Master's thesis

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

Magne Hov

Design and Implementation of Hardware and Software Interfaces for a Hyperspectral Payload in a Small Satellite

Master's thesis in Engineering Cybernetics
Supervisor: Tor Arne Johansen
June 2019

NTNU
Kunnskap for en bedre verden

Magne Hov

Design and Implementation of Hardware and Software Interfaces for a Hyperspectral Payload in a Small Satellite

Master's thesis in Engineering Cybernetics
Supervisor: Tor Arne Johansen
June 2019

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

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis details the design and implementation of a file transfer system for a hyperspectral imager payload on a small satellite. The payload produces large amounts of imaging data that must be downloaded to produce ocean colour forecasts and to provide researchers with hyperspectral data. A small satellite platform from NanoAvionics supports the payload with a radio link that provides connectivity through a network based on the CubeSat Space Protocol.

Several file transfer protocols are described and used as input to the design process. The satellite platform and communication architecture is detailed, and a Breakout Board for the imager payload is designed.

Mission requirements are mapped and used as input in the design process of the file transfer system. An implementation of the complete file transfer system is tested.

The implemented file transfer system is shown capable of transferring files across an unreliable network. A *store and forward* feature is used to circumvent a communication bottleneck between payload and satellite platform. This capability is demonstrated.

Sammendrag

Denne oppgaven beskriver design og implementasjon av et filoverføringssystem for en nyttelast på en liten satellitt med et hyperspektralt kamera. Nyttelasten produserer store mengder bildedata som må lastes ned for å lage havfargeprognoser, og for å forsyne forskere med hyperspektral data. En platform for småsatellitter fra NanoAvionics gir radioforbindelse til bakken gjennom et nettverk som er basert på CubeSat Space Protocol.

Flere filoverføringsprotokoller er beskrevet og brukt som grunnlag i designprosessen. Satellittplatformen og kommunikasjonsarkitekturen er presentert, og et grensesnittskort til nyttelasten har blitt designet.

Oppgragskrav er kartlagt og brukt i designprosessen til filoverføringssystemet. En komplett implementasjon av filoverføringssystemet har blitt testet.

Det blir demonstrert at det implementerte filoverføringssystemet er i stand til å overføre filer over et upålitelig nettverk. En funksjon som *lagrer og videresender* filer er brukt for å omgå en flaskehals mellom nyttelasten og satellittplatformen.

To Eva, Arne and Anna

Preface

This text is the results of a five months thesis project during the spring of 2019 following a four months specialisation project on the same topic during the autumn of 2018. The work was performed at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU).

This thesis details the design, implementation, and testing of a file transfer system for a hyperspectral imager payload that is being developed by NTNU SmallSat Lab. The inspiration to do this project stems from a three year long adventure volunteering for the student satellite project *NTNU Orbit* (formerly *NUTS*). It is my hope that the NTNU SmallSat Lab is able to use and build on the products of this thesis, ultimately achieving success with our HYPSO mission.

NTNU SmallSat Lab has provided me with a place to work, computer monitors and tools that were required to perform my tasks. They have made available the satellite subsystems that were required to develop and test my work. Cooperation between NTNU SmallSat Lab and NanoAvionics gave me access to the necessary information about the satellite subsystems. I had to seek out the documentation on the use of the network protocols, and experiment with ways of integrating them into the payload.

I would like to thank my co-supervisors Milica Orlandic and Roger Birkeland for answering questions, steering me in the right direction, and for reading my drafts while guiding me through the writing process. I am grateful to Amund Gjersvik for helping me design and manufacture payload hardware. Some of my results are thanks to Rimantas Žičkus from NanoAvionics, who has answered questions and assisted me with technical issues. I would like to thank my supervisor Tor Arne Johansen for giving me the opportunity to work with this project. Last, but not least, I would like to thank NTNU SmallSat Lab and Evelyn Honoré-Livermore for providing me with a wonderful team and work environment in which to do my thesis.

Magne Hov
Trondheim, June 3, 2019

Contents

List of Figures

List of Tables

Acronyms

ACK Acknowledgement. 11–13, 15, 29–31, 68, 69, 74, 91

ADCS Attitude Determination and Control System. 6, 7, 59, 99, 100

AMOS Centre for Autonomous Marine Operations and Systems. 3, 4

ARQ Automatic Repeat Query. 11, 15, 29, 31, 34, 58–60, 71

BOB Breakout Board. 7, 37, 43–46, 77, 78, 92, 93

CAN Controller Area Network. 17–20, 39, 45, 49, 51, 59, 75–78, 81, 82, 84–86, 89–92, 101

CBOR Concise Binary Object Representation. 29

CCSDS The Consultative Comittee for Space Data Systems. 26, 30

CFDP CCSDS File Delivery Protocol. 30, 31, 34, 72, 75

CIDR Classless Inter-Domain Routing. 20

CLI Command Line Interface. 37, 47, 51, 80–82, 86, 93

CoE Centre of Excellence. 3

COTS Commercial off-the-Shelf. 3, 43

CPU Central Processing Unit. 48

CRC Cyclic Redundant Check. 11, 18, 21, 25, 34, 54, 59, 65, 72

CSP Cubesat Space Protocol. 17, 19–21, 25, 32, 33, 39, 41, 46, 49, 51, 53, 54, 58, 59, 62, 64, 72, 74, 76, 80–82, 84, 85, 89–92, 98, 101

DSP Digital Signal Processing. 44

ECC Error Correcting Code. 11, 15, 44, 48

EM Electromagnetic. 10, 18

eMMC Embedded MultiMediaCard. 44, 46, 48

EOF End of File. 22, 29, 31

EPS Electrical Power System. 6, 7, 46, 49, 77, 78, 84, 98, 99

FAT File Allocation Table. 22, 48

FC Flight Computer. 6, 7, 49, 99, 100

FEC Forward Error Correction. 11, 15

FPS Frames Per Second. 44, 47

FSBL First Stage Boot Loader. 47, 48

FT File Transfer. 8, 19, 37, 39–41, 47–49, 53, 54, 56, 57, 59, 60, 72, 77, 85, 89–93

FTC File Transfer Client. 53, 54, 60, 62–65, 67, 69–71, 78, 80, 85, 86, 90, 92

FTP File Transfer Protocol. 26, 30

FTS File Transfer Service. 53, 54, 60, 62–64, 67–72, 78, 80, 81, 85, 86, 90

FUSE Filesystem in Userspace. 22

GPIO General-Purpose IO. 45

GPS Global Positioning System. 7, 45, 99, 100

HMAC Hash-Based Message Authentication Code. 21

HSI Hyperspectral Imaging. 5, 6, 39–41, 43–47, 77, 92

HYPSO Hyperspectral SmallSat for Ocean Observation. 3–8, 15, 17, 18, 20–22, 39–41, 43, 44, 48, 49, 53, 54, 59, 91, 93, 100

I2C Inter-Integrated Circuit. 101

iDS Imaging Development Systems. 43, 44, 46

IP Internet Protocol. 19, 49

JSON JavaScript Object Notation. 29

LEO Low Earth Orbit. 4, 97

LOS Line of Sight. 4, 10, 49

LUT Look-Up-Table. 44

M6P M6P Multi-Purpose Nano-Satellite Bus. 6, 7, 25, 26, 32, 39, 45, 49, 51, 53, 54, 59, 60, 62, 77, 78, 80, 82, 85, 86, 89, 91–93, 97

MCS Mission Control Software. 51, 63, 90, 92

MIO Multiplexed IO. 46

MPPT Maximum Power Point Tracking. 98

MTU Maximum Transfer Unit. 20, 33

NA NanoAvionics. 6, 7, 32, 39, 49, 51, 53, 59, 60, 72, 78, 90, 101

NACK Negative Acknowledgement. 11, 30, 31

NASA National Aeronautics and Space Administration. 39

NNG NanoMsg-Next-Generation. 51, 81

NTNU Norwegian University of Science and Technology. 3, 4, 45, 51

NTP Network Time Protocol. 45

NUTS NTNU Test Satellite. 3

OBC On-board Computer. 29, 99

OCM On Chip Memory. 48

OPU On-board Processing Unit. 4, 5, 8, 37, 39, 43–49, 57, 59, 77, 78, 84, 86, 89, 92, 93

OSI Open Systems Interconnect. 15, 17

PC Payload controller. 7, 45, 49, 51, 63, 64, 70, 72–78, 84–86, 90, 91, 101

PCB Printed Circuit Board. 45, 46, 97

PDU Protocol Data Unit. 17–19, 27, 29, 31

PL Programmable Logic. 44

PPS Pulse-Per-Second. 45, 77, 100

PS Processing System. 44

PZ PicoZed. 7, 43–46, 48, 77, 78, 92

QSPI Quad-SPI. 44, 46, 48

RAM Random Access Memory. 44

REPL Read–eval–print loop. 80

RFC Request for Comments. 26, 28

RGB Red-Green-Blue. 5, 43–45, 77, 92

ROM Read-Only Memory. 48

RTOS Real-Time Operating System. 47

RTT Round Trip Time. 12, 84, 85

SD Secure Digital. 22, 44, 46, 48, 77

SDR Software Defined Radio. 4, 6, 8, 39–41, 48, 49, 59, 89

SNR Signal-to-Noise Ratio. 10

SoC System on Chip. 44–47, 49

SoM System on Module. 43–46, 48, 78, 92

SPI Serial Peripheral Interface. 49, 59, 75, 101

SSBL Second Stage Boot Loader. 47, 48

TCP Transmission Control Protocol. 19, 29, 49, 51, 81

TFTP Trivial File Transfer Protocol. 26–28, 37, 85, 90

TM Telemetry. 51

TT&C Telemetry, Tracking and Command. 41

UART Universal asynchronous Receiver-Transmitter. 101

UHF Ultra-High Frequency. 6, 7, 49, 51, 90, 99, 100

UNIS The University Centre in Svalbard. 43

USB Universal Serial Bus. 43–45, 51, 77

WDT Watchdog Timer. 99

XTEA Extended Tiny Encryption Algorithm. 21

Part I

Introduction & Background

Chapter 1

Introduction

Technological advances are allowing electrical and mechanical systems to be miniaturised. This is apparent in many of the products that are being used every day, like laptops and mobile phones, and satellites are going through the same process.

A CubeSat is a small satellite that is made up of one or more *units*. Each *unit* has a 10 cm × 10 cm × 10 cm form factor. They are associated with low-cost Commercial off-the-Shelf (COTS) components and are used as a cost effective platform to get small payloads into space. The CubeSat satellite must adhere to the design rules defined in the CubeSat standard in order to qualify for integration into a launch vehicle [1].

Since the first wave of CubeSats in the early 2000's there has been a steady growth of small satellites being launched for scientific and commercial purposes. Over a thousand CubeSats have been launched to this date, and three thousand more are predicted to be launched during the next six years [2].

Companies that supply components for small satellites are offering matured systems, and launch service providers are benefiting from catering to small satellites. A CubeSat is increasingly seen as a platform to solve real world problems.

1.1 The HYPSO Mission

The SmallSat Lab at the Norwegian University of Science and Technology (NTNU) is a loosely structured organisation and team consisting of professors, postdocs, Phd. candidates, Msc. students, BSc. students and volunteers. The team is supported by the Centre of Excellence (CoE) Centre for Autonomous Marine Operations and Systems (AMOS), which administrates multiple research projects related to maritime operations and systems in Norway.

The SmallSat Lab was created in 2017 and has since been working to develop a mission plan and systems for a small satellite mission in association with AMOS. The mission has had several names, and is currently named Hyperspectral SmallSat for Ocean Observation (HYPSO). The HYPSO mission is designed with consideration to the needs of several users and beneficiaries.

The HYPSO mission is in itself an instrument used by the SmallSat Lab to develop and build competence. The SmallSat Lab is associated with NTNU's small satellite strategy.

The SmallSat Lab aims to promote space technology at NTNU, but is not the only project to do so. Previous efforts at NTNU include NTNU Test Satellite (NUTS), a student driven CubeSat project that ran from 2013 to 2017. The NUTS organisation has since been rebranded as Orbit NTNU, which still develops CubeSat projects. Propulse NTNU is a new addition to the space related activities at NTNU. This project is also student driven, and aims to develop and manufacture rockets for participation in Spaceport America Cup, an international student competition for rocketry.

The needs of the Norwegian maritime industries are communicated via collaborated projects that are managed through AMOS. The technological advances that the HYPSO mission aims to enable are particularly applicable to aquaculture. For this reason, many of the requirements that set the basis for the mission design are derived from the needs of this area of industry.

The fishing industry is naturally concerned about the wellness of its livestock. A common cause of concern for fish farmers is the effect that algal blooms can have on the fish farming environments. Certain algae are toxic to fish, and can reduce the quality of the fish or even kill it. Therefore, the fishing industry has a need for good methods of detecting and monitoring such blooms. If a bloom is detected before it reaches a farm, the farmers can carry out preventive measures before their livestock is damaged. The algae usually floats in the top layers of the ocean, which is also where the fish resides while being fed. By not feeding the fish when a bloom is detected, the fish will remain deeper within the fish enclosure and avoid contact with the damaging algae. Recent events saw 3000 tonnes of salmon being wiped out by algal blooms in Norway [3].

The technologies that the HYPSO mission seeks to demonstrate will be able to provide detection of these algal blooms. A hyperspectral imager will be able to record images which can be analysed to identify algal blooms. The requirements of the fish farmers are included in the mission design. The mission is designed for the ability to provide early signs of bloom detection. By being in Low Earth Orbit (LEO), the satellite gains a large field of view and a short revisit period. These two properties will enable the satellite to detect blooms in a large area. It can therefore provide warnings in a relatively short time frame.

Another goal which is collaborated through AMOS is the development of a network of sensory entities for surveillance of the oceans. This long term goal is what the HYPSO mission draws inspiration from for its name. In this project, the HYPSO satellite plays several roles. One role is to be a producer of coastal and oceanic imagery, which can be used for a wide range of monitoring and tracking purposes.

A secondary mission objective for the HYPSO mission is to test communication services for remote sensor nodes. Small satellites are being envisioned as communication nodes for a network of sensors [4]. These nodes are normally out of reach of conventional networks. Example are the sensory buoys that regularly collect environmental climate data from the arctic oceans. With conventional methods, the collected data is retrieved by visiting the buoy, which is only possible during certain periods of the year. A space borne network node such as the HYPSO satellite would be in Line of Sight (LOS) to the buoy on a daily basis, and would not be limited to seasonal periods of access.

The HYPSO mission carries two scientific payloads: an imaging system with an On-board Processing Unit (OPU), and a Software Defined Radio (SDR) system. Both systems are designed to carry out tasks and experiments that are of interest to the NTNU SmallSat Lab and its collaborators.

1.1.1 Hyperspectral Imager Payload

The primary payload of the HYPSO mission is a camera. It is capable of capturing hyperspectral images, which contain spectral (colour) information over a sampled range of wavelengths.

The camera works like a push broom scanner [5], as illustrated in Figure 1.1. Light is received through a slit and diffracted to distribute the spectral bands in the direction normal to the direction of the slit and the direction of travel. The image sensor captures a two dimensional frame where one of the two dimension corresponds to the spatial dimension along the slit, and the other dimension corresponds to the distribution of colour.

Multiple frames are merged to create a hyperspectral cube with three dimension. Two of the dimensions are spatial, and the last one is spectral, as illustrated in Figure 1.2. The last spatial

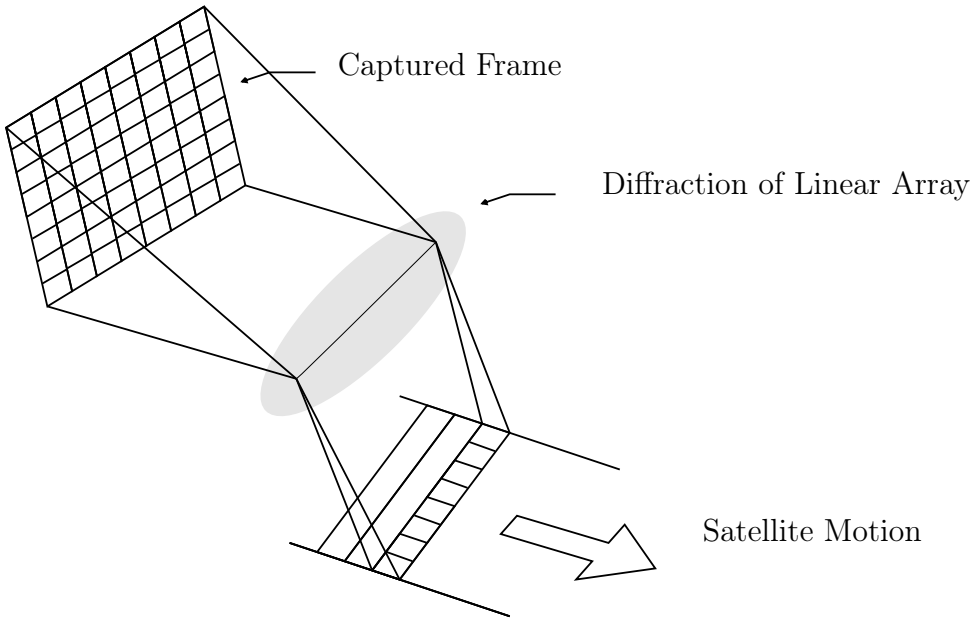


Figure 1.1: Illustration of push broom scanning. A slit of light is refracted onto a plane. One dimension of the plane corresponds to the spatial dimension of the linear array, while the other dimension of the plane corresponds to the spectral content of the linear array. The linear array covers new ground as the satellite moves.

dimension is governed by the pointing of the satellite as it progresses through time.

Image capture is controlled by an OPU. The OPU is responsible for processing the images that it receives from the Hyperspectral Imaging (HSI) sensor.

The image files produced by the HSI must be downloaded to the ground to be analysed and distributed to researchers. The files are compressed before transmission to reduce the required transfer time.

There are also systems being developed that will perform on-board analysis of the images. This will allow useful information to be extracted from the images without having to first download them to a ground station.

Regardless of whether the images are analysed on-board or on the ground, the captured information will be useful for a range of applications. The fishing industry can use the data to produce algae forecasts. The ocean colour community benefits from getting more hyperspectral data that can be used to monitor new areas and to develop new analytic methods.

An auxiliary, conventional Red-Green-Blue (RGB) camera is also included in the mission. This camera provides wide-angle imagery along the same direction as the HSI camera. The image data can be used to validate the HSI imager, to provide geographical data that is used for geo-referencing the HSI images, and to enable super-resolution techniques. The camera also serves the purpose of producing traditional images of the earth, which is in itself an interesting product.

Although the HSI is the primary function of the payload, the term OPU is used to refer to the whole payload, including HSI, RGB and other constituent components.

1.1.2 Software Defined Radio Payload

The second payload of the HYPSO mission is a Software Defined Radio (SDR) system. This is an experimental radio.

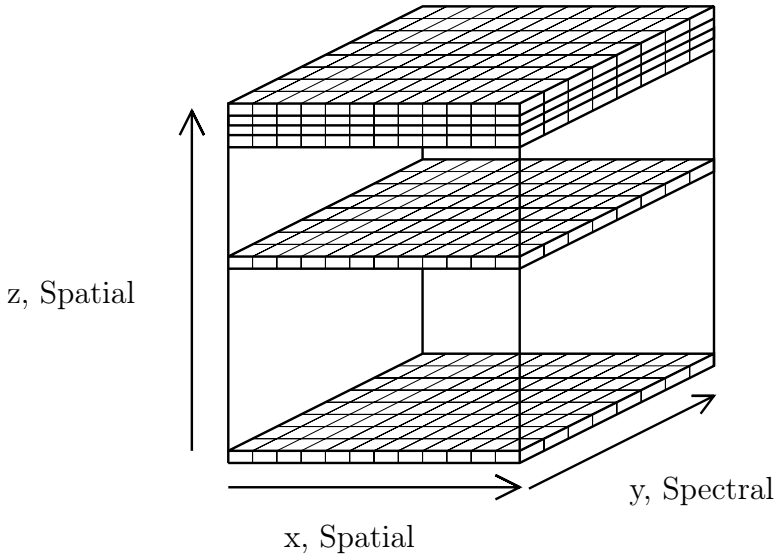


Figure 1.2: Illustration of hyperspectral data. Two dimensions are spatial, while the last dimension contains spectral information. Each plane represents a captured frame.

A SDR is a category of radio modules that has moved the functionality that was conventionally performed by analogue components into to the software domain. The components that have been replaced by software would previously perform processing on the signal after or before the signal would have been passed to the power amplifiers and antennas. Examples of processes that are replaced are mixers, filters, modulators/demodulators and detectors.

The SDR system is intended to carry out tests and experiments for the envisioned communication network that is required to operate remote sensor nodes. The fact that the radio is software programmable means that it can test different operating configurations.

1.2 M6P Satellite Platform

The mission payloads depend on a number of resources and capabilities from a supporting platform. Both payloads must be supplied with electrical power to operate. The HSI payload requires a radio link to transmit images to the ground, and depends on the satellite being able to control its attitude and point towards specifics locations on Earth. The SDR payload requires an external antenna.

The HYPSO mission procures satellite components from the Lithuanian company NanoAvionics (NA). This company specialises in developing and manufacturing subsystems and platforms for CubeSat missions. NA offers a series of satellite buses, with variants for the most common sizes: 2-unit, 3-unit and 6-unit CubeSats.

The HYPSO project is using the M6P Multi-Purpose Nano-Satellite Bus (M6P), NA's 6-unit CubeSat platform. It provides the following components:

- Mechanical frame and solar panels.
- Electrical Power System (EPS) module.
- Flight Computer (FC), which contains:
  - Attitude Determination and Control System (ADCS).
  - Ultra-High Frequency (UHF) radio module.

  - Global Positioning System (GPS) module.

- S-Band radio module.
- Payload controller (PC).
- Room to install payloads.

An architectural diagram is provided in Figure 1.3 to illustrate the relations between the satellite subsystems. The Breakout Board (BOB) and PicoZed (PZ) components are described later in Chapter 4.

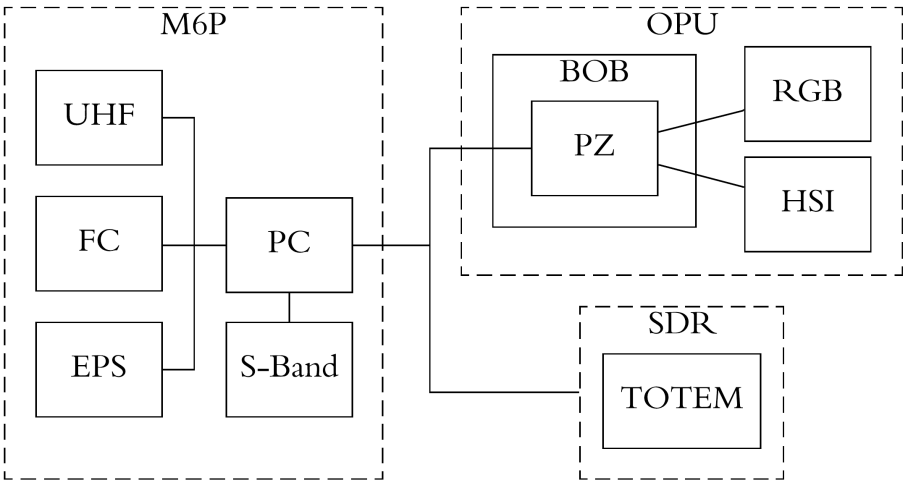


Figure 1.3: Relations between the subsystems of the HYPSO satellite.

The EPS subsystems collects energy from the solar panels, stores it in batteries, and provides regulated power to the other subsystems. The FC subsystem performs ADCS related activities such as pointing and slew manoeuvring, as well as collecting sensor and GPS data. The UHF radio and S-Band radio communicates with ground stations. The PC handles all interfaces between the payloads and the satellite platform, including a communication bus and power connections.

The M6P satellite platform is created with a reliable and redundant design in mind. At all times, except when lacking electrical power, all subsystems can operate independently of each other. The EPS is equipped with several fail-safe mechanisms to avoid electrical damage to itself and other subsystems. The PC is able to perform some of the FC's tasks in case of system failure. The ADCS has a redundant pair of actuators available, based on different technologies, and two radios operating on different frequency bands are installed. Accelerated lifetime tests, such as high radiation dosing, have been performed by NA to guarantee a minimum lifetime of 5 years in low earth orbit.

More details about the subsystems have been summarised in Appendix A.

1.3 Problem Statement

The student shall develop, implement and test solutions for file transfers for the HYPSO mission payloads. The task includes developing or applying derived components such as electronic hardware and an operating system for the payload processor system. The file transfer implementations shall make use of and integrate with the existing communication architecture provided by the satellite bus provider.

1.4 Thesis Outline

This thesis is concerned with the work of designing, implementing and testing a File Transfer (FT) system for the HYPSO mission. The OPU and SDR payloads both depend on file transfer capabilities to perform their responsibilities. The OPU payload must downlink captured image data, and the SDR payload must downlink experimental measurement data. The FT system is focused on software design, but some hardware was also designed as part of this thesis.

Part I provides information about the satellite project and background theory, while Part II is concerned with the contributions of the thesis.

Chapter 2 explores background material that is relevant for the design process of the FT system, which is the primary product of Part II.

Relevant mission requirements are mapped in Chapter 3. The HYPSO payloads and communication architecture is described in Chapter 4, together with design considerations for the OPU hardware.

Chapter 5 presents the design and implementation details of the FT system. The FT system is tested, and results presented in Chapter 6. Results are discussed in Chapter 7 before a final conclusion is presented.

Chapter 2

Background

This chapter introduces theory that is relevant in order to understand the work performed in this thesis.

2.1 Communication Theory

The topics of this thesis deal with the communication between a satellite and a ground station. A few communication concepts are discussed to provide some background.

2.1.1 Flow Control

Regardless of which technology is being used to transfer a message, there will always be some maximum speed at which the communication channel can transfer messages. The speed at which a channel can transfer messages is referred to as its *data rate*.

In the case of peer-to-peer communication through a direct channel, the data rate of the channel will create a self-regulating bottleneck which limits how quickly the sender will be able to dispatch messages. If the recipient is not able to receive messages at the same frequency, messages will be dropped.

Another problem appears when dealing with indirect communication. If there is more than one node that takes part in the communication chain, then each message must be buffered while it is processed and passed forward. A system will always have a limited amount of buffer space available. If a node receives messages faster than it is able to send them (mismatched sending frequency on total of input and output channels), then the buffer space will eventually be exhausted and messages will be dropped.

If there are multiple users on a network, congestion becomes a problem. When there are multiple users, they must either share the data rate of the network, or find that their packets are dropped due to congestion.

Packet loss can be prevented by measuring the loss rate and feeding it back to the sender which can adjust the sending rate.

2.1.2 Communication Errors

Errors manifest themselves on a communication channel in a number of different ways. The space environment introduces new sources of errors for electronic systems [6]. Strong radiation in the form of ionising particles (ions, electrons, photons) can deposit or disturb charges present in electrical components. The foreign charges can cause transient errors in communication equipment. An error that is caused by such an effect is called a *single event upset*.

A charge that is present in a capacitive storage device can be displaced as a result of radiation. The bit that was represented by that charge is then altered, resulting in data corruption.

The telecommunication technologies that are used in spacecrafts are inherently noisy. The transmission and reception of Electromagnetic (EM) waves is dependent on a number of analogue components that pick up noise, such as antennas, amplifiers, modulators and filters [6].

The atmosphere degrades an EM wave signal via effects such as absorption and reflection [6]. Through a full pass of a satellite, the degrading effect is most prominent when the ground station and the spacecraft has a low LOS on the horizon. At this point the radio waves must pass through the longest distance of the signal-degrading atmosphere. This and other effects will reduce the Signal-to-Noise Ratio (SNR). If the signal is degraded beyond the capability of the receiver to decode the signal, it will result in loss of messages. A low SNR also makes it more probable to incorrectly interpret a symbol, causing data corruption.

2.1.3 Error Detection and Correction

In the general case of data corruption, error detection and correction methods can be used to mitigate the effects.

Other communication errors such as total loss of message, duplication of message and reordering of messages must also be considered when discussing error mitigation techniques. Duplication and reordering of messages can be solved by employing a sequence number. A number is added to each message, and is increased by one for each successive message. The number is examined at the receiving node to determine what order the messages should be received in. Additionally, if two messages with the same sequence number are received, a duplication is detected and only one of the messages is kept. Depending on the application, to mitigate reordering by using sequence numbering, extra buffer space might have to be utilised to store reordered messages while waiting for delayed messages. Some applications do not care about the ordering of packets, and can also be unaffected by duplication of packets. The loss of a message necessitates retransmission techniques, like those detailed in Section 2.2.

The general idea of detecting data corruption is to append redundant information to the data. The additional data is added in such a way that the message fulfils a specific property. The property is checked when the message is received, and will indicate whether the message has maintained its integrity.

The simplest way to provide error detection is to add a `0` or `1` parity bit, which is calculated from the message. The receiver performs the same calculation and compares. This method is weak, however, because it can only detect an odd number of bit flips.

Checksum

A more robust technique is to use a checksum, a code that is computed by using the message as input to a hashing function. A hashing function takes data of arbitrary length as input and produces a fixed length output. Hash functions are characterised by being able to produce seemingly random and distinct output from similar inputs.

Polynomial codes have proved to be good checksum generators. A polynomial code is defined as the polynomial remainder R of a polynomial long division, where the input data is used as the dividend A and a static polynomial (a number) is used as the divisor B [7]. The natural number (integer) quotient of the division is discarded, and the remainder R is used as the checksum. The resultant quotient and remainder fulfil equation (2.1). The size of the remainder R is governed by the size of the static divisor B, which is also called the generator polynomial, and which must be known by both sender and receiver. The choice of the generator polynomial directly affects the error detection performance of the checksum.

$$A = B \cdot Q + R \tag{2.1}$$

Polynomial codes used as checksum are more commonly referred to as Cyclic Redundant Check (CRC) codes. There are different variants of CRC depending on how long of a checksum is wanted. Popular variants include `CRC16` and `CRC32`, which produce 16 and 32 bits long codes, respectively.

Like other error detection methods, CRC can only prove corruptness of data, not that the data is valid. The performance of a checksum is therefore reliant on the hashing function producing vastly different output for small differences in input. Then it becomes highly improbable for a checksum to be verified when it is in fact corrupt.

An important factor to the performance of CRC is that it is also very easy to implement in software or hardware, requiring few instructions to compute. Additionally, parts of the algorithm can be implemented through memoisation, meaning that the intermediary results from expensive operations can be stored and looked up cheaply from memory.

Forward Error Correction

In addition to error detection schemes, there are methods of Forward Error Correction (FEC). Given specific situation, these FEC methods can not only detect data corruption, but also correct for it. For certain applications, this type of error mitigation can be preferred over methods that are based on retransmission, like those later detailed in Section 2.2. It is especially useful for applications that cannot keep data stored, in order to enable retransmission of lost messages. It is also useful for applications where the delay that results from a retransmission would render the data unusable because of real time requirements.

In a similar fashion to error detection, redundant data is appended to the message to create a Error Correcting Code (ECC). The additional information is used to recover the original message.

All ECCs come with a performance cost. The additional data required to perform error correction increases a constant message overhead. Therefore, if a channel exhibits a low symbol corruption rate, the presence of ECC may ultimately decrease the overall performance.

2.2 Automatic Repeat Query

In Section 2.1.3, FEC was shown as an error mitigation technique, and here Automatic Repeat Query (ARQ) is detailed as an alternative approach of error mitigation.

There are two ways in which the successful delivery of a message can be expressed and communicated. One way is for the receiver of the data to send an Acknowledgement (ACK) for data that has been received. Upon receiving an ACK, the sender of the data will know that a specific piece of data was successfully received.

The other way is for the sender of the data to implicitly assume that all data that is sent will be received successfully. If the receiver of the data detects that it did not receive some piece of data that is should have received, it will issue a Negative Acknowledgement (NACK) for that specific data. Upon receiving a NACK, the sender of the data will resend the missing data.

The different strategies for sending ACKs and NACKs are referred to as ARQ, regardless of whether the query is a positive or negative acknowledgement. Several of these *retransmission strategies* are explained in the following sections.

2.2.1 Stop-and-Wait

The *stop-and-wait* strategy is one of the simplest communication schemes that also can provide some amount of reliability on channels with message loss.

For comparison, a strategy is imagined, where a sender is sending successive messages as quickly as it is able to send. The receiver accepts any valid message and interprets it as the next message in the stream of messages.

The *stop-and-wait* strategy improves on this *unrestricted* strategy by having the receiver respond with an ACK for every received message [8]. The sender must wait for an ACK of the previous message before sending the next one. A *stop-and-wait* exchange is illustrated in Figure 2.1.

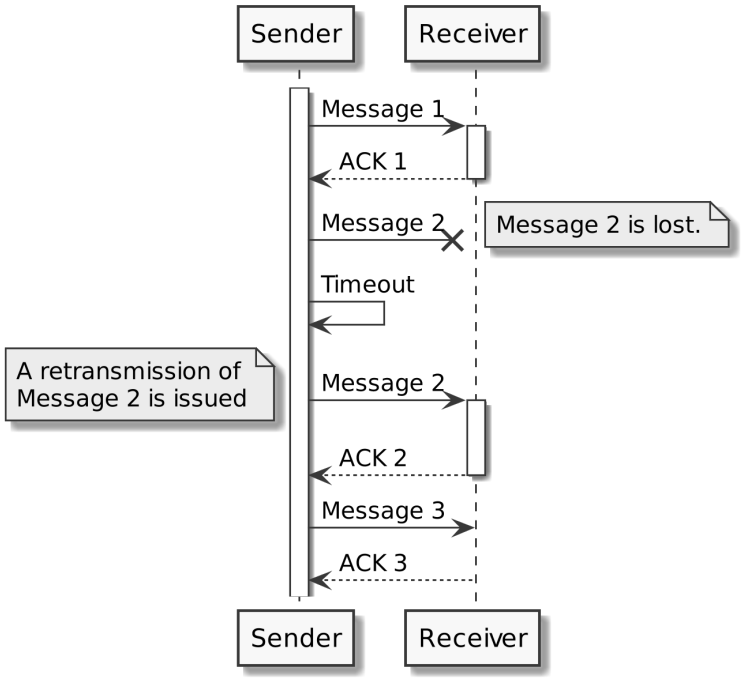


Figure 2.1: The *stop-and-wait* strategy. The sender waits for an ACK of the previous message before sending the next message. Message 2 is lost in transmission, and the sender times out because of a lacking ACK. As a result, Message 2 is retransmitted.

The act of waiting on an ACK before sending the next message has the effect of limiting the sending frequency. The sending period will roughly be the Round Trip Time (RTT) between sender and receiver, plus additional processing times at each node, and the transmission time of the data packet.

$$\text{Sending Period} = \text{RTT} + \text{Data Transmission Time} + \text{Processing Time} \tag{2.2}$$

The *stop-and-wait* strategy does not survive the loss of a message. The strategy is extended by adding a timeout on receiving ACKs. If the sender does not receive a ACK within a certain time period, it will assume that the previous message was lost, and will retransmit it. This can be repeated a number of times until the sender gives up after reaching a configurable threshold.

The extension with timeout on ACK can handle data message loss, but not the loss of ACK messages. If an ACK is lost, then the sender will time out and retransmit the *previous* message. The receiver, however, has successfully received the previous message, and has no mechanism in place to detect whether the ACK was received successfully by the sender. Therefore, the receiver

is not able to distinguish between the next or previous message. The receiver will incorrectly accept a retransmitted message as the next message.

The solution to this duplication problem is to include a sequence bit with each message. The sequence bit is toggled for each successive message. Since there are only two possible messages that the recipient can receive, namely the previously retransmitted and the next one, the receiver can examine the sequence bit to determine whether the message is a new one, or a retransmission of the previous one.

2.2.2 Sliding Window

The extended *stop-and-wait* strategy is reliable, but not very efficient. While waiting for the data message to be processed by the receiver, and for the ACK to reach the sender, the communication channel is not being utilised.

In order to improve performance, the notion of a sliding window is introduced. The idea is to let the sender transmit multiple data messages before waiting for ACKs. A sequence number is added to each data message, and a corresponding sequence number is added to every ACK message. This way an ACK can be linked to a specific data message.

The window length decides how many data messages can be in transmit at once. The *stop-and-wait* strategy is a special case of the sliding window class of strategies, where the window has a length of $n = 1$. The upper and lower bound of the `sender window` determines which messages are allowed to be sent, while the upper and lower bound on the `receiver window` determines which messages the receiver is ready to accept.

The windows are said to be *sliding* because the lower and upper bounds grow when message are received and successfully acknowledged. The exact way in which they are advanced is different for each variant of sliding window protocol.

The length of the windows affects performance. Increasing the sending window length n up from zero will initially increase throughput. However, at some point the sending frequency will be greater than the channel or receiver is able to handle. This could be because of congestion, mismatched interface speeds or other more complex issues. At this point, raising the sending window length might decrease the throughput because messages are dropped and will have to be retransmitted. There will be a specific window length which gives the highest effective throughput, but it might change over time along with a changing environment. The window can be regulated to achieve the highest throughput, or to avoid loss of packets.

Go-back-N

The *Go-back-N* protocol uses a sender window length of $N + 1$ and is based on the idea of resetting the transfer back to the last message that was lost.

The sending node dispatches all the messages in its sending window. Then it waits for ACKs for the transmitted messages. The sender never progresses further than N messages ahead of the previously acknowledged messages. A timer keeps track of every transmitted data message, and if any of them times out, retransmission is performed from the beginning of the sending window. An example of *Go-back-N* with a window length of three is illustrated in Figure 2.2.

The receiver has a window of length one, and will therefore only accept the next message in the sequence. Any message that arrives and that is not the next message in the sequence is discarded. This means that messages that arrive out of order will also be discarded. This is an advantage as it does not require the receiver to buffer messages that are stored out of order, but it is also a disadvantage in that it results in a greater number of retransmission.

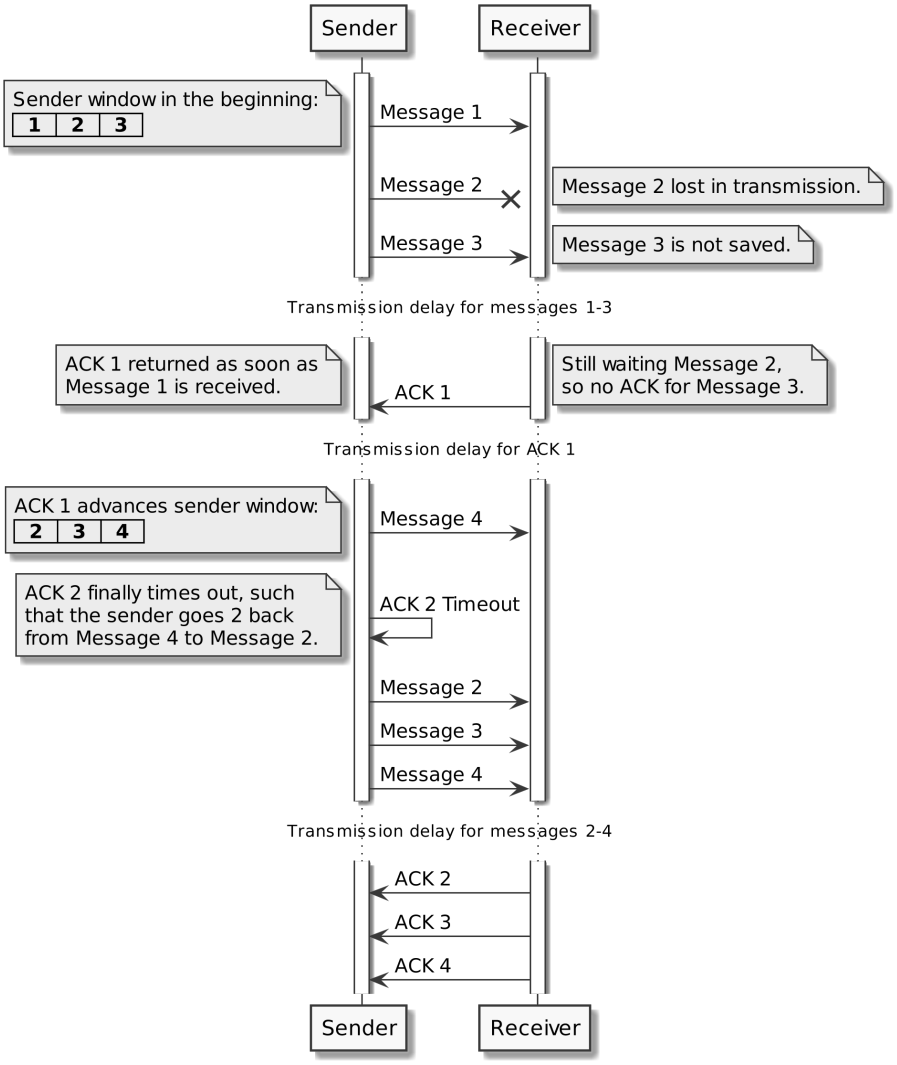


Figure 2.2: The *Go-back-N* strategy with a sender window of length 3. Message number 2 is lost in transmission, such that the sender times out on ACK 2. The second transmission of Message 2 succeeds, as does all subsequent messages. However, Message 3 and 4 are retransmitted even though they were successfully received the first time.

Selective Repeat

The *selective repeat* protocol attempts to improve performance by allowing the receiver to store out of order messages while waiting for earlier messages. An example of the *selective repeat* strategy is shown in Figure 2.3.

The idea is to let the receiver have a window length of more than one. The receiver may store any messages that are within its window. The window is moved one to the right whenever the earliest message (left-most in window) is received. If there are messages stored right after the earliest message, then the window is moved straight up to the first earliest message that is not received yet. The sender window is restrained such that only messages that are within the receivers window will be sent in the first place.

The sender window is anchored by the earliest message that has not yet been acknowledged. Whenever the sender receives an ACK, it can automatically assume that all messages earlier than the ACK are also received. An ACK therefore moves the sender window to one past the sequence number of the ACK.

The *selective-repeat* strategy will typically outperform the *go-back-n* strategy when there are frequent message losses. The cost is increased buffer space at the receiver side, and a greater number of states that increase the complexity of the implementation.

2.2.3 Hybrid ARQ

Instead of relying solely on FEC or ARQ for error mitigation, it is possible to combine the two approaches. This is especially helpful when a channel has multiple failure modes that produce different kinds of errors.

If on a constantly noisy channel there happens to be a single bit error in a large amount of messages, then a ECC is the best approach. A pure ARQ approach would end up having to retransmit almost every message. However, if the channel has periods of disruption, such that a sequence of messages is completely lost, then an ARQ approach is better. An ECC is of no use if the whole message is lost.

When a channel exhibits the failure characteristics of both, corrupted bits are frequently encountered and full messages are being sporadically lost, then the best approach is a combination of FEC and ARQ.

The two approaches must not necessarily be applied to the same technology or even placed on the same level in the network stack. It is entirely possible, and even widely adopted, to utilise FEC in lower level protocols, while relying on higher level protocols or even applications to perform ARQ.

A 2001 study looks at *Incremental Redundancy*, a retransmission scheme where additional parity bits are transmitted if the original message could not be decoded [9]. This combines the idea of retransmission and FEC into the same protocol. The study found that for many situations the hybrid solution performed better than retransmission of the whole original message.

2.3 Network Stack Model

The HYPSO satellite is comprised of a number of subsystems. These subsystems communicate between each other and form a network. Multiple technologies and protocols are stacked on top of each other to create the network.

In order to talk efficiently about the mission network, the protocols that are relevant to the HYPSO mission payloads will be discussed in the following sections. To arrange the protocols into a structure, the Open Systems Interconnect (OSI) reference model for network systems is used to organise the protocols into layers [8]. The lower level layers perform low level functions

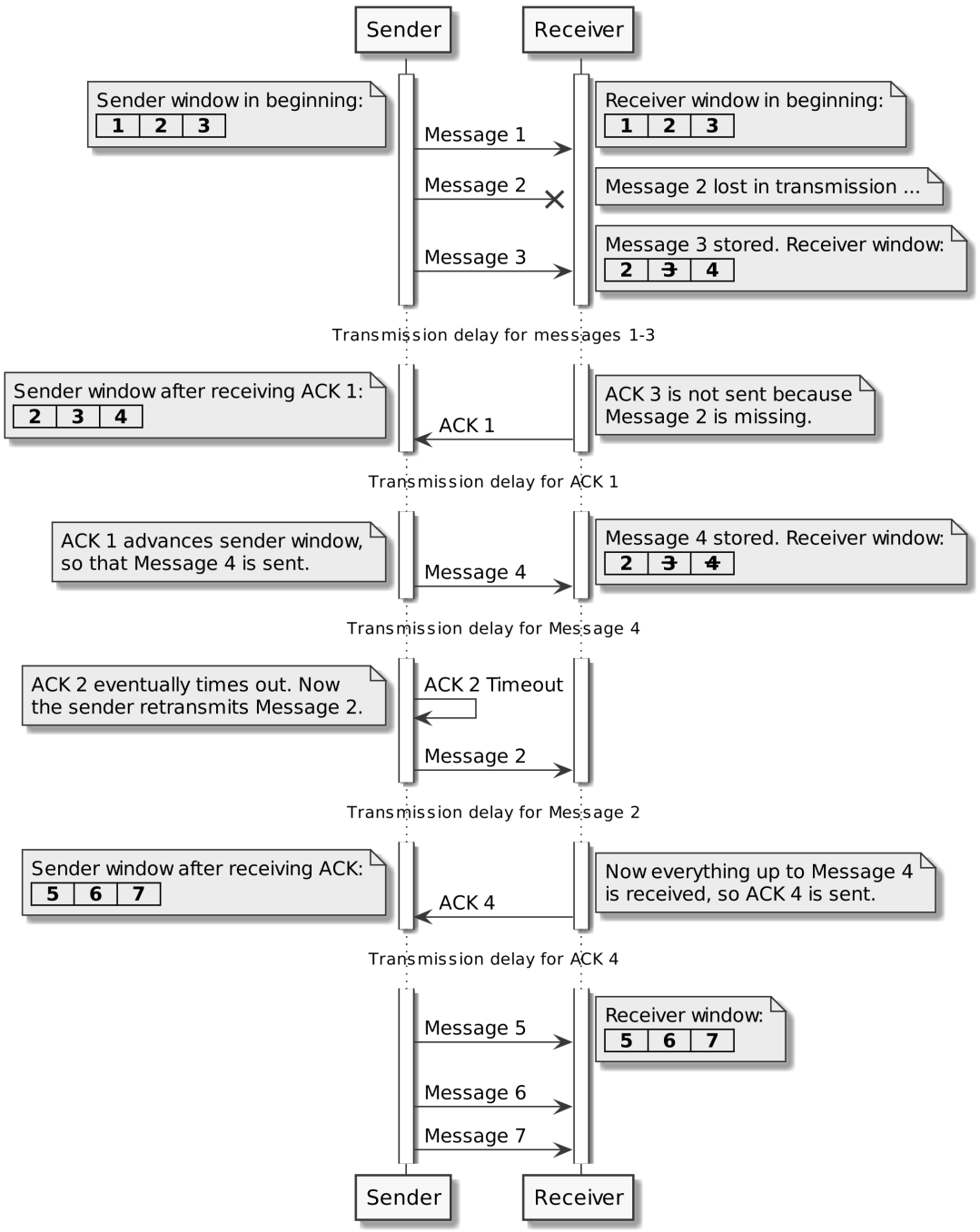


Figure 2.3: The *Selective Repeat* strategy with a sender and receiver window of length 3. Message 2 is lost in transmission, but Message 3 is still stored at the receiver. ACK 1 advances the sender window, such that Message 4 is sent. No ACK 2 is sent, so that Message 2 times out, issuing a retransmit of Message 2. When Message 2 arrives, ACK 4 can immediately be sent instead of ACK 2 because both Message 3 and 4 have already been stored. The ACK 4 is an implicit acknowledgement for also Message 2 and Message 3.

such as electrical signalling, while higher level layers perform higher level functions such as session handling and reliable message delivery. Figure 2.4 presents all the original OSI layers, their usual functions and the corresponding names of their data units.

Layers		Protocol Data Unit	Function Examples
Host Layers	Application	Packet Data	Services, file access, database access
	Presentation		Data encoding, compression
	Session		Handling of connections
	Transport	Datagram	Retransmission, fragmentation, acking
Media Layers	Network	Packet	Addressing, routing, traffic control
	Link	Frame	Transmission of data frames
	Physical	Symbol	Transmission of single symbols

Figure 2.4: OSI reference model for network protocols. Layers are stacked with increasing levels of abstraction.

Each layer has its own Protocol Data Unit (PDU), as shown in Figure 2.4. Most PDUs contain a small amount of auxiliary metadata which is used by the protocol to deliver the message. Metadata placed at the start of a message is called a header, while metadata placed at the end is called a tail.

There are primarily two specifications that define the protocols used in the HYPSO mission network stack. The two specification are the Controller Area Network (CAN) specification and the Cubesat Space Protocol (CSP) specification. The CAN specification defines protocols for the physical layer and the link layer, while the CSP specification defines protocols for the network layer and the transport layer, as illustrated in Figure 2.5.

Layer	Provided by	
Application	Payload Service	
. . .	. . .	
Transport Layer	**CSP** Datagrams	
Network Layer	**CSP** Core	**CSP** Router
Link Layer	Linux SocketCAN Driver	
Physical Layer	High speed CAN-bus	

Figure 2.5: HYPSO communication protocol stack. The CAN specification defines protocols that perform the responsibilities of the physical layer and the link layer. The CSP specification defines protocols and mechanisms that perform the responsibilities of the network layer and the transport layer.

2.4 Controller Area Network Bus

This section describes the CAN protocol. The ISO11898 specification series was originally developed for the automotive industry. It is used in the HYPSO mission to connect the satellite payloads to the satellite platform.

The physical bus is made up of a pair of twisted wires: *CAN High* and *CAN Low*, and is driven by differential signalling, which provides some protection against EM noise [10]. This linear bus must be terminated with 120 Ω resistors to provide a drain path for the differential signal, and also to prevent signal reflections.

Link Layer

The ISO11898-3 specification defines the link layer of the CAN protocol [11].

The CAN PDU, the extended frame, is illustrated in Figure 2.6. A length field encodes how many bytes are present in the variable length data field. A CRC checksum is used for the receiver to verify frame integrity. The remaining fields are primarily concerned with flow control, such as acknowledgement and frame boundaries.

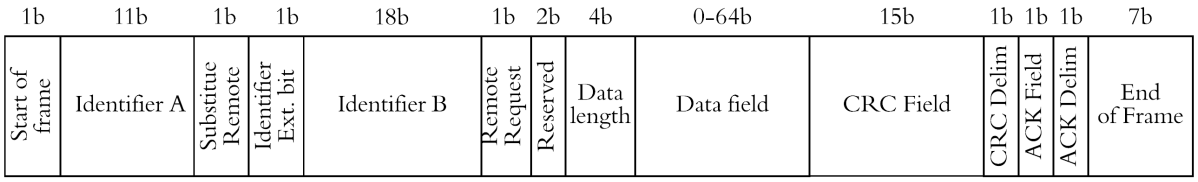


Figure 2.6: The extended CAN protocol frame, with various header fields.

A 29-bit *identifier* field is used for addressing (or 11-bit in the case of a regular CAN frame). For more details on CAN-bus, see report on integrating the network stack with the payloads [12].

Transfer Rate

Although the ISO11898-2 CAN-bus is able to operate at signalling speeds up to 1 Mbps, the effective data transfer rate is significantly lower.

In the best case, the extended CAN frame (shown in Figure 2.6) can hold 8 bytes of data. The remaining bits are used for addressing, CRC and control. Between each frame, an interframe space consisting of at least three recessive bits must be inserted. Additional data bits may be inserted by a bit stuffing rule, further lowering the effective data rate. In the worst case, 64 successive identical data bits causes 12 extra bits to be inserted. In the best case, no extra bits are required.

At full bus utilisation, the theoretical upper limit for effective data rate becomes 48.85 % or 488.5 kbps, as calculated in equation (2.3). At the worst case with 12 bits being stuffed, the effective data rate drops to 44.76 % or 447.6 kbps, as calculated in equation (2.4).

$$\begin{aligned} \text{Max utilisation, upper limit} &= \frac{\text{Data}}{\text{Data} + \text{Overhead} + \text{Interframe}} \\ &= \frac{64}{64 + 64 + 3} = 48.85\% \end{aligned} \tag{2.3}$$

$$\begin{aligned}\text{Max utilisation, worst case} &= \frac{\text{Data}}{\text{Data} + \text{Overhead} + \text{Interframe} + \text{Stuffing}} \\ &= \frac{64}{64 + 64 + 3 + 12} = 44.76\%\end{aligned} \tag{2.4}$$

It is unlikely, however, that the CAN controller is able to fully utilise the bus for extended periods of time. The controllers would need a constant feed of data.

It is also unlikely to experience long sequences of identical data bits during transfers of large files because they are compressed. The compression eliminates long sequences of identical bits. The bit stuffing is therefore not expected to cause a significant decrease in effective data rate for the FT system.

SocketCAN Driver

As part of the Linux kernel modules, SocketCAN provides a link layer interface to CAN devices. The programming interface is identical to that of the Transmission Control Protocol (TCP)/Internet Protocol (IP) network interface. The driver code initialises a Linux socket object from the networking protocol family `PF_CAN`, after which the Linux system calls `read` and `write` can be used to communicate data with the lower level CAN controller.

2.5 CubeSat Space Protocol

This section describes the mechanisms defined by the CSP specification.

CSP is a network library developed by *GomSpace* for use in CubeSats [13]. It performs the functions of the network layer (layer 3) and transport layer (layer 4). The protocol has been adopted by several small satellite designers and has flight heritage. CubeSats such as GOMX-3 and AAUSAT3 have flown successful missions with CSP [13].

2.5.1 Network Layer

When discussing CSP, the terms *ID* and *address* are used interchangeably.

The CSP protocol performs similar functions as IP, but offers a smaller implementation that is suitable for resource limited systems like those found in small satellites. It allows any node in the CSP network to send packets to any other node in the network. Every node in a CSP network has its own unique CSP address.

Packet Format

The format of the CSP PDU is shown in Figure 2.7. Each packet is fronted by a 2 byte data length field that encodes the number of bytes in the data field. The destination ID and source ID fields encode the address of receiver and sender. The destination port and source port fields encode which service port that the packet is sent to and from. Eight flag bits encode CSP options that can be enabled for the packet.

Buffer Management

The CSP stack maintains a bank of buffer memory for internal routing of CSP packets. Before creating a CSP packet, a packet buffer pointer must be acquired with `csp_buffer_get()`. Packet buffers must be manually released with `csp_buffer_free()` when they are no longer needed.

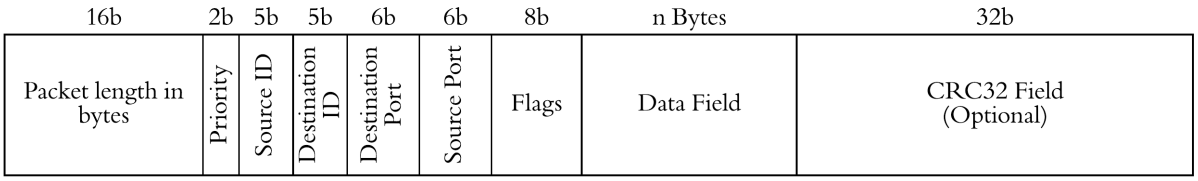


Figure 2.7: CSP packet format. The length field indicates the number of bytes in the data field. ID and port fields encode the address of sender and recipient.

When sending the packet, the pointer to the buffer is passed down the protocol stack, without having to copy the buffer contents. This *zero-copy* technique reduces unnecessary duplication of packet data.

Routing

Interfaces provide channels between CSP nodes. A node may receive and send CSP packets on an interface.

The CSP protocol maintains a single queue for incoming packets. All interfaces will feed packets to this queue. The CSP library provides a `csp_route_work()` function, which is usually run in its own thread (`csp_route_start_task()`). This function reads packets from the input queue and routes them to the correct interface. Packets that are addressed to the receiving node are appended to the CSP socket that is bound to the appropriate port number. If no such port has been bound, the packet is rightfully dropped.

The router refers to a routing table to resolve which interface it must send a packet to. This routing table uses the Classless Inter-Domain Routing (CIDR) method to specify address ranges. This method allows a range of addresses to be specified in a single entry, instead of having to specify every static address that is available on an interface.

CAN interface

The CAN protocol frame length is limited to a maximum of eight bytes. In order to fit larger CSP packets over a CAN interface, some kind of packet fragmenting must be implemented. The CSP code includes a CAN interface implementation with automatic fragmentation of CSP packets into CAN-bus frames. The Maximum Transfer Unit (MTU) over this interface is limited to 256 bytes.

2.5.2 Transport Layer

The network layer delivers the CSP packets to the correct node, while the transport layer provides ports that let an application address specific services.

The standard CSP transport unit is called *Unreliable Datagram Protocol*, reflecting the fact that it does not offer a mechanism to deliver the packet reliably. A reliable transport unit called *Reliable Datagram Protocol* is available, but is not used in the HYPSO network because it has previously been reported as glitchy [14].

2.5.3 CSP Options

Confidentiality, integrity, and authenticity (CIA-triad) forms the foundation of a secure service [10]. The CSP library offers mechanisms to support each of these concepts.

Cyclic Redundancy Check

If compiling the library with the CRC option enabled, packets can be flagged to use 32-bit CRC to protect their integrity. The packet feature is enabled by creating a socket or connection while passing a `CSP_O_CRC32` flag as a function parameter.

Hashed Message Authentication Code

When compiling with the Hash-Based Message Authentication Code (HMAC) option enabled, packets can be protected with a message authentication code field. Packets can then be verified to have been signed with the correct key, proving the authenticity of the packet. The key which is used to verify the packets must be distributed to the satellite in a safe manner, for example by installing the key before launch. This option is not employed in the HYPSO network because, authentication is provided in the radio link layer protocol.

Extended Tiny Encryption Algorithm

When compiling with the Extended Tiny Encryption Algorithm (XTEA) block cipher option enabled, packets may be encrypted using a symmetric key. The satellite must have the shared key installed in order to decipher the encrypted packets. Used together with HMAC, it allows new security keys to be safely uploaded. This option is not used or tested in this work, but could be included at a later point.

2.5.4 CSP in Linux

When using CSP with Linux, the compiled library is linked into a userspace process. Consequently, only threads that are spawned by that process are able to access the CSP stack.

This means that all services that want to communicate on the same CSP network are required to be part of the same program, as illustrated in Figure 2.8.

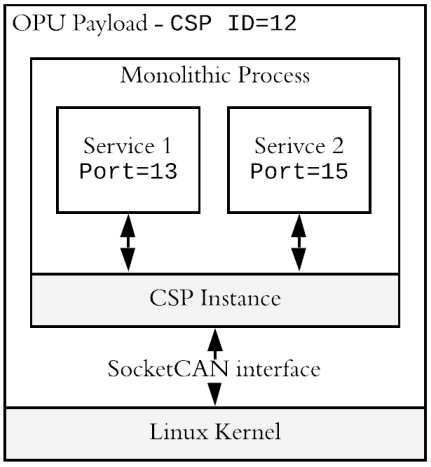


Figure 2.8: Usage of CSP in Linux. The CSP library is linked into the private memory space of the process. All services must be threads spawned from the same process.

Alternatives using various forms of inter-process communication to link multiple CSP processes have been explored [12]. The alternatives were found to cause new problems during implementation, and were dismissed as too expensive in terms of development effort.

2.6 File Systems

A file is an abstraction of information, a container of data. Computers use files to store and organise data. Various aspects of file management is touched upon in this section. Linux is used for most of the examples throughout the section because it is the operating system being used for the HYPSO mission.

Files are organised in a file system. These have a number of responsibilities, including how the file data is stored in the underlying memory technology, how the file operations are carried out, how the files are organised logically, and what set of metadata is stored along side the file.

Most often it is the computer operating system that implements and manages the file systems, although systems such as Unix's Filesystem in Userspace (FUSE) interface also allows userspace applications to implement file systems.

In addition to storing the content of a file, the file system must also store metadata. The location of the file in underlying medium must always be stored. Additional meta information that can be stored is filenames, ownership of the file and the time of creation or modification.

The File Allocation Table (FAT) file system is popular in multimedia card technologies such as Secure Digital (SD) cards, and is also used as internal storage in embedded devices. The FAT file system implements a table of entries that each represents a continuous length of storage called a *cluster*. A file is constructed as a linked list of entries, with each entry indicating which entry is the next fragment in the file, as illustrated in Figure 2.9. A special value is used to indicate End of File (EOF).

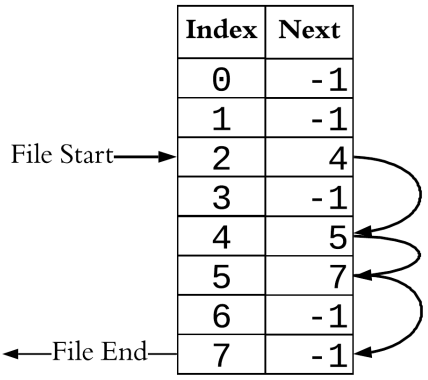


Figure 2.9: A file allocation table. Each entry indicates either the index of the next entry in the file, or the end of the file (`-1`). A file is shown to consist of the following entries: {`2`, `4`, `5`, `7`}.

Memory technologies place limitations on how data can be stored. For example, a consequence of each FAT entry only being able to hold a single address is that the maximum file size is limited by the maximum number of bytes addressable with a single address length. For example, the 32-bit version of FAT can at maximum hold a 4 GB file.

The `ext4` file system is a popular file system that is used in lots of modern Linux based operating systems. It offers multiple advanced file system features such as extents, journaling and checksum on metadata [15].

An *extent* is a long, contiguous length of storage that is defined by a starting index and a length (or end index). Compared to block allocation maps such as FAT, which are required to store the index of each constituent cluster, a scheme using extents can keep track of larger files while storing a minimal amount of metadata. While extents are suitable for tracking long contiguous files, they do not perform well for files that have a heavily fragmented layout.

A *journaled* file system aims to prevent file system corruption that results from unexpected shutdowns or system failures during file writes. The general concept is illustrated in Figure 2.10. Whenever the file system intends to modify a stored file it first records the changes that it intends to make in a structure called the *journal*. After recording the intended changes, the write of the actual file takes place. If a file is corrupted as a result of an unexpected shutdown or system failure during a write, the file system can refer to the journal to see what changes were suppose to be made. The changes can then be finalised, and the file system is returned to a consistent state. If a system failure occurs while the journal is being written, then the file is still consistent, and the corrupt journal entry can be discarded.

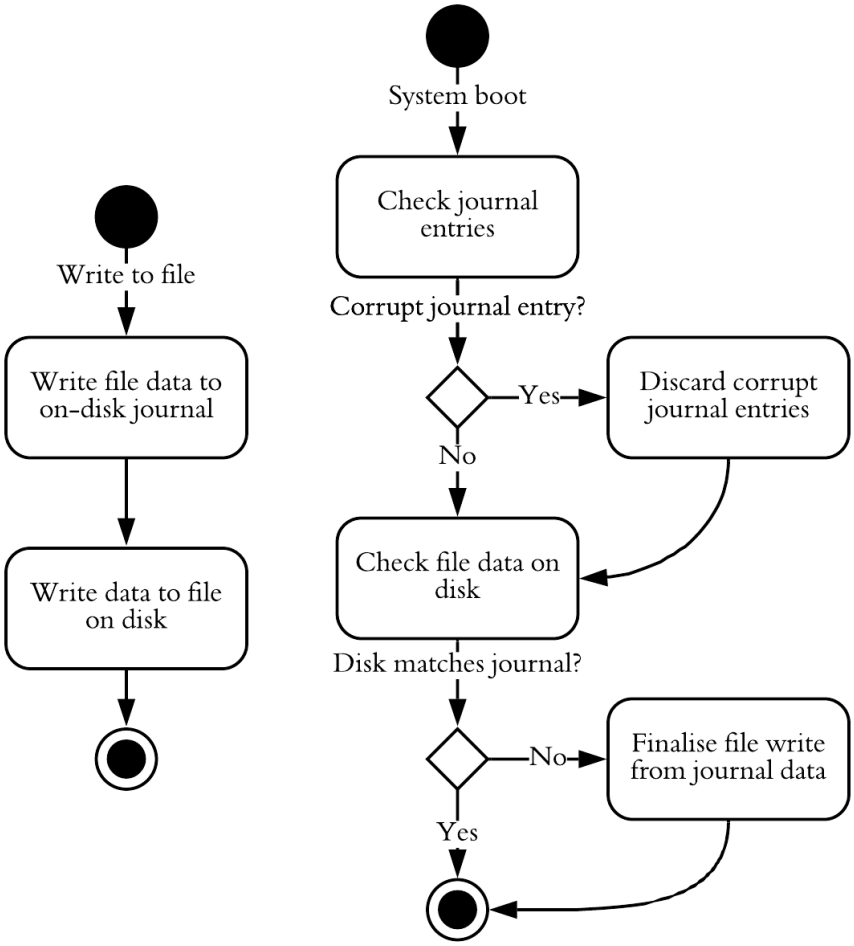


Figure 2.10: A journaled file system. Every file write is first recorded in a journal before actually writing to the file. In the event of an unexpected system failure, the journal is used to finish file transactions and restore the file system to a consistent state.

When file content data that is stored on disk becomes corrupted, only the single file is affect. When the metadata of a file is corrupted, it can affect other files and the file system itself. For example, a corrupted data allocation structure can cause a file to suddenly point inside another file. Writing to the corrupt file can then start corrupting other files. Therefore it is important to keep the metadata consistent. The file system can store a checksum for each file computed from the metadata. When corrupted, the checksum will not match the metadata. In this way, the file system can prevent itself from corrupting further files.

2.6.1 File Directories

File directories are used to organise files in a hierarchical way, like a tree. In Linux and other Unix-like operating systems, a file directory is usually represented as a path, a sequence of directory names, such as `this/is/a/directory/path/` and `another/directory/path`.

A file system will typically have a top level directory called the *root directory*. Every file in the file system must be placed directly in the root directory, or in some sub directory under the root directory. The root directory is indicated by a file or directory path starting with a `/` character, such as `/images` where `images` is a directory directly *under* or *inside* the root directory.

Operating systems usually enforce a standard layout on the root file system, which is where all the operating system files reside. The Unix directory layout has been extended and adopted by many operating systems. Linux defines root directories such as `/usr/bin/`, `/usr/lib/` and `/usr/include` as program, library and header file directories, and `/home/<username>/` as home directories for user files.

2.6.2 File Operations

Since files can store data, it is natural that they can be created, deleted, read from and written to. File system implementations are normally contained within the kernel, meaning that file operations must be communicated with the kernel. Low level file operations are implemented as kernel system calls.

Some operating systems require the user to explicitly `open` the file before reading or writing, and to `close` the file after reading or writing is finished. Opening a file allows the file system to allocate resources that are required during reading and writing. Linux will return a file descriptor number for the process to use as a handle to the opened file structure. These resources can also be allocated when a file is requested to be read or written, but it is expensive to do so for every read or write. For example, the contents of a file that is stored on a persistent medium will have to be copied into the computers working memory and cache before it can be used by an application. Explicitly telling the operating system to `open` and `close` a file limits these expensive setup and teardown operations to the start and end of the usage of the file.

The `read` and `write` operations transfers data between file and process memory by passing a buffer and a length to the kernel via system calls.

An internal file position state works as a cursor within the file, and is the point at which data will be read or written. Linux implements the `lseek` system call for manipulating the file position.

File streams

The low level system calls that operate on files are often sufficient to implement an application, but they are not always the most efficient. A source to their inefficiency is the fact that for every read and write the computer must perform a system call which can be an expensive operation. System calls are expensive because they cause the computer to do a context switch to and from a kernel thread, as well as switching the privilege mode of the machine. If the file that is being operated on must access memory on disk there will also be blocking delays while waiting for data to be fetched.

File operations can be bundled together and performed in a fewer amount of `write` and `read` calls. The implementation of file streams in the `C` programming language can do this.

A file stream provides an abstraction on top of a file descriptor. The `C` standard library implements a family of file stream functions for operating on file streams:

- `fopen` and `fclose` for creating, opening and closing files.

- `fread` and `fwrite` for writing from and reading to files.
- `fseek` and `ftell` to adjust and examine the file position of the stream.
- `fflush` to push stream buffer content from user-space to kernel.
- `fsync` to push kernel buffer content to the underlying storage medium.

When a file stream is created a chunk of memory is allocated as buffering space. When data is written to the stream, it is copied to the buffer instead of being passed to the `write` system call. When data is read from a stream, the operating system requests more data than is actually requested and copies it all into the stream buffer. For most applications this will improve performance because data is often accessed in a continuous or spatially close pattern. Since the buffers reside in the process' memory space, the operations are cheap to perform.

Certain conditions causes the stream to pass the buffer with its modified content to the operating system. This is called flushing, and happens when the buffer is full or file position jumps to a region outside the buffered region.

Memory Mapped Files

Another approach to reducing the number of system calls when reading and writing files is to utilise the *memory mapped file* mechanism provided by Linux. A file can be memory mapped by using the `mmap` system call.

Memory mapped files are mapped directly to the private address space of the process that requests the mapping. This is achieved by configuring a memory subsystem (which consists of software and hardware) to link the contents of a file to a specific address region in main memory. The first access to the memory mapped address will cause the corresponding section of the file to be paged into main memory. Every subsequent access to that region will then access the main memory instead of having to go back to the file via some system call. If a region is being accessed that has not been paged in to main memory yet, a context switch is still necessary for the kernel to fetch the new page.

File streams allocate memory in their own private memory space, which in turn is copied to kernel buffers before being written to disk. Memory mapped files, on the other side, depends on the memory subsystem to fetch and link parts of files directly into the process' memory space.

The performance of `fwrite` and `mmap` depends on the specific usage patterns. Ultimately, both `fwrite` and `mmap` end up having to switch to kernel mode, either to perform passing of buffered data, or to set up the memory controller and fetch new pages on page faults. The number of system calls depend on the buffer and page sizes. A large buffer or page size would result in wasteful copies if only accessing a small region of the file.

2.6.3 M6P File System

Original subsystems in the M6P bus employ a common, custom file system. The file system is accessible to other subsystems via a CSP service.

Each subsystem has a file store, where each file is listed with an unique identifier. The IDs are assigned starting at zero, and are increased by one for each additional file.

Files have an internal layout consisting of equally sized entries. The size of the entries is specified when formatting the file. When reformatting a file, the entry size can be changed, allowing the file to be repurposed. Within each file, each entry has an ID, which is incremented by one for each successive entry.

Each entry is internally composed of a header and a data field. The header contains a 32-bit CRC value and an entry length value. The CRC value encodes a checksum calculated from the

length field and the data field. When discussing the entry as a memory region, or when including the entry header, the term *cell* is used to refer to the entry as a whole. Each entry is stored in a cell. The structure is shown in Listing 1. The two header values occupy 6 bytes. The entry data occupies the remaining space in the cell.

```
struct cell {
  uint32_t CRC32;
  uint16_t entry_length;
  uint8_t entry_data[1..cell_size - 6];
}
```

Listing 1: Memory structure of an entry. The whole structure is referred to as a *cell*.

The file system supports file types. Log files are used to store telemetry, debug and error output, and configuration files. Static files are used as containers for larger files, and are used for uploads.

The files on the M6P subsystems are allocated on various physical storage mediums, which will affect how quickly they can be read or written. This must be taken into consideration when requesting a download or uploading a file.

2.7 File Transfer Protocols

This thesis is concerned with transferring files to and from satellite payloads. The primary function of a File Transfer Protocol (FTP) is to transfer files from one file host to another. Other functions may include the explicit creation and deletion of files, moving files and copying files. It is also useful to be able to manipulate directories, with operations such as for example *create*, *delete* and *list available files*.

The term *file store* is used to mean the medium in which files are stored. In some cases, it will be synonymous to a *file system*.

A number of FTP specifications have been studied and are used as inspiration to the work in this thesis. The following systems will be detailed:

- Trivial File Transfer Protocol (TFTP).
- The *File Transfer Protocol* (RFC-959).
- KubOS File Transfers.
- The Consultative Comittee for Space Data Systems (CCSDS) File Delivery Protocol.
- M6P File Transfer System

2.7.1 Trivial File Transfer Protocol

One of simplest implementations of a FTP is the TFTP. This protocol was defined by K. Sollins in Request for Comments (RFC)-1350 in 1992 [16]. The RFC describes a simple FTP. The protocol is defined as an application layer protocol, and depends on a functioning transport layer to deliver the TFTP packets to a specific host. The transport layer protocol being used is not required to be reliable, as TFTP handles retransmission of lost packets.

The protocol defines a set of packet types that are differentiated with unique opcodes. The packet formats are illustrated in Figure 2.11.

- `RRQ` & `WRQ`, file read and file write requests.
- `DATA`, PDU containing data fragments of files.
- `ACK`, acknowledgement of a data fragment.
- `ERROR`, containing a situational error code.

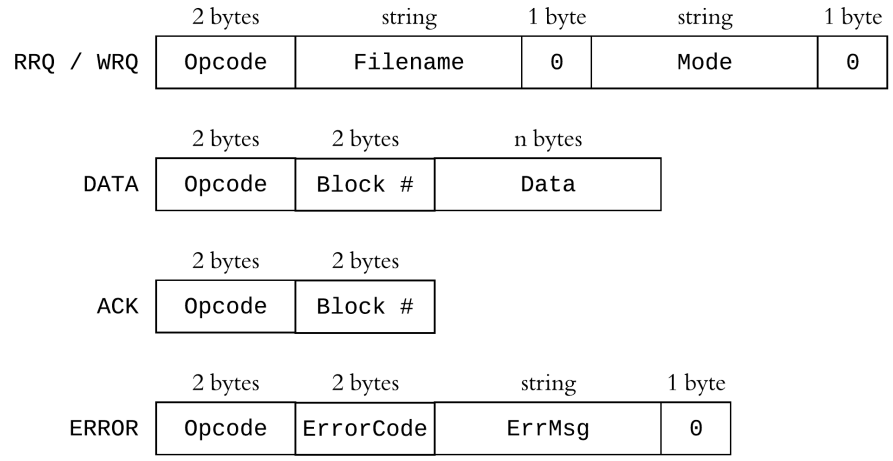


Figure 2.11: Packet formats in TFTP as defined in RFC-1350.

A transfer is initiated by sending a `RRQ` or `WRQ`. The `RRQ` and `WRQ` packets contain a string encoding the file name of the file that is being requested to be read from or written to. As seen in Figure 2.11 , there is also a mode field. The mode field must contain one of the following strings {`netascii`, `octet`, `mail`}. The modes `netascii` and `octet` both indicate that the filename field indicates an actual file to be read from or written to. The `netascii` mode demands that the transferred data be translated from network byte order to host byte order upon reception. This is necessary when a host operates with a different character encoding from the network. The `octet` mode tells the recipient to send the data in the same encoding as it is stored, regardless of which character encoding is being used. The data should be consistent if it is transferred back using `octet` mode. The `mail` mode is only valid for the `WRQ` opcode, and indicates that the received data should be sent as an e-mail from the receiving host, with the filename field used as an e-mail address.

A successful `RRQ` is immediately followed by a `DATA` response, while a `WRQ` is immediately followed by a `ACK` response. When transferring the file, TFTP uses the *stop-and-wait* strategy described in Section 2.2.1. Each `DATA` packet is acknowledged with a `ACK` packet before sending the next `DATA` packet.

In the case of a lost `DATA` packet, the sender will timeout because no `ACK` packet is received, issuing a retransmission of the lost `DATA` packet. If a `ACK` packet is lost, then the sender will also time out and resend the previous `DATA` packet. The duplicate `DATA` packet is detected by examining the sequence number, which TFTP calls a *Block* number.

Each TFTP packet can be up to 512 bytes long, and any `DATA` packet which is short of 512 bytes is interpreted as the last block in the transfer. This protocol effectively has a sending and receiving window of length one, meaning that it should not be possible to mistake one `DATA` packet for another.

Error packets are used to indicate status when requesting `RRQ` and `WRQ`. A missing file, or incorrect permissions will result in an `ERROR` packet with an appropriate error message and error code. Errors are also returned during transmission if the sender repeatedly times out.

2.7.2 The File Transfer Protocol RFC-959

The generically named *File Transfer Protocol* is defined in RFCs-959 [17]. The name *RFC-959* is used to refer to the protocol, to avoid confusion with the acronym for a general file transfer protocol.

The RFC-959 protocol is normally hosted on internet servers and allows multiple users to store and retrieve files.

RFC-959 is a connection oriented protocol, meaning that a connection to the server has to be established before making a file request. A reliable connection is assumed, such that the protocol does not need to worry about retransmission of individual packets.

A connection is first established between client and server. This connection is called a *control communication connection*, and handles requests and responses. On this connection, a range of operations may be requested. Some examples of commands are:

- `RETR` and `STOR` for retrieving and storing files.
- `RNFR`, `RNTO` and `DELE` for renaming (moving) and deleting files.
- `CWD` and `CDUP` for changing the current directory.
- `MKD`, `RMD` and `LIST` for making, deleting and listing directories.
- `PORT`, `TYPE` and `MODE` to change data port and type, and transfer modes.

When establishing the connection one can also prompt the user for a user name and password. Alternatively, user and account details can be configured with dedicated commands after connecting.

When a file transfer is requested, a separate data connection is established. All file data is moved over this connection, while protocol commands may still be exchanged on the control connection. This architecture is illustrated in Figure 2.12.

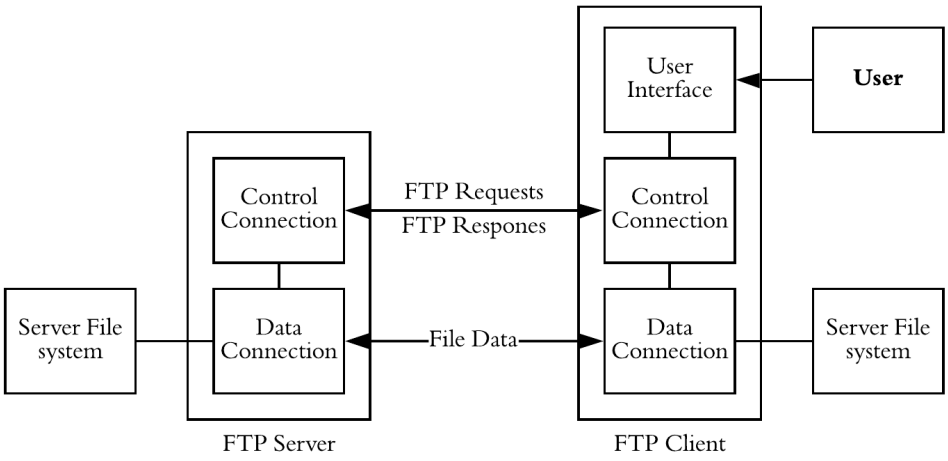


Figure 2.12: Connection architecture of RFC-959 FTP. A control connection is first established between client and server. All file data is transfered over a separate data connection. Illustration taken from RFC-956 [17].

In a similar way to TFTP, different data types can be specified. In `ASCII` mode, the data is translated from the host's representation of characters to a standardised 8-bit ASCII format.

In `Image` mode, each byte is sent unmodified, so as not to mangle image or program files. There are also other modes that take into account the byte alignment in data words of different length.

There are also several data transfer modes that are allowed in RFC-959. *Stream mode* is the simplest of the transfer modes, and is lacking in terms of reliability. The *block mode* breaks the file into *block* segments that are prepended with a segment header. The *compressed mode* employs a simple compression algorithm to compress the file on-the-fly before sending it.

In *stream mode*, the file data content is sent directly over a TCP connection, with the TCP implementation using the sliding window *go-back-N* as its ARQ strategy. In this way, the *stream mode* achieves reliable, in-order transfer on a packet to packet basis. The EOF is implicitly indicated by the termination of the data connection. The *stream mode* can therefore not differentiate between a successful termination initiated by the sender and an unexpected termination.

In *block mode* each file segment is prepended with a header. The header encodes special codes for indicating transfer status, such as EOF. In this way, if a data connection is terminated, the receiver will know that it has not yet received the EOF indicator, and can issue a `RESTART` command to resume an unfinished transfer. The `RESTART` mechanism is only available for *block mode* and *compression mode*.

The *compressed mode* can decrease the file size before transmission. It implements a lossless run-length encoding. This means that long repetitions of bytes are replaced by a structure defining the byte to be repeated, and a number defining how many times the byte should be repeated.

Since RFC-959 depends on TCP, it automatically gains the congestion control mechanisms of TCP. Packet loss is monitored via ACK timeouts, and is used as a control input to adjust the sending characteristic of the protocol. The sending frequency is strongly linked to the sending window size. Generally, packet loss will reduce the sending window and successful ACKs tend to increase the sending window. Several algorithms can be used in tandem to improve the channel utilisation.

2.7.3 Kubos File Transfer

KubOS is a flight software framework for small satellites. It implements a number of protocols and service modules that can be integrated into an On-board Computer (OBC). All protocol details described in this section are obtained and compiled from public *KubOS* documentation [18]. *KubOS* features include:

- A File protocol and service.
- A Shell protocol and service.
- A Communication service.
- A Telemetry Database.
- A Mission Application service.

The *KubOS* file protocol encodes all of its command messages in the Concise Binary Object Representation (CBOR) format. This format encodes *name-value pairs*, in the same way as JavaScript Object Notation (JSON), but with a smaller footprint which makes it suitable for resources limited space applications.

The file protocol prepares a file for transfer by splitting it into chunks whenever an `export` command or `import` message is received. Each chunk is then transferred as individual PDUs. The file is reassembled at the receiver side when all chunks are received.

Automatic Repeat Query

Initially, all chunks are transmitted. When no more chunks have been received for some period of time, the receiver will send a status message back to the sender. If all chunks have been received, an ACK is sent to indicate that all chunks have been received. In the case that a chunk has been lost in the unreliable network, a NACK will be sent.

The NACK message includes a list of ranges that indicates which chunks are still missing. Upon reception of a NACK, the sender retransmits the indicated missing ranges. This repeats until no more chunks are missing and an ACK is successfully sent and received.

KubOS maintains a session state for each transfer by creating a file directory for each transfer. The hashsum of the file being transfered is used as the name of the directory. This way two files with the same name but different content can be sent without interfering with each other's transfers. On the sender side, the directory is populated with the chunks to be transferred. Chunks are saved to the directory as they are received. The chunks are named with the same index as their position in the file.

2.7.4 CCSDS File Delivery Protocol

CCSDS File Delivery Protocol (CFDP) is a massive specification for file transfer systems intended for space applications. It was first defined by CCSDS in 2002, and has since been superseded by the latest edition from 2007 [19].

The specification assumes a single underlying communication layer, which it refers to as the *Unitdata Transfer* (UT) layer. This layer may or may not be reliable.

Protocol Classes

The specification defines several classes of file transfer systems:

- *Class 1* - Unreliable Transfer.
- *Class 2* - Reliable Transfer.
- *Class 3* - Unreliabel Transfer Via One Or More Waypoints In Series.
- *Class 4* - Reliable Transfer Via One Or More Waypoints In Series.

The specification does not attempt to define a single FTP system that should be used in all missions, but rather a clear and concise definition of components that can be used to construct a system that fulfils the mission needs. Reliable transmission is a component that can be included in such a system, but as *Class 1* indicates, it is not a requirement to do so in order to be CFDP compliant.

File Store Operations

The specification also defines a set of file store operations, and assumes that the underlying storage medium is able to provide them:

- *Create* and *Delete* files.
- *Rename* files (including *move* functionality).
- *Append* and *Replace* files.
- *Create* and *Remove* directories.

Unacknowledged File Transfer

Before a file can be transferred, a *Metadata* PDU must be exchanged with the receiving entity. This packet contains the length of the file, the source and destination file name and various flow control options.

In *Unacknowledged Mode* the sending entity will simply dispatch the whole file as file segments. The arrival of the last segment is indicated by an EOF PDU that is dispatched by the sending entity. The reception of the EOF PDU at the receiving side indicates the end of the transfer. The receiving entity offers no response as to whether all segments have been received, or whether the EOF has been received.

Automatic Repeat Query

If the underlying transport layer is not reliable, then CFDP can be run in *Acknowledged Mode*. Multiple ARQ strategies (or *retransmission strategies* as the specification names them) are defined and can be chosen from or even switched between to fit the exact need of the mission.

In *Acknowledged Mode*, *Lost Segment Detection* procedures monitor the PDUs as they arrive and are stored. Different detection modes are defined, but the main distinction is whether NACKs are sent immediately upon detection of a missing PDU, or whether they are accumulated and deferred until a later moment.

Every NACK PDU contains a file scope defined by a starting and ending offset. Following this scope declaration is an array of ranges that indicate which segments of the file are missing, as illustrated in Figure 2.13. Upon reception of a NACK, the sending entity can retransmit the indicated ranges.

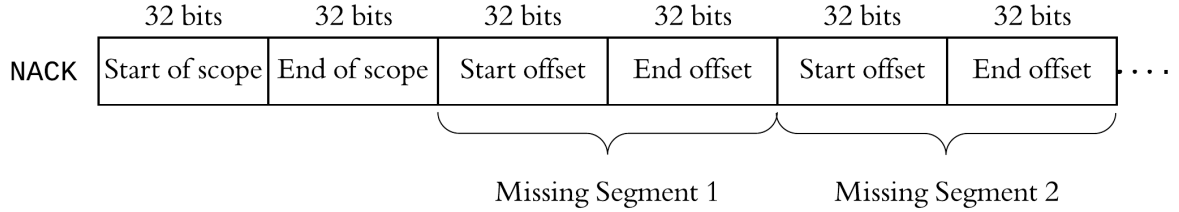


Figure 2.13: Contents of a CFDP NACK. The NACK includes a set of ranges that indicate which segments of the file are still missing.

Store and Forward

Class 3 and *Class 4* describes protocols where *waypoints* collaborate to perform file delivery. This is necessary when the source and destination entities can not communicate directly. This could be because the communication chain depends on multiple links that are available at different times.

Delivery via *waypoints* is achieved by using a *store and forward* strategy. An example is shown in Figure 2.14. The original source entity will first transfer a file to a proxy entity. It will then request that the proxy entity transfers the file to the destination entity. The proxy entity will transfer the file to the destination entity. In *acknowledged mode*, the destination entity will respond with a ACK to the file transfer, and the proxy (or multiple proxies in a chain) will propagate this indication of completion back to the original source entity.

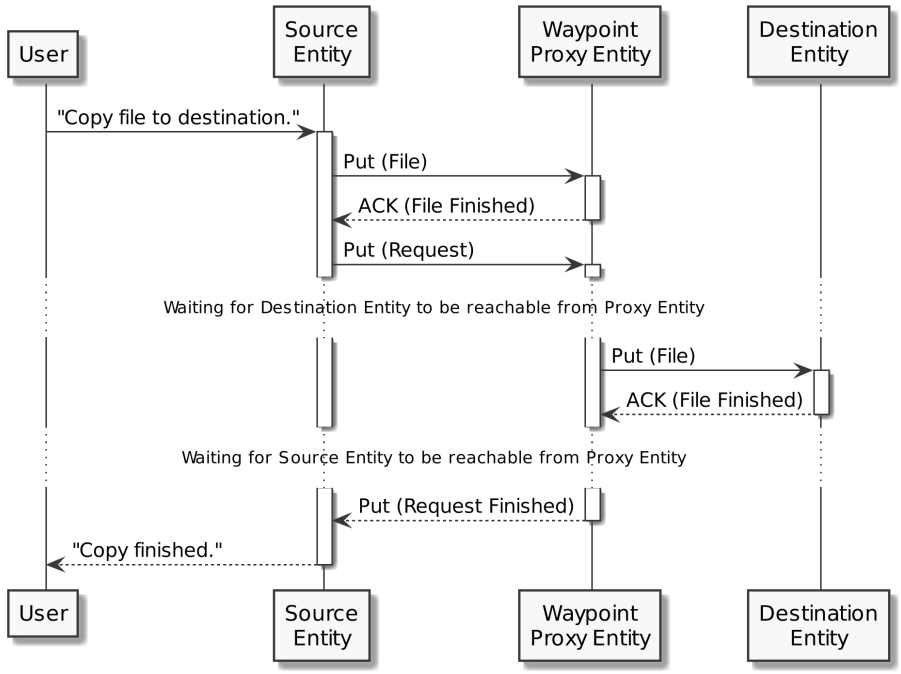


Figure 2.14: Illustration of the *store and forward* concept. A proxy entity stores a file and is requested to forward it to a destination entity.

An important property of this system is that the chain of waypoints is not required to be connected all the way from end to end at any moment. Files can be stored for a period of time and forwarded when a connection to the next waypoint has been established. This may be after the connection to the previous waypoint has disappeared. This is also illustrated in Figure 2.14, where the proxy entity must wait for the destination entity to be reachable, and then wait for the source entity to be reachable once again.

2.7.5 M6P File Transfer

NA provides file transfer capabilities for their M6P platform, based on the file system described in Section 2.6.3.

Each M6P subsystem hosts a file transfer service that listens for CSP packets on CSP port `10`. The services respond to a number of file requests. The available commands are described in the following sections.

`file info`

A file info request returns file metadata. The file is specified by a file ID. The following information is returned on a successful response.

- *File ID*: ID number of file being requested.
- *Last entry ID*: most recently written entry (log file).
- *Total entries*: number of entries available to be requested.
- *Cell size*: byte size of the memory region holding one entry.
- *Used cells*: number of cells that are being used.

- *Max cells*: total number of cells available.
- *Sector quantity*: number of sectors allocated for the file.
- *Sector size*: byte size of each sector.
- *File type*: Can be a `LOG` or `STATIC` type.
- *File name*: ASCII file name.

A file listing can be constructed by requesting `file info` for a range of file IDs.

`file clear`

A file clear request removes the contents of existing entries. It will reset a `LOG` file, and discard the data of a `STATIC` file.

`file format`

A file format request prepares a `STATIC` file for writing. It will allocate a specified number of entries with a specified entry data size.

The entries are allocated in the memory region of the file specified by its file ID. The specified file must have enough space to hold the specified number of entries. The 6 byte entry overhead must be considered when requesting a format.

When uploading, the entry size is limited by the size of the data field in a CSP packet.

`file check`

A file check request provides information about the condition and progress of a `STATIC` file. A file can be checked for data integrity or for presence of data. The former can detect data corruption, and the latter is useful for checking which parts of a file still need to be written to or be transferred.

The response contains a bitmap that encodes the status of the requested entry range. Each bit corresponds to one entry. A bit value of `0` indicates bad integrity or no data presence, while a bit value of `1` indicates good data integrity or data presence. Checks on large files must be split into multiple requests in order to fit the returned bitmaps into individual CSP packets.

`file download`

A file download request initiates a download of a range of entries from a specified file. The request specifies a period which determines the sending frequency. A maximum duration for the whole transfer is also specified. Additionally, the maximum data size of the data packets must be specified, and is limited to the MTU of the CSP network.

A successful request is acknowledged with a response that confirms the requested file ID. After a one second delay, a stream of data packets is sent. The stream of packets stops when the whole range of requested entries has been sent, or when the specified duration has timed out, or when a `cancel` request is received.

The file stream is made up of a series of *stream packets*, which contain a header and data blocks, as illustrated in Figure 2.15.

The header encodes which file that the stream originates from, the ID of the first entry in the stream packet, and optionally the byte offset of the first entry in the stream packet.

The data field of the stream packet is constructed from blocks that contain entries. Each block has a length field that encodes the length of the block. The block data field contains an

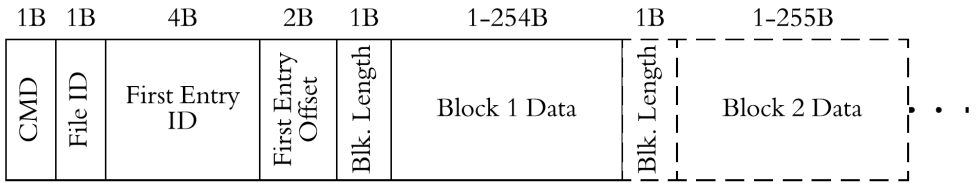


Figure 2.15: Format of a stream packet. The header encodes which entries are being sent and which file that they come from.

entry cell, complete with CRC field, length field and data. The data field of the stream packet is completely filled with blocks as long as there are more entries to be transferred. This results in entries being fragmented, and blocks containing partial entries. When an entry is fragmented, the remainder of the entry is sent as the first block of the next stream packet. The *first entry byte offset* field is then set appropriately to indicate that the first block is the remainder of a fragmented entry.

Entries that fail to be read are indicated with error blocks. These blocks have a fixed size and are distinguish from ordinary blocks by their length field being set to `0xFF`. The error blocks contain a single byte error code.

The received packets are not acknowledged, meaning that this is an unreliable form of transfer. After having received a stream of packets, it is recommended to inspect the integrity of the received data. Corrupt and missing entries can then be requested with a new file download request for the appropriate range.

This form of ARQ is similar to the the ones used in *KubOS* and CFDP.

`file cancel`

A file cancel request aborts an ongoing download. This request is useful for situations where an incorrect download has been requested. Whole satellite passes could go wasted if a wrong download was unable to be cancelled.

`file upload`

A file upload can only be performed for a `STATIC` file.

The target file must be formatted with the exact entry size and amount of entries of the source file. When a file has been formatted it will readily accept uploaded data, there is no need to explicitly request an upload.

File data is sent in stream packets, like in Figure 2.15. In contrast to the download procedure, the data block in the upload stream packets may only contain the data field of the entry cell. Only one entry block may be put in each stream packet, and the blocks may not be fragmented. This means that the target file must be formatted with an entry size that can fit within a single stream packet.

After having transmitted all entries, `file check` requests can be used to determine whether any entries were lost. Lost entries can then be transmitted.

Part II

Design & Implementation

This part contains details of the design and implementation of the OPU payload and of a FT system.

Contributions

The work contributing to this thesis is listed below:

- Specification and design of BOB hardware for the OPU payload (Section 4.1.4).
- Design and complete implementation of a FT system (Chapter 5).
- Design and implementation of a Command Line Interface (CLI) program for payload interfacing (Section 6.4).
- Design and implementation of a shell service for the payloads (Section 6.4.2).
- Design and implementation of TFTP module, which is used in comparison to measure the performance of the FT system in Section 6.6.

Proposed designs are presented in the following sections. Software implementations are kept within a Git repository [20]. The software implementation amounts to 10308 lines of C code produced for this thesis. Specification and design files for hardware are included as appendices.

Chapter 3

Requirements

This chapter outlines the main design requirements relevant for this work. The requirements are sourced from HYPSO mission design documents.

The HYPSO project maintains a central requirements document titled *Requirements HSI SmallSat* [21]. This document contains mission objectives, mission requirements and system requirements for the OPU payload.

Data budgets for the SDR payload are acquired from a *SDR-DR-001 System Design Report* document [22].

3.1 Satellite Bus Requirements

The HYPSO project is committed to using the M6P CubeSat platform from NA. The mission requirements state:

> **IF-001**: The payload shall comply with NanoAvionics mechanical and software ICD.

This requirement necessitates the use of the CSP network stack and CAN-bus protocol on the HYPSO payloads. A result of this is that the FT system must be implemented on top of the CSP network stack.

3.2 Downlink Requirements

The HYPSO mission requires downlink capabilities for different types of files. The National Aeronautics and Space Administration (NASA) model for data products in remote sensing is used to denote different types of data sets [23]. The HSI mission requirements state:

> **MS-0-011**: Shall downlink 1 hyperspectral image in L1A data format containing detectable optical signatures (Chl-a, CDOM etc.) to be processed on ground.
>
> **MS-0-012**: Should downlink 1 operational hyperspectral images in less than 3 hrs after successful onboard dimensionality reduction, classification and target detection with certainty of 10 % of positive optical signatures (Chl-a, CDOM etc.) to be ground truthed.
>
> **M-2-018**: L1A data shall have no more than 2234 frames and be less than 422.15 MB.
>
> **M-2-019**: Operational data should be less than 34.27 MB.

The two types of HSI data that are mentioned are:

- L1A data, which consists of image data. The image may have had its pixels *binned* to produce a lower resolution, and may be compressed. Timing and attitude data are also appended. According to requirement **M-2-018**, the L1A data set may not be larger than 422.15 MB.
- Operational data, which corresponds to L4 data. These data sets are heavily processed and compressed, and will only contain the information that is most relevant to the operational situation. According to requirement **M-2-019**, the operational data set may not be larger than 34.27 MB.

Other files such as telemetry data and output logs from services must also be downloaded. The mission requirements state:

> **MS-0-014**: Shall downlink house-keeping telemetry data for at least 1 pass per day.
>
> **M-2-026**: S/C shall communicate to ground and downlink house-keeping telemetry data of up to 200 kb for at least 1 pass per day.

From the perspective of a FT system, the transfer of an image file is the same as the transfer of a telemetry file or log. The image files are likely to be of a larger size and therefore take longer to transfer. Requirement **M-2-026** suggests a size for telemetry data that is significantly smaller than any of the image data sets. By designing the file transfer system to handle image files it should also be able to handle smaller files.

The SDR payload performs radio measurements. The SDR *System Design Report* [22] states a secondary mission objective:

> **SDR-SMO-1**: To measure downlink channel in UHF using sensor node antennas.

These measurements produce relatively large amounts of data. The data must be downlinked in order to be analysed. The SDR data budget estimates the size of various measurement files [22], the largest of which is a global heatmap. Two examples of files are:

- *Same band heatmap and stats in Arctic*: 22.75 MB
- *Same band global heatmap and stats*: 136.37 MB

3.3 Uplink Requirements

The HYPSO mission also requires uplink capabilities. Mission requirements state:

> **MS-0-013**: Shall enable flexible mission planning & scheduling and subsystem updates through successfully integrated uplinked mission data, FPGA programming logic and codes.
>
> **M-1-012**: Shall incorporate uploaded mission data prior to imaging/operations that includes updated commands (TT&C), reprogrammed code, FPGA programming logic, schedulers, parameters, radiometric and geometric coefficients
>
> **M-2-015**: Mission plan data and TT&C shall be updated on-board through uplinked data in the same pass in minimum 5 min prior to the observations are made

SBUS-3-017: S/C software & scheduling (including payload code) shall be open for updates (mission operations; change in objectives; bug fixes) and upgraded (functionality and efficiency) after launch.

Requirements **MS-0-013**, **M-1-012** and **M-2-015** state that mission plan and operations scheduling information must be uploaded to the satellite. Requirements **MS-0-013**, **M-1-012**, and **SBUS-3-017** indicate that it must be possible to deploy software upgrades to the HSI payload, which means that new software must be uploaded to the satellite.

These requirements specify that a FT system must have the capability to transfer files from the ground to the satellite.

M-2-014: Nominal ground station for uplink and downlink shall be NTNU Trondheim and 2 additional ground stations for downlink shall be KSAT Tromsø and Svalbard

Requirement **M-2-014** states that multiple ground stations should be able to downlink data from the satellite. This is requirement for the HYPSO communication network. As long as the communication network is maintained as a single CSP network, the requirement should not affect the design or functioning of the FT system.

The SDR *System Design Report* lists a secondary mission objective:

SDR-SMO-4: The system shall allow for update in flight.

In addition to **SDR-SMO-4** requiring the SDR payload to be able to receive software upgrades, it must also be able to receive python scripts for each new *GnuRadio* experiment.

3.4 Quality Requirements

In order to meet all top level mission objectives, time restrictions are introduced. The mission requirements state:

M-1-015: L1A data product shall be downlinked in less than 24 hrs and be ground truthed

M-1-016: Operational data shall be downlinked and ground truthed in less than 3 hrs

The requirements **M-1-015** and **M-1-016** are fulfilled by designing a file FT that can transfer files at a sufficient data rate. The primary factors are the data rates of the individual communication links in the CSP network. Packet overhead introduced in the FT system will further decrease the effective data rate.

There are no lower bounds on how fast the upload routines for software upgrades are required to be. Requirement **M-2-015** suggests that mission plans and Telemetry, Tracking and Command (TT&C) must be uploaded in the same pass. The required effective uplink speed therefore depends on the size of the mission plan data and TT&C.

The SDR mission documents do not place any restrictions on downlink or uplink time [22].

Chapter 4

Payloads & Communication Architecture

This chapter details the HYPSO payloads and supporting communication architecture. Some of the details are results of work performed in this thesis, such as the BOB in Section 4.1.4, and configuration of *PetaLinux* in Section 4.1.5.

4.1 Onboard Processing Unit

A part of the work for this thesis has consisted of configuring and installing the OPU system, as well as specifying and designing hardware for the OPU. As illustrated in Figure 1.3, the OPU payload encompasses the HSI and RGB imagers, as well as BOB and a PZ System on Module (SoM) (from *AVNET*) [24].

4.1.1 Imagers

The hyperspectral imager is constructed from optical parts from Thorlabs and *Edmund Optics* [5], and digital parts from Imaging Development Systems (iDS). The RGB imager is an integrated COTS camera from iDS.

The HSI camera is developed by Prof. Fred Sigernes at The University Centre in Svalbard (UNIS) and is produced with COTS sensor and optics. Some of the mechanical parts are manufactured by the SmallSat Lab. The image sensor is an UI-5260CP-M-GL system from iDS. A prototype of the HSI imager is depicted in Figure 4.1.

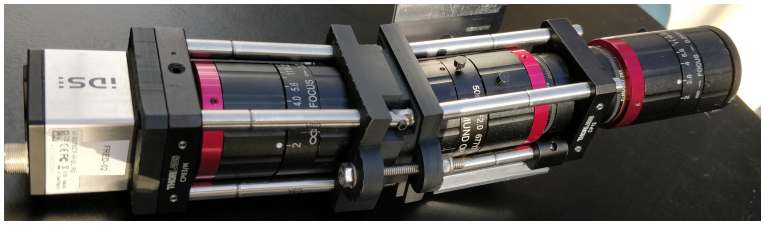


Figure 4.1: Engineering model of HSI camera. The gray box to the very left is a CMOS image sensor from iDS. The optical parts include lenses and a diffraction grating from *Edmund Optics*, and adaptors, mounting parts and a slit from *Thorlabs* [5].

The HSI imager has been selected to have a Gigabit Ethernet interface. Development and testing were carried out on a variant that had an Universal Serial Bus (USB) interface, but was

found to be too slow to transfer sufficient amounts of data. The image resolution in the direction of the push broom sweep is directly linked to the Frames Per Second (FPS) captured. In order to achieve a high enough FPS, the USB camera was changed to one with a Gigabit Ethernet interface, but still using the same image sensor.

The decision to change the HSI camera impacted the design of the BOB, as detailed in Section 4.1.4.

The auxiliary RGB camera consists of a UI-1250LE unit from iDS. The selected RGB camera has an USB interface.

4.1.2 PicoZed System-On-Module

Upon joining the project, the AVNET PZ SoM had already been selected as a computing unit for the OPU. This unit is a so called SoM, meaning that it is a *partially* integrated hardware system that is intended to be used as a module in a larger system.

The PZ SoM is based on a Zynq-7000 System on Chip (SoC) from Xilinx. This SoC combines a Processing System (PS) part with a Programmable Logic (PL) part. The PS part contains a dual ARM core. The PL part contains a generous amount of Look-Up-Tables (LUTs), Flip-Flops, adder circuits, block Random Access Memory (RAM) units and Digital Signal Processing (DSP) blocks.

The PL on the Zynq-7000 SoC is an instrumental part in implementing the image processing techniques that the HYPSO mission requires. Without the hardware acceleration that the PL provides, the HSI data can not be processed and compressed at a fast enough speed to fulfil the mission requirements.

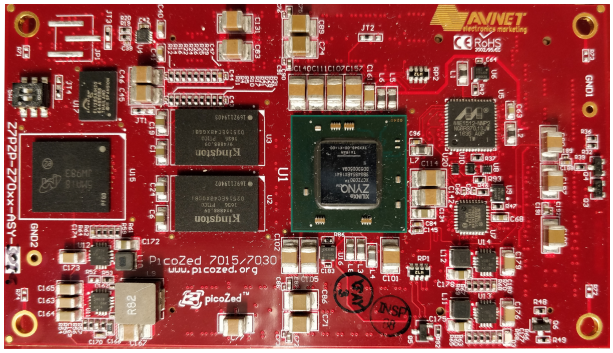


Figure 4.2: PicoZed System-on-Module from *AVNET*.

Memories

The PZ SoM has two non-volatile memories. It is equipped with a Cypress S25FL128S chip and a Micron MTFC4GMDEA-4M chip [24].

The S25FL128S chip is a 128 Mb NOR flash device, while the MTFC4GMDEA-4M is a 4 GB NAND Embedded MultiMediaCard (eMMC) flash device. The smaller NOR flash is interfaced with Quad-SPI (QSPI), while the eMMC NAND flash is interfaced with a standard SD controller. Both of these have internally managed ECC that can correct single bit errors.

4.1.3 ZedBoard Development Kit

Development of OPU software has primarily been carried out by testing on a *ZedBoard* development kit from AVNET. This kit contains a Zynq chip, similar to the one on the PZ SoM,

with only a few differences. It offers the same interfaces as the PZ SoM, but makes them more available by having connectors for every interface. The *ZedBoard* development setup is depicted in Figure 4.3.

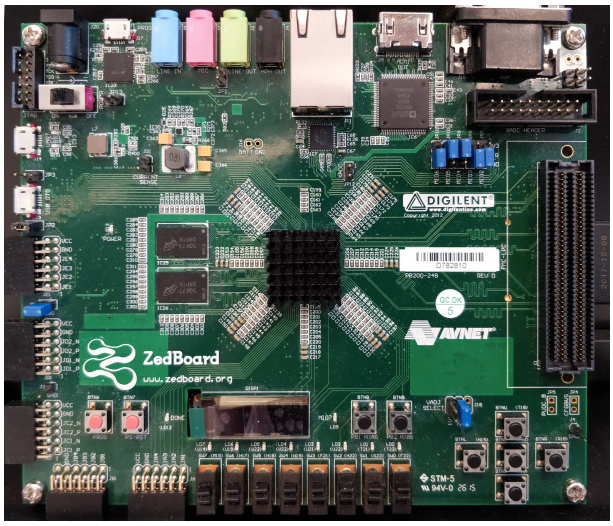


Figure 4.3: ZedBoard development kit used for testing OPU software.

4.1.4 Breakout Board

When building a system with SoM components, it is necessary to have a BOB or *carrier card* which provides a mechanical and electrical connection to the SoM. The BOB must provide power to the SoM.

The work performed for this thesis includes the specification and design of functions and interfaces that the BOB requires, as well as coordination of the implementation and production of the BOB. The specification and design is considered a contribution of this thesis, while implementation of the Printed Circuit Board (PCB), and logistics of manufacturing was handled together with NTNU SmallSat Lab staff.

A previous version of the BOB was used as a baseline, and only the differences from the baseline version are outlined in this thesis. The specification and design documents for the new BOB are included in Appendix B.

Interfaces

As illustrated in Figure 1.3, the OPU requires a BOB to connect the PZ SoM to the two imagers, and to the rest of the satellite via the PC. The PZ SoM has three *Micro Headers* that carry all interfaces. These headers slot into a corresponding set of connectors on the BOB.

The HSI camera requires the BOB to have an Ethernet connection, and the RGB camera requires the BOB to have an USB connection. Both of these were added in the new design.

A CAN bus transceiver had to be added to facilitate the connection to the M6P payload CAN-bus, `CAN2`. A few CAN transceiver chips were assessed. The chip MCP2562 was chosen because it is able to interface the Zynq-7000 SoC's 1.8 V logic pins directly.

The Pulse-Per-Second (PPS) timing signal that is provided by the GPS module on the M6P platform is required to achieve time synchronisation of high accuracy. A General-Purpose IO (GPIO) header was added to the BOB to connect to the PPS signal to the SoC. The PPS signal can be used by the Network Time Protocol (NTP) in Linux to perform the clock synchronisation.

The BOB also holds a SD-card which provides additional long-term storage memory. A SD controller chip was present on the previous BOB version, and is kept unchanged. Together with the PZ SoM, the OPU contains three different non-volatile memories:

- 128 Mb NOR Flash over QSPI.
- 4 GB NAND Flash, eMMC over SD.
- A SD-card, with 32 GB storage.

Removing Integrated Image Sensor

The previous version of the BOB had the additional responsibility of housing the image sensor of a previous version of the HSI camera. Since the decision to use the HSI V6 camera, the image sensor is integrated into the camera structure, so the next version of the BOB is not required to house the image sensor.

By removing the image sensor, a sizeable area of the PCB was freed up, and two voltage regulators could be retired, since their only purpose was to supply voltage levels that were only needed by the image sensor.

Power Supplies

Various components on the PZ SoM require a voltage level of 5.0 V, and the Multiplexed IO (MIO) banks in the Zynq-7000 SoC require voltages of 3.3 V, 1.8 V.

The BOB receives unregulated battery voltage `VBAT` from the EPS. This voltage depends on the level of charge in the EPS battery bank, and varies between 6.0 V and 8.4 V [25]. The voltage is regulated down to a stable 5 V, which in turn is used to synthesise the required 3.3 V and 1.8 V.

With the new version, the addition of using the Gigabit Ethernet transceiver on board the PZ adds a requirement of supplying 1.2 V and 1.0 V to the PZ SoM. The two voltage regulators that were made obsolete by removing the old image sensor have been repurposed to produce the voltages for the Gigabit Ethernet transceiver instead.

The HSI camera requires a input supply of 12 V - 24 V. Since this is higher than the main input voltage to the BOB (`VBAT`), it must either be boost regulated from the `VBAT`, or acquired from the EPS. In order to avoid another power synthesis chain on the BOB, an additional input connector for a 12 V supply was added. The EPS is then configured to output a 12 V channel, which is also used for the BOB.

4.1.5 Operating System

The Zynq-7000 SoC has dual ARM cores that can run a conventional operating system. This is necessary to run the CSP network stack.

The OPU payload will run a Linux based operating system. The operating system was chosen during spring 2018, during a time when much of the initial work with the *ZedBoard* development setup was being performed.

There are a few reasons why a Linux based operating system was chosen. The iDS cameras that are a part of the HSI payload require a proprietary driver, which is only available for *Windows*, *Linux x86* and *Embedded Linux ARM*.

Benefits of Using Linux

In an enquiry into the current use of Linux in spacecraft flight software, Leppinen outlines a few benefits and drawbacks [26]:

- The Linux code base is considered very reliable because of its wide adoption and high level of revision.
- There is a large community of developers for Linux, making it easier to find developers to recruit for the project.
- Parts of the testing can be carried out inside the development environment, as long as the development machines run Linux.
- Ready made libraries, protocols and applications can be selected from a large collection of free software and quickly integrated into the system.

Drawbacks of Using Linux

Since the Linux system is being used for so many different applications, it has grown to become a relatively complex system with a large number of customisation parameters. The large number of customisation options is what allows Linux to be used for so many different applications, but can also cause problems because there are more components to be tested, certified and debugged.

Leppinen also mentions a few drawbacks [26]. Linux has not been designed to be a Real-Time Operating System (RTOS). However, the FT system does not have any hard real time requirements.

The closest thing to a real time requirement for the OPU is the timing and control of the HSI imager, which is necessary to achieve a desired FPS when capturing frames. This task is not considered in this thesis.

With CLI access to a Linux system, and because of the large number of customisation parameters, it is possible to put the system into an unrecoverable state. It is difficult to freeze, lock or remove all of the extensive Linux features that makes this possible, but different strategies can be put in place to mitigate the risk and effect. Fallback boot images can be installed to allow the system to boot into a safe mode if the working copy of the boot image becomes misconfigured.

Linux System

A few variants of Embedded Linux for the Zynq-7000 series were assessed before *PetaLinux* was chosen. *PetaLinux* is not a Linux distribution in itself, it is a framework for customising and building a distribution. The framework is maintained by Xilinx and provides Linux support for several of their architectures, including the *Zynq-7000* SoC, *Zynq UltraScale* and *MicroBlaze* [27].

The *PetaLinux* framework provides the following components for the OPU system:

- A First Stage Boot Loader (FSBL) for the Zynq architecture.
- The popular Second Stage Boot Loader (SSBL) U-BOOT.
- Scripts for configuring and cross-compiling the Linux Kernel.
- Libraries and applications for Linux, such as `busybox` and `canutils`.

PetaLinux is based on the *Yocto* framework. The *Yocto* project maintains a library of recipes for configuring and building kernels, libraries and applications. These recipes are combined into a project, which is interpreted by the build tool *BitBake* to perform the configuration, compiling and integration of the components into boot images and root file systems. *PetaLinux* works as a wrapper around *Yocto*, adding additional recipes that are required by the Xilinx architectures. For more information on *PetaLinux*, and how to use it, see [27].

Boot sequence

The Zynq holds a small section of on-chip Read-Only Memory (ROM) that contains startup code. The code initialises Central Processing Unit (CPU) states, and searches for and loads the initial boot image. The boot image is usually a FSBL which performs board-specific peripheral initialisation, before handing over execution to a SSBL.

The FSBL can be executed directly from flash if it is located on a QSPI flash device, such as an SD-card. Otherwise, the image is first loaded into the On Chip Memory (OCM) before it can be executed [28].

When using PetaLinux, the FSBL is automatically generated, and *U-BOOT* is used as SSBL. The SSBL loads the Linux kernel into main memory and initiates its execution. The FSBL is compiled into a `BOOT.BIN` file, while the SSBL and Linux kernel image is compiled into a single `image.ub` file. Further details on booting the *ZedBoard* and PZ SoM can be found in [27].

File systems

The Zynq BootROM code is hard coded to read the first partition on the selected boot device as FAT. The boot images are therefore required to reside on a FAT file system. The boot device can be selected by setting a hardware switch on the PZ SoM.

Because the OPU payload runs Embedded Linux, a Linux compatible file system is being used to store the root file system.

In the current configuration, the Linux root file system resides on a `ext4` partition on the SD-card, while the Linux kernel is stored in a `FAT32` partition on the SD-card. The SD-card provides a large amount of storage for program files and image files. The `ext4` file system is being used because it is the default file system in the *petalinux* build system.

The primary reason for choosing the default file system was to avoid complications. Features such as *journaling* and *metadata checksums* suggest that `ext4` is a good choice for reliable operation. The choice of file system could be further justified by comparing more file systems features.

An additional feature that may be desirable is increased redundancy in the form of ECC. The `ext4` file system does not offer the ability to store files with ECC, but the underlying memory technologies do. The eMMC and the NOR Flash chips have integrated ECC, providing some level of protection against bit flip errors.

The file system configuration is subject to change, as the eMMC on the PZ and integrated memories in the Zynq offer storage that is so far not utilised.

Redundant boot images on separate memory technologies will reduce the risk of critical failures. Boot image management and fall-back mechanisms have not been studied for this report, but is a point of interest that the HYPSO project will look into.

4.2 Software Defined Radio Payload

No work has been performed on the SDR system in this thesis. A short description of the system is still included because the system is intended to use the same FT system.

The SDR payload is comprised of a TOTEM module and an UHF antenna system. The TOTEM SDR system from *Alén Space* combines an UHF frontend with a SDR processing system [29]. It is acquired as a complete hardware unit.

The UHF frontend card can be tuned to operate between 70 MHz and 6 GHZ. The UHF radio consumes 160 mW when receiving, and up to 3 W when transmitting [29].

The processing system is equipped with a similar Xilinx Zynq-7000 SoC as the one used in the OPU. This SoC also runs Embedded Linux. The TOTEM uses CAN-bus to connect to the `CAN2` bus, and must also use the CSP network stack on top of the CAN protocol.

The sampled, digital radio signals are internally processed by *GnuRadio*, which is a SDR framework. It is implemented in C++, but can interpret Python scripts to create and configure flow graphs.

The FT system that is produced in this thesis can be used to upload these *GnuRadio* files, letting the SDR system be upgraded as new experiments are designed.

4.3 Communication Architecture

The HYPSO mission employs a communication architecture based on the CSP network stack. In this section, a high level model of the network is described. A top level representation of the network is illustrated in Figure 4.4.

The HYPSO communication architecture uses CSP for the space segment and ground segment in order to interface with NA components.

The space segment and the ground segment connect to the same conceptual CSP network. The subsystems inside the spacecraft are connected via various data interfaces, allowing any two of the satellite subsystems to communicate with each other. The ground segment components are primarily connected via various protocols running on top of IP/TCP. The space segment and the ground segment are connected via a radio link that is established by one of several ground stations.

A radio link can only be active while the satellite is within LOS of a ground station, meaning that the space segment and the ground segment are only periodically and temporarily connected.

During an active pass, the two networks connect and allow the whole network to be treated as a single uniform network. This distributed topology allows any node to communicate with any other node that is connected to the network, even when one node is in space and another is on the ground.

4.3.1 Space Segment

As mentioned, the M6P subsystems and HYPSO payloads are connected via a CSP network. As illustrated in Figure 4.4, the subsystems are physically connected via two separate CAN-buses. One CAN-bus (`CAN1`) connects all the original M6P subsystems: EPS, FC, PC and UHF. The second one (`CAN2`) connects the payloads to the PC. The S-Band radio is connected directly to the PC over a Serial Peripheral Interface (SPI) interface. The roles of the NA subsystems are described in Section 1.2, and more details can be found in Appendix A.

The partition of the space segment network into two CAN-buses is justified with the following reasoning. From NA's perspective, having a dedicated CAN-bus for the payloads allows them to isolate their subsystems from whatever system the customer connects to the satellite. Instead of having to connect the payload to multiple of their systems, they only have to customise and maintain the interfaces between the PC and the payloads. Additionally, it may prevent the original M6P subsystems and the payload subsystems from blocking each other with data traffic. For example, the imaging payloads will have large amounts of data that must be transferred to

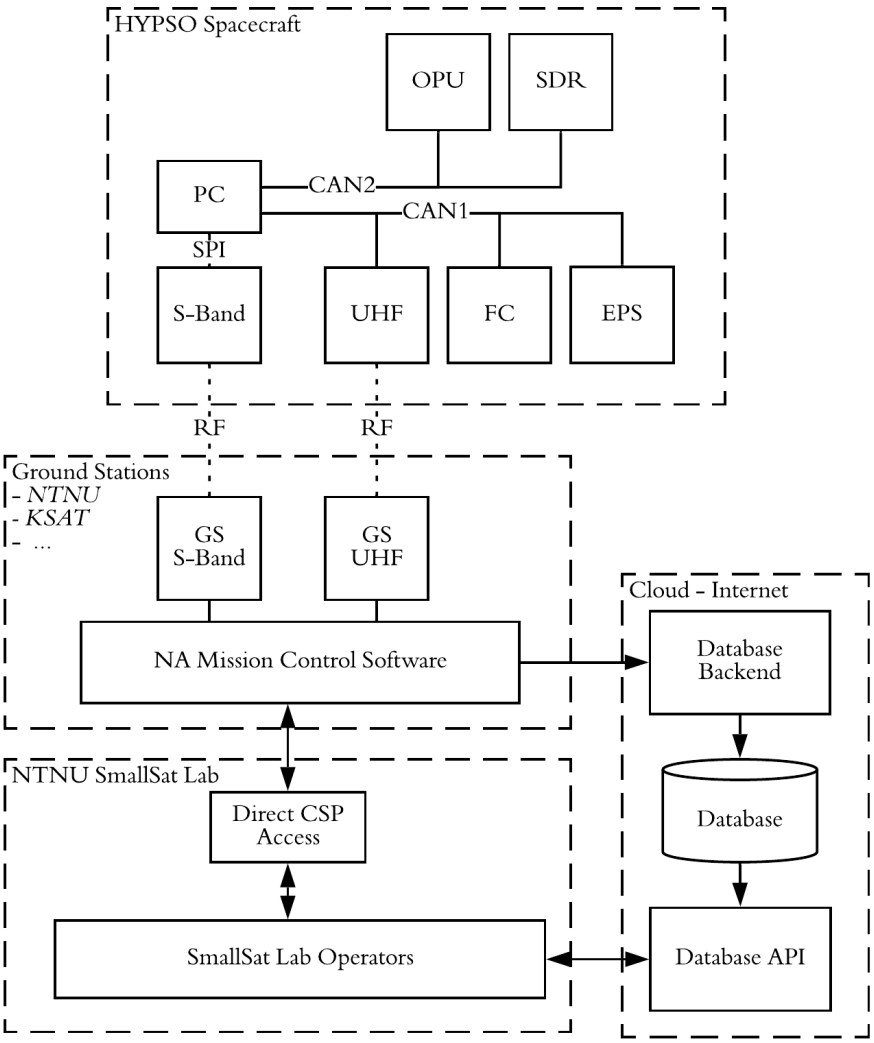


Figure 4.4: High level architecture of the communication network inside the satellite, and between the satellite and ground operators.

the PC, which thereafter is forwarded directly via the S-Band radio. If the two CAN-buses were merged, the transfer of the image data would cause congestion between the M6P subsystems. With this configuration, the payload may sustain a high data rate via S-Band, while other subsystems may still communicate over `CAN1` without obstruction.

4.3.2 Ground segment

The ground segment consists of one or more ground stations, a database, and one or more operators.

There may be several ground stations in the ground network. The operator must direct telecommands to the currently active ground station such that network traffic can be forwarded to the space network. NTNU SmallSat Lab intends to operate a ground station from NTNU. Additionally, there are plans to utilise ground stations from KSAT.

Each ground station must be equipped with antennas and transceivers for a UHF link and a S-Band link. NA provides Mission Control Software (MCS) which acts as a gateway by maintaining network connections to operators and a telemetry database. The precise details of this architecture are still not available at the time of writing.

A satellite operator may be a human operating a network connected computer, or a computer program running automatic procedures. Either way, an operator node must connect to the MCS gateway. The operator node can communicate CSP packets directly on the network through the gateway, allowing TM/TC to be exchanged. Since the operators can connect to the MCS over the internet, they do not need to be geographically present at the ground station.

A database is included in this network to store all Telemetry (TM) received from the satellite. All downlinked TM will be forwarded to the database, even when it was intended for an operator node. Consequently, all payload data that is downlinked will be stored in a database.

The database provides operational safety by backing up all downlinked data. Storing all data in a database allows the operator and end user to query historical data, and to replay stored records. By being stored in a cloud storage service, the data can be distributed to multiple end-users.

4.3.3 `nanoMCS` and Flatsat

NA provides a CLI application for the *Windows* operating system called `nanoMCS`. This program acts as a CSP node and can connect to the CSP network in several ways. It can connect directly to a CAN bus by using a CAN-to-USB adaptor. It can connect to MCS servers over a TCP connection using the NanoMsg-Next-Generation (NNG) protocol [1].

The `nanoMCS` software can also be used to connect to an engineering model of the M6P platform which is hosted by NA. This engineering model is called a *flatsat* and is used for early development of payloads. The intention of having a *flatsat* available for development is that software can be tested at a much earlier phase. The architecture when connected to the *flatsat* is illustrated in Figure 4.5.

[1] https://github.com/nanomsg/nng

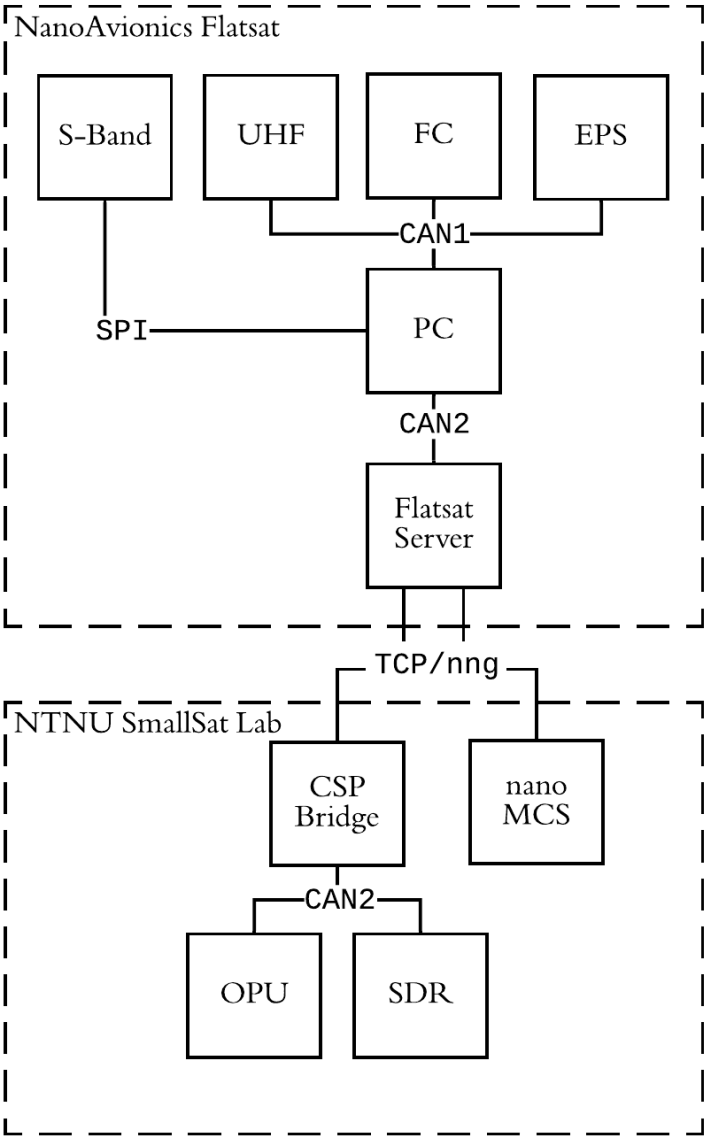


Figure 4.5: Architecture of the CSP network that connects the *flatsat* engineering model to payloads and `nanoMCS` at SmallSat Lab.

Chapter 5

File Transfer System

This chapter details the design and implementation of a file transfer system for the HYPSO payloads.

The file transfer system that has been designed and implemented for this thesis is referred to as the *FT system*. It should not be confused with the M6P file transfer system, which is installed on the M6P subsystems and provided as a specification.

M6P Foundation

The FT system which is detailed in this chapter is designed by the author and the implementation is entirely produced by the author.

However, the M6P file transfer system is used as an inspiration and foundation for designing the FT system. This is done for a number of reasons:

- Using an existing specification as a basis rather than creating a new one saves development time.
- By being compliant with the NA file transfer specification, the same file transfer utilities can be used for the M6P subsystems.
- The M6P file transfer system can be used to verify the FT system that is made for the HYPSO payloads to some degree.
- The NA file transfer specification has flight heritage.

Consequently, the FT system borrows from concepts described in Section 2.6.3 and implements the procedures described in Section 2.7.5.

Although the FT system is separate from the M6P file transfer system, the two are compatible to a certain degree. The FT system is able to interface the M6P file transfer services, but does also provide some features that are not supported by the M6P file transfer system.

5.1 Service and Client Architecture

The *request-response* pattern is used to implement the FT system, as is typical for CSP applications. A service receives requests from clients, performs the requested tasks, and returns a reply with data or a status indication. A File Transfer Service (FTS) is intended to run on each payload, while a File Transfer Client (FTC) is integrated on the ground operation nodes.

The term FTS is used to refer to the service that continually runs on then payload, while the term FTC is used to refer to an application that requests file operations and file transfers.

The architecture depends on the CSP network described in Section 4.3. The underlying network layers are transparent to the FT system, so the architecture can be simplified to the architecture shown in Figure 5.1.

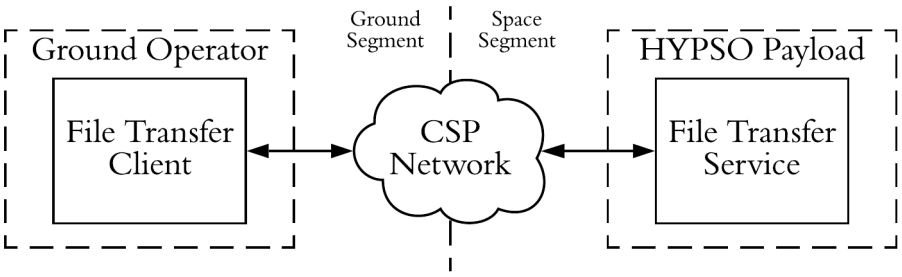


Figure 5.1: Service and client architecture. The underlying CSP network is transparent to the service and client.

The FT system is implemented such that the FTS stores as few states as possible. This is done to limit the complexity of the code that goes into the payload, in an effort to minimise bugs that are difficult to debug and correct. It is achieved by having the transfer state being stored in formatted files, and by making the FTC perform as much decision making as possible.

5.2 File Organisation

File management on the HYPSO payloads include the formats in which files are transferred, and the organisation of different types of files.

5.2.1 File Format

During transmission, a file is fragmented into segments that are reassembled at the receiving node. Segments that arrive *in-order* can be reconstructed into the original file by concatenating one segment's data to the previous. If the segments do not arrive *in-order*, then the segments must be stored until enough information has arrived to use them in the reconstruction.

The approach taken here adds a sequence number to each segment. The sequence number is used to place the segment at the correct offset in a destination file. A custom file format is used to store the incoming segments during a transfer.

Scope

A *formatted file* is only intended to be used by the FT system. Other modules will not and can not use the formatting metadata. A file should only be transformed into a formatted file when it is to be transferred over the FT system.

The metadata that is added to create a formatted file encodes the information that is necessary for keeping track of a transfer.

The M6P file system defines the *entry* structure in Listing 1 as a file segment. The entry data, CRC and length fields combined are defined as a *cell*. These *cells* are used as segments when sending file data in the M6P file transfer system, and this is adopted for the FT system.

The CRC field and length field in the cell indicate whether an entry is present and valid.

Layout

Regardless of which layout is used, the formatted files can be created by modifying the original file *in-place*, or by creating a new file with copied data.

Two memory layouts, (1) and (2), are considered for the formatted file.

The first layout has entry metadata interleaved with the entry data (1). It is then easy to extract an entry with data and metadata, as the memory region of the cell can be copied directly into a packed C structure.

When interleaving the metadata (1), the data offsets in the original file do not match the data offset in the formatted file. Every entry is shifted some amount to the right. A consequence of this is that if the formatted file (1) is created *in-place*, a large amount of copy operations is required to perform all the shifting. This must also be done if converting from a formatted file back to the original file *in-place*.

The alternative is to combine all of the metadata and append it at the end of the original file (2). The file can be converted back by truncating the end of the file. The layout with appended metadata (2) is therefore not required to shift the entry data.

A drawback of modifying the original file *in-place* is that the original file can be corrupted if an unexpected interruption occurs while it is being formatted. To prevent this, the contents of the original file should be copied to a new, separate formatted file. This requires more storage space, as two copies of the original data must be stored, but it is safer.

See Figure 5.2 for a visual comparison of the two.

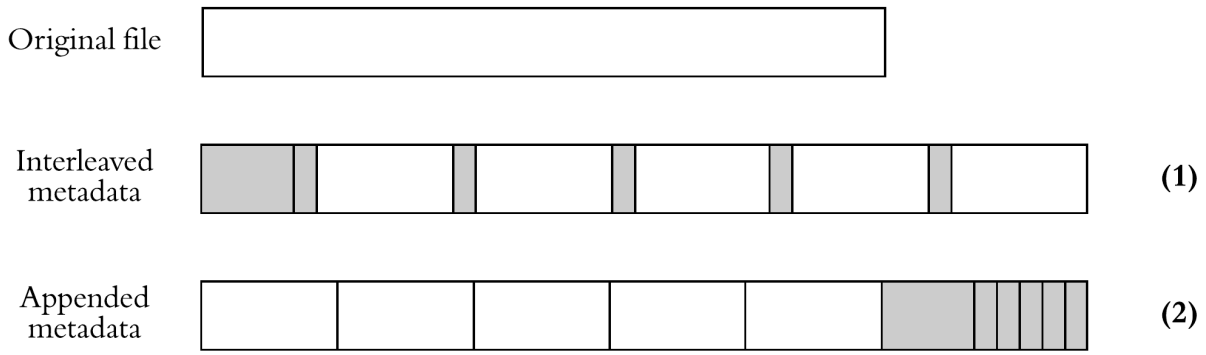


Figure 5.2: Comparison of layout and footprint of interleaved metadata and appended metadata. The first grey field represents a file header, while the thin grey fields represent metadata for each successive entry.

The formatted file with interleaved metadata (1) is chosen because it provides a simpler implementation of reading and writing entries when metadata and corresponding data is chunked together into a struct. The data is copied to a new file to avoid corrupting the original file, meaning that the appended format (2) would also need to copy each of the entries. Shifting the data at the same time as copying it (1), does therefore not cause a performance loss when compared with the alternative (2). The chosen layout (1) is shown in Figure 5.3.

For the remainder of this report, the term *formatted file* will mean a formatted file with *interleaved* metadata (1).

5.2.2 File System Module

The C module `fs` has been implemented to provide functions for handling formatted files. The implemented functions are shown in Figure 5.4.

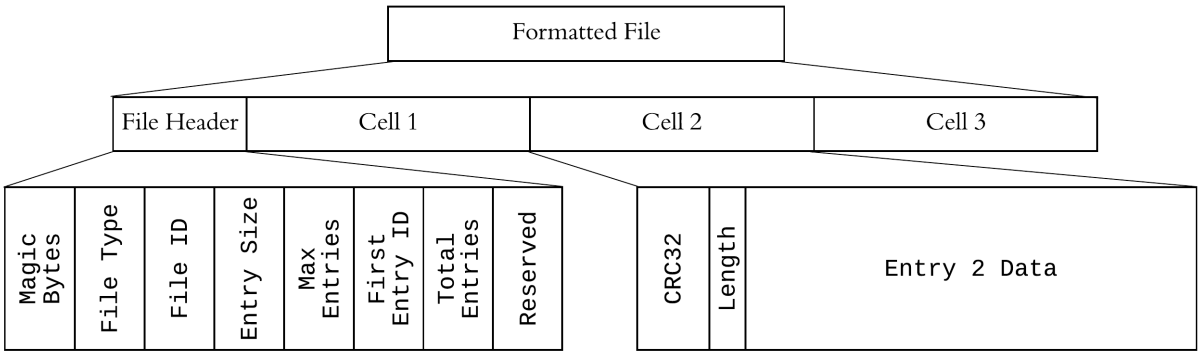


Figure 5.3: Memory layout of a formatted file with interleaved metadata (1). A file header stores information about the size and number of entries.

- The function `fs_format_file()` handles creation of a formatted file.
- The functions `fs_get_file_header()` and `fs_write_file_header()` retrieves and writes the header section of a formatted file.
- The functions `fs_write_cell()`, `fs_write_entry()` and `fs_clear_file()` handle writing and clearing of data in formatted files.
- Entries and cells are read with `fs_read_cell()` and `fs_read_entry()`.
- Transformations between original file and formatted file is handled by `fs_prepare_file()` and `fs_extract_file()`.
- The presence and integrity of entries is checked with `fs_check_cell()`, `fs_check_entry()`, `fs_check_file()` and `fs_print_bitmap()`.

The modules use the `fstream` class of functions to modify the formatted files. If the layout of the formatted file is ever changed this module can be reimplemented without significant changes to its interface.

5.2.3 File Mapping Module

In order to provide a compatible interface for the file download and file upload procedures that are defined in Section 5.3.5 and Section 5.3.6, a *file mapping* structure is defined for the FT system. The structure is illustrated in Figure 5.5.

The file ID map is stored in a configuration file, such that the mappings persist across reboots and unexpected shutdowns. The methods in the implemented module are shown in Figure 5.6.

- The functions `fs_idmap_init()`, `fs_idmap_close()` and `fs_idmap_format()` are used to load, close or create a new mapping file.
- The mappings can be modified with `fs_idmap_add_file()` and `fs_idmap_remove_file()`.
- File path can be queried from file ID with `fs_idmap_get_file_path()`.
- Helper functions `fs_idmap_open_r()` and `fs_idmap_open_rw()` are provided for opening a `fstream` to a mapped file by providing a file ID.

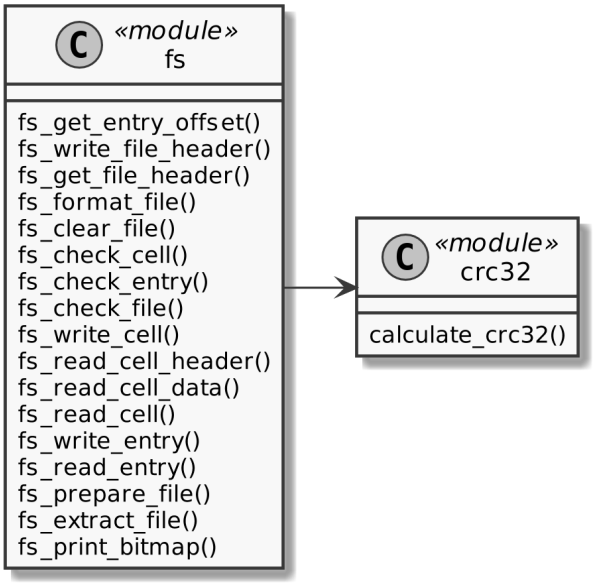


Figure 5.4: The `fs` module provides functions for creating, reading and writing formatted files.

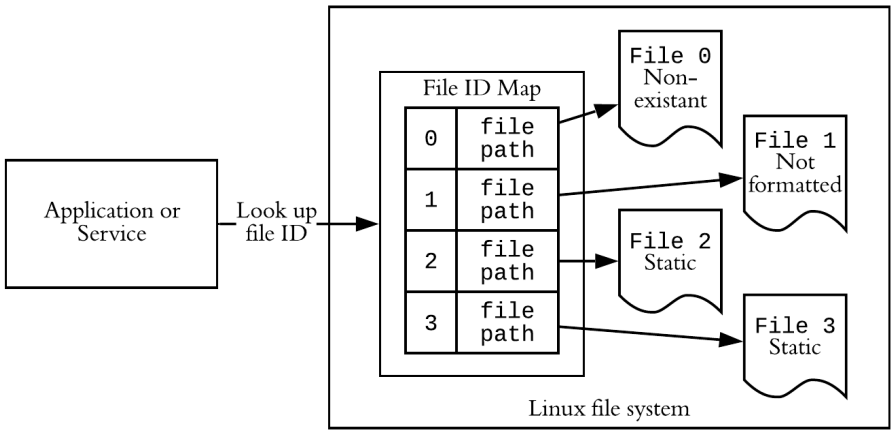


Figure 5.5: A file ID map that contains mappings from *file ID* to *file path*.

5.2.4 Proposed Directory Hierarchy

A file path has to be registered to a file ID before it can be used for transfers. File paths should be kept short to avoid long registration and deregistering command packets.

The example directory tree in Figure 5.7 is proposed as a baseline for the OPU payload. It contains various mission data, as well as a directory for upgrade files.

As mentioned in the *Flight Results from the Aalto-1 Mission* [30], it is recommended to separate the file system that contains mission data from the file system that contains Linux system files. The Linux system files should be read only, so as not to accidentally overwrite them while modifying mission data.

5.3 File Transfer

This section describes file transfers in the FT system. The following topics are covered in order:

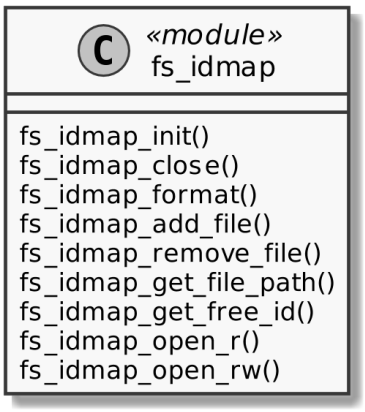


Figure 5.6: The `fs_idmap` module provides functions to get file path from file ID, and to register new file ID mappings.

```
/
└── home
    └── hypso
        ├── scripts ..... Often used script files
        │   ├── cap-hsi-buf ..... Capture HSI image and buffer on PC
        │   └── cap-rgb ..... Capture and store RGB image
        ├── hsi ..... Captured HSI images
        │   ├── 2019-05-25.12:30:45.dat ..... Example HSI file
        │   └── 2019-05-25.12:40:00.dat
        ├── rgb ..... Captured RGB images
        │   ├── 2019-05-25.12:30:45.png ..... Example RGB file
        │   └── 2019-05-25.12:40:00.png
        ├── log
        │   ├── fts ..... Debug and error output from File Transfer Service
        │   ├── hsi ..... Debug and error output from HSI Service
        │   └── rgb ..... Debug and error output from RGB Service
        └── upgrade ..... File patches and boot images
            ├── BOOT.BIN ..... Example upgrade file
            └── hypso-service.patch
```

Figure 5.7: Proposed directory tree for organising files on the OPU payload.

- The concept of a file stream (Section 5.3.1).
- The ARQ strategy being used (Section 5.3.2).
- Implementation details of modules (Section 5.3.3).
- Details of the transfer procedures (Sections 5.3.5 to 5.3.7).

5.3.1 File Stream

A *file stream* is a sequence of CSP packets that make up data in a file transfer. These *stream packets* contain entries from a formatted file.

When transferring a file, the original data is first fragmented into entries. The entries are prepended with a CRC checksum and length value to produce a cell, and then packed into a *formatted file*. These entries are inserted into stream packets, as shown in Figure 5.8.

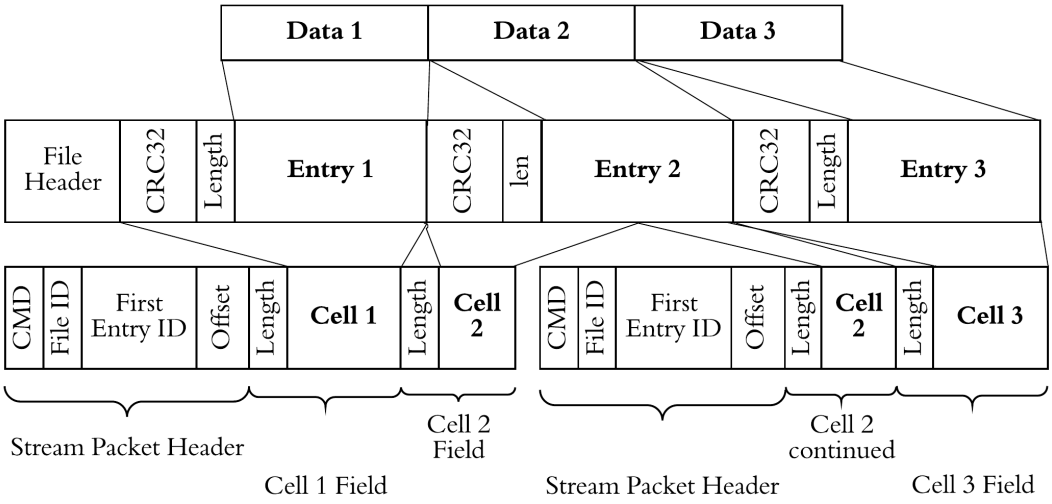


Figure 5.8: A file stream. The data of an original file is fragmented into the entries of a formatted file. The cells of the formatted file are inserted into stream packets.

In the NA specification, a single *stream packet* may contain multiple (and even fragmented) entries when downloading a file, but only a single entry when uploading a file. In order to be compliant, the HYPSO implementation is able to receive stream packets containing multiple (and even fragmented) entries. However, it is advised to keep the entry lengths equal to the stream packet lengths, as to not make the fragmentation of the original file redundant.

The first entry ID field in the stream packet header serves as sequence number. Together with the *first entry ID* field in the formatted file header, the sequence number can be used to determine where in the destination file the cell should be inserted. The inclusion of a sequence number allows entries to arrive out-of-order.

The offset field in the stream packet header is used in the case that the first entry in the packet contains a fragmented cell which must be placed at an offset.

5.3.2 Automatic Repeat Query

The mission requirements state that file transfer must be possible (**MS-0-011**, **MS-0-12**, **MS-0-013**) over the communication network provided by M6P (**IF-001**).

The CSP network is unreliable. CSP packets can be lost for a number of reasons:

- Communication errors in the lower layer protocols (CAN, SPI, UHF and S-Band radio).
- Congestion on links that are used by multiple users. The SDR and OPU payloads could try to communicate at the same time.
- Buffer exhaustion, which could happen because of congestion, or because of an excessive sending rate.
- Link disruptions, such as the sudden loss of ADCS pointing accuracy, causing the S-band radio connection to drop out.

Retransmission procedures are therefore implemented by the FT system. The ARQ strategy for the FT system is inspired by the M6P file transfer system.

Control Message Loss

The loss of control messages, such as requests to transmit a file, is handled by timeouts that issue retransmission of requests.

File Segment Loss

The NA documentation suggests that ARQ can be achieved by alternating between two phases. Figure 5.9 illustrates the concept. In one phase entries are being transferred in unacknowledged mode, while in the other phase the completeness of the file is checked. If packets were lost, it is detected, and the missing data can be retransmitted by requesting a new file stream for a range of entries.

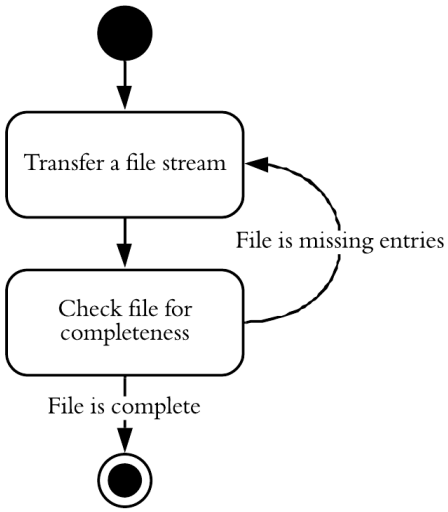


Figure 5.9: ARQ strategy of the FT system, with alternating phases of transfering data and checking file for missing entries.

This is the general ARQ strategy that is being used in the FT system.

The download and upload procedures automatically request the sending node to repeat the entries of the file that are still missing. The download and upload procedures are similar, but not identical since the FTC and FTS are implemented differently. The FTC will perform most ARQ related tasks in order to keep the FTS implementation as simple as possible.

5.3.3 File Transfer Modules

The file system module from Section 5.2.2 is the basis for the FT modules. In order to limit the scope of each module, separate modules for download and upload procedures have been implemented.

Client

Client modules are shown in Figure 5.10.

The FTC modules implement procedures for all of the original M6P file requests, which are also used in the FT system:

- `ft_client_info()` requests metadata for a specified file ID.

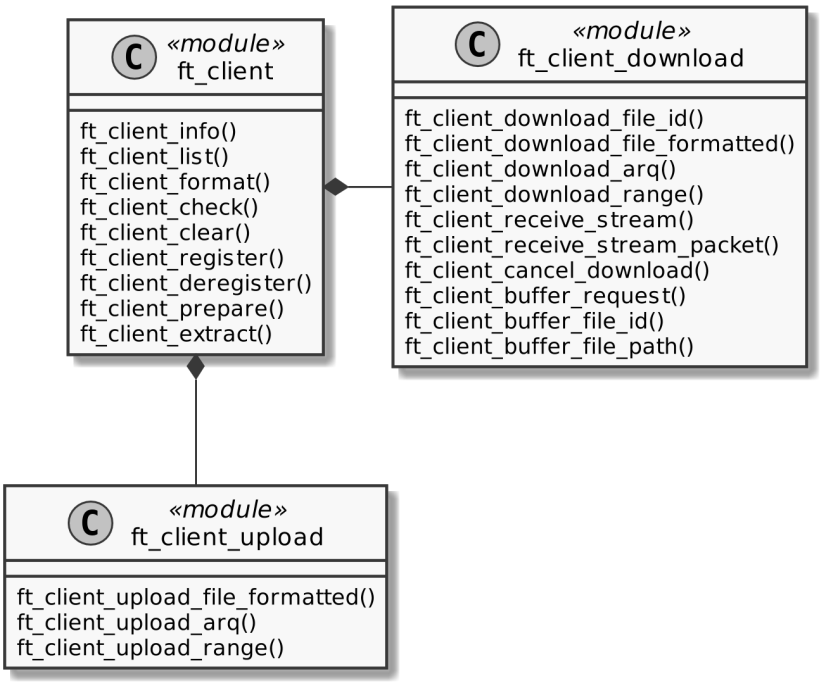


Figure 5.10: C modules for the FTC.

- `ft_client_list()` requests metadata for a range of file IDs.
- `ft_client_format()` requests a file to be reformatted to a new or different number or size of entries.
- `ft_client_check()` requests a bitmap encoded with the integrity of the file.
- `ft_client_clear()` requests a file to have its entries cleared of data.
- `ft_client_cancel_download()` requests an ongoing file transfer to be aborted.

File ID mappings are handled by a dedicated pair of requests:

- `ft_client_register()` requests a mapping from a specified file ID to a specified file path.
- `ft_client_deregister()` requests the removal of an existing file ID mapping, if it exists.

Transformations between original and formatted files are handled by a pair of requests:

- `ft_client_prepare()` requests the creation of a formatted file from an original file.
- `ft_client_extract()` requests the restoration of an original file from a formatted file.

The following upload and download client procedures are detailed in Sections 5.3.5 to 5.3.7:

- `ft_client_upload_file_formatted()`
- `ft_client_upload_arq()`
- `ft_client_upload_range()`
- `ft_client_download_file_id()`

- `ft_client_download_file_formatted()`
- `ft_client_download_arq()`
- `ft_client_download_range()`
- `ft_client_receive_stream()`
- `ft_client_receive_stream_packet()`
- `ft_client_buffer_request()`

Service

Service modules are shown in Figure 5.11.

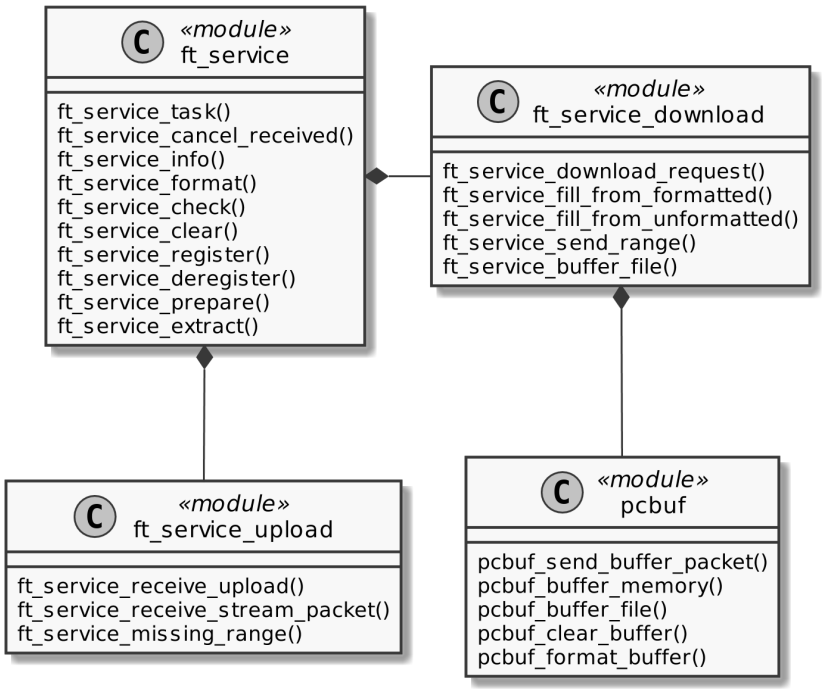


Figure 5.11: C modules for the FTS.

On the top level, the FTS is implemented as the `pthread` task `ft_service_task()`. It listens to a socket that accepts CSP connections, and responds to the requests received from FTCs.

All the original M6P requests are answered:

- `ft_service_info()` replies with metadata of a specified file, if it exists.
- `ft_service_format()` attempts to reformat a specified file and replies with a result code.
- `ft_service_check()` checks a specified range of entries in a file, and replies with a bitmap that encodes the result of the checking.
- `ft_service_clear()` clears every entry in a specified file, and replies with a result code.
- `ft_service_cancel_received()` replies with an error code when there is no ongoing transfer. Cancel requests are handled internally in the download procedure.

File ID mappings requests are handled:

- `ft_service_register()` performs a mapping of a specified file ID to file path and replies with a result code.
- `ft_service_deregister()` removes an existing file ID mapping, if it exists, and replies with a result code.

Transformations between original and formatted files are handled:

- `ft_service_prepare()` creates a formatted file from an original file.
- `ft_service_extract()` restores an original file from a formatted file.

The following upload and download service procedures are detailed in Sections 5.3.5 to 5.3.7:

- `ft_service_receive_upload()`
- `ft_service_receive_stream_packet()`
- `ft_service_missing_range()`
- `ft_service_download_request()`
- `ft_service_send_range()`
- `ft_service_fill_from_formatted()`
- `ft_service_fill_from_unformatted()`
- `ft_service_buffer_file()`
- `pcbuf_buffer_file()`

5.3.4 Transfer Modes

The following sections go into detail on the procedures that are used in the different transfer modes.

- Direct download mode (Section 5.3.5).
- Direct upload mode (Section 5.3.6).
- Buffered download mode (Section 5.3.7).

5.3.5 Direct Download

The data path of a direct download is shown in Figure 5.12. The stream packets are addressed directly to the FTC in the operator node. The file data is sent from the FTS and routed through the PC before being transferred over the S-Band link. The MCS distributes the packets to the operator node as well as backing up the traffic in the database.

The FTS is referred to as the source, and the FTC is the destination. The FTS holds the source file, and the FTC holds the destination file.

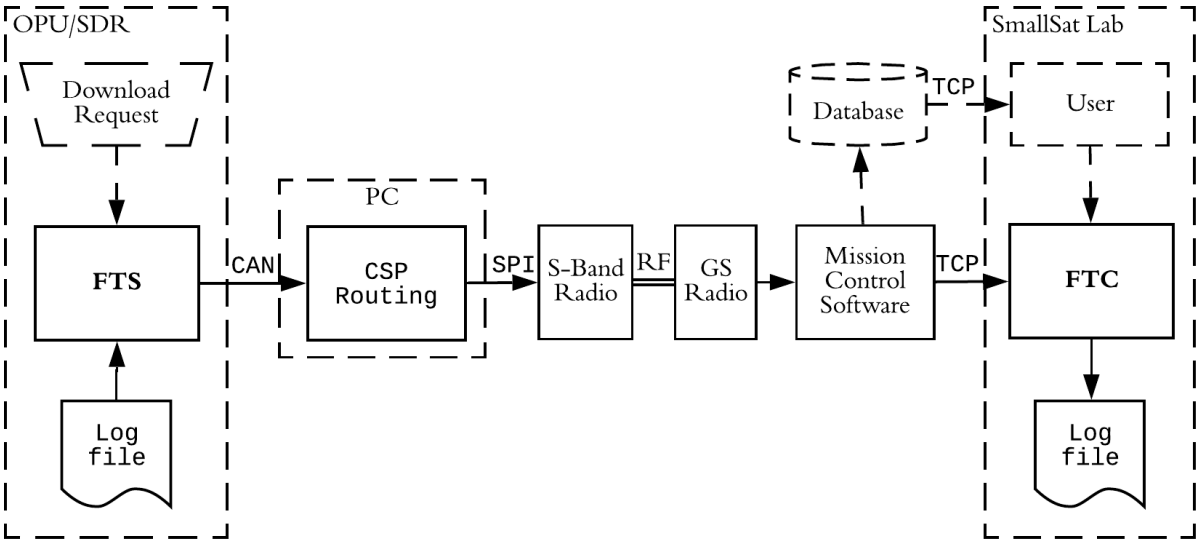


Figure 5.12: Data path of the direct download transfer mode. A file download command initates a file transfer from the FTS. The data is routed through the PC, S-Band radio and ground station before reaching the FTC.

Service

The FTS replies to download requests with file streams.

A few procedures are used together to answer each request:

- `ft_service_download_request()`, which validates the download request.
- `ft_service_send_range()`, which sends the file stream.
- `ft_service_fill_from_formatted()`, which creates stream packets.
- `ft_service_fill_from_unformatted()`, which creates stream packets.

There is only a single type of download request, and it is answered with the `download request` procedure. This procedure is illustrated in Figure 5.13.

Each download request must specify the file ID of the file that should be downloaded. The FTS queries the `fs_idmap` module to check whether the file exists. An error is returned if it does not.

The request must also include the start and end of the range of entries that should be downloaded from the file. The FTS checks whether the specified range exists in the file, and replies with an error code if it does not.

Once validated, the requested range of entries will be passed to the `send range` procedure, which is illustrated in Figure 5.14.

The `send range` procedure will first fetch a CSP packet buffer. If no buffers are available, it will sleep for a duration of time and try again. Once a packet buffer has been acquired, it will fill it with entry data by using different procedures depending on whether the source file is formatted or not.

The `fill from formatted` procedure is used to create stream packets from formatted source files. It uses the `fs` module to fetch cells, and inserts them into the stream packets.

The `fill from unformatted` procedure is used to create stream packets from source files that are not formatted. This function copies data directly into the stream packets, without

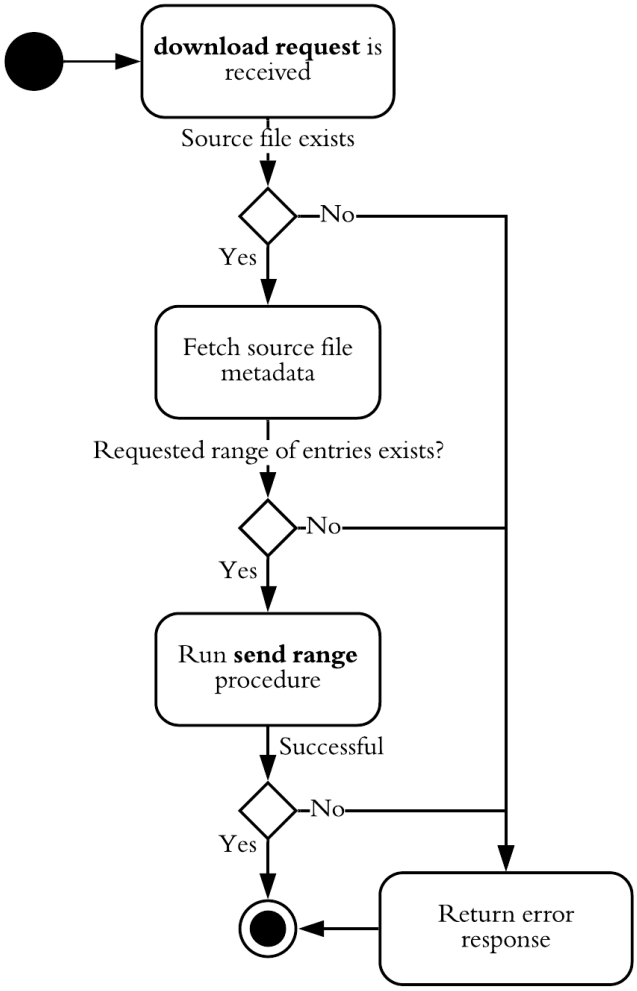


Figure 5.13: The `download request` service procedure. It validates the request before sending a range of entires.

creating a formatted file. The procedure requires an extra parameter to determine the entry size since the source file does not have a file header. The entry size is used to calculate the data offsets for entries in the unformatted file. The procedure uses the `pkt_sz` parameter of the download request to calculate the entry size. When inserting data into the stream packets, the procedure simultaneously inserts a calculated CRC code and length field.

The `send range` procedure terminates once it has sent the last entry in the specified range.

Client

The FTC conducts the download by sending download requests and receiving the resulting file streams.

Several nested procedures are called to perform the transfer:

- `ft_client_download_file_id()`, which creates a formatted destination file.
- `ft_client_download_file_formatted()`, which validates the formatted destination file.
- `ft_client_download_arq()`, which performs retransmission.

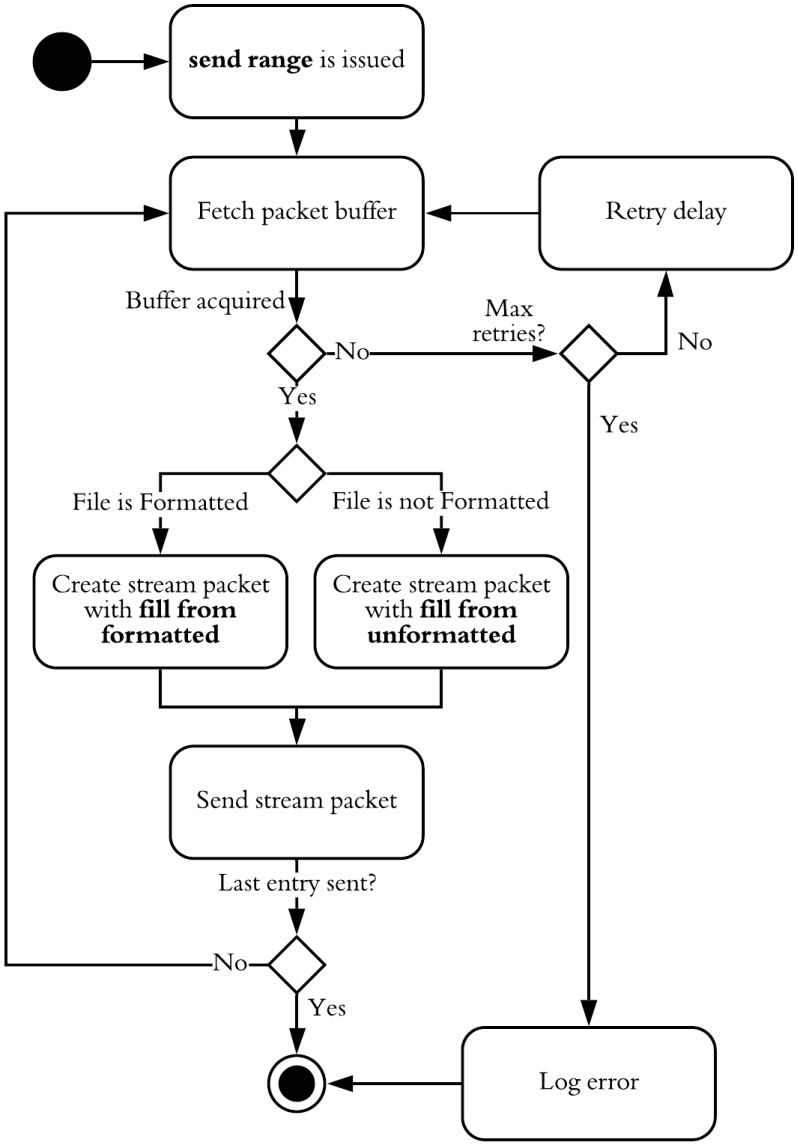


Figure 5.14: The `send range` service procedure. It creates stream packets from the specified range of entries. It uses one of two different procedures to fill the data field of the stream packet, depending on whether the file is formatted or not.

- `ft_client_download_range()`, which downloads individual ranges.
- `ft_client_receive_stream()`, which handles an incoming file stream.
- `ft_client_receive_stream_packet()`, which handles individual stream packets.

The top level `download file id` procedure, illustrated in Figure 5.15, will first request information about a specified source file at the source node. The requested information is used to create a formatted destination file that the source file can be downloaded into. If the destination file already exists, and has the correct format, it will be used as it is. This way, a transfer that was interrupted can be continued by downloading to the incomplete formatted file. If the destination file exists, but is not correctly formatted, the user is given a choice to reformat the file.

The source file can be formatted or not formatted. If the source file is formatted, an identical formatting is performed on the destination file. If the source file is not formatted, a formatted destination file of type `STATIC` is created, and the entry size is decided by the FTC. The entry size is then communicated via the `pkt_sz` parameter in the subsequent download requests.

When a formatted destination file has been created or validated, the `download file formatted` procedure is called to perform the next stage in the transfer.

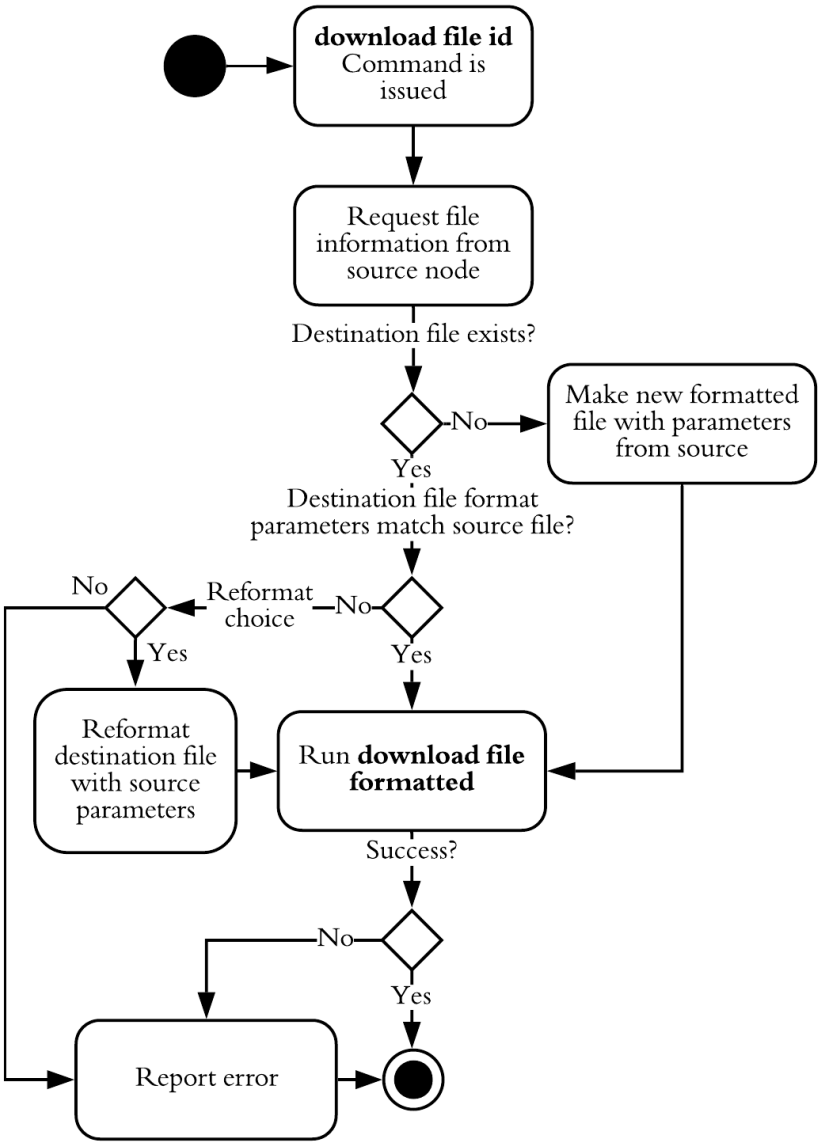


Figure 5.15: The `download file id` procedure prepares a formatted destination file before requesting a download.

The `download file formatted` procedure is illustrated in Figure 5.16. The procedure downloads a formatted file from a FTS, and assumes that a corresponding formatted destination file exists at the FTC node.

First, the FTC requests information about the formatted source file. If the formatted destination file matches the format parameters (file ID, entry size, entry number), then the `download arq` procedure is run. Otherwise, the formatted destination file cannot be used to download that formatted source file, and an error is reported.

The `download file formatted` procedure is similar to the `download file id` procedure, but will not create or reformat a formatted destination file, it will only verify that one exists. The function is included as a separate procedure to be integrated into automatic procedures that do not ask the user for input in the way that `download file id` does.

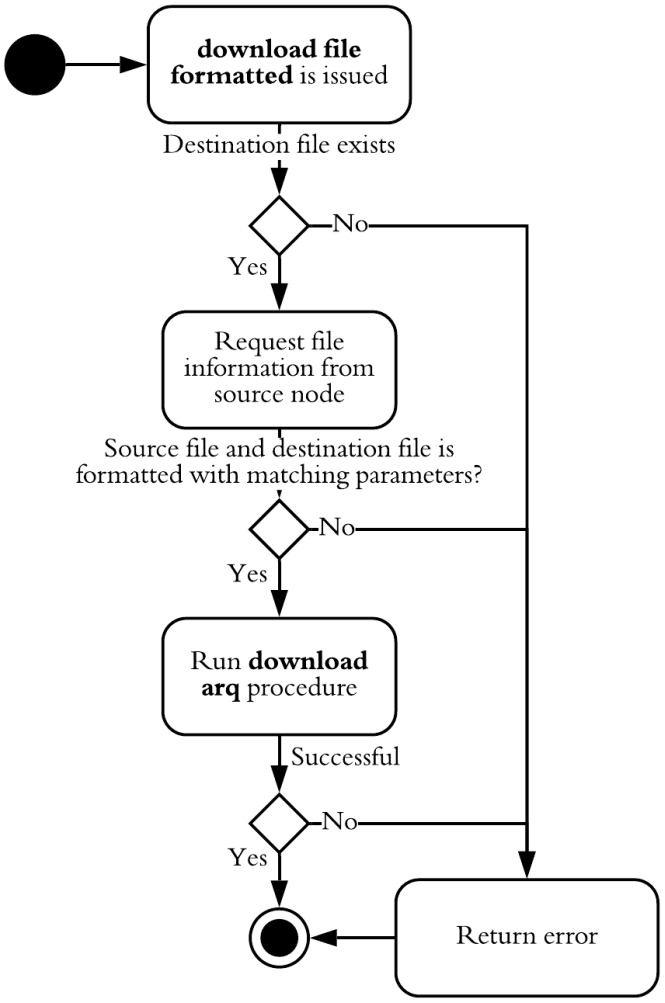


Figure 5.16: The `download file formatted` procedure verifies that a valid formatted destination file exists before starting the actual download procedure.

The `download arq` procedure, as illustrated in Figure 5.17, is responsible for performing the reliable transfer. It will find the first occurrence of a range of missing entries in the formatted destination file, and send a download request for that range with the `download range` procedure. For a new transfer, this will first be a request to download the whole file. After one iteration, any entries that have been lost or corrupted in transmission will be detected as the procedure again seeks out the first occurrence of a range of missing entries. This repeats until all entries are transferred successfully.

The `download range` procedure, as illustrated in Figure 5.18, sends a download request to a FTS and waits for a response acknowledging the request. If a successful ACK is received, the connection is passed to the `receive stream` procedure.

There are two scenarios that require extra attention. The request packet and the response packet that acknowledges the request may both get lost in transmission. In the first case, a response will not get sent, so the client will time out and resend the download request. This

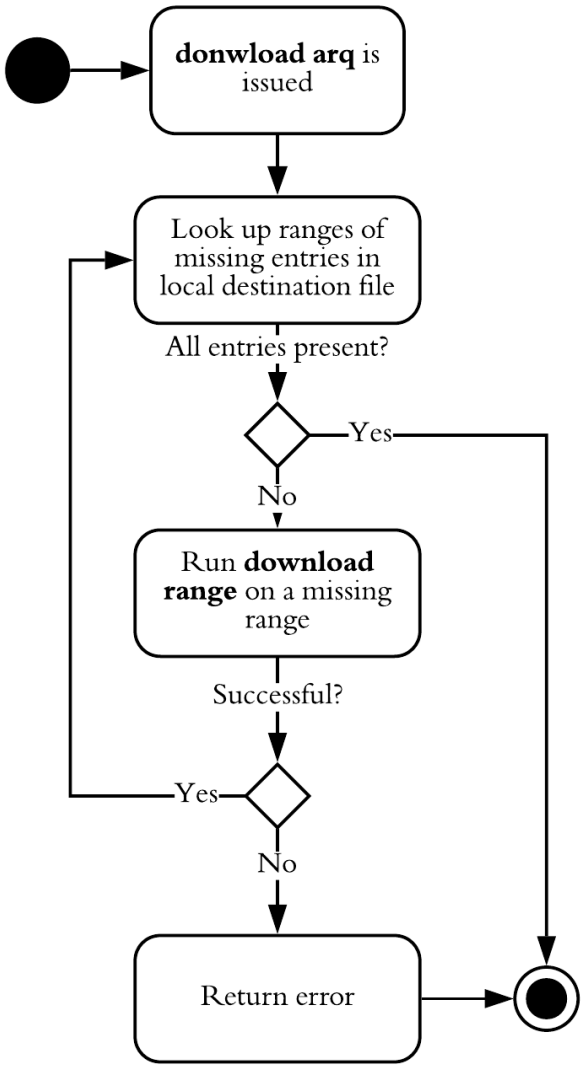


Figure 5.17: The `download arq` procedure requests transmission of missing entries until a complete file has been downloaded.

repeats until a maximum number of retries have been exhausted. In the second case, the ACK response is lost, such that the client receives one of the subsequent stream packets instead. The client must therefore check whether it has instead received a stream packet for the file range it just requested. If so, it can assume that the ACK response was lost, and will carry on receiving the stream as normal.

The `receive stream` procedure, as illustrated in Figure 5.19, waits for the arrival of stream packets and stores their entries into a formatted destination file. If the procedure times out while waiting for a stream packet, or if the final stream packet in the requested range arrives, it will terminate successfully.

Together, the described download procedures are able to reliably transfer a formatted file from the FTS to the FTC.

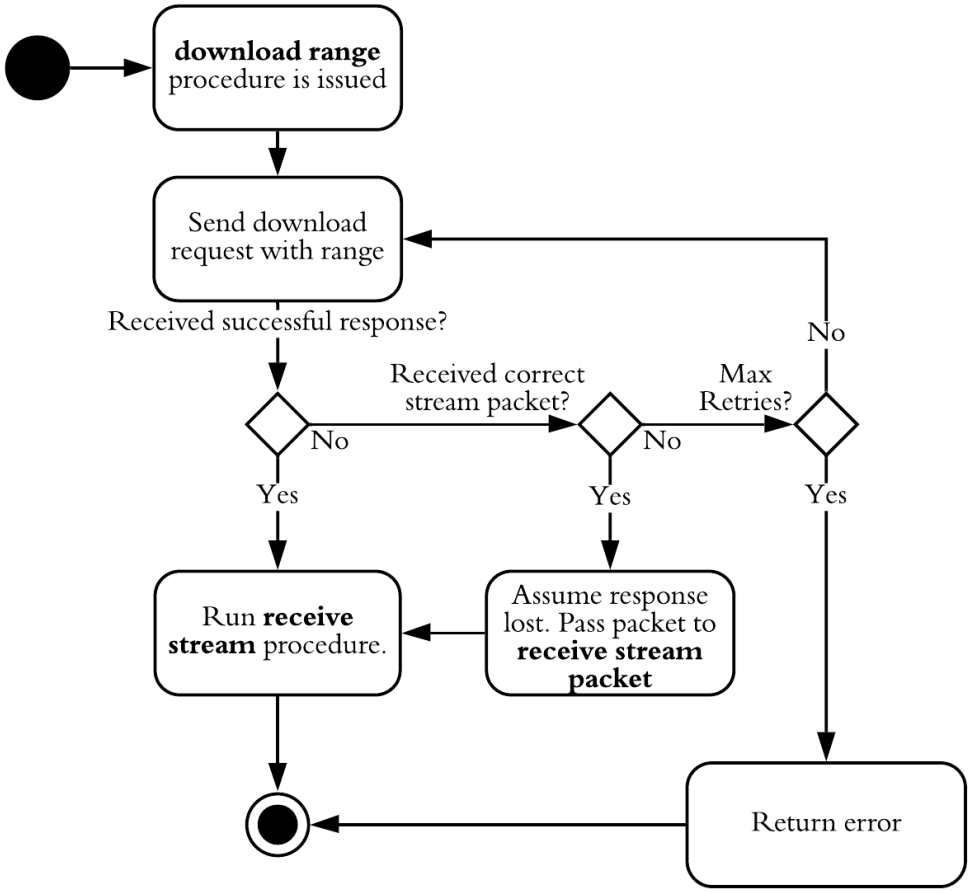


Figure 5.18: The `download range` procedure, which downloads a specified range of a formatted file.

5.3.6 Direct Upload

The data path of a direct upload is shown in Figure 5.20. The stream packets are addressed directly to the FTS in the payload. The file data is sent from the FTC via a ground station that routes the packets to the satellite over a connected radio link. The PC receives the packets from the S-Band radio, and routes them to the FTS on the payload.

The upload procedures are similar to the download procedures. The FTC is now referred to as the source, and the FTS is the destination. The FTC holds the source file, and the FTS holds the destination file.

Service

The following procedures are used for receiving uploads:

- `ft_service_receive_upload()`, which validates stream packets.
- `ft_service_receive_stream_packet()`, which stores the contents of stream packets.
- `ft_service_missing_range()`, which responds to *missing range* requests.

When the FTS receives a stream packet it is interpreted as part of an upload procedure. The FTS is always ready to accept file entries, but it will only store them if there is a valid, formatted

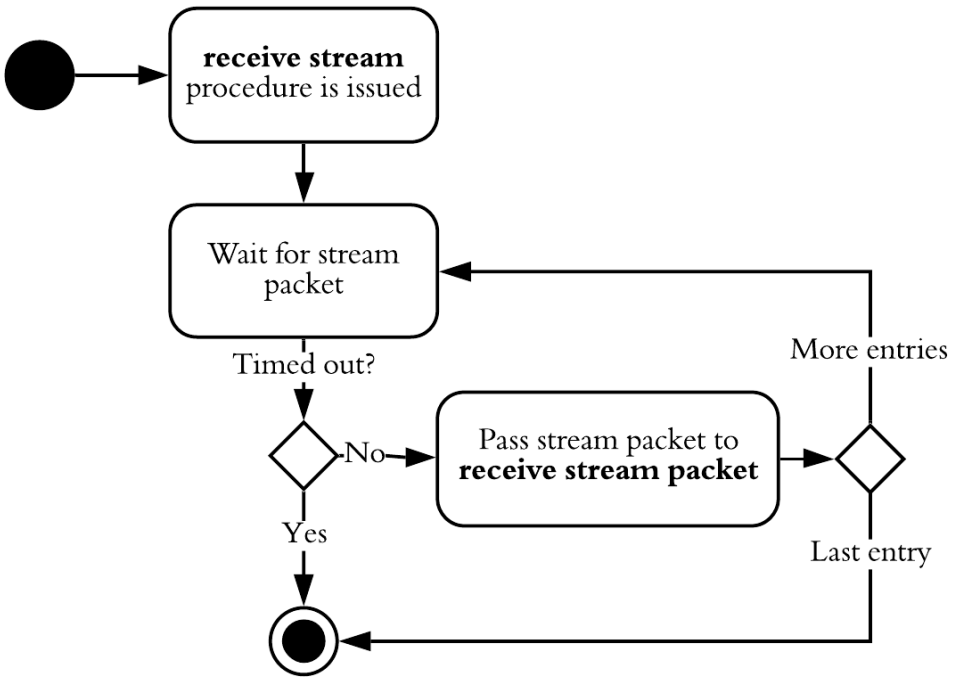


Figure 5.19: The `receive stream` procedure, which stores the entry data from incoming stream packets into a formatted destination file.

file prepared to accept them. Therefore, before uploading a file, a formatted file must be created at the FTS.

File streams are directed to the `ft_service_upload()` procedure, which examines the stream packets to determine whether a formatted destination file exists.

After verifying that a formatted destination file exists, the stream packets are passed to the `ft_service_receive_stream_packet()` procedure. This procedure examines the formatted destination file to check whether it already contains the received entries. Missing entries are inserted into the formatted destination file.

The `ft_service_missing_range()` procedure responds to the *missing range* requests, which is a part of the upload ARQ strategy. The response returns a list of ranges that indicate which entries of a formatted destination file are still missing. The list is used by the FTC to determine which ranges of entries it should upload.

Client

The FTC also conducts the upload procedures.

The following nested procedures are used to upload a file:

- `ft_client_upload_file_formatted()`, which verifies that a formatted destination file exists.
- `ft_client_upload_arq()`, which performs retransmission of missing entries.
- `ft_client_upload_range()`, which uploads individual ranges of entries.

The FTC is responsible for making sure that there is a valid formatted destination file at the FTS before attempting to send file entries.

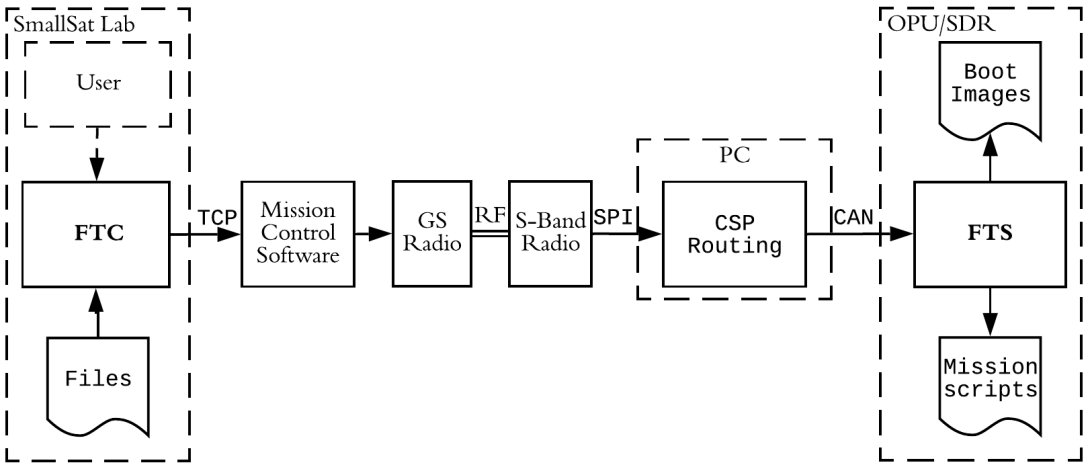


Figure 5.20: Data path of the direct upload transfer mode. The user transmits file data to the payload. The data packets are routed through the MCS and S-Band radio. The PC performs the final routing before the data is received at the payload and stored into a formatted file.

The top level procedure `upload file formatted`, illustrated in Figure 5.21, will attempt to upload a formatted source file. It will first request information from the FTS to check whether a valid, formatted destination file exists. If a matching formatted destination file (file ID, entry size, entry number) exists, the `upload arq` procedure is called, otherwise an error is reported.

The `upload arq` procedure, illustrated in Figure 5.22, will attempt to reliably upload a formatted source file. It will alternate between requesting ranges of missing entries from the FTS, and issuing the `upload range` procedure on the returned ranges. Initially, the request for missing ranges will yield a single range indicating that the whole file is missing. If packets are lost or corrupted in transmission, several requests for missing ranges must be sent. The `upload arq` procedure terminates once the response to missing ranges yields an empty list, or if an error occurs.

The `upload range` procedure, illustrated in Figure 5.23, will send a specified range of entries as a file stream. It will pack exactly one entry in each stream packet, without CRC or length fields. After sending the last entry, it will successfully terminate.

5.3.7 Buffering

Similar to CFDP's *store-and-forward* capabilities, the PC can be used to store a file before forwarding it. The datapath for this operation is shown in Figure 5.24.

Instead of sending a data stream directly to the intended recipient, the payload sends the file contents to a CSP service on the PC. The PC stores the data in a buffer file, which can then be transferred using the file transfer facilities provided by NA, or with the FT system that is described in this Chapter.

The FTS can be requested to act as a buffering client to the buffering service on the PC. The following procedures are used for buffering files:

- `ft_client_buffer_request()`, which sends the buffering requests.
- `ft_service_buffer_file()`, which validates the buffering request.
- `pcbuf_buffer_file()`, which performs the actual buffering.

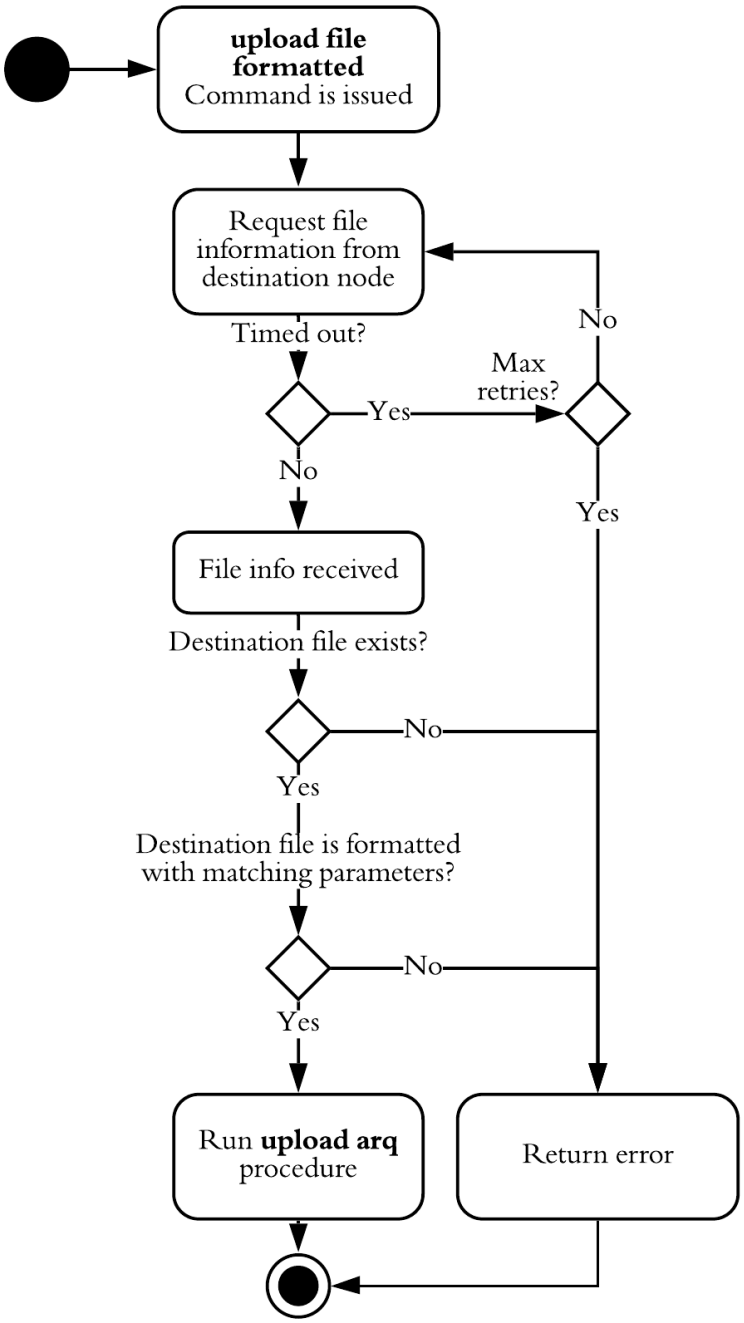


Figure 5.21: The `upload file formatted` procedure, which upon verifying that a valid, formatted destination file exists, will attempt to upload data to it.

Upon receiving a buffering request sent with `ft_client_buffer_request()`, the `ft_service_buffer_file()` checks that the specified file exists. A valid request is passed to the `pcbuf_buffer_file()` procedure which uses the `pcbuf` module to send the file contents to the PC buffering service.

When buffering a new file, the buffer file must be cleared before sending data to it. The whole procedure, with clearing, buffering and subsequent direct downloading is illustrated in Figure 5.25.

When buffering, the file format that is outlined in Section 5.2.1 is not used. The raw file

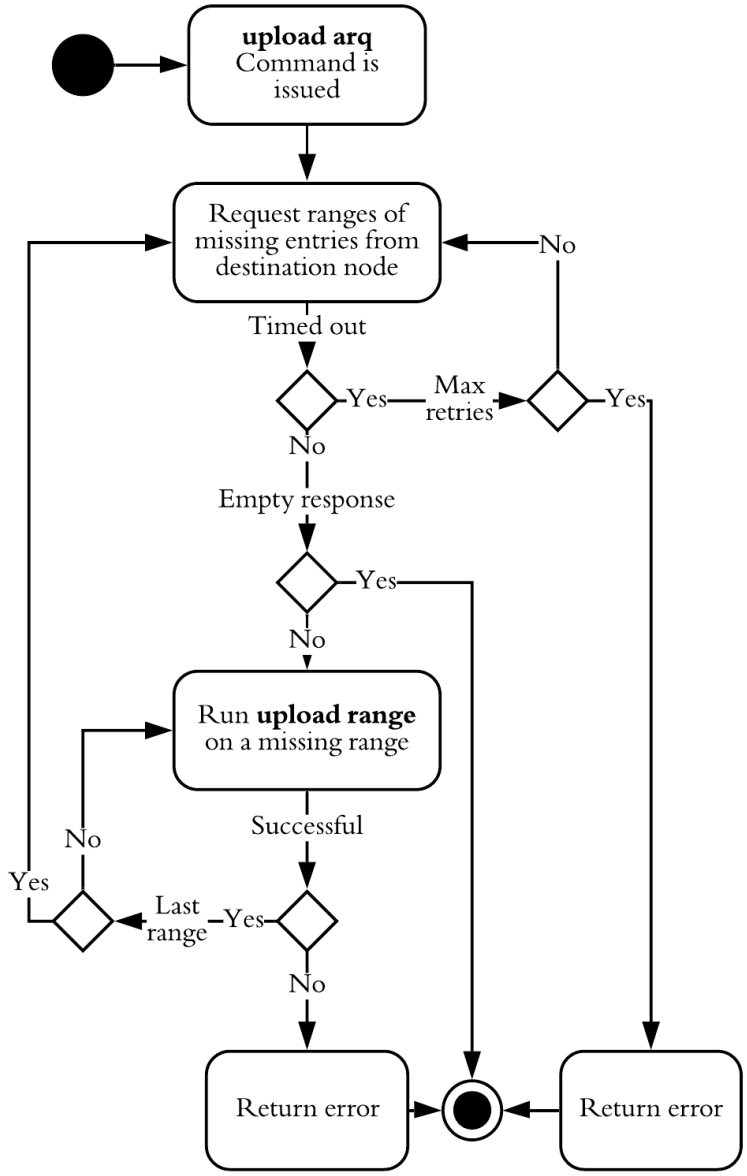


Figure 5.22: The `upload arq` procedure, which attempts to reliably send a formatted file by requesting missing entries.

data is inserted directly into CSP packets without any metadata. The buffering service in the PC accepts and appends the data of each packet at the end of the buffer file.

Reliability

The buffering procedure employs a *stop-and-wait* strategy. Each packet of raw file data is acknowledged with an ACK packet. There is no sequence number, meaning that the transmission may theoretically suffer from the problem described in Section 2.2.1. If the data packet is lost, then no ACK is generated, such that the buffering client can retransmit the data packet. However, if an ACK packet is lost and a retransmission of the data is issued, then the buffering service may incorrectly interpret the retransmitted data as the next data.

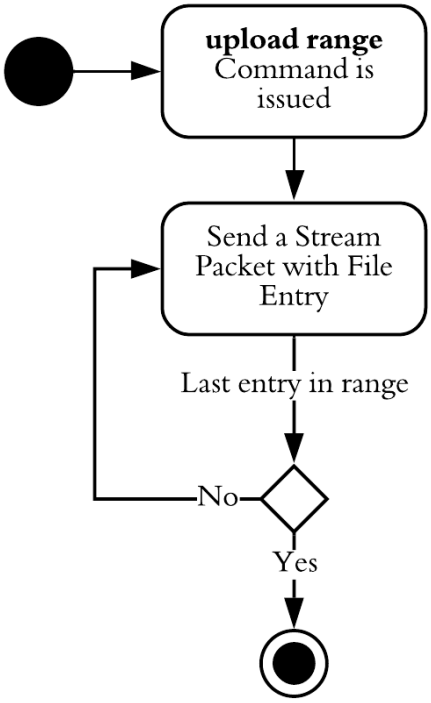


Figure 5.23: The `upload range` procedure, which sends a range of entries as a file stream.

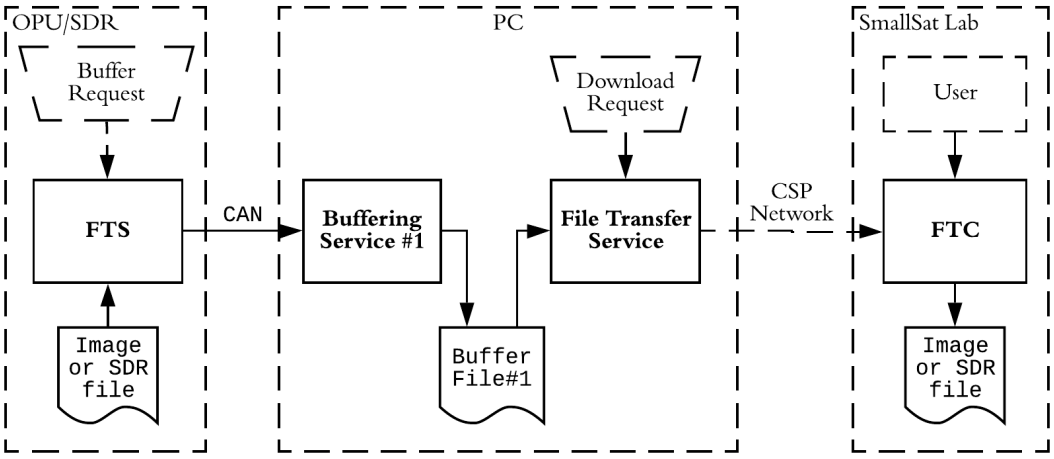


Figure 5.24: Buffered download mode. A file is buffered on the PC before being downloaded.

Store and Forward

Buffering is not used to solve the same problem as in CFDP. Whenever the radios have an established connection to the ground, the payloads also have a connection to the ground, so the feature is not being used to work around a partially connected network. The issue it solves is the CAN-bus being a bottle neck in communication chain. By storing the file in the PC, the faster SPI interface becomes the slowest link during a direct download from the PC.

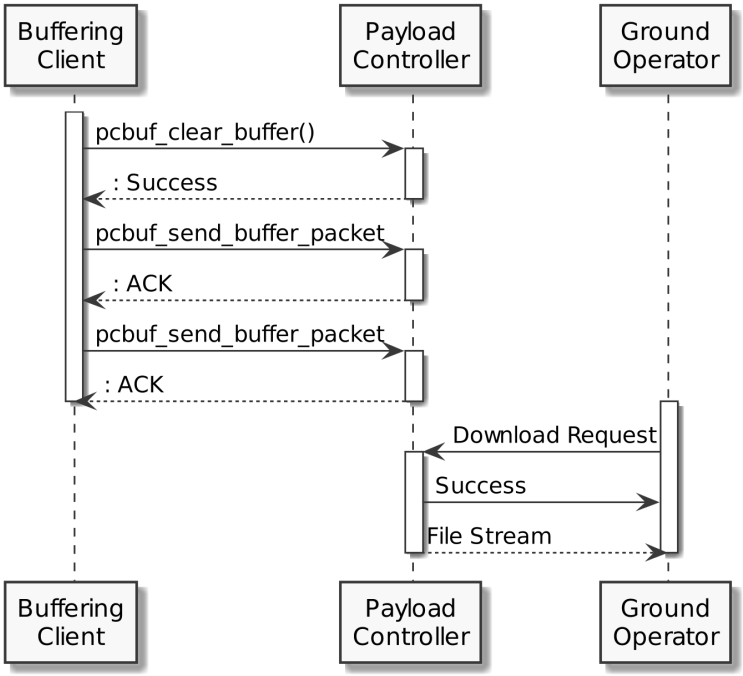


Figure 5.25: Sequence of buffering data on the PC, and then downloading it with a direct download procedure.

Transfer Speed

The transfer speed from the PC to the ground is fully dependent on the radio link quality. The effective data rate of the S-band link is approximately 0.8 Mbps [21], almost double the effective rate of CSP over the CAN-bus [12].

Chapter 6

Testing & Results

A BOB has been designed for the OPU payload, and a FT system has been designed and implemented.

This chapter presents the achieved results. The BOB is presented before the hardware test setup is outlined.

Tests are performed to check or measure:

- Correctness of `fs` and `ft` modules (Section 6.3).
- Correctness of service and client interactions (Section 6.4).
- Communication delays between subsystems (Section 6.5).
- Effective data rates in the network stack (Section 6.6).
- Reliability of the FT system (Section 6.7).
- Buffering capability of the PC (Section 6.8).

6.1 Breakout Board

The BOB design process has resulted in an implementation that should be able to fulfil the requirements of all connected components.

The achieved features of the implemented BOB are listed as results.

- The PZ is provided a connection to the M6P CAN-bus.
- The BOB is supplied with `VBAT` and 12 V power from the EPS.
- BOB distributes regulated power to PZ, HSI and RGB.
- The HSI camera is provided a data and control interface over Gigabit Ethernet.
- The RGB camera is provided a data, control and power interface over USB.
- The flash signal of the HSI camera is provided a connection, which is used to timestamp the captured images.
- The PPS signal from the PC is provided a connection, which will allow the OPU to synchronise its local real time clock.
- The PZ is provided with a SD storage device to store image data.

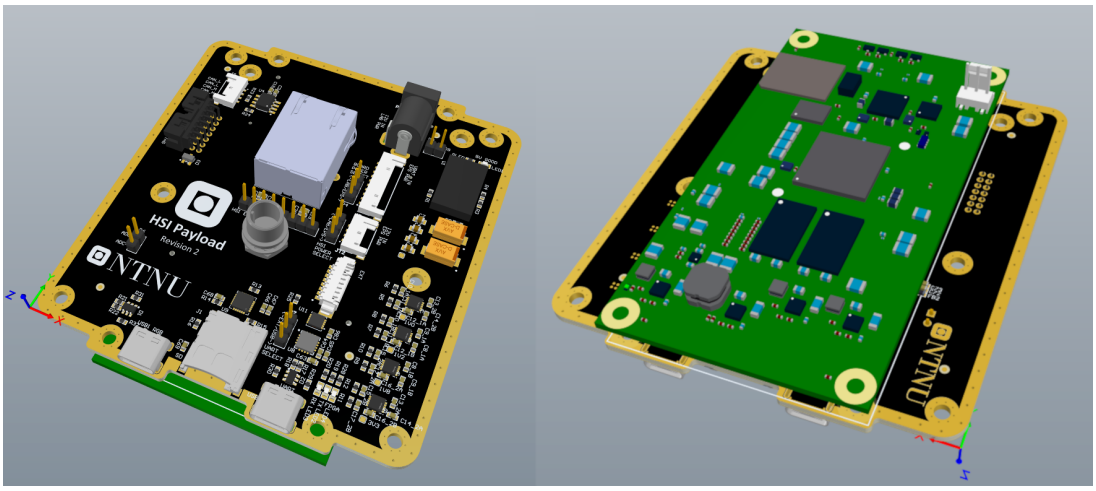


Figure 6.1: A rendering of front side and back side of the final Breakout Board PCB. A model of the PicoZed SoM is included on the back side render to illustrate how the Breakout Board and PicoZed will look like when they are connected.

A render of the final implementation is shown in Figure 6.1.

The BOB was not manufactured in time to be tested for this thesis. The PZ SoM was also not tested because it depends on the BOB.

6.2 Hardware Test Setup

All tests that include payload hardware were performed using the ZedBoard development kit (see Section 4.1.3).

The PC and EPS subsystems that were used during testing were pre-programmed with a 125 kbps CAN-bus rate. New firmware that enables a 1000 kbps CAN-bus rate was eventually provided by NA, but not in time for the testing for this thesis. In order to connect to the M6P subsystems, the OPU payload has therefore been tested at 125 kbps instead of the full CAN-bus rate of 1000 kbps.

A photo of the hardware setup that is used for testing is given in Figure 6.2. A block diagram of the same setup is provided in Figure 6.3.

The PC is in the top left corner. The EPS is in the top right corner. The ZedBoard development kit, which runs the OPU services, including the FTS, is in the bottom left corner. The M6P CAN-bus, `CAN1`, is annotated as (1). `CAN1` connects to the development workstation via a CAN-bus adaptor. The development workstation runs the FTC. `CAN1` also connects to the PC and EPS. The payload CAN-bus, `CAN2`, is annotated as (2). `CAN2` connects the PC to the Zedboard. The ZedBoard is connected to `CAN2` via a logic level shifter circuit and CAN transceiver circuit shown as (3). The ZedBoard is provided 12 V power from a wall socket adaptor (4). The PC is provided 3.3 V power from output channel 5 on the EPS (5). The EPS is being charged at 12 V from a wall socket adaptor (6).

6.3 Automated Module Testing

The modules created for this thesis are implemented for payloads that run Linux on an `ARM` architecture. Since the modules are implemented for Linux, they can be tested on the development workstation (`x86_64` Linux) with no change to the source code. This allows the modules to be

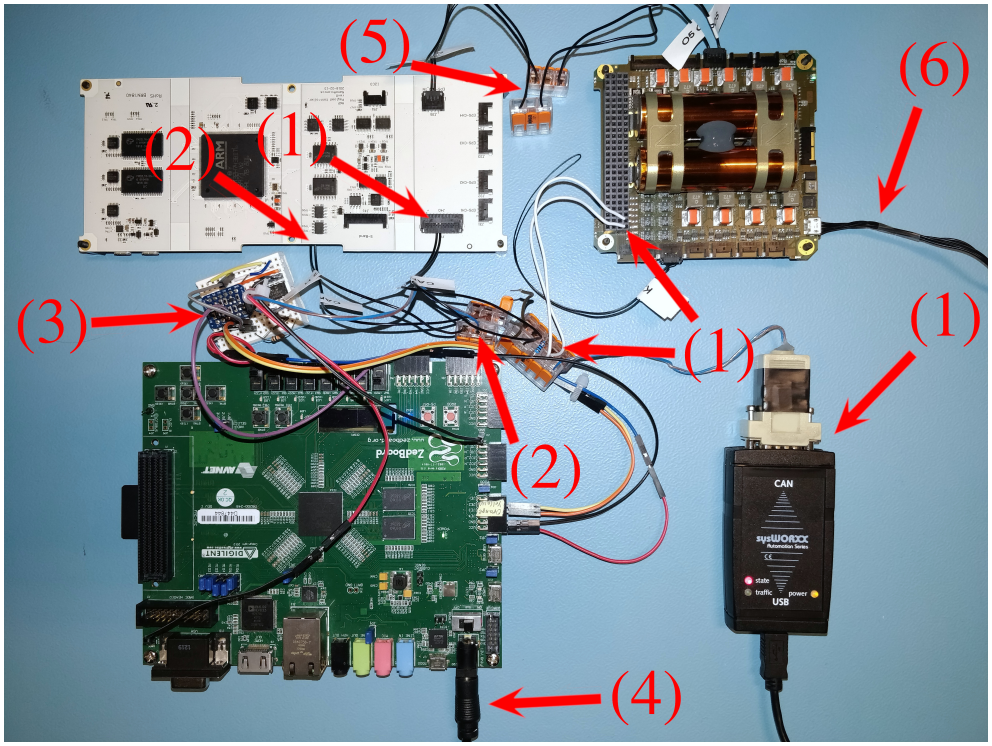


Figure 6.2: Hardware setup used for all tests that were performed on the ZedBoard. (1) shows `CAN1`. (2) shows `CAN2`. (3) shows CAN transceiver. (4) shows lab power for ZedBoard. (5) shows PC power from EPS. (6) shows charger cable for EPS.

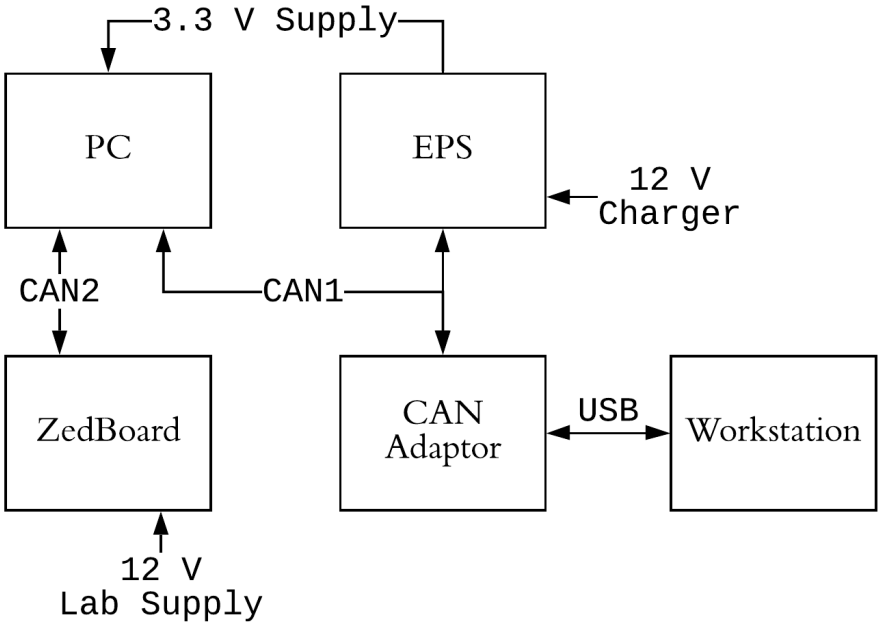


Figure 6.3: Block diagram of the hardware setup in Figure 6.2.

frequently tested during development, even automatically. This way, regressions can quickly be detected.

The CMake build system[1] is used to compile and link the applications, and the CTest subsystem of CMake is utilised to run tests [31]. CTest is CMake's test driver program, and is instructed via `CMakeLists.txt` configuration files to run specified programs as a test suite. Results are collected from the tests and presented on the command line. Jenkins[2] was used throughout parts of the thesis work to run the test suite automatically whenever new code was pushed to the Git repository.

Unit Tests

Some of the C modules, like `fs`, `fs_idmap` and `fs_log` (implemented to test the logging capabilities of the M6P subsystems) can be functionally tested with a limited amount of dependencies to other modules. The only dependency of `fs` is to `crc32`, and `fs_idmap` and `fs_log` is only dependent on `fs`. Unit tests have been created for these.

The unit testing framework *Check*[3] is used to create unit tests for the low level modules. The tests are compiled and then run with CTest.

Automated System tests

The *client* and *service* modules are challenging to automatically test because they have many interdependencies. Both the client node and the service node must be initialised and run concurrently in order to exchange CSP packets and perform meaningful tasks. This requires the test code to initialise the nodes and place them into a known state before initiating the actual test case. Consequently, it takes more time to write tests for these.

Despite the challenges, some of the FTC and FTS functionality has been given automatic test coverage. These cover basic request/response transactions and basic download functionality over a lossless channel. Most client and service functionality, however, does not have automatic test coverage.

Automatic tests for the flatsat have been created. These are not run to verify the correctness of the M6P subsystems, but rather to verify the communication between payload and flatsat. The tests are used to diagnose connectivity issues, which have proved to be a frequent occurrence.

Automated Test Results

The functionality that was given automated test coverage has been continuously tested throughout the thesis work. All tests are passing successfully. The CTest report is presented in Listing 2 as a result.

6.4 HYPSO CLI

The majority of system level functionality has been tested manually. A CLI program has been created to perform this testing.

The `hypso-cli` program is implemented as a Read–eval–print loop (REPL), it takes readable text commands as input from a human operator, executes them, and presents the results or output.

[1] https://cmake.org/
[2] https://jenkins.io/
[3] https://libcheck.github.io/check/

```
make -C build/x86 CTEST_OUTPUT_ON_FAILURE=1 test
make[1]: Entering directory '/home/magne/repos/hypso/build/x86'
Running tests...
Test project /home/magne/repos/hypso/build/x86
    Start 1: test_fs
1/5 Test #1: test_fs ........................   Passed    0.44 sec
    Start 2: test_fs_idmap
2/5 Test #2: test_fs_idmap ..................   Passed    0.17 sec
    Start 3: test_log
3/5 Test #3: test_log .......................   Passed    0.02 sec
    Start 4: test_ft_service
4/5 Test #4: test_ft_service ................   Passed    4.59 sec
    Start 5: test_shell_service
5/5 Test #5: test_shell_service .............   Passed    0.33 sec

100% tests passed, 0 tests failed out of 5

Total Test time (real) =   5.56 sec
make[1]: Leaving directory '/home/magne/repos/hypso/build/x86'
```

Listing 2: Reported results from automated tests run with CTest.

In most situations, the `hypso-cli` program acts as a client that sends requests and receives replies from services. It does this by connecting to a CSP network via a CAN interface or a NNG/TCP interface. The CAN interface can be connected to the engineering model of the satellite, or directly to a payload. The NNG/TCP interface can connect to the flatsat.

The capabilities of the `hypso-cli` program are summarised in Appendix C. A terminal dump is provided for the `help` command which prints the available commands, their arguments and a corresponding help text.

6.4.1 File Transfer Client

An example of a CLI interaction is illustrated in Figure 6.4, where a user downloads a file from a FTS. After starting the CLI program, the user can insert a text command. If the parser can deduce a valid command from the input, it is promptly executed with the provided arguments. A download command results in a series of `ft_client` module functions to be executed. A `file info` request is first exchanged to verify that the source file exists, and that the destination file is correctly formatted. Then a `file download` request is exchanged to initiate the download stream. All output from the `ft_client` module is printed for the user to see, and the final return code is reported.

6.4.2 Remote Shell

A remote shell module was implemented to aid development. The remote shell enables an user to execute arbitrary shell commands on a remote CSP node.

A shell service module runs as a task on the payload, listening for shell command packets. A corresponding shell client module accepts text strings which it transmits as commands to the shell service. An example where the shell command `uname -a` is executed is shown in Figure 6.5.

When executing a command, the shell service wraps the command in a `timeout` command. This prevents the shell service from locking up when waiting for a command that does not terminate in a reasonable amount of time.

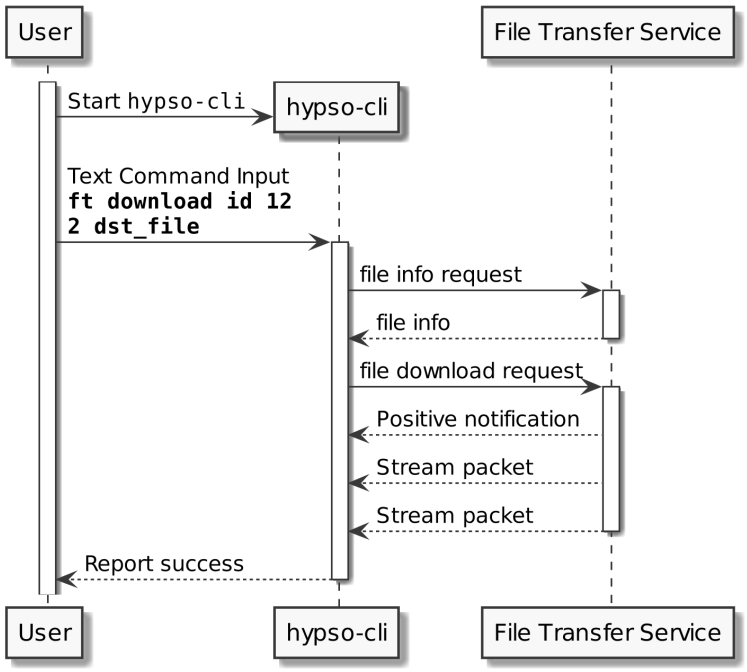


Figure 6.4: Example use of the `hypso-cli` program. A file transfer request is sent to a FTS, and the result is reported back to the user.

The implementation of the shell client module is made compatible with the remote shell capabilities of the M6P subsystems. This is useful for system integration because it lets the developer access the M6P subsystem shells over the CSP network, without having to attach an extra serial cable to each subsystem.

Although it is not a listed requirement, the author suggests that the service be included in the flight version of the payloads. Access to a shell on the payload can be a powerful tool for in-orbit debugging and development.

6.4.3 Loopback Services

In addition to performing the role of a client, the `hypso-cli` program can also run payload services locally. The payload services can be run concurrently with the command interpreter, and by utilising the loopback capabilities of the CSP library, the client and service can communicate with each other within the same Linux process. The concept is illustrated is Figure 6.6.

The primary reason for running the services locally in the CLI program is that the services can be tested quicker than if the client and service had to be deployed to different machines. If deploying on different machines, either the source code or the compiled program binaries need to be distributed before the client or service can be executed.

A downside to running the services in the same program as the CLI is that the service being tested is not the same program as would be run on the payload, and it is not compiled for the same architecture. It has other threads running concurrently, contesting for resources such as CSP packet buffers.

An alternative to running services in the same program is to run them in a standalone program, but still on the same computer. Just like the CSP enabled payload program must be able to connect to an external CAN-bus, it can connect to a *virtual* CAN-bus. A *virtual* CAN-bus is interfaced just like a regular Linux CAN device, but acts as a loopback interface. It does not represent an external CAN-bus, and only processes on the same machine can connect

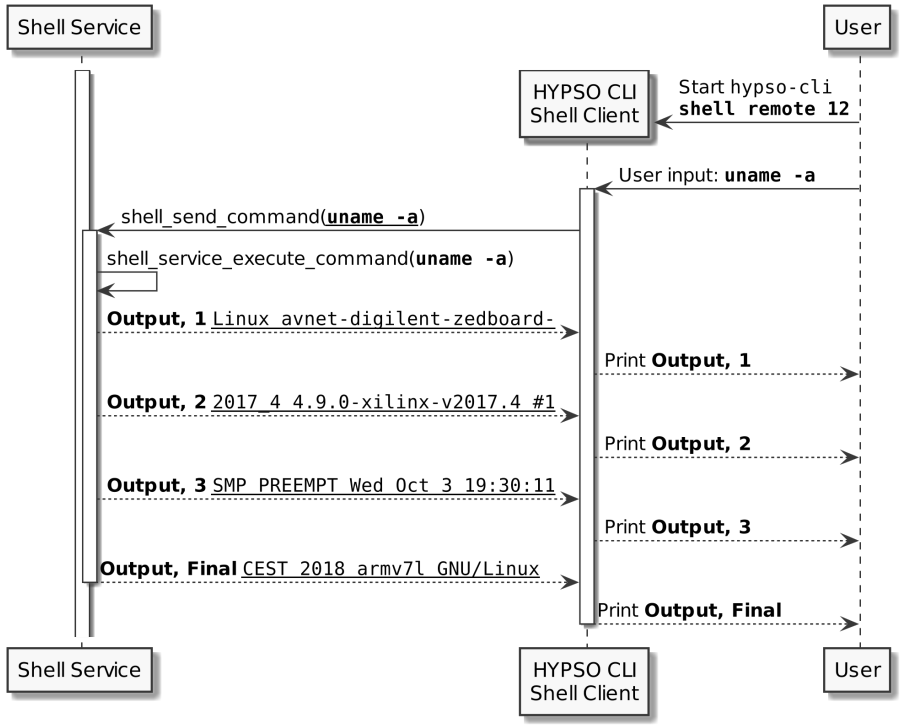


Figure 6.5: Remote shell sequence. An user inputs a command that is sent to the shell service and executed. The output from the shell command is returned to the shell client and printed for the user.

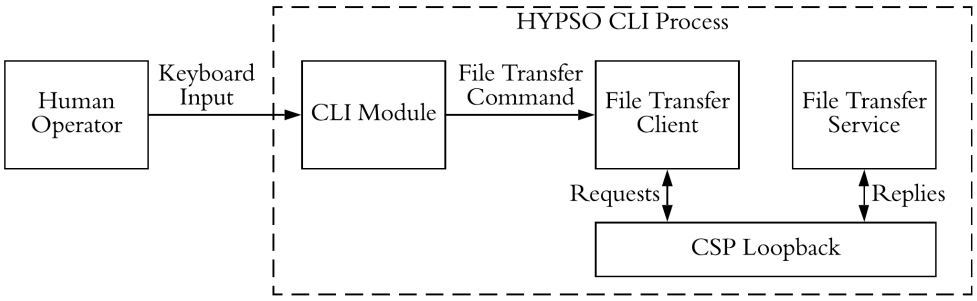


Figure 6.6: Setup for performing system tests on development workstation.

to it.

The CSP loopback feature and virtual CAN devices have both been leveraged to perform manual and automatic functional system testing on a single development machine.

6.5 Communication Delays

Communication delays between satellite subsystems are measured on the hardware setup. The listed delays are averages of measured RTTs. The RTT includes the time it takes for the interfaces to receive and transmit packets (transmission time), as well the time it takes CSP to exchange packets with service and client code (processing time), as summarised in equation (6.1).

$$\text{Communication Delay} = \text{Transmission Time} + \text{Processing Time} \tag{6.1}$$

The setup for measuring the communication delays is shown in Figure 6.7.

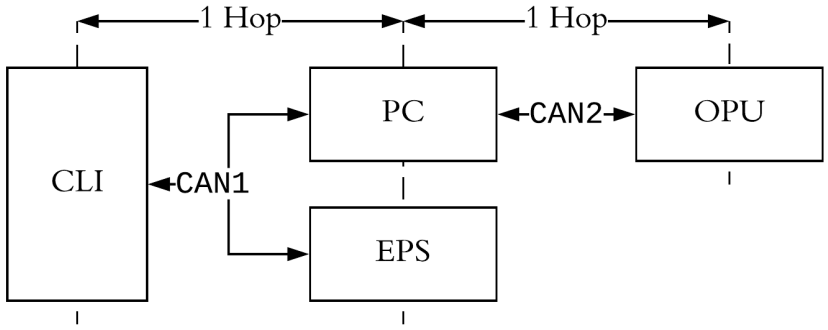


Figure 6.7: Setup for measuring communication delays.

The RTT is measured by sending CSP ping packets. The ping packets have empty data fields, but still contain six bytes worth of CSP identifier and length information. The transmission time of these six bytes is directly proportional to the CAN baud rate. For this reason, different results are expected when using the full CAN baud rate of 1000 kbps.

The measured communication delays for 125 kbps baud rate are listed in Table 6.1 as average RTTs.

Table 6.1: Round trip delay times for the subsystems in the test setup.

Node	Repetitions	Averaged RTT [ms]	Distance [Hops]
PC	1000	2.58	1
EPS	1000	2.78	1
OPU	1000	4.73	2

6.6 Effective Data Rates

The data rate at each level in the network model is measured. The setup from Figure 6.7 is used. All CAN-buses are run at 125 kbps.

- Link layer CAN-bus: data rate is measured with the `canbusload` Linux utility program.

- Network and Transport Layer CSP: raw data rate over CSP is measured by using TFTP in unacknowledged mode, and subtracting TFTP overhead.
- Application layer FT system: effective data rate is measured by transferring a file.
- Application layer TFTP: effective data rate is measured by transferring a file with the *send-and-wait* strategy.

The measured data rates are summarised in Table 6.2.

Table 6.2: Effective data rates of different layers and protocols in the network stack.

Layer	Protocol	Data rate [kbps]	Ratio of CAN [%]
Link	CAN	57.87	100.00
Network/Transport	CSP	56.50	97.63
Application	FT system	52.00	89.86
Application	TFTP	25.95	44.84
Application	TFTP, no PC	51.87	89.63

The measured effective data rate of the CAN-bus is 46.29 % of the signalling speed, and 94.77 % of the theoretical upper limit on maximum utilisation (equation (2.3)).

Table 6.2 shows that the effective data rate over CSP is 97.63 % of the measured CAN data rate. The combined overhead of CSP and the FT system puts the effective data rate of the FT system at 89.86 % of the measured CAN data rate. This suggests an average overhead of 10.14 % when transferring files, although this does not account for the communication delays for the requests.

The data rate of TFTP is included to compare the FT system with a *stop-and-wait* strategy. The significant decrease in data rate for TFTP is caused by the large RTT that is caused by having to go through the PC. For a *stop-and-wait* protocol such as TFTP, the data rate is directly proportional to the RTT. This was confirmed by repeating the experiment without the PC, which halved the RTT and doubled the effective data rate.

6.7 Packet Loss Test

The reliability of the FT system must be tested and demonstrated. This section details a setup for testing the FT system over an unreliable channel.

Setup

The test setup is illustrated in Figure 6.8. The `hypso-cli` program is designated as the FTC, while a ZedBoard is configured with a FTS. To simulate a situation with a more realistic CSP network, the M6P PC is included as an intermediary CSP node that acts as a packet router. The PC connects to the ZedBoard payload via `CAN2` and to the development workstation that runs the `hypso-cli` program via `CAN1`. Additionally, a CSP bridge that drops packets at random is included in the communication chain. This bridge program connects to the external `CAN1` bus, and to the `hypso-cli` program via a virtual CAN-bus.

Test parameters

The packet dropper bridge can be configured to drop a certain percentage of packets. It can also be configured to drop bursts of packets instead of dropping packets evenly. The bridge can

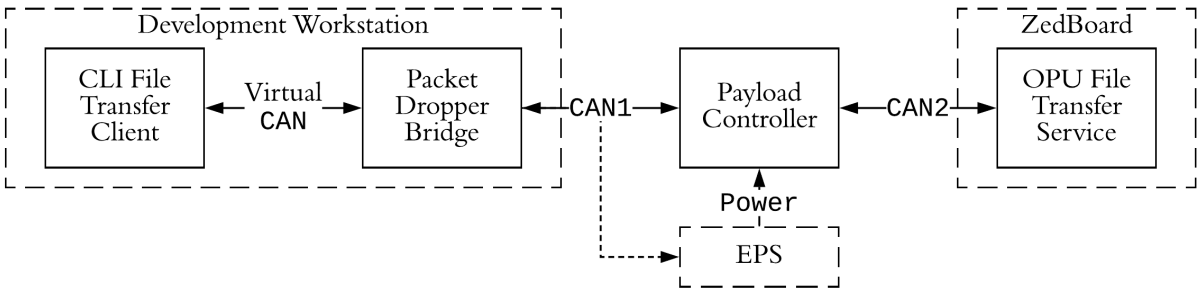


Figure 6.8: Setup for testing the reliability of the FT system. A CLI program acts as a FTC which sends file transfer requests to a FTS on the ZedBoard. A CSP bridge drops packets to mimick an unreliable radio channel.

decide to only drop packets in a single direction to simulate an asymmetric radio link, but for this test it does not discriminate between space-bound and Earth-bound packets.

Combinations of channel configurations are tested to characterise the system. A number of drop rates are tested and compared to equivalent drop rates with bursts.

Test results

The same 769.541 KB image, is transferred in all configurations. The transfer duration is recorded, and the effective data rate calculated.

The test results are shown in Figure 6.9. The collected measurement data is included in Appendix D.

6.8 Payload Controller Buffering

The buffering capabilities of the PC are tested. The typical use case for buffering is to download a large file.

The hardware setup from Figure 6.2 is used for testing the buffering capabilities. The data path is illustrated in Figure 6.10.

The payload (ZedBoard) FTS is connected to the PC via `CAN2`, and the PC is connected to the ground (workstation) FTC via `CAN1`. The PC hosts a buffering service, a buffer file, and a M6P file transfer service.

The test is carried out by moving a file through the entire data path, from FTS on the ZedBoard to the FTC on the workstation.

A buffering request is first sent to the FTS. The FTS then starts the buffering procedure, in which the file is fragmented and sent to the buffering service on the PC using the *stop-and-wait* strategy. When the whole file is buffered on the PC, it is downloaded from the PC file transfer service using the reliable direct transfer mode.

Functional testing shows that a 769.541 KB file could be buffered and then downloaded successfully. For the 125 kbps CAN bus, the data rate was 52.17 kbps when buffering from OPU to PC, and 47.50 kbps when downloading from PC to the CLI node.

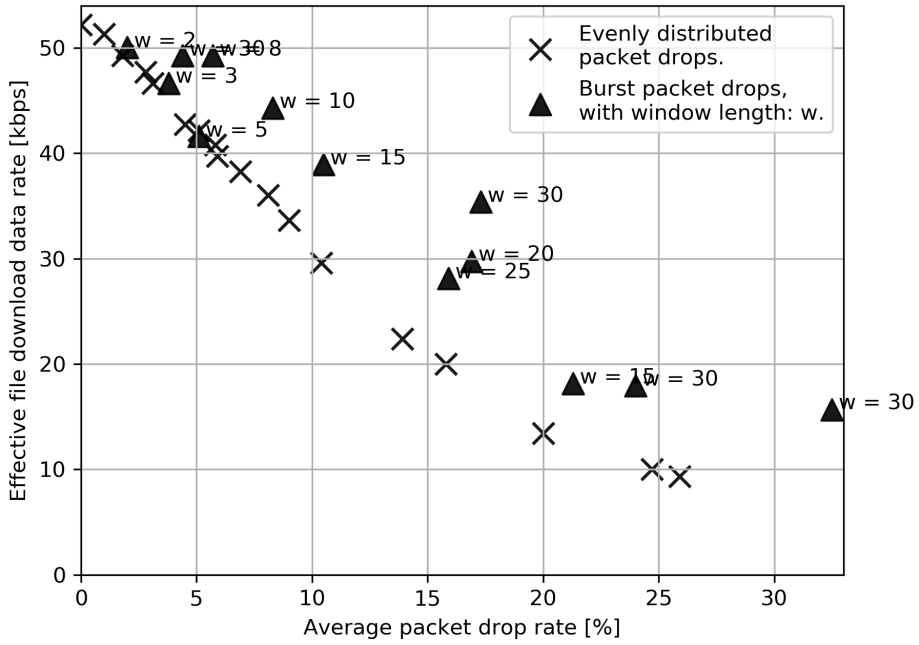


Figure 6.9: Effective data rates of the file transfer system for various packet drop rates. Configurations with packets dropped evenly are shown as crosses and configurations with packets dropped in bursts are shown as triangles.

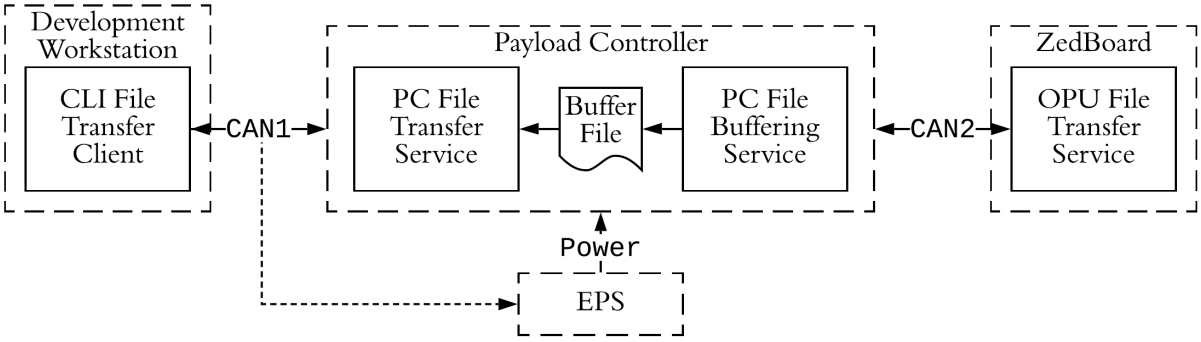


Figure 6.10: Setup for testing the buffering service on the PC.

Chapter 7

Discussion & Conclusion

In this chapter, the initial requirements are reviewed in order to discuss the achieved results. Concluding remarks are made.

7.1 Fulfilment of Requirements

The requirements from Chapter 3 are reviewed to determine whether the implemented FT system fulfils them. A summary of the requirements and their level of fulfilment is provided in Table 7.1.

Table 7.1: Summary of the design requirements, and the level to which they were fulfilled.

Requirement	Relevance for FT system	Level of fulfilment
IF-001	Compatibility with M6P.	Full
MS-0-011	Downlink 1 image of L1A data.	Full
MS-0-012	Downlink L4 data in 3 hours.	Partial
MS-0-14	Downlink telemetry in 1 pass.	Full
M-2-026	Downlink telemetry less than 200 kb.	Full
SDR-SMO-1	Downlink measurement data.	Full
MS-0-013	Uplink mission data and code.	Full
M-1-012	Uplink mission data and code.	Full
M-2-015	Uplink mission data 5 min before.	Partial
SBUS-3-017	Ability to upgrade OPU software.	Full
M-2-014	Up/downlink from multiple stations.	Out of scope
SDR-SMO-4	Ability to upgrade SDR software.	Full
M-1-015	Downlink L1A data in 24 hours.	Partial
M-1-016	Downlink L4 data in 3 hours.	Partial

The requirement of being compatible with the M6P bus (**IF-001**) is fully fulfilled by employing the CAN protocol and CSP protocol in the communication stack.

All requirements that state an ability to download or upload files, without specifying a time limit, have been satisfied as a result of producing a FT system capable of up downloading and uploading files. This includes **MS-0-011**, **SDR-SMO-1**, **MS-0-013**, **M-1-012**, **SBUS-3-017** and **SDR-SMO-4**.

The requirements **MS-0-12**, **M-2-015**, **M-1-015** and **M-1-016** state a time limit. The FT system was tested with a CAN-bus rate of 125 kbps, and achieved an effective data rate of 52 kbps when downloading over a perfect channel. The timing requirements have been created with the assumption that the files could be downloaded at the full 1 Mbps non-effective data rate of the S-Band radio [32]. The achieved effective data rate of 52 kbps (or even a projected 416 kbps if assuming a 1000 kbps CAN-bus) does not suggest that the timing requirements would be fulfilled if using the direct transfer mode.

If using the *store and forward* capabilities of the PC, where files are stored on the PC in advance of downlinking them, the full data rate of the S-Band radio can be used. In this case the timing requirements would be satisfied, but they are still listed as *partially* fulfilled because the S-Band radio was not available to demonstrate an effective data rate.

The requirements **MS-0-14** and **M-2-026** are timing requirements and have been listed as *fully* fulfilled. These timing requirements assume an UHF radio link with a 9.6 kbps data rate, which is within the data rates that the FT system is able to handle with the 125 kbps CAN-bus rate.

The requirement **M-2-014** is considered out of scope for the FT system. The ground station network is handled by the MCS that is provided by NA. The FT system will work as intended as long as there is CSP connectivity throughout the whole mission network.

7.2 Channel Utilisation

The effective data rate of the FT system is approximately 52 kbps, as shown in Table 6.2. If assuming a proportional increase in effective data rate when switching to a CAN rate of 1000 kbps, one can expect an effective data rate of 416 kbps. This is 85.2 % of the theoretical limit on the effective CAN data rate that was established in Section 2.4.

The communication delay between FTC and FTS is demonstrated to be the primary driver of the effective data rate in protocols using *stop-and-wait* strategies, such as TFTP, as shown in Table 6.2. This effect would be further increased by the communication delays added by the radio links and ground station processing. The FT system is less prone to these delays, as it attempts to transfer large ranges of entries without acknowledgement. However, the effect *is* apparent in the FT system during the final stages of a transfer. Lost packets or ranges must be requested individually, causing the FTC to experience delays while waiting for replies.

Reliability

The results of the packet loss test (Figure 6.9) show that the FT system is able to deal with packet loss rates of up towards 30 %. A significant drop in effective data rate is experienced as the packet drop rates increases. The plot suggests that the FT system will struggle with packet drop rates of 30 % or above.

The system seems to achieve higher effective data rates for channels that exhibit bursts of packet drops rather than evenly distributed packet drops. This was expected from the design of the FT system. A burst causes a continuous range of entries to be dropped, all of which can be requested with a single download request. Since evenly distributed packet drops are unlikely to be neighbouring, the majority of them must be requested with individual download requests. A large number of download requests results in a large accumulation of transmission delays, during which no data is being transferred. Therefore, it can be expected that the FT system will perform worse if the packet drops are evenly distributed.

The drop rate in the packet drop test is specified in terms of CSP packets, and not in terms of link layer drop rates. The effect of various drop rates for the S-Band radio should be investigated. If a CSP packet is fragmented into multiple frames at the S-Band link layer, a CSP packet could

be dropped as a result of a single link layer frame being dropped. In this case the FT system might benefit from sending smaller file segments, even though it increases the overhead of each packet.

Congestion Monitoring

The traffic on a CAN bus be measured by any node that is connected directly to it. This means that congestion on `CAN2` can be directly measured by the payloads. However, the payloads do not have direct access to `CAN1` and so can not measure traffic on `CAN1`. When transferring files with the S-Band radio, the payloads have no way of directly measuring the traffic from PC to S-Band radio.

Congestion must therefore be estimated by monitoring packet loss.

7.3 CSP Buffer Exhaustion

The CSP library allocates a bank of packet buffers. If an application is unable to process packets at the same rate that they arrive in, the buffer bank will be exhausted and incoming packets will be dropped.

This situation can occur under erroneous circumstances such as a task deadlocking, or a blocking system call that stalls a task for an unexpectedly long time.

The buffer bank was monitored during multiple file transfers. The bank was never found to drop below a few percent. These transfers were carried out on a 125 kbps CAN rate, which places an unrealistically low limit on the received data rate. Buffer exhaustion could be more likely on a 1000 kbps CAN-bus.

7.4 Buffering

An added benefit of the PC buffering capability is that it can be used to extend the effective storage space of the payloads. The M6P file transfer system also has a higher technology readiness level than the HYPSO payloads, demonstrated through flight heritage. If the direct download mode for any reason stops functioning, the buffering capability can be used as a redundant system for downloading files, provided that the buffering procedure on the payload still works.

The reliability concern that was described in Section 5.3.7 was monitored by checking the integrity of buffered files while testing the buffering capabilities . However, no attempt was made to provoke a fault.

Throughout the work of this thesis, a duplication of data as a result of a lost ACK was never detected. The problem was therefore not proven to exist. There is only a single CAN-bus between the payloads which send the data and the PC which receives the data. It is therefore thought to be unlikely, but not impossible, for buffered data to be lost.

7.5 Memory Footprint of Formatted Files

As mentioned in Section 5.2.1, the file format that is used in the FT system can be changed to append the metadata instead of interleaving it. The benefit of doing so is that a formatted file could be created *in-place* in the original file without moving the original data. This could be advantageous if storage space appears to be a scarce resource.

Such a change should only be made after an assessment has been carried out on the risk of corrupting the original file when modifying it *in-place*.

7.6 On the use of Linux

Some benefits and drawbacks of using Linux have already been discussed in Section 4.1.5.

M6P FreeRTOS

The M6P subsystems use FreeRTOS. This operating system is significantly less complex than Linux. In fact, it has been designed to be small and simple [33]. Consequently, the system has fewer states and is generally easier to verify.

The fact that the M6P subsystems and the payloads do not run same operating system means that they also cannot share source code that uses operating system primitives.

CubeSat Space Protocol

By using CSP, one ends up with ad-hoc protocol solutions, because CSP does not readily interface with standard Linux networking systems. One way of justifying the use of CSP is that small satellites require a smaller footprint than the machines that use the Linux networking system, which are designed for the requirements of the global internet. The M6P subsystems that use FreeRTOS are examples of such resource limited systems.

Some of the benefits of using Linux are therefore not leveraged. The OPU has available the Linux networking system, which is robust and dependable from having been heavily scrutinised and reviewed. Despite this, a custom networking solution had to be made to fit the M6P compatibility requirement **IF-001**.

7.7 Future Work

Breakout Board

The manufactured BOB must be tested. The voltage regulators must be verified to be providing the correct voltages. All interfaces must be tested after connecting the PZ SoM to the BOB.

Integration testing should verify that the BOB can successfully interface the HSI camera, the RGB camera and the M6P CAN-bus at the same time.

Channel Utilisation

Further work on the FT system could incorporate a mechanism to prevent congestion by adjusting the sending speed based on reported packet loss.

Communication Network

A few assumptions have been made about the ground station network. Since it is not fully defined or implemented yet, the FT system has been designed with the expectation that the FTC can connect directly to the CSP network (Figure 4.4) that is automatically routed through ground stations and to the space segment.

The integration of the FT system into the ground station network must be performed once documentation for the ground stations and MCS is available.

Operations

Higher level procedures that can automatically prepare and extract formatted files should be created, and must be integrated with procedures for capturing images. A scheduler is required to execute mission plans.

7.8 Conclusion

The work in this thesis documents the implementation of hardware and software that is required to integrate the HYPSO payloads with the M6P satellite platform. This includes integration of an operating system, hardware design for a BOB, and design, implementation and testing of a FT system.

Specific contributions:

- A BOB for the OPU payload has been specified, designed and manufactured with the help of NTNU SmallSat staff (Section 4.1.4).
- Configuration and application of an operating system for the OPU payload (Section 4.1.5).
- A functioning FT system has been designed and implemented, while keeping compatibility to M6P in mind (Chapter 5).
- A FT system has been implemented and tested with M6P hardware (Chapter 6).
- A CLI program for interfacing the payloads has been implemented (Section 6.4).
- A shell service for interfacing the payloads has been implemented (Section 6.4.2).

Test results show that the implementation of the FT system is capable of transferring files. Files can be transferred with effective data rates comparable to the data rates of the underlying protocols, and even over an unreliable network. The FT system should therefore be able to carry out the tasks necessary to achieve at least partial mission success.

The quality requirements in Section 3.4 have not been fully verified. The ultimate effective downlink data rate depends on the effective data rate of the S-Band radio, which was not available for testing. The FT system needs to be tested again when a more complete setup is available.

Further work remains to test and verify the BOB, and to fully integrate the FT system into the ground station network.

Bibliography

[1] California Polytechnic State University, *6U CubeSat Design Specification*, 7 2018.

[2] Nanosats Database, "Nanosatellite cubesat database," 2019. `https://www.nanosats.eu/#info`.

[3] "Andreas Budalen, Markus Thonhaugen, Andreas Nilsen Trygstad, "Oppdretter fortviler: – Algene har drept laks for opp mot 200 millioner," *NRK*, 2019.

[4] R. Birkeland, *On the Use of Micro Satellites as Communication Nodes - in an Arctic Sensor Network*. PhD thesis, Norwegian University of Science and Technology, 1 2019.

[5] Fred Sigernes, Mariusz Eiving Grøtte, Julian Veisdal, Evelyn Honore-Livermore, João Fortuna, Elizabeth Frances Prentice, Mikko Syrjäsuo, Kanna Rajan, Tor Arne Johansen, "Push-broom Hyper Spectral Imager version 6 (HSI v6) part list - Final prototype," tech. rep., University Centre in Svalbard (UNIS), Norwegian University of Science and Technology (NTNU), 2018.

[6] J. Wertz, *Space mission engineering : the new SMAD*. Hawthorne, CA: Microcosm Press Sold and distributed worldwide by Microcosm Astronautics Books, 2011.

[7] A. J. Menezes, *Handbook of applied cryptography*. Boca Raton: CRC Press, 1997.

[8] A. Tanenbaum, *Computer Networks*. Upper Saddle River, N.J: Prentice Hall PTR, 1996.

[9] E. D. Pål Frenger, Stefan Parkvall, "Performance Comparison of HARQ with Chase Combining and Incremental Redundancy for HSDPA," in *IEEE 54th Vehicular Technology Conference. VTC Fall 2001. Proceedings*, pp. 1829–1833, 2001.

[10] C. Doerr, *Network Security in Theory and Practice*. Christian Doerr, 2018.

[11] International Organization for Standardization (ISO), *ISO11898-3 Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface*, 2006.

[12] Magne Hov, "Project Thesis: Integration of a Network Stack on a Nano-Satellite Payload," 2018.

[13] H. D. Ledet-Pedersen J., Christiansen J.D.C, "Cubesat Space Protocol." `https://github.com/GomSpace/libcsp`, 2018.

[14] E. R. Jahren, "Design and implementation of a reliable transport layer protocol for nuts," Master's thesis, NTNU, 2015.

[15] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium*, pp. 21–34, June 2007.

[16] K. Sollins, "Rcc 1350 - the tftp protocol (revision 2)." https://tools.ietf.org/html/rfc1350, 1992.

[17] J. Postel, "Rcc 959 - file transfer protocol." https://tools.ietf.org/html/rfc959, 1985.

[18] "KubOS." `https://www.kubos.com/kubos/`, 2017.

[19] The Consultative Committee for Sapce Data Systems, *CCSDS File Delivery Protocol (CCSDS 727.0-B-4)*, 1 2007.

[20] D. L. Magne Hov, "HYPSO Software - Git Repository." `https://github.com/NTNU-SmallSat-Lab/hypso-sw/`, 2019.

[21] HYPSO Project Team, "Requirements HSI SmallSat," tech. rep., NTNU SmallSat Lab, 2019.

[22] HYPSO Project Team, "SDR System Design Report," tech. rep., NTNU SmallSat Lab, 2019.

[23] Liping Di, Ben Kobler, "NASA Standards for Earth Remote Sensing Data," in *International Archives of Photogrammetry and Remote Sensing*, vol. XXXIII, pp. 147–155, 2000.

[24] Avnet, *PicoZed 7Z015 / 7Z030 SOM - Hardware User guide*, 2018.

[25] NanoAvionics, *EPS Electrical Power System, Data Sheet, NA-EPS-G0-R0*, 2018.

[26] H. Leppinen, "Current use of linux in spacecraft flight software," in *IEEE Aerospace and Electronic Systems Magazine*, vol. 32, pp. 4–13, 2017.

[27] Xilinx, Inc., *PetaLinux Tools Documentation Reference Guide, UG1144*, 2018.2 ed., June 2018.

[28] Xilinx, Inc., *Zynq-7000 SoC Technical Reference Manual, UG585*, 1.12.2 ed., July 2018.

[29] Alén Space, *TOTEM Nanosatellite SDR Platform - Motherboard UHF Front end.*

[30] H. Leppinen, P. Niemela, N. Silva, H. Sanmark, H. Forsten, A. Yanes, R. Modrzewski, A. Kestila, and J. Praks, "Developing a Linux-based nanosatellite on-board computer: Flight results from the Aalto-1 mission," in *IEEE Aerospace and Electronic Systems Magazine*, pp. 4–14, 2019.

[31] Kitware, Inc, *CTest Command-Line Reference*, 3.14.3 ed., 2019.

[32] Grøtte, Mariusz et.al, "HYPSO Mission Budgets," tech. rep., NTNU SmallSat Lab, 2018.

[33] Real Time Engineers Ltd., "The FreeRTOS™ Kernel." `https://freertos.org/RTOS.html`, 2017.

[34] NanoAvionics, *SSTR6U 6U Structure, Data Sheet, NA-SSTR6U-G0-R0*, 2018.

[35] NanoAvionics, *Option Sheet, EPS Electrical Power System, NA-OS-EPS-R0*, 2018.

[36] NanoAvionics, *SatBus 3C2, Data Sheet, NA-OS-EPS-R0*, 2018.

Appendix A

M6P Satellite Platform

This appendix contains details about the M6P satellite platform subsystems.

A.1 Mechanical Frame

The satellite frame is compliant with the 6U CubeSat Design Specification, with only a few differences from the reference design [34]. The frame is geometrically constructed from a single rectangular cuboid and two cylinders protruding from one of the smaller faces, see figure A.1.

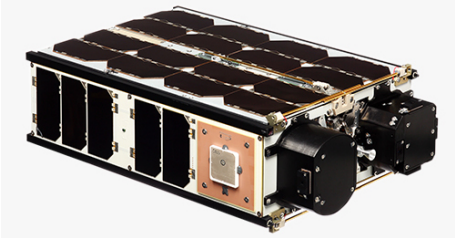


Figure A.1: M6P satellite bus mechanical frame and solar panels. The two cylindrical protrusions are nicknamed *tuna cans* by NanoAvionics.

The satellite frame holds all the subsystems together. It provides mechanical stability that is important to the survivability of the satellite during launch. The lack of thermal convection in space means that thermal connections to the frame are important in avoiding overheating problems. The payload computer(s) may have to be thermally coupled to the satellite frame to avoid overheating.

The frame also provide a mechanical interface for connecting the payload components. The components can be fixed to the frame using a grid of countersunk holes. The grid is organised to be compatibly with PC/104 PCB stacks, which will be useful in the design of the payload computer hardware. In terms of volume, approximately 4 CubeSat units of space are available for payload integration.

Each face of the satellite frame is covered by a solar panel carrying triple junction solar cells from AzurSpace. These cells are rated to an efficiency of 29.3 %. The maximum power output from a single panel (only one panel is illuminated at the optimal attitude) is rated to 19.4 W in LEO.

A.2 Electrical Power System

In many ways, the EPS is the most critical subsystem in the satellite, if it fails all other subsystems fail as well. The EPS is responsible for providing electrical power to the remaining subsystems, and for storing the electrical power supplied by the solar panels in lithium-ion batteries.

Many of the EPS features are managed via software by the EPS computer.

The EPS design can be organised into three sections: input, output, and storage [25].

The input section is primarily concerned with loading the solar panel outputs in the most efficient manner. Because of an internal impedance in the solar panel, there exists a certain load impedance that will extract the maximum amount of power from the solar panel. The input section achieves this by utilising Maximum Power Point Tracking (MPPT) circuits that continuously adjust their impedance while loading the solar panels. The MPPT chips also house converter circuits that regulate the output to a target voltage that is configurable by software. The six solar panels are distributed between four MPPT converter modules. Two panels can connected in parallel to the input of each MPPT converter. This allows panels that are placed on opposite faces of the satellite to use the same MPPT circuit, since only one will be sufficiently illuminated at any time. Ideal diodes are placed in series with each solar panel to avoid reverse currents between parallel panels.

The output section regulate a battery voltage to stable operating voltages that the subsystems can use. Four buck-boost converter circuits are employed to supply output channels with four distinct operating voltages. Two of the output converters are fixed at 3.3 V and 5.0 V, while the last two can be configured to any voltage in the range 3.0 V to 18 V by selecting hardware components. The regulated supplies can be individually routed to any of the ten output channels. There are two additional channels that provide *always on* 3.3 V and 5.0 V.

Each output channel has two types of overcurrent protection. One is hardware based and is typically triggered at 3.12 A, but can be customised to any value in the range 0 mA to 3130 mA by changing hardware components [35]. The other type is software enabled and run-time configurable, and switches the channel off when the current exceeds a specified value in the range 1 mA to 3000 mA. Additionally, the output converters (that feed the channels) perform internal current limiting.

The storage section implements the battery itself, and additional protection mechanisms to maintain the health of the lithium-ion battery and the EPS. Further overcurrent protection monitors the battery current and disconnects the battery completely if it detects currents that exceed the safe operating condition of the battery.

Undervoltage protection is implemented as different operating modes that are activated by a state machine in the EPS computer. At decreasing battery voltage levels, a decreasing number of output channels and features are enabled. The voltage threshold for each mode, and the organisation of output channels are configurable via software. Only the lowest and most critical mode is hardware triggered and will disconnect the battery even from the EPS computer. At this point only the MPPT outputs are connected to the battery. A smaller external circuit will then periodically monitor the battery voltage and reconnect the EPS computer when it reaches a safe voltage level. The mode transitions are illustrated in figure A.2.

Overvoltage protection ensures that the battery is never charged past its maximum capacity. The overvoltage mechanism is implemented in the MPPT converter circuits, which will adjust their impedance to change their output. When the battery reaches 8.33 V the MPPT converters shift their power points downwards until the battery voltage is perceived as stable.

There is also an external charging port which is connected directly to the battery assembly and bypasses the overvoltage protection of the MPPT circuits. Care must be taken to only use compatible chargers when using this port.

Extra features include a watchdog timer that is only reset when receiving CSP packets from

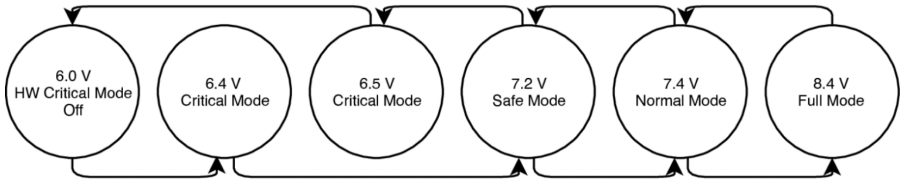


Figure A.2: EPS operating modes .Each mode has a software configurable vector describing which output channels will be turned on .Transitions are based on battery voltage. Illustration taken from [25].

a ground station. Upon expiration, the EPS disconnects the battery from the output converters and even itself, long enough to let all systems power completely down. This Watchdog Timer (WDT) is especially useful when testing new radio configurations. In the case that the satellite enters a state where communication can not be established, the dedicated WDT will power cycle the entire satellite to restore it to a default configuration.

The EPS is the only subsystem that has circuits that are powered even during launch, a period when all subsystems are technically require to be *off* [1]. One of the always-on circuits is an RTC.

After the satellite has been deployed from the its launch vehicle, it must refrain from deploying antennas for a duration of 30 minutes, and it must keep all radios silent for a duration of 45 minutes. The EPS performs the timekeeping necessary to fulfil these requirements, and switches on power to the appropriate subsystems when the deployment period has expired.

A.3 Flight computer

The flight computer subsystem hosts multiple modules: an OBC, an ADCS and a set of redundant UHF radio modules (COMM) [36]. The OBC and the ADCS are implemented one the same physical computer, meaning that they share computing resources. Additionally, the FC directly connects to an external GPS module.

A.3.1 On-board Computer

The responsibilities of the OBC are few; most tasks are performed by dedicated modules and subsystems. However, it is able to collect telemetry about the FC and other subsystems, it can run uploaded script files, and it can store data for other subsystems. It can also receive firmware updates while in orbit, providing capabilities for bug fixes and software upgrades to the ADCS. Thus, it can also be reprogrammed to take over computational tasks from failing subsystems.

A.3.2 Attitude Determination and Control System

The ADCS software module collects sensor data which is used to estimate the attitude of the satellite, and can run a variety of control modes to manipulate the satellite attitude with actuators. Sensor data from magnetometers, gyroscopes, sun sensors and a star tracker are fused in a kalman filter to produce the attitude estimate.

Attitude control is achieved by using two redundant actuators. A set of reaction wheels are able to produce a torque by accelerating the spin of a mass. The reaction torque of the accelerated spinning mass acts on the satellite, producing an angular acceleration. Three of the reaction wheels are oriented orthogonally relative to each other, such that a combination of the

three are able to produce an angular acceleration in any direction. Redudancy is achieved by installing a fourth reaction wheel at an angle relative to the three orthogonal reaction wheels. When combining the slanted backup wheel with any two of the orthogonal wheels, an arbitrary angular acceleration can still be achieved.

The ADCS software allows the satellite to be put into one of several mission control mode:

- Velocity direction pointing mode: Pointing in the direction of flight.
- Nadir mode: satellite points to nadir (towards centre of the earth).
- Sun maximum power tracking: Keeps one of the largest solar panels pointing towards the sun.
- Earth target tracking: Keeps pointing to a geographical point on earth.
- No mode: the ADCS system performs no control.

The satellite tumble rate is continuously monitored by the ADCS. If the satellite is found to have an angular velocity that exceeds a configurable threshold, the ADCS automatically transitions to a detumbling mode. In this mode, the ADCS computes (with B-dot algorithm) and applies a control vector that counteracts the tumbling motion. This action is typically performed until the unwanted angular velocity has been eliminated. In this mode, only the magnetometers are used to estimate the tumble rate, and only the magnetorquers are used to create the counter torque.

A.3.3 GPS Module

A GPS module provides the FC with accurate information about time and date. The GPS is connected to the FC via a serial interface and an analogue PPS signal.

When receiving a new timestamp over the serial interface, there is a communication delay rendering the received timestamp invalid by the time it can be merged with the local FC clock in software. One could attempt to correct the timestamp by adding a delay offset, but communication jitter makes it difficult to find a good estimate of the offset. Despite the error, the deviation from true time is significantly smaller than one second. The auxiliary PPS input is intended to solve this issue. The PPS signal exhibits a pulse with a rising edge that is precisely fixed to a period of 1 s, with an accuracy of 10 ns. This hardware signal is installed as an interrupt in the FC computer, and can therefore be handled with much smaller delay and jitter effects.

The typical initialisation time for the GPS module is 30 s, and it cannot provide a valid timestamp before it is initialised. The GPS module can also provide positional data and velocity data.

A.3.4 UHF Radio

The FC board can hold up to two UHF radios, where one is usually used as backup. The UHF radio establish a radio link connection to a ground station, and relays network packets between the subsystems in space and the ground station.

In the M6P configuration that the HYPSO mission will use, there is only one UHF radio present on the FC. This UHF is connected to a four monopole antennae system.

A.4 Payload Controller

The PC subsystem is intended to provide an interface that any payload can be connected to. It offers a CAN interface, 2x RS422 interfaces, 3x Inter-Integrated Circuit (I2C) interfaces, 3x SPI interfaces, and 2x Universal asynchronous Receiver-Transmitter (UART) interfaces. Any of these can be used to connect to communicate with payload systems, but only the CAN interface and the RS422 interfaces support CSP. The remaining interfaces all require custom modification to the PC software, and would have to be implemented in collaboration with NA.

The PC is running FreeRTOS.

Appendix B

Breakout Board Design Files

The included specification files have been created while defining the required interfaces of the Breakout Board.

The included design files show the final design of the Breakout Board. A few of the schematic pages that contain unimportant details have been removed to reduce page count.

PicoZed Default MIO Assignment

Function	Protocol	Zynq Interface	MIO Pins
eMMC 4GB NAND Flash	SDIO	SD1	0, 10-15
128Mb NOR Flash	QSPI	QSPI	1-6, 8
RGB Camera	USB 2.0	USB0	7, 28-39
HSI Camera	Gigabit Ethernet	Enet0	16-27, 52-53

Breakout Board Specific MIO Assignment

Function	Protocol	Zynq Interface	MIO Pins
SD Card Storage	SDIO	SD0	40-45, 46
UART Terminal	UART	UART1	48-49
Can-bus	CAN	CAN0	50-51
PPS, HSI Flash	GPIO	N/A	9, 47

SD Card [SD0] Pinout

Function	MIO Pin
clk	40
cmd	41
data[0]	42
data[1]	43
data[2]	44
data[3]	45
card detect	46

UART [UART1] Pinout

Function	MIO Pin
tx	48
rx	49

CAN [CAN0] Pinout

Function	MIO Pin
rx	50
tx	51

Various Pinout

Function	MIO Pin
PPS	9
HSI Flash	47

Connectors

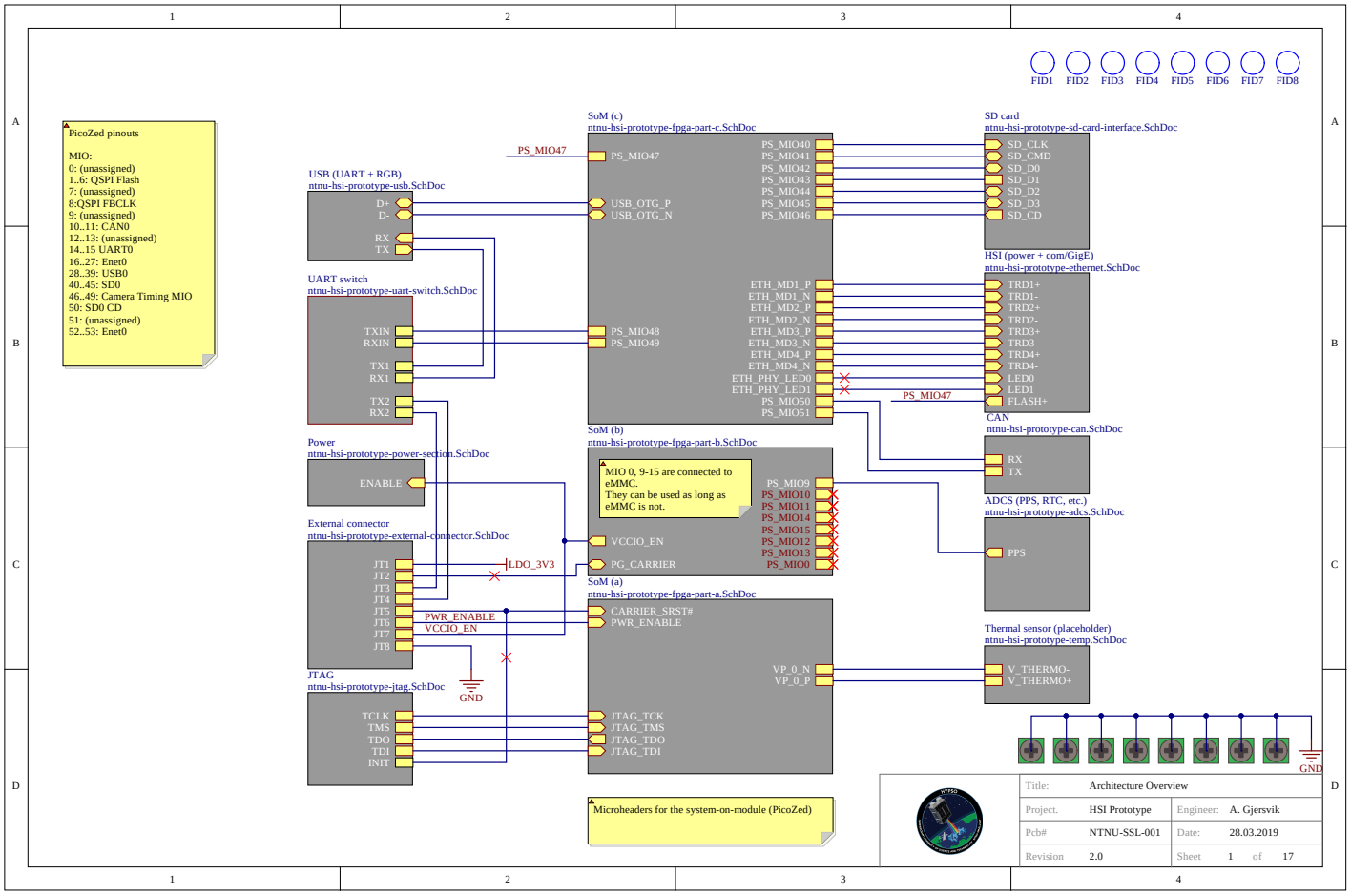
Hirose HSI Connector: **HR25-7TR-8PA**

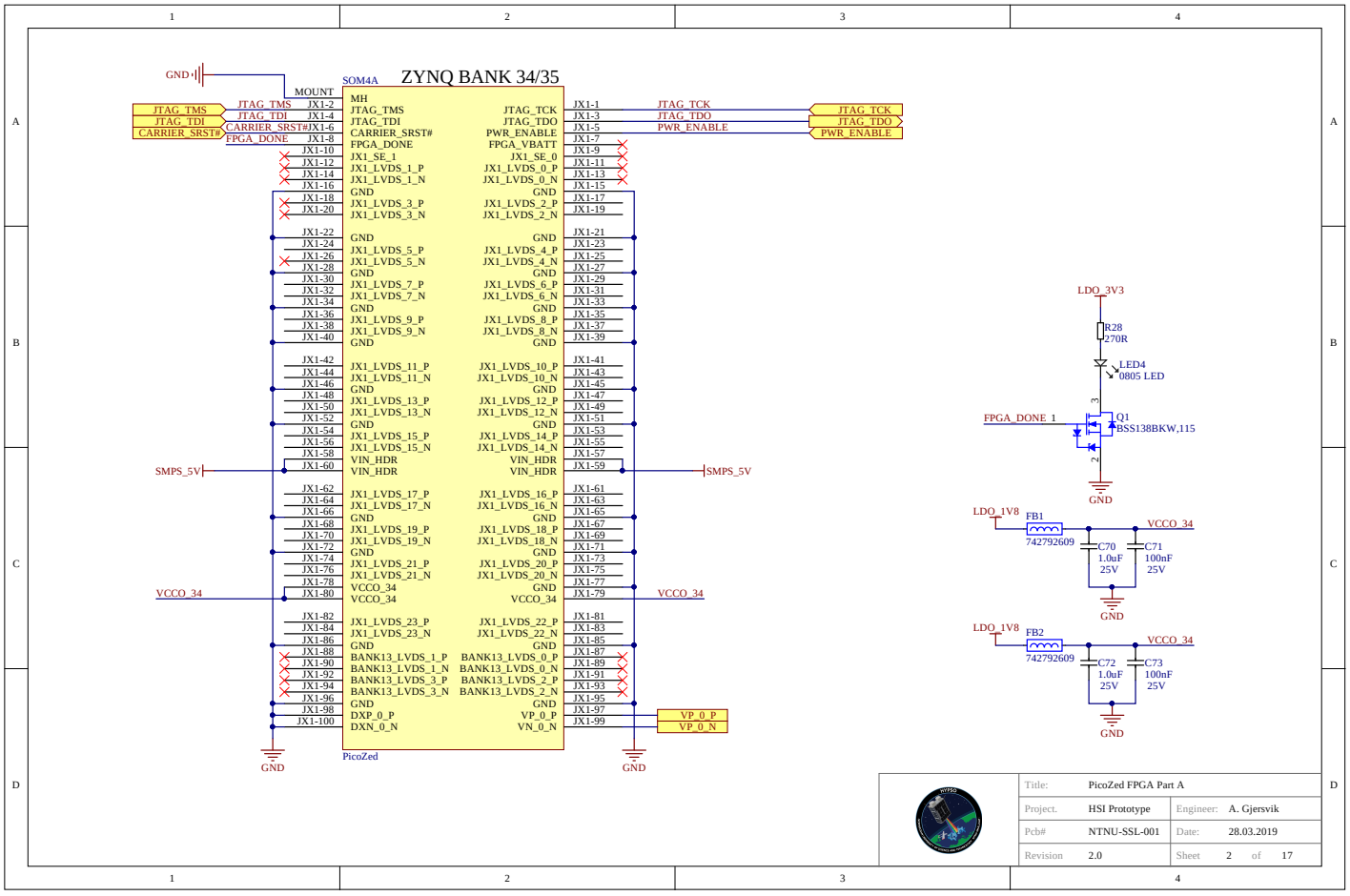
Function	Connector Pin	Connects to (all to a header?)
GND	1	GND
Flash optocoupled output negative (-)	2	GND
GPIO 1	3	
Trigger input with optocoupler (-)	4	
Flash optocoupled output positive (+)	5	MIO 47, 1V8 pull up
GPIO2	6	
Trigger input with optocoupler (+)	7	
Input power supply (VCC) 12-24 V DC	8	Power from EPS or lab,

Gigabit Ethernet RJ-45: ***L829-1J1T-43***

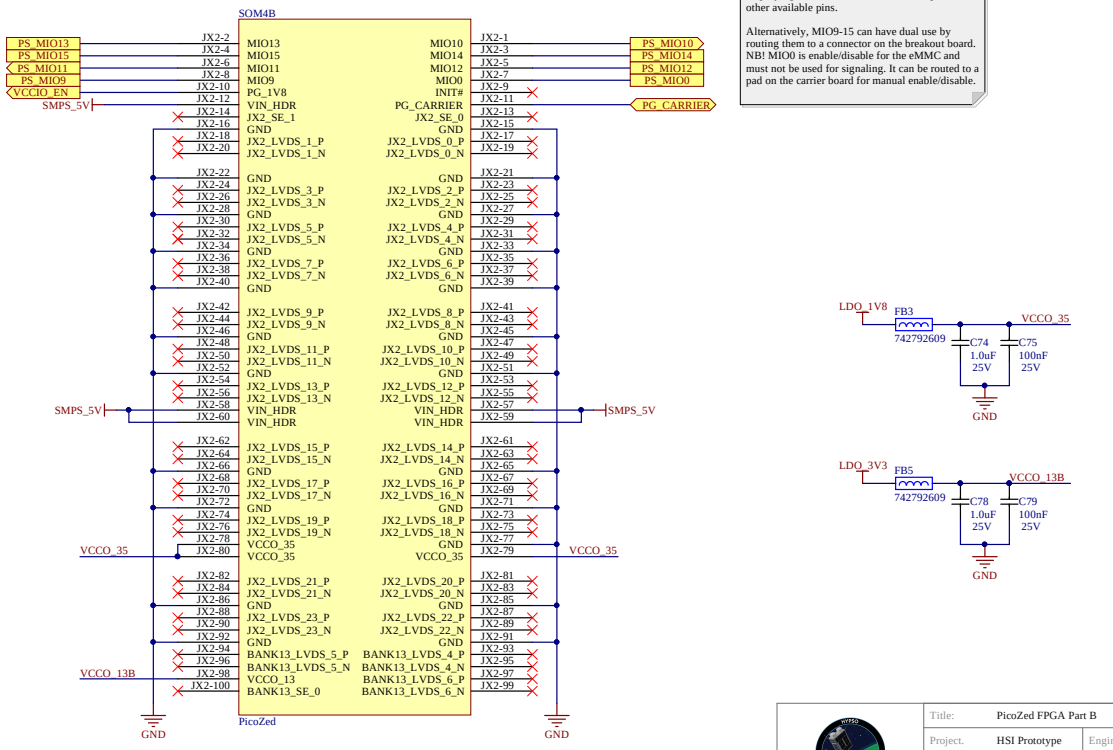
Function	Connector Pin	Connects to	
TRCT3	1	0.1uF in series to GND	
TRD3-	2	ETH_MD3_N	
TRD3+	3	ETH_MD3_P	
TRD2+	4	ETH_MD2_P	
TRD2-	5	ETH_MD2_N	
TRCT2	6	0.1uF in series to GND	
TRCT4	7	0.1uF in series to GND	
TRD4+	8	ETH_MD4_P	
TRD4-	9	ETH_MD4_N	
TRD1-	10	ETH_MD1_N	
TRD1+	11	ETH_MD1_P	
TRCT1	12	0.1uF in series to GND	
LED1 Yellow Negative	13	ETH_PHY_LED0	
LED1 Yellow Positive	14	3V3	
LED2 Orange Negative	15	N/C	
LED2 Common Positive	16	3V3	
LED2 Green Negative	17	ETH_PHY_LED1	

Parts	Part name	Part number (distributor)	Description	Quantity
Can bus tranceiver	MCP2562-E/SN	2362839	physical transceiver	1
USB protection	IP4220CZ6	1506629	Dual USB 2.0 integrated ESD protection	2
UART to USB adapter	FTDI230XQ-R	2081324	uart to usb converter chip	1
UART Mux	TS3A24159	296-21914-1-ND	Analogue Switch IC VSSOP-10	1
SD Card controller	TXS02612RTWR	2335617	Frequency Synthesizer, 60MHz, 1.1V to 3.6V, WQFN-24	1
LDO power regulator 1V8 3V3	TPS74801DRCR	2764833	Single-output 1.5A LDO regulator, adjustable (0.8V to 3.6V), programmable soft start	2
LDO power regulator 1V0 1V2	TPS74701DRCR	2764747	Single-output 500mA LDO regulator, adjustable (0.8V to 3.6V), programmable soft start	2
Switching power regulator 5V	LMZ31506HRUQT	2373558	DC/DC POL Converter, Adjustable, Buck, 780 kHz, 1.2 V to 5.5 V out, 6 A, QFN-47	1
FPGA-done LED mosfet	BSS138BKW,115	2053836	MOSFET Transistor, N Channel, 320 mA, 60 V, 1 ohm, 10 V, 1.1 V	1
Connectors				
USB RGB, USB C	105450-0101	2524076	USB Type C, USB 3.1, Receptacle, 24 Ways, Surface Mount, Right Angle	1
UART adapter, USB C	105450-0101	2524076	USB Type C, USB 3.1, Receptacle, 24 Ways, Surface Mount, Right Angle	1
CAN bus, PicoLock 4 pin	504050-0491	2392592	Mates with 504051-0401	1
Power EPS VBAT, PicoLock 10 pin	504050-1091	WM10138CT-ND	Mates with 504051-1001	1
Power EPS HSI, PicoLock 4 pin	504050-0491	2392592	Mates with 504051-0401	1
Lab power, barrel jack	FC681478	2450496	DC Power Connector, Jack, 2 A, 2.1 mm, Through Hole Mount, Through Hole	1
Power synthesis source manual short selector			3 pin 100 mil	1
HSI Power source manual short selector			3 pin 100 mil	1
XADC for Temperature			2 pin 100 mil	1
UART Mux manual short select			3 pin 100 mil	1
PPS Signal			2 pin 100 mil	1
HSI 8-pin external connector			8 pin 100 mil	1
HSI circular connector	HR25-7TR-8PA(73)	2929840	Mates with HR25-7TP-8S(72), HR25 Series, Bulkhead Receptacle, 8 Contacts, PCB Pin	1
HSI, Gig ETH, RJ45	L829-1J1T-43	2675997	Modular Connector, RJ45 Jack, 1 x 1 Port, 8P8C, Cat5e, Through Hole Mount	1
SD Card slot	502570-0893	2060731	Memory Socket, 502570 Series, Micro SD, 8 Contacts, Copper Alloy, Gold Plated Contacts	1
JTAG	TBD	TBD		
External connector, various functions	53261-0871	2119496	WIRE-BOARD CONNECTOR HEADER 8POS, 1.25MM	1


FID1 FID2 FID3 FID4 FID5 FID6 FID7 FID8
PicoZed pinouts
MIO:
0: (unassigned)
1..6: QSPI Flash
7: (unassigned)
8:QSPI FBCLK
9: (unassigned)
10..11: CAN0
12..13: (unassigned)
14..15 UART0
16..27: Enet0
28..39: USB0
40..45: SD0
46..49: Camera Timing MIO
50: SD0 CD
51: (unassigned)
52..53: Enet0
SoM (c)
ntnu-hsi-prototype-fpga-part-c.SchDoc
PS_MIO47
PS_MIO40
PS_MIO41
PS_MIO42
PS_MIO43
PS_MIO44
PS_MIO45
PS_MIO46
USB_OTG_P
USB_OTG_N
USB (UART + RGB)
ntnu-hsi-prototype-usb.SchDoc
D+
D-
RX
TX
SD card
ntnu-hsi-prototype-sd-card-interface.SchDoc
SD_CLK
SD_CMD
SD_D0
SD_D1
SD_D2
SD_D3
SD_CD
HSI (power + com/GigE)
ntnu-hsi-prototype-ethernet.SchDoc
ETH_MD1_P
ETH_MD1_N
ETH_MD2_P
ETH_MD2_N
ETH_MD3_P
ETH_MD3_N
ETH_MD4_P
ETH_MD4_N
ETH_PHY_LED0
ETH_PHY_LED1
PS_MIO50
PS_MIO51
TRD1+
TRD1-
TRD2+
TRD2-
TRD3+
TRD3-
TRD4+
TRD4-
LED0
LED1
FLASH+
UART switch
ntnu-hsi-prototype-uart-switch.SchDoc
TXIN
RXIN
TX1
RX1
TX2
RX2
PS_MIO48
PS_MIO49
CAN
ntnu-hsi-prototype-can.SchDoc
RX
TX
Power
ntnu-hsi-prototype-power-section.SchDoc
ENABLE
SoM (b)
ntnu-hsi-prototype-fpga-part-b.SchDoc
MIO 0, 9-15 are connected to eMMC.
They can be used as long as eMMC is not.
PS_MIO9
PS_MIO10
PS_MIO11
PS_MIO14
PS_MIO15
PS_MIO12
PS_MIO13
PS_MIO00
VCCIO_EN
PG_CARRIER
ADCS (PPS, RTC, etc.)
ntnu-hsi-prototype-adcs.SchDoc
PPS
External connector
ntnu-hsi-prototype-external-connector.SchDoc
JT1
JT2
JT3
JT4
JT5
JT6
JT7
JT8
LDO_3V3
PWR_ENABLE
VCCIO_EN
SoM (a)
ntnu-hsi-prototype-fpga-part-a.SchDoc
CARRIER_SRST#
PWR_ENABLE
Thermal sensor (placeholder)
ntnu-hsi-prototype-temp.SchDoc
VP_0_N
VP_0_P
V_THERMO-
V_THERMO+
JTAG
ntnu-hsi-prototype-jtag.SchDoc
TCLK
TMS
TDO
TDI
INIT
JTAG_TCK
JTAG_TMS
JTAG_TDO
JTAG_TDI
GND
Microheaders for the system-on-module (PicoZed)
Title: Architecture Overview
Project. HSI Prototype
Engineer: A. Gjersvik
Pcb# NTNU-SSL-001
Date: 28.03.2019
Revision 2.0
Sheet 1 of 17


SOM4A
ZYNQ BANK 34/35
MOUNT
MH
JTAG_TMS
JTAG_TDI
CARRIER_SRST#
FPGA_DONE
JTAG_TCK
PWR_ENABLE
FPGA_VBATT
JX1_SE_1
JX1_SE_0
JX1_LVDS_1_P
JX1_LVDS_1_N
JX1_LVDS_0_P
JX1_LVDS_0_N
JX1_LVDS_3_P
JX1_LVDS_3_N
JX1_LVDS_2_P
JX1_LVDS_2_N
JX1_LVDS_5_P
JX1_LVDS_5_N
JX1_LVDS_4_P
JX1_LVDS_4_N
JX1_LVDS_7_P
JX1_LVDS_7_N
JX1_LVDS_6_P
JX1_LVDS_6_N
JX1_LVDS_9_P
JX1_LVDS_9_N
JX1_LVDS_8_P
JX1_LVDS_8_N
JX1_LVDS_11_P
JX1_LVDS_11_N
JX1_LVDS_10_P
JX1_LVDS_10_N
JX1_LVDS_13_P
JX1_LVDS_13_N
JX1_LVDS_12_P
JX1_LVDS_12_N
JX1_LVDS_15_P
JX1_LVDS_15_N
JX1_LVDS_14_P
JX1_LVDS_14_N
VIN_HDR
JX1_LVDS_17_P
JX1_LVDS_17_N
JX1_LVDS_16_P
JX1_LVDS_16_N
JX1_LVDS_19_P
JX1_LVDS_19_N
JX1_LVDS_18_P
JX1_LVDS_18_N
JX1_LVDS_21_P
JX1_LVDS_21_N
JX1_LVDS_20_P
JX1_LVDS_20_N
VCCO_34
JX1_LVDS_23_P
JX1_LVDS_23_N
JX1_LVDS_22_P
JX1_LVDS_22_N
BANK13_LVDS_1_P
BANK13_LVDS_1_N
BANK13_LVDS_0_P
BANK13_LVDS_0_N
BANK13_LVDS_3_P
BANK13_LVDS_3_N
BANK13_LVDS_2_P
BANK13_LVDS_2_N
DXP_0_P
DXN_0_N
VP_0_P
VN_0_N
PicoZed
SMPS_5V
VCCO_34
VP_0_P
VP_0_N
GND
LDO_3V3
R28
270R
LED4
0805 LED
FPGA_DONE_1
Q1
BSS138BKW,115
LDO_1V8
FB1
742792609
C70
1.0uF
25V
C71
100nF
25V
FB2
C72
C73
Title: PicoZed FPGA Part A
Project. HSI Prototype
Engineer: A. Gjersvik
Pcb# NTNU-SSL-001
Date: 28.03.2019
Revision 2.0
Sheet 2 of 17

ZYNQ BANK 34/35

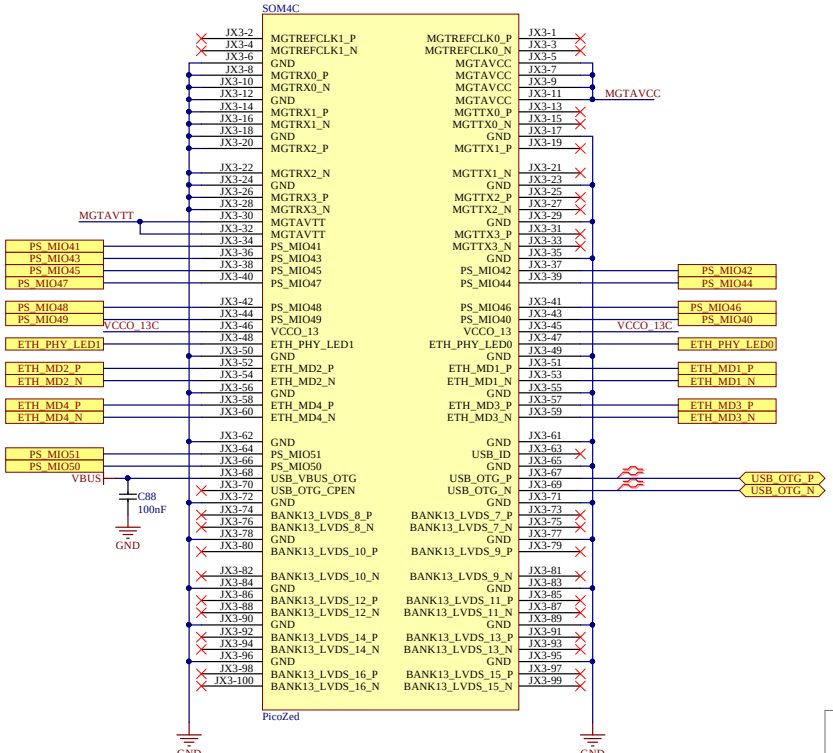


Pins MIO9 through MIO15 (inclusive) and MIO0 must be kept free so the onboard eMMC on the PicoZED can be utilized. All signals currently employing MIO9-15 and MIO0 must be place on other available pins.

Alternatively, MIO9-15 can have dual use by routing them to a connector on the breakout board. NB! MIO0 is enable/disable for the eMMC and must not be used for signaling. It can be routed to a pad on the carrier board for manual enable/disable.

Title:	PicoZed FPGA Part B		
Project.	HSI Prototype	Engineer:	A. Gjersvik
Pcb#	NTNU-SSL-001	Date:	28.03.2019
Revision	2.0	Sheet	3 of 17

ZYNQ BANK 13, 501

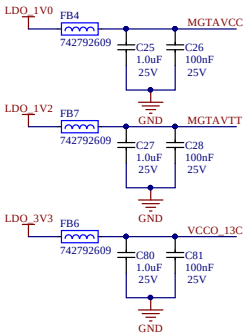


USB Connector should be connected to:

- USB_OTG_N
- USB_OTG_P
- USB_ID
- USB_OTG_CPEN

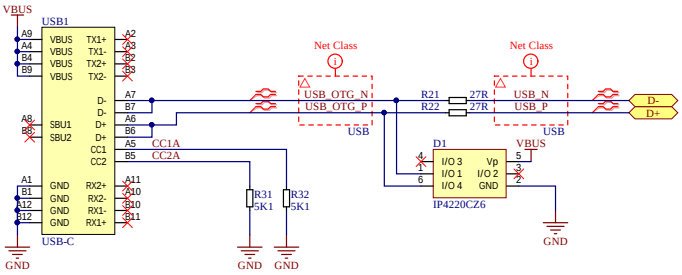
USB_OTG_CPEN signal allows the user to control an external power source for USB VBUS on the carrier board and is only used when USB operates in host mode.

JX3 connector used for microSD Card, UART, USB2.0 and bank 13 PL I/0



Title:	PicoZed FPGA part C		
Project.	HSI Prototype	Engineer:	A. Gjersvik
Pcb#	NTNU-SSL-001	Date:	28.03.2019
Revision	2.0	Sheet	4 of 17

USB OTG Device



UART is connected to UART1 MIO48(tx)-MIO49(rx).

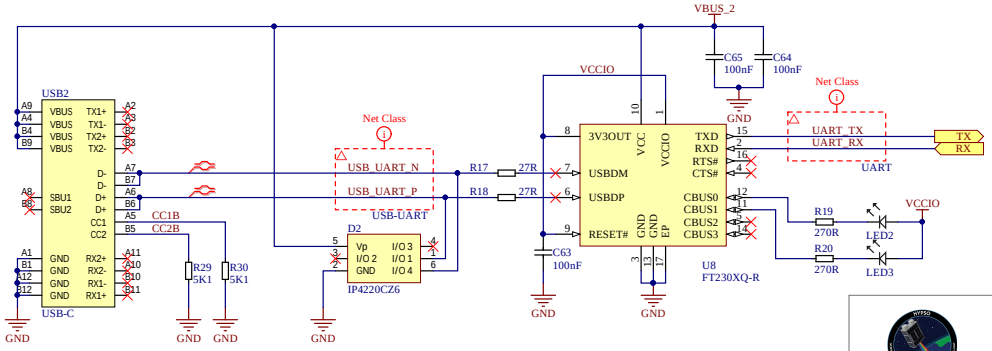
The USB device is setup to act like a USB Device. See Chapter 15 of the Zynq7000 for USB implementation details.

CPEN is not used (used to control external pwe supplies in Host mode). ID is left floating according to USB-B device standard.

Add TUSB320 for USB-C CC role detection etc.

IN UART C desing use two 5.1K pull down resistors on CC1 and CC2

USB UART BRIDGE

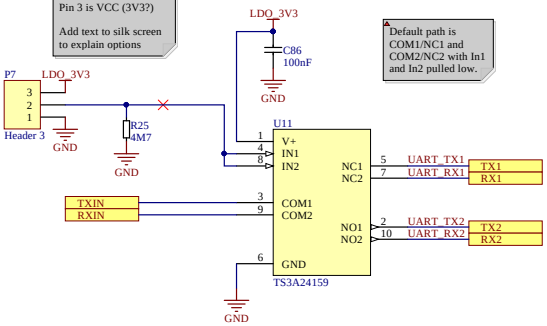


Title:	USB Manager		
Project.	HSI Prototype	Engineer:	A. Gjersvik
Pcb#	NTNU-SSL-001	Date:	28.03.2019
Revision	2.0	Sheet	5 of 17

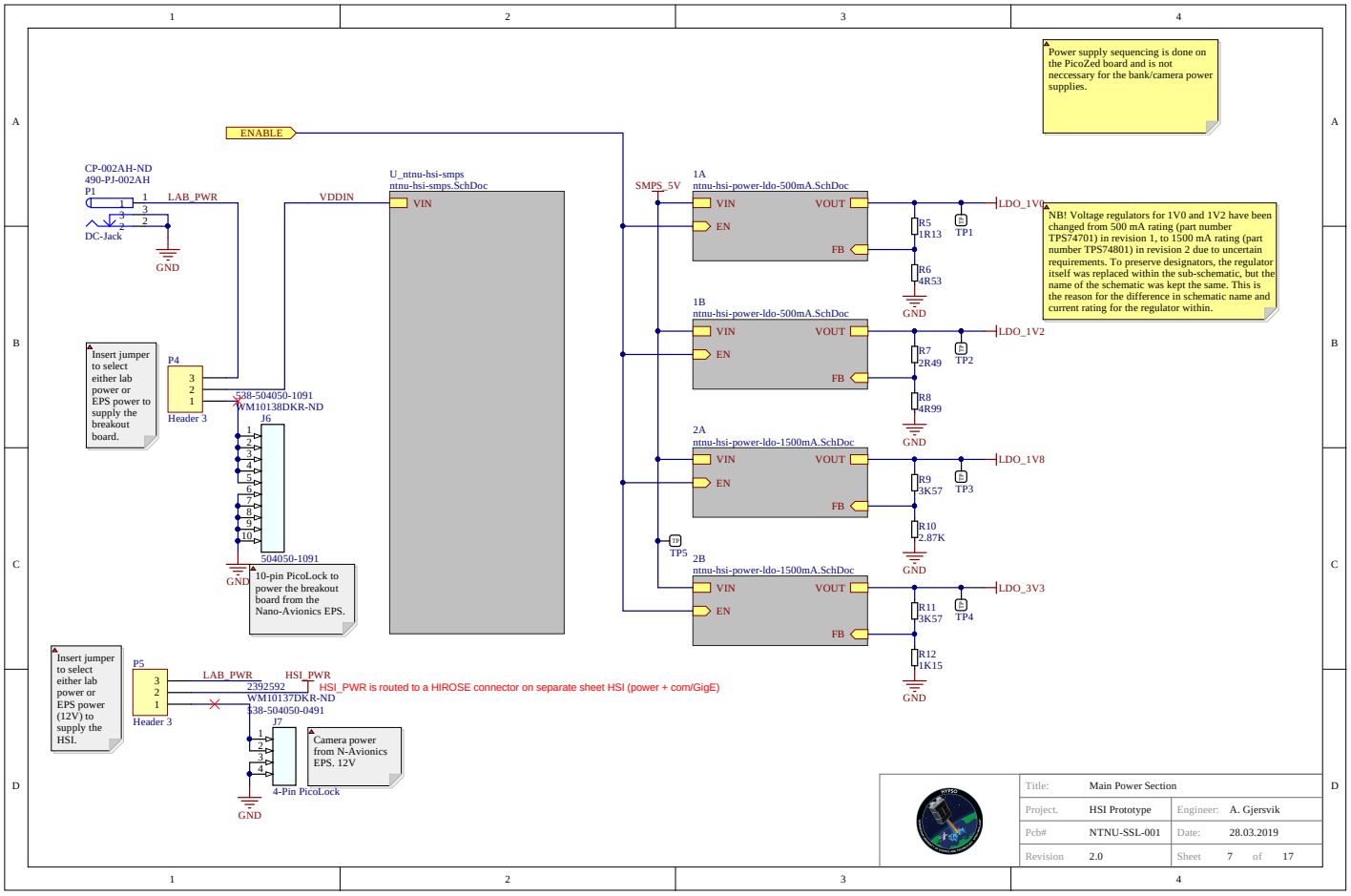
Manual selector short circuit:
Center (Pin 2) is common
Pin 1 is GND
Pin 3 is VCC (3V3?)

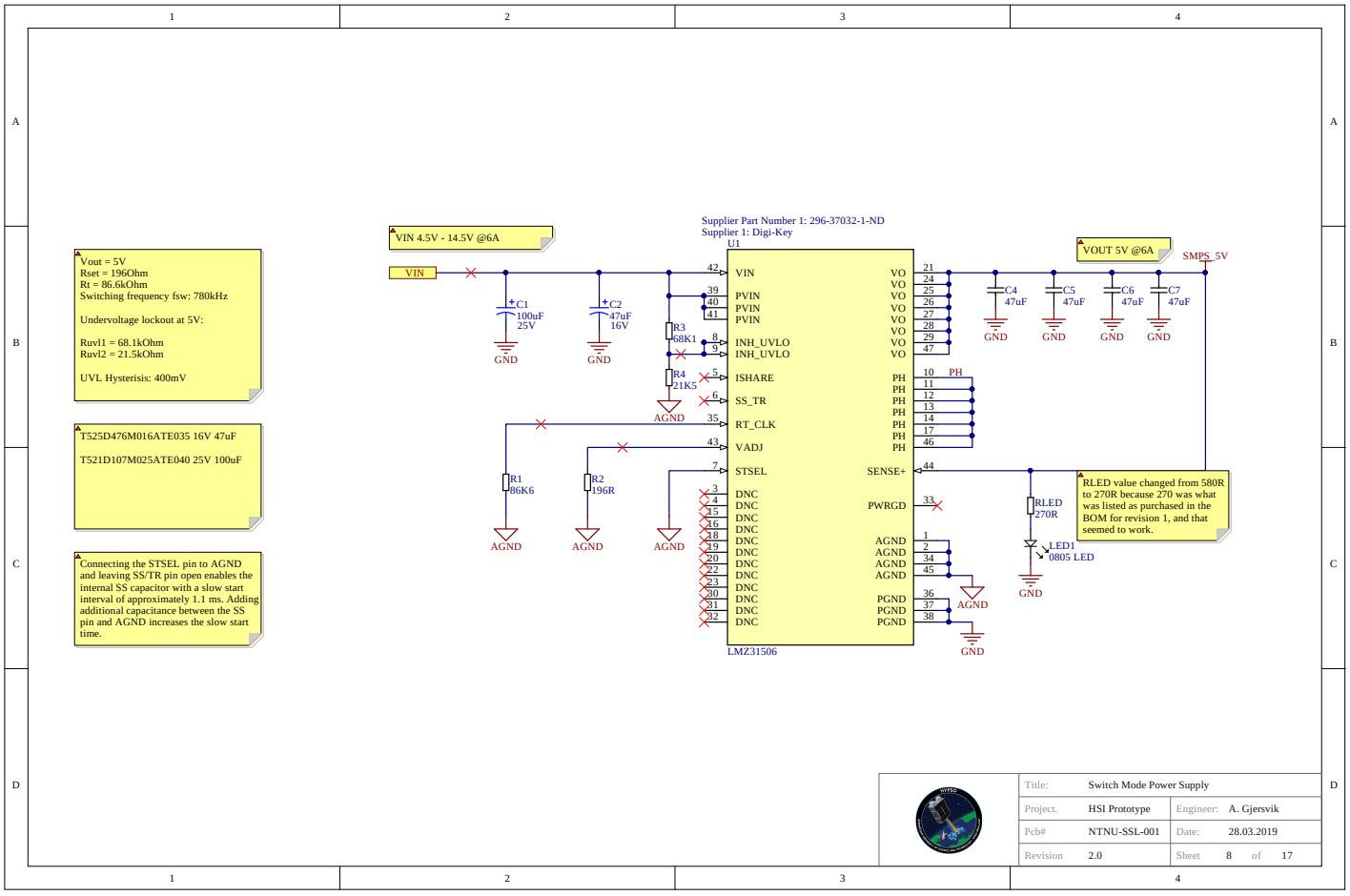
Add text to silk screen to explain options

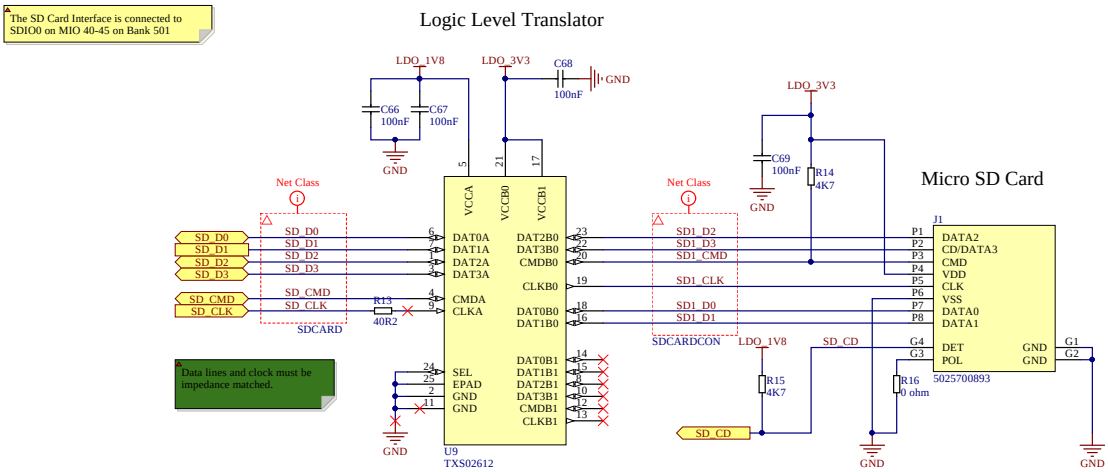
Default path is COM1/NC1 and COM2/NC2 with In1 and In2 pulled low.



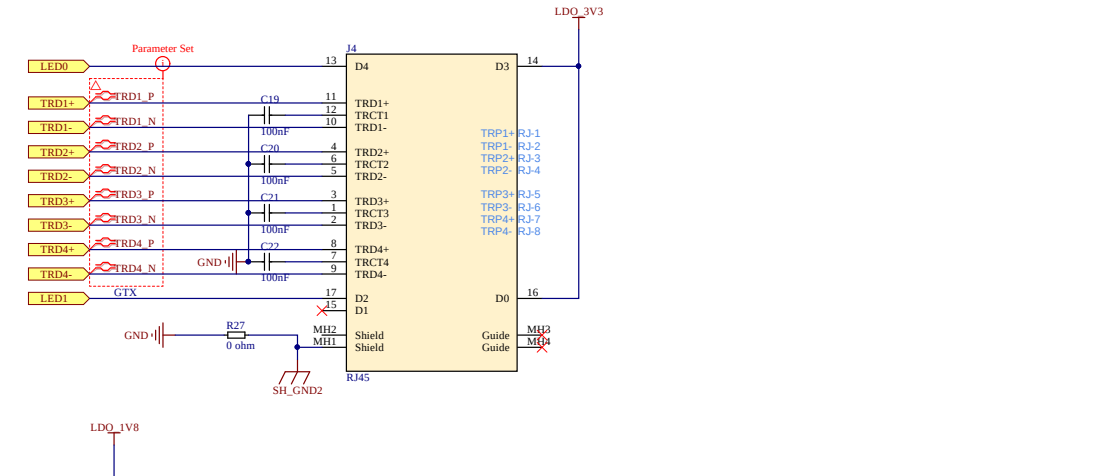
Title:	Digital UART Switch		
Project.	HSI Prototype	Engineer:	A. Gjersvik
Pcb#	NTNU-SSL-001	Date:	28.03.2019
Revision	2.0	Sheet	6 of 17


Power supply sequencing is done on the PicoZed board and is not neccessary for the bank/camera power supplies.
ENABLE
CP-002AH-ND
490-PJ-002AH
P1
DC-Jack
LAB_PWR
GND
VDDIN
U_ntnu-hsi-smps
ntnu-hsi-smps.SchDoc
VIN
SMPS_5V
1A
ntnu-hsi-power-ldo-500mA.SchDoc
VIN
VOUT
EN
FB
R5
1R13
R6
4R53
TP1
LDO_1V0
1B
ntnu-hsi-power-ldo-500mA.SchDoc
R7
2R49
R8
4R99
TP2
LDO_1V2
2A
ntnu-hsi-power-ldo-1500mA.SchDoc
R9
3K57
R10
2.87K
TP3
LDO_1V8
TP5
2B
ntnu-hsi-power-ldo-1500mA.SchDoc
R11
3K57
R12
1K15
TP4
LDO_3V3
NB! Voltage regulators for 1V0 and 1V2 have been changed from 500 mA rating (part number TPS74701) in revision 1, to 1500 mA rating (part number TPS74801) in revision 2 due to uncertain requirements. To preserve designators, the regulator itself was replaced within the sub-schematic, but the name of the schematic was kept the same. This is the reason for the difference in schematic name and current rating for the regulator within.
Insert jumper to select either lab power or EPS power to supply the breakout board.
P4
Header 3
538-504050-1091
WM10138DKR-ND
J6
504050-1091
10-pin PicoLock to power the breakout board from the Nano-Avionics EPS.
Insert jumper to select either lab power or EPS power (12V) to supply the HSI.
P5
Header 3
LAB_PWR
HSI_PWR
2392592
WM10137DKR-ND
538-504050-0491
J7
4-Pin PicoLock
Camera power from N-Avionics EPS. 12V
HSI_PWR is routed to a HIROSE connector on separate sheet HSI (power + com/GigE)
Title: Main Power Section
Project. HSI Prototype
Engineer: A. Gjersvik
Pcb# NTNU-SSL-001
Date: 28.03.2019
Revision 2.0
Sheet 7 of 17

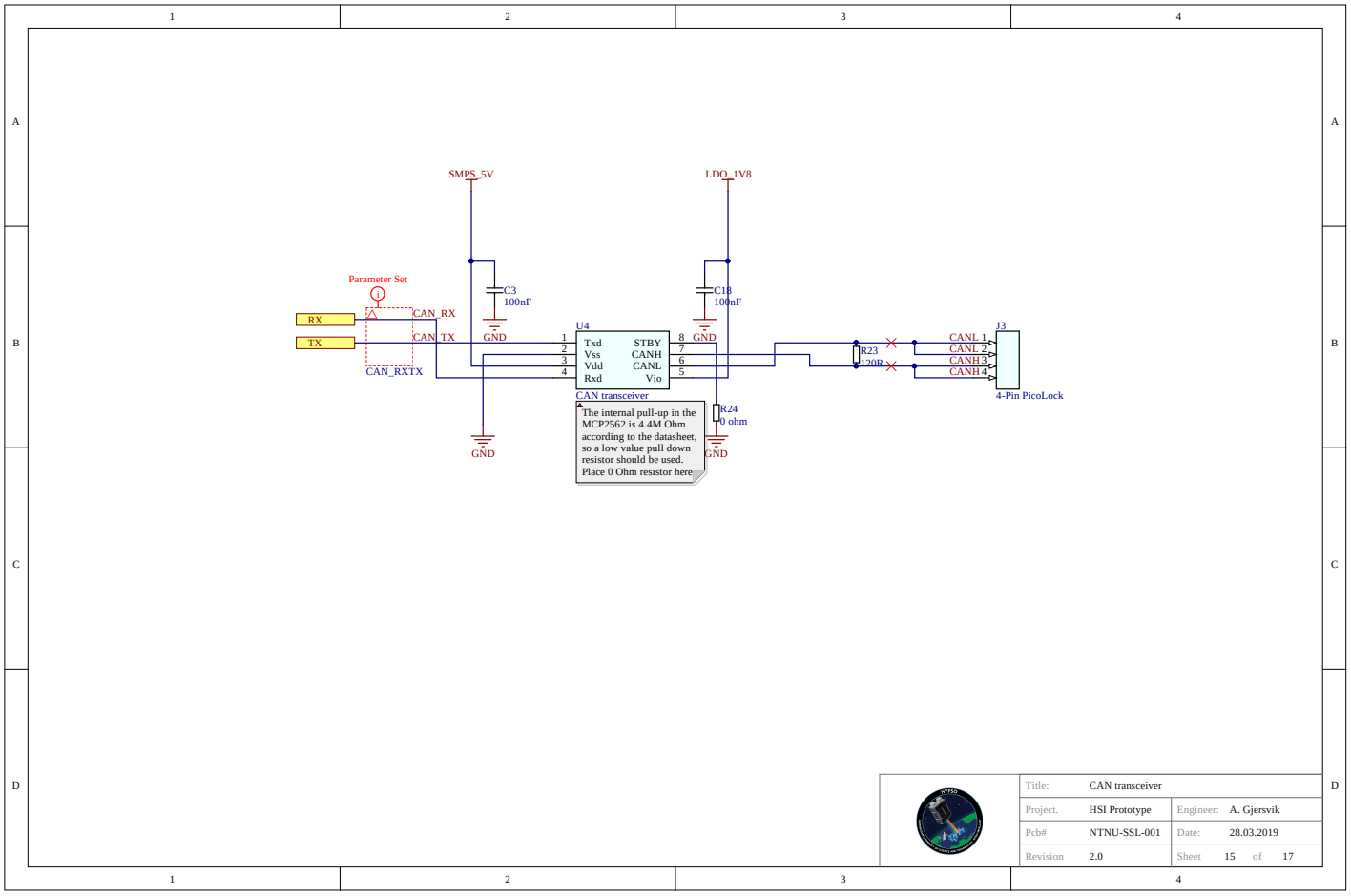

Vout = 5V
Rset = 196Ohm
Rt = 86.6kOhm
Switching frequency fsw: 780kHz
Undervoltage lockout at 5V:
Ruvl1 = 68.1kOhm
Ruvl2 = 21.5kOhm
UVL Hysterisis: 400mV
T525D476M016ATE035 16V 47uF
T521D107M025ATE040 25V 100uF
Connecting the STSEL pin to AGND and leaving SS/TR pin open enables the internal SS capacitor with a slow start interval of approximately 1.1 ms. Adding additional capacitance between the SS pin and AGND increases the slow start time.
VIN 4.5V - 14.5V @6A
VIN
C1
100uF
25V
C2
47uF
16V
R3
68K1
R4
21K5
R1
86K6
R2
196R
AGND
GND
Supplier Part Number 1: 296-37032-1-ND
Supplier 1: Digi-Key
U1
LMZ31506
VIN
PVIN
INH_UVLO
ISHARE
SS_TR
RT_CLK
VADJ
STSEL
DNC
VO
PH
SENSE+
PWRGD
PGND
VOUT 5V @6A
SMPS_5V
C4
47uF
C5
47uF
C6
47uF
C7
47uF
RLED
270R
LED1
0805 LED
RLED value changed from 580R to 270R because 270 was what was listed as purchased in the BOM for revision 1, and that seemed to work.
Title: Switch Mode Power Supply
Project. HSI Prototype
Engineer: A. Gjersvik
Pcb# NTNU-SSL-001
Date: 28.03.2019
Revision 2.0
Sheet 8 of 17

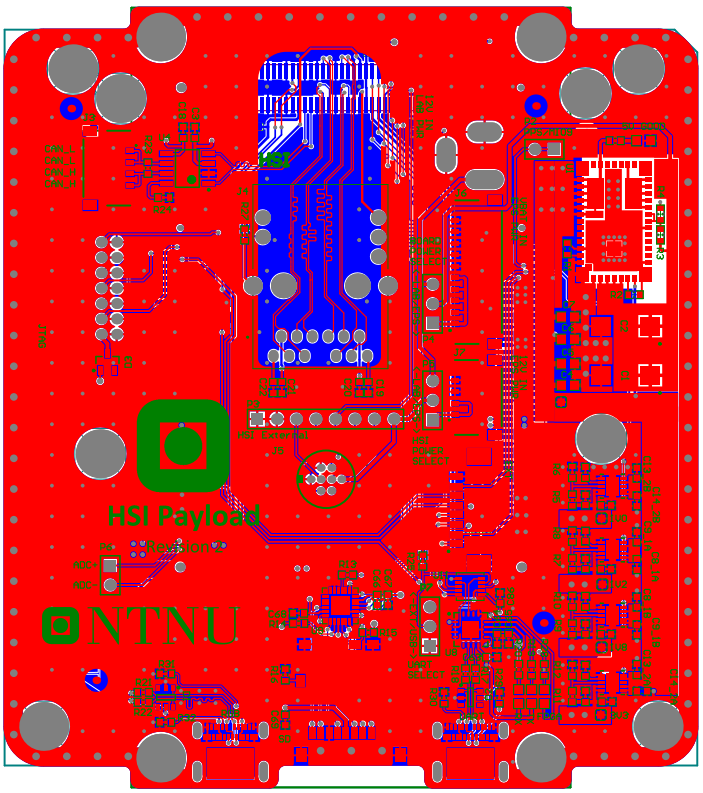


Title:	SD Card Interface		
Project.	HSI Prototype	Engineer:	A. Gjersvik
Pcb#	NTNU-SSL-001	Date:	28.03.2019
Revision	2.0	Sheet	13 of 17



Title:	HSI power and Comms		
Project.	HSI Prototype	Engineer:	A. Gjersvik
Pcb#	NTNU-SSL-001	Date:	28.03.2019
Revision	2.0	Sheet	14 of 17


SMPS_5V
LDO_1V8
Parameter Set
RX
TX
CAN_RX
CAN_TX
CAN_RXTX
C3
100nF
C18
100nF
GND
U4
Txd
Vss
Vdd
Rxd
STBY
CANH
CANL
Vio
CAN transceiver
The internal pull-up in the MCP2562 is 4.4M Ohm according to the datasheet, so a low value pull down resistor should be used. Place 0 Ohm resistor here
R24
0 ohm
R23
120R
J3
CANL 1
CANL 2
CANH 3
CANH 4
4-Pin PicoLock
Title: CAN transceiver
Project. HSI Prototype
Engineer: A. Gjersvik
Pcb# NTNU-SSL-001
Date: 28.03.2019
Revision 2.0
Sheet 15 of 17


HSI Payload
Revision
NTNU

Component list

Bill of Materials for BOM Document [NTNU-HSI-Prototype.BomDoc]

Source Data From: NTNU-HSI-Prototype.BomDoc
Project: NTNU-HSI-Prototype.PrjPcb
Variant: None

Report Date: 28.03.2019 19:08
Print Date: 28-Mar-19 7:08:21 PM

#	LibRef	Manufacturer 1	Manufacturer Part Number 1	PartType	Description	FootPrint	PackageReference	Quantity
1	T521D107M025ATE040	KEMET	T521D107M025ATE040	100uF		Tantalum-D-7343		1
2	T525D476M016ATE035	KEMET	T525D476M016ATE035	47uF		Tantalum-D-7343		1
3	Capacitor	TDK	CGA3E2X7R1E104K080AA	100nF		0603_CAPACITOR		26
4	Capacitor	Murata	GRT21BR60J476ME13L	47uF		0805_capacitor		4
5	Capacitor	TDK	CGA3E1X7R1C105K080AC	1uF		0603_CAPACITOR		4
6	Capacitor	KEMET	C0603C102K3RACAUTO	1nF		0603_CAPACITOR		4
7	Capacitor	Murata	GRT188R61C106KE13D	10uF		0603_CAPACITOR		8
8	Capacitor	Kyocera AVX	06033D105KAT2A	1.0uF		0603_CAPACITOR		7
9	IP4220CZ6	Philips	IP4220CZ6,125	IP4220CZ6		TSOP6		2
10	Diode BAT17	Infineon	BAT17E6327HTSA1	Diode BAT17	Silicon RF Schottky Diode for Mixer Applications in the VHF/UHF Range	SOT-23_N	SOT-23	1
11	0603 Ferrite Bead	Wurth Electronics	742792609	742792609		0603 Ferrite BEad		7
12	microSD Card	Molex	502570-0893	5025700893		5025700893		1
13	14-pin JTAG	Molex	87833-1420	14-pin JTAG		878331420		1
14	4-Pin PicoLock	Molex	504050-0491	4-Pin PicoLock	4-Pin Molex PicoLock	504050-0491		2
15	RJ45	Bel	L829-1J1T-43	RJ45		L8291J1T43		1
16	HIROSE-8PIN	Hirose	HR25-7TR-8PA(73)	HIROSE-8PIN	8 Pin Through Hole HIROSE Socket (HR25-7TR-8PA(73))	HIROSE_placeholder		1
17	10-pin PicoLock	Molex	504050-1091	504050-1091	PicoLock SMT Wire-to-Board Header 10Way Molex 504050 Series, 1.5mm Pitch 10 Way 1	504050-1091		1
18	8-Pin PicoBlade	Molex	53261-0871	8-Pin PicoBlade		8-Pin PicoBlade		1
19	0805 LED	Vishay Lite-On	LTST-C170KFKT	0805 LED		0805-LED		3
20	0805 LED	QT-Brightek	QBLP650-IW	0805 LED		0805-LED		1
21	DC-Jack	CUI	PJ-002AH	DC-Jack		dc-power-jack		1
22	Header 2	Samtec	TSW-101-07-G-D	Header 2	Header, 2-Pin	HDR1X2		2
23	Header 8	Wurth Electronics	61300811121	Header 8	Header, 8-Pin	HDR1X8		1
24	Header 3	Samtec	TSW-103-07-L-S	Header 3	Header, 3-Pin	HDR1X3		3
25	BSS138BKW,115	NXP Semiconductors	BSS138BKW,115	BSS138BKW,115	N-Channel Trench MOSFET, 60 V, 0.32 A, -55 to 150 degC, 3-Pin SOT323, Tape and Reel	NXP-SOT323_N	SOT323	1
26	Resistor	Vishay	CRCW060386K6FKEA	86K6		0603_RESISTOR		1
27	Resistor	Panasonic	ERJ-3EKF1960V	196R		0603_RESISTOR		1
28	Resistor	Panasonic	ERJ-3EKF6812V	68K1		0603_RESISTOR		1
29	Resistor	Panasonic	ERJ3EKF2152V	21K5		0603_RESISTOR		1
30	Resistor	Vishay Dale	CRCW06031R13FKEA	1R13		0603_RESISTOR		1
31	Resistor	Stackpole Electronics	RMCF0603FT4R53	4R53		0603_RESISTOR		1
32	Resistor	TE Connectivity Neohm	CPF0603F2R49C1	2R49		0603_RESISTOR		1
33	Resistor	TE Connectivity Neohm	CPF0603F4R99C1	4R99		0603_RESISTOR		1
34	Resistor	Panasonic	ERJ-3EKF3571V	3K57		0603_RESISTOR		2
35	Resistor	Panasonic	ERJ-3EKF2871V	2.87K		0603_RESISTOR		1
36	Resistor	Panasonic	ERJ-3EKF1151V	1K15		0603_RESISTOR		1
37	Resistor	Panasonic	ERJ3EKF40R2V	40R2		0603_RESISTOR		1
38	Resistor	Panasonic	ERJ3EKF4701V	4K7		0603_RESISTOR		2
39	Resistor	Panasonic	ERJ-3GEY0R00V	0 ohm		0603_RESISTOR		3
40	Resistor	Panasonic	ERJ3EKF27R0V	27R		0603_RESISTOR		4
41	Resistor	Panasonic	ERJ3EKF2700V	270R		0603_RESISTOR		4
42	Resistor	Panasonic	ERJ3EKF1200V	120R		0603_RESISTOR		1
43	Resistor	Vishay Dale	CRCW06034M70JNEA	4M7		0603_RESISTOR		1
44	Resistor	Panasonic	ERJ3GEYJ103V	10K		0603_RESISTOR		1
45	Resistor	Panasonic	ERJ3EKF5101V	5K1		0603_RESISTOR		4
46	Resistor	Panasonic	ERJ3GEYJ101V	100R		0603_RESISTOR		1
47	PicoZed_SOM			PicoZed	PicoZed SOM Carrier Footprint	PicoZedSOM		1
48	LMZ31506	Texas Instruments	LMZ31506RUQT	LMZ31506		LMZ31506		1
49	TPS74801DRCR	Texas Instruments	TPS74801DRCR	TPS74801DRCR	Single Output LDO, 1.5 A, Adjustable 0.8 to 3.6 V Output, 0.8 to 5.5 V Input with Programmable	TI-DRC10-2400X1650TP_V	DRC10-2400X1650TP	4
50	MCP2562	Microchip	MCP2562-E/SN	CAN transceiver	MCP2562 CAN transceiver	MCP2562-SOIC127P600X175-8M		1
51	FT230XQ-R	FTDI	FT230XQ-R	FT230XQ-R	USB to Basic UART Interface Chip, USB V2.0 Compliant, USB 2.0	QFN-16_N	QFN-16	1
52	TXS02612	Texas Instruments	TXS02612RTWR	TXS02612		QFN-RTW		1
53	TS3A24159	Texas Instruments	TS3A24159DRCR	TS3A24159		TS3A24159		1
54	USB-C	Molex	105450-0101	USB-C		PCBComponent_1		2

Approved

Notes 131

Appendix C

HYPSO CLI

Following is a terminal dump from the `hypos-cli` program. The `help` command is executed to show the available commands.

```
Welcome to: HYPSO CLI
Model:      GS Linux CLI
Revision:   May 21 2019
Type help for all commands.
Type help <command> for specific help.
$ help
help [Command]                              - Print help for all
   ↪ functions, or a subset of functions.
exit                                        - Exit the CLI.
q                                           - Exit the CLI.
csp init can [<canX>]                       - Initialise CAN bus for
   ↪ CSP.
csp init flatsat [<Address> <Port>]         - FlatSat for CSP over NNG
   ↪ interface.
csp init service all                        - Initialise all payload
   ↪ services locally with loopback.
csp init service ft                         - Initialise the FT service
   ↪  locally with loopback.
csp init service shell                      - Initialise the shell
   ↪ services locally with loopback.
csp init usart                              - Initialise USART
   ↪ interface over KISS.
csp ping <CSP ID>                           - Send a CSP ping and wait
   ↪ for reply.
csp ping all                                - Send CSP ping to all
   ↪ subsystems.
csp ping rtt <CSP ID> <Number of pings>     - Estimate round trip time
   ↪ with Pings.
csp hello <CSP ID> <CSP Port>               - Send hello world over CSP
   ↪ .
csp mem <CSP ID>                            - request free memory from
   ↪ CSP node.
csp buf <CSP ID>                            - request free buffers from
```

```
   ↪  CSP node.
csp up <CSP ID>                                - request uptime from CSP
   ↪ node.
csp route                                      - Print CSP routing table
   ↪ for this node.
csp conn                                       - Print CSP connection
   ↪ table for this node.
csp if                                         - Print CSP interfaces for
   ↪ this this.
csp debug [<Debug Level>]                      - Toggle CSP debug level.
csp reboot <CSP ID>                            - Request a CSP node to
   ↪ reboot.
csp shutdown <CSP ID>                          - Request a CSP node to
   ↪ shutdown.
ft info <CSP ID> <File ID>                     - Request file info from a
   ↪ node.
ft list <CSP ID>                               - Request file listing from
   ↪  a node.
ft check <ALL|PRESENCE|INTEGRITY> <CSP ID> <File ID> <First Entry
   ↪ ID> <Last Entry ID>
                                               - Request a check of
                                                  ↪ integrity or presence
                                                  ↪  of file entries.
ft check local <ALL|PRESENCE|INTEGRITY> <File Path> <First Entry ID
   ↪ > <Last Entry ID>
                                               - Check integrity or
                                                  ↪ presence of local
                                                  ↪ file entries.
ft clear <CSP ID> <File ID>                    - Request a file to be
   ↪ cleared.
ft clear local <File Path>                     - Clear a local file.
ft format <CSP ID> <File ID> <Entry Size> <Entry Count>
                                               - Request file formatting
                                                  ↪ from a node.
ft format local <Filename> <LOG|STATIC> <File ID> <Entry Size> <
   ↪ Entry Count>
                                               - Format a local file.
ft download cancel <CSP ID>                    - Send request to cancel
   ↪ ongoing download to CSP node.
ft download range <CSP ID> <SRC ID> <Start> <End> <DST Path> <
   ↪ Period[ms]> <Duration[s]> <MTU>
                                               - Download range of entries
                                                  ↪  from file.
ft download id <CSP ID> <File ID> <DST Path> <Period[ms]> <Duration
   ↪ [s]>
                                               - Download complete
                                                  ↪ formatted file with
                                                  ↪ ARQ. Auto-create
                                                  ↪ formatted file.
ft upload <CSP ID> <Filename> <Period (ms)>
```

```
                                        - Upload a formatted file,
                                           ↪ without
                                           ↪ acknowledgement.
ft upload range <CSP ID> <Filename> <Start> <End> <Period>
                                        - Upload a range of a
                                           ↪ formatted file.
ft upload arq <CSP ID> <Filename> <Period>
                                        - Upload complete formatted
                                           ↪  file, with ARQ.
ft register <CSP ID> <File Path> <File ID>
                                        - Register a link for a
                                           ↪ file path to a file
                                           ↪ ID.
ft deregister <CSP ID> <File ID>        - Deregister link for a
   ↪ file ID.
ft prepare <CSP ID> <Source File> <Destination File ID> <Entry Size
   ↪ >
                                        - Create new formatted file
                                           ↪  with data from
                                           ↪ existing file.
ft prepare local <Source File> <Destination File> <Destination File
   ↪  ID> <Entry Size>
                                        - Create new local
                                           ↪ formatted file from
                                           ↪ data from existing
                                           ↪ local file.
ft extract <CSP ID> <Source File ID> <Destination File Path>
                                        - Create new file with data
                                           ↪  from existing
                                           ↪ formatted file.
ft extract local <Source File> <Destination File>
                                        - Create new local file
                                           ↪ with data from
                                           ↪ existing local
                                           ↪ formatted file.
ft buffer file <CSP ID> <Buffer Port> <Period (us)> <File ID | File
   ↪  Path>
                                        - Request a file to be
                                           ↪ buffered on the PC.
rgb init <config file path>             - Initialise RGB camera
   ↪ with config file.
rgb capture <HW trigger {y, n}> <file type {raw, bmp, png, jpg}> <
   ↪ file name>
                                        - Initiates one RGB image
                                           ↪ capture
rgb configure <exposure time (double)> <gamma (int)> <color
   ↪ temperature (int)> <pixel clock (int)>
                                        - Set new parameters values
                                           ↪ . Negative values are
                                           ↪  ignored
```

```
rgb deinit                                  - Uninitialize the camera.
rgb configfile <config file path>           - Load a different camera
   ↪ configuration
rgb print                                   - Print current parameter
   ↪ configuration
eps tm                                      - Request and print EPS
   ↪ telemetry.
clear                                       - Clear the terminal
ls                                          - ls -l --color=always
shell <Command>                             - Run any shell command
   ↪ locally. Ex, "ls -l"
shell remote <CSP ID> <Timeout (ms)>        - Enter remote CLI mode for
   ↪  a node. Enter "exit" to quit.
```

Appendix D

Packet Loss Test

The measurements from the packet loss test are included on the next page.

Parameters					Recorded			Derived			
Input Drop Rate [%]	Burst Window Length [#]	File Size [B]	Entry size [B]	Entries [#]	Minutes [#]	Seconds [#]	Channel average drop rate [%]	Transfer Duration [s]	Effective Datarate [kbps]	Estimated datarate @ 1Mb [kbps]	Decrease in effective datarate [%]
0	0	769541	236	3261	1	58	0	118	52.17227119	417.3781695	0
0	0	769541	236	3261	1	58	0	118	52.17227119	417.3781695	0
1	1	769541	236	3261	2	0	1	120	51.30273333	410.4218667	1.666666667
2	1	769541	236	3261	2	5	1.8	125	49.250624	394.004992	5.6
3	1	769541	236	3261	2	12	3.1	132	46.63884848	373.1107879	10.60606061
3	1	769541	236	3261	2	9	2.8	129	47.72347287	381.7877829	8.527131783
4	1	769541	236	3261	2	24	4.5	144	42.75227778	342.0182222	18.05555556
5	1	769541	236	3261	2	26	5.1	146	42.16663014	337.3330411	19.17808219
6	1	769541	236	3261	2	31	5.8	151	40.77038411	326.1630728	21.85430464
6	1	769541	236	3261	2	35	5.9	155	39.71824516	317.7459613	23.87096774
7	1	769541	236	3261	2	41	6.9	161	38.23806211	305.9044969	26.70807453
8	1	769541	236	3261	2	51	8.1	171	36.00191813	288.015345	30.99415205
9	1	769541	236	3261	3	3	9	183	33.64113661	269.1290929	35.51912568
10	1	769541	236	3261	3	28	10.4	208	29.59773077	236.7818462	43.26923077
15	1	769541	236	3261	4	35	13.9	275	22.38664727	179.0931782	57.09090909
15	1	769541	236	3261	5	8	15.8	308	19.98807792	159.9046234	61.68831169
20	1	769541	236	3261	7	39	20	459	13.4124793	107.2998344	74.291939
25	1	769541	236	3261	10	14	24.7	614	10.02659283	80.21274267	80.78175896
25	1	769541	236	3261	11	0	25.9	660	9.327769697	74.62215758	82.12121212
2	2	769541	236	3261	2	3	2	123	50.05144715	400.4115772	4.06504065
3	3	769541	236	3261	2	12	3.8	132	46.63884848	373.1107879	10.60606061
5	5	769541	236	3261	2	28	5.1	148	41.59681081	332.7744865	20.27027027
8	8	769541	236	3261	2	5	5.7	125	49.250624	394.004992	5.6
10	10	769541	236	3261	2	19	8.3	139	44.2901295	354.321036	15.10791367
15	15	769541	236	3261	2	38	10.5	158	38.96410127	311.7128101	25.3164557
20	20	769541	236	3261	3	27	16.9	207	29.74071498	237.9257198	42.99516908
25	25	769541	236	3261	3	39	15.9	219	28.11108676	224.8886941	46.11872146
30	30	769541	236	3261	2	54	17.3	174	35.3811954	283.0495632	32.18390805
50	30	769541	236	3261	6	33	32.5	393	15.66495674	125.3196539	69.97455471
40	30	769541	236	3261	5	43	24	343	17.94847813	143.5878251	65.59766764
5	30	769541	236	3261	2	5	4.4	125	49.250624	394.004992	5.6
30	15	769541	236	3261	5	39	21.3	339	18.16025959	145.2820767	65.19174041

Magne Hov

Design and Implementation of Hardware and Software Interfaces for a Hyperspectral Payload in a Small Satellite

NTNU
Kunnskap for en bedre verden