ENGR 302 Avionics Package Technical Manual

Mohammad Al-Rubayee, Ciaran King, Nicholas Lauder, Ikram Singh, Yee Young Angus Tan, Angus Weich, Finn Welsford-Ackroyd

Table of Contents:

1. Introduction

2. Software System

File Structure

The software package is kept in Software_Development/AvionicsPackage. Within this folder there are the following sub-folders:

- /docs: contains various notes on the software system
- /lib_avionics: the embedded software package
- /test: contains the test suite and relevant libraries

2.1 Board Modes

There are three modes that the avionics package can be placed in. These are the *Remote* and *Base Station*, which are programmed onto a physical board, and *Testing* which is compiled and run on a desktop. The mode can be selected by specifying the `IS_REMOTE` variable in `globals.h` to switch between *Remote* and *Base Station*. To enable *Testing*, you must define the `TESTING_ENABLED` variable in `globals.h`. Having `TESTING_ENABLED` overrides any other mode selection. When the `IS_REMOTE` variable is set to true, the code will run as if it were on the board that is connected to the rocket and will be flying. When set to false, it is expecting to be the Base Station. As the base station, the only function component is the RF module, and it is expecting to remain stationary, connected to a computer for the user to read output information from the rocket.

Remote

The purpose of the remote setup is to provide support for the boards that will be place directly onto the rocket. It handles that flight statics, control and logging of data. Currently, the remote setup supports the following components:

- SD Card
- Radio Transceiver
- GPS Receiver
- Inertial Measurement Unit (IMU)
- Servo's

Base Station

The purpose of the base station setup is to provide support for receiving data wirelessly from a remote rocket. The base station should be plugged into a laptop that the user can monitor during a launching process. The information displayed should provide the user with insight to what state the rocket is in, the current reported location, and any necessary debug information that may be beneficial. Currently, the base station only supports the Radio Transceiver module

Testing

The testing mode adds support for the software package to be compiled and run on an x86 based computer. It provides mocks for all the necessary Arduino libraries allowing it to compile without error. This mode allows the user to add sufficient testing to support future development. By being able to run the code from a desktop environment, without the need for a microcontroller, continuous integration can be supported in full.

Settings

TESTING_ENABLED

Supports: `definition`
Where: `globals.h`
This setting specifies whether or not to let the package use the mock Arduino libraries and be compiled to run on a desktop

DEBUG

Supports: `true/false`
Where `globals.h`
The debug setting enables or disables the debugging functionality of the system. The implementation of the debugger will be explained later but effectively this command enables and disables the serial output of the rocket. Having this option is very helpful from a debug standpoint, as println's are the only real method of debugging available on the teensy. But being able to easily disable this output is crucial so that it does not unnecessarily slow down the system during a launch.
NOTE: WHEN THIS IS ENABLED, YOU MUST CONNECT TO THE BOARD OVER SERIAL FOR ANYTHING TO FUNCTION

IS_REMOTE

Supports: `true/false`
Where: `globals.h`
This setting specifies which board mode to be in, as detailed above in the **Board Modes** section.

GPS_ENABLED

Supports: `true/false`
Where: `globals.h`
Specifies whether or not the system utilises the GPS module code. Disable if there is no GPS hardware module connected.

IMU_ENABLED

Supports: `true/false`
Where: `globals.h`
Specifies whether or not to read from, and log the IMU data.

IMU_ENABLED_2

Supports: `true/false`
Where: `globals.h`
Specifies whether or not to read from, and log the second IMU's data.

SD_ENABLED

Supports: `true/false`
Where: `globals.h`
Enables or disables SD Card logging.

RF_ENABLED

Supports: `true/false`
Where: `globals.h`
Enables or disables radio communication between the rocket and the base station.

CONTROL_ENABLED

Supports: `true/false`
Where: `globals.h`
Specifics if the system should perform the calculation necessary for active stabilisation.

GIMBAL_ENABLED

Supports: `true/false`
Where: `globals.h`
Specifics if the system should write positional data to the gimbal

2.2 Software Components

Main

Where: `main.cpp` `main.h`
Author: Nick Lauder
The Main module is essentially a binding module for all other components within the system. It has overall control of what code is being executed and at what time. This module holds both the `setup()` and `loop()` methods required by Arduino. The main module is not an object. But it does handle initialisation of all external libraries and hardware component objects. This module also holds code for some generic debug output. In the `setup()`, all the modules are initialised and checked for a successful initialisation. A successful initialisation **does not** guarantee that the component is function correctly, just that we can communicate with it. The `loop()` function just chooses which state loop to run, but each iteration of the loop still returns to this function. In the case of the base station, the `loop()` function just continuously reads from the radio module.

Debugger

Where: `Debugger/debugger.cpp` `Debugger/debugger.h`
Author: Nick Lauder
The debugging module is essentially a wrapper for the serial prints that are built into Arduino. These functions

are always available for use throughout the software base, but the functionality will only get run if `DEBUG` is enabled in the `globals.h` file. The debugging module has been created in this way so that you can easily print to serial anywhere throughout the software. But you can also easily disable all of this output in situations where performance is paramount. By doing it this way, we are still achieving our performance goals, but we do not have to have precompiler checks for the `DEBUG` setting scattered throughout the system. Ensuring that the system is running quickly and efficiently, while also remaining concise and easy to read. The wrapped methods are `Serial.print()`, `Serial.println()` and `Serial.printf()`. You can essentially thing of the debugger as just, print to serial if `DEBUG` is enabled.

States

The states are only used by the remote rocket. These are used to dictate what the rocket is currently running, and therefore dictates which code to run. Each state has it's own role, and it's own set of conditions that need to be met in order to progress to the next state. The state changing itself is handled by the main component. There are currently seven states: SETUP, STARTUP, IDLE, PRELAUNCH, LAUNCH, POSTLAUNCH, and ERROR.

SETUP Where: `main.cpp` Author: Finn Welsford-Ackroyd SETUP is the initial state of the rocket. It stays in SETUP until the `setup()` method has completed successfully. The `setup()` method initalises all of the componenents of the rocket.

STARTUP
Where: `States/startup.cpp` `States/startup.h`
Author: Nick Lauder
STARTUP handles the checking of the hardware components, different from the `setup()` method in main, the components are actually checked for expected functionality here. The GPS is checked for a lock, IMU checked for real values etc. The state only check a component until it is successful, but will continue checking any specific component until it is verified.

IDLE
Where: `States/idle.cpp` `States/idle.h`
Author: Finn Welsford-Ackroyd
The IDLE state is just a waiting state used for when the rocket is being moved around. Currently, the only way to move on from this state is if the remote station receive the "arm" message over radio. This state allows the rocket to be moved around freely without fear of it thinking it has been launched due to getting dropped etc.

PRELAUNCH
Where: `States/prelaunch.cpp` `States/prelaunch.h`
Author: Nick Lauder
The PRELAUNCH state currently just waits for a significant force on the accelerometer, indicating that a launch has started. This state continuously logs the IMU data and only proceeds to the next state when there is a force of over 3gs. This state will be replaced when the rocket itself initialises the launch process.

LAUNCH
Where: `States/launch.cpp` `States/launch.h`
Author: Nick Lauder
The LAUNCH state handles the rocket while the rocket in actually flying. In this state the control system is being utilised which allows for the active stabilisation of the rocket. During this state both GPS and IMU data is being logged to the SD card but it is not being broadcast. This is because a typical radio broadcast takes

over 150ms, which is far too long for the system to be blocked while performing the active stabilisation. This state progresses after 1.8 seconds, which is the estimated amount of time it takes for the rocket motor to burn.

POSTLAUNCH
Where: `States/postlaunch.cpp` `States/postlaunch.h`
Author: Nick Lauder
This state handles the rockets actions after the burn has completed. So during the free fall and when it's on the ground. During this time, the rocket is logging both IMU and GPS data, but also broadcasting the GPS data to allow for the user to retrieve the rocket if it has flown out of sight. This is the final state and will continue this logging until the rocket is restarted.

ERROR Where: `States/error.cpp` `States/error.h` Author: Finn Welsford-Ackroyd If there are any errors prior to the rocket's launch (e.g. a component failing to initialise correctly) the rocket will be put in an ERROR state. This is a simple while(1) loop which prevents the rocket from progressing further through the states.

Sensor Objects

Where: `SensorObject/sensor.cpp` `SensorObject/sensor.h`
Author: Nick Lauder
There are two generic object within the system that allow for a nice, generic way of reading and storing sensors and their readings. These objects enforce a generic set of methods that can be used by the rest of the system, without needing to care about specifically what sensor it is trying to read from. Currently there is a Sensor object and a Sensor_Data object. The Sensor object is an abstract class that requires certain methods to be implemented by a subclass. Currently these subclasses are the IMU object and the GPS object. The sensor class handles the logic for recording data from a sensor and storing in static memory. It does this by utilising the abstract `read()` method that needs to be implemented in the subclass. The sensor object also has the function `to_string(Sensor_Data)` that converts a Sensor_Data Object into a string, printable, format. Each sensor object has a static memory buffer that is set with the `BUF_LEN` definition set in `sensor.h`.The Sensor_Data object handles the information from a specific sensor reading. It does this by using a UNION of two structs, one for IMU data and one for GPS data. It is done this way to ensure that any Sensor_Data objects take up the same amount of space in static memory. These objects provide a layer of abstraction required by the logging system to handle all of the data in a clean and generic way.

Control

P/PI/PID Control

Where: `Control/control.cpp` `Control/control.h` `Control/pid.cpp` `Control/pid.h`
Author: Ikram Singh
This control module uses a PID controller to stabilise the rocket. Currently implemented and tested but further tuning can be done for refinement.

Quaternion Control

Where: `Control/control.cpp` `Control/control.h` `Control/quaternion.cpp` `Control/quaternion.h`
Author: Ciaran King

This control module uses the concept of quaternions to actively stabilise the rocket. Currently not implement but this is the end goal.

Gimbal

Where: `Gimbal/gimbal_driver.cpp` `Gimbal/gimbal_driver.h`
Author: Ciaran King and Nick Lauder
The gimbal driver is a very basic module used to control the servos that make up the rockets gimbal. The code to determine the positions is located in the Control module, as this module only handles the direct, analogue, communication. This module is where the user would enter their gimbal calibration data, and when a position is requested by the system, this module ensures that the gimbal is not trying to move to an impossible position.

GPS

Where: `GPS/gps_driver.cpp` `GPS/gps_driver.h`
Author: Nick Lauder
The GPS module handles the GPS location and time retrieval. The module we have implemented is a subclass of the Sensor object. To meet the requirements of the Sensor class, the `read()` method had to be implemented. This `read()` method handles the behaviour of reading and translating the raw GPS messages. To do this, it utilises the TinyGPS++ library. https://github.com/mikalhart/TinyGPSPlus. This is a very popular GPS library that handles the inner working of the GPS handling very effectively. Our implementation of the GPS module heavily piggybacks off this. We have connected to the GPS hardware module over a UART interface. During the setup of this module, a number of steps have to be taken in order to prepare the antennas and the UART bus. The pins that are being connected are specific to the hardware board. There is also a sequence in the `check_start()` method. That attempts to initialise the module then prove that there is some communication between the board and the hardware GPS module. It performs this check a maximum of two times. This module also provides the method `get_status()` which provides some helpful GPS debug information such as the amount of satellites found, the number of successful sentences and a few more.

IMU

Where: `SensorObject/sensor.cpp` `SensorObject/sensor.h`
Author: Nick Lauder
The IMU is the Inertial Measurement Unit. This unit include an accelerometer, gyroscope and a magnetometer. This module, like the GPS, makes use of the Sensor object. Implementing the `read()` method in a way specific to reading from the IMU. In order to read from the IMU, we use the BolderFlight MPU9250 library. https://github.com/bolderflight/MPU9250. Although in the future, we would look at implementing the reading and writing of the module ourselves. The IMU module connects to the teensy3.6 over the SPI protocol, which allows for reading of the data at upto 20MHz.

RF

Where: `SensorObject/sensor.cpp` `SensorObject/sensor.h`
Author: Nick Lauder
The RF module handles the transmission and receiving of wireless communication between the remote board and the base station. Our implementation is heavily reliant on the RadioHead RF drivers. https://www.airspayce.com/mikem/arduino/RadioHead/. These drivers provide a generic, abstraction layer for

a huge amount of radio modules. Because of this, the radio module we are using is not required by the code base, and could be fairly easily swapped out with little changes to the code. We connect to the RF module over the SPI protocol. Which allows for high speed communication for broadcasts. In the RF module, there are two key ways of broadcasting some information. The first is just a generic write, which will broadcast a message and just leave it at that. The second is a write with an `ack` requirement. An `ack` is an acknowledgement from the receiver that they have received the message. This will continue to send the same message for a specified amount of attempts, until an `ack` has been received from the base station. It is strongly advised not to use too high numbers for this method as it block the rest of the system until a message has sent.

SD

Where: `SensorObject/sensor.cpp` `SensorObject/sensor.h`
Author: Nick Lauder
The SD card is our primary source of logging on the system. It logs almost all data such as debug info, GPS, IMU and Gimbal positional data. The SD card on the teensy3.6 in built in and runs over the SPI protocol. The main implementation is handled from withing the generic Arduino libraries which allows for very easy reading and writing to the card.

2.3 Testing

Continuous integration is implemented using GitLab CI/CD. Every time a commit is pushed to the repository, a series of tests are run. These include the Google Test/Google Mock tests and a series of PlatformIO compilation tests. The CI jobs are defined in a file names `gitlab-ci.yml` in the root directory of the repository.

Google Test Suite

To ensure the functionality of the code, a test suite was created which uses the Google Test and Google Mock frameworks. The test suite uses Google Test for basic unit tests and assertions, and relies on Google Mock for mocking libraries and tracking function calls.

Google Mock is a framework for creating *mock objects*, which are objects which replace real objects for testing purposes. In the case of the Avionics Package, there is a mock object for each external library. This means the software can be compiled and run in a desktop testing environment, and tests can be written to ensure that the correct functions are called in each stage of a launch.

The test suite files are kept in the /test subfolder of AvionicsPackage, which contains these subfolders:

Arduino-Mock (/arduino_mock)

Author: balp, ikeyasu, Finn Welsford-Ackroyd
A modified version of the arduino-mock library which can be found here. This library provides mocks for most of the core Arduino library. The team ported WString Arduino class to x86 and moved it into this library so that the codebase could continue to use the user-friendly Arduino Strings instead of C char arrays.

Avionics_Mock (/avionics_mock)

Author: Finn Welsford-Ackroyd
This library contains the mock classes for the non-Arduino external libraries. It is compiled into its own library so that the Avionics Package can use the mock objects at run-time while the test suite tracks them.

- **gps_library_mock:** Mocks TinyGPS++ library
- **imu_library_mock:** Mocks MPU9250 library
- **rf_library_mock:**: Mocks RadioHead RF library
- **sd_library_mock:**: Mocks PlatformIO SD library

In order to intercept the calls to external libraries, the team had to templatize the source code as described here. This means each driver class makes function calls to a generic Library object which is specified at compile-time. If the code was compiled with TESTING_ENABLED, mock objects are instantiated and sent to the drivers at run-time. Otherwise, the real library objects are used.

Google Test Suite (/tests)

Author: Finn Welsford-Ackroyd
This contains the actual tests, which are compiled into an executable using CMake (discussed in the next section).

Google Test organises tests by Test Suites and Test Cases. The Test Suite encompasses all tests for a software package, and contains multiple Test Cases. Test Cases are groups of single Tests which are related to (but do not depend upon) each other. Generally, all tests in the Test Suite are run in no particular order from a single command. However, the Avionics Package is designed to run on an embedded system and as such has different testing requirements. The preprocessor statements mean that the system's different configurations are set at compile-time rather than run-time. Also, the wide use of global variables means that the system state is near-impossible to tear down and reset between tests. Therefore, the default runner is not used and is instead replaced with command line calls.

There is a Test Case for each component that needs to be tested - and these tests expect that the code was compiled with only that component enabled. Therefore, the GitLab CI Runner compiles the code with the appropriate pre-processor flags for each Test Case, and then runs each test individually so that the program state is fully reset between tests. This is a achieved with the `--gtest_filter` command line argument, e.g:

```
./avionics_tests --gtest_filter=TestCaseName.TestName
```

Each Test Case has its own .cpp and .h file. The .h file defines the fixture class, which performs set-up and tear-down for each test. The .cpp file implements the tests and helper methods for the Test Case.

CMake Build System

The testing code compiles on x86 desktop/server architecture with the help of CMake, which is a build system for C/C++. CMake links dependencies and compiles files according to a *CMakeLists.txt* script in each sub-directory. This is the structure of the CMake project along with what the *CMakeLists.txt* file in each directory does:

- **/AvionicsPackage**: Downloads and installs GoogleTest and GoogleMock libraries so that all subdirectories can access it as a dependency. Adds */test* and */lib_avionics* subdirectories:
    - **/test**: adds */arduino_mock*, */avionics_mock*, and */tests* subdirectories
        - **/arduino_mock**: compiles arduino-mock library. Also has a */test* subdirectory of its own which is useful when making changes to this library.
        - **/avionics_mock**: compiles avionics mocks into a library.
        - **/tests**: makes tests into an executable
    - **/lib_avionics**: makes the embedded software into a library. As the main library, this has two-way dependencies with all other libraries.

PlatformIO Compilation Tests

Upon each commit, the GitLab Runner tests whether PlatformIO can successfully compile the software in various configurations. These configurations are set by adjusting the enabled modules/board mode using the preprocessor flags:

- Remote Mode
- Base Station Mode
- RF Module Only
- SD Module Only
- GPS Module Only
- IMU_1 Module Only
- IMU_2 Module Only
- Both IMU Modules
- Control Module only

2.4 Other Information

External Libraries Used

TinyGPS++ - https://github.com/mikalhart/TinyGPSPlus BolderFlight MPU9250 - https://github.com/bolderflight/MPU9250 RadioHead RF - https://www.airspayce.com/mikem/arduino/RadioHead/

Precompiler

There is extensive precomplier usage throughout out software system. Primarily, this is used to set configurations for different board setups. These different settings are described in the Settings section of this document. The benefit of using the precomplier is that when we have a setting disabled, the code is completely removed from the compiled system. Resulting in better performance as the package is not having to constantly check for things that will either permanently be disabled or disabled. The downside of using the precompiler syntax is that is can make the code look very messy, especially to someone who is not used to reading and writing around a lot of procompiler code.

PlatformIO

PlatformIO is an open source, microcontroller ecosystem. That has support for a wide range of microcontrollers. Including the Teensy3.6 that we are using. PlatformIO comes in multiple different packages.

Core, which is a command line tool and is the most basic implementation of the tool. **IDE**, which extends the core to a range of graphical tools that is implemented in some existing IDE's such as Atom and Visual Studio Code. **Plus**, which included a range of premium features. Plus is not open source, and therefore was not used in our development tool chain in any way. Primarily using PlatformIO from within Visual Studio Code, we were able to take advantage of the easy to use tools and functionality that PlatformIO added. These included, building the project, programming to the board, connecting over serial. These tools helped considerably with the speed at which we could develop and test functionality

Visual Studio Code

Visual Studio Code is on open source IDE developed by Microsoft. We have use this as our primary development environment as it fully supports PlatformIO, as well as a range of other extensions that make the development cycle a lot easier to maintain.

3. Control System

3.1 Purpose

The control system is a means of stabilising the rocket so that it is able to fly in an upward trjectory. The current implementation of the control system is in the form of a PI control system.

3.2 Control Software

The control software was written in C++ and implemented on a Teensy 3.6 microcontroller. The control code was written using PlatformIO. In order to run the control plant on the Teensy 3.6, the two servos that gimbal the rocket motor must be plugged into the assigned pins on the Avionics Package PCB. The Avionics Package must be powered, either by an external battery, benchtop power supply, and/or a MicroUSB connection for the microcontroller. The IMU(s) must be set up for SPI connection.

3.3 PID Design

The approach of designing the PID controller for the rocket is as follows

- Calibrating the rockets servos.
- Characterising the servos behvaiour with respect to accelerometer readings.

The servos individually have different nominal positions depending on how they were placed within the gimbal so it is crucial to calibrate them with the rockets nominal position as the servos reference frame. This is done by simply setting the servo to its midpoint position (via a PWM signal) and placed in the gimbal.

Characterising the servos behvaiour with respect to accelerometer ADC values is the best way to develop a P/PI/PID controller. Note that currently a PI controller is implemented but the differential portion of a PID controller is in the code if needed in the future. These characterisations are based on a Servo PWM to Acceleromet ADC value relashionship. Since there are two servos, then each servo is assigned to a particular axis respective to the accelerometer and the follwing characteristics are obtained

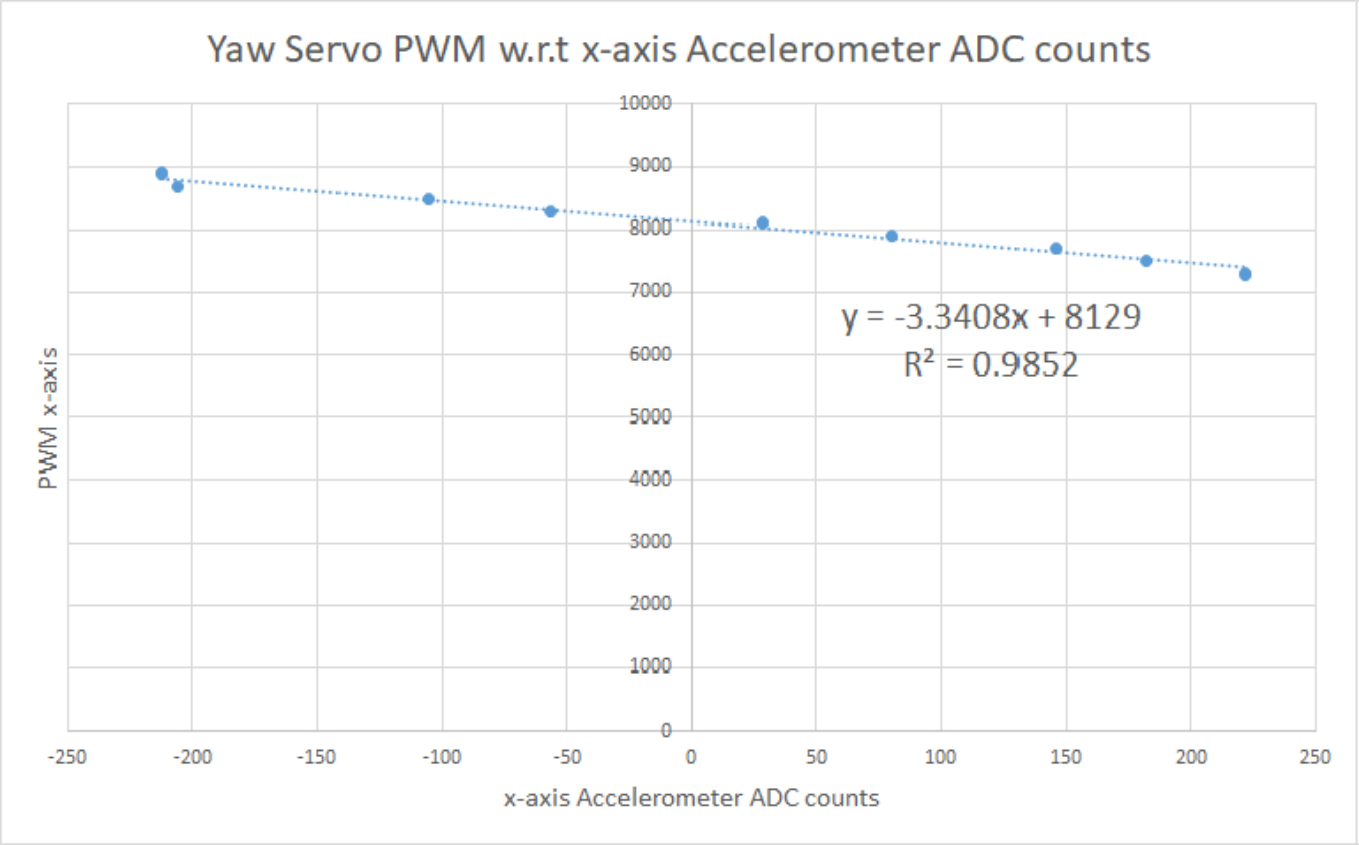

Figure 1: Yaw servo characteristic with respect to z-axis accelerometer ADC values.

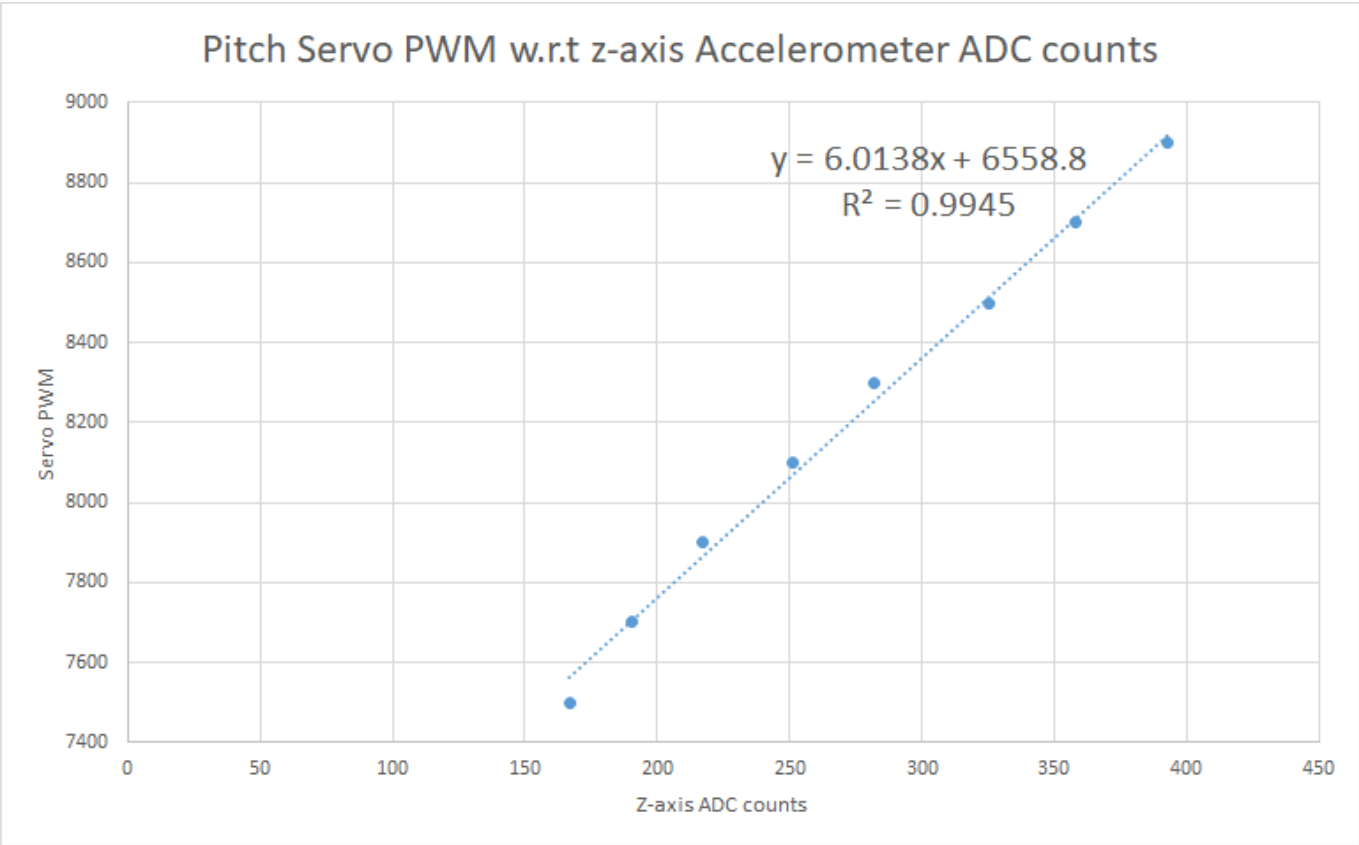

Figure 2: Pitch servo characteristic with respect to x-axis accelerometer ADC values.

Note that the mapping performed in these characteristics is actually the inverse of the Yaw/Pitch servos actual behaviour since the objective of the control system of the stabilise the rocket which is done by negative feedback. The linear regression in both cases are a strong fit as the correlation is above 98% for both

characteristics and thus the PI controller should be able to control the rocket nicely. In other words, the lines of best fit equations are the 'plant' of each servo and as such, each servo has its own PI controller so that the Yaw and Pitch axis are controlled independently with minimum conflict between each other.

The code written for the PI controller can be found in `Control/pid.cpp` `Control/pid.h` and is designed to minimise the error between discrete acccelerometer ADC samples and the servos setpoint. Due to the sensitivity of the accelerometer, a FIR (Finite Impulse Response) filter is performed on 1000 ADC readigns and this will achieve a more accurate representation of the true ADC value provided by the accelerometer. The following figures are the current control systems for the Yaw and Pitch axis of the rocket.

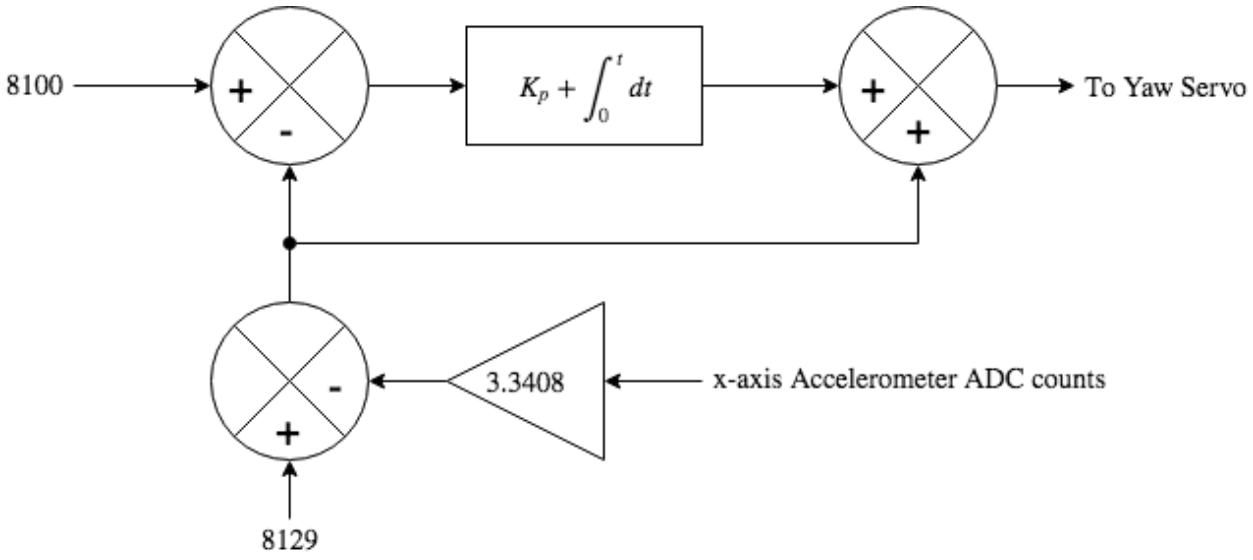

Figure 3: Yaw axis control system.

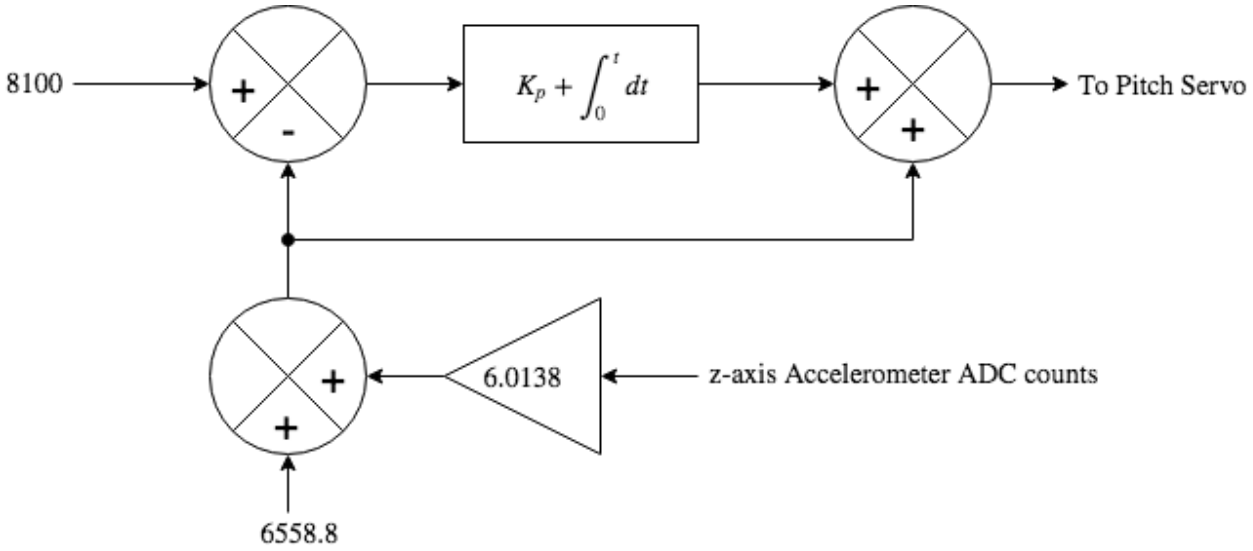

Figure 4: Pitch axis control system.

3.4 Testing and Performance

The stabilisation of the rocket is very delicate but this can be fixed simply by tuning the P/PI/PID controller paramters Kp,Ki and Kd. For the rocket, these values were chosen to be such that

- There is little to no overshoot in the transitioning of the servio to its setpoint position.

- The rise time and settling time of the servo is relatively quick.
- The closed loop response of control systems (i.e. figures 4,5) is critically damped.

These parameter values can be easily accessed and modified in the code.

3.5 Evaluation and Future Work

The PI controller functions just as designed and it stabilises the gimbal to its set point which is the rocket being in its upright position.

Future work regarding the rockets control system(s) would be to refine the P/PI/PID parameters so that different system resonses can be obtained and chosen. Finding a way of smoothing the ADC values from the accelerometer would be beneficial as currently a FIR filter is used which is expensive but fortunely for the rocket the FIR filter functions as expected.

4. Recovery System

4.1 Purpose

The recovery system of the rocket contains the microcontroller programming for the GPS system, the GPS module and antenna, the radio transmitter, and the data logging during a launch. The PCB provides the system required to ensure that these components are operational and can be tested. This document details the motivation for this package, how to implement it, any major design issues encountered, and finally any potential future work to build on. Typically with larger scale amateur rockets, flight distances often range in the kilometres. Locating the rocket afterwards can be a serious issue. To mitigate these long-range detection issues, a recovery system was implemented into the rocket package. This module provides convenient positional information of the rocket's whereabouts during and after a launch. A GPS lock is obtained during start up and tracks the location of the rocket. Furthermore, it can log data into the MicroSD card slot provided with the Teensy 3.6 microcontroller. The system was made robust to handle possible failures such as final orientation of the rocket after landing, long range radio systems for obstructive terrain, and the inclusion of a helical antenna for maximising range. The system can be used by uploading the appropriate code to the avionics package. This system should not need to be reconfigured and rebuilt if the RFM96W is kept as the radio transmitter.

4.2 Features

- Long range RFM96W 433MHz radio system transmitting consistently and constantly to provide a signal to track and detect. It provides an operating range of at least 600 metres in an obstructive and involved environment (multiple concrete and metal buildings). This is based on testing performed by the 2017 project team.
- Transmitted GPS coordinate data of the rocket system, as well as operational time and signal strength.
- GPS achieves lock within 10-15 minutes
- Single antenna GPS system to allow for maximal coverage and chance for the GPS system to get useable satellite signals.
- The Recovery System is fully enclosed, making it more robust for outdoor use.
- Micro SD card writes at every time a message is initialised, allowing for diagnostic information of the rocket to be tracked and maintained.
- Serial output receiver system on base station that can pick up the transmitted data remotely from the rocket.

4.3 Evaluation and Future Work

The system has been end to end tested in the lab, it was succesful and perfromed as expected however it has never been tested on a real flight. Real world testing will be required for an accurate evaluation.

A major limitation of the recovery system is that telemetry is not being broadcast while the control system is running. This is because transmitting a chunk of data over the radio takes on the order of three hundered IMU reads and is a blocking operation on the main thread. Transmitting data in this way would be disasterous to the control system. Future work would include re writing the radio software interface to be non blocking. The micro controller we are using is capable of non blocking SPI so an asynchonous radio driver could be written but this would require a lot of work and likely a retooling to move away from the ardiunio environment.

5. Printed Circuit Board

5.1 Design

The printed circuit board (PCB) of the rocket was designed using KiCad, an open source PCB design software. Because many of the components used are non-standard no libraries exist and so the footprints needed to be custom made using the information in the data sheets.

The footprint of the board was designed to fit within a standard model rocket airframe which has an internal diameter of 30mm. To meet this the PCB was designed to be 29mm wide, the length was not a constraint as the majority of model rockets are quite long.

5.2 Features

- Teensy 3.6
- Maestro A2200-A GPS
    - Connected to Teensy by UART connection
    - Pullup resistors used to to select communication interface
- RFM9x LoRa Radio Unit
    - Connected to Teensy by SPI connection
    - SMD footprint used to reduce space
    - Wire whip antenna for easy mid-range communication
- MPU-9250 InvenSense IMU
    - Connected to Teensy by SPI connection
    - Two QFN-24 footprints used to reduce space
    - SparkFun breakout footprint included as backup
- Two servo footprints
    - Standard JST footprints used
- Ignition system (Untested)
    - Uses three dual N-channel MOSFETs to allow ignition of three different charges with backups
    - Includes reservoir capacitor to prevent board brownout
- Regulated Power Supply
    - Implements P-channel MOSFET for reverse polarity protection
    - Low dropout voltage regulator provides constant 3.3V from battery
- Ferrite Beads
    - Used on high sensitivity components to suppress high frequency noise in the power lines

- Communication Bus Breakouts
    - SPI and I2C bus breakouts included for future development
- Mounting Holes
    - 2mm mounting holes included in each corner for easy installation

5.3 Evaluation and Future Work

- Correct faulty grounding on QFN-24 footprints
    - Pin 20 to GND
- Redesign ignition system
    - Regulate voltage to supercapacitor to prevent overcharging
    - Update supercapacitor footprint to fit provided 1F capacitor
    - Rewire reservoir supercapacitor to properly act as reservoir
- Add indicator LEDs
- Replace radio antenna footprint with larger, more convinient through-hole
    - Alternately replace wire whip for more reliable long range communication

6. Rocket Designs

The rocket designs were first modelled in OpenRocket to determine stability during flight. OpenRocket is an open-source model rocket simulator that allows designing rockets before building and flying them. The parts were then converted to CAD models to be 3D printed using OnShape, an open-source cloud based, CAD program. Rocket parts were manufactured by 3D printing, using ABS as the material. The parts were manufactured at ECS, either by the ECS technicians or by the 3D printers available for use. Using ABS presents two advantages over PLA, the first being its slightly lighter weight. Secondly, ABS has a higher melting temperature of 210-240C compared to 180-220C for PLA which deforms at above 60C.

6.1 Launch 1.0 - Jesse

Design

Marginally passively stable with active stabilisation, single stage, D-motor rocket. Active stabilisation is accomplished using the MK4 Gimbal. It is aerodynamically stable due to its conic shape that greatly increases the drag force that the rocket experiences. A square hole was placed adjacent to the motor mount to fit the rocket on a launch rail. This allows for the rocket to have a vertical initial trajectory. While this does reduce the aerodynamic stability of the rocket, due to the low apogee point this was deemed to be an acceptable drawback. The gimbal was mounted to a ring attached to six evenly distributed structural supports attached to the bottom of the cone. Screws were used to securely hold the gimbal to the airframe. The motor mount is incorporated directly into the gimbal in the centre of the cone and a hole is placed in the cone directly above the motor to vent the ejection charge. An ejection chamber above the large cone allows the ejection charge to safely discharge without damaging the battery.

The avionics package is powered by a 2S 7.4 V LiPo battery which is fitted at the nose in a sealed-off compartment in order to protect it from impacts. The mount for the PCB and sensors is placed at the bottom of the rocket for easy access, next to the gimbal. The motor used for this rocket is a D-class model rocket motor.

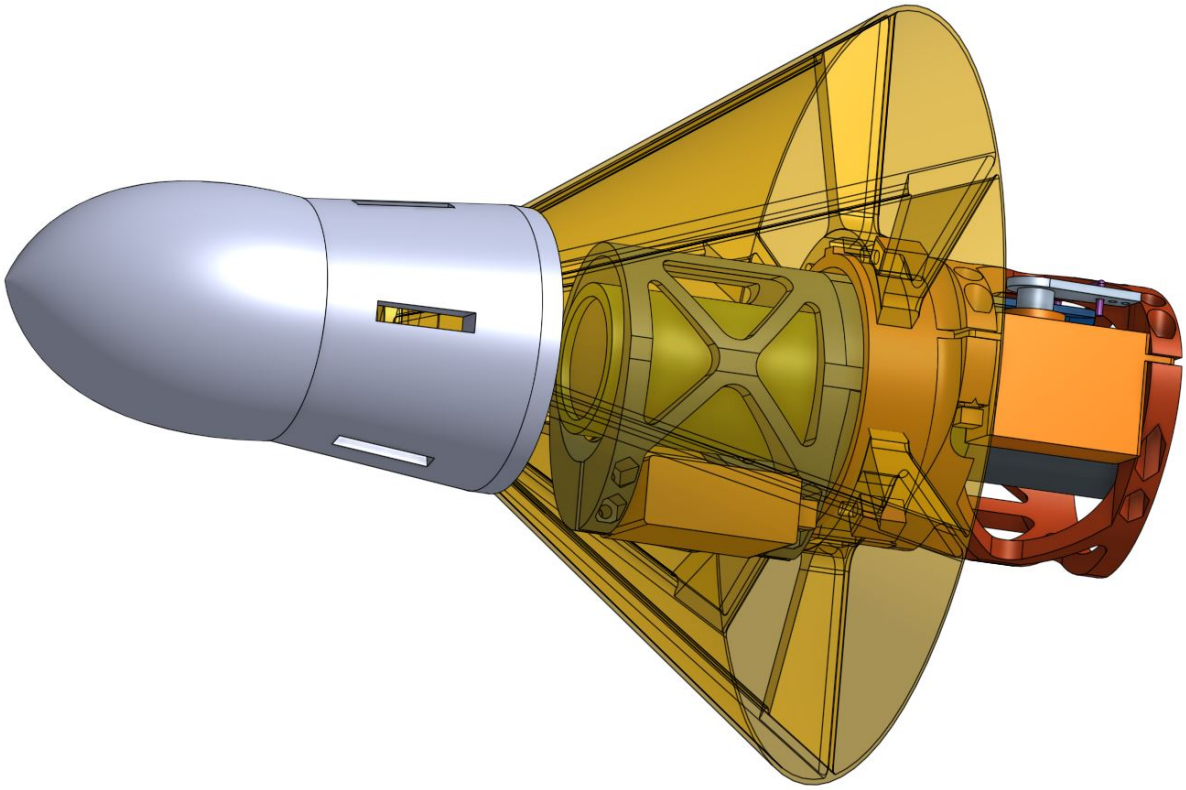

Figure 5: OnShape model for first launch

Hardware

```
* Avionics Package
* 2S LiPo
* 2X M3x6mm
* 2X M3x10mm
* 2X Silicone tubing 10cm
* Electrical tape
* Superglue
```

N.B. Hardware for the gimbal is listed seperately in the Gimbal Documentation section near the end of this document.

Evaluation

The airframe is aerodynamically stable due having a large cone shaped base, centre of gravity and centre of pressure were very close, however. Having the cone meant that once on its initial trajectory, it would be difficult for the gimbal to control the flight path. Centre of gravity and centre of pressure were calculated using OpenRocket to ensure the launch vehicle is marginally aerodynamically stable in order to show the control system is capable of stabilisation. The rocket has an overall height of 286 mm and weighs 600 g when fully assembled.

- Centre of gravity: 13.75 cm from the base of the rocket when standing upright
- Centre of pressure: 12 cm from the base of the rocket when standing upright

6.2.1 Launch 2.0 - James

Design

Passively stable with active stabilisation, single stage, D-motor rocket. The body and nose of the rocket have been extended to have a longer, more aerodynamically design. The PCB and sensors are housed inside the main fuselage of the rocket for improved balance and aerodynamic shape. Active stabilisation is accomplished using a gimbal based off the MK4. The revised gimbal design balances the weight offset from the servos and the addition of cut-outs decreases the weight while retaining structural strength. A square hole was placed adjacent to the motor mount to fit the rocket on a launch rail. This allows for the rocket to have a vertical initial trajectory. The gimbal was mounted to a ring attached to six evenly distributed structural supports attached to the bottom of the cone. Screws were used to securely hold the gimbal to the airframe. The avionics package is powered by a 2S 7.4 V LiPo battery which is fitted at the nose in a sealed off compartment in order to protect it from impacts.

Hardware

```
* Avionics Package
* 2S LiPo
* 8X M3x10mm
* 2X Silicone tubing 10 cm
* Electrical tape
* Superglue
```

N.B. Hardware for the gimbal is listed seperately in the Gimbal Documentation section near the end of this document.

Evaluation

The airframe is aerodynamically stable due having a large cone shaped base. Having the cone meant that once on its initial trajectory, it would be difficult for the gimbal to control the flight path, thus the design was revised to reduce the cone area. Centre of gravity and centre of pressure were calculated using OpenRocket to ensure the launch vehicle is marginally aerodynamically stable in order to show the control system is capable of stabilisation. The rocket has an overall height of 479 mm and weighs 560 g when fully assembled.

- Centre of gravity: 21.2 cm from the base of the rocket when standing upright
- Centre of pressure: 14.1 cm from the base of the rocket when standing upright

6.2.2 Launch 2.1 - Meowth

Design

Nearly identical to Launch 2.0, but for some small key design changes. The cone has been redesigned to slope into a larger tube section rather than tapering to a point. This reduces the stabilising effect of the cone which will allow the control system to better affect the flight path. The fuselage section housing the electronics has also been extended by 2 cm to fit the slightly longer revised PCB.

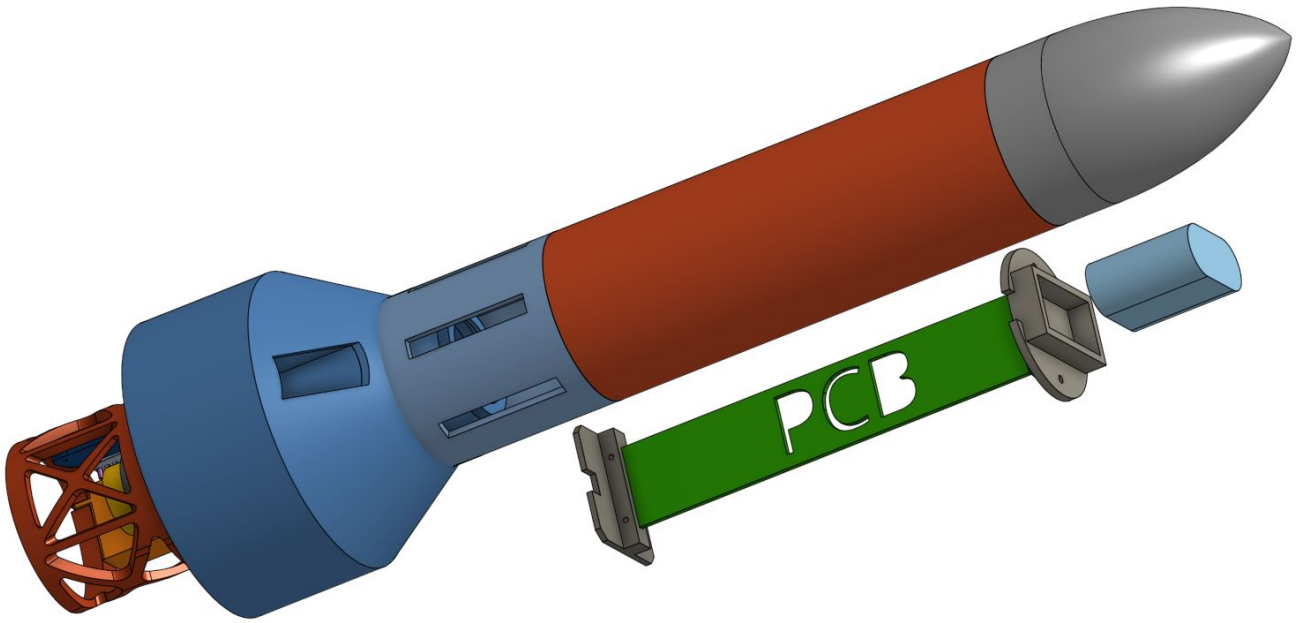

Figure 6: OnShape model for second launch

Hardware

```
* Avionics Package
* 2S LiPo
* 8X M3x10mm
* 2X Silicone tubing 10 cm
* Electrical tape
* Superglue
```

N.B. Hardware for the gimbal is listed seperately in the Gimbal Documentation section near the end of this document.

Evaluation

The airframe was revised from the previous version to reduce the area of the cone, thus improving the amount of control the gimbal orientation has over its flight trajectory. Centre of gravity and centre of pressure were calculated using OpenRocket to ensure the launch vehicle is marginally aerodynamically stable in order to show the control system is capable of stabilisation. The rocket has an overall height of 493 mm and weighs 530 g when fully assembled.

- Centre of gravity: 24.9 cm from the base of the rocket when standing upright
- Centre of pressure: 23.1 cm from the base of the rocket when standing upright

6.3 Future Work

Currently, in order to access the gimbal, the servo horn for the upper servo (pitch) has to be disconnected. The support structures on the airframe can be modified so their distance from the gimbal is increased,

improving ease of access. The lower section of the gimbal (yaw) can be refitted to accomodate this by extending the extrusions to better secure the motor gimbal structure.

7. Rocket Gimbal

7.1 Purpose

Amateur rockets are typically flown with only passive control element such as the shape of the airframe and the stability of the rocket. Passive elements will normally provide an approximately straight flight path but can be prone to disturbances such as winds. These disturbances can often cause changes to a rocket's flight course, especially low altitude winds near the launch site. Additionally, the passive control elements do not provide control at low speeds which causes the rockets to rely on a launch rail when launching. The motor gimbal system will provide an active control element so that the rocket can react to disturbances. This gimbal will also remove the rockets requirement for the launch rail meaning that the rocket can be launched from various locations including airborne platforms.

7.2 Operation

The gimbal system is built upon the gyroscope concept, using two rings that have a pivoting axis for either x rotational movement or y rotational movement. Using this, the gimbal has the ability to freely rotate the rocket motor through pitch and yaw. This motor movement will then give the rocket the ability to make any course corrections that the control system instructs. Servo motors are used to rotate the gimbal rings via control rods connecting the servo horns with the motor gimbal rings.

7.3 Design

The gimbal is manufactured by 3D printing. ABS or PLA is a suitable material. It is recommended to use an infull level between 20 and 65%. This ensures structural strength while reducing the weight of components. The rotational movement is driven by two Hc2422T servos, one per axis. These servos are to be connected to ports 4 and 5 on the PCB, at this point of development the orientation of the servos (X and Y) is undetermined. The gimbal rings are constructed using an ABS 3D printer. The gimbal rings are assembled using four M3x10mm screws where the smaller rings are placed inside the larger rings. The inner ring is orientated so that its control arm is parallel to the servo placed in the outer rings servo housing. The motor mount ring is then oriented so that the majority of its mass is on inner rings servo housing side. The servos are mounted using a collection of M3 screws. The rings are then connected to their adjacent servos using a 5 cm control rod. The material used was 1.6 mm diameter steel rod. This rod is bent and threaded through the control rod holes in both the servo control arms and ring control arms.

7.4 Launch 1 - MK4.2

The rocket is actively stable with the use of the gimbal with its outer ring and support structures directly connected with the rocket body. The gimbal used for this launch is the G-Max MK4.2, designed by the 2017 team. The gimbal's outer ring is screwed into the airframe's support structures. The gimbal motor mount is designed to fit an E-G class model rocket motor. The motor mount was revised to include a step to prevent the motor from moving around during flight. A motor adapter was used to allow the gimbal to fit a D-class motor.

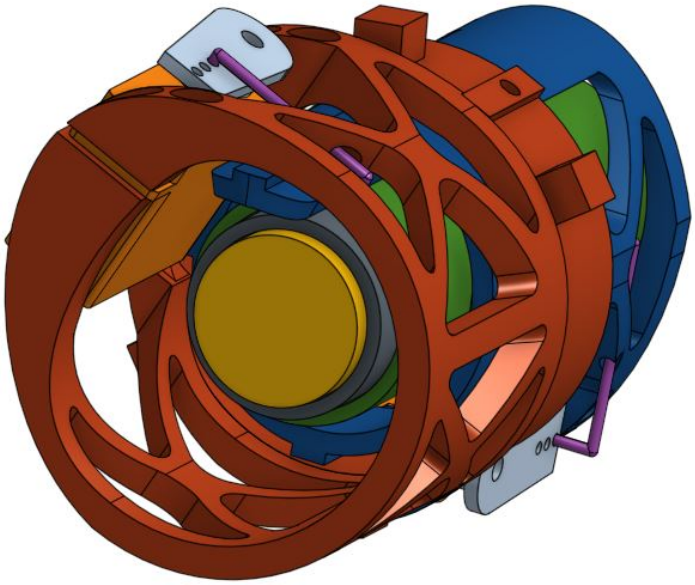

Figure 7: OnShape gimbal model for second launch

7.5 Launch 2 - MK4.3

The gimbal used for this launch was based on the G-Max MK4. It was revised to reduce the weight from 60 g of the MK4.2 to 45 g. The weight was reduced by adding cutouts and reducing the infill level; increasing the apogee and stability of the rocket. The upper servo was shifted to balance its weight along the centre of the gimbal and increase the range of motion of the motor by making the gyro rings thinner. The gimbal's outer ring is screwed into the airframe's support structures. The gimbal motor mount is designed to fit an E-G class model rocket motor. The motor mount revision from the first launch was retained to prevent the motor from moving around during flight. A motor adapter was used to allow the gimbal to fit a D-class motor. The two axes of movement, pitch and yaw, were marked as such on the gimbal next to the servos - embossed into the gimbal as part of the CAD design.

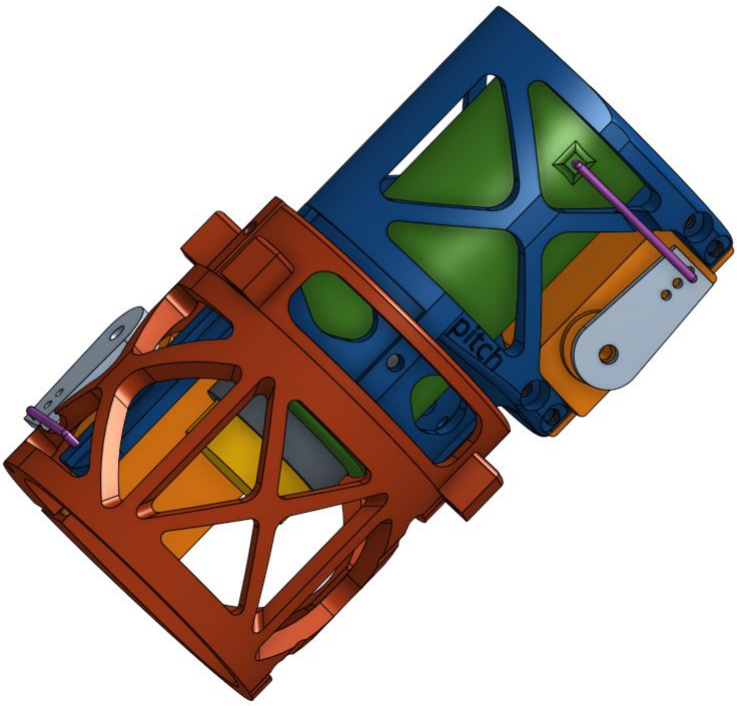

Figure 8: OnShape gimbal model for second launch

7.6 Future Work

Currently, the gimbal has a excess of space on both axes of movement on either side of both axes. It is possible to optimise size taking the servos into account. By making the gimbal narrower, the overall weight and size can be reduced.