CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Control Engineering

Bachelor's Thesis

Adaptable Flight Computer for a Rocket

Design and implementation of a scalable flight computer for the ILLUSTRIA Block II student rocket

Andrej Zlámala
Cybernetics and Robotics

Prague, May 2024
Supervisor: Doc. Ing. Pavel Pačes, Ph.D.

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Zlámala Andrej**	Personal ID number:	**491960**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		

II. Bachelor's thesis details

Bachelor's thesis title in English:

Adaptable flight computer for a rocket

Bachelor's thesis title in Czech:

Přizpůsobitelný letový počítač pro raketu

Guidelines:

The work aims on design and Implementation of a scalable flight computer for Illustria rockets. It is expected to cover following topics:
Extension of the existing PX4 autopilot solution to include a flight module with a state machine for a general rocket flight.
Use of SITL simulation to test the correct implementation of the solution for the new flight module.
A simulation model for the Illustria rocket flight created in Matlab and Simulink will be used.
Design and implementation of a custom PCB for the flight computer using an STM32 microcontroller.
The PCB will be externally powered and will contain I2C, SPI, CAN and USB peripherals.
Implementation of data acquisition from sensors on the PCB such as accelerometer, gyroscope, magnetometer, barometer, and possibly others.
Use of HITL simulation to test the correct implementation of the solution on the actual PCB.

Bibliography / sources:

PX4 User Guide, https://docs.px4.io/main/en/

Name and workplace of bachelor's thesis supervisor:

doc. Ing. Pavel Pačes, Ph.D. Artificial Intelligence Center FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **19.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **15.02.2026**

doc. Ing. Pavel Pačes, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student's signature

/ Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 24. May 2024

....

Andrej Zlámala

Abstrakt / Abstract

Táto práca pojednáva o tvorbe prispôsobiteľného letového počítača pre výkonovú študentskú raketu. Začneme stručným úvodom o výkonových raketách a letových počítačoch. Predstavíme si a vysvetlíme požiadavky, ktoré boli stanovené študentským tímom CTU SR, na základe zmien v novom systéme rakety Illustria Blok II. Odhalíme nástroje a programy, ktoré sme použili pri tvorbe hardvérových a softvérových riešení. Podrobne predstavíme kroky návrhu a osádzania dosky plošných spojov. Priblížime postup tvorby stavového automatu a správ komunikačného protokolu MAVLink pre letový počítač. Poskytneme detailný návod pre implementáciu firmvéru. Predvedieme funkčnosť hardvérového riešenia a najdôležitejších senzorov. Pre overenie softvérových riešení použijeme metodiku HIL testovania, ktorá využije simulácie 1D letu rakety v programe Matlab.

Preklad titulu: Prispôsobiteľný letový počítač pre raketu (Návrh a realizácia škálovateľného letového počítača pre študentskú raketu ILLUSTRIA Blok II)

This work addresses the creation of an adaptable flight computer for a high-power student rocket. A brief introduction on high-power rocketry and the purpose of flight computers is provided. The requirements set by the CTU Space Research student team, based on changes in the new Illustria Block II rocket system, are introduced and described. Additionally, the chosen tools used during the creation of the hardware and software solutions are detailed. The design steps and assembly of the printed circuit board are presented in detail. The design process of the flight commander state machine and a MAVLink communication protocol for the flight computer is demonstrated. A comprehensive guide to the firmware implementation is provided. The functionality of the hardware solution and on-board sensors is demonstrated. A hardware-in-the-loop testing methodology, employing a Matlab simulation of a one-dimensional rocket flight, is employed to verify the software solutions.

Keywords: High-Power Rocket, Flight Computer, CTU SR, Avionics Systems, Illustria, Printed Circuit Board, State Machine, MCU, PX4, Bootloader, NuttX, MAVLink, Hardware-in-the-Loop, Matlab

Contents /

Tables / Figures

Chapter 1 Introduction

This work is both a stand-alone project and a key component of the CTU Space Research Team's transition from the legacy Illustria Block I avionics systems to the new Illustria Block II, Andromeda modular systems.

The main objective is to provide a novel approach to the existing flight computer software and hardware solution by making it adaptable to the new modular system.

We will look at the old Illustria Block I avionics systems, their shortcomings and compare them with the new modular system, for which we will also design an easy-to-integrate flight computer hardware board that will support new, more adaptable and powerful software with a full-featured RTOS.

1.1 CTU Space Research team

The CTU Space Research Team[1] is a student-led initiative at the Czech Technical University (CTU) in Prague, focusing on the development of high-power rockets and other space technologies. Founded in 2021, the team comprises over 50 members from various faculties within the university, including engineering and technology disciplines.

One of their main projects is the Illustria rocket, the largest student-built rocket in the Czech Republic. Illustria features a hybrid rocket engine, modular design and advanced avionics, and aims to reach a target altitude of 3 kilometres. This rocket was showcased at the European Rocketry Challenge (EuRoC) 2023 in Portugal[2], where it competed against other international student teams.

The team's work extends beyond rocket development. They have also created the Stratosat project, which successfully tested avionics in the stratosphere by reaching an altitude of 32,600 metres. These projects help students gain practical experience and contribute to the advancement of the aerospace sector in the Czech Republic.

1.2 High-Power Rockets

High-power rockets are advanced rocketry systems designed to reach higher altitudes and carry larger payloads than standard model rockets. These rockets are an integral part of the research and development activities of academic and amateur rocketry groups. The primary purpose of high-power rockets is to provide a platform for educational, experimental and competitive purposes, allowing students and hobbyists to gain hands-on experience in rocketry and aerospace engineering. For example, the CTU Space Research team's projects, such as the Illustria rocket, are platforms for testing different types of rocket engines, modular design concepts and gaining expertise in the complexities of rocket design, aerodynamics, propulsion and avionics.

[1] https://spaceresearch.cvut.cz/en [cit. 24/05/2024]
[2] https://previous-editions.euroc.pt [cit. 24/05/2024]

The Development of a high-power rockets can be divided into several areas:

- **Structural Design**
  The structure of these rockets is designed to withstand the stresses of high-speed flight and rapid acceleration. Advanced materials such as carbon fibre composites are often used to ensure strength while minimising weight. The modular design of the Illustria rocket makes it easy to integrate different payloads and components, increasing the rocket's versatility.
- **Propulsion Systems**
  High-power rockets typically utilize more advanced propulsion systems than standard rockets. These include solid, liquid or hybrid engines. Hybrid engines, such as those used in the Illustria rocket, combine aspects of solid and liquid propulsion to provide a balance of safety, performance and complexity.
- **Avionics and Control**
  Modern high-power rockets are equipped with sophisticated avionics for navigation, telemetry, and control. These systems include on-board computers, sensors and communications equipment that monitor and control the rocket's flight. For example, the Illustria rocket's avionics system, developed by the CTU Space Research team, ensures communication with mission control and collection of current state data during flight.
- **Flight Phases & Recovery**

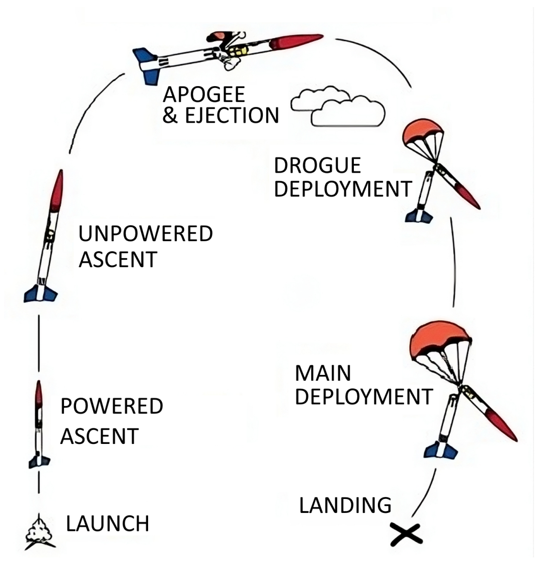


Figure 1.1. Diagram of High-Power Rocket Dual Deploy Flight Sequence

The flight of a high-power rocket is typically divided into several phases: launch, powered ascent, unpowered ascent, apogee, recovery, and landing. Each phase requires careful planning and execution. During the ascent, the rocket's engines provide thrust to overcome gravity and atmospheric drag. When the engines burn out, the rocket coasts to its apogee, the highest point in its trajectory. Recovery systems, such as parachutes, are then deployed to ensure a safe return to the ground. The Illustria rocket uses two parachutes, drogue chute and main chute. The smaller drogue parachute is deployed as soon as possible after apogee to slow down the rocket's descent. Then, after passing a certain height, usually around 300 metres, the larger main parachute is deployed to further slow the rocket for a safe landing and minimise damage to the vehicle.

1.3 Flight Computer

A flight computer is a sophisticated electronic device used in high-power and model rocketry to manage, monitor, and control various aspects of a rocket's flight. At its core, a flight computer performs several key functions. First, it handles pre-launch checks and system arming, ensuring all sensors and control mechanisms are operational. During flight, the computer continuously monitors sensor data, which includes acceleration, altitude, orientation, and velocity. One of the primary roles of the flight computer is to manage the deployment of recovery systems. For instance, during ascent, the computer tracks the rocket's altitude to detect when it reaches apogee, the highest point in its flight. At this critical moment, the flight computer deploys a drogue parachute to stabilize the descent. As the rocket descends further, typically around 300 meters above the ground, the computer then deploys the main parachute to ensure a soft landing.

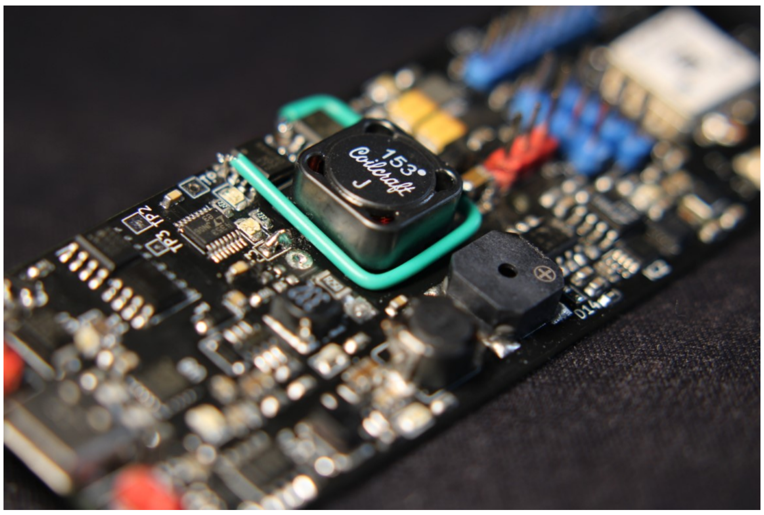


Figure 1.2. Closeup photo of Cimrman V2 flight computer board used by the CTU Space Research team

Additionally, flight computers can be used to control other avionic systems, such as telemetry modules that transmit real-time data back to the ground crew. This data allows for the monitoring of the rocket's status throughout its flight, enabling quick responses to any anomalies. They may also control cameras, scientific payloads, or other experimental equipment, coordinating their operation to gather valuable data during the flight.

In our work we will push this idea further by using a real-time operating system (RTOS) as the basis for our flight computer, which will offer significant benefits in handling real-time data processing and multitasking efficiently. An RTOS ensures that critical tasks such as sensor data acquisition, flight control algorithms and communication processes are executed within strict time constraints, ensuring timely and predictable responses. Developers can add new features or upgrade existing ones without overhauling the entire system. In the context of high-power rocketry, where missions can range from simple launches to complex scientific experiments, the flexibility and scalability offered by an RTOS is invaluable in enabling the development of sophisticated and adaptable flight control systems.

Chapter 2
System Overview and Requirements

2.1 The Rocket

The Illustria rocket is a modular system made up of 7 sections that are connected to each other by a system of section joints called RADAX[1].

Most of the sections are made out of carbon fibre sections. The avionics section is the only one made specifically from fibreglass, as it must not interfere with telemetry systems and radio waves.

The modularity of the rocket ensures that it can be easily reconfigured to a smaller or larger size when required. For example, the smaller configuration is more suitable for testing lower height solid rocket launches or obtaining data from wind tunnel tests, while the larger size is preferred for the main hybrid configuration and accommodates larger oxidiser and pressuriser tanks, as well as a larger engine section.

The brain of the rocket is the Cimrman V2[1] flight computer located in the avionics section. It is connected to several Zora V1 measurement boards in other sections of the rocket via a series of RocketBus cables, which provide power distribution to all the boards, as well as a communications link in the form of a CAN bus.

2.1.1 The Old Illustria system

The Cimrman V2 flight computer[1] uses the STM32WLE5JC Sub-GHz wireless microcontroller with Arm Cortex-M4 @48 MHz with 256 Kbytes of flash memory. For telemetry it uses the integrated LoRa wireless module.

It is connected to an extender board that enhances its capabilities, but most importantly holds batteries and has connectors for the RocketBus cables. Without the extender the Cimrman V2 flight computer could not communicate with the measurement boards in the rocket alone. Extender board also has a GPS GNSS module. Yet the flight computer still has limited memory, with only a 4MB eeprom and no SD card. It has limited connectivity even with the extender board. So there is no SPI, I2C, and only one UART as external peripherals.

There are 3 Zora V1 boards inside the rocket. One is located in the recovery section and handles the drogue and main parachute deployment logic, as well as powering the shear pins. As the Illustria rocket can have a hybrid engine configuration that requires two tanks to hold the oxidiser and pressuriser, the next Zora board is in this section and handles the servo motors used to fill and connect the tanks. The last Zora board is in the engine section, where it is responsible for opening the main valve immediately after engine ignition.

In the mission control, the communication with the rocket is handled through another Cimrman V2 board with the LoRa wireless module, that is programmed simply as a radio-usb transceiver connected to a computer.

[1] Ježek, Vojtěch. Návrh spojů sekcí modulární studentské rakety. ČVUT, 2023[cit. 24/05/2024]. Bachelor's Thesis. Available from http://hdl.handle.net/10467/109891/

2.1.2 Changes in the new Andromeda system

There are several big changes in the new system.

The most important of these is the separation of power distribution and battery management from the flight computer into a dedicated section called the Battery Optimisation & Balancing Enhancement System (BOBES). The other key part is the creation of a new telemetry system (RocketLink), which now includes a new board in the rocket for dedicated handling of communications with the mission control.

These two changes have freed the main flight computer from managing both power distribution and telemetry communications, giving it more room to become a dedicated flight control system. The new Zora V2 board now also supports the CAN FD protocol, has more peripherals and is half the size of Zora V1. Therefore new main requirements for the new flight computer have shifted greatly from its predecessor.

The main requirement is to support the modular adaptable design of the Andromeda system, meaning that the flight computer needs to provide:

- Should be primarily powered from RocketBus and be able to connect with other boards via CAN.
- Have multiple UART peripherals for connecting to RocketLink and possible other radio boards and have at least one SPI and I2C peripheral.
- Have a robust controller and state management for rocket flight.
- Provide an easy way for programming and configuring the firmware.
- Have IMU and Baro sensors.
- The PCB should use the same drill hole placement as the expander board

2.2 Mission control and simulation software

The mission control play a pivotal role in ensuring the success and safety of rocket launches. The team is responsible for a wide range of critical tasks, including pre-launch setup and configuration, real-time monitoring of the rocket's status, and post-launch recovery operations.

To communicate with the rocket and manage configuration and system checks, the CTU SR team uses a custom Matlab GUI application, that allows the ground crew to seamlessly interface with the rocket's flight computer, facilitating the input of pre-flight parameters, monitoring telemetry data in real-time, and making necessary adjustments on the fly. The GUI provides a comprehensive display of vital statistics such as altitude, velocity, and system health indicators, ensuring that the ground crew can maintain situational awareness throughout the mission.

For pre-flight verification and HIL testing of the rocket flight computer state machine, the Matlab application uses a Simulink program to create a simple 1D simulation of a rocket flight. The simulation takes the current rocket and parachute states as input and simulates the vertical acceleration, velocity and altitude of the rocket flight based on this information and a predefined engine thrust curve. It then sends a message back to the computer simulating data collected from the IMU and Baro sensors.

Until now, communication has been done via a custom message protocol that is difficult to scale and extend. For each new message type, both the flight computer and the Matlab application have to be painstakingly rewritten and rebuilt or recompiled. As a result, the CTU SR team decided to use a common solution such as the MAVLink communications protocol. This placed a new requirement on our flight computer, which now has to natively support the MAVLink protocol and provide an easy way to modify its messages.

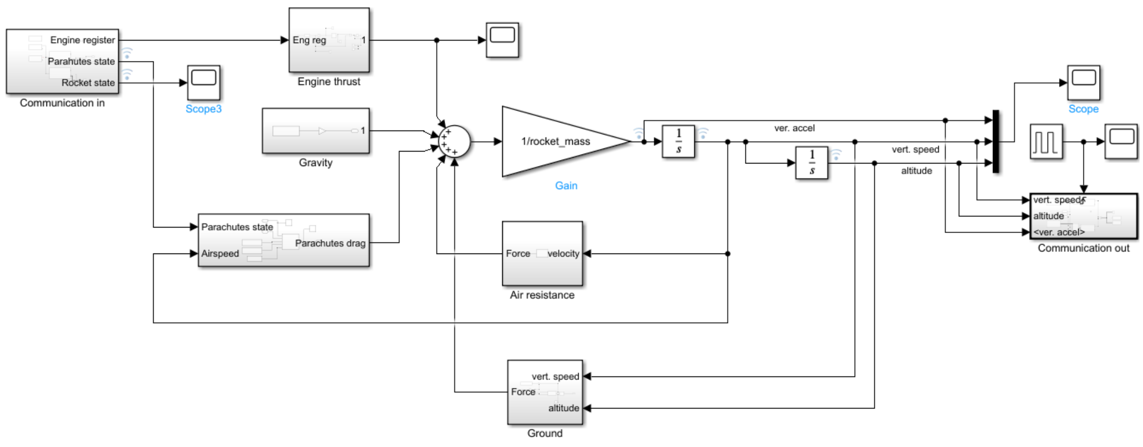


Figure 2.1. Net of the Simulink program that will be used for simulating 1D rocket flight during HIL testing

2.2.1 MAVLink protocol

MAVLink (Micro Air Vehicle Link) is a lightweight communication protocol designed to exchange information between unmanned vehicles and ground control stations. It is widely used in robotics, particularly in applications involving drones, UAVs and other autonomous systems. Developed by Lorenz Meier in 2009, MAVLink has become something of a standard for drone communication due to its robustness, efficiency and flexibility.[2]

It works by encoding messages in a binary format, which ensures efficient transmission and minimises bandwidth usage. Each MAVLink message consists of a predefined structure that includes headers, payload and checksums. Two versions of the protocol are supported: MAVLink v1.0 and MAVLink v2.0, the latter offering enhanced features such as extended message length, IDs and improved security mechanisms.

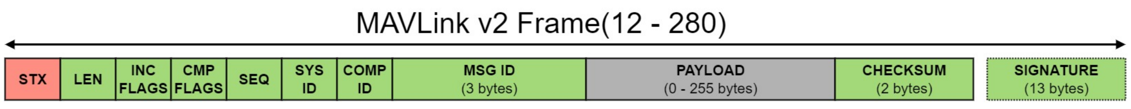


Figure 2.2. Structure of MAVLink v2.0 message frame

One of the key benefits of MAVLink is its simplicity in defining new messages using XML files. These files define the message formats, including fields, types and message IDs. Users can easily add custom messages by editing these XML files, which are then used to automatically generate source code for encoding and decoding messages in various programming languages. This flexibility allows MAVLink to be adapted to a wide range of applications and use cases, making it highly versatile. MAVLink encourages the use of the provided predefined dialects with common messages that are made for standard UAV operations.

PX4 natively supports this protocol, which is used to send telemetry data, receive commands, and coordinate missions. The flight stack includes a MAVLink module that handles message encoding, decoding, and dispatching, ensuring seamless integration with MAVLink-compatible tools and software.

Chapter 3
Toolkit

The purpose of this chapter is to introduce the tools we have chosen to complete our work, as there are quite a lot of them and it will therefore be necessary to give a brief explanation of each one.

3.1 Draw.io (v21.6.8)

Draw.io is a versatile and intuitive diagramming tool that lets you create a wide range of visual representations, from flowcharts and network diagrams to wire-frames [3]. With an easy-to-use interface and an extensive library of shapes and icons, it is a popular choice for brainstorming ideas, planning projects or illustrating complex concepts. Its main purpose is to provide a flexible and efficient solution for communicating information visually. We used it mainly to brainstorm, plan and visualise our implementation of the requirements for our new flight computer hardware board.

3.2 KiCad (v8.0.0)

KiCad is an open source electronic design automation (EDA) software suite used primarily for printed circuit board (PCB) design [4]. It provides a comprehensive set of tools for schematic design, PCB layout and component footprint management, as well as features for generating manufacturing files such as gerber files. It also includes the option of 3D visualisation and design rule checking, enabling users to create professional PCB designs. It also has an active community and provides extensive documentation with plenty of resources and examples for new users. All this makes it an ideal choice for our purposes.

3.3 MATLAB and Simulink (R2022a)

MATLAB is a programming platform designed specifically for engineers and scientists to analyze and design systems. The heart of MATLAB is the MATLAB language, a matrix-based language allowing the most natural expression of computational mathematics [5]. We use it in conjunction with Simulink to create real-time simulations for our HIL testing.

Simulink is a graphical programming environment and simulation tool widely used for modelling, simulating, and analysing dynamic systems [6]. It provides a visual interface for designing block diagrams that enable users to simulate and test control systems, signal processing algorithms, and other dynamic systems. It is integrated with MATLAB, allowing seamless transfer of data and algorithms.

The CTU SR team provided us with a simple 1D simulation model of a rocket flight with a predefined engine thrust curve, which will be used to verify the correct implementation of our flight computer state machine design in the HIL testing as can be seen on Figure 2.1.

3.4 NuttX RTOS

NuttX is a real-time operating system (RTOS) designed for embedded systems with an emphasis on standards compliance and small footprint [7]. Developed under the Apache Software Foundation, NuttX adheres to POSIX and ANSI standards, providing us with a familiar environment close to Unix-like systems. It also supports a wide range of microcontroller architectures, including most of ST STM32 microcontrollers, making it highly versatile for a variety of embedded applications and our work.

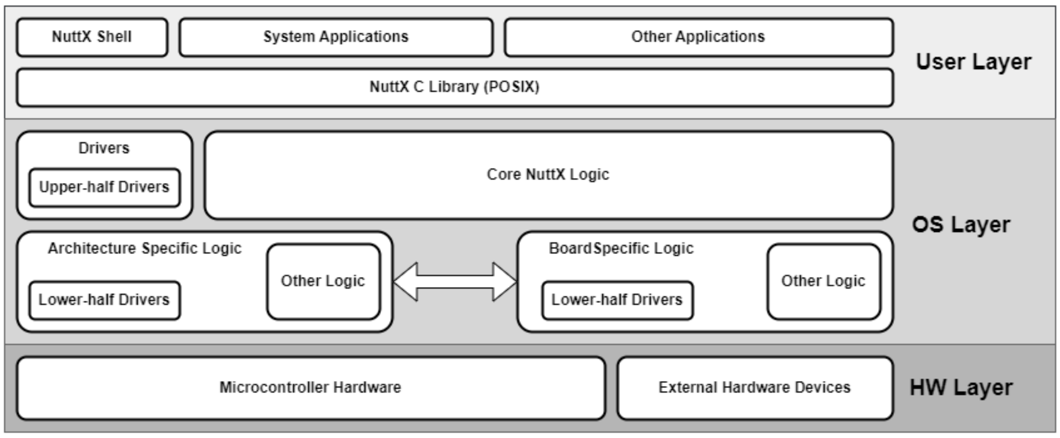


Figure 3.1. Diagram of NuttX Architecture (based on [8])

NuttX's architecture is modular and scalable, designed to run efficiently on resource-constrained devices. It consists of several key functionalities of a proper RTOS that make it suited for our work:

- Task scheduling: The primary scheduling policy is strict priority scheduling, where higher priority tasks precede lower priority tasks. If multiple tasks have the same priority, they are scheduled on a First-In-First-Out (FIFO) basis.
- Inter-Process Communication: Uses queues for message exchange between tasks, semaphores for signalling and resource access, and signals for asynchronous notification and interrupts.
- Pseudo Root File System: It serves as the root file system in NuttX. It provides a directory structure and supports standard file operations such as open, close, read, and write. Character and block device drivers are inherently supported within the /dev directory of this pseudo file system similar to POSIX systems.
- Device Drivers: NuttX provides a wide range of drivers for different hardware interfaces, such as USB, SPI, I2C, CAN, and more. These drivers can be divided into 3 categories:
  - Character Device Drivers: Used for devices that handle data streams like serial ports.
  - Block Device Drivers: Used for storage devices that handle data in blocks such as eeprom or SD cards.
  - Specialized Device Drivers: Include drivers for Ethernet, USB, and other specific hardware that often require additional configuration and interaction protocols

The structure of the NuttX device drivers generally adheres to a two-layer structure consisting of an upper-half and a lower-half:

- Upper-half Driver: This part of the driver is responsible for interfacing with the NuttX RTOS. It implements the standard file operations (open, close, read, write, etc.) and interacts with user space applications.
- Lower-half Driver: This part is hardware specific and implements the actual interaction with the device. It is typically located at the architecture or board level. The upper-half driver communicates with the lower-half driver via a set of callbacks defined in the driver interface.

3.5 PX4

The PX4 platform is an advanced open-source flight control software designed to operate drones and other unmanned vehicles, but there is no implementation or documented vehicle type that could be used to manage the flight states of a rocket. It provides the functionality required for vehicle stabilisation, control and navigation, and integrates seamlessly with a wide range of hardware components. PX4 is widely used in both research and commercial applications due to its robustness, flexibility and support for a wide range of vehicles [9].

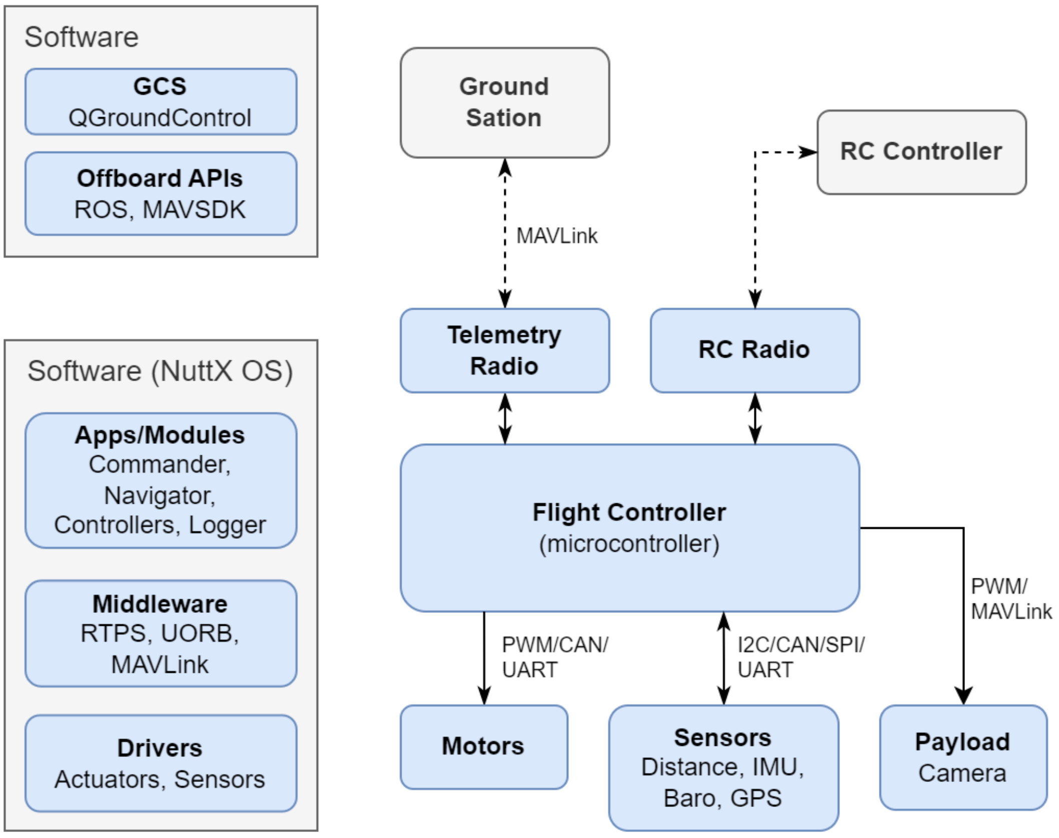


Figure 3.2. High level overview of a typical PX4 System

Let's start by looking at what a typical PX4 system looks like and how it works. The figure above shows a common layout of how the PX4 could be used. The MCU hosts the flight controller software, i.e. the underlying NuttX RTOS and the PX4 software on top of it.

The PX4 software can be divided into 3 layers. Drivers, which handle the IO communication with the external devices through the NuttX high level drivers. Middleware, which handles the inter-process or inter-application communication. And Modules or Applications in themselves, which handle the logic for higher level organisation and control of flight behaviour.

Folder	Description
platforms	Platform-specific code, providing abstraction layers for different hardware architectures.
boards	Board-specific configurations and drivers, tailored for different hardware platforms.
src/lib	Core and support libraries used across the PX4 codebase.
src/drivers	Hardware drivers for sensors and peripherals.
src/modules	Core flight modules like navigator, commander, and sensors.
msg	Definitions for uORB message topics.
ROMFS	File system images including startup scripts, configuration files and parameters.
cmake	Build configuration files that guide the build process.
Tools	Various utilities and scripts to aid in development, testing, and deployment.

Table 3.1. PX4 project folder structure.

PX4 drivers are organised into blocks, each responsible for a specific type of hardware component, such as sensors (IMUs, magnetometers, barometers), actuators (motors, servos), and communication devices (telemetry radios). These are stored in the src/drivers directory of the PX4 source tree. Each driver module typically contains:

- Initialisation routines: Functions to set up the hardware, configure parameters, and initialise communication protocols.
- Data Input Handling: Functions to read data from sensors, process it, and make it available to other parts of the system.
- Hardware Control Interfaces: Functions to send commands to hardware components such as actuators or to control data transmission.

The NuttX drivers, being a POSIX compliant, provide standardized device driver interfaces by their upper-half implementation. PX4 drivers leverage these interfaces to interact with the underlying hardware through read and write operations. The drivers communicate with other PX4 modules primarily through the uORB (micro Object Request Broker) messaging system middleware. This publish-subscribe framework allows different modules within PX4 to exchange data efficiently and asynchronously. For example, a sensor driver will publish raw data messages to uORB topics, which can then be subscribed to by sensor fusion algorithms, estimators and other control modules.

PX4 applications and modules are higher-level functions that use the core systems and drivers to perform specific tasks. They process data from sensors and provide functions for automated procedures using sensor data and control algorithms. They collect and store flight data and monitor the overall state of the system.

Chapter 4
Design of Flight Computer Hardware Board

Given our requirements, we set out to create a new hardware board that would still be based on the previous generation Cimrman V2 flight computer, but would also integrate and support the major changes made in the new Andromeda system.

4.1 From Concept to Diagram

We started by creating a graphical representation in a form of a diagram for the core systems required, the power organisation they would need, and the desired peripherals and protocols we would implement.

As the new Andromeda system is predominantly modular, the most important would be the connection to the rest of the subsystems via the RocketBus link. Therefore we started by specifying the layout of the RocketBus link, which includes the main 11.4V and secondary 5V power line coming from BOBES, as well as the two-wire CAN bus, completed by the two grounding wires to form a six-wire RocketBus. This was followed by the selection of key components such as the MCU, IMU, barometric sensor and power step-down ICs.

4.1.1 Core systems and power distribution

As the earlier versions of the Cimrman flight computer used STM32 MCUs, and almost all other boards in the Illustria and Andromeda systems follow suit, we decided to continue with this approach. Our familiarity and experience with STM32 MCUs also influenced our decision. The STM32H7 series was specifically chosen for the MCU because of its binary compatibility with the latest PX4 FMUv6 software, high reliability, widespread use, good overall documentation, performance and versatility.

The deciding factor for choosing an STM32 chip from the H7 series was mainly the amount of flash size and the maximum processor speed. Although some of the H7 series processors are even dual-core, the PX4 would not be able to take advantage of this, and there is still very limited support from NuttX RTOS for the new H7 series at the moment. This leaves us with the H742, H743 and H753, which are all very similar in terms of performance, amount of peripherals and supported features. Based on price, availability and the package type and size, we chose the STM32H743VIT6 [1].

The next thing every good MCU needs to work precisely is an external crystal oscillator to provide it with an accurate clock signal. Even though the STM32H743VIT6 is brandishing an internal high-speed oscillator its frequency drift could reach around 1000ppm while a proper external one can drop that to less than 50ppm. We chose a solid automotive grade ceramic smd crystal oscillator from the ABRACON-ABM8AAIG[2] series at a frequency of 8MHz.

Thinking back to the modularity of the whole system, connecting to the RocketBus is the next priority. It is our main power supply and a means of communicating with the

[1] https://www.st.com/resource/en/datasheet/stm32h743vi.pdf [cit. 24/05/2024]
[2] https://abracon.com/datasheets/ABM8AAIG.pdf [cit. 24/05/2024]

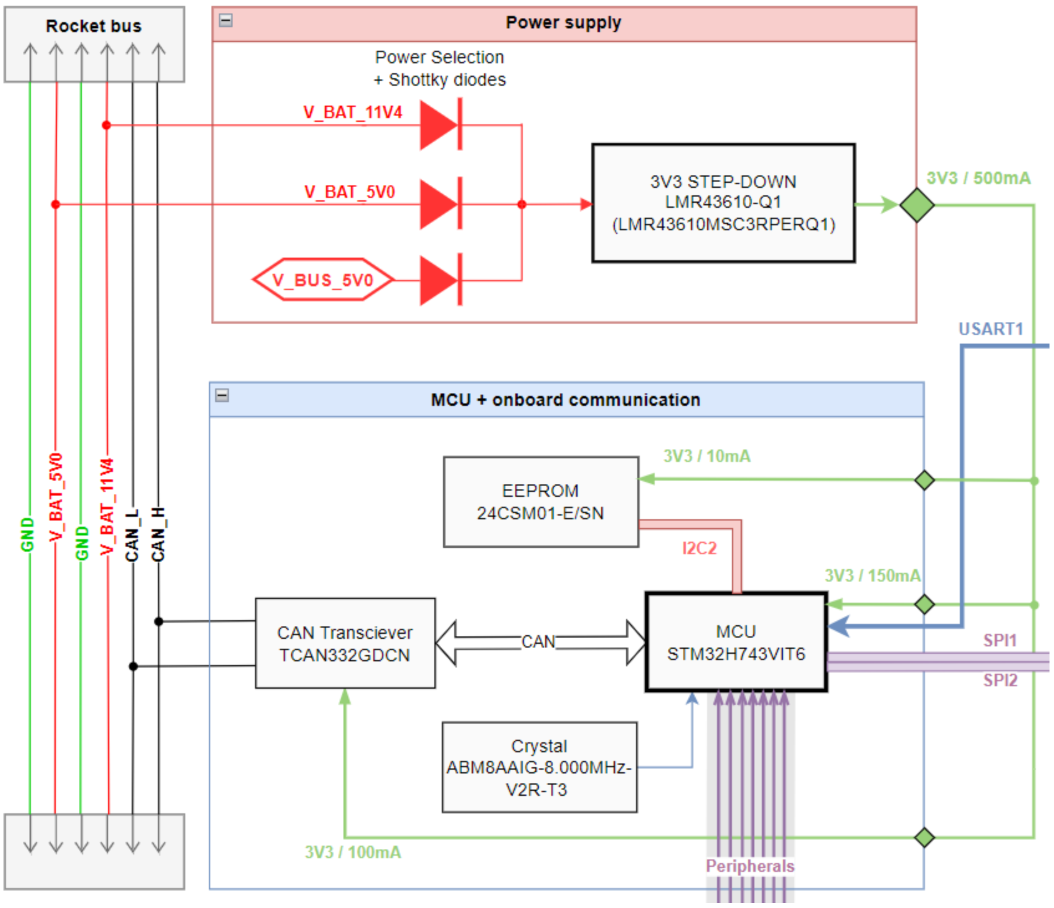


Figure 4.1. Diagram of core systems and power distribution, without sensors or peripherals.

rest of the rocket's subsystems. Although we will not need more than 5V to power all our board components, which can come from the on-board USB port and the RocketBus, we will still support power supplies up to 12V for the sake of adaptability. For this reason we are going to include the possibility of connecting up to a 3S LiPo battery to the board, or selecting the 11.4V power supply from the RocketBus.

As our board's core systems primarily use 3.3V voltages, we have chosen a suitable step-down to reduce the input voltages to acceptable levels. As the voltage difference between input and output can be quite large when using batteries, we chose to use a more efficient step-down rather than a simple LDO, which will produce less heat and current drop during operation. For the power selection we use simple solder jumpers combined with the efficiency of Shottky diodes that have low voltage drop and a forward current of up to 2A. For the step-down we chose one from the LMR43610[3] series by Texas Instruments. We opted for the 1A version, as we do not expect our max current to exceed more than 500mA.

As CAN bus uses differential signals, CAN high (CANH) and CAN low (CANL) we need a CAN transceiver that will be capable of translating this differential pair into CAN TX and CAN RX signals that our STM32 MCU can understand. For this role we chose a popular chip from the TCAN332G[4] series by Texas Instruments, which can also support the CAN FD protocol if we ever have a need for higher speeds (this is also

[3] https://www.ti.com/lit/gpn/lmr43610-q1 [cit. 24/05/2024]
[4] https://www.ti.com/lit/gpn/tcan332g [cit. 24/05/2024]

supported by our MCU). Although our MCU already has 2MB of flash memory, which can be used for fast data storage alongside our programme, it is not much. Looking back at the previous versions of the Cimrman flight computer, they use an eeprom precisely to avoid this bottleneck. Therefore, we have also included an eeprom in a standard SOIC8 package in our design, which is connected to the MCU via an I2C bus.

4.1.2 Sensors

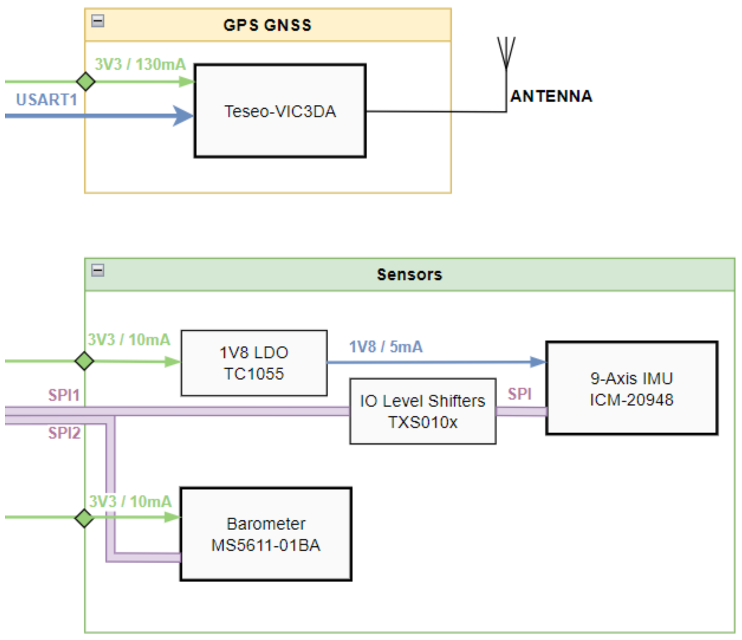


Figure 4.2. Diagram of sensors on board.

In order for our flight computer to carry out the required tasks of recognising the different stages of flight, we need two commonly used sensors. The Inertial Measurement Unit (IMU) for linear and angular acceleration, and a barometric sensor (Baro) to measure the surrounding atmospheric pressure, which can then be used to calculate the current altitude from lift-off. The selection of the specific sensors was made on the basis of these qualifications in the following order of priority:

- Minimise the number of sensors as we have limited space on the board.
- Choose sensors that are supported by PX4 or NuttX and already have an implemented driver.
- Select the sensors that have the best accuracy while being efficient in their power consumption.

For the barometric sensor we chose the MS5611-01BA03[5] series from TE Connectivity. This sensor measures absolute pressure over a range more than sufficient for our rocket flight, from sea level(1bar) to over 20km(10mbar). It has an accuracy of down to 0.012mbar and an operating supply current of only 12.5uA. It is also supported by PX4 and already has a driver already written. All of this meets the requirements for our sensors as it also has a tiny footprint on board.

[5] https://www.te.com/en/product-MS561101BA03-50.html [cit. 24/05/2024]

For the IMU we chose the ICM-20948 which is a 9-axis IMU sensor in one package, that includes an accelerometer, gyroscope and magnetometer. It is an InvenSense IMU from TDK, promoted as „World's Lowest-Power 9-Axis MEMS MotionTracking® Device“.[6] Judging by the qualifications, the 3-in-1 package helps to limit the space used on board. The IMU is also already supported by PX4 and has a driver written. Last but not least it also meets the low power with high enough accuracy requirement. The only downside to all this is that the IMU runs on 1.8V instead of 3.3V like the rest of our ICs. We were unable to find any other suitable replacement that met all three of our requirements, so we decided to just add a 1.8V LDO and a voltage level translator rather than split the IMU into several ICs and potentially having to write multiple drivers. Both the IMU and Baro provide fast SPI peripherals, so we leveraged the multitude of SPI peripherals on our microcontroller to grant the IMU and Baro sensors the advantage of an exclusive SPI Bus for each of them.

The GPS was a late addition, as it was previously intended to be housed on the RocketLink telemetry board. However, due to space constraints, the CTU-SR team decided to move the GPS from the telemetry board to the flight computer. That way, the flight computer has more options if it is used in a self-contained environment without the telemetry board, and the telemetry board has more room to specialise.

The choice of TESEO-VIC3DA from STMicroelectronics as the GNSS / GPS Module was made based on its successful usage on previous projects by the CTU-SR team.

4.1.3 Peripherals

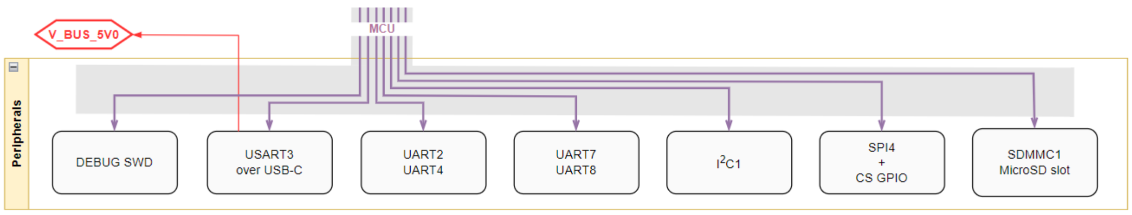


Figure 4.3. Diagram of supported peripherals.

One of the most important connections on the board is definitely the ground and the SWD pins for programming the MCU. These are connected to the PC via a programmer, such as ST-Link, to upload new firmware into the MCU's memory. The same pins can also be used for advanced software debugging at run-time using breakpoints.

As we want to store as much data as possible from our sensors throughout the flight, and also have an easy way of accessing this data, we chose the standard option of integrating an SD card on our board. The MCU provides support for SDMMC protocol with dedicated SDIO pinout interface, which we can use in the fast 4-bit mode for read/write operations on the SD card.

We provide 4 UART peripherals as an easy way for interconnection with other boards, such as the aforementioned telemetry board.

One additional UART is routed through an FTDI chip to support an USB-C connection straight to a Host PC. The same USB-C port is also used as another option of power delivery. This is mostly useful during the programming and debugging phase of setting up the flight computer, where we want to have a single connection to the host PC via a USB cable. When implemented inside the rocket the RocketBus will be used as a primary power source.

[6] https://invensense.tdk.com/products/motion-tracking/9-axis/icm-20948 [cit. 24/05/2024]

4.2 Creating the Schematic

In the creation of Schematics and PCB layout design we opted for KiCad software as its open-source and easy-to-use.

We started by creating a new project and find all the core parts and components, such as the MCU, Sensors, Power Regulators and peripheral devices.

For easy ordering of all components we selected only those available through the Mouser Electronics (`cz.mouser.com` website).

For many symbols and footprints of our parts we had to include them manually as they were not in the default KiCad library. We used online platforms such as `snapeda.com`, `ultralibrarian.com` or directly `mouser.com` to download free footprint and symbol libraries for these parts.

4.2.1 Power delivery and source selection

We began by placing symbols and nets for power delivery peripherals, including RocketBus, USB port and battery ports. The 3.3V Step-Down is the most complex power supply on the board. To ensure proper function of the integrated circuits(ICs), including the MCU, the 3.3 V power supply should have minimal voltage ripple. This can be achieved by using output blocking capacitors and an LC filter. The use of these components results in steady voltage, and means the ADC readings will be accurate and there won't be any voltage drops during current peaks. The power supply uses a numerous amount of capacitors with different capacitance's to achieve this.

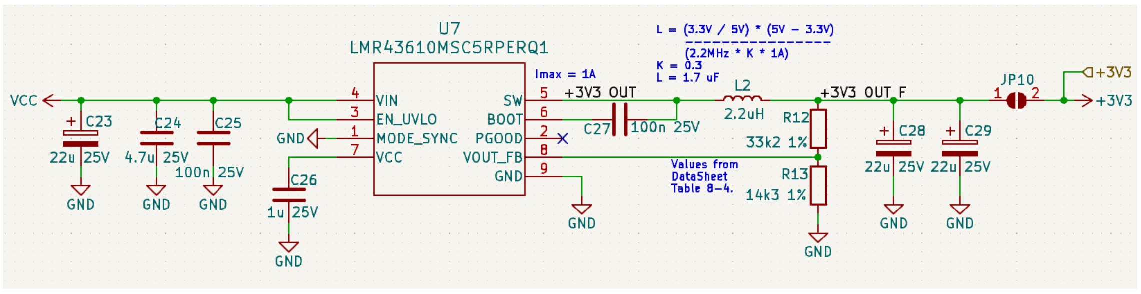


Figure 4.4. Power Delivery 3.3V Step-Down Schema.

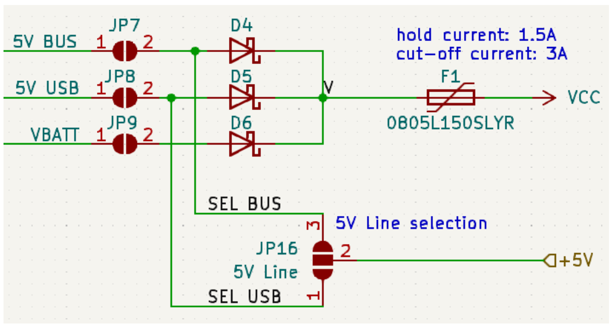


Figure 4.5. Power Selector Schema.

Based on the power delivery requirements set, we provide a way to select the power source. To select the power inputs a set of solder jumpers is provided with a set of Shottky Diodes guarding against reverse current. A resetable smd fuse is utilized to prevent damage to the board in case of a short or over-voltage situation.

4.2.2 MCU

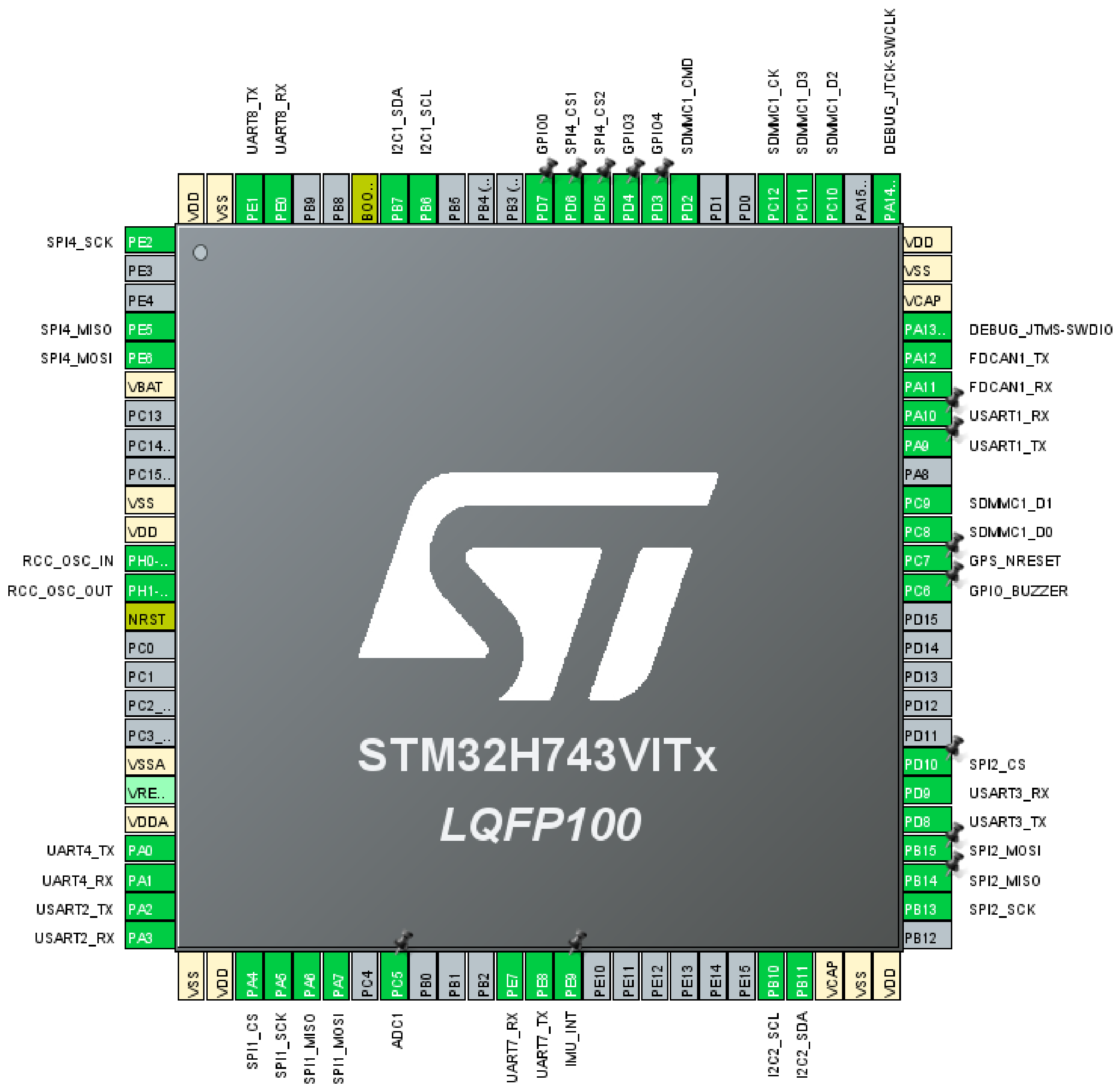


Figure 4.6. MCU pinout mapping.

We configured the correct pin mapping composition based on the peripheral protocols used using the STM32CubeMX [7] program. We enabled and placed the pinout for SWD, SDMMC, SPIs and UARTs used by Sensors and other ICs, which gave us an origin layout. We then refined the mapping during the schematic and PCB layout creation process, adding peripheral and less important connections such as GPIO for the buzzer and SPI chip select or ADC pinout.

The schematic was initiated by placing the MCU symbol in the central position, as it is the primary controlling component. The ground ports were then connected to the GND net symbol, and the VDD ports were linked to the 3.3V power source net. For each VDD pin, a standard decoupling capacitor of 100nF was used to regulate peak

[7] https://www.st.com/en/development-tools/stm32cubemx.html [cit. 24/05/2024]

power draw. We also added one 4.7uF capacitor as power storage for the MCU which will help with smoothing the overall power draw during the startup and booting phase. For the VDDA ports, that are used as power source for analog operations inside the MCU, we detached a net from the 3.3V one using a ferrite bead, that helps filter any high frequency noise further, and we also added another 1uF and 100nF decoupling capacitors.

After connecting the power, we continued by giving the NRST port an external pad for easy debugging and also a 100nF capacitor that will aid in keeping it in a powered state by default. By connecting it to ground we can later power down and force a restart of the MCU. We also added a pull-down resistor to the BOOT0 port which can be used to force the MCU into Bootloader mode when connected to the VDD voltage. The MCU also provides two VCAP ports for connecting additional external decoupling capacitors for further smoothing the internal analog operations. We connected a 2.2uF capacitor to each port.

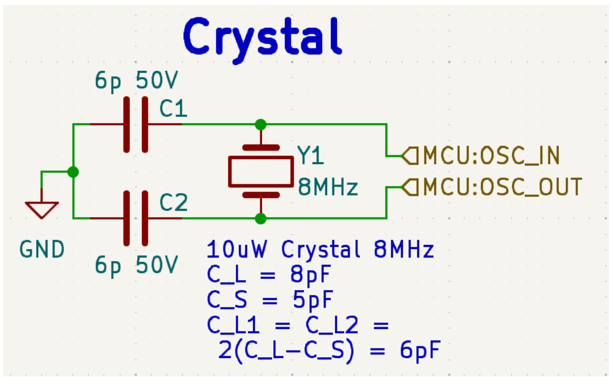


Figure 4.7. MCU Crystal Oscillator Schema.

The external crystal oscillator that will provide the MCU with a clock needs to have two capacitors as shown on the Figure 4.7. These need to be the same value and precisely calculated using a formula to minimize external interference and provide clear clock signal without disturbances based on the crystal used.

The final step was to connect the SWD ports, SWDCLK and SWDIO, with the NRST port to a connector that will be used for debugging the MCU firmware. The connector also included a connection to GND and 3.3V, with overvoltage protection using a Zener Diode in accordance with standard practice.

4.2.3 Sensors

Connecting the sensors was the next crucial step in the schematic design. Each sensor power connection is done via a solder jumper, so we can measure current draw and isolate the sensor during assembly and debugging. Additionally, a 100nF decoupling capacitor is attached to the VDD port of each sensor.

As previously stated in the design chapter 4.1.2, the IMU runs on 1.8V and not 3.3V as the other ICs. This is addressed by using an LDO that drops the voltage from 3.3V to 1.8V with minimal losses as the sensor draws only milli-amps at peak usage. The more significant issue is that the MCU communicates on its ports at 3.3V voltage level, so it is not possible to connect it directly to the IMU, which uses only 1.8V. We therefore utilise the power of voltage level shifters.

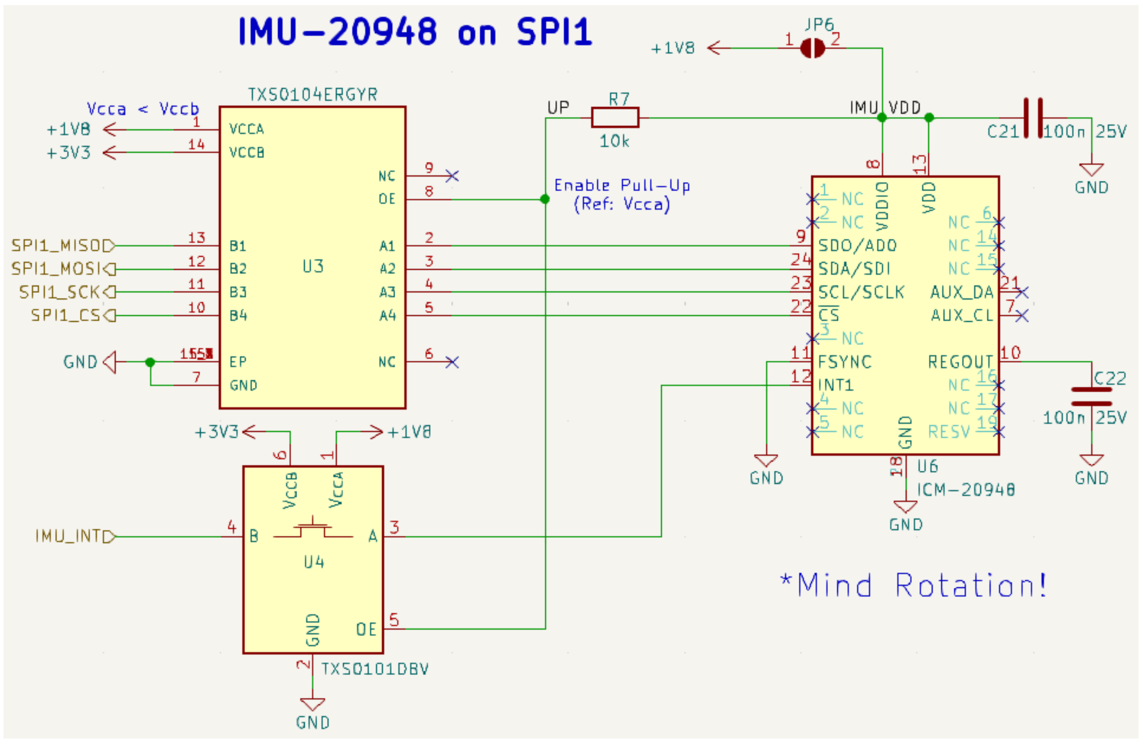


Figure 4.8. IMU Connection through Volatge-Level Shifters Schema.

The voltage level shifters, in their bidirectional form, enable us to connect two components operating at different voltage levels by translating the signal voltages sent between them. They employ MOSFET transistors and pull-up resistors to rapidly and flexibly regulate the voltage levels of signals passing through them. The voltage regulators chosen also have an enable port that we tied using a pull-up to the VDD of our IMU. In this way we were able to connect the IMU using a dedicated SPI bus to the MCU.

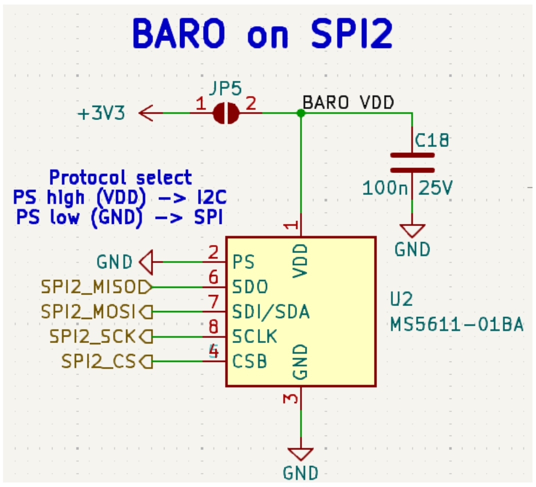


Figure 4.9. Baro Sensor Schema.

The connection of the baro sensor was straight forward. Based on the datasheet we correctly set the PS port of the Sensor IC to GND, selecting the SPI interface. Then we connected it to the MCU using the SPI2 bus. This means attaching the MISO, MOSI, SCK and CS pinouts of both ICs.

4.2.4 Peripherals

One of the requirements for the ease of programming and debugging was the selection of the USB-C connector as one of the boards' power sources and a serial communication interface.

The USB-C standard differentiates between current sink and current source devices and supports active power delivery up to 5 A at 20 V (100 W). However, such advanced power delivery is not necessary for our application. The fundamental USB-C connector is capable of delivering up to 500 mA at 5 V without any modifications. The addition of a pair of 5.1 kΩ pulldown resistors between the CC pins and GND configures the device as a power sink, increasing the power limit to 1.5 A at 5 V, which is more than sufficient for our application.

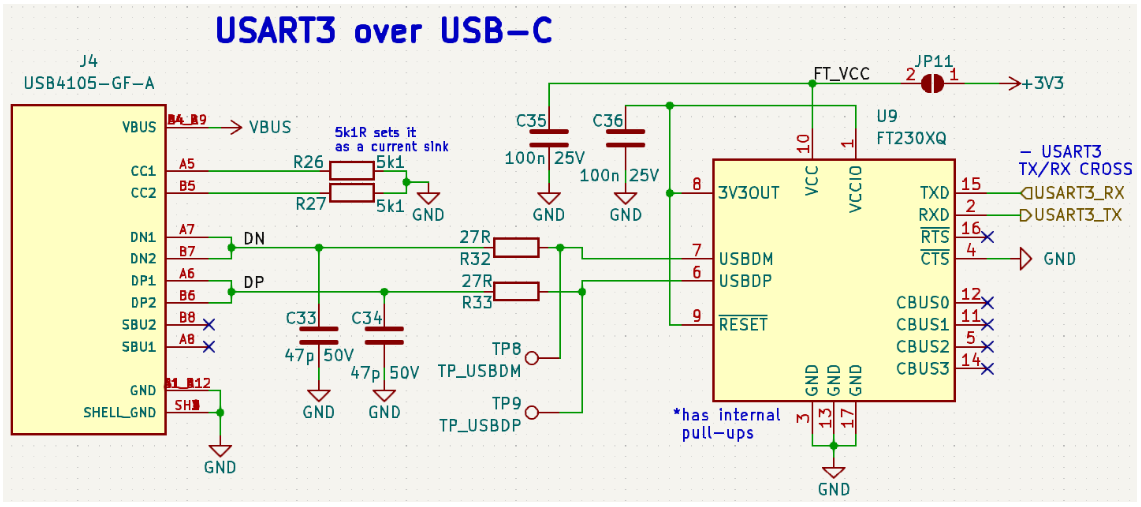


Figure 4.10. USB-C VBus FTDI Connection Schema.

As the connector is only used for power delivery and communicating via UART using an FTDI chip, it is not a concern that the physical connector is only capable of supporting USB 2.0 speeds. The advantage is that it is equipped with a smaller number of pads in comparison to a USB 3.0 connector, which makes the assembly process easier.

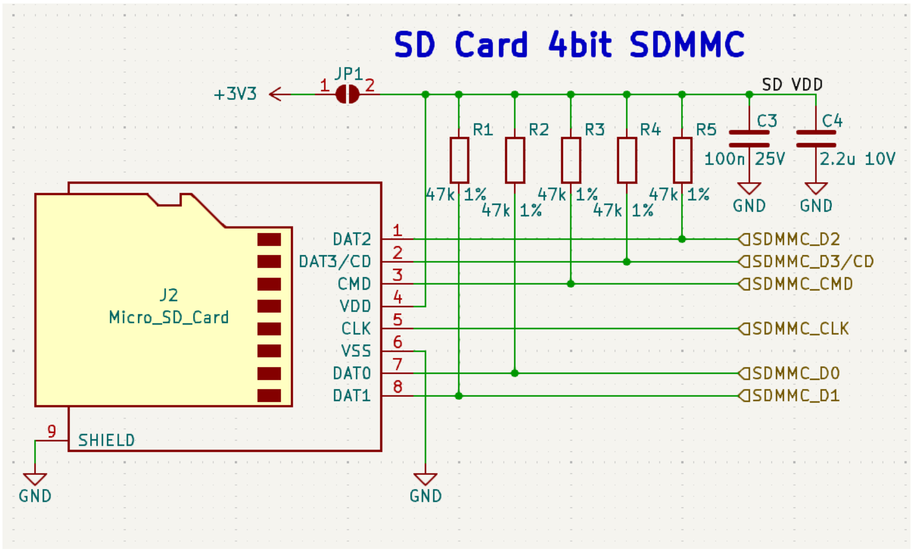


Figure 4.11. SD Card slot Schema.

For the SD card connector, we chose one that allows full opening and closing with a locking mechanism, which is particularly useful in high vibration environments such as inside a rocket. To connect the SD card to the MCU, we took advantage of using the SDMMC SDIO interface with a 4-bit data bus mode for higher transfer speeds. We also added external pull-up resistors to the data lines and decoupling capacitors to the SD card VDD line.

4.3 Printed Circuit Board Layout Design

To guarantee the optimal outcome, the schematic underwent a meticulous review by multiple individuals from the CTU SR avionics team. This comprehensive review process aimed to identify or rectify any potential issues and improvements. Once this in-depth evaluation was complete, we proceeded to the next crucial step: designing the printed circuit board (PCB). This design phase translated the refined schematic into a physical layout, carefully planning the placement of components and routing of electrical connections to create a functional and efficient PCB.

For our PCB Design we chose a 4 layer board, mainly to minimize manufacturing costs, as we anticipated the routing on board with this size should not be a problem. The top and bottom layers should include signal routing and ground, while the middle ones will be preferably only the ground and power planes with minimal signal routing.

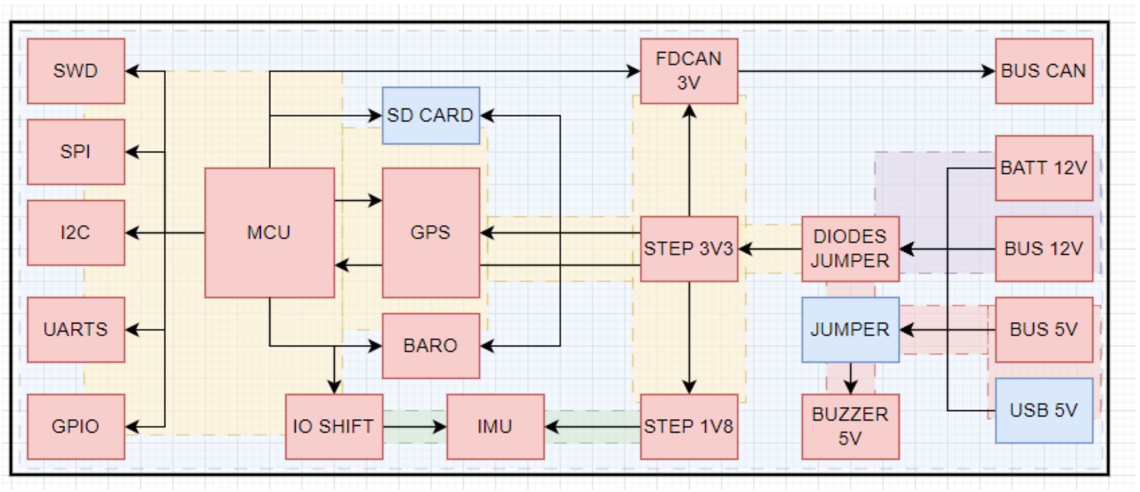


Figure 4.12. Diagram plan for Power Delivery on board the PCB.

To ensure the most logical placement of components and minimize routing lengths we created a diagram to help us identify and specify the power zones and power delivery pathways, as well as the most important logic signal routes, using a graphical representation seen in Figure 4.12.

We have placed all the power delivering devices and connectors on one side of our board. On the other side we placed our MCU and all its peripherals. We then placed the rest of our components in the middle, while trying to minimize the power and logical signal routes lengths between all the components on the board.

Following this guide we first created a PCB board inside the KiCad PCB Editor with the required arrangement of drill holes. We configured the default editor layers to correspond with ur 4 layer design. We also setup the Design Rules and Constraints based on the JLCPCB Manufacturing & Assembly Capabilities[8].

[8] https://jlcpcb.com/capabilities/Capabilities [cit. 24/05/2024]

Based on this, we specified the track and via sizes that would be used. For the tracks, these were widths of 0.2, 0.3, 0.5 and 0.75mm. During placement, we selected which based on the maximum current that could be applied to the track. With regard to the vias, two sizes were employed: one with a 0.2mm hole and the other with a 0.4mm hole. Once again, the selection of the appropriate size during the placement process was based on the prevailing current requirements or space constraints.

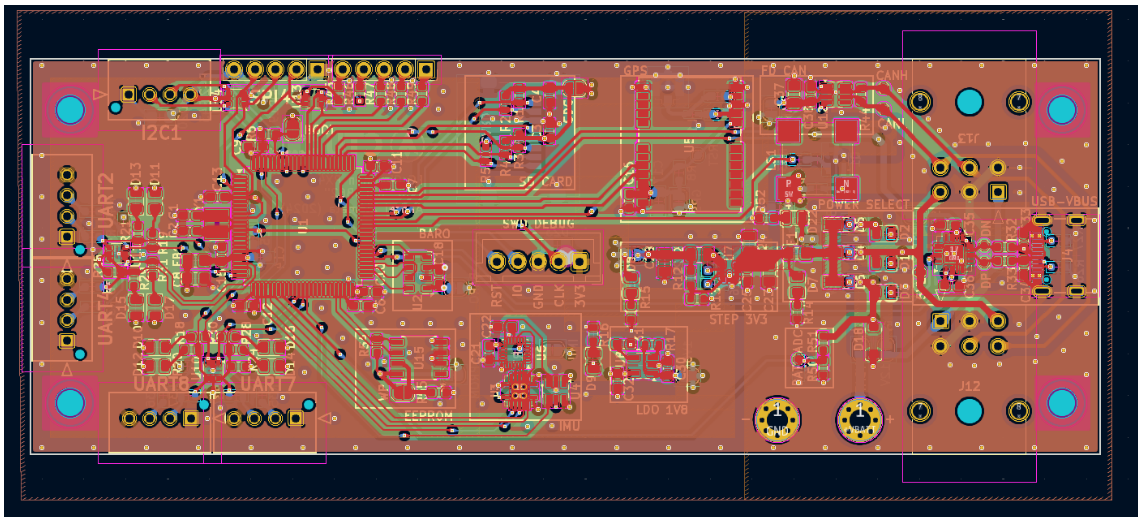


Figure 4.13. PCB Routing inside the KiCad PCB Editor.

The initial step in the PCB layout design process was to transfer all components from the schematic to the PCB Editor. This was followed by grouping components that belonged together with their respective passive components based on the schematic. This allowed us to identify the approximate area each component group would occupy on the PCB board. Finally, the groups were placed on the board in accordance with the previously specified layout diagram 4.12.

We proceeded to add the footprints for the thought-hole connectors, which provide the inputs and outputs for our PCB. These were placed on the edges of our PCB. This gave us a starting point from which to begin routing the power connectors on the right hand side of the board to the centre of the board, where the power selection and regulation would be placed. On the left side of the board we placed the MCU and routed the connections for peripherals such as UARTs, SPIs, I2C and GPIO pins. Finally, we connected the components in the centre, which included the IMU and barometric sensor, eeprom and GPS. When routing the logic signals, we aimed to minimise trace length and the use of vias. Our aim was to place the majority of our components on the top side of the board to simplify the assembly process.

With the main components placed and the initial routing completed, we focused on optimising the signal and power distribution routing. Higher speed signals were routed with great care to avoid crossovers. Decoupling capacitors were placed close to the power pins of the ICs to filter out high frequency noise and ensure a stable power supply. Over-current and over-voltage protectors were placed as close as possible to the input connectors. Finally, as the last step in the process, we added the ground and power planes to the PCB. The ground planes were added to each layer to minimise the size of the return current loops and connected with vias around the edge to reduce the effect of external signal noise. We also added vias where necessary around the board to avoid large voltage differences across the ground planes.

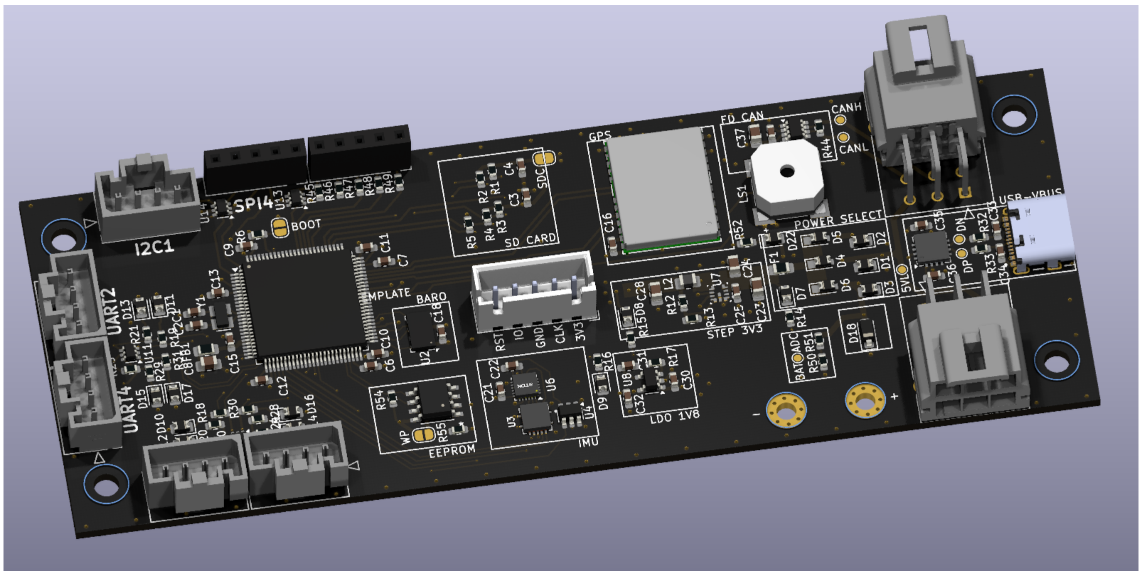


Figure 4.14. Render of the finished PCB Layout.

4.4 Board Assembly

We used the services of JLCPCB[9] to have our PCB boards manufactured. With the PCB we also ordered a stencil to simplify the assembly process. A stencil is a thin sheet of metal with precisely cut openings that correspond with the soldering pads on the PCB. This helps to correctly apply and spread the solder paste during the assembly process. Hand soldering is possible for most larger smd components, but a stencil is essential for consistent and reliable solder joints.

Once the solder paste had been applied precisely to the soldering pads on the top side of the board using the stencil, we picked and placed each and every smd component by hand. Once all the components had been placed on the top side of the board, excluding the connectors, we carefully placed the whole board inside a programmable laboratory oven. This oven slowly melts the solder paste and solidifies the connections by heating and then cooling them based on a predefined temperature curve. After the board cooled we visually and electrically checked for any bad connections, and then moved to hand soldering also the bottom-side smd components. As last we soldered the through hole plastic connectors, as these cannot be baked in the oven.

When the board was fully embedded with parts and visually and electrically checked for any errors, we proceeded with reviving the board. We began by checking for any shorts or undesirable miscellaneous connections. Once everything was checked, we started to connect the power selector solder jumpers and the board through USB-C to the power source. The power indicator LEDs lit up one by one, and we checked that the step-down and LDO were giving out the correct and steady voltage values of 3.3V and 1.8V.

The board was now successfully powered and the MCU was ready to be programmed. We connected the board to an ST-Link debugger via the SWD connector and the debugger to our PC. Using the STM32CubeProgrammer[10] software we were successful at connecting to our MCU and were able to erase or flash its memory.

[9] https://jlcpcb.com [cit. 24/05/2024]
[10] https://www.st.com/en/development-tools/stm32cubeprog [cit. 24/05/2024]

Chapter 5
Flight Commander Design

One of the main goals of our work is to design and implement a flight commander that will be able to control and manage the state of the rocket based on commands and sensor data. This system must be highly responsive and capable of real-time processing to adjust the behaviour of the rocket during flight. The software architecture should support modularity and scalability, allowing for future upgrades and the inclusion of more advanced features.

We chose the PX4 platform, which is widely used as an advanced open source flight control software for operating drones and other unmanned vehicles. It runs on the NuttX RTOS, which is modular and scalable, designed to run efficiently on resource constrained devices such as our microcontroller in the rocket.

The integration of our flight commander as a module into the PX4 platform enables sophisticated functionality such as precise state control, data processing and flight control under varying conditions. The PX4's rich set of drivers and middleware components enables seamless interfacing with a wide range of sensors and peripherals, ensuring full situational awareness and control.

In addition, the real-time capabilities of the NuttX RTOS ensure that the system can handle the high-frequency data acquisition and processing required for rapid in-flight adjustments. By leveraging the modular nature of PX4 and the NuttX RTOS, we can continuously improve our flight control algorithms and incorporate community-driven enhancements, ensuring that our system can easily adapt to changing requirements in the future.

5.1 State Machine Design and Architecture

We begin the process of designing our flight commander module by first analysing the various phases of rocket flight to create a comprehensive state machine that will manage the various processes and transitions throughout the flight.

By creating a visual representation in the form of a diagram, it helps us to systematically outline the rocket's flight phases, state transitions and control processes. By showing the precise behaviour in each state, its actions during transitions and interactions with the rest of the system, we can identify potential problems early and ensure a thorough understanding of the system's requirements and behaviour. This approach reduces the likelihood of errors in later design and implementation phases by allowing us to address complexity and dependencies up front. Ultimately, these diagrams serve as a blueprint to guide our development and implementation processes.

The first state the commander starts in is the CONFIG state, which is important because we want to clearly separate the Armed and UnArmed states for our rocket. So we start with the rocket unarmed, which means that the high-power devices such as shear pins, servos and igniters are inert and safe to handle. In this state the software can be configured with new parameters that will define the behaviour during launch.

From the config state, we can switch to the ARM state. This can be done by running a command directly on our module from the nsh shell, or by sending a command message via Mavlink. In the armed state we now have permission to send signals to the high-power devices, such as rotating servos or enabling power delivery to shear pins and igniters. It is still possible to return from the armed state to the unarmed state, as the armed state can be used for testing different configurations of high power devices prior to launch.

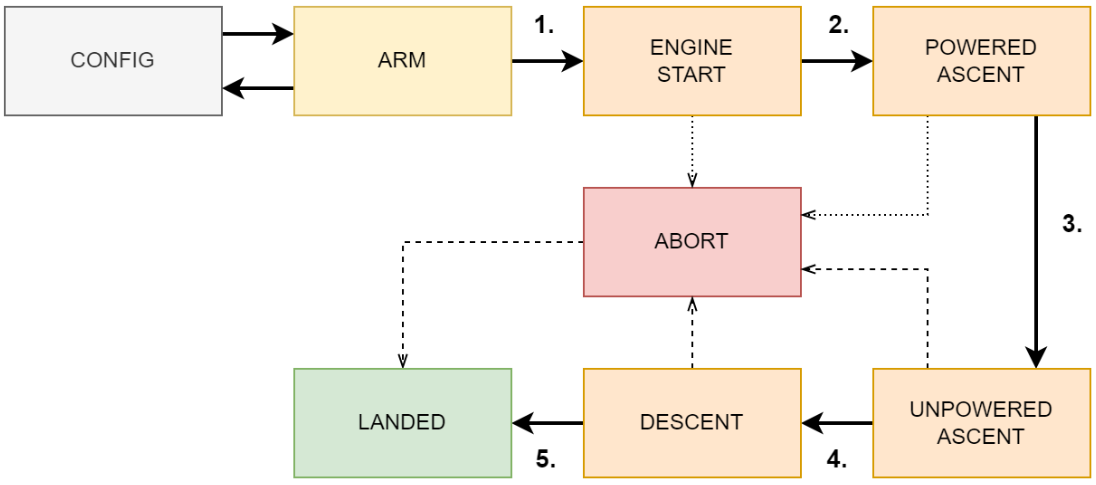


Figure 5.1. Diagram of rocket flight states.

In the same fashion as we transitioned from config to arm state, we can transition the state further to ENGINE START state. It is important to note that this is the first non-return state as the countdown and engine ignition process has begun and the only way back is to abort the launch. In this state, the rocket engine is successfully ignited and we are waiting for lift-off detection to move us to the next state.

After successful lift-off detection we enter to the POWERED ASCENT state. In this state the rocket engine is still firing and we are accelerating. The next transition occurs when we detect the engine cut-off and the end of acceleration.

The UNPOWERED ASCENT state is when the rocket is coasting to the highest point in the trajectory, the apogee, while losing speed. Apogee detection triggers the next transition to the descent state.

At the beginning of the DESCENT state, when we have minimum vertical speed, it's most advantageous to deploy the drogue parachute. Later in the descent, after reaching a critical height above the ground, we will deploy the full main parachute to slow us down further for a safe landing.

When we detect zero vertical speed and drop to the launch height we enter the LANDED state. In this state we know the rocket is back on the ground after flight, so we begin the shutdown procedures of powering down and unarming to facilitate safe recovery by the team.

Now let us talk a little about the ABORT state. If the launch or flight are not nominal and the rocket could become dangerous to its surroundings during any of the irreversible states, we can manually trigger the ABORT state. The transition to ABORT state will be different depending on the previous state. It can be divided into two main possibilities. Abort during engine start or powered ascent, and abort during unpowered ascent or descent. Let us work our way from the end.

If we abort during the descent, it will usually be because the parachutes have not deployed. Therefore, the action taken during this abort will be to continually attempt to deploy parachutes in rapid succession to restart the deployment process.

In the event of an abort during the unpowered ascent phase, which could be due to the rocket going off course and becoming dangerous, we deploy both parachutes again in rapid succession to ensure the deployment.

Now, if we abort while the rocket is still on the launch pad, it would be pointless and perhaps even more dangerous to try to eject our parachutes. Therefore if the rocket is powered bya solid motor, there is nothing we can do. If by chance we have the possibility to regulate the engine power by closing the fuel or oxidizer intake, than thats the only action we will take in this case, and await manual reset.

In the case of abort in the powered ascent phase, it is up to debate. If we cant regulate the motor then it might be unsafe based on the rocket airframe to eject parachutes. It would be best to provide the operator with a configuration during the config state to set the best behaviour during this step for their use case.

Parachute Deployment Sequence

- UNARMED State, the parachute deployment mechanism is unpowered
- ARMed State, the parachute deployment mechanism is now powered.
- DROGUE State, ejected Drogue parachute from the rocket
- MAIN State, ejected Main parachute from the rocket (The Drogue is already out)

State	Description
CONFIG	UnArmed State. Operator can set up and configure the Flight Commander parameters.
ARM	All high power devices (servos, igniters, shearing pins, ...) are now armed. The Operator can still return back to the Configuration phase.
ENGINE START	We await Engine Ignition and start the Lift Off Detection. Can be Aborted.
POWERED ASCENT	Rocket engine is firing and we are accelerating. Can be Aborted depending on configuration.
UNPOWERED ASCENT	We are coasting to the Apogee while losing speed. Can be safely Aborted.
DESCENT	The Rocket is now falling to the ground and we start the parachutes deployment sequence. Can be safely Aborted.
LANDED	We have detected landing. We UnArm high power devices for safe recovery and conserve power while waiting.
ABORT	Behavior based on configuration and depending on current state. Can be configured. In most cases will override the parachute deployment sequence and try to launch both drogue and main parachute repeatedly until successful landing.

Table 5.1. Rocket flight state machine.

5.2 MAVLink Messages and Matlab connection

We have created our own dialect which extends the common messages that are provided by MAVLink protocol and used by the PX4 platform by default. In this dialect we define the Rocket State message which encompasses both the state of the Rocket and the Parachutes. You can view the `rocket.xml` file, 9. It includes definitions for two enums. One contains the Rocket States as they are defined in the Table 5.1. The second one defines the Parachtute States as shown in the Parachute Deployment Sequence.

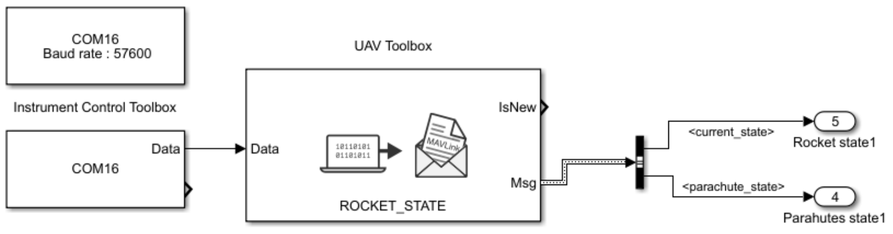


Figure 5.2. Simulink communication input.

To handle MAVLink messages inside Matlab, we use the UAV Toolbox[1] package, which provides support for deserialization and serialization of MAVLink messages. This can be seen on Figure 5.2, where it is able to handle raw data from the incoming connection and only act if Rocket State message gets detected. Similarly on the Figure 5.3 we use it to create new MAVLink messages and serialise them for the outgoing connection.

The Instrument Control Toolbox[2] is used for configuring and creating UDP or Serial I/O connections. It always needs one single setup block, as can be seen on Figure 5.2, which handles the configuration. Then you can have any number of input blocks that provide you with raw data from the connection, or any number of outgoing blocks for sending raw data.

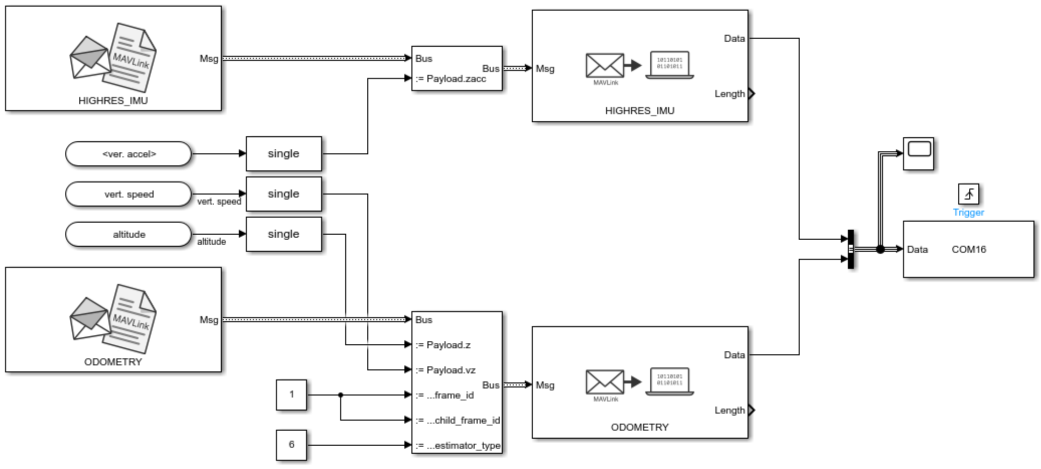


Figure 5.3. Simulink communication output.

[1] https://www.mathworks.com/products/uav [cit. 24/05/2024]
[2] https://www.mathworks.com/products/instrument [cit. 24/05/2024]

Chapter 6
Firmware Implementation Guide

We have chosen our MCU so that it is already supported by the NuttX RTOS. However if our selected MCU was not yet on the list of Supported Platforms in the NuttX RTOS documentation[1], we can still use the handy and exhausting porting guide to port NuttX RTOS on to the new device.[2]

Although NuttX RTOS supports our chosen MCU, we still need to configure the pin mapping and clock settings to ensure proper functionality. Using the PX4 platform on top of NuttX RTOS requires a different configuration setup than just plain NuttX RTOS.

One key difference with the PX4 platform is the inclusion of a bootloader. A bootloader is a critical piece of firmware that initialises the hardware and loads the main firmware application at power-up. In addition, the PX4 bootloader also facilitates the flashing of new firmware onto the microcontroller. This is particularly useful for updating the flight control software without the need for physical access to the microcontroller via a debugger. The bootloading process ensures a reliable and secure startup sequence for the hardware and applications managed by the NuttX and PX4 stack.

We will now take a look at implementing a custom board configuration on top of the PX4 and NuttX RTOS stack.

6.1 Preparing the Environment

For our Linux environment we used a fresh installation of Ubuntu 20.04 running on Windows Subsystem for Linux (WSL) v2 under Windows 10.

To prepare the environment we followed the handy PX4 development setup tutorial on their page[3] Starting with the PX4 project, we will have to download the PX4 source code from [10]. It uses git submodules so don't forget to clone it using the `--recursive` switch to also fetch all the submodules.

```
git clone https://github.com/PX4/PX4-Autopilot.git --recursive
```

Then we have to setup our development tools. There is a script that can do this for us in the PX4 Project. Just by running

```
bash ./PX4-Autopilot/Tools/setup/ubuntu.sh --no-sim-tools
```

all the important development tools and dependencies got installed. Before continuing do not forget to relogin or restart your computer.

[1] https://nuttx.apache.org/docs/latest/platforms [cit. 24/05/2024]
[2] https://cwiki.apache.org/confluence/display/NUTTX/Porting+Guide [cit. 24/05/2024]
[3] https://docs.px4.io/main/en/dev_setup/dev_env_linux_ubuntu [cit. 24/05/2024]

6.2 Implementing custom PX4 Board configuration

Best first step would be to explain the folder structure for the board configuration files on the PX4 platform similarly as before in Table 3.1.

Folder	Description
`./`	Board root folder. Includes `firmware.prototype` file and `*.px4board` files, as well as the following folders.
`src`	Where source code for our board is located. Also includes `hw_config.h` and `bootloader_main.c` files. Includes `board_config.h` and `init.c` files. But Most importantly hosts the `CMakeLists.txt` file.
`cmake`	Any bonus `*.cmake` files. For example, we can use it to create the uploader target.
`nuttx-config`	Holds the NuttX specific configuration folders.
`nuttx-config/bootloader`	`defconfig` file for the bootloader.
`nuttx-config/nsh`	`defconfig` file for the px4 system.
`nuttx-config/include`	Clock and pinout configurations in `board.h` and dma configuration in `board_dma_map.h` files.
`nuttx-config/scripts`	`script.ld` and `bootloader_script.ld` files that define the memory structure of the MCU.

Table 6.1. PX4 new Board folder structure.

We have the root Vendor folder, in our case named `ctusr`. Then we have the root Board folder, we will name it `cimrman-v1`. The Table 6.1 explains how the folder structure inside `/boards/ctusr/cimrman-v1/` works.

```
{
    "board_id": 601,
    "magic": "CTUSRV1",
    "description": "Firmware for the FC Cimrman PX4 V1 boards",
    "version": "0.1",
    "image_maxsize": 1966080
}
```

The first important file is the **firmware.prototype** located right in the root Board folder. It defines some of the important constants used by the PX4 system and the bootloader. The `board_id` should be a unique integer identifier for our board. It will be used by the firmware uploader script when communicating with bootloader to recognize the board. Before uploading, the new firmware size will be first checked against the `image_maxsize` value, which is denoted in bytes. The file can also include other non-critical values such as `magic`, `description`, `version` or `git_identity`.

6.2.1 Building the PX4 Bootloader

To build the PX4 bootloader for the STM32H7 MCU series, the newer PX4 FMUv6 versions of firmware provide the bootloader inside the main PX4 repository[10]. For older STM32 MCUs, mainly the F series the PX4 provides a standalone bootloader build repository[11]. In this work we won't be going over the standalone bootloader and will instead use the one provided inside the main project repository.

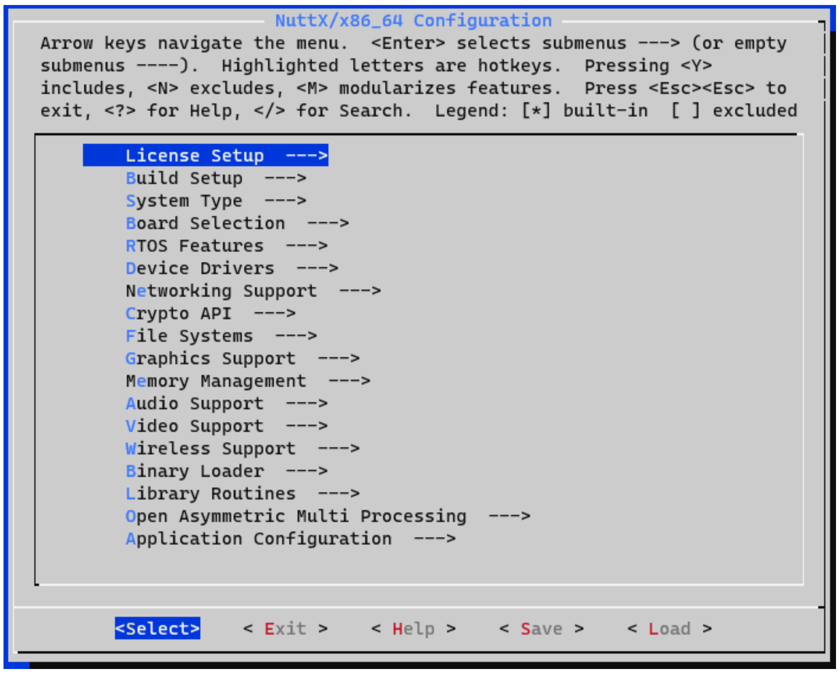
For the build to successfully complete, we needed to create and properly configure these files: Inside the `nuttx-config` folder the `bootloader/defconfig`, `scripts/bootloder_script.ld` and `include/board.h` files. Inside the `src` folder the `hw_config.h`, `board_config.h`, `bootloader_main.c` and `CMakeLists.txt`. And inside the board root folder the `bootloader.px4board` file and the previously mentioned `firmware.prototype` file.

Starting with the nuttx-config folder and NuttX configuration, we are provided with a handy terminal GUI tool, menuconfig, which will help configure for us the defconfig files, so we don't have to make them manually.

Just write this command into the console,

```
make ctusr_cimrman-v1_bootloader menuconfig
```

and the menuconfig GUI will popup.



```
                    NuttX/x86_64 Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded

          License Setup  --->
          Build Setup  --->
          System Type  --->
          Board Selection  --->
          RTOS Features  --->
          Device Drivers  --->
          Networking Support  --->
          Crypto API  --->
          File Systems  --->
          Graphics Support  --->
          Memory Management  --->
          Audio Support  --->
          Video Support  --->
          Wireless Support  --->
          Binary Loader  --->
          Library Routines  --->
          Open Asymmetric Multi Processing  --->
          Application Configuration  --->

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 6.1. NuttX RTOS menuconfig gui tool for easy configuration of defconfig files.

With this tool we easily selected and configured the correct board, in this case its was a custom board with path going to our folder `/ctusr/cimrman-v1/nuttx-config`, and the arm chip as our `stm32h743vi`. We enabled the desired peripherals, in our case usart3, which will be used for flashing the firmware. Its also desireable to configure the entrypoint to the `bootloader_main` file. We won't go deeper into explaining the precise sub menus and configurations as it would go over the scope of this work, but for anyone feeling lost, there are great videos on the NuttX youtube channel[4], that go over navigating and configuring with the menuconfig tool.

[4] https://www.youtube.com/@nuttxchannel [cit. 24/05/2024]

Next step was to provide correct mapping of the memory space and the clock configuration. Starting with the memory mapping, inside the datasheet for our MCU we find the Memory mapping in chapter 4 [5]. The table outlines each memory section starting point and size. We copied a `bootloader_script.ld` file inside `nuttx-config/scripts` directory from another already preconfigured board inside the PX4 project with the same STM32H7 chip and modified the starting section like this, based on the values from our datasheet.

```
MEMORY
{
   FLASH     (rx) : ORIGIN = 0x08000000, LENGTH = 2048K
   DTCM1    (rwx) : ORIGIN = 0x20000000, LENGTH =   64K
   DTCM2    (rwx) : ORIGIN = 0x20010000, LENGTH =   64K

   ITCM_RAM (rwx) : ORIGIN = 0x00000000, LENGTH =   64K
   DTCM_RAM (rwx) : ORIGIN = 0x20000000, LENGTH =  128K
   AXI_SRAM (rwx) : ORIGIN = 0x24000000, LENGTH =  512K

   SRAM1    (rwx) : ORIGIN = 0x30000000, LENGTH =  128K
   SRAM2    (rwx) : ORIGIN = 0x30020000, LENGTH =  128K
   SRAM3    (rwx) : ORIGIN = 0x30040000, LENGTH =   32K
   SRAM4    (rwx) : ORIGIN = 0x38000000, LENGTH =   64K
   BKPRAM   (rwx) : ORIGIN = 0x38800000, LENGTH =    4K
}
```

For the correct clock configuration we got help from the STM32CubeMX program. We used it previously for easily mapping our pinout in Section 4.2.2, but it can also be used as a handy clock configurator. Again we copy a `board.h` file from another preconfigured board PX4 project with the same STM32H7 chip as our template and modify it. This file includes definitions for configuring the correct input frequencies and clock prescaler and divider values. We have an external crystal clock source running on 8MHz and we want our `SYSCLK` to utilize the fastest possible speed for our MCU, that is 480MHz. We input these values into the STM32CubeMX Clock Configuration window and run the auto-solver, that will provide us with everything we need for the rest of the values. Then we can modify the `board.h` configuration with our values, as shown bellow on some of the important excerpted lines from the file.

```
#define STM32_BOARD_XTAL         8000000ul
#define STM32_HSE_FREQUENCY      STM32_BOARD_XTAL
...
#define STM32_BOARD_USEHSE
#define STM32_PLLCFG_PLLSRC       RCC_PLLCKSELR_PLLSRC_HSE
...
#define STM32_PLLCFG_PLL1M        RCC_PLLCKSELR_DIVM1(1)
#define STM32_PLLCFG_PLL1N        RCC_PLL1DIVR_N1(120)
#define STM32_PLLCFG_PLL1P        RCC_PLL1DIVR_P1(2)
...
#define STM32_VCO1_FREQUENCY      ((STM32_HSE_FREQUENCY / 1) * 120)
#define STM32_PLL1P_FREQUENCY     (STM32_VCO1_FREQUENCY / 2)
```

[5] https://www.st.com/resource/en/datasheet/stm32h743vi.pdf [cit. 24/05/2024]

In the same `board.h` file we also have definitions for our pinnout mapping. For the bootloader we only utilise this to specify our USART ports, that will be used for programming and flashing our firmware later. The configurations in this file are not for bootloader built only, but are shared with the PX4 system build as well.

```
#define GPIO_USART3_RX GPIO_USART3_RX_3 /* PD9 */
#define GPIO_USART3_TX GPIO_USART3_TX_3 /* PD8 */
```

Next continues probably the most important file in the whole bootloader configuration. The `hw_config.h`, in which we defined the primary constants used by the main bootloader file. This file includes following important values:

```
#define APP_LOAD_ADDRESS                0x08020000
#define BOOTLOADER_DELAY                3000 // 3s

#define INTERFACE_USART                 1
#define INTERFACE_USART_CONFIG          "/dev/ttyS0,115200"

#define BOARD_TYPE                      601

// Flash memory size register address Reference Manual: Page 3292
#define _FLASH_KBYTES                   (*(uint32_t *)0x1FF1E880)
#define BOARD_FLASH_SECTORS             (15) // Without the zero 0 sector
#define BOARD_FLASH_SIZE                (_FLASH_KBYTES * 1024)
```

Starting with the `APP_LOAD_ADDRESS` which defines the point in memory where the bootloader will be flashing the uploaded firmware. Next we have `BOOTLOADER_DELAY`, which specifies the amount of time before the bootloader process jumps to initialize the application code, and will therefore no longer listen for start upload commands. `INTERFACE_USART` and `INTERFACE_USART_CONFIG` specify that we will be using usart as our primary connection interface and configure NuttX device driver path and baudrate. If you remember we specified the board_type value inside the `firmware.prototype` file previously. The `BOARD_TYPE` value here needs to match, otherwise the upload script would not identify the board correctly. Last three definitions specify the size and amount of our flash sectors on the chip at runtime.

The `board_config.h` is mainly for definitions used by the PX4 system and the application code. We will talk about it later, but for now, the bootloader only needs this piece of code in it to build successfully.

```
#include <px4_platform_common/px4_config.h>
#include <nuttx/compiler.h>
__BEGIN_DECLS
#ifndef __ASSEMBLY__
extern void board_peripheral_reset(int ms);
#include <px4_platform_common/board_common.h>
#endif /* __ASSEMBLY__ */
__END_DECLS
```

The `bootloader_main.c` needs to define a few functions for the bootloader to run. Mainly the `board_timerhook` function that calls the sys_tick_handler. And then it has to export the 3 functions, `board_app_initialize`, `stm32_boardinitialize`, and `board_on_reset`, which do not need to really do anything, only exist.

```
extern void sys_tick_handler(void);
void board_timerhook(void) {sys_tick_handler();}
__EXPORT void board_on_reset(int status) {}
__EXPORT void stm32_boardinitialize(void) {}
__EXPORT int board_app_initialize(uintptr_t arg) {return 0;}
```

And lastly, to actually have our boards custom code built, we added our bootloader code to the build order inside the CMakeLists.txt like this:

```
if("${PX4_BOARD_LABEL}" STREQUAL "bootloader")
add_library(drivers_board bootloader_main.c)
target_link_libraries(drivers_board
PRIVATE nuttx_arch nuttx_drivers bootloader)
target_include_directories(drivers_board
PRIVATE ${PX4_SOURCE_DIR}/platforms/nuttx/src/bootloader/common)
endif()
```

Finally the bootloader.px4board file is used during the build by cmake to enable, disable and configure different aspects of the PX4 application based on current needs. For the bootloader we only need to specify used toolchain, chip architecture and possibly a custom filesystem from the ROMFS folder.

```
CONFIG_BOARD_TOOLCHAIN="arm-none-eabi"
CONFIG_BOARD_ARCHITECTURE="cortex-m7"
CONFIG_BOARD_ROMFSROOT=""
```

Now that we have provided configuration for the NuttX RTOS in a form of correct memory mapping, clock definitions and `defconfig` file, and also specified the needed definitions for our bootloader inside `hw_config.h` and `bootloader_main.c` we should be able to successfully build it using this command in the terminal

```
make ctusr_cimrman-v1_bootloader
```

After the build finishes, it will print out the Used Size by the binary file for each specified Memory region, the most important one being the FLASH, and creates the .bin and .elf files in the build folder for the project tree. Thanks to the memory summary we can easily see the remaining size of flash we have onboard the MCU.

```
Memory region         Used Size  Region Size  %age Used
           FLASH:       32140 B         2 MB      1.53%
           DTCM1:           0 GB        64 KB      0.00%
           DTCM2:           0 GB        64 KB      0.00%
        ITCM_RAM:           0 GB        64 KB      0.00%
        DTCM_RAM:           0 GB       128 KB      0.00%
        AXI_SRAM:        4932 B        512 KB      0.94%
           SRAM1:           0 GB       128 KB      0.00%
            ...
```

Now we flash our binary file on to the board using an ST-Link debugger[6] connected via the SWD pins. During the process we provide power to the board using the USB-C port. In the next step we will show how to configure and build the PX4 application, as well as upload it to the board using our bootloader.

[6] https://www.st.com/en/development-tools/st-link-v2 [cit. 24/05/2024]

6.2.2 Building the PX4 Application

The mapping of our memory space that we have done for the bootloader in `bootloader_script.ld` can be copied over to `script.ld` for our PX4 application. The important difference is that our FLASH memory for the application starts at 0x08020000 as the first sector is occupied by the bootloader.

Now comes to the `defconfig` file. We again use the menuconfig tool so we can easily configure it correctly by just navigating a GUI. The important distinctions from the bootloaders defconfig file are these:

- Switching init entrypoint to `nsh_main`
- Enabling Timer (CONFIG_STM32H7_TIM*) and RTC (CONFIG_STM32H7_RTC*)
- Enabling the NuttX Task Scheduling interface (CONFIG_SHED_* declarations)
- Enabling and Configuring all needed peripherals, such as UARTs, SPIs, I2Cs and MMCSD SDIO.
- Enabling the NuttX console shell (CONFIG_NSH_*) and setting one UART as our serial console.
- Enabling the NuttX FAT filesystem features (CONFIG_FS_*, CONFIG_FAT_* declarations)

Again, if you are not sure how to navigate the menuconfig, refer to the link mentioned in Section 6.2.1.

We used the `board.h` file to setup our clock configuration. Now we will also add at the end of this file the definitions for the peripheral pinout we have enabled in the defconfig file, for our UARTs, SPIs and SDMMC peripherals.

```
/* SDMMC */
#define GPIO_SDMMC1_D0 GPIO_SDMMC1_D0_0 /* PC8 */
#define GPIO_SDMMC1_D1 GPIO_SDMMC1_D1_0 /* PC9 */
#define GPIO_SDMMC1_D2 GPIO_SDMMC1_D2_0 /* PC10 */
#define GPIO_SDMMC1_D3 GPIO_SDMMC1_D3_0 /* PC11 */
#define GPIO_SDMMC1_CK GPIO_SDMMC1_CK_0 /* PC12 */
#define GPIO_SDMMC1_CMD GPIO_SDMMC1_CMD_0 /* PD2 */
/* UART/USART */
#define GPIO_USART3_RX GPIO_USART3_RX_3 /* PD9 */
#define GPIO_USART3_TX GPIO_USART3_TX_3 /* PD8 */
/* SPI */
#define ADJ_SLEW_RATE(p) (((p) & ~GPIO_SPEED_MASK) | (GPIO_SPEED_2MHz))
#define GPIO_SPI1_SCK    ADJ_SLEW_RATE(GPIO_SPI1_SCK_1) /* PA5  */
#define GPIO_SPI1_MISO   GPIO_SPI1_MISO_1               /* PA6  */
#define GPIO_SPI1_MOSI   GPIO_SPI1_MOSI_1               /* PA7  */
#define GPIO_SPI2_SCK    ADJ_SLEW_RATE(GPIO_SPI2_SCK_4) /* PB13 */
#define GPIO_SPI2_MISO   GPIO_SPI2_MISO_1               /* PB14 */
#define GPIO_SPI2_MOSI   GPIO_SPI2_MOSI_1               /* PB15 */
```

We find the correct values in the predefined pinmap files provided by the NuttX platform for the STM32H7x3 MCUs, and select them based on our pinout, which we specified using the STM32CubeMX tool in Chapter 4. The two SPI buses will be used for our IMU and barometric sensor. The USART will provide an additional connection for our MCU, and the SDMMC peripheral will be used in a 4-bit SDIO mode for writing to and reading from an SD card.

Continuing with the previously mysterious `board_config.h` file in the src directory, it will now be useful to define the number of constants needed by the PX4 application to initialise drivers for our peripherals, as well as providing definitions for external initialisation functions.

```
/* SDIO/SDMMC */
#define SDIO_SLOTNO                     0
#define SDIO_MINOR                      0
/* BOARD INTERFACES */
#define BOARD_ENABLE_CONSOLE_BUFFER
#define BOARD_USB_VBUS_SENSE_DISABLED   1
#define BOARD_NUM_IO_TIMERS             1

__BEGIN_DECLS
#ifndef __ASSEMBLY__
extern int stm32_sdio_initialize(void);
extern void stm32_spiinitialize(void);
#endif /* __ASSEMBLY__ */
__END_DECLS
```

Now that we have configured all the required constants for the NuttX and PX4 application, we can move to the implementation. Starting with the `init.c` file in the `src` directory, we call the external initialization functions for our drivers, inside the `board_app_initialize` function, such as `stm32_spiinitialize()` or `stm32_sdio_initialize()`. This will initialize register the NuttX device drivers, which will then appear as devices in the filesystem under `/dev`. So if you have problem finding your devices, try to verify that you have done all these steps correctly, before moving to hardware debugging. Most of the times it will be an incorrectly configured software problem. The other *.c and *.cpp file inside the `src` directory will be used to provide constexpr arrays where we map which drivers should the PX4 system use for the specified peripherals, like for example our sensors on SPI buses:

```
constexpr px4_spi_bus_t px4_spi_buses[SPI_BUS_MAX_BUS_ITEMS] = {
    initSPIBus(SPI::Bus::SPI1, {
initSPIDevice(DRV_IMU_DEVTYPE_ICM20948, SPI::CS{GPIO::PortA, GPIO::Pin4})
    }),
    initSPIBus(SPI::Bus::SPI2, {
initSPIDevice(DRV_BARO_DEVTYPE_MS5611, SPI::CS{GPIO::PortD, GPIO::Pin10})
    })
};
static constexpr bool unused = validateSPIConfig(px4_spi_buses);
```

The last file inside the `src` directory we will mention is the `CMakeList.txt` file. In here similarly to when we added the bootloader code to the build order, we will now append the if statement with this code for our PX4 application code.

```
add_library(drivers_board
   init.c spi.cpp sdio.c)
add_dependencies(drivers_board arch_board_hw_info)
target_link_libraries(drivers_board
    PRIVATE arch_io_pins arch_spi
    arch_board_hw_info nuttx_arch nuttx_drivers px4_layer)
endif()
```

Now that we have configured and initialised the necessary peripherals for our PX4 application, there is one last thing we need to do. We have to tell the PX4 system to actually build and use these drivers and modules. This is done in the `default.px4board` file in the board root directory, similar to the bootloader.px4board file.

```
CONFIG_BOARD_TOOLCHAIN="arm-none-eabi"
CONFIG_BOARD_ARCHITECTURE="cortex-m7"
CONFIG_BOARD_ROMFSROOT="ctusr_common"
# PX4 TASK SPAWN interface for modules
CONFIG_MODULES_LOAD_MON=y
# Rocket commander and Mavlink
CONFIG_MODULES_ROCKET_COMMANDER=y
CONFIG_MODULES_MAVLINK=y
CONFIG_MAVLINK_DIALECT="rocket"
# Sensor Drivers
CONFIG_DRIVERS_IMU_INVENSENSE_ICM20948=y
CONFIG_DRIVERS_BAROMETER_MS5611=y
# System commands
CONFIG_SYSTEMCMDS_TOPIC_LISTENER=y
CONFIG_SYSTEMCMDS_UORB=y
CONFIG_SYSTEMCMDS_PARAM=y
```

We tell the PX4 system that we want to use our `ctusr_common` ROMFS, which we will talk about more in the next section. Then we activate the `load_mon` module, which is a primary PX4 interface for creating and managing other modules, which we will need for our custom `Rocket Commander` module, which we can activate in a similar way to any other module. We also enable and configure the `mavlink` module with our rocket dialect. Next we specify our sensor drivers for the IMU and barometer, which have already been implemented by the PX4 community. Finally, we enable some system commands that can be used by the init script at boot time or directly in the nsh console.

6.2.3 Startup Process and Init scripts

After successfully starting the PX4 application, it will search for and execute the initialisation scripts within the NuttX RTOS file system. Custom initialisation scripts can be created in the `ROMFS` folder, which are then included in the binary and made accessible in the MCU's pseudo file system when NuttX RTOS is booted. These scripts can contain commands that are automatically executed in the NuttX shell on startup, as if they had been entered manually.

We create a folder with a custom name in the ROMFS directory; in our case it will be called `ctusr_common`. This folder will contain only one file, CMakeLists.txt, which will be used to include all subsequent subdirectories. The convention is to name the folders within our ROMFS directory init.d and init.d-posix, distinguishing them by the platform they are built for. Inside each init.d directory we have our initialisation scripts and a CMakeLists.txt file which uses the `px4_add_romfs_files()` function to add them to the build. The individual init scripts are just plain text files with no extension, containing a list of commands to be executed in a sequence at startup. You can also call other scripts from the file and create a hierarchical initialisation of the subsystems you need. For example, we use this to initialise the mavlink module, which has its own separate init script. All we need to do to make our board use our new ROMFS is to specify it in the *.px4board file.

This is a simplified version of what our init script looks like.

```
ver all
set RC /etc/init.d/
set FRC /fs/microsd/etc/rc
mount -t vfat /dev/mmcsd0 /fs/microsd
if [ -f $FRC ]
then
   . $FRC
else
    rocket_commander start
    rocket_commander launch
fi

. ${RC}rc.mavlink
unset FRC
unset RC
mavlink boot_complete
```

You might see, that we can also launch init scripts from a removeable media such as the SD card. In this way we can easily modify the startup behaviour of our flight computer without the need of reprogramming or flashing new firmware.

6.2.4 Flashing software

After connecting the board to a PC via the configured port in the bootloader, we should see it in our connected devices. Then by executing

```
make ctusr_cimrman-v1_default upload
```

the PX4 application will get built, and the upload script will run right after. It will try to find the USB device and communicate with the bootloader to flash the application binary.

Once the application is successfully flashed and running, we can now connect to the peripheral we chose in the configuration as our main serial console, and write to it using a program for serial communication of your own choosing. It is also possible to use the `mavlink_shell.py` script under Tools and connect through a serial port that is being used for communication over MAVLink messages.

6.3 Flight Commander Module Implementation

The implementation of a custom PX4 module is simple. We followed the example of other modules that already exist. We created a directory inside `src/modules` named `rocket_commander` for our new flight commander module. Inside this directory we can create files and sub directories with our source code. To register our module with the PX4 system we just need to create 3 simple files. The `module.yaml` file, which just specifies the module name, in our case it has just this one line.

```
module_name: Rocket Commander
```

Then we have a `Kconfig` file, which is used to specify any config constants that if defined inside any board *.px4board file can enable parts of our module. We will only have one constant that will be used to enable our module.

The content of the Kconfig file looks like this.

```
menuconfig MODULES_ROCKET_COMMANDER
   bool "rocket_commander"
   default n
```

Lastly we have our `CMakeLists.txt` in which we specify which source files and dependencies should be built for our module during the build process.

```
px4_add_module(
MODULE modules__rocket_commander
MAIN rocket_commander
SRCS
   RocketCommander.cpp
MODULE_CONFIG
   module.yaml
)
```

Our code is written in c++ and includes just a single class with the name of Rocket Commander which extends the public ModuleBase and ModuleParams classes from the PX4 platform code. Each module has to therefore implement functions for their instantiation, run cycle and handling commands.

The purpose of the instantiation function is to use the `px4_task_spawn_cmd` function from the `load_mon` module mentioned at end of Section 6.2.2, and to create an instance of the module class. Then we have several functions for sending and handling commands that can be sent to our module from the nsh shell, or even from other modules. We can also specify our own custom commands that can trigger a function call. Finally, we have the actual module run cycle function, which is the body of the spawned task. Inside the run function we almost always want to have a while cycle where we repeatedly process the work of our module until the module class instance is destroyed.

The inter-module communication heavily rests on the subscription and publication of uORB messages as mentioned in Section 3.5. We also Subscribe to multiple uORB message topics. These include `vehicle_imu` and `vehicle_odometry` messages which provide us with current information about the vehicle position, angle and linear and angular velocity. As we are required to handle commands sent to our module, we also subscribe to `vehicle_command` uORB topic, which is used for this purpose.

The published topics again include the `vehicle_command` and `vehicle_command_ack` uORB messages, which are needed because we can send commands to modules and need to acknowledge that we have received commands from other modules. But for us, the important uORB topic that we will be publishing is the `rocket_state` message. This is the message that will contain the current rocket and parachute state, and will also be sent to external systems using MAVLink messages over a serial connection.

Now for our rocket commander module to fulfil the tasks of a state machine, we use enums to define our states, as designed in chapter 5. We declare two variables that hold the current state of both the rocket and the parachute. We subscribe to the uORB messages for `vehicle_odometry` and `vehicle_imu`. During the run cycle, we evaluate whether the values have changed and update our internal state.

We than have two switch case structures that handle behaviour based on the current rocket and parachute states. At the end of the run cycle, if there was a change in the current state, this is immediately published to our `rocket_state` uORB message topic to notify any interested subscribers. For the actual code of our module you can refer to the included PX4Project.zip file under 9.

Chapter 7
Hardware and Software Verification

7.1 Flight Commander HIL Test

Hardware-in-the-Loop (HIL) testing is crucial in the development and validation as we can simulate real-world conditions and record the behaviour of our software and hardware before deployment.

For our HIL testing we use a 1D rocket flight simulation made in Simulink program, that was created and is currently used by the CTU SR team for rocket flight state machine validation. It will communicate with the on-board MCU via a USB serial connection. The communication protocol uses MAVLink messages from our custom dialect, that includes information about the current state of the rocket and parachutes. The Simulink program and MAVLink protocol are described in Section 2.2. as part of the requirements. Our implementation of the serial communication with the MAVLink messages is then shown in Section 5.2.

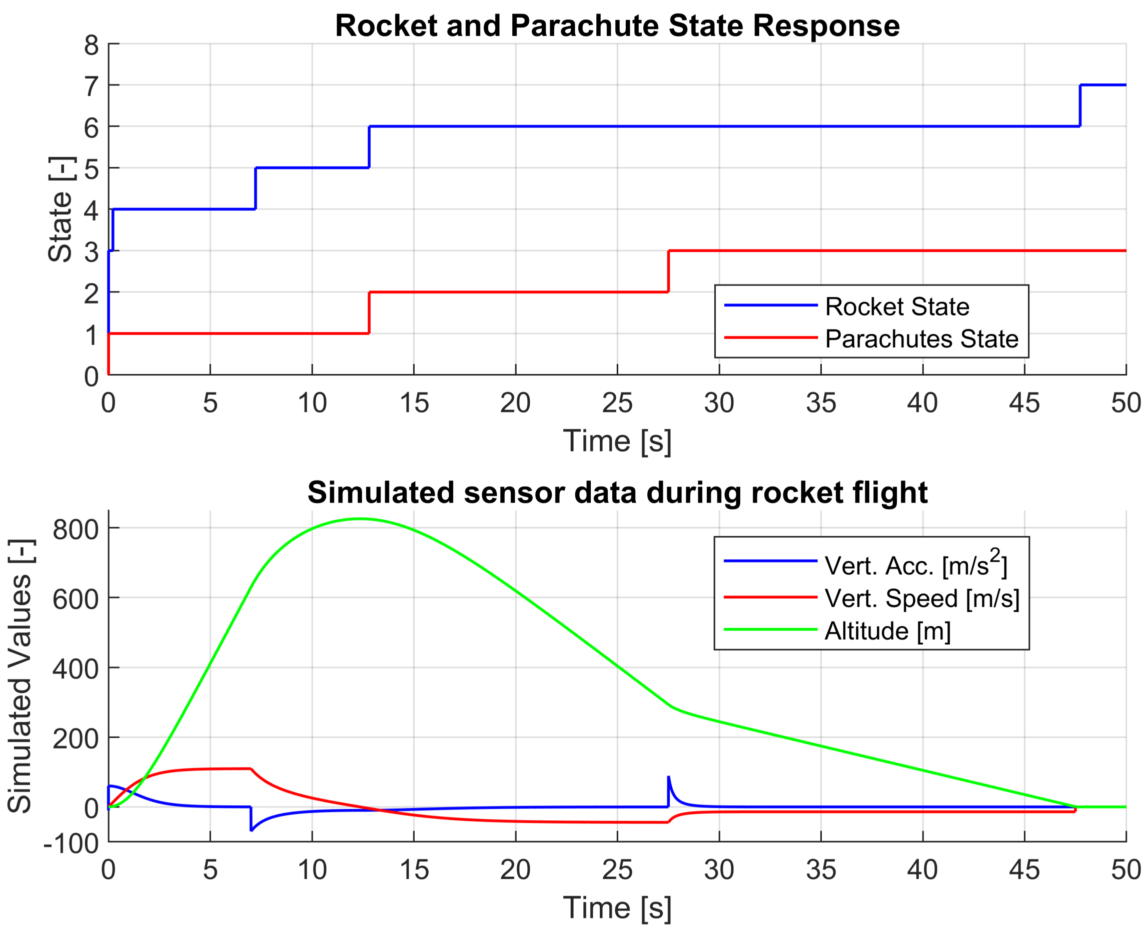


Figure 7.1. Graphs of simulated sensor data and rocket state response from the HIL test.

Before the HIL testing the module was moved to the `ENGINE_START` (3) rocket state where it awaited data from the simulated sensors. After launching the Simulink simulation, it started sending sensor data to the flight computer, where they were picked up by our rocket commander module, which successfully identified the simulated liftoff and switched to the `POWERED_ASCENT` (4) rocket state.

At the 6 second mark, the simulator stopped the engine thrust which can be seen by the sudden drop in acceleration. Our module was fast to react and switched the state to the `UNPOWERED_ASCENT` (5) phase. It then monitored closely for the apogee and beginning of descent, which it detected precisely, and changed the rocket state to `DESCENT` (6), as well as switched the parachute state to `DROGUE` (2).

Then during the descent it detected passing the preset 300m threshold in altitude and switched the parachute state to `MAIN` (3), deploying the main parachute. This was also detected by the simulation, which modified the descent speed accordingly. The last mark was at about 47 seconds from launch when the rocket commander module successfully detected the simulated landing and switched its state ti `LANDED` (7).

We repeated the HIL test multiple times, always getting the same results, verifying the reliability of our implementation. From these results we concluded, that the main branch of our state machine, which we designed and implemented in our rocket commander module, was successfully verified by the HIL testing as can be seen on the Figure 7.1.

7.2 IMU and Baro Allan deviation verification

The IMU and barometer are the most critical sensor systems inside a rocket. We therefore decided to analyse the noise of these sensors. This will hopefully show the correct functioning of our sensors on board and could also be used to develop an accurate sensor simulation in later work.

To show that our sensors are working properly, we did a little experiment. We measured the data from our sensors over a long period of time, two hours to be precise, at a frequency of 250Hz to get about two million samples for each sensor. We then used the Allan deviation test to determine their Bias Instability during prolonged measurement.

Allan deviation was originally derived to measure the long-term stability and noise characteristics of clock oscillators. This was later transferred to also analyse the noise and stability of sensors. All sensors, regardless of quality, will have a bias in their measurements. The short-term effects of this are called bias instability and show random fluctuations in the sensor output. But bias can also change slowly over time due to external factors, this is called bias drift and is a long term indicator of sensor stability.

Sensor	**Avg. Deviation** $\sigma(\tau)$	**Bias Instability**
Accelerometer $[m/s^2]$	$5.67 \cdot 10^{-5}$	$2.04 \cdot 10^{-1}$
Gyroscope $[rad/s^2]$	$1.90 \cdot 10^{-6}$	$6.84 \cdot 10^{-3}$
Barometer $[mBar]$	$2.19 \cdot 10^{-4}$	$7.88 \cdot 10^{-1}$

Table 7.1. Allan Deviation Sensors Bias Instability over 1 hour period.

From the measured sensor data, we have calculated allan variation values using the `alanvar` function in Matlab, and by applying root square on the result we got the Allan Deviation graphs. Allan Deviation graphs compare the change in sensor measurement

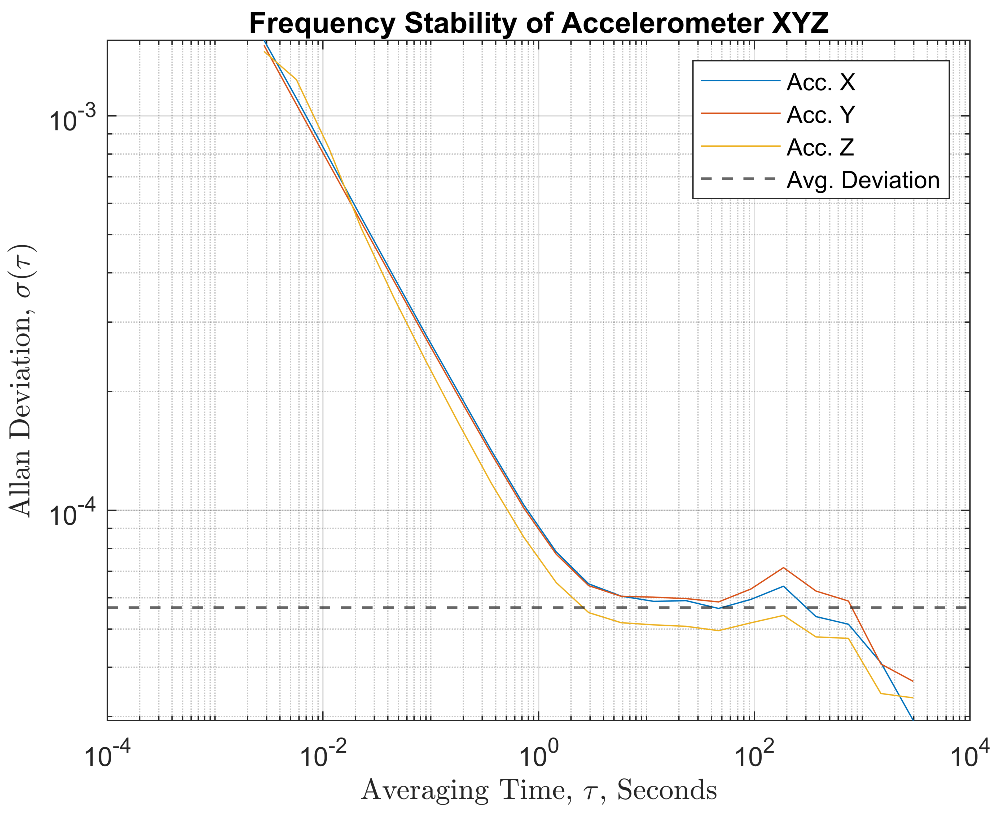


Figure 7.2. Graph of Allan Deviations for Accelerometer values.

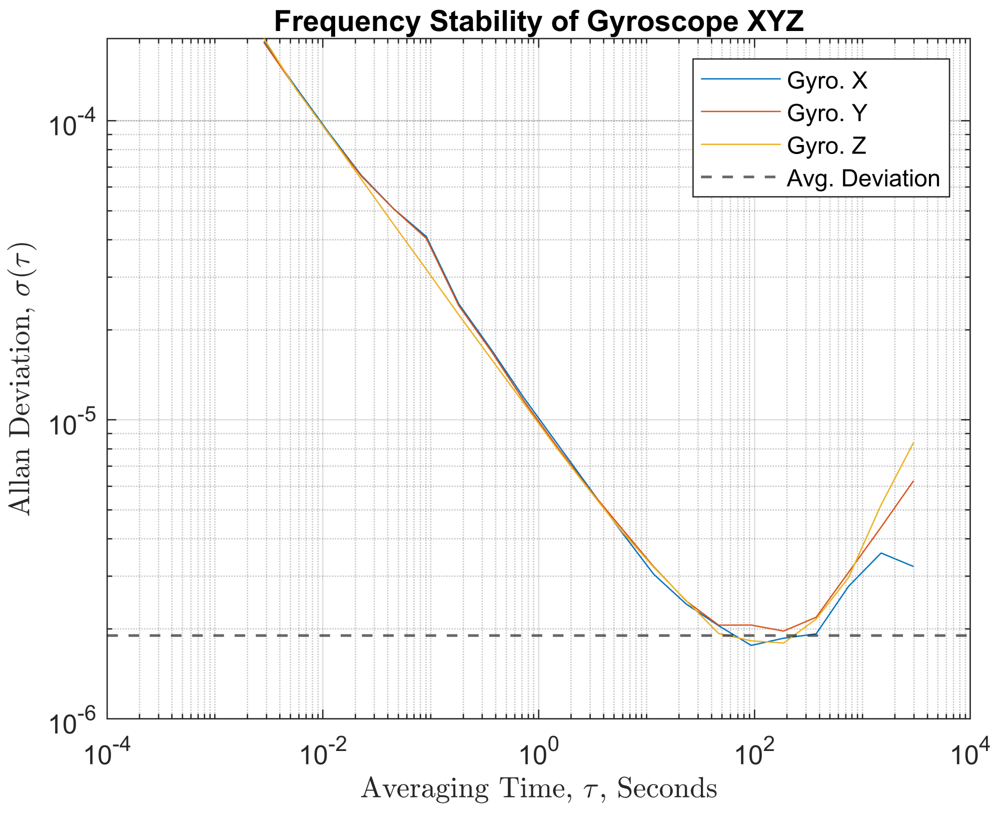


Figure 7.3. Graph of Allan Deviations for Gyroscope values.

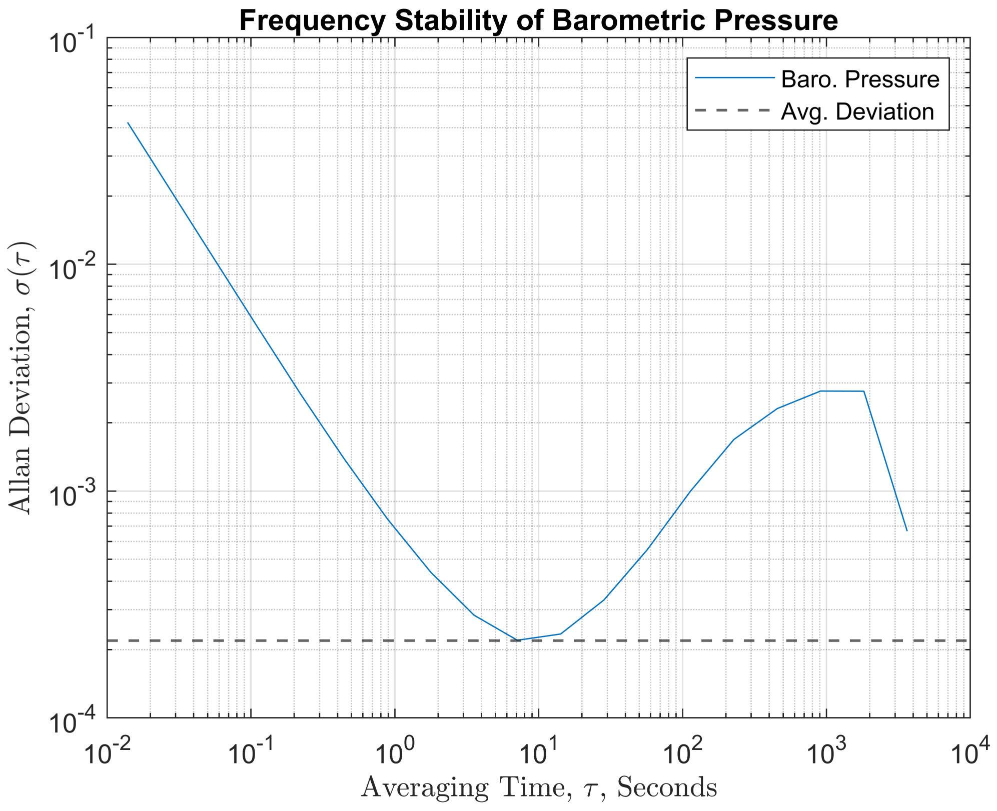


Figure 7.4. Graph of Allan Deviations for Barometric values.

stability over different time periods. We have calculated the overall short-term Bias Instability for each sensor and presented it in Table 7.1. The Avg Deviation values were taken at the curves minimum, where it is flat and its derivative is close to zero. We have then multiplied this value by 3600 seconds to figure out the average change over a period of one hour.

We have demonstrated that we can continuously measure data from our IMU and Baro sensors over long periods of time. The impact of the measured IMU sensor bias instability and inaccuracy is insignificant in the context of the high performance rocket flight mission, where the flight time from launch to landing is typically 10 minutes or less. The bias instability of the barometric sensor is at least two orders of magnitude greater than that of the IMU, and this change could be reflected as a difference of a around one meter in the measured altitude over the flight time. However, this could also have been caused by inaccuracies in the measurement, as we did not have the proper equipment to completely isolate the board from the environment and it was therefore affected by the constant change in atmospheric pressure. In future work, further analysis and proper filtering of the barometric sensor could be carried out to improve the quality of this measurement.

The data used for the Allan Variance test were gathered on to an sd card via the SDMMC Peripheral. We created a simple module, just for recording the sensor data in an appropriate format, for quick importing over to matlab and further analysis. This module used the underlying NuttX RTOS filesystem interface for opening, writing and closing files on the sd card. In its run cycle it detected when a sensor pushed a new updated data, and wrote them directly to a file on the sd card. In this way, we have also successfully verified the proper functioning of our on board SD card peripheral implementation.

Chapter 8
Results

A working Printed Circuit Board (PCB) was successfully designed and assembled for our flight computer that met requirements set in the Chapter 2 as shown in Chapters 4. We then designed our flight commander module in Chapter 5, successfully implemented it in Chapter 6 and verified our design with a Hardware-in-the-Loop test in Chapter 7.

The board uses a state of the art STM32H743VI microchip with 2MB of Flash memory and runs the PX4 and NuttX RTOS stack. It includes a barometric sensor and a 9-axis IMU with accelerometer, gyroscope and magnetometer, each connected to the MCU via a dedicated SPI bus. The board features drill holes in the same arrangement as those on the Expander board used by the previous generation of the Cimrman flight computer as specified in the requirements. Next the board features both an eeprom on dedicated I2C bus, and an SD card, connected via a SDMMC SDIO interface in 4-bit mode configuration, as storage enhancing solutions. The components were chosen based on our requirements in Chapter 2, and the ease of implementation, taking into account their support by NuttX RTOS and PX4 system.

In accordance with the requirements, a connection to other boards inside the rocket can be assured by the two RocketBus connectors, which the board also used as its primary source of power. The board includes a CAN FD transceiver for facilitating the communication between connected systems on the RocketBus. The board provides an easy connection for debugging and testing via an USB-C port, that might also be selected as a power source. As a bonus, the board features solder pads for possible external battery connection as a third power source option in standalone solutions. Peripherals include at least 4 UART connectors, mainly for linking with Telemetry boards. One external I2C bus connector, one SPI bus with chip select pin and a multitude of GPIO pinouts for additional ADC, PWM or just simple digital I/O capabilities.

As the software solution, a PX4 and NuttX RTOS combination was chosen for its adaptability and ease of future extensibility. These were introduced in Chapter 3, and the process of configuration for our board was detailed in Chapter 6.

In order to demonstrate the proper functionality of the on-board sensors, we created a module capable of recording sensor data to an SD card over extended period of time. Using the gathered data, we analyzed the bias instability of the sensors shown in Section 7.2, confirming that they were adequate for the requirements of the problem.

Finally, a flight commander module was designed as shown in Chapter 5, with a robust state machine for handling the behaviour of a high-power rocket during different flight phases. Subsequently, our implementation, described in Section 6.3, was successfully tested and verified, with the provided Simulink simulation of a 1D rocket flight by the CTU SR team described in Section 2.2, using Hardware-in-the-Loop testing as shown in Chapter 7.

References

[1] Zapadlo, Aleš. *Flight Computer for Small Rocket*. ČVUT, 2023 [cit. 24/05/2024]. Bachelor's Thesis. Available from `http://hdl.handle.net/10467/108621/`.

[2] Dronecode. *MAVLink Developer Guide*. [cit. 24/05/2024]. Available from `https://mavlink.io/en/`.

[3] *"About draw.io"*. [cit. 24/05/2024]. Available from `https://www.drawio.com/about/`.

[4] *"About KiCad"*. [cit. 24/05/2024]. Available from `https://www.kicad.org/about/kicad/`.

[5] *"What is MATLAB?"*. [cit. 24/05/2024]. Available from `https://www.mathworks.com/discovery/what-is-matlab/`.

[6] *"Simulink: Design. Simulate. Deploy."*. [cit. 24/05/2024]. Available from `https://www.mathworks.com/products/simulink/`.

[7] Foundation, The Apache Software. *About Apache NuttX*. [cit. 24/05/2024]. Available from `https://nuttx.apache.org/docs/latest/introduction/about/`.

[8] Abbas, Asad. *Real Time Operating Systems(RTOS) For Drones | Asad Abbas*. Available from DOI 10.13140/RG.2.2.10243.35369.

[9] Dronecode. *PX4 Autopilot User Guide*. [cit. 24/05/2024]. Available from `https://docs.px4.io/main/en/`.

[10] Dronecode. *PX4 Autopilot Software*. [cit. 24/05/2024]. Available from `https://github.com/PX4/PX4-Autopilot/`.

[11] Dronecode. *PX4 Bootloader for PX4FMU, PX4IO and PX4FLOW*. [cit. 24/05/2024]. Available from `https://github.com/PX4/PX4-Bootloader/`.

Chapter 9
Appendencies

`AndromedaSystem.pdf`	Overview diagrams of the full Andromeda Rocket System.
`rocket.xml`	Custom MAVLink dialect with the Rocket State message.
`KiCadProject.zip`	KiCad project of the Schematic and PCB Layout.
`PX4Project.zip`	PX4 project with our source code and implementation.

* All files are available online for download from:
`https://github.com/Aandrej2/Adaptable-Flight-Computer-for-a-Rocket`