Rocket Flight Computer with a Raspberry Pi Pico

Dervla Gargan

Course: Immersive Software Engineering

University of Limerick

Supervisor: Dr. Mark Burkely
University *of* Limerick
Ireland

Abstract

A flight computer in a rocket is essential for its function. It is often the difference between a nominal flight and a ballistic projectile falling back to the ground. This paper describes the process of designing and building a flight computer for high powered rocketry, while keeping modularity in mind. The flight computer is made using a Raspberry Pi Pico W and utilise the PIO functionality to increase modularity. A PCB design as well as a prototype board were developed to test the functionality of the flight computer. It utilise a Kalman filter for altitude and speed estimation and uses a state machine to control the deployment of the parachutes. This paper resulted in a flight computer design that can deploy two parachutes and code that can estimate the altitude and speed of a rocket.

This flight computer will be flown in rockets made by the University of Limericks High powered rocketry team in future rocket launches and competitions.

Contents

List of Figures

Glossary

AGL Above Ground Level

Apogee when a rocket has hit its maximum height, also known as when it's velocity has hit 0

ASL Above Sea Level

CircuitPython CircuitPython is a fork of MicroPython created by Adafruit.

Coasting the motor has fully burnt out, but the rocket is still ascending but slowing down, acceleration is less than $0m/s^2$.

COTS Commercial of the shelf

DPS Degrees Per Second

Drogue a small parachute deployed on or just after apogee

E-Bay lectronics bay, also known as the avionics bay, is the section on the rocket the flight computer sits in

GNSS global Navigation Satellite System, refers to any satellite constellation that provides global positioning, navigation, and timing services.

GPIO General Purpose Input Output

IMU Inertial Measurement Unit

Main a larger parachute deployed once the rocket is nearer the ground, usually between 600-200m

Micropython "MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments."

PIO Programmable Input Output

Powered Flight when the motor is actively burning, generating thrust for the rocket

SRAD Student researched and designed

1

Introduction

The goal of the project is to create a modular flight computer for high powered rockets. A flight computer records the flight data, like height and speed, as well as deploying the parachutes of the rocket. Flight computers can also provide tracking through GNSS and radio telemetry I wanted to ask the question "Is it feasible to replicate all of the functionality of a COTS Flight Computer using a Raspberry Pi Pico, while utilise the PIO feature to increase modularity?"

According to the National Association of Rocketry (2025), a High Power Rocket is defined by the type of motor used in the rocket. These motors range on a scale of A and upwards, with O being the largest that can be commercially bought. Motors are defined on this scale based on the impulse generated. Each category doubling the impulse of the motor. Generally, A-D is considered low power, E-G is mid-power and H and up is high power. This means a rocket is considered high power if the impulse is greater than 160 Newton-Seconds. National Association of Rocketry (2024)

The flight computer aims to be modular through the use of the Raspberry Pi RP2XXX Series of microcontrollers, which includes the RP2040 and RP2350. Both of these chips have programmable input/output blocks (PIO), which can be used to emulate different protocols. These blocks increase modularity as the flight computer will not be limited by the microcontrollers inbuilt protocols.

Classification	Impulse Range	Impulse Limit	Category
Model Rocket	1/8A	0.3125	Micro Power
	1/4A	0.625	Low Power
	1/2A	1.25	
	A	2.5	
	B	5	
	C	10	
	D	20	
	E	40	Mid Power
	F	80	
	G	160	
High Power	H	320	Level 1
	I	640	
	J	1280	Level 2
	K	2560	
	L	5120	
	M	10240	Level 3
	N	20480	
	O	40960	

Figure 1.1: Rocket Motor Scale, Source: National Association of Rocketry (2024)

Some of the reasons for creating a custom flight computer are:

1. Commercial off the shelf (COTS) flight computers are expensive and hard to acquire in Ireland.
   As there are no flight computer manufacturers or resellers in Ireland, ULAS HiPR have to resort to importing flight computers from America, the UK, or Switzerland. This process is expensive due to the import fees. Parts have also been delayed for significant amounts of time due to customs issues.

2. Ranking higher in rocketry competitions.
   The scoring of competitions favors student researched and designed (SRAD) systems compared to commercial off the shelf (COTS) systems. As the team has been using COTS flight computers so far, switching to SRAD will help their performance in competitions.

3. Designing for the future.
   As ULAS HiPR advances, we are looking to develop more complex rockets. This would include things like liquid rocket engines or non-explosive deployment. Both these systems require more complex software and hardware capabilities than are offered by COTS computers. Things like monitoring heat and pressure for a liquid engine would be an example of something not currently supported by COTS computers.

   PIO opens up the options for connecting different systems within the rocket as well. If ULAS HiPR wanted multiple flight computers or systems, PIO means it can communicate over protocol like CAN.

4. Limited resources or availability.
   The modular aspect of this computer means that the production of the flight computer will not be majorly impacted due to cost or electronics availability. The nature of the code and PIO functionality will mean sensors can be easily switched out for cheaper or in stock ones, no matter the protocol needed.

An example of using PIO to make the flight computer more modular would be adding a secondary GNSS sensor. GNSS sensors are at risk of not working due to antenna placement. If we wanted full redundancy for tracking we could have two sensors, each will have an antenna placed in a different location in the rocket. GNSS sensors commonly use UART as the communication protocol. The current PCB design has GP2, GP3 and GP4 exposed as breakout pins. None of these pins has full UART breakouts, but using PIO I can make GP2 and GP3 have UART functionality so the additional GNSS can be used

1.1 Objectives

The Objectives of this project are as follows:

- Altitude estimation within a $\pm 10\%$ range of COTS flight computers results
- Ability to deploy two individual parachutes
- Broadcast its location using GNSS and a telemetry system
- Breakout pins that can leverage the RP2XXX's PIO functionality

2

Literature Review

2.1 Determining Altitude

2.1.1 Barometric Altitude

To determine altitude from a barometer, the international barometric formula can be used. This is:

$altitude = 44330 * (1 - \frac{p}{p_0})^{\frac{1}{5.225}}$, where p_0 is pressure at sea level.

This is taken from the NOAA US Department of Commerce (n.d.) and is used in a popular barometer the BMP180 by Bosch Sensortec (2013). This formula assumes that temperature is 15°C and that sea level pressure is 1013.25Pa, which is the standard Sea Level Pressure. These two measurements will vary depending on time and location, so it will not return and accurate ASL (Above Sea Level) altitude, but once the first measurement is taken, it can be used to accurately determine the AGL (Above Ground Level) altitude.

2.1.2 Kalman Filter

When detecting apogee, often a barometer and accelerometer are primarily used. These sensors can be quite noisy. A barometer is prone to a transonic effect, when a rocket is travelling around Mach 1. This effect can cause the pressure to change, making the flight computer appear to descend when it is actually ascending at a

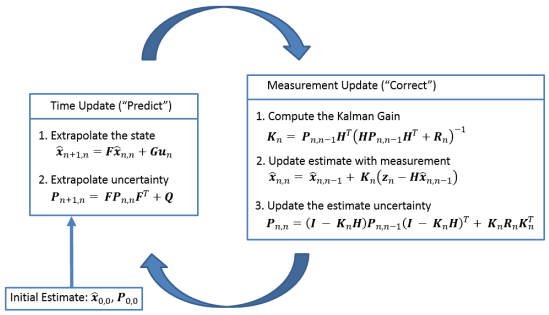

Figure 2.1: Kalman Filter, Source: Becker (n.d.)

rapid rate. In comparison, an accelerometer is not affected by this issue, but is less reliable at predicting altitude compare to a barometer. Schultz (2002)

Kalman filters have been used in rockery for decades and was used in the Apollo Program. Mcgee and Schmidt (1985) It is a recursive algorithm for state estimate. It takes in liner equations based on the system it is modelling, in the case of rocketry linear motion equations, and sensor measurements. These have an associated weight based on confidence and noise. Based on these factors, it estimates the next state of the rocket. MathWorks (2021)

A standard Kalman Filter is meant for use on linear system. A rocket is not linear due to acceleration being non-linear. For non-linear systems, an extended Kalman filter (EKF) is more accurate. The EKF uses the Jacobian values of F and H instead of just F and H in the filter. While this does increase accuracy, it is also more computationally expensive. MathWorks (n.d.)

2.2 RP2XXX Series of Microcontrollers

The Raspberry Pi Company has made 4 Pico-series Microcontrollers utilising the RP2XXX series of chips. These include the Raspberry Pi Pico, Raspberry Pi Pico W, Raspberry Pi Pico 2 and Raspberry Pi Pico 2W. The first iteration of Pico boards use the RP2040 chip, and the second use the RP2350. The W stands for wireless. Both W iterations of the Pico has the CYW43439 chip which adds Bluetooth and WiFi capabilities. The boards all use the same C/C++ SDK, so

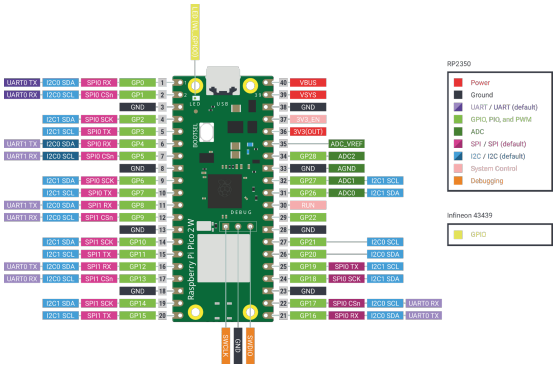

Figure 2.2: Kalman Filter, Source: Raspberry Pi (2024*b*)

code written is compatible with all the boards. All Pico boards have the identical pin-outs as well, so can be drop-in replacements for each other. Raspberry Pi (2024*b*)

All Pico's have:

- 2× SPI
- 2× I2C
- 2× UART
- 3× 12-bit 500ksps Analogue to Digital Converter (ADC)
- 24× controllable PWM channels

2.2.1 Bluetooth

On the Pico W variants, they utilise the BlueKitchen BTStack for Bluetooth communication. It is capable of Bluetooth LE (low energy) and Bluetooth classic. Raspberry Pi (2024*a*)

2.2.2 PIO

The Raspberry Pi's RP2XXX series of microcontrollers have programmable input/output blocks (PIO). The first RP2 chip, the RP2040, has 2 of these blocks, and the second version, the RP2350, has 3 blocks.

These block each have 4 state machines in them, with 2 32-bit FIFO buses per state machine to communicate with the main processor. Each PIO can execute independently and has a focus on precise timing creating a versatile hardware interface. Through the custom instruction set developed by Raspberry Pi, these PIO blocks can be used to emulate many different protocols. Raspberry Pi (2024*d*)

2.3 Basics of High Powered Rocketry

The flight of a rocket can be broken in clear stages. For a rocket flight to be considered nominal, it must go through these stages. Milligan (2024)

1. Motor Ignition - An e-match ignites the motor using an ignition system on the ground
2. Powered Flight - The motor is burning sending the rocket up
3. Coasting - The motor is finished burning, the rocket continues to ascend, acceleration is decreasing
4. Apogee - the point where rocket has hit its maximum height, at this point speed is 0m/s
5. Recovery System (Single or Dual Deployment)
   (a) Single Deployment
      i. Main Deployment - A Parachute will deploy at or just after apogee, slowing the rocket to a safe landing speed

   (a) Dual Deployment

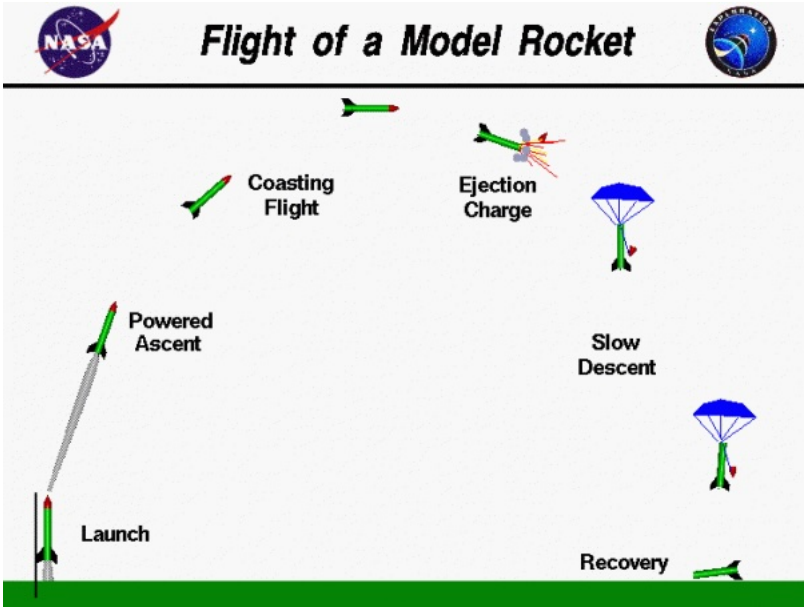

Figure 2.3: Stages of a Rocket Flight, Source: NASA (2023)

i. Drogue Deployment - A small parachute will deploy at or just after apogee

ii. Main Deployment - A larger parachute will deploy at a pre-defined height, slowing the rocket to a safe landing speed

6. Landing

3

Background

3.1 Rocket Avionics

3.1.1 Deployment Method

The most commonly used deployment system is to use black powder (also known as gunpowder) to generate pressure, forcing the rocket apart, deploying the parachutes. This is done by igniting an e-match (electronic match).

In a dual deployment system, the ignition is done by a flight computer. For redundancy, two flight computers, which are completely electronically separate, are recommended. The electronics are stored in an electronics bay, also known as an avionics bay or e-bay. Cavender, Daniel (2015)

3.1.2 Set Up of an E-Bay

A flight computer is used on launch day to control the parachutes of the rocket. There is a specific order of events that is done when setting up a rocket, and the flight computer for a launch. This order is important for safety while preparing a rocket launch.

1. Setting flight parameters
   The first thing is the input the flight settings. This includes the height the main parachute needs to be deployed at, any delays for triggering parachutes (e.g. parachute deploys 2 seconds after apogee), or some computers need

an acceleration liftoff threshold to detect liftoff. Then the Flight computer is turned off and placed into electronics bay. The flight computer will not be turned on again until right before launch, so it's essential the parameters are saved.

It's important to note that flight setting may need to be changed on the day, depending on weather factors like wind.

2. All other flight preparation is done
   This includes loading the motor, placing parachutes and setting up the deployment system. For a standard deployment this means that black powder and e-matches have been loaded in the rocket. The flight computer should not be turned until the rocket is about to be launched after this step, as there is a safety risk the computer could trigger the deployment, setting off the black powder.

3. The rocket is set up for launch
   The rocket will be placed on a launch rail, and the rail will be raised vertically. The flight computer will be turned on, generally a beeping pattern is done to indicate the flight computer is ready for launch.

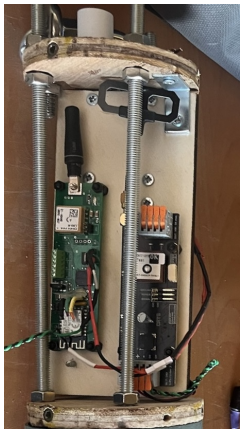

Figure 3.1: E-Bay with flight computers

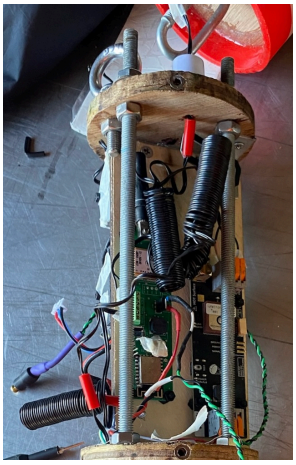

Figure 3.2: E-Bay with flight computers and e-matches loaded

3.2 Analysis of COTS Flight Computers

I research three COTS Flight Computers to determine features that they all have. These computers are some of the most common in high powered rocketry.

3.2.1 Cats Vega

The Cats Vega is the official flight computer for EuRoC, the European Rocketry Challenge, which is the biggest college rocketry competition in Europe. This computer has:

- IMU
- Barometer
- GNSS
- Radio

It has 2 pyro channels, 2 servo channels and general purpose I/Os for deployment events. It is a full open source project and is coded using C++. It uses a Kalman Filter which accounts for acceleration and barometric altitude. The computer uses the 2.4Ghz band for radio telemetry and is configured over USB through a program on a laptop. It costs around €394, not including shipping and import fees. Control and Telemetry Systems (2023)

3.2.2 Eggtimer Quasar

The Eggtimer Quasar is the highest end flight computer of the Eggtimer series of flight computers. It has:

- Barometer
- GNSS
- Radio

It supports dual deployment and 1 additional pyro event, as well as servo channels for each event. This project is not open source. It uses a Kalman Filter with only takes barometric altitude into account as there is no accelerometer. It is configured over WiFi. It costs around €212 for the assembled version, not including shipping and import fees. Eggtimer Rocketry (2024)

3.2.3 Telemega

This computer is by AltusMetrum, and it is the highest end of their product line. It has:

- IMU
- Accelerometer
- Magnetometer
- Barometer
- GNSS
- Radio

It supports dual deployment and 4 additional pyro events. This is an open source project. Its Kalman filter accounts for acceleration and barometric altitude. The computer is configured over USB through a program on a laptop It costs around €490, not including shipping and import fees. Altus Metrum (2024)

Each of these computers has a buzzer and LED to indicate the state of the computer. The buzzer will beep in a pattern to indicate the state of the computer. The LED will flash in a pattern to indicate the state of the computer. This is important so that the user can verify the computer is in the correct state before launch.

3.3 Analysis of Existing Flight Data

For testing my flight computer, I will use data obtained by ULAS HiPR for a rocket launch done as part of EuRoC 2024. The rocket flew with a Cats Vega and Eggtimer Quasar. This flight would not be considered fully nominal. Due to high winds on launch day, the main parachute was not allowed to be deployed. For the flight, the main parachute was removed, allowing the rocket to separate as expected at the set altitude (200m) but it didn't slow down as much as would have been expected. The vertical lines on the graphs represent the states of each computer. The Vega stores its states as numbers, below is the map of what each number maps to.

State	Number
INVALID	0
CALIBRATING	1
READY	2
THRUSTING	3
COASTING	4
DROGUE	5
MAIN	6
TOUCHDOWN	7

The Eggtimer uses the term LDA, which is Launch Detect Altitude. It also uses N-O, which is nose-over detection. This represents 1 second after apogee has been detected.

As seen, the Eggtimer tends to detect events earlier than the Vega. It estimates a lower velocity and a lower apogee height, but after apogee, it estimates a higher altitude. These differences likely come down to the fact the Eggtimer does not have an accelerometer. Additionally, the Cats Vega has a higher sampling rate for its sensors. (Cats Vega samples at 100Hz, Eggtimer samples at 50Hz). These factors along with potentiality different weightings in the Kalman Filters being used, likely account for these differences.

The overall results of this flight was an apogee at 2275m.

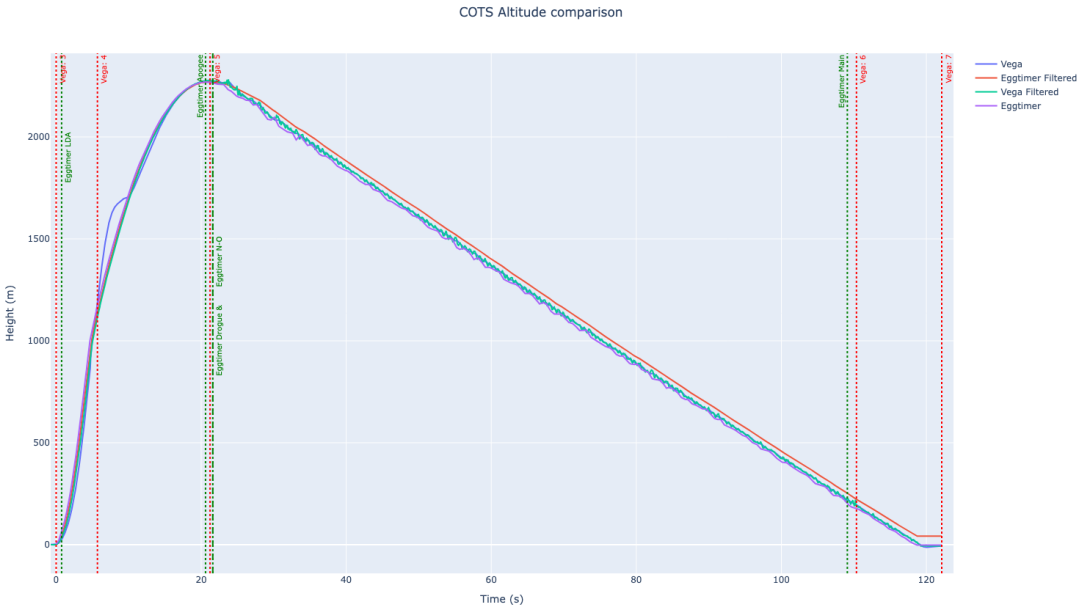

Figure 3.3: Altitude Data

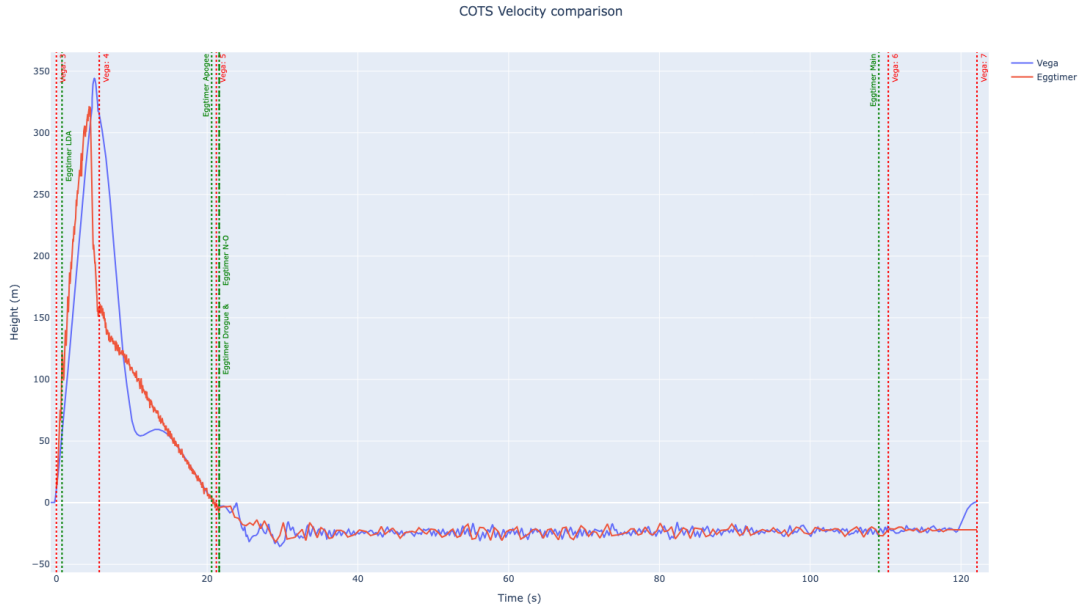

Figure 3.4: Velocity Data

4

Methodology

4.1 Defining Features

Before starting the design and coding process, I defined all the hardware and software features I wanted for this project. As well as, features I needed to meet my requirements.

4.1.1 Hardware Needed

From analysing the COTS computers, I determined I would at bare minimum need:

- Barometer
- GNSS
- Radio
- 2 deployment channels
- Some visual or auditory cue to indicate the state of the computer

As a way to gather more data, I decided to incorporate two accelerometers into the flight computer. One will be an IMU which can gather low-g acceleration data and gyroscope data. The other will be a high-g accelerometer. I will use the low-g accelerometer as the primary sensor. In the future the high-g accelerometer can be used as a backup in case of errors in the primary sensor.

4.1.2 Software Needed

For looking at the setting up of an e-bay and looking at the features of the COTs computers, I decided on these software requirements:

- Interface with sensors
- Change settings and have them save, even when powered off
- Accurately detect rocket flight stages, and take the appropriate action
- Display the status of the computer, either with a LED and/or buzzer
- Save flight data, even in the event of a power failure

For changing the settings, I wanted to be able to change them wirelessly. From experience, once the computer is placed in the e-bay the port used to connect a cable can be blocked, making wireless much easier to use on launch day.

4.2 Tech Stack

4.2.1 Hardware

Throughout this project, I used a Raspberry Pi Pico W (using an RP2040 chip) as the main flight computer. I used a Raspberry Pi Pico as a debugger board and when testing also used a Raspberry Pi Pico to stream the test data. It was setup as seen in Figure 4.1.

The debugger was connected to the two debugging pins on the debugging Pico, as well as UART so the serial output of the main flight computer could be sent to the debugger Pico. The testing Pico was also connected over UART to stream test data.

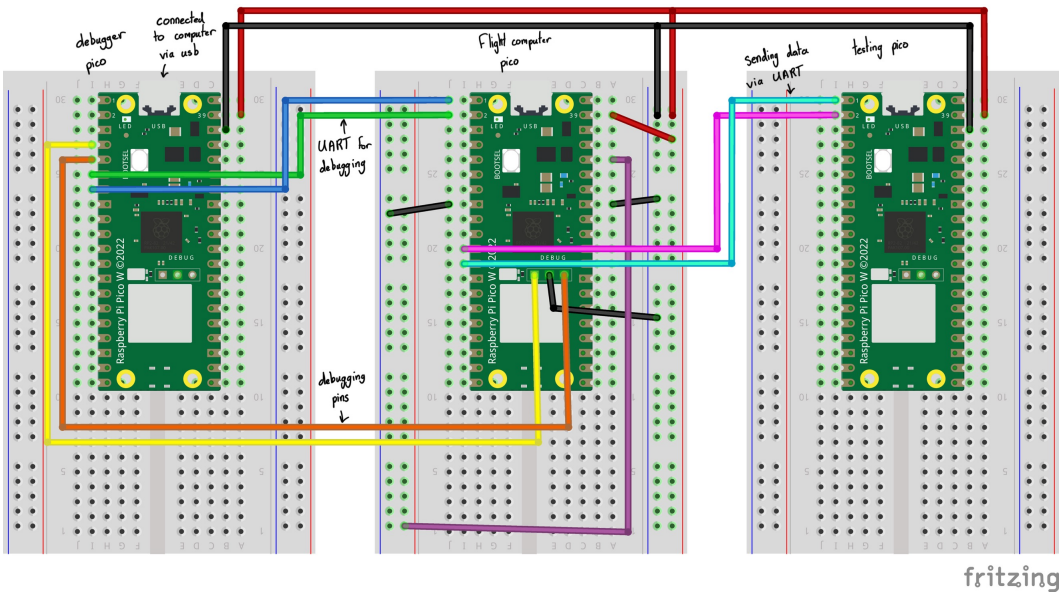

Figure 4.1: Wiring Diagram of the debugging and testing setup

4.2.2 Software

The Pico is officially supported with a C/C++ SDK and a MicroPython Port, additionally there is a CircuitPython port.

C++

For the flight computer, I used C++ for this project as it's faster than Python, and provides classes which helped me implement modularity in my code. Additionally, using C++ meant I could use FreeRTOS which was used for its real time qualities, as well as its features like queue and priorities. For this flight computer, it is import that critical tasks, like calculating height and firing parachutes can take priority over tasks that matter less, like GPS and telemetry.

CircuitPython

CircutPython was used initially with sensors to verify all the wiring was correct. As most of the sensors I was using for initial testing were Adafruit sensor modules. Adafruit writes libraries and examples for all their modules in CircuitPythont, so it was quick and easy to verify all the sensors were working as expected.

Micropython

For the testing Pico, I used MicroPython. I personally have found this faster and easier to use compared to CircuitPython, so when creating a simple script to read a text file and send that info to the flight computer, I opted to use MicroPython.

PCB Design

I used KiCad to develop my schematic and PCB. It is a free, open source software that I had been familiar with from previous projects. It also provides suitable output files for PCB production using sites like JLCPCB.

For creating the similar wiring diagrams, I used Fritzing. This is an open source tool that I have been using for many years. It is easy to use and has lots of support from hobbyist focused companies like Adafruit, making it easy to create wiring diagrams with their sensors.

4.2.3 Development Environment Set Up

For developing on the Raspberry Pi Pico in C++, I used the Pico VS Code Extension, which installs and sets up all the necessary packages to develop on the Pico. I used 3 total Pico boards for the development my code.

When using the VS Code Extension, the code will be compiled into a '.uf2' image, which then needs to be flashed to the Pico when in boot mode. This is an inconvenient process as it involved plugging the Pico in and out of the computer every time a change is made to the code. To avoid this I used a second Pico as a debugger.

I installed the 'debugprobe_on_pico.uf2' image by the Raspberry Pi foundation onto the debugger Pico. This was connected to the main Pico, and it enabled me to flash new images onto my main Pico without unplugging and plugging it in again. It also enabled me to use the debugging feature to go step by step though code and see variables values.

I had my main Pico, which runs the flight computer software. This will be what is mounted on the PCB and is connected to all the sensors.

I used a third Pico for testing. This Pico was connected to the main Pico via UART. It streamed data from a COTS flight computer that had gathered from a previous rocket flight. This meant I could get a sense of how the computer would

act during flight. This Pico ran a simple python program, reading in a text file, and writing them over UART to the main Pico with to simulate a real flight.

4.3 PCB Design

4.3.1 Deciding Sensors

From the hardware requirements I choose each of the following sensors:

Radio Module

The RA-01H is a LoRa radio module that uses the SX1276 chip. The SX12 family of chips is commonly used in flight computers. The RA-01 range of chips was recommended to the ULAS HiPR team by another rocketry team. The H variant was chosen as it operates in the ranges of 803 - 930MHz, which includes the free the 868MHz frequency which can be used without a license in the EU. The alterative RA-01 operates the in 410MHz 525MHz, which can experience more interference than the 868MHz frequency. It is easy to integrate into a PCB and uses the SPI protocol.

GNSS Module

The MAX-M10S was chosen based on the common use of the MAX-MX series for flight computer. The MAX-M10S is the most recent in the series and is the most frequently in stock. It also had comprehensive documentation on schematics and coding.

Accelerometers

I decided to use 2 accelerometers, one for low-g measurements and one for high-g. The low-g is the primary sensor which will be used to determine acceleration during flight. The second is purely to gather data and help with any post flight analysis. In the future both accelerometers could be integrated, or use one as a backup in case of failure.

The LSM6DSO32 was chosen as the low g accelerometer. It is a 6-axis IMU, so has an accelerometer and gyroscope, provide extra data to analyse post flight. It has an acceleration range of ±4/±8/±16±32 g and an angular rate range of ±125/±250/±500/±1000/±2000 DPS (degrees per second).

The ADXL375 was chosen as the high-g accelerometer. It is 3 axis and can measure up to 200±g.

Barometer

I chose the MS5607 barometer for its common use in flight computer and comprehensive documentation.

4.3.1.1 Communication Protocol

Sensor	Protocol	Reasoning
LSM6DSO32	SPI	As a core sensor, this will be read more frequently, so a high speed protocol is needed
ADXL375	I2C	This is not vital to functionality so can use a lower speed protocol
MS5607	SPI	As a core sensor, this will be read more frequently, so a high speed protocol is need
MAX-M10S	UART	UART makes it easier to parse the GNSS data compared to using I2C, the other option for this sensor
RA-01H	SPI	Only option for this sensor

4.3.2 Design Process

The PCB design was done in KiCad. For this process, I followed each sensors' documentation on how to wire it up. I have separated each component into a block to make managing the PCB easier.

All the sensor symbols were downloaded from DigiKey, or from the existing KiCad library.

I added a voltage regulator to step down the voltage so the Pico would have the correct voltage. The voltage regulator can take in up to 20V. It then steps it down to 5V, the required voltage needed to run a Raspberry Pi Pico. All of the sensors run off 3.3V, they are powered from the 3V3 pin from the Pico. The Pico internally steps 5V down to 3.3V for this.

The battery is assumed to be 7.4V as that is what is standard use within ULAS HiPR.

There are four terminals on this board, one is to connect the battery to. The second is to connect an external switch, so the flight computer can be turned on externally. This is done once it has been placed into the avionics bay.

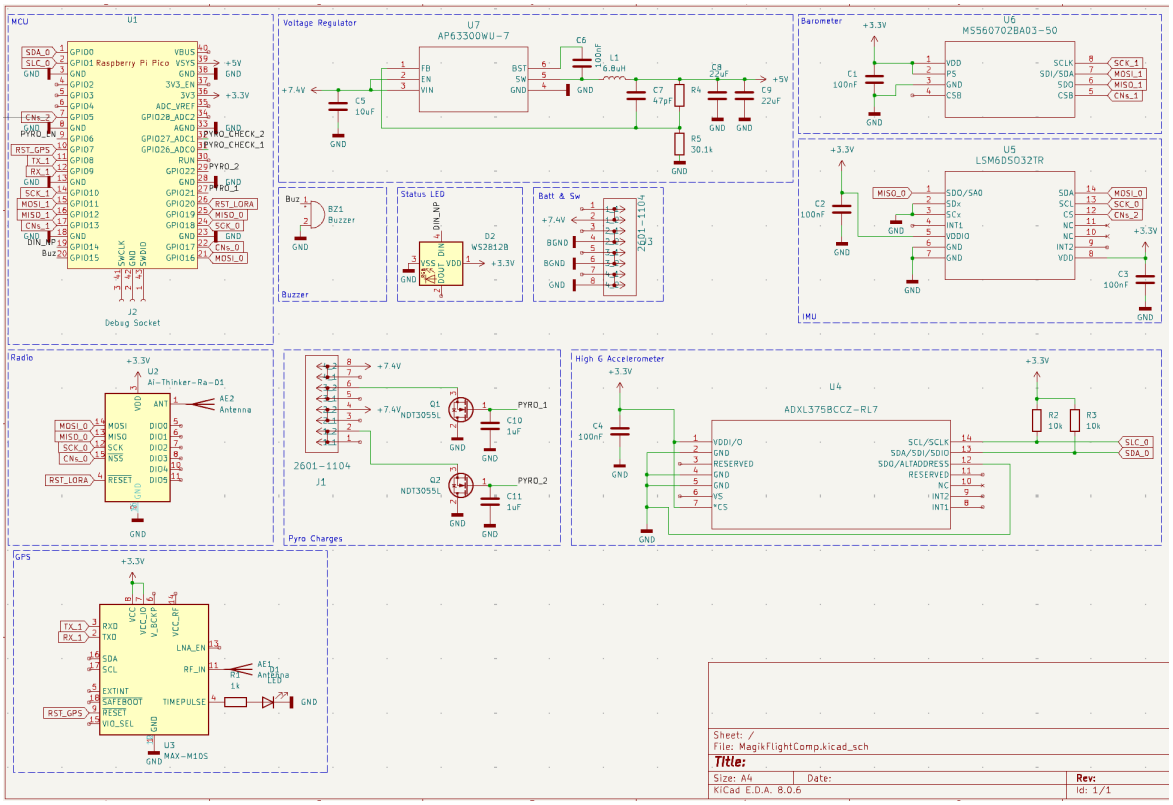

Figure 4.2: Schematic for the PCB

The two remaining terminals are for connecting two e-matches to. These terminals are controlled by n-channel MOSFETs. They are connected to straight from the battery input to make sure there is enough current to trigger the igniters. When the respective pyro pin is pulled high, it will allow current to flow, triggering the igniter.

Extra Features

On this board I have also added a connector for an antenna for the radio. Additionally, there is buzzer and status led, which is a WS2818, so the user can know what state the computer is in. The last thing on the board is a standard JST_PH 3 pin socket that is for the 3 breakout pins. Additionally, there are two header pins, when connected these trigger the settings mode

For the physical design, I attempted to keep it as small as possible. The terminals on the top left is for the battery and external on switch. The terminal on the bottom right are for the main and drogue igniters.

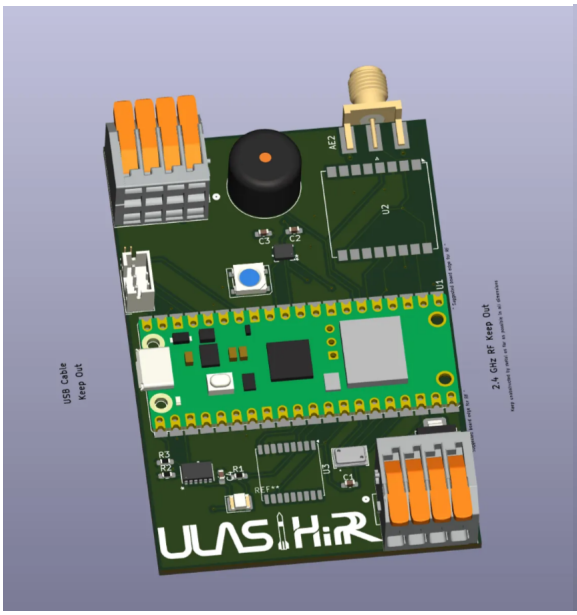

Figure 4.3: 3D Render of the PCB Front

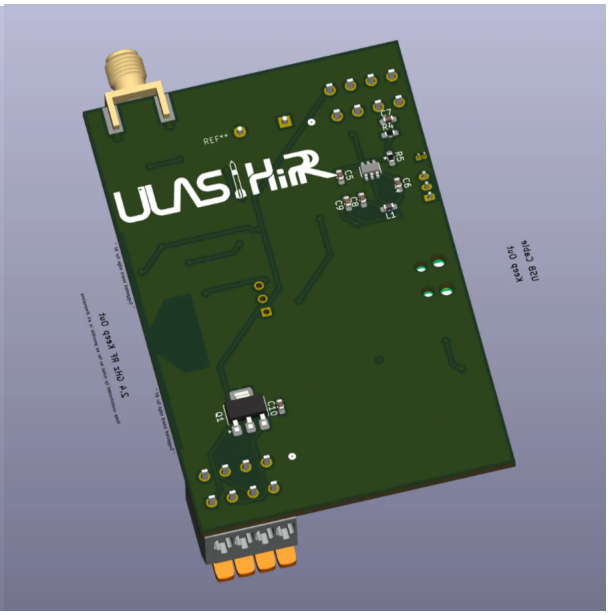

Figure 4.4: 3D Render of the PCB Back

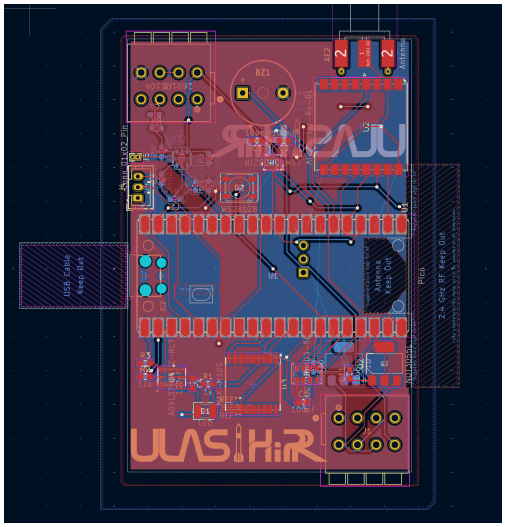

Figure 4.5: Top layer of the PCB

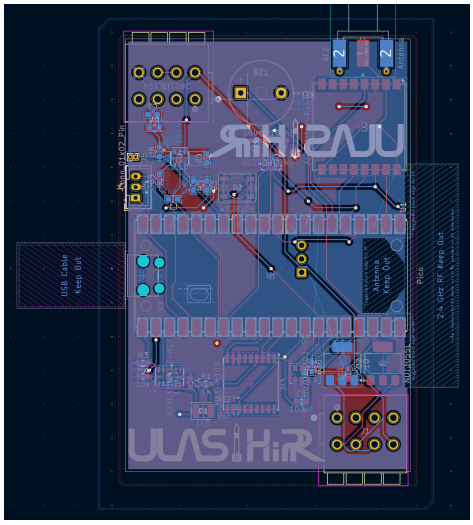

Figure 4.6: Bottom layer of the PCB

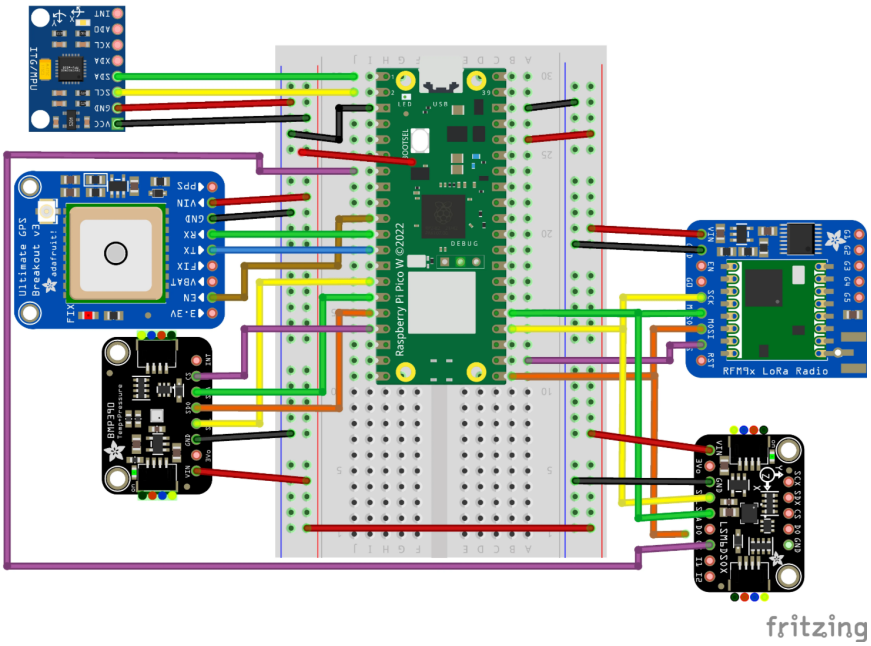

Figure 4.7: Wiring Diagram for basic sensor setup

4.3.3 Prototype

Due to time constraints, I was not able to produce the PCB. For development, I used a prototype with a basic sensor setup. This was made with sensors modules I already had on hand. A basic through hole board was used to solder female headers on. The sensors and Pico had male headers, so it was easy to slot them into the board. This prototype used the exact same pins and communication protocols that the PCB would use to ensure easy switchover once the PCB was produced.

While developing the code for this prototype, the secondary accelerometer was removed so the debugger Pico could be connected via UART for serial data.

Sensor Function	Sensor Module	Sensor in module
Primary Accelerometer	Adafruit LSM6DSOX	LSM6DSOX
Secondary Accelerometer	Fermion: MPU-6050	MPU6050
Barometer	Adafruit BMP390	BMP390
Radio	Adafruit RFM95W LoRa	RFM95W
GNSS	Adafruit Ultimate GPS Breakout	PA1616D

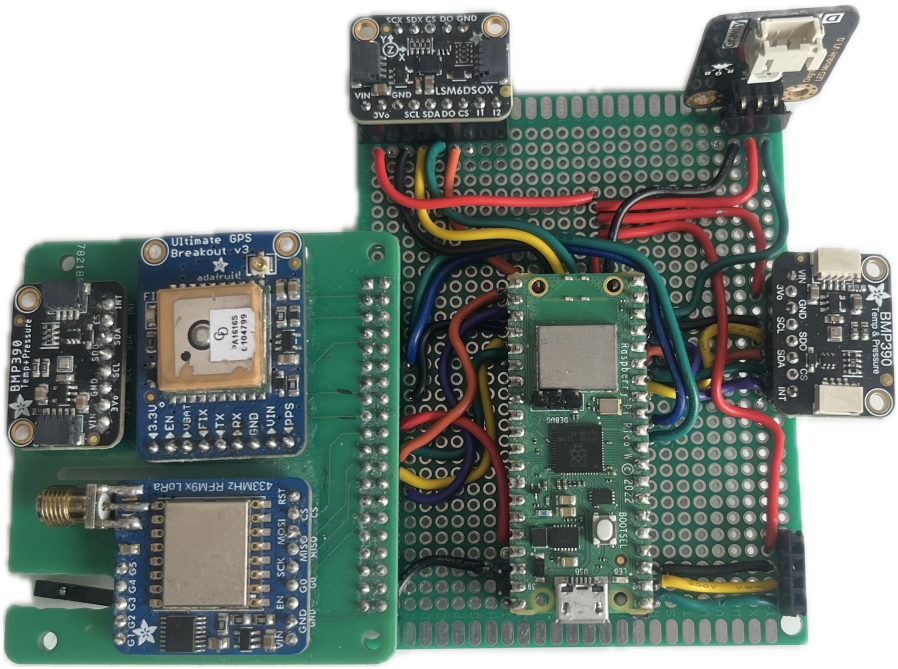

Figure 4.8: Prototype sensor setup

I used an RGB LED Module from DF Robot, which used a WS2812B LED as a stand in for the WS2818 on the PCB.

As I wanted to test having a pyro lock for safety, I used a Digital 5A Relay Module by DFRobot to act as an AND gate. In actual flight the delay a relay adds would be too long, but for testing it is good enough. The relay was triggered by a GPIO pin from the Pico, which once the powered flight state was reached, would pull the pin high. This would then trigger the relay, which would connect to ground, allowing current to flow.

I used simple LEDs in place of e-macthes. The positive pin of the LED was connected to one of the Pico's GPIO pins, and the negative pin was connected to the Relay. If the relay was triggered it connected the negative pin to ground, and then when the GPIO pin was pulled high, current would flow lighting up the LED.

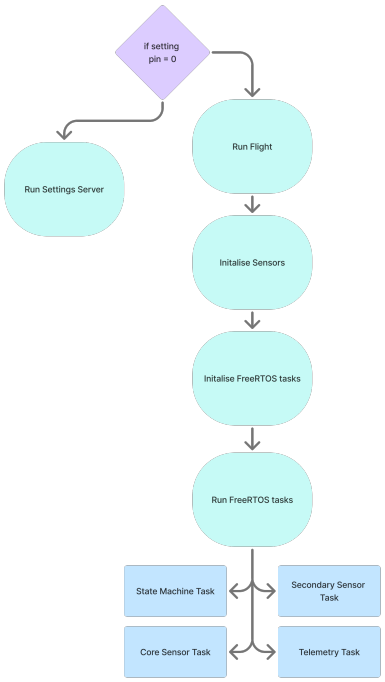

Figure 4.9: Overall Flowchart

4.4 Code Implementation

On startup, the main code loop is determined by whether a setting pin has been pulled low. If it is, then the Flight Computer will enter into a settings mode. This starts a BLE Gatt server that can be used to set flight parameters.

If the pin is not pulled low, then the flight computer will enter into a flight mode. This initialises all the sensors and FreeRTOS tasks, and starts each task.

4.4.1 Settings

When the setting mode is pulled low, setting mode will initialise. This starts up a BLE Gatt server. The endpoints are below.

Endpoint	Function	
set_main_height	Sets the height the main parachute will deploy	write
get_main_height	Returns the main	write
get_drogue_delay	Returns the delay the drogue parachute will deploy after apogee	read
set_liftoff_threshold	Sets the acceleration threshold for liftoff	write
get_liftoff_threshold	Gets the acceleration threshold for liftoff	read
confirm_settings	allows the user to confirm settings and save them to flash	write

The server can be interacted with any device with Bluetooth. An easy app to use is nRF Connect by Nordic Semiconductor. This app allows for reading and writing to BLE Gatt servers.

For implementation, I followed a guide by Adams, V.Hunter (2025). I first created a GATT (Generic ATTribute Profile) file, which had all the details about my endpoints. I used a randomly generated 128-bit UUID as I didn't want to use one of the predefined services from Bluetooth (n.d.). Next, I initialised a GAP (Generic Access Profile) so the flight computer could be found by other devices. It advertises itself, so when a client scans, it can be found and connect to. Once connected the client is able to see the GATT profile and call its features.

4.4.2 Flight Mode

If the settings pin is not pulled low, the flight computer will enter into flight mode. This first creates all the FreeRTOS queues that will be used. Next it initialises all of the sensors being used. Sensors handlers and the state machine are then created. The status LED is set green, indicating start up has been a success. Finally, the FreeRTOS tasks are created, and the scheduler is started.

4.4.3 State Machine

As seen in the background, the flight of a rocket has clear and defined stages. For each stage a certain action needs to happen. I decided to use a deterministic state machine to control the flow of event during the flight. This ensures a level of safety during flight, for example the parachutes can only be triggered once the

"Apogee" State has been reached, reducing the risk of early deployment which will likely cause critical failure.

The state machine takes in a list of function, each one associated with a state. To make sure each function is only called once, there is an array of Booleans. When the machine transition to a new state, it will call the associated function, given it hasn't been called before. It will then make the function as complete so it cannot be re-run. Below is the transition table I based my code on

State	Triggered Action	Transition Condition	Next State
Calibrate	Checks queues are receiving data	All sensors returning data for 10s	Ready
Ready		acceleration > lift-off threshold	Powered Flight
Powered Flight		acceleration $<m/s^2$	Coasting
Coasting	Unlock pyro_en pin	velocity $<0m/s$	Drogue
Drogue	Trigger drogue parachute deployment	altitude < main deployment altitude	Main
Main	Trigger main parachute deployment	acceleration = $0m/s^2$ and velocity = $0m/s$	Touchdown
Touchdown	Increase the telemetry tasks priority		

4.4.4 Sensors

To try and keep code as modular as possible, I separated out the sensors from the drivers being used. Each sensor inherits from the base class 'Sensor'. This enables the sensors to be passed through to the sensor handler. The sensor class has the property of a 'Driver'. Depending on what has been defined in the 'config.h', it will initialise the correct driver. This enables easy switching from one type of sensor or interface the sensor is using.

For example

I was initially working the MPU6050 accelerometer which only had I2C breakouts, I then switched to the LSM6DSOX, which used SPI.

```
Accelerometer::Accelerometer(SPI *spi, int cs) {
    printf("Accelerometer created, spi\n");
    gpio_init(cs);
    gpio_set_dir(cs, GPIO_OUT);
    gpio_put(cs, 1);

    bi_decl(bi_1pin_with_name(cs, "SPI CS ACCEL"));

    #ifdef ACCEL_LSM6DSOX
        printf("Accelerometer LSM6DSOX\n");
        accelerometer = new LSM6DSOX(spi, cs);
    #endif

}

Accelerometer::Accelerometer(I2C *i2c, int addr) {
    printf("Accelerometer created, i2c\n");

    #ifdef ACCEL_ADXL345
        printf("Accelerometer ADXL345\n");
        ADXL345 accelerometer = new ADXL345(i2c, addr);
    #endif
    #ifdef ACCEL_MPU6050
        printf("Accelerometer MPU6050\n");
        accelerometer = new MPU6050(i2c, addr);
    #endif

}
```

When a read occurs on a sensor, it will call the set driver to return the raw values. It will then preform the necessary processing for each sensor.

Accelerometer

Due to gravity, the 'up' direction of the accelerometer must have 9.8 subtracted from it. This can be used on initialisation to verify the flight computer is in the correct orientation.

Barometer

To calculate altitude, I use:

$$altitude = 44330.0 * (1.0 - (pressure/1013.25)^{0.1903})$$

This is a simplistic version of the barometric formula. It assumes temperature is 15C and pressure at sea level is 1013.25Pa. Initially I did not recover the starting altitude and just used the altitude above sea level (ASL), but to improve the performance of my Kalman Filter I added a change to my barometer so that on initialisation it takes the first reading, as stores that as it's 'start_altitude'. This is then taken away from each measurement from then on, giving a more accurate representation of the distance the rocket has ascended.

4.4.5 Storage

I decided to use the inbuilt flash storage on the Pico, which has 2mb of flash memory. Some of this memory is used to store to program itself, so it was import to be careful when erasing and writing to flash.

To gauge the size of my program is used a command to get the size of the .elf image. I then set the memory offset for 512 bytes bigger than that to avoid any clashing.

The most important data is the flight settings, to ensure safety I saved this in the last 4096 bytes of the flash. The Pico has to erase a minimum of a 4096 byte sector. The settings storage the following data:

- Main deployment height
- Drogue deployment delay
- liftoff threshold
- last written offset

These settings are read in on initialisation to the settings class. If the Bluetooth mode is enabled, the user can connect to the flight computer change and change these values through the Bluetooth server. Once they have 'logged off', this data will be written back to the flash, ready for flight. To do this, the code will first erase the existing data in the given sector, it will then write the data back into that section.

For the data itself, it will start writing data to what the 'last written offset' sector was set as. The logger will store the data in a variable until it reaches the

required size of 4096 bytes of data. It will then write all the data. This is to ensure there isn't excessive erasing of the flash. Additionally, the logger will only start recording the data once the system has reached the 'powered flight' stage, ensuring the Pico does not run out of space. It will stop logging data 10 seconds are touchdown has been detected.

The last written offset will be updated within the config class, every time there is a portion of the flash erased. Once the logging has been completed (10 seconds after touchdown). This value will be written to the flash, in the flight storage settings section.

This will ensure, if the Pico is to turn on again the data will not be overwritten before there is a chance to recover the data.

4.4.6 FreeRTOS

FreeRTOS was used for its real time qualities, as well as its features like queue and priorities. For this flight computer, it is import that the most vital tasks, like calculating height and firing parachutes can take priority over tasks that matter less, like GPS and telemetry.

For this I split the sensors on the board into two groups, the core sensors which the Kalman filter will use to estimate the state of the flight computer and the secondary sensors which consist of sensors that are nice to have, but don't affect the flight or safety at all.

The core sensors are the barometer and the low-g accelerometer.

The secondary sensors are the high-g accelerometer and GNSS.

Sensor

There are two of these tasks, one for the core sensors and one for the secondary. They used the exact same code. For each task a sensor handler is initialised, this contains a vector that sensors can be added to and a queue that the task will add data too. The core sensor handler contains the barometer and low-g accelerometer, as well as the CoreDataQueue. The secondary sensor handler contains the high-g accelerometer and GNSS, as well as the SecondaryDataQueue. The core sensor handler is given a higher priority than the secondary sensor handler. When the handler is run, it will start a loop that will call the read

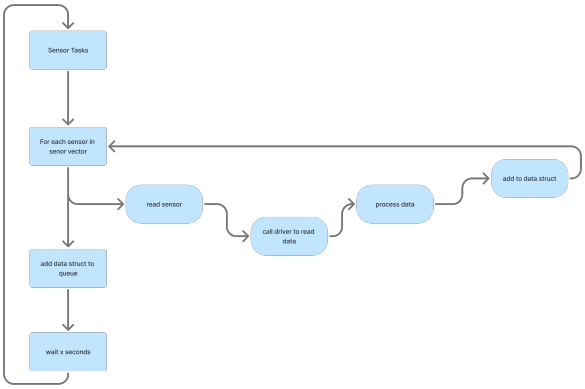

Figure 4.10: Sensor task flowchart

function of each sensor in its respective vector. Then the data is pushed into its queue. This is repeat throughout the flight.

State Machine Task

The state machine task relies on data coming in from the CoreDataQueue, this is fed into the Kalman Filter which updates its acceleration, altitude and velocity calculations. It will then update the state machine based on these updated calculations. If there is a change in state, it will call the function associated with that state.

The core data, Kalman data, and secondary data are combined and then pushed to the FlightDataQueue to be passed to the telemetry

The code will run in a loop for the duration of the flight

Telemetry Task

This task runs in a loop. When flight data comes in from the flight data queue, it logs the data and then sends it through the telemetry.

4.4.7 Kalman Filter

As seen in the background, the Extended Kalman Filter is suited more for rocketry. This comes at the cost of computation power. I decided to use standard

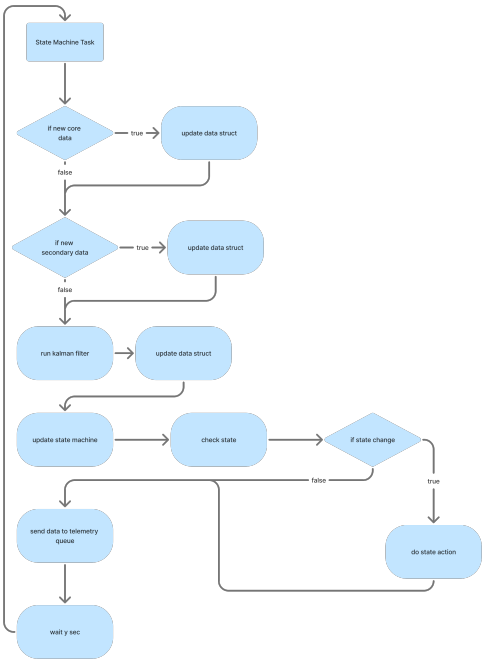

Figure 4.11: State machine task flowchart

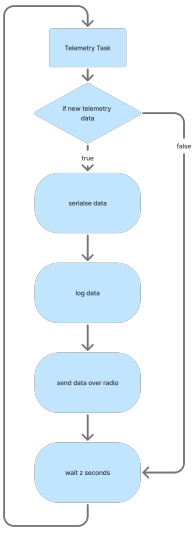

Figure 4.12: Telemetry task flowchart

Kalman Filter for this computer to save on that computational power, it was also easier and faster to implement. To compensate for this I set my acceleration covariance matrix (Q) to high value

The Kalman filter was implemented with the help of the Eigen C++ library for easy matrix multiplication. For this filter I am estimating altitude, velocity and acceleration, so my x matrix is :

$$x = \begin{bmatrix} s \\ v \\ a \end{bmatrix}$$

where s = altitude, v = velocity, a = acceleration

4.4.7.1 Initialisation

I initialise all of x as 0.

I am confident my starting acceleration and velocity will be close to 0, so I have low P value (P representing the confidence I have in the X matrix values) for them. I am less confident in my altitude as I used the standard $44330.0 * (1.0 - (pressure/1013.25)^{0.1903}$, this formula takes sea level pressure as 1013.25 and temperate as 15C. As sea level pressure and temperate change depending on time and location and this formula will give altitude relative to sea level and not the current altitude, it is unlikely the starting altitude will be 0. As of this, I have set the P value higher for altitude.

Similarity for Q (Q representing the confidence I have in the F matrix), I am confident in the altitude and velocity equations in my F matrix, but have made the Q value for acceleration a high number. As discussed in the background this will help offset the fact that the rocket is not a linear system, and the acceleration will not be linear, like the Kalman filter will predict.

$$x = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.0001 & 0 \\ 0 & 0 & 0.0001 \end{bmatrix} Q = \begin{bmatrix} 0.001 & 0 & 0 \\ 0 & 0.001 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

The H matrix is used when including the actual sensor readings. As I am using a barometer and accelerometer for inputs, it is initialise as 1 for the altitude and 1 for the acceleration measurements.

The R matrix are the known sensor noise. These are known constants that need to be measured on a sensor by sensor basis

This was implemented in code below:

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} R = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.01 \end{bmatrix}$$

```
KalmanFilter::KalmanFilter() {
    // Initialize matrices
    x = Eigen::VectorXd(3);
    x(0, 0) = 0; // Initial position
    x(1, 0) = 0; // Initial velocity
    x(2, 0) = 0; // Initial acceleration

    // confidence in x matrix prediction
    P = Eigen::MatrixXd(3, 3);
    P << 1, 0, 0,
         0, 0.0001, 0,
         0, 0, 0.0001;

    F = Eigen::MatrixXd(3, 3);

    //confidence in f matrix prediction , could scale for time in fu
    Q = Eigen::MatrixXd(3, 3);
    Q << 0.001, 0, 0,
         0, 0.001, 0,
         0, 0, 10;

    H = Eigen::MatrixXd(2, 3);
    H << 1, 0, 0,
         0, 0, 1;

    R = Eigen::MatrixXd(2, 2);
    R << 0.1, 0, //baro noise
         0, 0.01; //accel noise
}
```

4.4.7.2 Prediction

This step first uses the given equations and measurements from the last iteration the estimate the next state of the rocket.

Then based on the weighting provided by the user in the Q matrix, the P matrix is updated. As mentioned above P is the confidence in the X matrix, this is also known as the estimation error.

The F Matrix is based off the equations of motion where:

s = altitude, v = velocity, a = acceleration, t = time

$$s_n = v_{n-1}t + 1/2at^2$$

$$v_n = v_{n-1} + at$$

$$a_n = a_{n-1}$$

When put into matrix form, this looks like:

$$F = \begin{bmatrix} 1 & t & 0.5t^2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

To predict the new state:

$$\mathbf{x}_{n+1} = F\mathbf{x}_n$$

$$\mathbf{P}_{n+1} = F\mathbf{P}_n F^\top + \mathbf{Q}$$

This is as follows in code:

```
void KalmanFilter::predict(int t) {
    // based of s = ut + 0.5at^2, v = u + at, a = constant

    F << 1, t, 0.5 * t * t,
         0, 1, t,
         0, 0, 1;

    x = F * x;
    P = F * P * F.transpose() + Q;
}
```

4.4.7.3 Update Step

This step takes in the actual sensor measurements. It forms a K matrix which is the Kalman Gain. This is used to update the estimate state from the last step. It takes the actual measured values from the sensors and the estimated state and applies the Kalman Gain to get an updated state estimation.

Finally, the P matrix is updated to better represent the estimation error, where :

z_{baro} = most recent altitude reading from the barometer

z_{accel} = most recent acceleration reading

$$z = \begin{bmatrix} z_{\text{baro}} \\ z_{\text{accel}} \end{bmatrix}$$

$$K_n = P_{n-1}H^\top (HP_{n-1}H^\top + R)^{-1}$$

$$x_n = x_{n-1}K_n(z_n - Hx_{n-1})$$

$$P_n = (I - K_nH)P_{n-1}(I - K_nH)^\top + K_nRK_n^\top$$

```
void KalmanFilter::update(float z_baro, float z_accel) {
    Eigen::VectorXd z(2);
    z(0) = z_baro; // barometer measurement
    z(1) = z_accel; // accelerometer measurement

    Eigen::MatrixXd K = P * H.transpose() * (H * P * H.transpose()
        + R).inverse();

    x = x + K * (z - H * x);
    P = (Eigen::MatrixXd::Identity(3, 3) - K * H) * P *
        (Eigen::MatrixXd::Identity(3, 3) - K * H).transpose()
        + K * R * K.transpose();
}
```

4.4.7.4 Iterations

The first iterations I did was to better calculate height, as mentioned in the Barometer breakdown, instead of taking the height as above sea level, I wanted to use AGL (Above Ground Level). Once implemented the altitude now roughly

start at 0, as the X matrix says. The second iteration I slightly altered the Q matrix, adding in a high uncertainty for the acceleration calculation. I also slight increased the velocity P matrix.

4.4.8 PIO Usage

Status LED

My primary use for PIO was for the status led I was using. This is a WS2818 led that uses a specific protocol to work. Using the PIO is ideal for this purpose, as the protocol requires precise timing.

This led was used to indicate the status of the computer. For the code, I used the official example given by Raspberry Pi. I wrapped this in a class so it was easy to call throughout the code to indicate the state of the computer.

The LED will turn red when the computer is initially powered on, if it is in settings mode it will turn blue, and if it's in flight it will turn green.

I2C

While wiring my prototype board, I accidentally wired the secondary accelerometer incorrectly, switching the SLC and SDA pins. Instead of fix this on the hardware, I decided to use PIO to fix it. Using the I2C example from Raspberry Pi, I was able to use the pins that I had initially wired wrong.

This shows how using PIO can make the computer more modular.

5

Testing and Evaluation

5.1 Getting Data

As mentioned in the background, I am using flight data from ULAS HiPR's EuRoC 2024 Flight for testing my code

This data was taken off a Cats Vega which records all it's raw and processed data in their custom '.cfl' file. This can then be extracted into CSV files with their python tool.

From the CSV files, I used the time, barometer pressure, IMU acceleration (x, y and z) and GNSS latitude and longitude. In Python, I parsed these values and used a specific struct depending on the type of sensors. Sensors like the barometer and accelerometer are samples at a much higher rate than the GNSS so I couldn't just add all three into a struct every time. Once parsed, all this data was saved to a text files.

All of this data was saved with a timestamp of when it was recorded, a starting character and the data itself. The starting character is used to identify which sensor the data is from. Here is an example line from the text file:

-0.755 b'b\x16\x10\x00\x00n\x85\x01\x00'

Sensor Type	Starting Char	Data
accelerometer	a	float[0] float [4] float[8]
barometer	b	int [0] int [4]
GNSS	g	double[0] double [8]

This text file was saved onto the testing Pico. This ran a simple while loop, reading a line of the text file, separating out into the data itself and the timestamp. It sends the data over UART to the flight computer Pico. This data is then handled in the Test Handler Driver. The Pico then sleeps for a specified amount on milliseconds. While testing, I could vary this speed to see how the flight computer Pico behaved.

The timestamp was initially put in so I could emulate the rocket flight at real speed, unfortunately the flight computer Pico struggles to read the UART values correctly when the data is quicker than 0.05 seconds, so the data was sent every 0.5 seconds. The flight computer Pico is running at 50Hz, so it can read the data at 0.05 seconds. These combined means the test runs at 10Hz The highest sampled sensors, the barometer and accelerometer, are sampled every 0.01 seconds on the Cats Vega, so the test takes 10 times longer than the actual rocket flight. (20 minutes instead of 2 minutes).

5.2 Test Handler

The test handler acts as a sensor driver on the flight computer. When TESTING is defined in the config, an extra FreeRTOS task is run. This task reading the UART port that the tester Pico is connected too. It parses the data being sent based on the starting char. It then saves this data into the appropriate struct.

Each data type has its own test driver. This driver behaves the same as the standard drivers, but instead of reading sensor, it reads in the last test handler data for that sensor type.

5.3 Getting the results

To make getting data off the flight computer Pico quick and easy, I simply used print-outs. As the Pico was connected to my laptop via the debugger Pico, I used minicom, a serial communications program, to open a serial connection. I let the testing Pico run through all the flight data, and then saved the flight computer serial output as a text file. Using a simple python script, I parsed this data and used it to make graphs using pandas and plotly.

Due to the print-out task being at a low priority, the data was occasionally corrupted when being received by minicom, resulting in outliers in the data. Some of this data was filtered out due to being in an invalid format when being parsed, but some outliers do remain in the graphed data.

Additionally, as a Cats Vega starts logging data slightly before liftoff, the test data starts at -0.75s (0 being when lift-off occurs). I shifted back all the results by -0.75 as well to accurately compare the Kalman Filter predictions.

As the data is processed at a lower rate than the COTS data, I scaled the data to be the same length as the COTS data. This was done by dividing the time by 10, as the COTS data is sampled at 100Hz and the test data is sampled at 10Hz. This means the data can be accuracy compared to the COTS data.

5.4 Functionality

5.4.1 Settings

There are two aspects for the settings. The first is the storage of the settings. This preformed nominally. To set the settings, I compiled code that used the save_settings function, saving the settings data.

I ran this code and then complied code that only used the read_settings function to read the flash, the settings were the same as the ones set in the initial code.

The second aspect is the Bluetooth server to edit the settings. When triggered, a phone was used to attempt to connect the flight computer, and the access each endpoint. The phone was able to connect, and the endpoints could be viewed, but did not perform the expected actions.

5.4.1.1 Table of results

Description	Result
Writing to flash	Success
Reading from flash	Success
Connect device over Bluetooth	Success
set_main_height endpoint	Failure
get_main_height endpoint	Failure
get_drogue_delay endpoint	Failure
set_liftoff_threshold endpoint	Failure
get_liftoff_threshold endpoint	Failure
confirm_settings endpoint	Failure

5.4.2 Sensors

I planned to do two phases of sensor testing. The first was to test the wiring of the sensors. This was done in CircuitPython with examples from the Adafruit documentation. Every sensor bar the secondary accelerometer performed nominally. As mentioned the secondary accelerometer was wired correctly, so I switched to utilising PIO instead of the standard I2C for the next test.

The second phase was to test the computers in C++. Due to no being able to produce the PCB, and then not having time to write drivers for the majority of the sensors in C++, I was unable to fully complete this phase of testing. I was able to test an accelerometer using I2C and I2C with PIO, which both returned correct readings.

5.4.2.1 Table of results

Description	Protocol	Wiring (Python)	C++ Functionality
Accelerometer (MPU6050)	I2C	Success	Success
Accelerometer (MPU6050)	I2C using PIO	Fail	Success
Accelerometer (LSM6DSOX)	SPI	Success	Not Complete
Barometer (BMP390)	SPI	Success	Not Complete
Radio (RFM95W)	SPI	Success	Not Complete
GNSS (Adafruit GPS)	UART	Success	Not Complete

6

Analysis

6.1 Kalman Filter

6.1.1 Altitude

When estimating altitude, the Kalman Filter was accurate for the majority of then flight. As seen in both the ±100 meter bound, and the ±10% bound, the Kalman filter can accurately predict up to apogee. After apogee, it starts to drift further away from the COTS predictions. I believe this is due to the velocity and acceleration values not being correct.

6.1.2 Velocity and Acceleration

These graphs are both very far off the values from the COTS computer. The velocity is heavily influenced by the acceleration so, I believe that the incorrect acceleration values are causing the incorrect velocity. One of the factor that might have caused the inaccuracy of velocity is that I did not use an extended Kalman filter. Using this would have accounted for acceleration more accurately and likely resulted in a higher initial spike in velocity compared to the plateaued curve seen in between the lift-off and apogee phases.

As seen, the acceleration does follow the general shape of the COTS values but on a much smaller scale. I think one of the main reasons for this is I have not accounted for gravity correctly. This should not be as much of an issue when

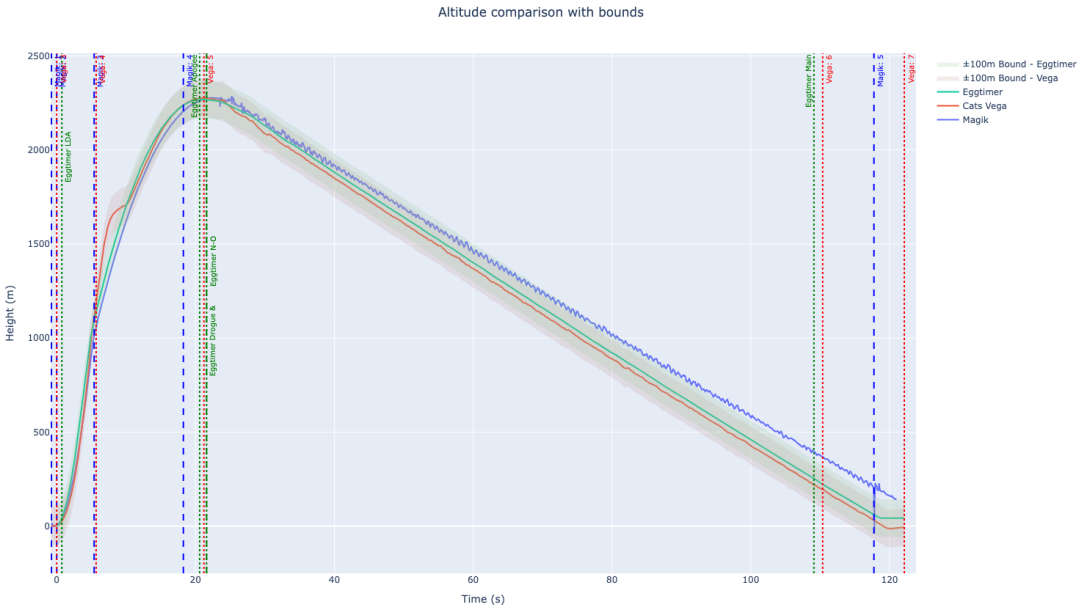

Figure 6.1: Altitude Comparison with ±100 meter bound

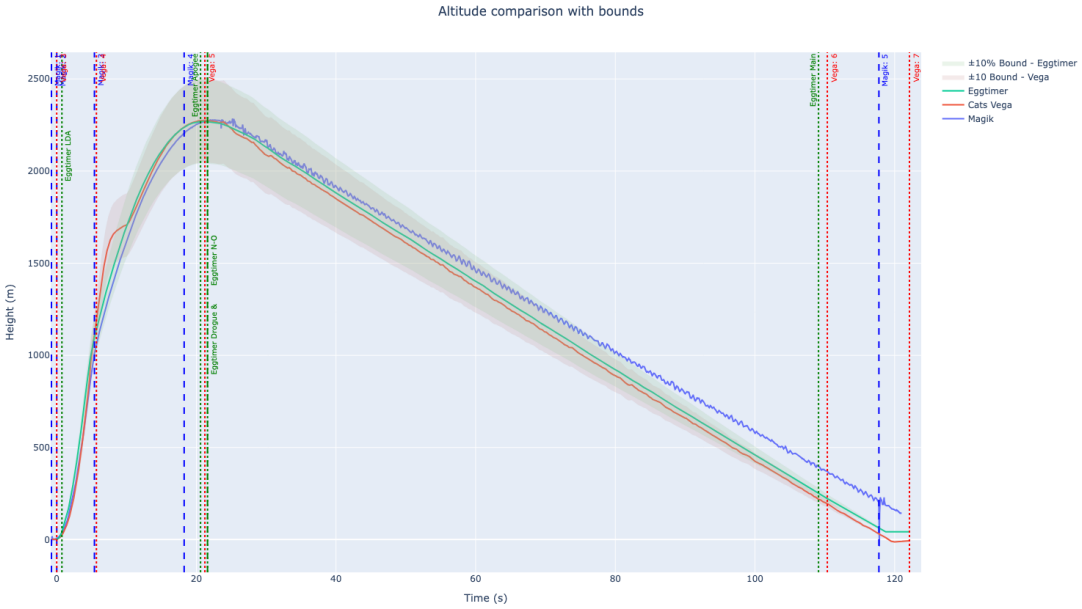

Figure 6.2: Altitude Comparison with ±10% bound

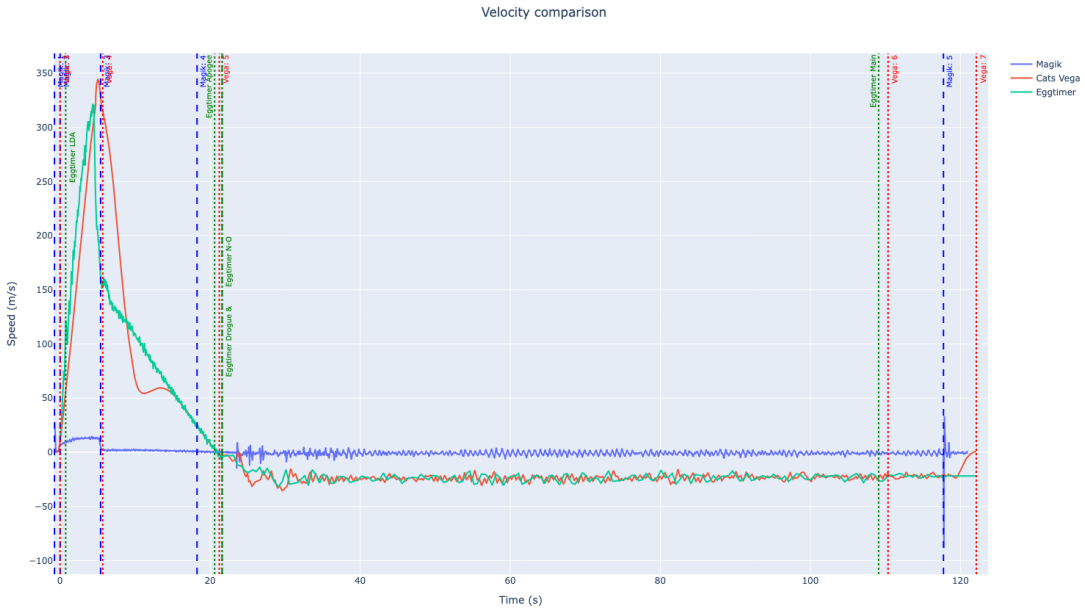

Figure 6.3: Velocity Comparison

using the flight computer in practice as I will control of how my acceleration data is being processed.

6.2 State Recognition

As seen in the graph the Flight computer accurately detected all events except the Main deployment. The Main not occurring when expected is due to the altitude on the decent not being accurate as mentioned in the Kalman Filter section.

The Pico flight computer, marked as Magik on the graphs maps its states to the numbers in the table below.

State	Number
INIT	0
READY	1
POWERED	2
COASTING	3
DROGUE	4
MAIN	5
TOUCHDOWN	6

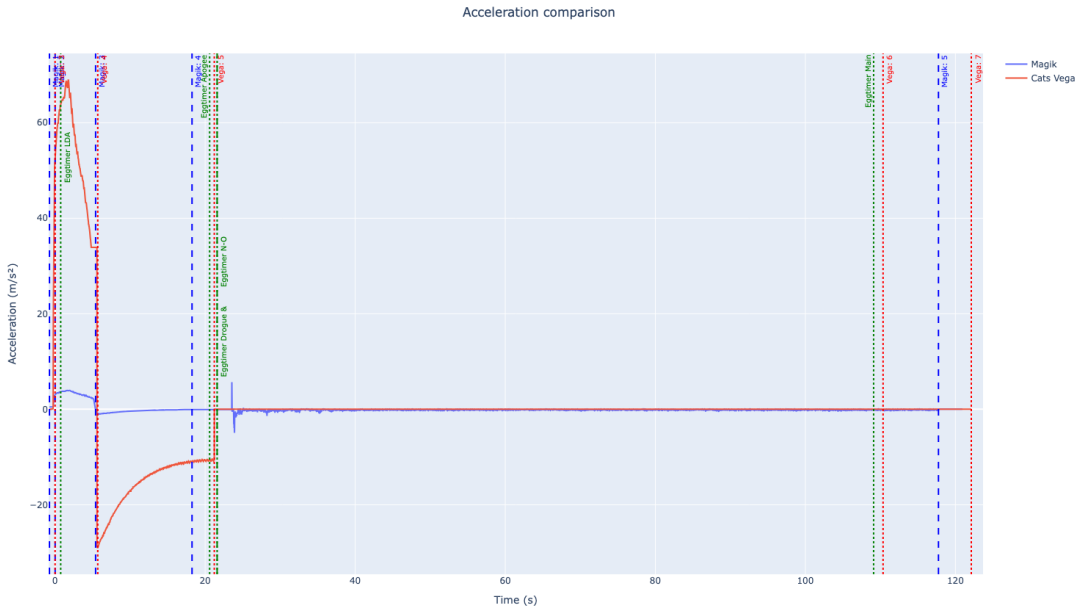

Figure 6.4: Acceleration Comparison

6.3 Settings

As seen the setting themselves were stored correctly. For the Bluetooth server, while a device is able to connect and view the endpoints, the endpoints didn't do anything when called. The callback functions for read and write were not being triggered. This also happened when attempting to debug using the basic Bluetooth example provided by the Raspberry Pi Pico SDK. So I suspect it is a hardware or configuration issue. I will need to investigate this further.

6.4 Sensors

All the wiring for the sensors was correct, except for the secondary accelerometer, which I fixed using the PIO feature. For the C++ code that I was able to complete, I had no issues and the sensors worked as expected. I believe all the other sensors would have worked as expected as well given the Python code worked, and the protocols being used are standard.

7

Conclusions and Future Directions

7.1 Conclusions

Overall I believe this project was a success and I believe that it is "feasible to replicate all of the functionality of a COTS Flight Computer using a Raspberry Pi Pico, while utilise the PIO feature to increase modularity"

I was able to replicate the majority of functionality I defined to be on par with a COTS Flight Computer, and with some small code changes will be able to replicate all the features I defined.

The Kalman Filter was accurate for a majority of the flight, and with further adjustment can be made more accurate. The computer was able to detect the majority of the key launch event, launch, powered flight, coasting, apogee. The main event was inaccurate due to the Kalman Filter.

The computer showed its ability to deploy parachute by lighting up LEDs when the apogee and main events were triggered.

While I wasn't able to confirm the GNSS and radio capabilities due to not having time to write the drivers for the prototype sensors, I believe from using the test handler that the logic and data flow are working, so on once the drivers are working the telemetry would work.

While I wasn't able to fully implement logging, but from having the settings being successfully saved in the flash storage, I expect the logging to work as

expected once some small logic changes are made to the current implementation.

The flight computer settings can currently be saved through editing the code itself, providing the basic functionality needed. With some investigation into the issues in the Bluetooth implementation, it will create a wireless solution for a more advanced user experience.

The status led indicates the state the flight computer is in, providing a simple visual cue without the need to plug in the computer.

For modularity, the 3 broken out pins provide the opportunity for expansion in the future. A display of the modularity is using the PIO is replicate I2C when I miss-wired an accelerometer. This shows how PIO can help when there is a hardware mistake, so that the board itself didn't need altering but instead there was fixed through software

7.2 Future Work

7.2.1 Getting Ready for flight

To get this flight computer ready to be launched, the first step is to produce the designed PCB. Along with that, a ground station needs to be created to receives and display the tracking data needs to be produced.

Next, I will fine tune the Kalman Filter to get more accurate results. I will complete the changed needed for logging and write the drivers needed for the sensors. Once the drivers for the sensors specific to the PCB have been written, additional tests like battery life and radio range testing will need to be complete.

Once this is done, the computer will be flown as part of a CanSat (a small experiment ran onboard a rocket). It will act as tracking, and record all the data to see if it detected launch, apogee and main deployment correctly.

Once data has been collected and performance is deemed OK, it will be used as one of two main flight computers in a launch.

7.2.2 PCB Improvements

The PCB has some updates and improvements that can be done. Including an AND gate before the MOSFETs can be triggered means that a pyro lock can be

added, which will mean increased safety as the pyro lock will only 'unlock' the pyro pins after the powered flight phase has commenced. I implemented this in the prototype with the relay, but haven't added it to the PCB yet.

Continuity checks can be added to the pyro outputs. This gives a way for the team to verify the e-matches are seated properly in terminals and, that they aren't faulty. This works by running a small amount of current, but not enough to trigger them, through the igniters to check that current is passing through them as expected.

A GPS antenna can be mounted to the board. This will improve general ease of use as external GNSS antennas are often bulky and have long cables.

The overall footprint can be made smaller. Often electronics bays are tight on space, so a smaller flight computer is easy to accommodate in a rocket.

Adding external flash memory or an SD card to avoid any risk of erasing program data when logging the flight. This will also reduce the wear on the Pico boards flash memory, increasing its lifespan.

Transitioning from a Raspberry Pi Pico board to just a RP2XXX chip would majorly improve the board in terms of space and modularity. The RP2350B chip has 48 GPIO, increasing the option to have more breakout GPIO and utilise the PIO aspect of this project more.

7.2.3 Software Improvements

Improving error recognition and handling. This would involve having watchdogs in case of restart, and checking sensors aren't freezing. It would also involve displaying more states on the status LED and adding in a buzzer for an auditory cue.

Improving testing. Instead of running everything through a serial connect, I would like mock each sensor and create comprehensive unit testing. They would be able to run on a computer before being flashed to the Pico.

Adding in compile flags. Instead of relying on the config file to define what sensors are used, I would like to add compile flags. This could also be run in a pipeline on GitHub, so when code it pushed, and image is created which can be

flashed to multiple Pico boards. This will allow members of ULAS HiPR to flash Pico's without needing to know how to compile the code.

Implementing the Bluetooth. I would like to find the issue in the Bluetooth server so, the settings can be changed wirelessly. Additionally, a simple app could be created for a nicer user interface.

Implementing the logging fully. To fix the logging, I need to add better handling of the flash memory. This would involve adding bounds to make sure the settings, or previous flight data aren't erased. I also need to implement a method get data off the flight computer.

Implementing an extended Kalman filter. This would improve the accuracy of the Kalman Filter, giving the team more confidence in the computer working as expected.

Appendix

I made a design for a ground station, which receives telemetry data from the flight computer. This used a Raspberry Pi Pico W, and a RA-01H radio. It displays data on a TFT screen. There is a case design as well as a PCB design in the images below.

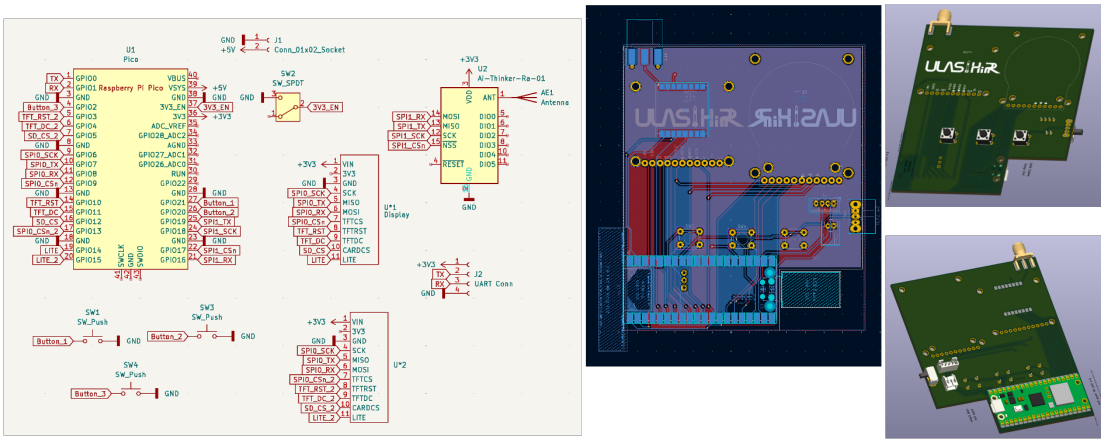

Figure 1: Schematic and 3D render for a PCB for a compatible ground station

Figure 2: Render of a 3D printed ground station case

References

Adams, V.Hunter (2025), 'Gatt server'.
URL: https://vanhunteradams.com/Pico/BLE/GATT_Server.html 26

Altus Metrum (2024), 'Altus metrum'.
URL: https://altusmetrum.org/index.html 12

Becker, A. (n.d.), 'Online kalman filter tutorial'.
URL: https://www.kalmanfilter.net/multiSummary.html vii, 5

Bluetooth (n.d.), 'Assigned numbers'.
URL: https://www.bluetooth.com/specifications/assigned-numbers/ 26

Bosch Sensortec (2013), 'Bmp180 digital pressure sensor'.
URL: https://cdn-shop.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf 4

Cavender, Daniel (2015), 'Nasa high powered video series counterpart documents'.
URL: https://www.nasa.gov/wp-content/uploads/2016/02/sl_video_instruction_book.pdf 9

Control and Telemetry Systems (2023), 'Control and telemetry systems gmbh'.
URL: https://www.catsystems.io 11

Eggtimer Rocketry (2024), 'Eggtimer rocketry electronic altimeter'.
URL: https://eggtimerrocketry.com/ 12

MathWorks (2021), 'Linear kalman filters'.
URL: https://www.mathworks.com/help/fusion/ug/linear-kalman-filters.html 5

MathWorks (n.d.), 'Extended kalman filters'.
URL: https://www.mathworks.com/help/fusion/ug/extended-kalman-filters.html 5

Mcgee, L. and Schmidt, S. (1985), 'Discovery of the kalman filter as a practical tool for aerospace and industry'.
URL: https://ntrs.nasa.gov/api/citations/19860003843/downloads/19860003843.pdf 5

Milligan, V. (2024), 'Phases of a rocket's flight'.
URL: https://www.apogeerockets.com/Tech/Phases-of-a-Rockets-Flight 7

NASA (2023), 'Flight of a model rocket'.
URL: https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/flight-of-a-model-rocket/ vii, 8

National Association of Rocketry (2024), 'High power rocketry - national association of rocketry'.
URL: https://www.nar.org/content.aspx?page_id=22&club_id=114127&module_id=667402 vii, 1, 2

National Association of Rocketry (2025), 'National association of rocketry rocket motor resources - national association of rocketry'.
URL: https://www.nar.org/content.aspx?page_id=22&club_id=114127&module_id=669253 1

NOAA US Department of Commerce (n.d.), 'Pressure altitude calculator'.
URL: https://www.weather.gov/epz/wxcalc_pressurealtitude 4

Raspberry Pi (2024*a*), 'Connecting to the internet with raspberry pi pico w-series.'.
URL: https://datasheets.raspberrypi.com/picow/connecting-to-the-internet-with-pico-w.pdf 6

Raspberry Pi (2024*b*), 'Pico-series microcontrollers - raspberry pi documentation'.
URL: https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html vii, 6

Raspberry Pi (2024*c*), 'Rp2350 datasheet'.
URL: https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf 7

Schultz, D. (2002), 'Barometric apogee detection using the kalman filter a nar research and development report "c" division naram 44, 2002'.
URL: *http://davesrocketworks.com/rockets/rnd/2002/KalmanApogee.pdf*
5