Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Marton, Per Amund Roaulsson

Satellite Communications Testing on a Smallsat using Payload Hardware-in-Loop

Master's thesis in Cybernetics and Robotics
Supervisor: Johansen, Tor Arne
Co-supervisor: Garrett, Joseph
July 2021

NTNU
Norwegian University of
Science and Technology

Marton, Per Amund Roaulsson

Satellite Communications Testing on a Smallsat using Payload Hardware-in-Loop

Master's thesis in Cybernetics and Robotics
Supervisor: Johansen, Tor Arne
Co-supervisor: Garrett, Joseph
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Acknowledgment

This project was performed in cooperation with the NTNU HYPSO team. Special thanks to Joseph Garrett for taking his time integrate me into the HYPSO team, and guiding me through the process of writing this thesis. I also want to thank Line Rønning for always supporting me throughout my studies, and especially while writing this thesis.

P.A.R.M

Remark:

Thanks to Professor Emeritus Marvin Rausand for creating and distributing the LaTeX template used for this thesis.

Summary

English

This thesis discusses the testing of the HYPSO-1 cubesat using a Payload Hardware-in-Loop Jenkins testing setup, with a focus on the testing of the communications of the satellite. In the thesis, the implementation of a way to utilize a NNG connection to a flatsat in Vilinus in order to simulate in-flight communication is developed for use with the preexisting Jenkins testing setup. The result of the work was that new issues with the HYPSO software was identified and addressed, and new avenues of improvement was discovered.

Norsk

Denne masteroppgaven diskuterer testing og implementering av HYPSO-1 cubsat-en ved bruk av en Payload Hardware-in-Loop Jenkins testoppsett, med søkelys på testing av kommunikasjon-systemene til satellitten. I denne oppgaven ble det implementert en metode til å bruke en NNG forbindelse til en Flatsat i Vilinus for å simulere in-flight kommunikasjon, utviklet for bruk med det eksisterende Jenkins testoppsettet. Resultatet av arbeidet var at nye problemer med HYPSO programvaren ble identifisert og adressert, og nye forbedringspotensialer ble oppdaget.

Contents

Chapter 1

Introduction

When developing a Small Satellite such as the HYPSO-1 cubesat, it is crucial that the software and hardware on-board the payload operates as intended, as there is no way to physically inspect or alter the payload once it is launched. However, it is still possible to write updates to the software and upload it after launch, given that the communications can be upheld with the satellite. This means that rigourusly testing the communications are one of, if not the most important, part of the system testing. HYPSO uses a Payload Hardware-in-Loop testing setup using a Jenkins server and a flatsat, and this thesis explains the work performed to improve this testing setup, implement new tests and the testing of the communications of the HYPSO-1 satellite.

1.1 The HYPSO Project

Figure 1.1: The logo of the NTNU HYPSO smallsat team.

HYPSO stands for HYPer-spectral Smallsat for ocean Observation, and is a student driven small-satellite team at the Norwegian University of Science and Technology (NTNU), made consisting of arround 30 students and staff. The team is developing a cubesat with a Hyper Spectral camera as the main payload. The first satellite developed by the HYPSO team; the HYPSO-1, is currently set to launch in the 4^{th} quarter of 2021, and the team are in the planning stages of the next iteration of the satellite; the HYPSO-2.

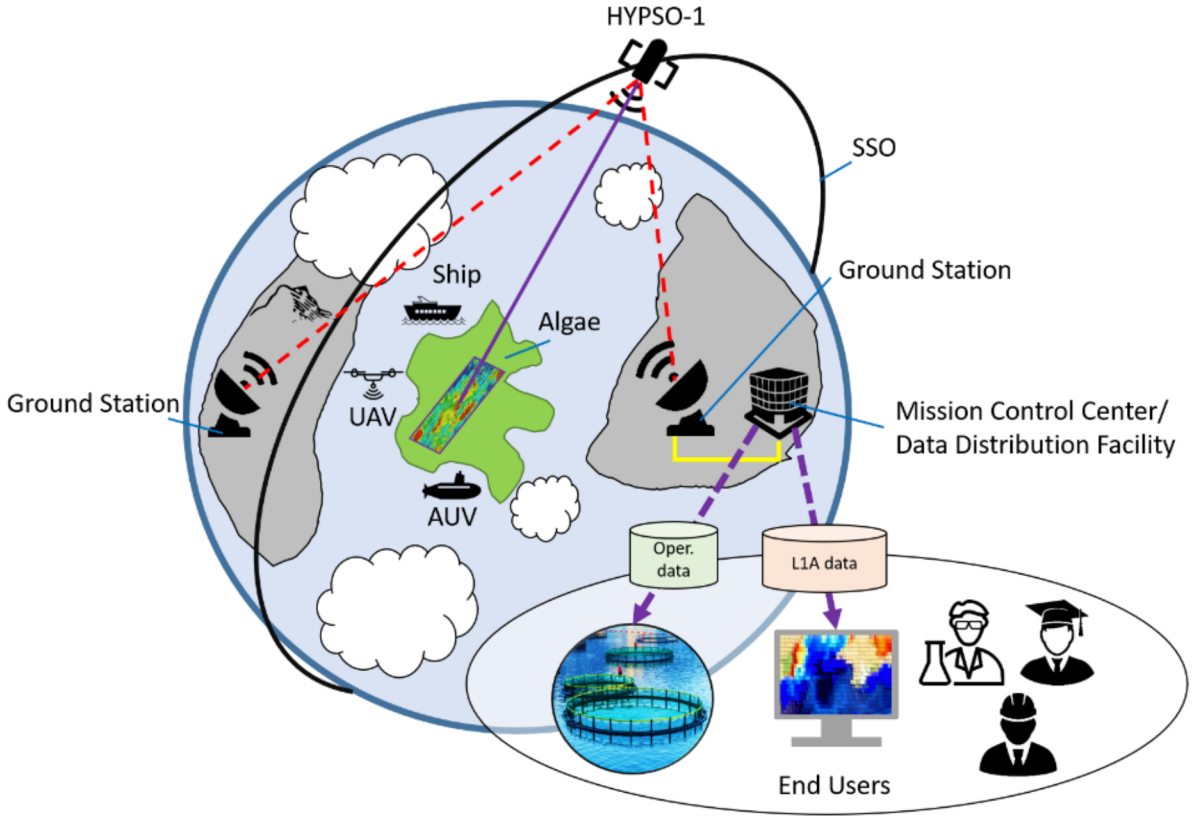


Figure 1.2: An overview of the HYPSO mission. Credit; Mariusz E. Grøtte

1.1.1 The HYPSO Mission Statement

The HYPSO mission is defined as;

> **The HYPSO mission shall provide and support ocean color monitoring and mapping through hyperspectral imaging, autonomously processed data, and on-demand autonomous communications in a concert of robotic agents at the Norwegian coast.** (HYPSO Project Team 2020*c*, p. 18)

In simpler terms, the HYPSO team is launching a satellite carrying a Hyper Spectral Imaging (HSI) camera into space in order to observe the hyper-spectrum of the coast of Norway. This data will be processed on-board the satellite before down-linking the data, as this reduces the size of the data. The data will be available for researchers, and the data will be used in cooperation with autonomous drones for researching the ocean (HYPSO Project Team 2020*c*).

1.2 Satellites

1.2.1 A generic satellite

A general satellite can in broad strokes be divided into two parts; the satellite platform and the payload; The satellite platform are all the subsystems required for the operation of the satellite, while the payload is subsystems performing the mission of the satellite (Braun 2012). After all, there is no point in launching a satellite with no purpose.

The satellite platforms are in large satellites usually custom made, but for smaller satellites there exists commercial of-the-shelf satellite platforms called satellite busses. The payload houses all the hardware related to the satellites mission, for example telescopes, communication devices or in the case of the HYPSO-1 payload; cameras.

1.2.2 Smallsats & Cubesats

A cubesat is a term elegantly defined by the National Aeronautics and Space Administration (NASA) as "a satellite smaller than a fridge" (Mabrouk 2017). With such a broad definition a lot of spacecraft fall into this category, but by far the most common one is the cubesat; a standardized sizing of smallsats. The smallest cubesat is defined as 1 U; a cube with the sides of 10 cm and a weight limit of 2 kg following the standards set by Wenschel Lan (2020). cubesat comes in larger sizes as well, see fig. 1.3. In fact, the HYPSO-1 satellite is a 6 U cubesat.

Cubesats has grown in popularity since the first one was first launch in 2002. According to the paper Villela et al. (2019), the 1000^{th} cubesat is expected to be launched in 2021, although

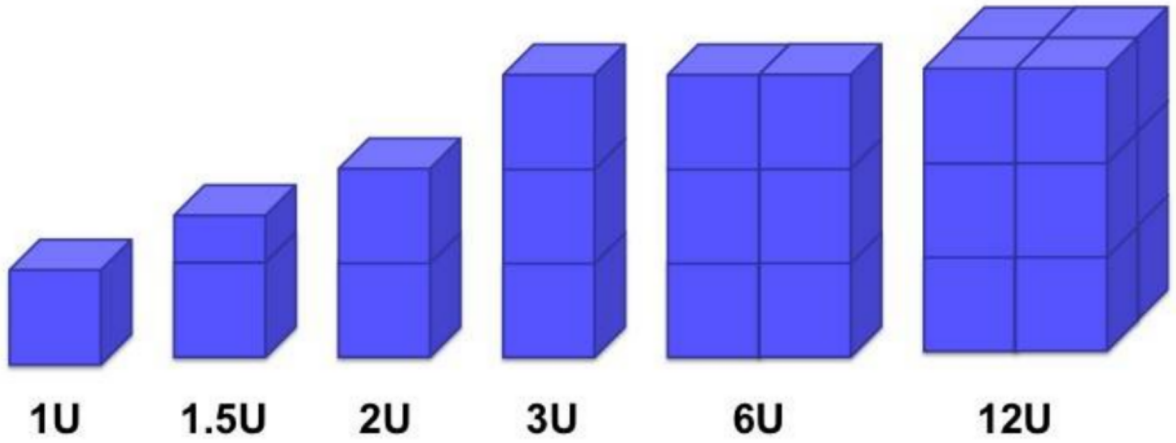


Figure 1.3: The standardized sizes of cubesats (Mabrouk 2017).

this is probably a little behind the schedule due to COVID-19. One of the reasons for this is that its relatively cheaper; by having standardized sizes and requirements multiple cubesats can piggyback a launched of another payload, and the cost thus goes down. Another reason for their popularity is that they are aimed towards research, especially at universities, leading to smallsat teams such as HYPSO being created (Villela et al. 2019).

However, the backside of the cheaper development is that the cubesats does not always get tested as much as regular satellites. As seen in table 1.1, a growing percent of missions are successful. This is partially due to improved launch techniques, lessons learned from previous cubesats, and the existence of commercial cubesat busses that gives a solid framework to build upon.

Time range	No. cubesats	Mission success	Unknown	Launch failure	Cubesat failure
2005—2012	99	48%	0%	19%	33%
2005—2015	417	56%	0%	19%	22%
2005—2018	848	61%	2%	12%	21%

Table 1.1: The mission statuses of cubesats over time (Villela et al. 2019)

.

One major reason for cubesat failures are "infant mortality". This is when the cubesat does not operate after deployment for whatever reason. It can be issues such as faulty software, broken hardware from the launch or plain old human error (Villela et al. 2019). But infant mortality aside, the fact is that the main type of cubesat failure is still failure during flight, gives an indication that the cubesats have probably not been as rigorously tested as they should have been.

This highlights the importance of testing the cubesat software repeatedly and thoroughly to prevent mission failure during flight.

1.2.3 The HYPSO-1 Satellite

HYPSO-1 a cubesat of the size 6 U, and is the first satellite developed by the HYPSO team. It is further described in section 1.4, but for now it is important to note that all references to "the satellite" in this thesis refers to the HYPSO-1, unless explicitly stated otherwise.

The payload of HYPSO-1 is developed by the HYPSO team, and the satellite platform is a satellite bus bought of-the-shelf from NanoAvionics (Tran 2019), and is further explained in section 1.4.1. The payload has a HSI camera based of the design proposed by Sigernes et al. (2018), as well as a RGB camera. The design of the payload can be seen in fig. 1.7 and is further explained in section 1.4.2.

1.2.4 The HYPSO-2 Satellite

The HYPSO Team is currently in the planning phase of the next smallsat: HYPSO-2 of the size6 U. This new version will build upon the HYPSO-1 design, but with some improvements, most notably deployable solar cells to increase the power budget and a RS422 connection between between the payload and the payload controller.

1.2.5 The Importance of Testing Satellite Software & Hardware

In general, performing hardware maintenance is out of the window when it comes to satellites in orbit, and uploading software updates can be dangerous and slow, even if the satellite is designed for it. This means that in order to give the satellite mission the best chance as possible the software and hardware of the satellite must be rigorously tested before launch. It is important to note that testing can only find issues, not guarantee the absence of issues.

There are many ways to do testing on satellite software and hardware, for example using a Payload Hardware-in-Loop testing setup such as the one at HYPSO, see section 4.1.2.

1.3 Hyperspectral Imaging

HSI is a special type of imaging where where optics are used to separate the different wavelengths from each other, making it possible to take a picture of an object and identify what wavelengths we are actually seeing. This means that, if the spectrum is distinct enough, one

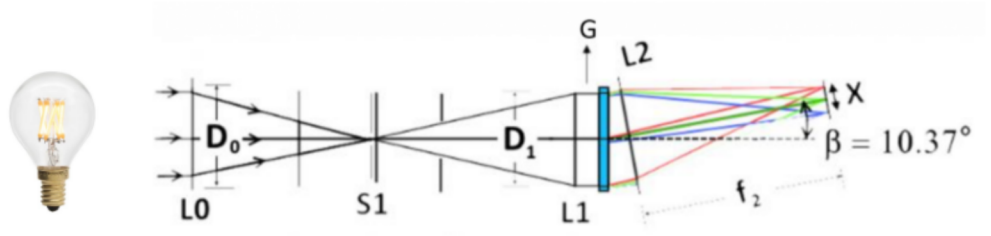


Figure 1.4: The principle of a HSI camera used at HYPSO. Credit; Fred Sigernes

can identify what materials the object the camera points at is made up of (Sigernes et al. 2018).

A design requirement of a HSI camera is that the light going into the camera has to go through a narrow slit before using optics to refract and diffract the light through a narrow grate and onto the sensor, fig. 1.4. This means that when taking an HSI image, one is actually just taking an $1 \times n$ pixel "image" in the normal sense of the word; One of the axes normally reserved for the image now represents the different intensities of wavelengths! In order to take a full width image, the camera has to move; It can either rotate, move physically, or a combination thereof. This is called the push broom technique, as the sensor is pushed over the object that is to be imaged, see fig. 1.5. This technique will be utilized in the HYPSO satellites to create hyper spectral images of the ocean.

1.4 The Hardware of HYPSO-1

The hardware of HYPSO-1 is divided into two parts; the payload developed by HYPSO and the satellite bus delivered by NanoAvionics, see fig. 1.6.

1.4.1 The Satellite Bus

The Satellite Bus is a 6U NanoSatellite Bus M6P, bought off-the-shelf from, built by, and delivered by NanoAvionics, see fig. 1.6a. It contains all the vital functions of a satellite, but without the payload. It has an UHF antenna and transmitter, S-Band antenna and transmitter, and interfaces with the payload using a subsystem called the Payload Controller (PC). There are a few other subsystems worth mentioning in relevance to this thesis; the Flight Computer (FC) in charge of the positioning of the satellite and the Electrical Power System (EPS) in charge of managing power to the different subsystems of the payload and satellite bus (NanoAvionics 2018).

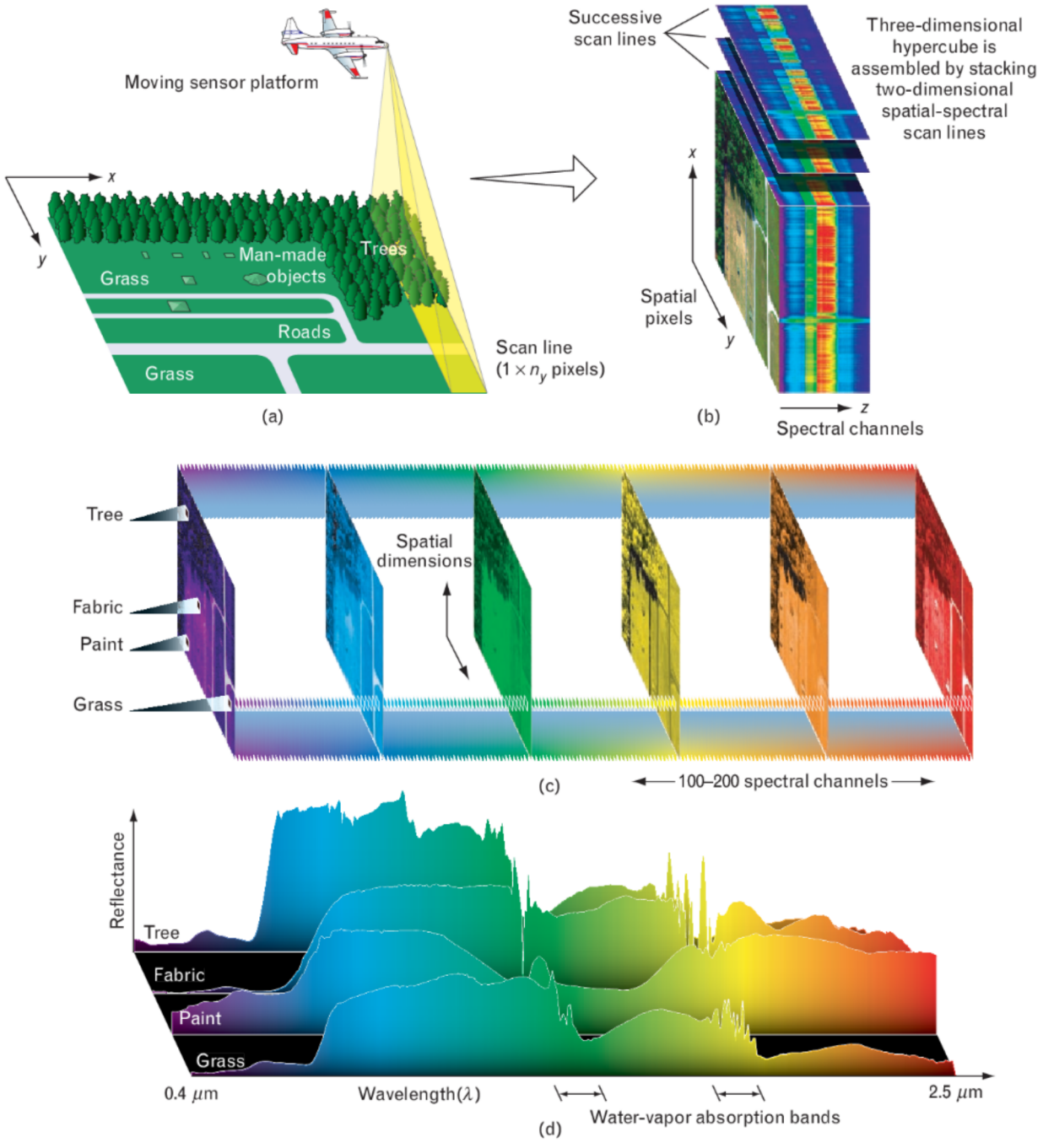
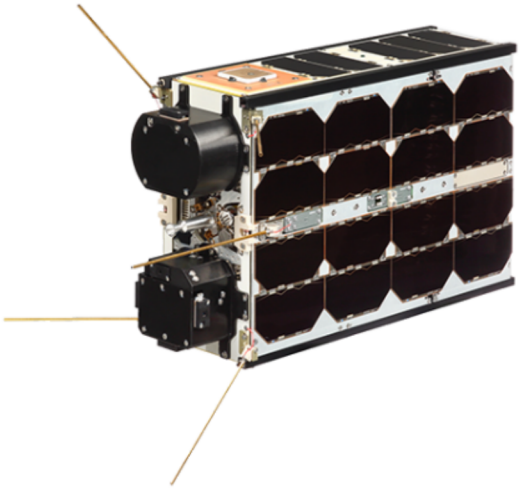
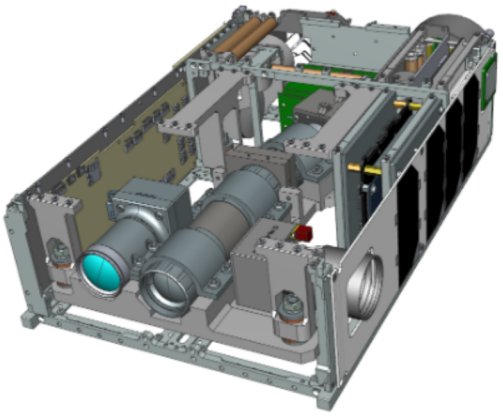


Figure 1.5: A demonstration of the push broom technique. Shaw & Burke (2003)



(a) The 6U NanoSatellite Bus M6P
Credit; NanoAvionics



(b) HYPSO-1 without the solar cells.
Credit; Elizabeth Prentice & Martine Hjertenæs

Figure 1.6: The outside and inside of the HYPSO-1 SmallSat

1.4.2 The HYPSO Payload

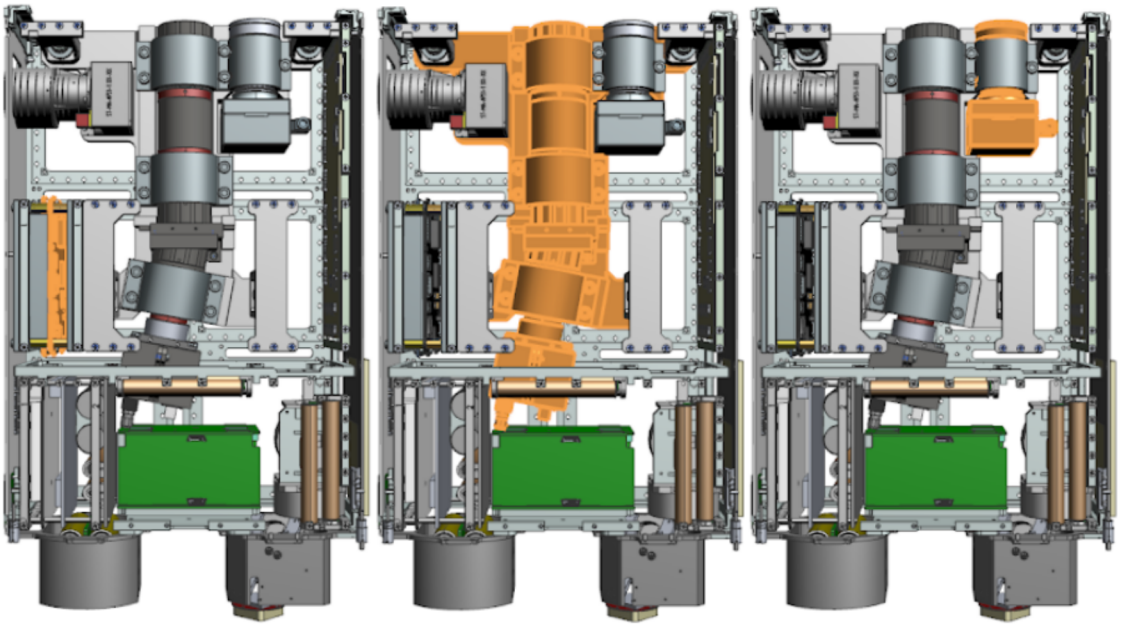
The HYPSO the payload of the system (payload) is the part of the satellite that performs the HYPSO mission. It consists of several subsystems, but the major ones — the ones most discussed in this thesis — are the OPU, HSI, RGB and SDR.

1.4.3 OPU — On-board Processing Unit

The On-board Processing Unit (OPU) is the system in the top of the hierarchy of the payload. It interfaces with the PC using the CAN1 netwrork, and with the HSI, RGB and SDR using the CAN2 network. The OPU is housed in a PicoZed 7030 System-On-Module board (Pico) that has a HYPSO-developed Breakout-Board (BOB) mounted on top. By hyphenation the the combined system has been baptized the PicoBOB, and is housed mounted as seen in fig. 1.7a. (HYPSO Project Team 2020*b*)

opu-system

The On-board Processing Unit system (opu-system) is the operating system of the OPU. It is designed to perform the mission related computing, such as software compression, and it runs the opu-service software (section 3.2.2) on top to interface with the payload and the Payload Controller.



(a) OPU payload subsystem (b) HSI payload subsystem (c) RGB payload subsystem

Figure 1.7: Different subsystems of the HYPSO-1 payload. Credit; Elizabeth Prentice & Martine Hjertenæs

1.4.4 HSI — Hyper Spectral Imaging

The Hyper Spectral Imaging (HSI) is a payload subsystem containing a HSI camera and its required hardware, see fig. 1.7b. The HSI camera will be used to capture hyper spectra images of the ocean of the Norwegian coast. The design of the camera is explained in fig. 1.4 and is based on the design proposed by Sigernes et al. (2018), and in fact, Sigernes has been of great help in the development of the HSI camera of the HYPSO-1.

1.4.5 RGB — Red, Green & Blue

Red, Green & Blue (RGB) is a payload subsystem containing a RGB camera and its required hardware. Note that the name RGB refers to the entire payload subsystem, not just the camera. The RGB camera is a bare PCB camera of the type UI-1252LE-C with a resolution of 2 MP, and is mounted to HYPSO-1 using a custom housing (Langer & Hjertenæs 2020), see fig. 1.7c. The camera will be used for positioning and supplementing the HSI camera.

1.4.6 SDR — Software Defined Radio

Software Defined Radio (SDR) is the tertiary payload of the HYPSO-1, and is a containing a RGB camera and its required hardware. Its mission statement is defined as; "To measure ra-

dio interference and perform downlink channel measurements for future communications in the Arctic."(HYPSO Project Team 2019). The SDR is not further discussed in this thesis, but is mentioned as it is a part of the HYPSO-1.

Chapter 2

Testing using Satellite Communications

2.1 Communications Protocols and Connections

This section describes some of the communications protocols and connections in use at HYPSO, see fig. 2.1, as the they are used in the testing at HYPSO, see section 4.1.2.

2.1.1 CAN — Controller Area Network

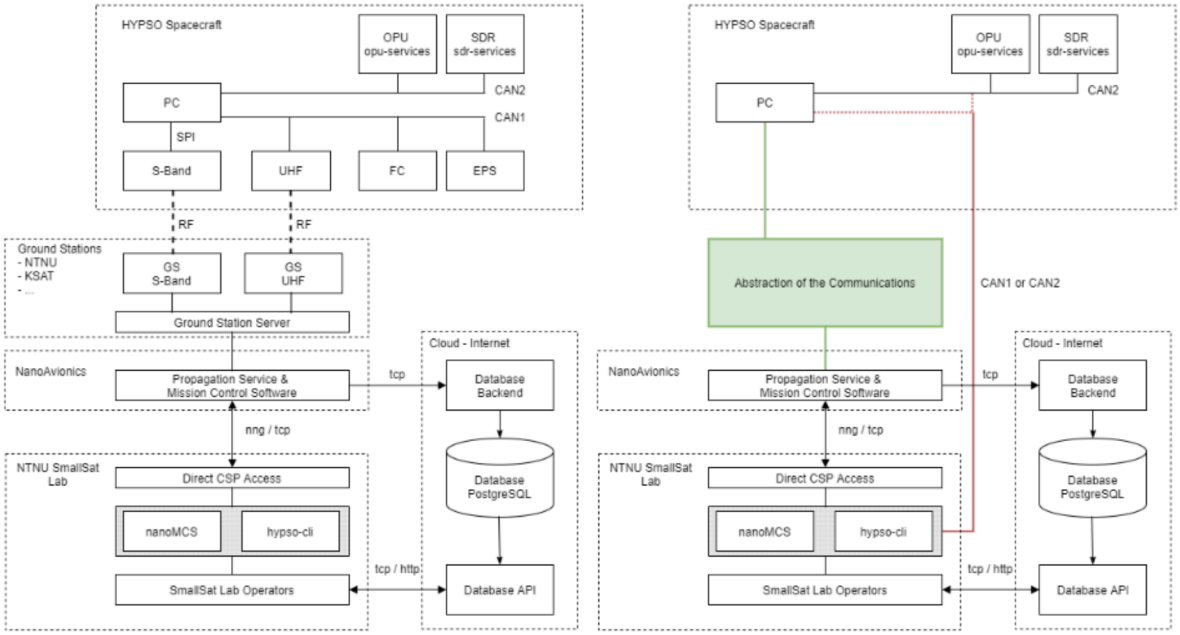
Controller Area Network (CAN) is a communications network designed to allow multiple systems to share one connection to pass messages without using a master. The HYPSO-1 uses the CAN1 network between the payload subsystems and the PC, and the CAN2 network between the PC and the rest of the satellite bus (Di Natale 2012). The Payload Hardware-in-Loop (PHiL) testing setup at HYPSO can be connected to the flatsat using either the CAN1 to communicate with the payload directly, or CAN2 to communicate through the PC HYPSO Project Team (2020*b*).

2.1.2 CSP — Cubesat Space Protocol

The Cubesat Space Protocol (CSP) is a robust protocol developed to simplify the communication between distributed embedded systems in a small networks, such as in a Small Satellite (smallsat). It is based on the CAN protocol, but has developed to support multiple other protocols (Jahren 2015). All CAN networks at HYPSO are using the CSP protocol (HYPSO Project Team 2020*b*).

2.1.3 NNG — nanomsg-next-gen

nanomsg-next-gen (NNG) is a library offering a API to solve reoccurring issues related to messaging, such as connection management (Garrett D'Amore 2021). At HYPSO the NNG is used to connect to the NanoAvionics Mission Control Software (MCS), who again routes the data to



(a) Overview of the communications off the HYPSO-1 mission

(b) Abstraction of the communications off the HYPSO-1 mission

Figure 2.1: Detailed overview of the communications at HYPSO, and an abstracted version used for testing. Credit for the figure these figures are based upon; Amund Gjersvik

and through the satellite bus and to the Payload Controller. The author of this thesis developed a way for the PHiL to use the NNG to run tests, see section 5.1.3.

2.1.4 Ultra High Frequency

Ultra High Frequency (UHF) is defined by the Institute of Electrical and Electronics Engineers (IEEE) defines it as the range from 300 MHz to 3 GHz in the electromagnetic spectrum (Raab et al. 2002). A common characteristic of these frequencies is that they propagate mainly by line-of-sight, and is good at penetrating the ionosphere, meaning it a good pick for communicating when HYPSO-1 is in line-of-sight.

2.1.5 S-Band

The short-band (S-Band) is defined by the IEEE as a range of the electromagnetic spectrum, specifically the microwave band. It ranges in frequency from 2GHz to 4GHz, and is mainly used for radar systems such as communication satellites. This is also the same wavelength band that many commercial products use, for example WI-FI, but by the varying focus and power there is no significant interference (Srinivas et al. 2011).

2.2 Satellite Communications — Uplink & Downlink

When HYPSO-1 is in orbit, the combined communications can be described as following; The user uses the HYPSO client (hypso-cli) command-line interface to send a message to the MCS using NNG. The MCS then sends the message to the Ground Station Server who passes the message using either S-Band or Ultra High Frequencies (UHF). The HYPSO-1 receives the message in the satellite bus, and passes the message to the PC, who finally sends the message from the hypso-cli to the On-board Processing Unit service (opu-service) (HYPSO Project Team 2020*a*). This is called uplinking, and when the order of operations are in reverse it is called downlinking. See fig. 2.1a for reference.

2.3 Satellite Communications for Testing — Uplink & Downlink

The preexisting testing setup at HYPSO, section 4.1.2, is connected to the lidsat (lidsat) (section 4.1.1) by CAN1 or CAN2 to the PC or payload respectively, but since the PC passes messages to the CAN2 the end result is almost equivalent, but routing the message through PC is preferable as this test more parts of the final system. This is shown by the red part of fig. 2.1b.

The newly implemented NNG connection in the testing setup takes this a step further; the NNG connection is used to pass messages to the flatsat (section 4.1.1) by sending messages to the MCS over NNG. When the message is recieved at MCS, NanoAvionics takes over the communication between the MCS and the PC. This means that by testing over the NNG connection, the UHF and S-Band can be abstracted away, as we have little to no control at that point, wether we are testing or in-flight, we trust NanoAvionics to do their part correctly. This is shown by the green part of fig. 2.1b.

Chapter 3

Software and Coding Practices in use at HYPSO

3.1 Software Used for Testing at HYPSO

Among the software used at HYPSO, some are especially important to mention in relevance to this thesis, as they are all in use in the Jenkins testing setup, see section 4.3.

3.1.1 Jenkins

Jenkins is an open-source, java based, Continuous Integration server that is most commonly used for detecting issues in software at early stages. It can perform tasks like building the software, testing the software and releasing the software if the tests pass and the user desires so. The way Jenkins works is that the software is built at regular intervals, and then it runs tests on that software to see if it passes or fails a set of criteria. If it fails it means that a recent change to the software has created the issue, and this issue can quickly be addressed (Simpson 2015).

At HYPSO, a Jenkins Testing Setup has been built to test the HYPSO software, see section 3.2. The server is configured to run whenever there is a change to the HYPSO software, testing the software for a potential regression. This is done by declaring a "declarative pipeline" in a file called the *Jenkinsfile*, written in Groovy. The Jenkinsfile starts by downloading the source files of HYPSO software (hypso-sw) from a GitHub repository, then opens a Docker container to build the source files. After this a set of python scripts is run in consecutive order, first to turn on the power to the HYPSO-1, and then to test the software. The results of each test is are written to a database using PostgreSQL. At the end of the pipeline the power is turned back off, and if the Jenkins server has come this far without encountering an issue then the software is flagged as passed.

3.1.2 Groovy

Groovy is an open-source language created for the Java Virtual Machine, and has a syntax similar to Java. One of the advantages Groovy has over Java is that it supports both dynamic and static typing, instead if Java's strict static typing. But the main advantage of Groovy, and the reason for its use in the declarative pipelines of Jenkins, is that it is a testing oriented programming language, having built-in support for unit testing (Davis 2019).

3.1.3 Python 3

Python 3 is the newest iteration of the Python programming language, officially taking the place of Python 2 after the support for Python 2 ended in 2020. The "core values" of python can be said to be simplicity, explication and name-spaces. Python comes with standard libraries that cover most of what an average user needs, and if not there are thousands of additional libraries one can use. Another advantage of python is that it is a lean programming language, meaning that it can do the same as what another language could do, but with fewer lines, resulting in faster development and making it easier to learn. However an disadvantage of python is that it is not optimized for fast execution, such as a programming language such as C (Stross-Radschinski et al. 2014), which is why python is only used for testing the HYPSO-1 software, not to create it.

At HYPSO, python is used in the Jenkins testing setup as it makes the vast variety of tasks required for the testing simpler to implement as there is "a library for everything". The Jenkins testing setup for example requires ways of reading and writing to files, executing command line arguments, running the hypso-cli as a subprocess, interacting with a database, communicating using different protocols, creating and moving files. All of these are easy to implement using python as there is an easy way to do all this in python.

3.1.4 PostgreSQL

PostgreSQL, often just reffered to as Postgre, is one of the most popular open-source relational database management system (RDBMS), and it is well known for its reliability and extensebility. Although having Structured Query Language (SQL) in its name, it does not conform perfectly with the SQL standard, but it tries to do so when applicable (Maymala 2015).

At HYPSO, PostgreSQL is used in the testing of the hypso-sw. Every test run start by updating the database-table of that test with all relevant information about the test, and after the test is executed it updates the table with the result, if the test gets that far.

3.1.5 Docker

Docker is a Platform as a Service (PaaS) platform used to create isolated environments, called containers, for running a specific software. This means that all the dependencies of that software is included in the container, such that the environment of the computer the docker container is run on is abstracted away, eliminating dependency issues. While different types of PaaS has existed for a while, Docker has become the leading PaaS on the market due to its many features and ease of use (Anderson 2015).

At HYPSO, Docker is used for building the hypso-sw, meaning that any member of the team can download the code through Git and build the software independent of operating system and their virtual environment, and all of this right after downloading the code.

3.1.6 Git and GitHub

Git is an open-source distributed version control program, and is used for tracking changes during the development of a software. An important functionality of Git is the ability to branch a repository, creating a new version of the software. This branch can be changed without affecting the original branch, before being merged with the original when the development of the new branch is done and approved. This means that multiple parts of the software can be worked on and then combined together at a later date, increasing the number of people that can work on the same project (Spinellis 2012).

By using a web-hosting service called GitHub to coordinate the Git repositories, the HYPSO team can quickly download and edit code from all parts of the project. By the use of submodules the team can quickly include other parts of the project into another project, as in the `hardware_in_loop` repository. This repository is used for the Jenkins testing setup, and by adding the `hypso-cli` as a submodule it can be built and tested from this repository.

In the Jenkins testing setup at HYPSO the branching functionality is used to create different branches of the `hardware_in_loop` repository. Each branch is used for testing different connection types, and these branches are almost identical with the exception of some definitions. This might seem like an over-complicated way of doing it, but by doing so the team can add new connection types to test that might require larger changes without having to redo the entire code-base in every branch, while at the same time keeping the common material up to date.

GitHub is also used by the HYPSO team due to a functionality called project boards. This is a virtual notice board that can be used to create issues — a post-it-note with a feature to im-

plement or problem to fix — and then organize and assign them to developers. This is exactly the way the team uses the project boards, and it greatly assist in the agile development the team uses. The GitHub boards is very helpful in the context of testing; every time an issue is encountered in the software testing an issue is created in the project boards, and then during next sprint meeting a developer is assigned to fix the issue.

3.2 The HYPSO Software

The hypso-sw is a term used for the top level software developed for the HYPSO mission. The hypso-sw is divided in two categories, the hypso-cli and the opu-system, and this section describes both of them as they are both heavily used in this thesis.

3.2.1 hypso-cli

The HYPSO client (hypso-cli) is the software used to operate the ground segment of the HYPSO mission. It is a command-line interface that can communicate to the opu-service through CAN, NNG, S-Band and UHF. It is used to issue commands to the payload, get mission data, download files, upload files and update software on the the payload.

To summarize, hypso-cli is how the HYPSO team communicate with the HYPSO-1 satellite, making the testing of this system extremely important, as all of the aforementioned tasks must work exactly as intended to uphold communications with the satellite, otherwise the mission is a failure. In fact, six of the most critical failure modes of the HYPSO mission is communication related (Jordheim 2020), see table 3.1.

S-Band	—	Does not send data
S-Band	—	Can not resume transmitting the missing packets
S-Band	—	Sends corrupt data
UHF	—	Does not recieve data at all
UHF	—	Recieve corrupt data
UHF	—	Does not send data

Table 3.1: FMECA — Critical failure modes of the HYPSO-1 related to communications (Jordheim 2020)

3.2.2 opu-service

The opu-service is the software that run on the opu-system, it is in charge of the communications of the space segment of the HYPSO mission as well as managing the payload. It com-

municates internally with the payload and the PC using the CAN2 bus, and externally with the hypso-cli over S-Band and UHF through the PC (HYPSO Project Team 2020*a*).

In regards to testing, this part of the HYPSO software has a many systems to test, as the opu-service interfaces with the entire payload. Currently there are some tests for the HSI cameras, as well as the upload and download of files from the satellite. Some more tests has been implemented by the author, especially to further test the communications, see section 5.4.

3.3 Coding Practices for the HYPSO Testing

There are some steps one can take in order to keep a large programming project, organized, expandable and reusable. This section talks about the coding practices the HYPSO team has implemented in the Jenkins testing setup.

3.3.1 Consistent Naming of Files, Variables, and Functions

It is a fact that each software developer have different coding style and preference when it comes to naming files, variables and functions, and it is usually not a problem when only one developer works on the same project. But when a software is developed by multiple people it is important to keep it consistent. Therefore does all tests written for the Jenkins testing server require the functions and files to be written in `snake_case` and the variables to be in `camelCase`. Note that there are exceptions to this rule, but for the most part, this is the rules used.

3.3.2 Library Hierarchy and Folder Structure

A pitfall many developers fall into when starting a project is not setting up a proper folder structure. The files is put in the same folder and before you know it there are a lot of them. This happened in the Jenkins testing setup at HYPSO, see appendix B.2. It quickly become hard to navigate the files, and even harder to know which file uses another. Therefore it is important to have a library hierarchy; The one used in the Jenkins testing setup is simplistic, it places the files used by other files into the `lib` folder, and have two configuration files where the common data can be fetched, see appendix B.3.

3.3.3 The Advantages of Using Classes & Object Oriented Programming

When programming large sets of code, re-usability is key. This is especially true when writing software tests, as most of the code is repeated. This were not the case with the code written for the HYPSO testing, see section 5.1.1. Many functions were repeated in different files, sometimes

as exact copies, and other times as almost identical. By instead implementing these common functions in a separate library, and using classes with default values a lot of repeated coding can be avoided. For example; the `opu_upload.py` and the `opu_partial_upload.py` have almost identical code, and by implementing a `upload_test` class that can be imported the tests can use `subclasses` to modify the `upload_test` class to their purpose. This has now partially been implemented in the Jenkins testing setup, but there is still a lot of work to complete.

Chapter 4

The Software Testing at HYPSO

Most of the software testing at HYPSO is performed on the Payload Hardware-in-Loop using the Jenkins testing server. The main part of the testing performed are regression testing, as the testing of the hypso-sw sadly does not have a full test suite and is nowhere comprehensive and broad enough for acceptance testing.

4.1 Hardware for Software Testing

4.1.1 Flatsat & Lidsat

A flatsat is a board where the different components of a satellite is mounted, making it easy to change or configure the components. At HYPSO there are two flatsats; The first one, with the unoriginal name of just the flatsat, is located at NanoAvionics lab in Vilnius. The second one is in use by the PHiL at HYPSO.

4.1.2 PHiL — Payload Hardware in Loop

The Payload Hardware-in-Loop (PHiL) is a Jenkins testing setup connected to the same hardware that is used in the HYPSO-1. This means that ideally the system works the same as the satellite will during its mission, as all components are the same. At the time the author joined the HYPSO team the testing was only done locally over the CAN network, using the lidsat, but one of the tasks he implemented was to be able to testing the flatsat over the NNG, see section 5.1.3.

4.2 Software Testing at HYPSO

This section talks about the most relevant types of testing in relation to this thesis, and how it is used at HYPSO.

4.2.1 Unit Testing

A unit test can be seen as the smallest test possible; Does a function do what it is supposed to do? Writing unit tests can be done quickly, as a predetermined data-set going into the system should give a predetermined result coming out for the test to pass (Myers et al. 2004, p. 91). At HYPSO unit testing is implemented by the creator of the software that is to be tested creates the test.

4.2.2 Regression Testing

Whenever a change is made to a software we risk that it breaks something that used to work. Regression testing is to test for this; the new version of the software is only accepted if it at least passes the same tests as before (Myers et al. 2004, p. 19). At HYPSO this is implemented in the Jenkins testing server, as new versions of the hypso-sw is only accepted as long as it passes the tests defined in the Jenkins pipeline.

4.2.3 Acceptance Testing

Acceptance testing is to test the entire system and measure it up against its initial requirements set when it were designed Myers et al. (2004). At HYPSO this will be performed manually by comparing the software and hardware of the satellite up against the design outlined in the document HYPSO Project Team (2020*d*).

4.2.4 Hardware-in-Loop Testing

Hardware in Loop (HiL) is a way to test software on its intended hardware by simulating inputs to the model. By doing this one can get an idea on how the real system will react to a situation (Bacic 2005). At HYPSO, the HiL is implemented in the testing of the payload; the Payload Hardware-in-Loop (PHiL), see section 4.1.2.

4.3 Jenkins - Testing Server

The Jenkins testing server is the software part of the PHiL that runs the hypso-cli which interfaces with the lidsat using the CAN network and the flatsat by using the NNG connection. It is

used to streamline the continuous integration, and is run every time a change is made to the hypso-sw.

4.3.1 Qualification Model

Qualification is defined as a part of verification which shows that the system meets the qualification margins (ECSS Secretariat 2012).

A Qualification Model (QM) is by the ECSS defined as a "model, which fully reflects all aspects of the flight model design, used for complete functional and environmental qualification testing" (ECSS Secretariat 2012). It is in intended for use in the development and testing of the system, and includes all parts of the system that is to be tested. The QM is not a part of the flight, but a way to prepare for it.

At HYPSO the QM is used to run the software testing on. The desire is to have the QM as similar to the payload as possible. This is done by using the same hardware and communications as the satellite counterpart. In order to test the QM different parts of the system must be turned on/off. This part of the QM has been automated by the author, see section 5.5.

4.4 Testing of Satellite Software using the PHiL

4.4.1 Communications, Software and Updates

It is extremely difficult to perform maintenance on a satellite once it leaves earth, as maintenance is difficult when you have no wired connection. The the relatively low value of a SmallSat means that once it enters orbit it will have to manage on its own. This means that the hardware and software must be thoroughly tested before launch. There is however a little bit leniency on the software of the satellite, because as long as one can communicate with it one can potentially update the software. This means that in a way the most important part of satellite software testing is to ensure we can uphold communication channels. Following this reasoning one should focus more of the testing of the satellite software on the communications. This is not to say that the rest of the software can be shipped early, there would be no point in launching a non.functional satellite, but a bug in the communications software can likely mean mission failure, whereas a bug in the payload software can be updated. But this update is still a costly process, it will take valuable time and energy away from the mission.

Chapter 5

The Development of the HYPSO Testing Infrastructure

The work undertaken for this thesis were performed in cooperation and for the HYPSO team for the purposes of testing the software and hardware of the HYPSO-1 satellite. The majority of the work revolved around the Jenkins testing setup used at HYPSO, and can be summarized as;

- Refactoring and restructuring the code base for the Jenkins testing server used to test the HYPSO software.
- Creating tests for the HYPSO software, with a focus on the communications.
- Testing the HYPSO software, identifying issues and creating GitHub-issues for those issues so the software team can work on them.
- Creating a digital interface for multiple power supply units to program and monitor their status for qualification testing.

5.1 Refactoring and Strengthening the Jenkins Test Scripts

The preexisting Jenkins testing setup at HYPSO clearly had room for improvement; There were duplicate files, some temporary fixes had become permanent fixes, and the library were disorganized due to the nonexistent folder hierarchy as seen in appendix B.2. There were sporadic bugs that lead to failed pipelines, and it was apparent to the team that a major refactoring had to be undertaken on the Jenkins testing setup.

5.1.1 Refactoring the Jenkins Test Scripts

Thus, one of largest tasks I tackled this semester, in cooperation with co-supervisor Joseph Garrett, was to refactor the code-base of the testing setup.

The refactoring started off by sorting the testing files from the library files, and placing the library files in its own folder, `lib`. By doing this the `scripts` folder was declared solely for the files used for testing, making it easier to navigate the files. The next step were to rename files and variables, and extracting common code and declarations to one file; `regression_test_settings.py`. The final step was to tie it all together by restructuring the Jenkins declarative pipeline, explained in section 5.1.2.

The resulting folder structure can be seen in appendices B.2 and B.3, and is based on the coding practices outlined in section 3.3.

5.1.2 Restructuring the Jenkins Declarative Pipeline

At the time I joined the HYPSO team the main issues with the Jenkins Declarative Pipeline, the `Jenkinsfile`, were as following;

- Indentation: Groovy, the language the Jenkinsfile uses, does not require specific indentation, making it hard to read the file when the indentation is not kept consistent, this was a rather quick but noticeable fix.
- Syntax: The placement of brackets and variable names should be kept consistent, but were unfortunately not, something I corrected.
- Static scripting: The Jenkinsfile was calling hypso-cli directly, instead of calling a script to do so. This meant that if there were a change in a function call this had to be manually changed at multiple places in the file. This was not an issue at the time of overtaking the project, but posed an issue when implementing different connection types.

The end result can be seen in appendix C.1, the indentation and syntax is consistent, and all the scripts called uses command line arguments when applicable, instead of calling hypso-cli.

Implementing Proper Cleanup of the Pipeline

Whenever a Jenkins Pipeline is built and run, it is assumed The OPU is turned off. To ensure this the final stage of the pipeline is dedicated to turn off the power of the OPU and HSI.

Unfortunately this did not turn out to be the case, as whenever the pipeline does not complete this step is skipped and the OPU is never turned off, and thus must be turned of manually.

By changing it from a regular declarative stage to a `post` stage, the execution of the stage will happen regardless of the exit condition of the pipeline.

5.1.3 Implementing Testing Through NNG

The importance of the last point in the previous section becomes apparent when considering that the PHiL had been solely used for testing the hypso-cli and opu-system over the CAN networks. The team saw great potential in testing using the NNG connection, and I took it upon myself to implement this.

By replacing the static scripting with dynamic scripts the testing setup could import the address and connection settings of the opu-system from one file instead of having it defined multiple places.

I created a common file for all connection settings related information, `connection_settings.py` as seen in appendix C.2, and changed the existing scripts to import the variables from there. This meant that by just changing one variable different connection settings can be selected. This led to the discoveries of multiple issues, as described in section 5.4.

5.2 Issues Identified and Corrected

Memory Issue - Infinite Print

To ensure uploads and downloads work for larger file-sizes, the `opu_download.py` and `opu_upload.py` test scripts downloads relatively large files, up to the size of 55 MB. By modern communication standards this is a small file, but for the satellite communications on a smallsat this is a large file that will take multiple flybys to transfer. The size of the files are chosen as a trade-off between time and ensuring that the transfer of the files works as intended. This means that it can take from 10 to 30 minutes to run a single communications test, making the Jenkins pipeline run slower.
A recurring issue during the development of the testsing was that hypso-cli lost contact with the opu-system without an error message, leading to the test never finishing due to a faulty exit-clause. If this happened and the pipeline were not manually terminated the pipeline would continue to run until the next time someone wanted to use the Jenkins server, leaving the hypso-cli

and Jenkins server to continue logging the commands, printouts and timestamps of the messages.

Upholding Murphy's law, this exact scenario happened on a Friday evening, leaving the test to run over the weekend; The test hung and was not manually shut down, resulting in 54 GB of logs and miscellaneous files. The issue were soon narrowed down to a blank line being printed inside a polling function, meaning that the entire 67 hours the test run this line were printed and logged on repeat by both hypso-cli and Jenkins. After removing the log-files and the printout from the code, it was clear that proper timeouts had to be implemented. The first solution were to add a realistic timeout to the `opu_download.py` and `opu_upload.py` scripts, and by adding a larger timeout to each stage using the stage property of the Jenkins pipeline syntax: `options{timeout(time: 30, unit: 'MINUTES')}`.

Memory Issue - Docker

Along with the previous issue there were discovered that there were still memory issues on the Jenkins server, over 200 GB worth of files were unaccounted for. The obvious solution were to look in the directories used by the Jenkins server, but this lead nowhere. So a closer examination had to be performed on each step of the Jenkins pipeline. The most obvious issue were the amount of times the the tests had been run.

On the development branch used to develop the work performed in this thesis the pipeline has been run over 400 times (to varying degrees of success), and at the time this issue were examined, about 250 times. For every one of those 250 times, the entire git repository for the testing setup and all its submodules had been downloaded, extracted and built. This seemed like a good place to start, but was fruitless as the combined size of the software including the logs was still so small that it took less than 5 GB in total. This meant the culprit had to be the way we build the hypso-sw using Docker.

It turned out that each time Jenkins started a docker container to build the hypso-cli it downloaded and initialized the containers, and kept that container saved. By running one command, `docker system prune`, and being patient, this issue were closed.

In order to prevent this problem in the future an issue was created on GitHub; `Docker should overwrite the previous image using docker tag, not create a new one each time hypso-cli is built.`

Workspace Directory Issue

A major issue that perplexed me and the team was that the pipeline just froze sometimes for no apparent reason. Some of the tests started as normal but then when they started a sub-process running hypso-cli the response from opu-system were not displayed, and no error message were shown. This happened at an estimated 5% of the times hypso-cli were run, and the fact that it just happened at seemingly random and when rerunning the pipeline the issue often disappeared made it hard to narrow down.

I identified the bug to be a workspace directory issue; The way Jenkins handles the execution of the pipeline is by handing the resources to different agents, and there is no guarantee that the same agent runs the entire pipeline. Each agent has a local copy of the code, so when hypso-cli is built using docker in one stage it is just happenstance that the same agent runs the script at a later stage.

The first fix were to disable concurrent builds in the Jenkinsfile so that there is just one pipeline running at the same time. This had the positive side-effect of allowing queuing pipelines. This improved the issue, but did not eliminate it. By using the Jenkins function `stash` in the docker stage and then `unstash` in the stages requiring hypso-cli a copy of the built hypso-cli is always available regardless of workspace directory, eliminating this issue.

Missing Hypso Client

One of the reasons the workspace directory issue took so long to identify were that it only appeared when hypso-cli were in use, and the way the pre-exisitng python scripts were implemented were in a working but silent way. In other words; when opening a subprocess in python there are two pipes to listen to; the output - `stdout`, and the error messages - `stderr`. The existing implementation was only listening to the `stdout` pipe, and thus all warnings and error messages were ignored for the entire pipeline.

I believe this was overlooked by the creator, judging by the code, and most likely rooted in prioritizing other work. By routing the `stderr` to the `stdout` error messages were finally actually being handeled and recorded in the logs, leading to the workspace directory issue being identified.

Resuming Pipeline after Failed Stages

When running a testing pipeline it is in some situation desirable to resume the pipeline after a stage has failed. In many situations one can learn from where a stage fails and where another

does not, and I therefore rewrote the pipeline to continue after the non-essential tests fail. To illustrate the difference better one can look at these situations:

- When pinging the OPU: If we do not get a response there is no use in running tests on it, thus the entire pipeline should fail.
- When downloading a file from OPU: If the download fails it can be many reasons; it can take too long so it times out, the file is not being found, etc. In this case it is desirable to continue the pipeline, as the next test may still pass, meaning the issue is probably with the download.
- In the case of my script `opu_partial_download` fail, and the `opu_download` passes then we can assume the issue is most likely related to the stopping and resumption of the download, not the upload itself.

5.3 New Tests Implemented

During my work at HYPSO I implemented multiple tests for the Jenkins testing server, this section briefly describes them. All files can be found in the hardware_in_loop GitHub repository at HYPSO.

5.3.1 opu_partial_upload.py

This script is based upon the preexisting `opu_upload.py` script, but with one major difference, every 3 minutes the upload stops and is then resumed. By doing this the resumption of the upload can be tested. Only some of the tests are added in the appendix, as the length of the files would have made this thesis very long. See appendix C.4.
Usage: `opu_partial_upload.py [NODE]`

5.3.2 opu_partial_download.py –timeout

This script is based upon the preexisting `opu_download.py` script, but with one major difference, every 3 minutes the download stops and is then resumed. By doing this the resumption of the download can be tested.
Usage: `opu_partial_download.py [NODE]`

5.3.3 ping_timeout.py

This script uses the CSP ping functionality of hypso-cli to ping a specific node and waits for 30 seconds before continuing the pipeline. See appendix C.4.1

Usage: `ping_timeout.py [NODE] -timeout [optional TIMEOUT]`

5.3.4 ping_simple.py

This script uses the CSP ping functionality of hypso-cli to ping a specific node, it is less robust than the `ping_timeout.py`. See appendix C.4.2.
Usage: `ping_simple.py [NODE] -timeout [optional TIMEOUT]`

5.3.5 ping_all.py

This script uses the CSP ping functionality of hypso-cli to ping all nodes on the network, and will only pass if all pings come back. See appendix C.4.3.
Usage: `ping_all.py -timeout [optional TIMEOUT]`

5.4 Issues Found as a Result of Testing

When creating a test for a software, one often finds out that the software criteria are not met, and although the test "fails" it is exactly this behaviour that desired as it highlights an area that must be further improved. This section explains the different issues encountered with the hypso-sw during testing in Q1 and Q2 of 2021, and to illustrate the importance of testing an example from the previous semester is provided;

A potential mission critical bug were found in the hypso-sw using the PHiL testing in the Q4 of 2020; Whenever packets were dropped between the hypso-cli and opu-service, the lost packets would be transmitted again at the end of the download. This patching was performed one dropped packet at a time, and if one of those patched packets were not sent correctly, then the system would wait for the timeout to expire before attempting the next. This took a long time, and the OPU were unreachable in the timeout period. This could have lead to exceeding the power budget of the HYPSO-1 if this had happened in-flight, leading to a potential mission failure. But not only that, if this bug persisted it would take tremendous long time to upload a patch to the payload to fix this issue, leading to valuable mission time being wasted (Marton 2020).

5.4.1 hypso-cli not Working over NNG

After the implementation of the NNG into the PHiL hypso-cli were unable to establish contact with the OPU using NNG.

After some investigation, made more difficult by the issue in section 5.2, the root of the issue were identified to be the permissions of the computer running the Jenkins server, as well as the

way the NNG address were passed to the hypso-cli. The hypso-cli has a flag `-nng` that will set the address to the string following the flag, but if that flag is left empty the default address is chosen. This default address had also changed.

The first part of the issue were quickly dealt with by Roger Birkeland and the second part were implemented by me as soon as the issue were identified, but the team made a GitHub issue to show progress in the sprint; `Connect Jenkins to LidSat (and FlatSat) hardware -- update NNG permissions`.

5.4.2 Upload and Download not Working over NNG

After the previous issue the `opu_upload` and `opu_download` still did not make contact with the NNG. It is worth mentioning this happened before the changes made in section 5.1.1 had been fully implemented.

The issue turned out to be rooted in the arguments passed when starting hypso-cli as the way the script worked at the time were by directly using hypso-cli in the Jenkinsfile to ping the opu-service over NNG. These pings connected, and so did the `hsi_get_temp.py` that were using the common source files. It turned out that the issue was that the connection settings were not actually imported in the faulty scripts.

This issue were addressed by me by importing the connection settings file.

5.4.3 Upload over NNG Unable to Upload Large Files to OPU

After the previous issue were addressed the `opu_upload` test still failed. It seemed as if the opu-service just stopped responding to the upload, as it never responded back to the hypso-cli that there were an error or that the upload were complete. This issue only happened by uploading large files, and perplexed the team quite a bit.

Most likely the root of this issue is that the opu-service got flooded with messages faster than it could handle, as the MCS can handle higher speeds than the OPU, making it go out of synchronization. This is a valuable lesson, as it means that the team have to throttle the NNG connection to the MCS to ensure this does not happen in the future, which makes sense, the bandwidth from the MCS to the satellite bus will have be throttled anyways, as team had assumed this was fully handled by the MCS.

Having identified the problem, and a probable root of the problem, a issue was created in the GitHub board in order for the HYPSO team to assign during the next sprint; `opu_upload does not exit cleanly (throws an error/timeout) when run through NNG and uploading large files`

5.5 Software for Mechanical Testing

In order to test the Qualification Model of the system, the team had created a manual setup of using a set of Power Supply Unit (PSU)s to enable and disable power to certain subsystems of the satellite. These subsystems are the OPU (requires 7.5 V), the HSI (requires 12 V) and the RGB (requires 5 V). This temporary setup worked as intended, but the use of it had some drawbacks;

- Whenever the power of a subsystem had to be switched on/of it had to be performed physically, requiring a person to be present in the room with the satellite during testing.
- Having a team-member turning power on/of introduces human error, and makes it difficult to ensure changes happens at exact times.
- It can be incredibly difficult to reproduce issues that appear during the change of the power state of the system.

This resulted in the need for the qualification model to have an automated way of for managing power. In order to automate this task the team ordered two *Rs PRO SPD3303C Programmable DC Power Supply* units, one to use as the main power supply, and one as a backup. The specifications of the PSUs can be found here (RS PRO n.d.). One of the deciding factors of selecting this model was that it has 3 channels, two of which are programmable, while the last one is limited to a set of predefined voltages. This means that the subsystems can be turned on/off remotely and independently at will.

5.5.1 Implementation

The digital PSUs were programmed in python3 using the `PyVISA` library, as it has functionality for the USBTMC protocol the PSUs use. The way the program works is by having two threads, one for the messaging, and one for the GUI. The GUI is implemented using the python library `curses` and works in both Windows and Linux. Although the GUI is nice and simplistic, it is not needed for the operation of the PSUs, as the script can be run from command-line to allow for scripting the testing, for example using Jenkins. See appendix C.3 for the implementation.

However, it soon became apparent that there had been an oversight when selecting the PSUs, as the third channel could not be switched on/off remotely. When buying the PSUs the team had assumed that although selecting the voltage was manual the output could be controlled remotely, it could not. This meant that the team had to use both PSUs to do the work the one unit were supposed to be used for, and it was good that the team had ordered an extra. This could have increased the workload, but by implementing the PSU as a class in pyhton, a list of PSUs could be spawned and managed with the only limit being the maximum amount of connections

Figure 5.1: The PSUs in use during testing.

the USB-protocol allows for; 127. By implementing it this way it the system can quickly be scaled if required, which can be useful for the HYPSO-2 smallsat.

The program automatically detects all PSUs connected to the computer, and will show them in the GUI if the `PSU_GUI.py` script is run, or the script can be run from command line using `PSU_connection.py`. In fig. 5.1 and fig. 5.2 the physical and digital representation of the PSUs can bee seen, notice how the values are the same on both. See appendix C.3.1 for the different commands that can be issued to the script, either from command-line or the GUI.

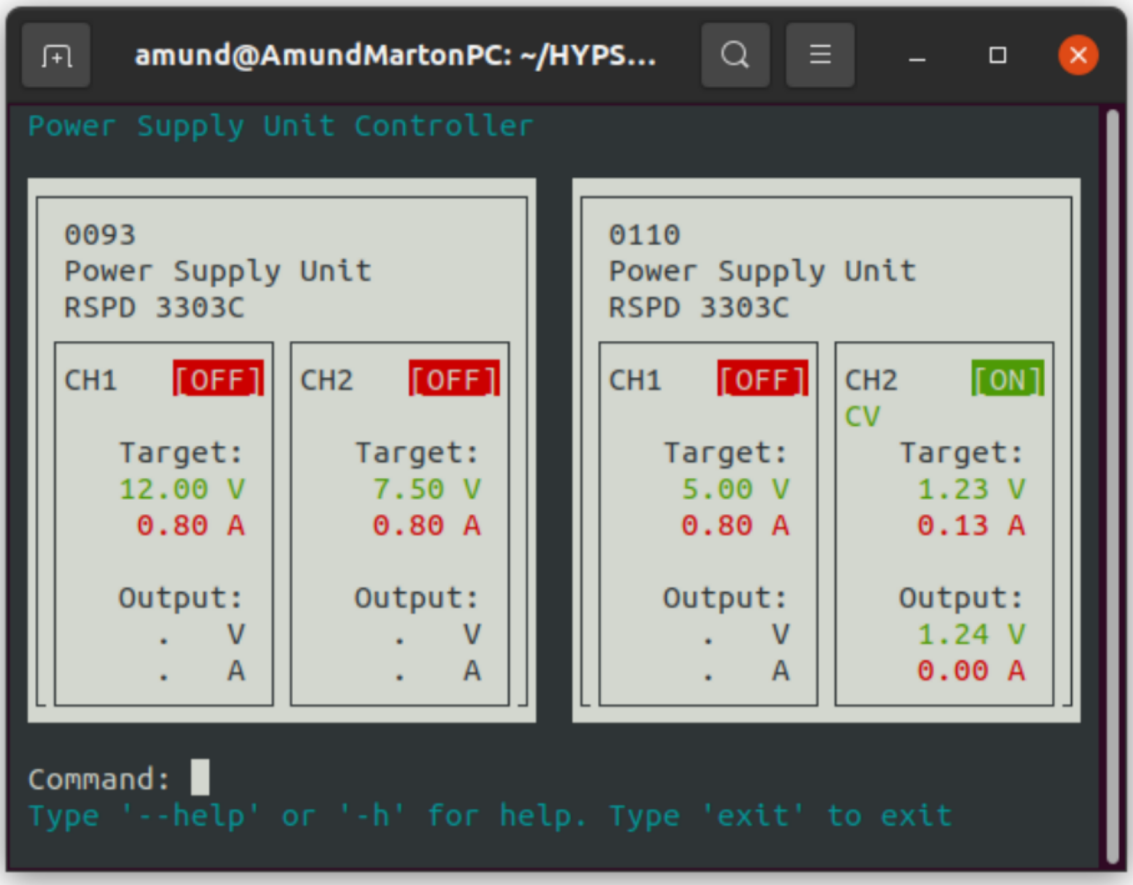


Figure 5.2: The digital representations of the PSUs during testing

Chapter 6

Discussion

As a disclaimer I want to point out that the testing performed and implemented during this thesis is nowhere extensive enough to qualify as acceptance testing; there is too many testing suites that are yet to be fully implemented. Indeed, testing in itself is not a perfect science, you are only testing for issues you can predict or expect, so a thorough Software Verification and Validation Plan is needed in combination with the PHiL testing, see (Orlandic 2019) for the HYPSO Software Verification and Validation Plan.

As the previous chapters have mentioned the testing of satellite software is especially crucial due to the nature of space missions and the lack of physical maintenance that can be performed once they are launched. This is extra critical when it comes to the communications. The Cubesat Space Protocol (CSP) creates a robust internal communication between the subsystems of HYPSO-1, but the communications between the space segment of the mission and the ground control — using the NNG, UHF and S-Band — must be thoroughly tested.

By extensively testing the communications before launch the team are able to alter the software at a later date if need be, but neglecting it can lead to mission failure. This can be seen in that the testing performed in this thesis, chapter 5, led to the identification of issues that could, at worst, have been mission critical.

Regression testing is a good tool for testing the development of software, as the tests can be run frequent and autonomously using a testing server such as Jenkins. By doing this we enables straight-forward testing on many systems, and the ability to re-use testing code. The fact that regression testing tests if the software does not lose functionality between each new version means that the development can only go in a positive direction, not regress. This, combined with the use of GitHub repositories, makes it easier for multiple developers of the HYPSO team to work on the hypso-sw, as all functionality must work before it is accepted.

A drawback of mainly doing regression testing is that it makes it easy to proclaim that once a program passes the test, it is working as intended. But that is not necessarily the case, the behaviour might comply to the expected results, but that might just be because the specific inputs fed into the system are within a range giving expected results. During flight the inputs of the satellite might be completely different to the simulated inputs and cause unforeseen issues. This is the reason why the PHiL testing is performed at HYPSO, as the system runs on the same hardware as used in the satellite, making the testing of the system as close to the in-flight system as possible. The issue then is to fed realistic and challenging inputs to the PHiL to test if the software can handle it. However, this does not eliminate the problem of creating tests for the software, not tests of the software.

Take for example the test created to shut down the opu-system and then turn of the power to the OPU, `opu_shutdown.py`. When the test was initially implemented it sent a message using hypso-cli to turn the opu-system off, then it pinged the OPU; if no reply got back then the power were turned off and marked as passed. The issue with this implementation is that if for some reason the ping did not get an answer it was marked as completed, as occurred due to the missing hypso-cli described in section 5.2. There were no handling of a case where the ping fails to send the message, the response is lost or the opu-service has frozen. By communicating with the EPS we can get the power status of the system and ensure that we are actually testing that the power of the OPU is off, not just the symptoms of it, as explained in section 5.4.

By implementing a way to test the flatsat in the PHiL using the NNG connection, a layer of abstraction between the PHiL and the HYPSO-1 was removed, as the previous testing were exclusively implemented over the CAN networks. The remaining layer of abstraction is between the NanoAvionics Propagation Service & Mission Control Software and the PC. As mentioned on section 2.3 this layer can for the purposes of HYPSO-1 be abstracted away as it is handled by the satellite bus. This means that for all intents and purposes, communicating with the flatsat over the NNG network, i.e. using the NNG connection for the PHiL testing, is very similar to how the satellite communicates in-flight, the biggest difference being the faster bandwidth, as shown in section 5.4.3. If this bandwidth were to be throttled to mission-level speeds, then the testing of the PHiL using the flatsat and NNG connection could indeed be very beneficial to the team.

The many bugs and issues that has arisen during the software testing is a clear indication that the testing is working. As previously stated, the lack of bugs does not a bug-free system make, but the issues found could have threatened the mission, so it is good that they have been addressed.

Chapter 7

Conclusion

The work performed and discussed in this thesis has highlighted different issues with the HYPSO software, and those issues has been addressed or delegated to the appropriate developers. The advantages of implementing a Payload Hardware-in-Loop testing system such as the one at HYPSO is that it can both test for regression, while at the same time simulate the in-flight operation of the satellite. If I were to change one thing done in this thesis, it would be to spend less time on the Power Supply Graphical User-interface, as it turns out the team prefers the command line version, and spend that time implementing new ways to regression test and performance test the connection between hypso-cli and opu-service. Additionally, there exists a set of unit tests for the hypso-cli that is not implemented in the testing setup, and if I had the time I would like to implement those tests to be run after building hypso-cli and before the PHiL tests in the Jenkins pipeline.

The work outlined in this thesis can be used to implement testing over multiple connections, and can be built upon by other smallsat teams. Here are some tips to future smallsat teams wishing to implement a PHiL Jenkins testing setup as the one discussed in this thesis; Testing takes time, allocate enough for it and start early, it is hard to implement them when the software is already finished. You can never test enough, so make sure you test the essential parts of the system just a little bit less than enough. When writing unit and regression tests, try to get input from another teammates if you are the one that wrote the software to be tested, or you can quicly overlook something.

To summarize; the arguments and advantages presented on how the software and PHiL testing is implemented at HYPSO there is great potential for further expanding the work and for other smallsat teams to implement similar testings setups, especially focusing on, but not excluded to, the communications of the satellite.

Appendix A

Acronyms & Definitions

BOB Breakout-Board — A circuit making it easier to connect devices

CAN Controller Area Network — a communications protocol used between the subsystems of HYPSO-1.

CAN1 CAN1 — the CAN network between the PC and the satellite bus.

CAN2 CAN2 — the CAN network between the PC and the payload subsystems.

cubesat cubesat — A standardized size of smallsats

CSP Cubesat Space Protocol

EPS Electrical Power System — the system in charge of the power of the HYPSO-1

ECSS European Cooperation for Space Standardization

FC Flight Computer — The subsystem of satellite bus in charge of the positioning of the satellite.

flatsat flatsat — the components of a satellite mounted on a flat surface for easy access.

HiL Hardware in Loop

HSI Hyper Spectral Imaging — at HYPSO, this acronym is used for the payload subsystem containing the HSI camera

HYPSO HYPer-spectral Smallsat for ocean Observation

HYPSO-1 HYPSO-1— The 1st and current satellite being built and tested by the HYPSO team

HYPSO-2 HYPSO-2— The 2nd and upcoming satellite that the HYPSO team is working on

hypso-cli HYPSO client

hypso-sw HYPSO software — this includes the hypso-cli and the opu-service

IEEE Institute of Electrical and Electronics Engineers

lidsat lidsat — A flatsat used for the PHiL at the HYPSO office.

MCS Mission Control Software

NASA National Aeronautics and Space Administration

NNG nanomsg-next-gen — a solution to avoid recurring messaging problems using sockets

NTNU Norwegian University of Science and Technology

OPU On-board Processing Unit — The main computation unit of the payload. It interfaces with the payload subsystems and the PC

opu-service On-board Processing Unit service — this is the software used to interface with the payload and the payload controller

opu-system On-board Processing Unit system — this is the operating system running on the OPU

PaaS Platform as a Service

payload the payload of the system — the hardware and software on-board HYPSO-1 that is developed by the HYPSO team

PC Payload Controller — The subsystem of the satellite bus that interfaces with the payload.

PHiL Payload Hardware-in-Loop — The HiL testing setup used to test the payload.

Pico PicoZed 7030 System-On-Module board — The hardware housing the opu-system

PicoBOB PicoBOB — the PicoZed with a Breakout-board mounted on top.

PSU Power Supply Unit

QM Qualification Model

RDBMS relational database management system

RGB Red, Green & Blue — at HYPSO, this acronym is used for the payload subsystem containing the RGB camera

satellite bus the Satellite Bus — A commercial off-the-shelf "6U NanoSatellite Bus M6P" supplied by NanoAvionics, used to carry the HYPSO payload

S-Band short-band — Frequencies between 2 GHz and 4 GHz

SDR Software Defined Radio — The tertiary payload of the HYPSO-1 satellite.

smallsat Small Satellite

SQL Structured Query Language

SQL Structured Query Language

UHF Ultra High Frequencies— Frequencies between 300 MHz and 3 GHz

Appendix B

Folder Structure of Jenkins Testing Setup

On the following pages are the "scripts" folder structure of the Jenkins testing setup used at HYPSO, before and after refactoring. The three folders `extract-frames`, `fpga-modules` and `hypso-sw` are git submodules that are imported from other parts of the HYPSO project, and are built and tested using the `Jenkinsfile` and the contents of the `scripts` folder.

B.1 Common Folder Structure

```
hardware_in_loop
├── extract-frames
│   └── ..... git submodule
├── fpga-modules
│   └── ..... git submodule
├── hypso-sw
│   └── ..... git submodule
├── scripts
│   └── ..... Testing files, see appendices B.2 and B.3
├── .gitmodules
├── Jenkinsfile ..... Declaration of the Jenkins Pipeline
└── README.md
```

B.2 Folder Structure Before Refactoring

```
scripts
├── collect_data.py
├── config.py
├── database.ini
├── database_update.py
├── download.py
├── eps_output.py
├── hsi_capture.py
├── hsi_get_temp.py
├── make_folder_and_cp.py
├── opu_download.py
├── opu_upload.py
├── read_DB.py
├── regression_test_settings.py
├── simple_ping.py
├── start_docker_python.py
├── test_functions.py
├── tools.py
├── upload.py
├── verification_hsi_capture.py
├── verification_opu_download.py
├── verification_opu_upload.py
└── write_DB.py
```

B.3 Folder Structure After Refactoring

```
scripts
├── lib
│   ├── collect_data.py
│   ├── config.py
│   ├── connection_settings.py
│   ├── database_read.py ..... renamed from: read_DB.py
│   ├── database_write.py ..... renamed from: write_DB.py
│   ├── database.ini
│   ├── docker_start_python ..... renamed from: start_docker_python
│   ├── download.py
│   ├── regression_test_settings.py
│   ├── test_functions.py
│   ├── tools.py
│   ├── upload.py
│   ├── verification_hsi_capture.py
│   ├── verification_opu_download.py ..... verification rewritten
│   ├── verification_opu_partial_download.py ..... verification added
│   ├── verification_opu_partial_upload.py ..... verification added
│   └── verification_opu_upload.py ..... verification rewritten
├── database_update.py
├── eps_output.py
├── hsi_capture.py
├── hsi_get_temp.py
├── initialize_and_build.py ..... renamed from: make_folder_and_cp.py
├── opu_download.py
├── opu_exit.py
├── opu_partial_download.py ..... test added
├── opu_partial_upload.py ..... test added
├── opu_shutdown.py
├── opu_upload.py
├── opu_update.py
├── ping_all.py ..... test added
├── ping_simple.py ..... renamed from: simple_ping.py
└── ping_timeout.py ..... renamed from: timeout_ping.py
```

Appendix C

Code

In this appendix are parts of the code from the Jenkins testing setup that is referenced in the text.

C.1 The Jenkins Declarative Pipeline - Jenkinsfile

The Jenkinsfile is the file used for declaring the Jenkins Declarative Pipeline. It is written in the language Groovy, and is used for defining the stages

```
pipeline
{
  agent any
  options { disableConcurrentBuilds() }
  stages
  {
    stage('copy new hypso-cli and opu-services')
    {
      agent any
      steps
      {
        sh 'rm -rf hypso-sw'
        sshagent(credentials : ['REDACTED'])
        {
          sh 'git clone git@github.com:NTNU-smallsat-lab/hypso-sw'
        }
        dir("scripts")
        {
```

```
        sh 'python3 initialize_and_build.py'
      }

      // Stash hypso-cli so it can be used in a later workspace
      stash includes: 'hypso-sw/build/x86/hypso-cli', name: 'hypso-cli-stash'
    }
  }
  stage('make CCSDS Compression Software')
  {
    agent any
    steps
    {
      sshagent(credentials : ['REDACTED'])
      {
        sh 'pwd'
        sh 'ls -l'
        sh 'git checkout regression_tests'
        sh 'git fetch'
        sh 'git pull'
        sh 'git submodule update --init --recursive'
      }
      dir("fpga-modules/compression/ccsds123/SOFTWAREA")
      {
        sh 'make'
      }
    }
  }
  stage('make extract_frames')
  {
    agent any
    steps
    {
      dir("extract-frames")
      {
        sshagent(credentials : ['REDACTED'])
        {
         sh 'git checkout master'
```

```
            sh 'git fetch'
            sh 'git pull'
            sh 'git submodule update --init --recursive'
          }
          sh 'make'
        }
      }
    }
    stage('Setup CAN')
    {
      agent any
      steps
      {
        sh 'sudo ip link set can0 down'
        sh 'sudo ip link set can0 type can bitrate 1000000'
        sh 'sudo ip link set can0 up'
        script
        {
          try
          {
            sh 'echo checking vcan0'
            sh 'sudo ip link show vcan0'
          }
          catch (err)
          {
            sh 'echo no virtual can bus exists. Adding.'
            sh 'sudo ip link add vcan0 type vcan'
            sh 'sudo ip link show vcan0'
          }
        }
        sh 'sudo ip link set vcan0 up'
      }
    }
    stage('EPS Test Ping')
    {
      agent any
      steps
```

```
    {
      // Load hypso-cli in case it is not in the current workspace
      unstash 'hypso-cli-stash'
      dir("scripts")
      {
        sh 'pwd'
        sh 'python3 -u ping_simple.py 4 --timeout 30'
      }
    }
  }
  stage('Deployment')
  {
    agent any
    steps
    {
      // Load hypso-cli in case it is not in the current workspace
      unstash 'hypso-cli-stash'
      sh 'echo turning on power to OPU and HSI'
      dir("scripts")
      {
        sh 'pwd'
        sh 'python3 -u eps_output.py 7 --s 1'
        sh 'python3 -u eps_output.py 9 --s 1'
        sh 'python3 -u ping_timeout.py 12 -t 30'
        //sleep 15
        sh 'python3 -u opu_update.py'
        sh 'python3 -u opu_exit.py'
        sh 'python3 -u ping_timeout.py 12 -t 60'
      }
      echo 'Copying opu-services to zedboard eventually\n'
    }
  }
  stage('Check table in database')
  {
    agent any
    steps
    {
```

```
        dir("scripts")
        {
          sh 'pwd'
          sh 'python3 database_update.py'
        }
      }
    }
    stage('HIL Test Ping')
    {
      agent any
      steps
      {
        // Load hypso-cli in case it is not in the current workspace
        unstash 'hypso-cli-stash'
        dir("scripts")
        {
          sh 'pwd'
          // Ping OPU
          script{
            try {
              def exit_code = sh( 'python3 -u ping_all.py --timeout 30' )
            } catch(Exception e) {
              echo 'e'
              echo e.toString()
              echo exit_code
              stageResult.result = 'FAILURE'
              if (exit_code != '1')
              {
                currentBuild.result = 'FAILURE'
                unstable('Ping failed')
                echo 'first'
                echo ('first')
              } else {
                echo 'else'
                echo ('else')
              }
              if (exit_code == '1')
```

```
            {
              echo ('Ping failed')
              echo 'Ping failed'
            }
          }
        }
        sh 'python3 -u ping_simple.py 12 --timeout 30'
        // Ping the rest of the nodes to ensure they are present
        sh 'python3 -u ping_all.py --timeout 30'
      }
    }
  }
  stage('Run test hsi gettemp')
  {
    agent any
    steps
    {
      // Load hypso-cli in case it is not in the current workspace
      unstash 'hypso-cli-stash'
      //sh 'git submodule update --init --recursive'
      dir("scripts")
      {
        sh 'pwd'
        sh 'python3 hsi_get_temp.py'
      }
    }
  }
  stage('Run test opu upload')
  {
    agent any
    options {
      timeout(time: 30, unit: 'MINUTES')
    }
    steps
    {
      // Load hypso-cli in case it is not in the current workspace, and show current w
      unstash 'hypso-cli-stash'
```

```
      catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE')
      {
        dir("scripts")
        {
          sh 'python3 -u opu_upload.py'
        }
      }
    }
  }
  stage('Run test opu download')
  {
    agent any
    options {
      timeout(time: 30, unit: 'MINUTES')
    }
    steps
    {
      // Load hypso-cli in case it is not in the current workspace, and show current w
      unstash 'hypso-cli-stash'

      catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE')
      {
        dir("scripts")
        {
          //sh 'python3 -u opu_download.py'
        }
      }
    }
  }
  stage('Run test hsi capture')
  {
  agent any
  options {
    timeout(time: 30, unit: 'MINUTES')
  }
  steps
```

```
      {
        // Load hypso-cli in case it is not in the current workspace, and show current w
        unstash 'hypso-cli-stash'
        catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE')
        {
          dir("scripts")
          {
            //sh 'python3 -u hsi_capture.py'
          }
        }
      }
    }
  }
  post {
    cleanup {
      // Load hypso-cli in case it is not in the current workspace
      unstash 'hypso-cli-stash'
      dir("scripts")
      {
        sh 'pwd'
        sh 'python3 -u opu_shutdown.py'
        sh 'python3 -u eps_output.py 7 --s 0 --d 10'
        sh 'python3 -u eps_output.py 9 --s 0 --d 10'
      }
      echo 'TODO: Copying opu-services to zedboard eventually\n'
    }
  }
}
```

C.2 connection_settings.py

The common code used for switching between the different connection types.

```
#this file is to quickly switch between testing different connections.
from os import path, getcwd

#Change this to CAN, NNG or (yet to be implemented) UHF
connectionType = "NNG"
```

```
__hypsoCliPath__ = "./../hypso-sw/build/x86/hypso-cli"

if path.isfile(__hypsoCliPath__) == True:
    print("hypso-cli found at {}\nrelative to {}".format(__hypsoCliPath__, getcwd()))
else:
    print("ERROR: hypso-cli NOT found at {}\nrelative to {}".format(__hypsoCliPath__, ge

if connectionType == "NNG":
    __hypsoCliAddrCSP__ = "8"
    __hypsoCliConnectionAddress__ = "mcs-serv.hypso.ies.ntnu.no:5015=0/0 \n"
elif connectionType == "CAN":
    __hypsoCliAddrCSP__ = "15"
    __hypsoCliConnectionAddress__ = "can0=0/0 \n"
elif connectionType == "UHF":
    # Yet to be implemented, but the opiton is here
    pass
else:
    pass

hypsoCliLogin = __hypsoCliPath__ + " " + __hypsoCliAddrCSP__ + " -n" + __hypsoCliConnect
print("hypsoCliLogin using {}:\t{}".format(connectionType, hypsoCliLogin))
```

C.3 PSU Interface — Implementation

The implementation of the digital PSU units. The `PSU_Controller.py` handles the reading and writing of the PSUs as well as the parsing, while `PSU_GUI.py` creates the user-interface if desired

C.3.1 README.md — PSU

```
# psu-control-interface
# Thanks to Clay McLeod for publishing the example code for a GUI using curses
# That this curses is built upon
Python tool to control multiple RSPD 3303C PSUs using the USBTMC protocoll.

All scrupts must be run as root.


How to use the PSU_Controller.py to issue commands to the PSUs:
$ sudo python PSU_Controller.py

usage: PSU_Controller.py [-h] [--status] [--list] [--reset]
                         [--recall {1,2,3,4,5}] [--save {1,2,3,4,5}]
                         [--voltage SET_VOLTAGE] [--current SET_CURRENT]
                         [--power SET_POWER] [--device PSU]
                         [--channel CHANNEL]

This script is used toconnect to one or multiple RSPD 3303C Power Suply Unit
through USBTMC.Must be run as root

optional arguments:
  -h, --help            show this help message and exit
  --status, -s          Display the current status of the connected PSUs
  --list, -l            Display the currently connected PSUs
  --reset, -r           Reset the given device or a channel on the device if
                        provided
  --recall {1,2,3,4,5}, -R {1,2,3,4,5}
                        Recall a saved set of values from the device, [1 - 5]
  --save {1,2,3,4,5}, -S {1,2,3,4,5}
                        Recall a saved set of values from the device, [1 - 5]
  --voltage SET_VOLTAGE, -v SET_VOLTAGE
```

```
                        Set the Voltage of a PSU channel, [0.0 - 32.0] V
  --current SET_CURRENT, -a SET_CURRENT
                        Set the Current of a PSU channel, [0.00 - 3.20] A
  --power SET_POWER, -p SET_POWER
                        Set the Power status channel, ["ON"/"OFF"]
  --device PSU, -d PSU  The PSU device to select in order to change values
  --channel CHANNEL, -c CHANNEL
                        The Channel on the selected PSU device


Or you can use the GUI! The commands are the same as above,
    so "-d 0110 -R 1" will recall saveslot 1 on PSU 0110
$ sudo python PSU_GUI.py
```

C.3.2 PSU_Controller.py

```
import pyvisa
import time
import sys
import os
import argparse
import logging

MSG_DELAY = 0.03
MUTE_PRINT = False

if MUTE_PRINT:
    sys.stdout = open(os.devnull, 'a')

#Helper function to send a message over USBTMC
def write_USBTMC(instrument, message, delay=MSG_DELAY):
    instrument.write(message)
    time.sleep(delay)

#Helper function to send AND recieve a message over USBTMC
def read_USBTMC(instrument, message, delay=MSG_DELAY):
    instrument.write(message)
    time.sleep(delay)
```

```
    data = instrument.read_bytes(100, break_on_termchar='\r\n')
    data = str(data)[2:-3]
    time.sleep(MSG_DELAY)
    return data

# Class to represent the
class Channel:
    def __init__(self, parrent, ID, instrument):
        #read the current status of the PSU to avoid reseting it when starting
        self.parrent = parrent
        #self.instrument = instrument
        self.ID = ID
        #self.powerStatus, self.CVCC  =
        self.powerStatus = "INIT"
        self.powerStatus, self.CVCC = self.get_status(instrument)
        self.outputVoltage = self.get_output_voltage(instrument)
        self.outputCurrent = self.get_output_current(instrument)
        self.targetVoltage = self.get_target_voltage(instrument)
        self.targetCurrent = self.get_target_current(instrument)
    def reset(self, instrument):
        self.set(instrument, 0, 0 , "OFF")
    def set(self, instrument, voltage, current, powerStatus):
        self.set_target_voltage(instrument, voltage)
        self.set_target_current(instrument, current)
        self.set_power_status(instrument, powerStatus)
    def print(self):
        print(self.parrent+'-'+self.ID+':\t'+self.powerStatus)
        print(
            "Target\t"+str(self.targetVoltage) +'V\t'
            + str(self.targetCurrent) + 'A\t'
        )
        print(
            "Output\t"+str(self.outputVoltage) +'V\t'
            + str(self.outputCurrent) + 'A\t'
         )
    def poll(self, instrument):
        self.get_target_voltage(instrument)
```

```
        self.get_target_current(instrument)
        self.get_status(instrument)
        self.get_output_voltage(instrument)
        self.get_output_current(instrument)
    def get_target_voltage(self, instrument):
        data = float(read_USBTMC(instrument, self.ID+':VOLTage?'))
        self.targetVoltage = data
        return data
    def get_target_current(self, instrument):
        data = float(read_USBTMC(instrument, self.ID+':CURRent?'))
        self.targetCurrent = data
        return data
    def get_output_voltage(self, instrument):
        if self.powerStatus == "ON":
            #only poll the output voltage and current if the channel is ON
            data = float(read_USBTMC(instrument, 'MEASure:VOLTage? '+ self.ID))
        else:
            data = 0
        self.outputVoltage = data
        return data
    def get_output_current(self, instrument):
        if self.powerStatus == "ON":
            #only poll the output voltage and current if the channel is ON
            data = float(read_USBTMC(instrument, 'MEASure:CURRent? '+ self.ID))
        else:
            data = 0
        self.outputCurrent = data
        return data

    def set_target_voltage(self, instrument, voltage):
        if voltage < 0:
            print(
                "error: you are trying to set a negative voltage."
                + " Defaulted to 0 [V]"
            )
            voltage = 0
        if voltage > 32:
```

```
            print(
                "error: you are trying to set a voltage "
                + "over 32 V. Defaulted to 32 [V]"
            )
            voltage = 32
        self.targetVoltage = voltage
        write_USBTMC(instrument, self.ID+':VOLTage ' +str(voltage))

    def set_target_current(self, instrument, current):
        if current < 0:
            print(
                "error: you are trying to set a negative current."
                +" Defaulted to 0 [A]"
            )
            current = 0
        if current > 3.2:
            print("error: you are trying to set a current over 3.2 A."
            +" Defaulted to 3.2 [A]")
            current = 3.2
        self.targetCurrent = current
        write_USBTMC(instrument, self.ID+':CURRent ' +str(current))

    def set_power_status(self, instrument, powerStatus):
        if powerStatus == "ON" or powerStatus == "OFF":
            self.powerStatus = powerStatus
            write_USBTMC(instrument, 'OUTPut '+self.ID+", "+powerStatus)
        else:
            print(
                "error: you are trying to set a powerStatus that "
                + "does not exist. [ON/OFF]"
            )

    def get_status(self, instrument):
        # Function to get the CC/CV and the status of the PSU
        powerStatus = "??"
        CVCC = "??"
        data = read_USBTMC(instrument, 'SYSTem:STATus?')
```

```
        if (self.ID == "CH1"):
            if ((0x0010 & int(data,16)) > 0):
                powerStatus = "ON"
            else:
                powerStatus = "OFF"
            if (self.powerStatus == "ON"):
                if ((0x0001 & int(data,16)) > 0):
                    CVCC = "CC"
                else:
                    CVCC = "CV"
            else:
                CVCC = "--"
        if (self.ID == "CH2"):
            if ((0x0020 & int(data,16)) > 0):
                powerStatus = "ON"
            else:
                powerStatus = "OFF"
            if (self.powerStatus == "ON"):
                if ((0x0002 & int(data,16)) > 0):
                    CVCC = "CC"
                else:
                    CVCC = "CV"
            else:
                CVCC = "--"
        self.powerStatus = powerStatus
        self.CVCC = CVCC
        return powerStatus, CVCC



# Class used to represent each PowerSupply
class PowerSupply:
    def __init__(self, ID, instrument):
        #self.instrument = instrument
        self.debug = "Nothing yet"
        self.ID = ID
```

```
        self.CH1 = Channel(ID[-4:], "CH1", instrument)
        self.CH2 = Channel(ID[-4:], "CH2", instrument)
        self.channels = {}
        self.channels["CH1"] = Channel(ID[-4:], "CH1", instrument)
        self.channels["CH2"] = Channel(ID[-4:], "CH2", instrument)
        self.alias = ID[-4:]
    def print(self, channel=True):
        print(self.alias+'\t'+self.ID)
        if (channel == True):
            for c in self.channels:
                self.channels[c].print()
        print("-------")
    def poll(self, instrument):
        for c in self.channels:
            self.channels[c].poll(instrument)
    def reset(self, instrument):
        # Function to set all channels to zero/off
        for c in self.channels:
            self.channels[c].reset(instrument)
    def recall(self, instrument, slot):
        # Function to recall saved values
        write_USBTMC(instrument, '*RCL ' + str(slot))
    def save(self, instrument, slot):
        # Function to save the current values of the PSU
        write_USBTMC(instrument, '*SAV ' + str(slot))

def get_PSUs():
    print("Fetching the devices")
    rm = pyvisa.ResourceManager()
    devices = rm.list_resources()
    print("Devices: "+str(devices))
    PSU_dict = {}
    instruments = {}

    print("->\nInitializing the devices")
    for d in devices:
        try:
```

```
            instrument = rm.open_resource(d)
            data = read_USBTMC(instrument, '*IDN?', MSG_DELAY)
            temp_ID = str(data).split(',')[0][4:]
            instruments[temp_ID[-4:]] = instrument
            PSU_dict[temp_ID[-4:]] = PowerSupply(temp_ID, instrument)
        except Exception as e:
            raise e
    #return the list of the power supply units, and the list of devices
    return PSU_dict, devices, instruments

def init_string_parser():
    customParser = argparse.ArgumentParser(description='This script is used to' +
        'connect to one or multiple RSPD 3303C Power Suply Unit through USBTMC.' +
        'Must be run as root')

    customParser.add_argument('--status'  ,'-s',
        dest='request_status',
        action='store_true',
        help='Display the current status of the connected PSUs')

    customParser.add_argument('--list'  ,'-l',
        dest='request_list',
        action='store_true',
        help='Display the currently connected PSUs')

    customParser.add_argument('--reset'  ,'-r',
        dest='reset',
        action='store_true',
        help='Reset the given device or a channel on the device if provided')

    customParser.add_argument('--recall' ,'-R',
        type=int,
        choices=range(1, 6),
        dest='recall_value',
        help='Recall a saved set of values from the device, [1 - 5]')

    customParser.add_argument('--save' ,'-S',
```

```
        type=int,
        choices=range(1, 6),
        dest='save_value',
        help='Recall a saved set of values from the device, [1 - 5]')

    customParser.add_argument('--voltage' ,'-v',
        type=float,
        dest='set_voltage',
        help='Set the Voltage of a PSU channel, [0.0 - 32.0] V')

    customParser.add_argument('--current' ,'-a',
        type=float,
        dest='set_current',
        help='Set the Current of a PSU channel, [0.00 - 3.20] A')

    customParser.add_argument('--power'   ,'-p',
        type=str,
        dest='set_power',
        help='Set the Power status channel, ["ON"/"OFF"]')

    customParser.add_argument('--device'  ,'-d',
        type=str,
        dest='psu',
        help='The PSU device to select in order to change values')

    customParser.add_argument('--channel' ,'-c',
        type=str,
        dest='channel',
        help='The Channel on the selected PSU device')

    return customParser

def parser_error_handling(message):
    if __name__ == '__main__':
        pass
        #parser.error(message)
    return message
```

```
def parse_list(list):
    pared_list = customParser.parse_args(list)
    return pared_list


def handle_parsed_string(PSU_dict, instruments, args):
    logging.info(args)
    # If invalid device is given
    if (args.psu is not None and isinstance(PSU_dict, dict)):
        if args.psu not in PSU_dict.keys():
            return parser_error_handling(
                "--psu is an invalid device. "
                + "List off available devices: {}".format(PSU_dict.keys())
            )

    # If invalid channel is given
    if (args.channel is not None):
        if (args.channel != "1" and args.channel != "2"):
            return parser_error_handling("--channel must be either 1 or 2")

    # Check that all required arguments for setting voltage
    if (args.set_voltage is not None):
        if (args.set_voltage >= 0.00 and args.set_voltage <= 32.0):
            if args.psu is None:
                return parser_error_handling(
                    "--voltage requires --psu, "
                    + "or the psu is not found "
                )
            if args.channel is None:
                return parser_error_handling("--voltage requires --channel")
            # Set voltage in device
            PSU_dict[args.psu].channels["CH"+args.channel].set_target_voltage(
                instruments[args.psu],
                args.set_voltage
            )
        else:
```

```
            return parser_error_handling(
                "--voltage must be in the range"
                + " [0.00 - 32.0]"
            )

    # Check that all required arguments for setting current
    if (args.set_current is not None):
        if (args.set_current >= 0.00 and args.set_current <= 3.20):
            if args.psu is None:
                return parser_error_handling(
                    "--current requires --psu,"
                    + " or the psu is not found "
                )
            if args.channel is None:
                return parser_error_handling("--current requires --channel")
            # Set current in device
            PSU_dict[args.psu].channels["CH"+args.channel].set_target_current(
                instruments[args.psu],
                args.set_current
            )
        else:
            return parser_error_handling(
                "--current must be in the"
                + " range [0.00 - 3.20]"
            )

    # Check all required arguments for toggling power
    if (args.set_power is not None):
        if (args.set_power.upper() == "ON" or args.set_power.upper() == "OFF"):
            if args.psu is None:
                return parser_error_handling("--power requires --psu")
            if args.channel is None:
                return parser_error_handling("--power requires --channel")
            # Toggle power in device
            PSU_dict[args.psu].channels["CH"+args.channel].set_power_status(
                instruments[args.psu],
                args.set_power
```

```
            )
        else:
            return parser_error_handling("--power must be \"ON\" or \"OFF\"")

    # Check all required arguments for toggling power
    if (args.reset is True):
        if args.psu is None:
            return parser_error_handling("--reset requires --psu")
        if args.channel is not None:
            # Reset the channel
            PSU_dict[args.psu].channels["CH"+args.channel].set(
                instruments[args.psu],
                0,
                0,
                "OFF"
            )
        else:
            for c in PSU_dict[args.psu].channels:
                PSU_dict[args.psu].channels[c].set(
                    instruments[args.psu],
                    0,
                    0,
                    "OFF"
                )

    if args.recall_value is not None:
        PSU_dict[args.psu].recall(instruments[args.psu], args.recall_value)

    if args.save_value is not None:
        PSU_dict[args.psu].save(instruments[args.psu], args.save_value)

    # Print the status of the different devices
    if args.request_status == True:
        for d in PSU_dict:
            PSU_dict[d].print()

    # Print the connected devices if the --list argument is given,
```

```
    # or there is no arguments
    if args.request_list == True or len(sys.argv) == 1:
        print("Connected PSU devices")
        for d in PSU_dict:
            print(d)
    return "ok"

# Always initialize the parser
customParser = init_string_parser();

def main():
    # This function will only run if it is not inclued as a library

    # parse the args
    args = customParser.parse_args();
    PSU_dict, devices, instruments = get_PSUs()

    handle_parsed_string(PSU_dict, instruments, args)

if __name__ == '__main__':
    # Only run main if this is the top level script
    main()
```

C.3.3 PSU_GUI.py

```
# Thanks to Clay McLeod for publishing the example code for a GUI using curses
# That this curses is built upon
import sys,os
import curses
import time
import logging
import multiprocessing
import PSU_Controller

def ChannelWindow(stdscr,height,width,y_offset,x_offset,CH_N):
    # This is a curses box for each channel in a PSU
    chanwin = stdscr.subwin(height,width, y_offset, x_offset)
    chanwin.box()
```

```
    chanwin.addstr(1,1,CH_N.ID, curses.color_pair(3))

    chanwin.addstr(7,1,'Output:'.rjust(width-3), curses.color_pair(3))
    if CH_N.power_status == "ON":
        if CH_N.CVCC == "CC":
            chanwin.addstr(2,1,CH_N.CVCC, curses.color_pair(4))
        elif CH_N.CVCC == "CV":
            chanwin.addstr(2,1,CH_N.CVCC, curses.color_pair(5))
        else:
            chanwin.addstr(2,1,"--", curses.color_pair(3))

        chanwin.addstr(1,8,"[ON]", curses.color_pair(7))
        chanwin.addstr(8,1,
            "{:.2f} V".format(CH_N.output_voltage).rjust(width-3),
            curses.color_pair(5)
        )
        chanwin.addstr(9,1,
            "{:.2f} A".format(CH_N.output_current).rjust(width-3),
            curses.color_pair(4)
        )
    elif CH_N.power_status == "OFF":
        chanwin.addstr(1,7,"[OFF]", curses.color_pair(6))
        chanwin.addstr(8,1,'  .   V'.rjust(width-3), curses.color_pair(3))
        chanwin.addstr(9,1,'  .   A'.rjust(width-3), curses.color_pair(3))
    else:
        chanwin.addstr(1,9,"  [?]", curses.color_pair(3))
    chanwin.addstr(3,1,'Target:'.rjust(width-3), curses.color_pair(3))
    chanwin.addstr(4,1,
        "{:.2f} V".format(CH_N.target_voltage).rjust(width-3),
        curses.color_pair(5)
        )
    chanwin.addstr(5,1,
        "{:.2f} A".format(CH_N.target_current).rjust(width-3),
        curses.color_pair(4)
    )
    chanwin.bkgd(' ', curses.color_pair(3))
    chanwin.refresh()
```

```
def PowerSupplyWindow(stdscr,height,width,y_offset,x_offset,PSU):
    # This is a curses box for each PSU
    psuwin = stdscr.subwin(height, width -2, y_offset, x_offset+1)
    psuwin.box()
    psuwin.addstr(1,2,PSU.alias, curses.color_pair(3))
    psuwin.addstr(2,2,"Power Supply Unit", curses.color_pair(3))
    psuwin.addstr(3,2,"RSPD 3303C", curses.color_pair(3))

    psuwin.bkgd(' ', curses.color_pair(3))
    #ChannelWindow(stdscr,20,40,2,0,"PSU_ID")
    ChannelWindow(
        stdscr,
        height-4,
        int((width-4)/2),
        4+y_offset,
        2+x_offset,
        PSU.channels["CH1"]
    )
    ChannelWindow(
        stdscr,
        height-4,
        int((width-4)/2),
        4+y_offset,
        x_offset+int(width/2),
        PSU.channels["CH2"]
    )
    psuwin.refresh()

def polling_function(
        PSU_shared_dict,
        state,
        exit_condition,
        command_queue,
        text_feedback
    ):
    # Get the PSUs
```

```
    PSU_dict_tmp, devices, instruments = PSU_Controller.get_PSUs()

    # initialize the variables
    tmpstate = state
    for d in PSU_dict_tmp.keys():
        PSU_shared_dict[d] = PSU_dict_tmp[d]
    logging.info(devices)

    # Errorhandling
    if len(devices) == 0:

        logging.info(
            "ERROR - No power supply found\nNo power supply found."
            + " \nHave you remembered to run this command as root?"
            + " \nHave you remembered to turn the power on?"
            + " \nAlternatively you can try the command:"
            + " \n$ lsusb\nAnd look if the power supply show up"
        )
        tmpstate[0] = "ERROR"
        tmpstate[1] = "No Power Supply Found"
        state = tmpstate
        exit_condition.value = True
    else:
        tmpstate[0] = "IDLE"
        tmpstate[1] = "Ready to poll / send"
        logging.info("Initialization of devices complete.\nReady to poll / send")
        state = tmpstate

    # Start polling
    while (exit_condition.value == False):
        if (state[0] == "POLL"):
            # If there is a command to send, send it
            if (len(command_queue) > 0):
                # get command and update command queue
                tmp_command_queue = command_queue
                command = tmp_command_queue.pop()
                command_queue = tmp_command_queue
```

```
                logging.info("command: {}".format(command))

                # Split command into a list
                split_command = command.split(" ")
                logging.info("split_command: {}".format(split_command))
                # Parse the command
                parsed_command = PSU_Controller.parse_list(split_command)
                logging.info("parsed_command: {}".format(parsed_command))

                # handle the command
                PSU_Controller.handle_parsed_string(
                    PSU_shared_dict,
                    instruments,
                    parsed_command
                )
            else:
                # if there is no command, continue polling
                for d in PSU_dict_tmp.keys():
                    tmpstate = state
                    for c in PSU_dict_tmp[d].channels:
                        tmpstate[1] = ("Polling " + PSU_dict_tmp[d].alias
                            + "-" + PSU_dict_tmp[d].channels[c].ID)
                        state = tmpstate
                        PSU_dict_tmp[d].channels[c].poll(instruments[d])
                        PSU_shared_dict[d] = PSU_dict_tmp[d]
                        state = tmpstate
                #update the state
                tmpstate[0] = "IDLE"
                tmpstate[1] = "Finished POLLing"
                state = tmpstate
                #info.logging(tmpstate)
        else:
            pass

    # When this thread stops the other thread must stop
    logging.info("Turning off the POLLing function")
    exit_condition.value = True
```

```
def queue_command(command, exit_condition, command_queue):
    logging.info("Command entered: %s", command)
    if command.upper()[0:4] == "EXIT":
        exit_condition.value = True

    # Get a local copy of queue
    tmp_command_queue = command_queue

    # Add to queue
    tmp_command_queue.append(command)

    # Update queue
    if command_queue != tmp_command_queue:
        command_queue = tmp_command_queue

def main():
    # initialize the logging
    logging.basicConfig(
        filename='PowerSupplyGUI.log',
        level=logging.INFO,
        format='%(asctime)s %(message)s'
    )
    logging.info('Started %s',os.path.basename(__file__))

    #i nitialize manager for synchronizing variables between threads
    manager = multiprocessing.Manager()
    PSU_shared_dict = manager.dict()
    command_queue = manager.list()
    state = manager.list(["INIT", ""])
    exit_condition = multiprocessing.Value('b', False)
    text_feedback = multiprocessing.Array(
        'c', b"Type \'--help\' or \'-h\' "
        + " for help. Type \'exit\' to exit"
    )

    # Start the pricessing
```

```
    POLL_process = multiprocessing.Process(
        name="POLL",
        target=polling_function,
        args=(
            PSU_shared_dict,
            state,
            exit_condition,
            command_queue,
            text_feedback,
        ),
        daemon=True
    )
    GUI_process = multiprocessing.Process(
        name="GUI",
        target=curses.wrapper,
        args=(
            draw_menu,
            PSU_shared_dict,
            state,
            exit_condition,
            command_queue,
            text_feedback,
        ),
        daemon=True
    )

    GUI_process.start()
    logging.info('The POLL_process is started')

    POLL_process.start()
    logging.info('The GUI_rocess is started')
    POLL_process.join()
    logging.info('The POLL_process is joined')
    print("POLL_process stopped")
    GUI_process.join()
    logging.info('The GUI_rocess is  joined')
    print("GUI_process stopped")
```

```
    logging.info('exit')

def draw_menu(
        stdscr,
        PSU_shared_dict,
        state,
        exit_condition,
        command_queue,
        text_feedback
    ):
    timeout = 0
    # wait for the Initialization to complete
    while (state[0] == "INIT"):
        timeout = timeout + 1
        time.sleep(1)
    if (state[0] == "ERROR"):
        exit()
    # Get local copy of the shared dictonary
    PSUs = PSU_shared_dict

    # ensure the terminal window is large enough do display the PSUs
    height, width = stdscr.getmaxyx()
    if (height < 20 or (width < (30 * len(PSUs)))):
        exit_condition.value = True
        print("The terminal window is too small to show the PSUs!")
        time.sleep(10)

    tmpstate = state
    k = 0

    # Start colors in curses
    curses.start_color()
    curses.init_pair(1, curses.COLOR_CYAN, curses.COLOR_BLACK)
    curses.init_pair(2, curses.COLOR_RED, curses.COLOR_BLACK)
    curses.init_pair(3, curses.COLOR_BLACK, curses.COLOR_WHITE)
    curses.init_pair(4, curses.COLOR_RED, curses.COLOR_WHITE)
    curses.init_pair(5, curses.COLOR_GREEN, curses.COLOR_WHITE)
```

```
    curses.init_pair(6, curses.COLOR_WHITE, curses.COLOR_RED)
    curses.init_pair(7, curses.COLOR_WHITE, curses.COLOR_GREEN)


    stdscr.timeout(100)
    tic = time.time()
    inputs = ""

    while (exit_condition.value == False):
        # Initialization
        stdscr.clear()

        # Declaration of strings
        title = "Power Supply Unit Controller"[:width-1]

        # Rendering the tittle, input and feedback text
        stdscr.addstr(0, 1, title, curses.color_pair(1))
        stdscr.addstr(19, 1, text_feedback.value, curses.color_pair(1))
        stdscr.addstr(18, 1, "Command: "+inputs)

        # Update state
        state = tmpstate

        # render the PSUs
        x_offset = 0
        for d in PSU_shared_dict.keys():
            #PSUs[d].poll()
            PowerSupplyWindow(stdscr,15,30,2,0+x_offset, PSU_shared_dict[d])
            x_offset = x_offset + 30

        # Get inputs
        k = stdscr.getch()
        if k > 0:
            if k == 263:
                inputs = inputs[:-1]
            elif k == 10:
                logging.info("inputs\t" + inputs)
```

```
                queue_command(inputs, exit_condition, command_queue)
                inputs = ""
            else:
                inputs = inputs + chr(k)


        # Refresh screen
        stdscr.refresh()
        if time.time() - tic > 0.8 * len(PSU_shared_dict) and state[0] == "IDLE":
            # it is estimated to take 0.6 seconds to poll per machine,
            # so 0.8 is just to give it a bit extra timeout
            # if it is idle for that it will start polling
            tmpstate[0] = "POLL"
            tmpstate[1] = "Polling"
            state = tmpstate
            logging.info("Setting state to POLL")
            tic = time.time()

    # When this thread stops the other thread must stop
    exit_condition.value = True

if __name__ == "__main__":
    main()
```

C.4 Implemented Tests

Due to the length of the files, only some tests are included here.

C.4.1 ping_timeout.py

```
from lib.regression_test_settings import hypsoCliLogin
from lib.tools import (
        send_system_command,
        csp_ping
)
import subprocess
import argparse
import time
import sys

parser = argparse.ArgumentParser(description='Ping a CSP node.')
parser.add_argument('node', metavar='loc', type=str,
                    help='which node to ping')
parser.add_argument('-t', type=int, default=10, help='timeout')

args = parser.parse_args()

hypso_cli = subprocess.Popen(
    [hypsoCliLogin],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT,
    universal_newlines=True,
    shell=True,
    bufsize=0)


start_time = time.time()
delta=0
while delta < args.t:
    connection = csp_ping(hypso_cli, args.node)
    print(connection)
```

```
    if connection:
        print("the booting took {}".format(time.time()-start_time))
        break
    hypso_cli.terminate()
    delta=time.time()-start_time
    print(delta, args.t)

    hypso_cli = subprocess.Popen(
                [hypsoCliLogin],
                stdin=subprocess.PIPE,
                stdout=subprocess.PIPE,
                stderr=subprocess.STDOUT,
                universal_newlines=True,
                shell=True,
                bufsize=0)

if connection == True:
    sys.exit(0)
else:
    sys.exit(-1)
```

C.4.2 ping_simple.py

```
from lib.regression_test_settings import hypsoCliLogin
from lib.tools import (
        send_system_command,
        csp_ping
)
import subprocess
import argparse
import sys

parser = argparse.ArgumentParser(description='Ping a CSP node.')
parser.add_argument('node',  metavar='loc', type=str,
                    help='which node to ping')
parser.add_argument('--timeout','-t', metavar='timeout', type=int,
                    help='how long to wait for the ping', default=2 )
args = parser.parse_args()
```

```
print("Pinging node {}, with a max timeout of {} seconds".format(
        args.node,
        args.timeout
))
hypso_cli = subprocess.Popen(
    [hypsoCliLogin],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT,
    universal_newlines=True,
    shell=True,
    bufsize=0)

result = csp_ping(hypso_cli, args.node, delay=args.timeout)
if result == True:
    sys.exit(0)
else:
    sys.exit(-1)
```

C.4.3 ping_all.py

```
from lib.regression_test_settings import hypsoCliLogin
from lib.tools import (
        send_system_command,
        csp_ping
)
import subprocess
import argparse
import sys

parser = argparse.ArgumentParser(description='Ping a CSP node.')

parser.add_argument('--timeout','-t', metavar='timeout', type=int,
                    help='how long to wait for the ping', default=30 )
args = parser.parse_args()

print("Pinging all nodes, with a max timeout of {} seconds".format(
```

```
        args.node,
        args.timeout
))
hypso_cli = subprocess.Popen(
    [hypsoCliLogin],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT,
    universal_newlines=True,
    shell=True,
    bufsize=0)

result = csp_ping_all(hypso_cli, delay=args.timeout)
print("The result of the test:", result)
if result == "ALL":
    sys.exit(0)
elif result == "SOME":
    sys.exit(1)
elif result == "NONE":
    sys.exit(-1)
elif result == "TIMEOUT":
    sys.exit(-2)
else:
    # If there is an error or the result == "ERROR"
    sys.exit(-3)
```

C.4.4 tools.py

```
import time
import subprocess
import os
import datetime
import numpy as np
import re

from os import path
from PIL import Image
from lib.database_write import update_data
```

```
from lib.database_read import (
        get_stored_data,
        check_table_empty
)
from lib.regression_test_settings import (
        SleepDuration,
        picobobLoginShellRemote,
        decompress,
        camera_number,
        opu_caminfo,
        hypsoCliLogin
)


# some lines of code has been removed to reduce the number of lines

def csp_ping_all(hypso_cli, delay=30, wait=0.1):
    start = time.time()
    send_command(hypso_cli, "csp ping all")
    flag="NONE"
    while True:
        lstart = time.time()
        while time.time() - lstart < wait:
            waiting = True
        line = hypso_cli.stdout.readline().strip('\n')
        if "error" in line.lower() or "not found" in line.lower():
            print(line)
            return "ERROR"
        if "Pinging node" in line:
            print(line)
        if "Ping received" in line:
            print(line)
            # Will only be encountered the first time a ping is received
            if flag == "NONE":
                flag = "ALL"
        if "Failed to ping" in line:
            print(line)
```

```
            # If there has been a ping, but not on this one, then at least one
            #   has succeeded, and one has failed, this some has succeeded
            if flag == "ALL":
                flag = "SOME"
        if time.time() - start > delay:
            print("csp_ping timed out after",delay,"seconds.")
            return "TIMEOUT"
        if "exited with return value" in line:
            duration = time.time() - start
            print("duration: {}".format(duration))
            return flag
    return False

def csp_ping(hypso_cli, id_str, delay=2, wait=0.1):
    start = time.time()
    send_command(hypso_cli, "csp ping" + id_str)
    flag=False
    while True:
        lstart = time.time()
        while time.time() - lstart < wait:
            waiting = True
        line = hypso_cli.stdout.readline().strip('\n')
        if "error" in line.lower() or "not found" in line.lower():
            print(line)
            return False
        if "Ping received" in line:
            flag = True
            duration = time.time() - start
            print("duration: {}".format(duration))
            return flag
        if "Failed to ping" in line:
            print(line)
            return False
        if time.time() - start > delay:
            print("csp_ping timed out after",delay,"seconds.")
            return flag
    return False
```

```
# the rest of the lines of code has been removed to reduce the number of lines
```

Bibliography

Anderson, C. (2015), 'Docker [Software engineering]', *IEEE software* **32**(3), 102–c3. Place: LOS ALAMITOS Publisher: IEEE.

Bacic, M. (2005), On hardware-in-the-loop simulation, *in* 'Proceedings of the 44th IEEE Conference on Decision and Control', pp. 3194–3198. ISSN: 0191-2216.

Braun, T. M. (2012), 'Satellite communications payload and system'. ISBN: 1-283-54966-2 Place: Hoboken, N.J.

Davis, A. L. (2019), *Learning Groovy 3: Java-Based Dynamic Scripting*, Apress LP, Berkeley, CA.

Di Natale, M. (2012), 'Understanding and Using the Controller Area Network Communication Protocol : Theory and Practice'. Edition: 1st ed. 2012. ISBN: 1-4614-0314-6 Place: New York, NY.

ECSS Secretariat (2012), ECSS system, Glossary of terms, Technical report, ESA-ESTEC Requirements & Standards Division, ECSS.
URL: *http://ecss.nl/get_attachment.php?file=standards/ecss-s/ECSS-S-ST-00-01C1October2012.pdf*

Garrett D'Amore (2021), 'NNG Reference Manual'.
URL: *https://staysail.tech/books/nng_reference/*

HYPSO Project Team (2019), SDR-DR-001 System Design Report, Technical report, NTNU, Internal Report. Non-Published.

HYPSO Project Team (2020*a*), HYPSO-DR-001: System Design Report, Technical report, NTNU, Internal Report. Non-Published.

HYPSO Project Team (2020*b*), HYPSO-DR-010 RGB Camera Payload, Technical report, NTNU, Internal Report. Non-Published.

HYPSO Project Team (2020*c*), HYPSO-MRD-001 Mission Requirements Document, Technical report, NTNU.

HYPSO Project Team (2020*d*), HYPSO-SRD-002 HYPSO System Requirements Document, Technical report, NTNU, Internal Report. Non-Published.

Jahren, E. R. (2015), 'Design and Implementation of a Reliable Transport Layer Protocol for NUTS', p. 6.
URL: *https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2371549/13673_FULLTEXT.pdf*

Jordheim, O. (2020), 'FMECA Review+Detection'.
URL: *Internal Document. Non-Published*

Langer, D. & Hjertenæs, M. (2020), HYPSO-DR-010 RGB Camera Payload, Technical report, NTNU, Internal Report. Non-Published.

Mabrouk, E. (2017), 'What are SmallSats and CubeSats?'.
URL: *http://www.nasa.gov/content/what-are-smallsats-and-cubesats*

Marton, A. (2020), Satellite Software Testing with Hardware andHumans in the Loop, Technical report, NTNU, Internal Report. Non-Published.

Maymala, J. (2015), *PostgreSQL for data architects : discover how to design, develop, and maintain your database application effectively with postgreSQL*, Community experience distilled, 1st edition. edn, Packt Publishing, Birmingham, England.

Myers, G. J., Badgett, T., Thomas, T. M. & Sandler, C. (2004), *The art of software testing*, Vol. 2, Wiley Online Library.

NanoAvionics (2018), 'High-Performance multi-Purpose 6U nano-Satellite Bus'.
URL: *https://nanoavionics.com/wp-content/uploads/2021/06/M6P-2021-06-online-single.pdf*

Orlandic, M. (2019), HYPSO-TPL-002 Software Verification and Validation Plan, Technical report, NTNU, Internal Report. Non-Published.

Raab, F., Caverly, R., Campbell, R., Eron, M., Hecht, J., Mediano, A., Myer, D. & Walker, J. (2002), 'HF, VHF, and UHF systems and technology', *IEEE Transactions on Microwave Theory and Techniques* **50**(3), 888–899.

RS PRO (n.d.), 'Quick Start SPD3303CProgrammable DC Power Supply'.
URL: *https://uk.rs-online.com/web/p/bench-power-supplies/1236467/*

Shaw, G. & Burke, H.-H. (2003), 'Spectral imaging for remote sensing', *The Lincoln Laboratory journal* **14**(1), 3–28.

Sigernes, F., Syrjäsuo, M., Storvold, R., Fortuna, J., Grøtte, M. E. S. & Johansen, T. A. (2018), 'Do it yourself hyperspectral imager for handheld to airborne operations'. Publisher: Optical Society of America.
URL: *http://hdl.handle.net/11250/2579680*

Simpson, D. (2015), *Extending Jenkins*, Community Experience Distilled, Packt Publishing, Birmingham, UK.
URL: *http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1134493&site=ehost-live*

Spinellis, D. (2012), 'Git', *IEEE software* **29**(3), 100–101. Place: Los Alamitos, CA Publisher: IEEE.

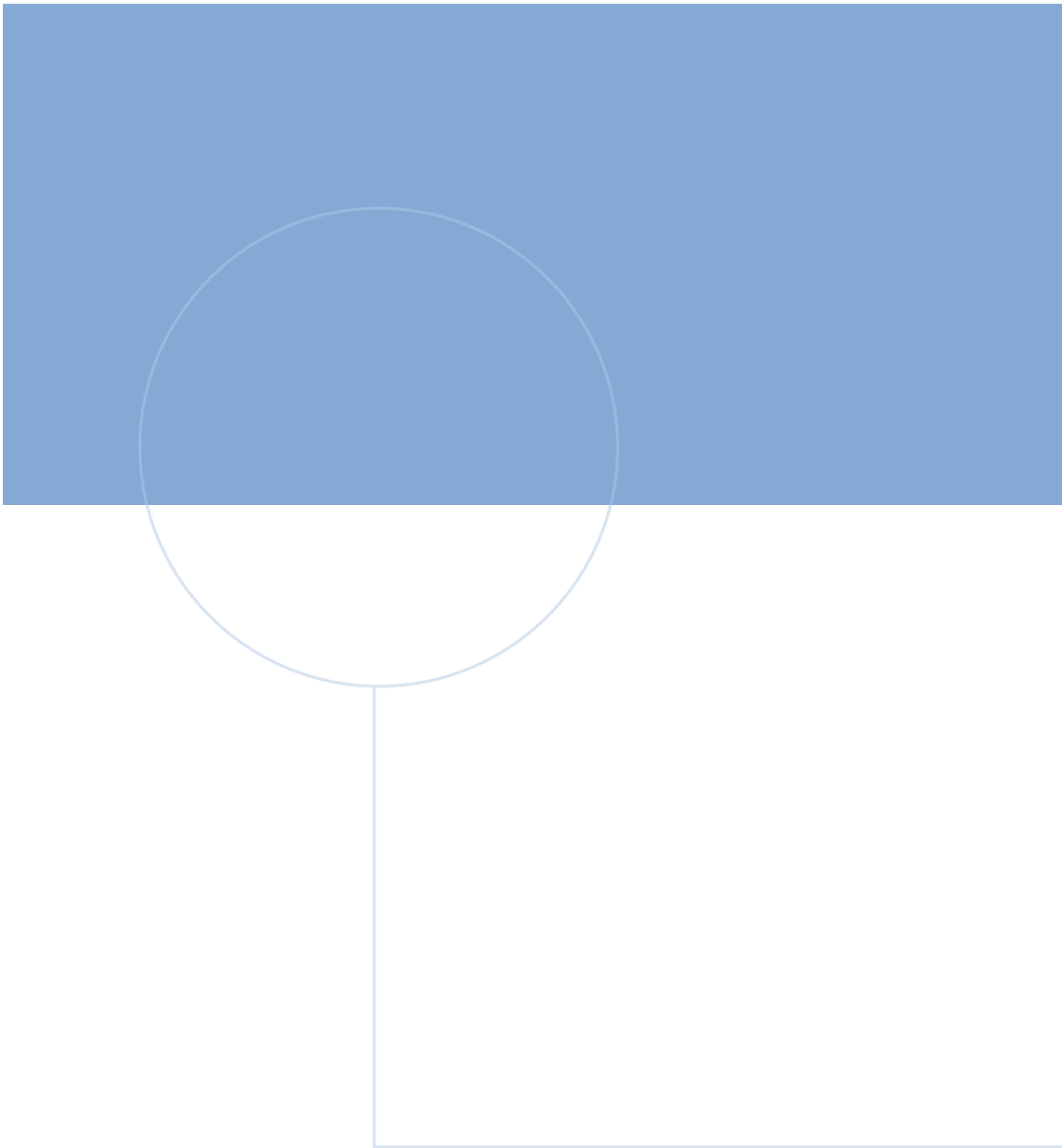
Srinivas, R., Nithyanandan, L., Umadevi, G., Rao, P. V. V. S. & Kumar, P. N. (2011), Design and implementation of S-band Multi-mission satellite positioning data simulator for IRS satellites, *in* '2011 IEEE Applied Electromagnetics Conference (AEMC)', pp. 1–4.

Stross-Radschinski, A. C., Hasecke, J. U. & Lemburg, M.-A. (2014), 'PSF Python Brochure Vol. I final Download .pdf —'.
URL: *https://brochure.getpython.info/media/releases/psf-python-brochure-vol.-i-final-download.pdf*

Tran, T. A. (2019), 'Thermal Design Analysis and Integration of a Hyperspectral Imaging Payload for a 6U CubeSat'. Accepted: 2019-10-18T14:01:47Z Publisher: NTNU.
URL: *https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2623206*

Villela, T., Costa, C. A., Brandão, A. M., Bueno, F. T. & Leonardi, R. (2019), 'Towards the Thousandth CubeSat: A Statistical Overview', *International journal of aerospace engineering* **2019**, 1–13. Publisher: Hindawi Limited, Hindawi.

Wenschel Lan (2020), 'CubeSat Design Specification'.
URL: *https://org.ntnu.no/studsat/docs/proposal_1/A8%20-%20Cubesat%20Design%20Specification.pdf*



NTNU

Norwegian University of
Science and Technology