimension>
			</DimensionList>
		</ArrayParameterRefEntry>

		<ParameterRefEntry parameterRef="Temp"/>

		<ParameterRefEntry parameterRef="GyroX"/>
		<ParameterRefEntry parameterRef="GyroY"/>
		<ParameterRefEntry parameterRef="GyroZ"/>

		<ParameterRefEntry parameterRef="AccelX"/>
		<ParameterRefEntry parameterRef="AccelY"/>
		<ParameterRefEntry parameterRef="AccelZ"/>

		<ParameterRefEntry parameterRef="MagnetoX"/>
		<ParameterRefEntry parameterRef="MagnetoY"/>
		<ParameterRefEntry parameterRef="MagnetoZ"/>

		<ArrayParameterRefEntry parameterRef="Laser_temps">
			<DimensionList >
				<Dimension>
					<StartingIndex>
						<FixedValue>0</FixedValue>
					</StartingIndex>
					<EndingIndex>
						<FixedValue>7</FixedValue>
					</EndingIndex>
				</Dimension>
```

```
        </DimensionList>
</ArrayParameterRefEntry>

<ParameterRefEntry parameterRef="Padding"/>
<ParameterRefEntry parameterRef="Padding"/>

<ParameterRefEntry parameterRef="Laser_status"/>
<ParameterRefEntry parameterRef="Laser_error"/>
<ParameterRefEntry parameterRef="Potentio_value"/>

<ParameterRefEntry parameterRef="MW_minFreq"/>
<ParameterRefEntry parameterRef="MW_maxFreq"/>
<ParameterRefEntry parameterRef="MW_attenuation"/>
<ParameterRefEntry parameterRef="MW_steps"/>

<ParameterRefEntry parameterRef="MW_broad_steps"/>

<ParameterRefEntry parameterRef="Padding"/>

<ArrayParameterRefEntry parameterRef="MW_registers">
        <DimensionList >
                <Dimension>
                        <StartingIndex>
                                <FixedValue>0</FixedValue>
                        </StartingIndex>
                        <EndingIndex>
                                <FixedValue>5</FixedValue>
                        </EndingIndex>
                </Dimension>
        </DimensionList>
</ArrayParameterRefEntry>

<ParameterRefEntry parameterRef="Freq_skip"/>
<ParameterRefEntry parameterRef="Broad_sweep_steps"/>

<ParameterRefEntry parameterRef="Padding"/>
<ParameterRefEntry parameterRef="Padding"/>

<ParameterRefEntry parameterRef="Op_mode"/>
<ParameterRefEntry parameterRef="FPGA_mode"/>
<ParameterRefEntry parameterRef="Measure_flag"/>
<ParameterRefEntry parameterRef="LastPingTick"/>
<ParameterRefEntry parameterRef="HasConnection"/>
```

```
            <ParameterRefEntry parameterRef="Padding"/>

            <ParameterRefEntry parameterRef="Average_Optical"/>
            <ParameterRefEntry parameterRef="Average_FPGA"/>

            <ParameterRefEntry parameterRef="Padding"/>
            <ParameterRefEntry parameterRef="Padding"/>

            <ParameterRefEntry parameterRef="Tick_count"/>
            <ParameterRefEntry parameterRef="Block_index"/>
            <ParameterRefEntry parameterRef="ADC_settings"/>

            <ArrayParameterRefEntry parameterRef="AccelGyroSettings">
                <DimensionList >
                    <Dimension>
                        <StartingIndex>
                            <FixedValue>0</FixedValue>
                        </StartingIndex>
                        <EndingIndex>
                            <FixedValue>2</FixedValue>
                        </EndingIndex>
                    </Dimension>
                </DimensionList>
            </ArrayParameterRefEntry>

            <ArrayParameterRefEntry parameterRef="MagnetoSettings">
                <DimensionList >
                    <Dimension>
                        <StartingIndex>
                            <FixedValue>0</FixedValue>
                        </StartingIndex>
                        <EndingIndex>
                            <FixedValue>1</FixedValue>
                        </EndingIndex>
                    </Dimension>
                </DimensionList>
            </ArrayParameterRefEntry>
            <ParameterRefEntry parameterRef="DevicesInit"/>

        </EntryList>
        <BaseContainer containerRef="Header">
            <RestrictionCriteria>
```

```
				<ComparisonList>
					<Comparison parameterRef="apid" value="0"/>
				</ComparisonList>
			</RestrictionCriteria>
		</BaseContainer>
</SequenceContainer>
```