WINSAT COMMAND, DATA, AND RADIO FREQUENCY

Submitted in partial fulfillment of the

requirements for the course

ELEC-4000: Capstone Design Project

Department of Electrical and Computer Engineering

University of Windsor

August 2020

WINSAT COMMAND, DATA, AND RADIO FREQUENCY

By

Grebe, Jon 104371501

Simard, Pierre 104589669

Al-Khazraji, Adam 104245871

Frim, Justin 104109690

Faculty Advisor: Dr. Rashidzadeh

Department of Electrical and Computer Engineering

University of Windsor

August 2020

WINSAT COMMAND, DATA, AND RADIO FREQUENCY

“No action by any design team member contravened the provisions of the Code of Ethics and we hereby reaffirm that the work presented in this report is solely the effort of the team members and that any work of others that was used during the execution of the design project or is included in the report has been suitably acknowledged through the standard practice of citing references and stating appropriate acknowledgments”.

The presence of the author's signatures on the signature page means that they are affirming this statement.

Grebe, Jon	104371501		August 7, 2020
Simard, Pierre	104589669	Pierre Simard	August 7, 2020
Al-Khazraji, Adam	104245871	Adam	August 7, 2020
Frim, Justin	104109690	Justin Frim	August 7, 2020

Abstract

This report includes an overview of the development, design, simulation, and implementation of the Payload, CDH, and RF subsystems onboard the WinSAT CubeSat satellite. The WinSAT team's objective is to build a 3U Earth observation satellite for the CSDC, and this report covers the design of three of the onboard submodules.

For the Payload subsystem, the team selected the hardware, including computer, camera modules, and cameras lenses, and developed the software to facilitate both satellite mission objectives of a primary single-point access and secondary area or global coverage. The team working in AGI STK simulation software to perform access and coverage analysis and to verify camera parameters. A web application was also built for public access to satellite imagery.

For the CDH subsystem, the team selected the hardware and software architecture for the main onboard satellite computer. The flight software was developed to provide communication between and integration with all other satellite subsystems. The software manages onboard data, commands, and telemetry. The team also developed lower-level hardware interfaces as well as the ADCS system software.

Regarding the RF subsystem, the team simulated antenna characteristics and worked on developing a communication model based on the given results. Physical satellite antenna and deployment mechanisms were designed to receive and transmit all data. The RF electrical components were selected and configured to process signals for given frequencies.

The team was able to implement the software design into a "Flat-Sat" test setup. This provides all of the connections and interfaces between all satellite subsystems without the components being housed in the final satellite structure. This allows for testing of the flight software as well as integration testing. This implementation demonstrated that the design met all of the CSDC competition requirements.

Table of Contents

Abbreviations

Abbreviation	Description
ADCS	Attitude Determination and Control System
CDH	Command and Data Handling
EPS	Electrical Power System
RF	Radio Frequency
WinSAT	University of Windsor Space and Aeronautics Team
CSDC	Canadian Satellite Design Challenge
MCC	Mission Control Center
ARO	Amateur Radio Operator
RTOS	Real-Time Operating System
MBM2	Motherboard Module 2
BBB	Beaglebone Black
LEO	Low-Earth Orbit
P-POD	Poly Picosatellite Orbital Deployer
OBC	Onboard Computer

Table of Figures

1 Introduction

This project is part of the WinSAT team's objective to design, build, and test a 3U earth observation cube satellite (CubeSat) for the CSDC competition. The CSDC is a Canada-wide satellite design competition involving a number of universities across the country involving the construction of a CubeSat for Earth observation in low-earth orbit. CubeSat's are small satellites, typically made of cheaper components, that allow easier access for companies and universities to space and space research through a simple, modularized platform. The main objective or primary operation of this specific satellite involves the acquisition of a "Space-Selfie" image whereby amateur radio operators around the world are able to establish contact with the satellite during a pass, request an image of their current coordinates, and then have that image downlinked to them for immediate viewing. The CSDC guidelines [1] have provided a concept of operations of the primary mission objective.

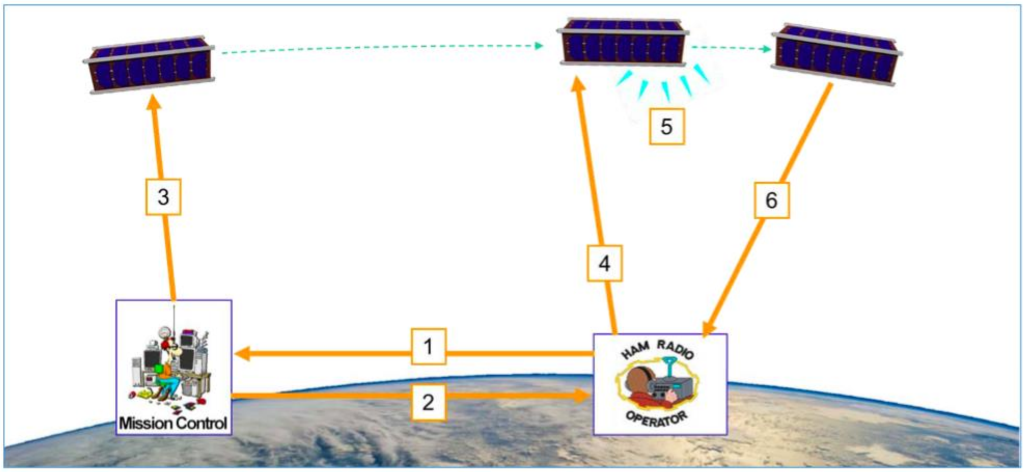


Figure 1 - Selfie-Sat concept of operations [1].

The figure can be summarized in the following steps as specified in the CDSC rules and requirements [1]:

1. Once Selfie-Sat has been commissioned and is ready for nominal operations, ARO's from around the world will contact the MCC at the University of Windsor in order to request a "Selfie-Sat Pass". The operator will send the location (latitude, longitude) of the centre point of the desired space selfie image.

2. The MCC will reply to the ARO with information about the Selfie-Sat Pass start and end times, and a password that the ARO will use for the pass.
3. They will uplink the required roll angle (≤ 25 deg) for Selfie-Sat to be able to point the camera boresight (middle of the image) at the ARO's specified location during the pass.
4. During the pass, the ARO will contact Selfie-Sat, and will uplink the command to acquire an image when Selfie-Sat is over the desired imaging area. Only one image will be acquired during a Space-Selfie Pass.
5. As soon as the command is uplinked, Selfie-Sat will immediately take the "space-selfie" photo with the Selfie-Cam payload imager.
6. Selfie-Sat will then immediately begin to downlink the Space-Selfie photo to the ARO.

The project goal is to support these mission operations through the design, building, and testing of three of the onboard satellite subsystems. This project involves the design of the following subsystems:

1. **CDH** – handles all onboard data and communication between all subsystems on the satellite.
2. **Radio** – handles all command and data transfer and overall communication between the ground station and satellite.
3. **Payload** – includes the onboard imagers or cameras for acquiring satellite imagery as well as providing the interface between the cameras and the rest of the satellite.

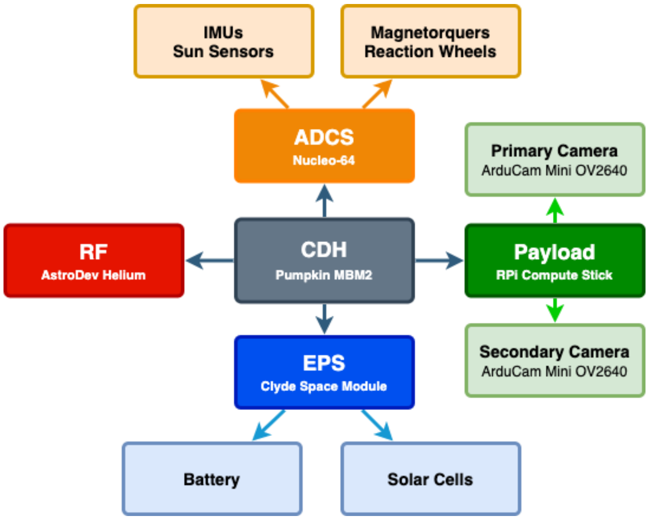


Figure 2 - Satellite system architecture.

1.1 Payload

The payload subsystem incorporates the onboard imagers for Earth observation as well as the hardware and software needed to interface the payload imagery to the rest of the satellite as well as control the onboard imagers. As outlined previously, the payload directly services both the primary and secondary mission objectives:

1. **Primary Mission Objective:** Acquisition of a single-point access “Space-Selfie” image for downlink to ARO’s.
2. **Secondary Mission Objective:** Acquisition of several images in sequence to be filtered and stitched to provide area coverage or global coverage.

The payload design involves the selection of imagers to service both of these mission objectives and provide the software and hardware design of the payload module that will control the cameras and prepare them for downlink to ground station operators.

1.2 CDH

The purpose of the CDH subsystem is to be the main “brain” of the satellite by coordinating all telemetry, commands, and other data between all subsystems onboard the satellite. The CDH subsystem is the main interface between all subsystems and performs the decision-making for the satellite. The goal involves the selection and implementation of a hardware and software design that will allow for safe and robust communication with all of the subsystems onboard the satellite and seamless integration with all of the components.

1.3 Radio

Establishing a good communication link between the ground station and the satellite is essential to all CubeSat missions. CDH, EPS as well as the Structural divisions are directly affected by the performance of the RF communications. As the satellite reaches a certain elevation angle, the ground station will begin to uplink commands and the satellite will downlink its imagery data of the Earth from the previous pass in accordance with the mission objective. To ensure that the satellite and ground station relay all data to one another, proper transmission and reception must be provided by antennas and radio frequency electronics to properly interpret the data. The RF and Structural divisions are both responsible for the construction of the antennas and their release mechanisms, ensuring proper orientation upon deployment.

2 Benchmarking

CubeSat satellites have been used in countless previous missions for a number of different research objectives and goals. Much time was spent looking into previous CubeSat designs to weigh the current state-of-the-art and technologies in the area to allow for a more informed decision regarding the design of the WinSAT CubeSat.

2.1 Software Frameworks

[Jon]

Previous CubeSats have used a variety of different onboard software and hardware architectures for the main onboard computer with advantages and disadvantages for each configuration that are more suitable for certain mission objectives and requirements. A thorough investigation was completed into previous software frameworks used in previous missions that now have flight heritage. A recent publication [2] summarizes the most popular designs used on some previous CubeSat missions:

1. *Nasa's Core Flight System (cFS)*
    - software framework for satellite missions designed and maintained by NASA
    - significant flight heritage and reliability
    - 3-layered architecture with several layers of abstraction above the RTOS
    - Requires significant development above lightweight and minimal architecture
2. *FreeRTOS*
    - Open-source, real-time operating system for embedded systems.
    - Large open-source community and documentation
    - Very good performance and reliability
    - Not specifically designed or intended for satellite use, so it would require significant development of satellite specific functionality
3. *KubOS Linux*
    - Lightweight, Linux distribution made specifically for satellites
    - Provides ease of use of Linux which handles all lower-level functionality
    - Includes tooling, community, and development support that comes with Linux

- Comes pre-packaged with general satellite functionality common to all satellite missions (logging, telemetry database)
- Allows for higher-level development (i.e. Python)
- Requires much higher RAM/ROM resources due to Linux being heavier than a simple RTOS.

2.2 Hardware Protection

[Adam]

Hardware protection needs to be in place for processors, memory, and internal device communication. Most safety features seen in embedded systems in the automotive industry are needed on top of material protection for equipment to function in the atmospheric environment. NASA [3] writes about the hardware protective measures they find critical for reliable operation of small spacecrafts and especially with CubeSats:

1. *CPU Lockstep*
   - Controllers have multicore processors where each core will run the same set of instructions and make sure they generate the same output before proceeding
   - Needed for controllers operating in space as equipment is susceptible to bit flipping (bit errors) due to high energy particles colliding with the circuitry
2. ECC Memory
   - Error correcting code that checks memory devices (RAM and FLASH) for bit errors
   - Code compares memory values to a checksum (compressed image of the memory using hashing algorithm)
   - Memory blocks will be locked if error is detected and will be corrected using the checksum
3. CRC
   - Cyclical Redundancy Check for communication and memory error checking
   - Data is divided by a polynomial (modulo division) and the remainder is used to verify the data after transmission through a communication protocol or after writing the data to memory

    - CRC is calculated then appended to the data payload before transmission so the receiver can compare CRC field of data packet to its payload field ensuring data is valid
    - CRC is calculated before writing to memory devices so that the memory can be verified after written
    - CRC can’t be used to correct code if an error is detected with it (unlike checksum)

4. Watchdog timers
    - Timer circuit that must be “fed” by the processor or the watchdog will reset the controller to refresh the processors in case something went wrong
    - Clock drivers of watchdog timers can be internal (resistor capacitor circuit) or external (crystal oscillator)
5. Rad-Hard
    - Radiation Hardened components are modelled after COTS (commercial off-the-shelf) devices that developers use on test benches
    - Protects from single-event upsets (logic errors) and single-event latch-ups (transistor malfunctions)[3] by reducing the total ionizing dose the components get from high energy particles in orbit

2.3 Radio Frequency

[Pierre]

The selection and design of radio frequency communications systems for CubeSats differ based on the data rate, orbital pattern, and satellite structure. The antennas, satellite antenna release mechanisms, transceivers and ground station components can vary dramatically between different CubeSats to provide good communications links. Analyzing various RF designs from other CubeSat teams provided different options for components and design methods that could be used.

As these thesis and masters papers explore in depth [4], [5], [6], there are several antenna configurations and structures to deploy them.

1. Satellite Antennas
    - Monopole

- Dipole
- Patch
- Turnstile

2. Ground Station Antennas
   - Yagi-Uda
   - Dish

Transceivers and low noise amplifiers can present large costs for CubeSat teams to design and manufacture. Depending on the number of members on the RF division, some teams have either chosen to design a transceiver from scratch, modify an existing one, use various components to build one or purchase a space-rated module from a RF distributor. A master's published paper from the Virginia Polytechnic Institute and State University [7] describes in detail their ground station transceiver layout using purchased components as well as fully designed and constructed ones. An MIT Masters student's research on CubeSat communications [8] presents a thorough list of previously completed satellites and the purchased transceivers used on onboard.

3 Design Criteria, Constraints, and Deliverables

As described by the CSDC guidelines [1], the following criteria were considered when designing and testing various scenarios.

3.1 Orbit

[Pierre]

Teams should design their missions to be able to operate in a LEO, between 400 km and 800 km. At the time of the release of this document a launch has not been procured, and more specific orbit parameters cannot be given; thus, it is advantageous to have a mission and satellite design which can operate in both a sun-synchronous orbit (with an Equator Crossing Time of approximately 10:30 ± 1 hour), and in the orbit of the International Space Station (inclination = 51.66°).

- **[CSDC-0050] Orbit Knowledge** It shall be possible to determine the spacecraft's orbit parameters throughout the mission, to an accuracy as required by the mission [1].

Assuring communications stay within the amateur radio band, frequency ranges were limited to 144-148 MHz in the very high frequencies and 420-450 MHz in the ultra-high frequencies. In turn, these frequency ranges restricted the possible bandwidth.

3.2 Payload

[Jon]

[CSDC-0010] The spacecraft shall fulfill the "Selfie-Sat" mission as outlined in the Concept of Operations (Section 3), to acquire a "Space-Selfie" optical image when commanded by an Amateur Radio Operator, and to downlink it immediately to that Operator.

[CSDC-0020] The spacecraft shall acquire at most one image per Space-Selfie Pass.

[CSDC-0030] The Space-Selfie image shall encompass an area of at least 40km x 40km, and not more than 100km x 100km, on the ground at nadir, assuming a 400km altitude.

[CSDC-0040] The spacecraft may contain additional payloads at the Team's discretion and choice.

3.3 Command and Data

[Adam]

[CSDC-0180] The spacecraft shall have the ability to receive and execute immediate or time-tagged commands from the Mission Control Centre.

[CSDC-0190] The spacecraft shall have a subset of commands for Amateur Radio Operators (ARO’s) to establish a connection with the spacecraft during a Space-Selfie Pass, and to command the spacecraft to immediately acquire an image with its Selfie-Cam. The spacecraft shall not accept any other immediate or time-tagged commands from ARO’s.

[CSDC-0200] All uplinked commands from the Mission Control Centre shall be encrypted, using as a minimum 56-bit-key Data Encryption Standard (DES). Commands uplinked by Amateur Radio Operators during a Space-Selfie pass are not required to be encrypted.

[CSDC-0210] If the spacecraft has not received a valid encrypted command for 48 hours, it may accept un-encrypted commands.

[CSDC-0220] The spacecraft shall have the ability to store time-tagged commands for up to three days prior to executing them.

[CSDC-0230] Any safety-critical or deployment commands shall be implemented as a two-step process.

[CSDC-0240] The spacecraft shall record at least four different points of spacecraft health telemetry, each at a frequency of at least one sample every minute.

[CSDC-0250] The spacecraft shall time-tag all telemetry data.

[CSDC-0260] Under nominal operational conditions, the spacecraft shall have the ability to downlink telemetry data with a latency of less than 12 hours from when it was recorded.

3.4 Radio

[Pierre]

- **[CSDC-0120]** The spacecraft shall comply with the International Telecommunications Union (ITU) applicable radio licensing regulations for the required Radio Frequency operations
- **[CSDC-0130]** The spacecraft shall not generate or transmit any radio signal from the time of integration into the launch dispenser until at least five minutes after on-orbit deployment; however, the spacecraft can be powered ON immediately following deployment.
- **[CSDC-0140]** The spacecraft shall use the AX.25 protocol for communications with the Amateur Radio Operators during a Space-Selfie pass. The uplink communications rate shall be at least 1200 bps; the downlink communications rate shall be 9600 bps. Communications shall be duplex.
- **[CSDC-0150]** Communications with Amateur Radio Operators for Space-Selfie passes shall be possible above an elevation angle of 10 degrees, with a link budget margin of at least 6dB.
- **[CSDC-0160]** Under nominal operational conditions, the spacecraft shall have the ability to downlink the entire Space-Selfie image within 90 seconds.

3.5 Deliverables

[Jon]

The team will deliver a final “Flat-Sat” implementation of the satellite that will simulate the basic satellite functionality and behavior while in orbit. A “Flat-Sat” is the final satellite setup in which all the satellite components are connected and integrated together without the components being housed in the final satellite structure. This allows for testing and simulation of the satellite software as well as the integration with all the satellite subsystems’ hardware and software to test not only individual subsystem designs but also the interfacing and communication between all of them as well. This will verify the design and help to discover system vulnerabilities and weaknesses before the final release as well as prove that the design meets the competition requirements specified above.

4 Design Methodology

The following sections will describe the process through which the final design was achieved, including all of the development and testing procedures.

4.1 Payload

[Jon]

This section will provide an overview of the system architecture and high-level interface of the payload subsystem onboard the CubeSat unit as well as the general operation of the payload subsystem during flight. All calculations for the following design are summarized here, and full calculations can be found in the Appendix A.

The payload subsystem is composed of the Raspberry Pi Compute stick that interfaces both the primary and secondary payloads to the CDH OBC. The compute stick is responsible for accepting commands from the OBC, sending image requests to both the primary and secondary payload sensors, image processing, and transferring any other needed image data or telemetry back to the OBC. Sample software written for the payload module can be found in Appendix H.

Both the primary and secondary payload sensors communicate with the RPi Compute Stick over I2C and SPI protocols. I2C connection is made directly from the compute stick to both the image sensors and is responsible for sensor configuration (setting resolution, exposure, brightness, etc.). SPI is used for general data transfer, such as image data as well as command and telemetry information, because of its much faster speed compared to I2C.

The following diagram presents a high-level overview of the hardware interfaces between the camera sensors, the payload module, and the OBC.

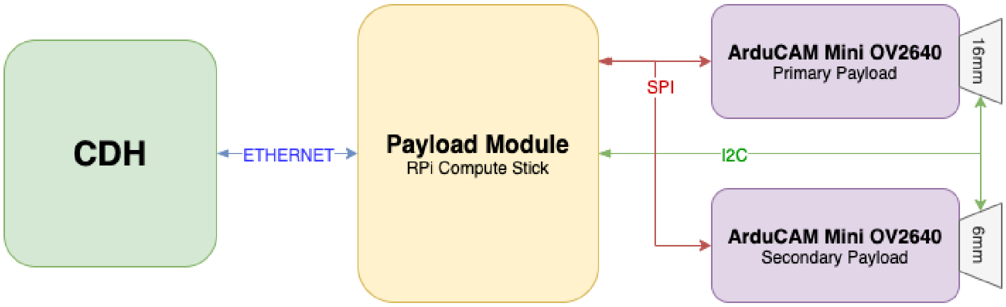


Figure 3 - Payload subsystem architecture.

4.1.1 Payload and Secondary Payload Sensor

After thorough research and investigation, the Arducam Mini Camera Module Shield with OV2640 image sensor was selected to be used for both cameras that service both the primary and secondary mission objectives.

Various sensors were researched but due to previous flight heritage, reliability, and suitable camera resolution and pixel size for primary payload mission requirements, the Arducam Mini camera chip was ultimately selected. The specifications for the Arducam Mini, including an internal block diagram, and a comparison of several optical sensors specifications that were researched can be found in Appendix A.

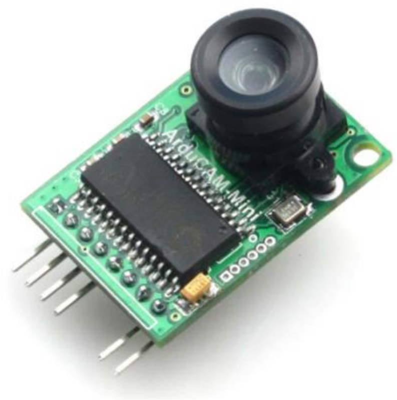


Figure 4 - ArduCam Mini 2MP Camera Sensor [9].

The device incorporates both the Omni Vision OV2640 image sensor with the ArduChip - a camera controller that hides all of the hardware and camera timing while providing a friendly SPI interface for command and data transfer and an I2C interface for camera configuration. The unit has been flight-proven and tested on previous CubeSat missions. An open source library is also provided for the unit, with APIs and documentation, for faster start-up and integration times. The architecture is shown in Appendix A of the ArduChip shield along with the camera. It should be noted that the I2C connection (for configuring the camera) is made directly to the image sensor from the payload board. The Arducam Mini Shield also provides numerous built-in functions or modes depending on current mission status or goals [9]. These are described below:

Single Capture Mode

After issuing a capture command via SPI port, the module will capture a frame and load it into the frame buffer while setting the completion flag bit in the register. The CDH module will poll this register to check if capture is done and load image into main memory.

Multiple Capture Mode

This is an advanced capture mode. By setting the number of frames in the capture register, the ArduCam will capture consequent frames after issuing the capture command.

JPEG Compression

This function is implemented in the image sensor. With proper register settings, the user can get different resolution with the image output.

Low Power Mode

Reduces power consumption by shutting down sensor and memory circuits.

Normal Read and Burst Read Mode

Normal read reads each image data by sending a single SPI command. Burst read allows the user to send one command and receive multiple image data.

Image Sensor Control

Allows the user to set image sensor settings like exposure, white balance, brightness, contrast, and color saturation.

4.1.2 Primary Payload Lens Selection

Following the camera sensor selection, calculations and analysis are completed to determine the correct lens to mount to the camera module.

Competition requirements specify that at an altitude of 400km, the satellite is required to output an image that covers between a 40km x 40km area to a 100km x 100 km area at nadir and is able to obtain an image from +25 degrees cross-track off nadir to -25 degrees cross-track off nadir [1]. However, no spatial resolution requirement is provided. Often in remote sensing applications, a certain degree of spatial resolution is required in order to view certain features or areas on Earth with enough detail.

The team selected a spatial resolution of around 50m/pixel as a reasonable resolution of satellite imagery for this application. The team would attempt to achieve the most coverage area obtained by the sensor while comprising between the detail or spatial resolution of the image itself and the amount of data that would be produced by a single image. 50m/pixel seemed to be an appropriate compromise between image quality and amount of data. Spatial resolution would be one of the main factors that would go into the primary payload camera design decisions.

Optical calculations and tabulated results for spatial resolution and image coverage for the OV2640 image sensor with a variety of lens mounted are shown in Appendix A.

From these results, a 16mm focal length lens is an appropriate compromise between spatial resolution and image coverage that satisfy the primary payload requirements. The specifications summary, of an 16mm lens mounted on the Arducam Mini Shield, is also shown in Appendix A.

4.1.3 Secondary Payload Lens Selection

The objective of the secondary payload is to provide area coverage over countries or other large areas around the world. The goal of the secondary payload sensor is capture continuous images over its lifetime that will be filtered, edited, and stitched together to create an overall coverage map of the nation. Due to limiting capabilities onboard the satellite, the coverage acquisition will be restricted to single countries. The goal is to then setup an online resource or dataset that will provide real-time coverage maps and images to users for big data or algorithm development for remote sensing on a national scale.

To achieve the objective of the secondary payload, the secondary payload sensor is, again, a optical imager and uses the same camera sensor as the primary payload. Using the same camera sensor for both primary and secondary payloads allows for less development time as much of the integration, testing, and development will be the same or similar for both payloads allowing more time to be focused on other aspects of the satellite design.

However, due the difference in objectives between the primary and secondary payloads, the secondary payload will be acquiring considerably more data and images than the primary payload because of its main objective of national coverages. Because of this, and the limitations of the satellite in terms of downlink bandwidth and speed, the secondary payload sensor will be acquiring images that cover much more area than the primary payload sensor to limit the number of images being captured during operation. In other words, the secondary sensor will be using the same number of pixels to cover a much larger area, hence reducing the spatial resolution of the secondary sensor from the spatial resolution of the primary payload “Selfie-Sat” camera. Also, with a much larger coverage area, the sensor is able to obtain full national coverage much faster. A compromise between spatial resolution, image coverage speed, and amount of downlinked data was made and mostly revolves around the lens selection.

Appendix A provides a tabled list of a variety of focal lengths that can be used for the secondary payload sensor, including at nadir image coverage and pixel resolution, as well as image capture rate and data rate assuming the same image resolution of the ArduCam defined above for the primary payload. No information is provided about off-nadir coverage and resolution because the secondary payload will only be acquiring images when pointed at nadir.

It should also be noted that the data rate calculation assumes a JPEG compression of about 10:1 within the ArduCam sensor, although JPEG compression varies considerably depending on the image contents and pixel values.

From the calculations and analysis, the 6 mm lens was selected as the lens for the secondary payload sensor as good compromise between spatial resolution and the amount of data that needs to be downlinked from the satellite. A summary of the ArduCam camera sensor with the mounted 6mm lens specifications is shown in Appendix A.

4.1.4 Payload Operation

This section will describe the general operation of the payload subsystem, including all of the steps it takes in order to complete its mission objective. The payload subsystem is responsible for the acquiring images for both the Selfie-Sat and secondary mission objectives, processing them, and then transferring them to the OBC. In order to do this, the payload subsystem will follow the general operation shown in Figure 5.

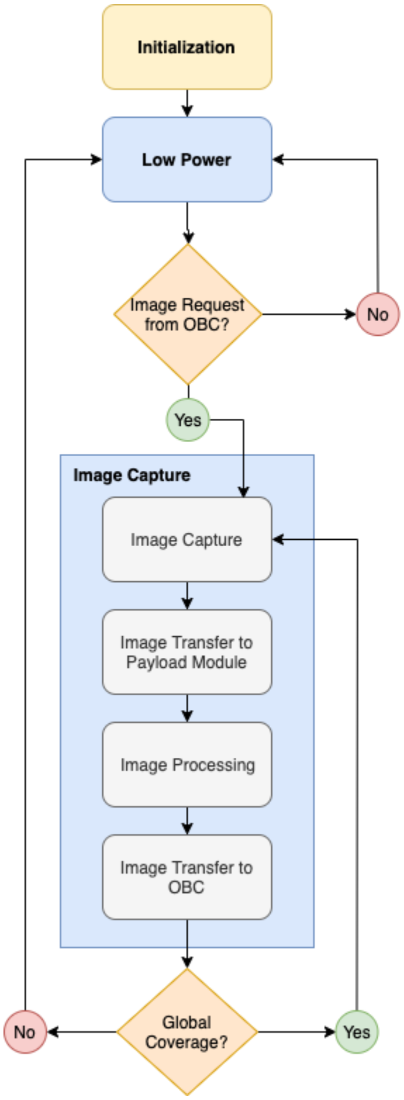


Figure 5 - Payload module operation state diagram.

Initialization

This stage encompasses the time from when the satellite is undergoing deployment activities. In other words, this the deployment stage of the payload subsystem. During this stage, both cameras will be powered on, sent initialization commands to initialize hardware information of the system as well as configure the sensors where necessary (setting image resolution, brightness, image format, etc.). After initialization and deployment are complete, the payload will be placed into low-power mode.

Low-Power Mode

This mode occurs during nominal operation but only when the primary and secondary payload camera sensors are not in operation and are shut down completely.

Coverage Mode

This mode occurs during the time when the secondary payload sensor is turned on and is actively acquiring national coverage images and transferring them to the CDH module for eventual downlink.

Capture Mode

This mode occurs during the time when the primary payload sensor is turned on and is actively acquiring Selfie-Sat images and transferring them to the CDH module for eventual downlink.

4.2 CDH

[Jon]

This section will provide an overview of the system architecture and high-level interface of the CDH OBC subsystem onboard the CubeSat unit as well as the general operation of the satellite software during flight. An in depth overview is provided for the hardware and software design used.

4.2.1 Software Framework – KubOS Linux

The target platform of the CDH OBC software is KubOS Linux. KubOS is an open-source, flight software framework that provides a full-fledged Linux distribution for easier development and start-up times. KubOS comes prepackaged with hardware APIs, core services, developer tools, and heavy documentation and support community to allow most of the design work and time to be focused on mission specific features of the satellite. KubOS also comes ready from day one with many other features already in place including logging, telemetry handling, error handling. Due to the limited development time, being a first-year team, and not an extensive amount of software experience in the field, the team considered that it may be difficult to build our own command and control framework from scratch on top of a standard RTOS system. The KubOS software framework seemed to be the best decision.

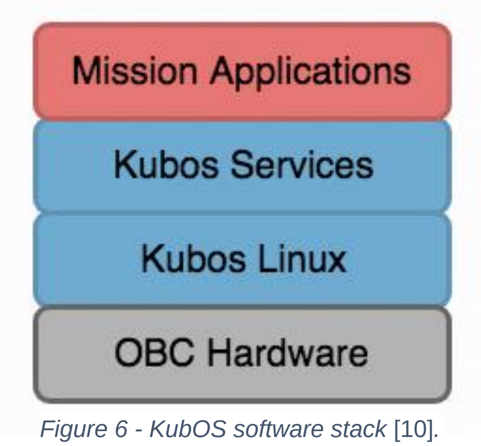


Figure 6 - KubOS software stack [10].

KubOS does have some key disadvantages from a regular RTOS system that should be discussed. One of the main disadvantages of the KubOS Linux platform is its limited number of supported OBCs. However, the team decided to choose the OBC hardware after deciding on the software framework, so this was no issue. Additionally, KubOS Linux, being a Linux distribution, does not meet real-time requirements, as a regular RTOS system would. Since real-time requirements on the OBC are not required to meet mission requirements, again, this was no issue in the design process.

KubOS is designed for taking care of all communication and data handling between every subsystem of the satellite. The KubOS system is built off three main aspects, described in the following sections [10] and shown above in Figure 6.

4.2.1.1 KubOS Linux

KubOS Linux is the custom Linux distribution for satellites and runs directly on the OBC hardware. It provides much of the low-level functionality already in place, including hardware abstraction (I2C, SPI), logging, telemetry and error handling, system monitoring, and many other typical services provided by an OS.

4.2.1.2 KubOS Services

On top of KubOS Linux is what are called KubOS services which are any processes that interact with the satellite. Services do not make decisions but merely provide an interface between satellite subsystems or hardware and the mission applications. Services expose their functions to mission applications with GraphQL requests over HTTP. There are three main types of services:

1. a) *Core Services* – core functions already included with KubOS like monitoring, telemetry, etc.
2. b) *Hardware Services* – expose mission software to hardware devices (ADCS, EPS, etc.)
3. c) *Payload Services* – type of hardware service specific to a mission and its payload.

4.2.1.3 Mission Applications

Mission applications are what describe the behaviour of the CDH subsystem on the satellite. These are modularized applications that can be run continuously or just once, and they control the behaviour of the satellite. They are designed to be lightweight and portable and control certain isolated tasks onboard. Small, simple mission applications reduce the possibility of a global failure due to edge cases in certain applications.

4.2.1.4 Boot Loader

As seen in the comparison above, KubOS Linux is a much more abstract OS meaning it is much more resource intensive and does come with more risk. Because of this, KubOS pairs itself with U-boot, a widely used bootloader that manages the Linux kernel and the core of the system. The bootloader actually provides the ability to update the entire OS during flight if needed. [10]

The following [10] describes a high-level view of the boot sequence of a system running KubOS Linux.

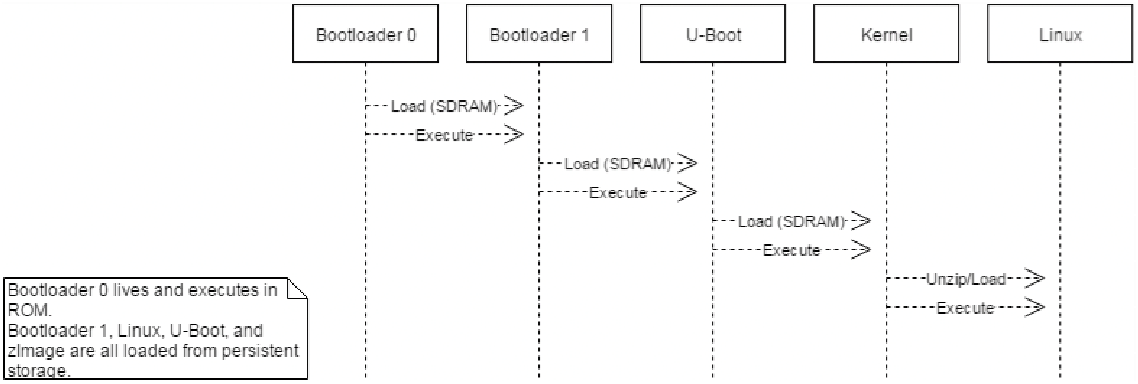

Bootloader 0
Bootloader 1
U-Boot
Kernel
Linux
Load (SDRAM)
Execute
Load (SDRAM)
Execute
Load (SDRAM)
Execute
Unzip/Load
Execute
Bootloader 0 lives and executes in ROM.
Bootloader 1, Linux, U-Boot, and zImage are all loaded from persistent storage.

Figure 7 - KubOS Linux boot sequence [10].

Bootloader 0

This comes with the OBC and is specific to that OBC. Bootloader 0 is responsible for the transfer of the next bootloader into SDRAM to be executed. It is stored in system ROM.

Bootloader 1

This is the second boot loader that loads U-Boot from main storage into SDRAM for execution.

U-Boot

The main responsibility of U-Boot is to load the kernel form the SD card into SDRAM and provides a basic CLI for modifying the kernel before it’s loaded. It can also be used as an OS upgrade and recovery system before the kernel is loaded. U-Boot also has environment variables that can be used to store information of high impact to the system through reboots.

4.2.2 Software Architecture

The high-level software architecture of the CDH subsystem module is shown in the below figure. It should be noted that everything coloured blue in the below diagram is provided with KubOS and anything coloured in red is mission software that will be written by team.

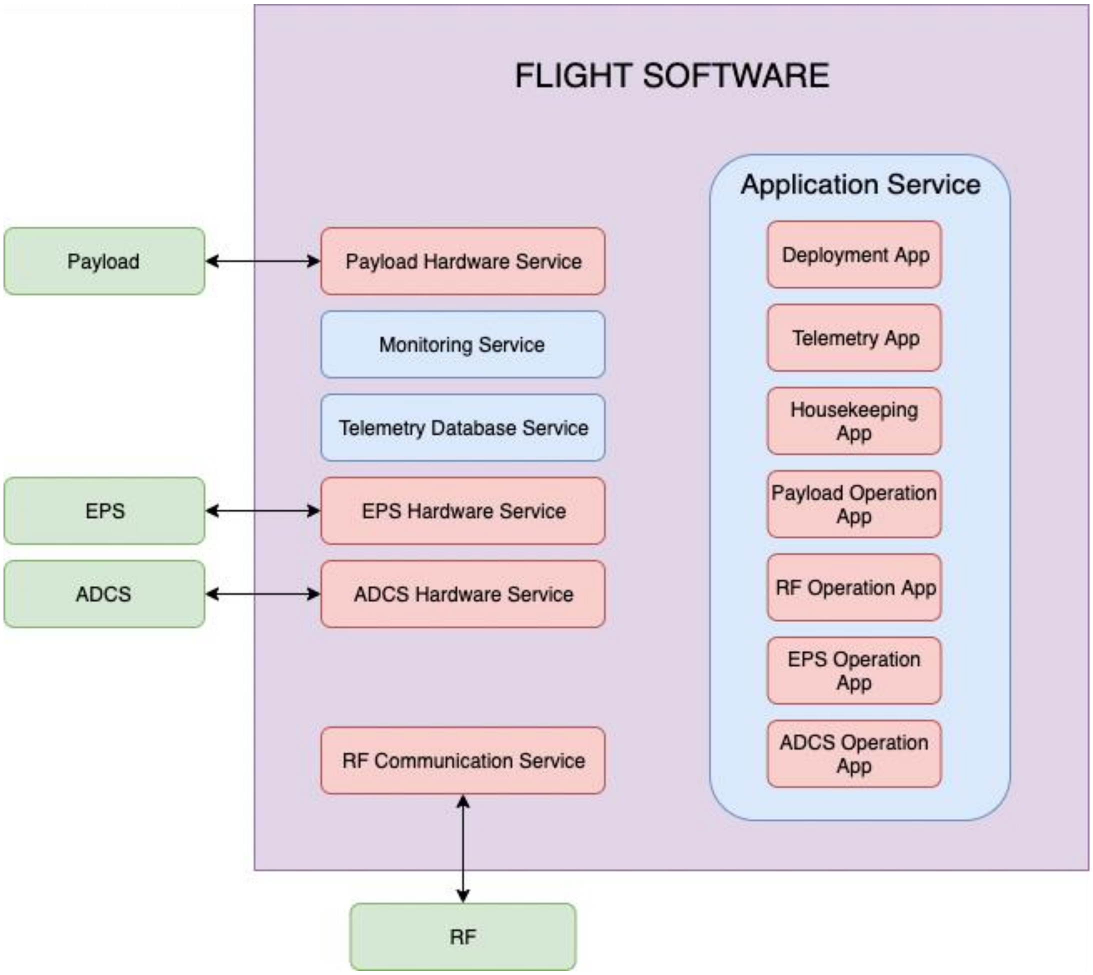


Figure 8 - Flight software architecture.

4.2.2.1 Services

Application Service

The application service is responsible for monitoring and managing all mission applications for the system. Multiple versions can be tracked here allowing for easy upgrades and rollbacks. This is a core service provided by KubOS.

Monitoring Service

This is a hardware service that deals with the OBC itself. The monitoring service provides functionality to check current running processes and memory usage. This is a core service provided by KubOS.

Telemetry Database Service

The telemetry database service uses a SQLite database to store telemetry data generated by hardware and payload services until it is requested for transmission to the ground station. This is a core service provided by KubOS.

RF Communication Service

The communication service processes all the communication between the ground or radio operators that is received and sent through the RF hardware that simply performs the communication between the satellite and grounds stations.

Payload Service

This is a hardware service specific to the payload subsystem. It is responsible for the communication and interface between the payload module, including both the primary and secondary payloads, and the OBC software. This module is be able to facilitate data and commands between the primary payload and the rest of the software architecture.

EPS Service

Hardware service that facilitates the interface between the OBC and the EPS module. It is responsible for the bi-directional communication of commands, telemetry, and other data between the EPS module and the OBC. KubOS comes providing an API for the Clyde Space EPS module that will be used for the EPS subsystem.

ADCS Service

This is a hardware service that facilitates the interface between the OBC and the ADCS subsystem. It is responsible for the bidirectional communication of commands, telemetry, and other data between the ADCS module and the OBC.

4.2.2.2 Applications

Deployment Application

This mission application is responsible for handling the deployment sequence and follows the following procedure:

1. Keep track of hold time provided by the launch provider
2. Deploy deployables (burn wire)
3. Powering on and configuration of subsystems.
4. Detumbling and stabilization of CubeSat.

Figure 9 is a state diagram providing a high-level overview of the deployment sequence in regards to the OBC software.

“Deployed” and “deploy_start” variables are stored as U-boot environment variables, the most reliable and persistent storage on the OBC. These track if the satellite has been deployed and how long since the deployment sequence has begun, respectively. This application uses the RTC.

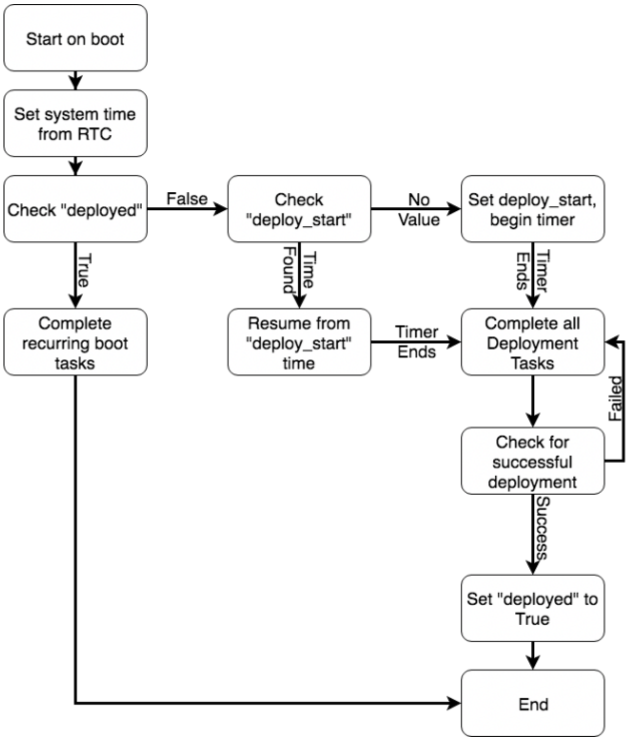


Figure 9 - Deployment application procedure [10].

Telemetry Application

The telemetry application is responsible for collecting and storing telemetry in the telemetry database using the telemetry database service. This application will poll all hardware services

(services that interface to satellite subsystem hardware) in a 1 minute interval for all required telemetry items.

Housekeeping Application

The housekeeping application is responsible for monitoring all satellite hardware, including the OBC itself. The application uses the monitoring service and other hardware services to keep track of health telemetry, monitoring status and other critical items such as memory and power. The application will operate on a 1 hour interval. Other functions of this application include but are not limited to:

- Cleaning the telemetry database
- Checking file system and memory usage
- Issuing test queries to services
- Checking critical telemetry items
- Shutting off non-essential hardware when battery reaches critically low status
- Cancelling operations and going into power generation state
- Monitoring battery temperature

Payload Operation Application

This application is responsible for the general operation of the payload subsystem. This involves sending commands and requests for image acquisition or camera configuration as well as receiving image data and telemetry from the payload module.

ADCS Operation Application

This application is responsible for the general operation of the ADCS subsystem. This involves sending commands and requests for ADCS behaviour as well as receiving vital telemetry from the ADCS module. For example, this application would be responsible for commanding ADCS to place the satellite in the appropriate attitude.

EPS Operation Application

This application is responsible for the general operation of the EPS subsystem. This involves sending commands and requests for EPS behaviour as well as receiving vital telemetry from the EPS module. For example, this application would be responsible for collecting telemetry, such voltage and current levels, from the EPS module.

RF Operation Application

This application is responsible for the general operation of the RF subsystem. This involves sending commands and requests for RF behaviour as well as receiving vital telemetry from the RF module. This application is also responsible for regularly downlinking critical telemetry data to ground station using the RF communication service.

4.2.3 Software Operation

Figure 10 is a high-level description of the boot or deployment sequence of the satellite from the moment it is ejected from the launch vehicle.

4.2.3.1 Deployment Task

This task involves the procedures from the moment the satellite is ejected from the P-POD. A 30 minute wait is required after P-POD ejection according to CubeSat standards. After the deployment task is completed, the initialization task begins.

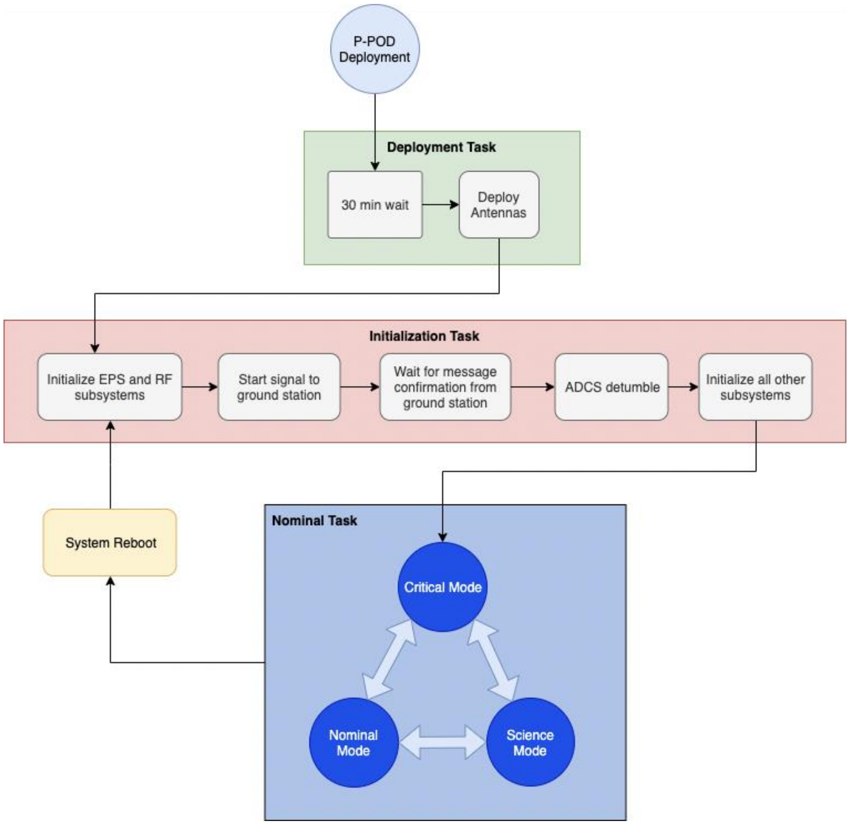


Figure 10 - CDH software operation state diagram.

4.2.3.2 Initialization Task

Following deployment, the satellite will begin to initialize satellite subsystems. Most important, however, is the EPS subsystem to begin charging and the RF subsystem to initialize communication with ground station. While EPS begins charging the battery, RF will send the ground station an ack and wait for a confirmation message. Following confirmation of communication with the ground, the satellite will initialize ADCS and detumble and then initialize payload. System health checks will also be completed, and then the satellite will enter nominal operation.

4.2.3.3 Nominal Task

This is where the satellite will spend nearly all of its time. This is normal operation and nominal modes will be discussed in the next sections. At any point during the satellite lifetime, if a satellite reboot is required, for example due to some system or unrecoverable error, the satellite will rerun the initialization task.

For proper operation, the satellite has predefined modes and transition between the modes occur autonomously depending on system information, health, status, and other data. Each mode or task has predefined and dedicated tasks based on what the satellite encounters.

Critical Mode

This mode occurs when the battery has very low capacity. All non-critical subsystems and functionality are powered down or stopped until the battery is charged to a certain threshold. The transmit rate to the ground station is also reduced. Only the most critical and essential components are operational to conserve power.

Nominal Mode

The satellite is in normal mode of operation, monitoring system health and receiving and sending commands from the ground station. The payloads are off.

Science Mode

This is identical to nominal except the only difference here is that the payloads are powered on and “Selfie-Sat” images or area coverage images are being collected.

4.2.3.4 Logging

Since the CDH is utilizing KubOS, a full Linux distribution, most of the logging and file system functionality is provided by the OS itself.

The CDH module will utilize rsyslog to route log messages to the correct log file and rotate them when they become too large. Every service and application will route its log messages to its own specific log file, including a time-tag as well as description and level of the logging item.

4.2.3.5 Log Levels

There are four levels of logging that separate and identify messages based on their level of criticality:

1. INFO – General housekeeping items that highlight the process of applications and services.
2. WARN – Describe potentially harmful situations.
3. ERROR – Describe serious events but still allow the application and service to continue running. Further system health checks and specific error handling procedures will be completed to resolve the error. If unable to resolve the error, the message will be escalated to a fatal message.
4. FATAL – Describe very serious events that have led to applications or services aborting. Fatal messages result in full system reboot.

4.2.3.6 Log Rotation

Log files have a maximum size of 10KB. Once the log file reaches this maximum size, the file is renamed as an archived file and a new log file is started. The archived files are renamed using the same name they had previously but include a time stamp of when they were generated. A total of ten log files are stored for all applications and services at all times, 9 archived files and 1 current log file. Older archived log files are deleted.

4.2.4 ADCS Controller

[Adam]

The ADCS subsystem controls the trajectory of the satellite, orientates the solar panels to face towards the sun, and provides positional information to the OBC. A separate MCU (microcontroller unit) is needed to control all the sensors and devices (gyroscope, sun sensors, magnetorquers, and reaction wheels). The CDH team is responsible for the communications between all the devices, sensors, and both MCU (OBC and ADCS controllers); the ADCS algorithms are not this team's responsibility.

4.2.4.1 Hardware Abstraction

The ADCS controller doesn't have flight software like the OBC (KubOS) so the CDH team had to "bring-up" the ADCS board. Board bring-up involves writing firmware or code specific to the hardware of both the processor and the board's peripherals: pins, communication buses, clocks, etc. All the hardware specific code needed to be abstracted from the application level; in other words, application designers should never be able to reference hardware specific code. The reference manual [11] for the ADCS MCU contains all the memory addresses for the control registers, control bit offsets in the registers, peripheral buses, and clock tree circuit (*Appendix F. 1 and F.2)*. All memory addresses, offsets, and control flags needed for our application were defined in device driver header files only to be referenced by driver source code *(Appendix G)*. Abstracting all the device specific addresses and flags allow future WinSAT teams the flexibility to use any MCU from the STM family of boards by simply updating the device specific addresses in the MCU driver header file to the MCU of their choice.

4.2.4.2 I2C Driver

I2C (Inter-Integrated Circuit) or two-wire protocol was chosen for the main communication protocol between physical devices in the CubeSat. Subsystem design is very easy using this protocol as only two wires are needed - SDA (serial data line) and SCL (serial clock line). The OBC is running KubOS Linux and the ADCS controller is running Mbed OS (Arm's embedded OS [12]).

The following figure shows the configuration of the ADCS subsystem:

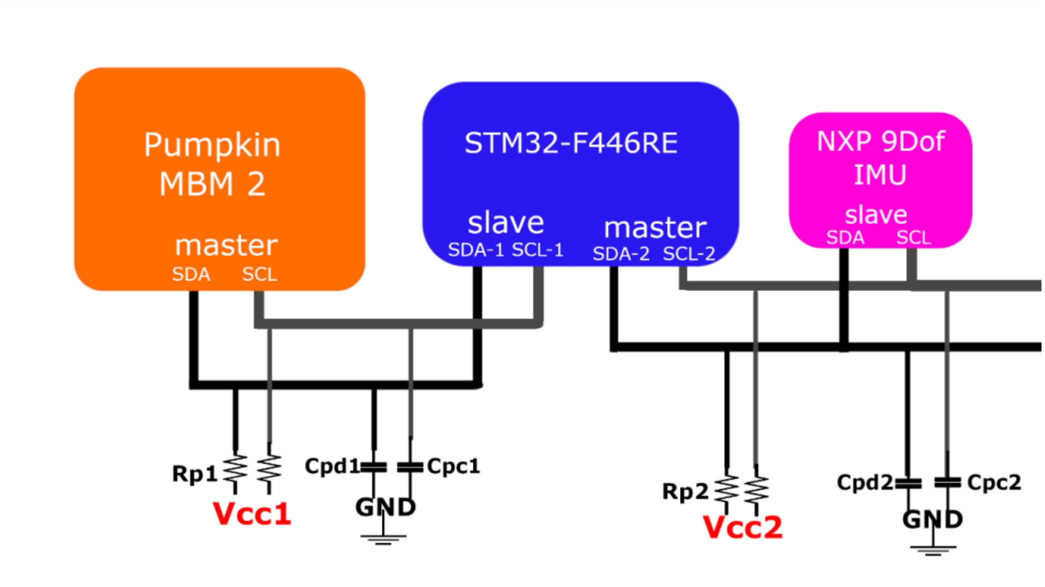


Figure 11 - CDH I2C Diagram for ADCS controller.

The STM32-F446RE is the ADCS MCU, the Pumpkin MBM 2 contains the OBC, and the NXP 9Dof is an IMU sensor on the I2C bus with the ADCS controller as master.

The ADCS MCU has SDA-1 and SCL-1 configured as slave with the OBC as master. It is over this connection that the OBC will send commands to the ADCS system and can retrieve telemetry data from the subsystem. The OBC is communicating on a driver level giving it the ability to trigger reset interrupts if the system needs to be refreshed or rebooted in low power mode.

The SDA-2 and SCL-2 are configured as master on the ADCS controller with the NXP gyroscope sensor connected as slave. Notice how this bus extends past the sensor to indicate that this design is scalable where future WinSAT teams can add more I2C slave devices on this bus with the ADCS MCU as I2C master. With I2C clock stretching, the ADCS master servicing all the slave devices are scheduled on a hardware level. This simplicity of design saved a lot of development time as OS processes and thread didn’t need to be written to schedule the communication of all these devices. Clock stretching is done by simply holding the SCL low signaling that the device is busy and to wait to begin serial transmission of data.

The resistors Rp1 and Rp2 (two of each respectively) are pull-up resistors to “pull-up” voltage of SCL and SDA to the logic voltage of +3.3V (*Appendix F. 3)*. Pull-up resistors are needed due to parasitic capacitance of the PCB and bus material causing a rise time of the voltage that doesn’t hit the logic level in some cases (pull up resistors prevent this). The parasitic capacitance is shown by Cpd and Cpc for the capacitance of the data and clock bus respectively. (*Appendix F. 3)*

4.2.4.3 GPIO Driver

The ADCS MCU has multiple I2C peripherals on the APB1 (advanced peripheral bus). The scalability of this project is important for future WinSAT teams so instead of hard coding pins to access the I2C peripherals on APB1; the GPIO driver was written to configure the pins as I2C pins while providing macros to configure the GPIO pins for other communication protocols. Refer to *Appendix F-3* for the GPIO configuration used for I2C pins and *Appendix G* for driver header files.

4.2.5 Hardware

After the software framework, KubOS Linux, was selected for the onboard satellite computer, there were limited options in terms of board selection and board compatibility with KubOS Linux. KubOS Linux is a relatively new Linux distribution with a relatively limited set of compatible boards, but the benefits of using KubOS Linux, including reduced development time, higher-level development, and the ease of use of Linux, all outweighed the negatives of using the framework, one of which was the limited board selection.

The team selected the Pumpkin MBM2 as the onboard computer. The major reason for this decision was its compatibility with the BBB development board. The MBM2 is shown in the below figure.

Figure 12 - Pumpkin MBM2 [13].

The MBM2 houses the BBB on top of the motherboard and provides further interfacing functionality and other critical services that the BBB does not on its own contain, such as an RTC. This would allow the team to do all of the main software development and testing on BBB and then be only required to do simple integration of the MBM2 when the final flight model is built.

More information about the MBM2 can be found in the datasheet [14].

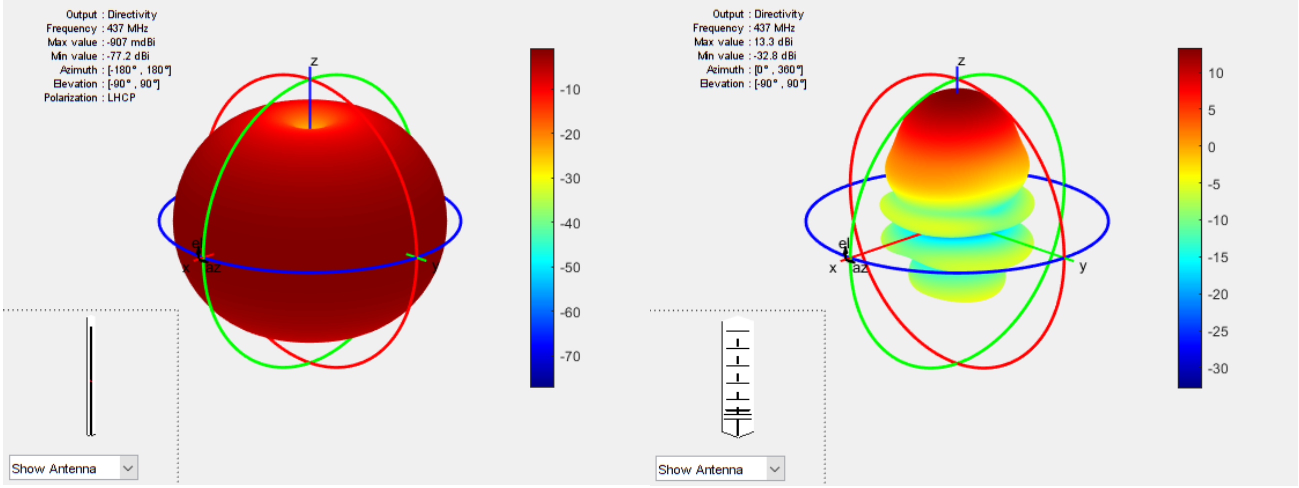
4.3 Radio

[Pierre]

Simulating both satellite orbital paths and generating graphical data to verify the capabilities of our proposed RF system required the following process. The creation of a link budget was critical to consider and include for antenna simulations. Upon having calculated and included all constraints, a final uplink and downlink margin was generated for the ISS and Sun-Synchronous orbits. These constraints and other parameters would represent the worst-case scenario (threshold) whereby the transmission and reception of our frequencies would be at its least favorable.

Consequently, simulating both satellite orbital paths, generating graphical data, and verifying the capabilities of our proposed RF system yielded the necessary proof to finalize our design. The first step was to create the antennas and their respective radiation patterns were created for our chosen Yagi-Uda ground station antennas and for our dipole satellite antennas.

Using the antenna designer application tool within MATLAB, the script could be written to define antenna properties such as frequency, load impedance and directional range. The final generated antenna would be created at the appropriate length and posses the defined characteristics. Refer to Appendix J for all radiation pattern MATLAB code.



Output : Directivity
Frequency : 437 MHz
Max value : -907 mdBi
Min value : -77.2 dBi
Azimuth : [-180° , 180°]
Elevation : [-90° , 90°]
Polarization : LHCP

Output : Directivity
Frequency : 437 MHz
Max value : 13.3 dBi
Min value : -32.8 dBi
Azimuth : [0° , 360°]
Elevation : [-90° , 90°]

Figure 13 - Satellite Dipole Antenna Radiation Pattern (left) and Ground Station Yagi-Uda Radiation Pattern (right).

Once these antenna radiation patterns were generated, a second script file was written to properly convert radiation patterns into phi, theta, gain vectors that could be imported into the Systems Tool Kit software for communication link simulations. Now that the radiation patterns were input into STK, testing could begin regarding the dipole antenna orientation on the satellite. To evaluate the various angles at which the antennas must communicate during orbit, three basic orientations were simulated by sending and receiving data. Attempting to find the optimal orientation, the perpendicular receiving and transmitting dipole antennas were oriented horizontally, vertically and at a 45-degree angle to compare their results. In addition, all constraints were added such as the link margin threshold from the link budget, uplink and downlink communication rates, bandwidth, minimum elevation angle, system noise temperature, LNA gain and loss, transmission power and simulation time. To ensure consistency, all three scenarios were tested for both orbital paths with the ground station tracking and pointing towards the satellite.

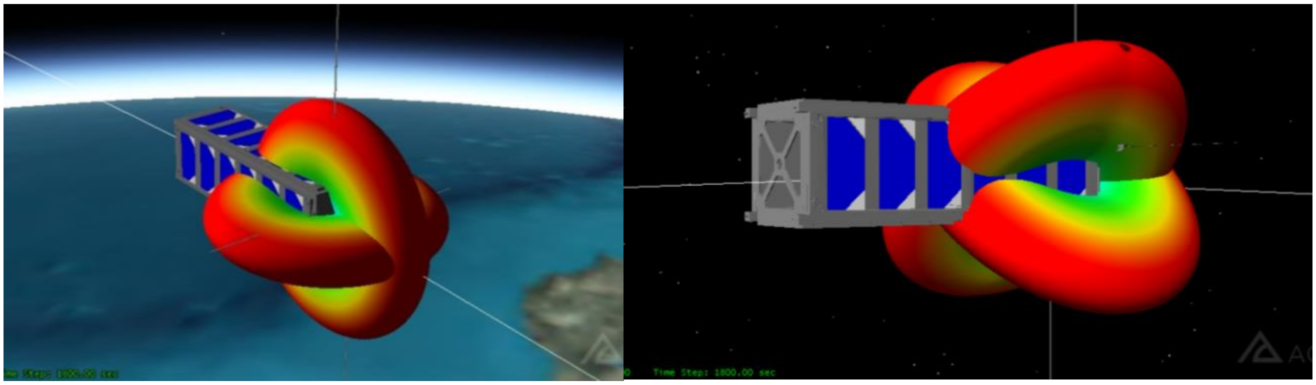


Figure 14 - Horizontal Transmitting Radiation Pattern (Left) 45 deg. Transmitting

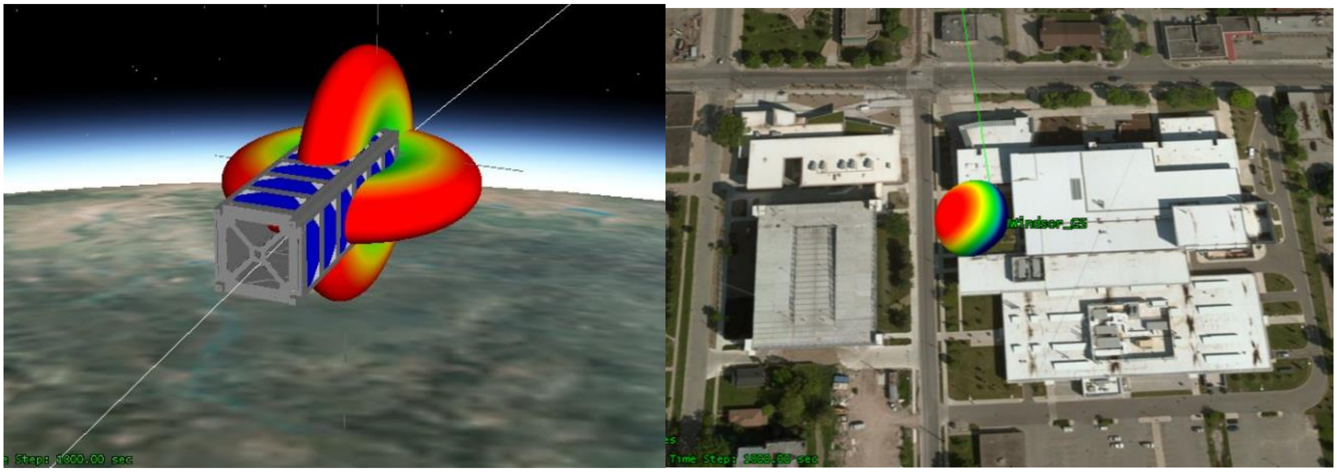


Figure 15 - Vertical transmitting Radiation Pattern (left) and Ground Station Radiation Pattern (right).

As can be seen from the ground station figure, STK uses Bing maps to properly illustrate ground scenery. In hopes of one day building a ground station on University of Windsor campus, simulations were conducted with the ground station located on the green roof of the Center for Engineering Innovation building.

5 Physical Implementation/Simulation Development

5.1 Payload

[Jon]

The WinSAT CubeSat payload was simulated using a model generated in AGI STK, a satellite software simulation environment. The camera and lens parameters for the both the primary and secondary payloads were verified and tested in simulation to ensure that mission requirements were met and to observe access and coverage statistics of the payloads respectively. This would allow for allow for verifying how often the payload imagers could access certain locations and how long coverage of certain areas and countries could be retrieved for viewing.

5.1.1 Primary Payload Access Statistics

The following is a description of the access that the satellite primary payload sensor would have to singular cities or places on Earth, simulated with AGI STK software. This is a calculation of the amount of time that the satellite's sensor, or in this case a camera, can observe the place or city. Additionally, for these targeted access statistics, the satellite would be enabled to move +25 and -25 degrees off-nadir cross-track according to the CSDC requirements [1].

The following simulation results and access statistics were run assuming an sun-synchronous orbit of equator crossing time of 10:30:00 UTC, a 400 km altitude, and the sensor optics calculated and described in the previous sections.

Table 1 - Access statistics of primary payload in sun synchronous orbit at 400 km altitude over 1 year.

Access Statistics	Iqaluit	Windsor	Manaus
Min Duration (s)	8.982	8.825	8.542
Max Duration (s)	9.92	9.672	9.412
Mean Duration (s)	9.441	9.249	8.97
Total Duration (s)	623.121	342.2	260.143
Number of Accesses	66	37	29
Mean Accesses per Month	5.5	3.08	2.42

Sample images from the AGI STK simulation software that demonstrate the satellite accessing these locations can be found in Appendix C.

5.1.2 Secondary Payload – US Case Study

Area coverage of the secondary payload on the satellite was simulated with AGI STK software. To gain an estimate on feasibility, analysis was completed for national coverage to observe coverage statistics and behaviour. The results and comments on this analysis and its results are described in this section. The eventual goal is to downlink the coverage images, perform filtering and stitching, and then provide an online resource or API for users to access these images for remote sensing analysis or algorithm development and other big data projects where coverage images are needed.

Calculations were done to verify the amount of coverage in an area and the amount of data and what rate it would be produced by the satellite while attempting coverage of the US. The results are shown in area coverage analysis results in Appendix C.

The amount of data being produced by the secondary payload imager to complete the fastest coverage of the US is considerable. Being a secondary mission objective, coverage images are of a much lower priority than the “Space-Selfie” images and other critical command, data, and telemetry. As a result, the team may need to reduce the amount of coverage data being downlinked to the ground station, which will result in a longer time to provide full coverage of countries and other large areas, like the US.

ISS Orbit

Images can be found in the appendix that demonstrate the coverage analysis of the US for the secondary payload sensor, assuming the specifications specified in the above design description. These calculations assume that the satellite is at 400km altitude and is in ISS orbit. An example image of coverage over 3 days is shown below.

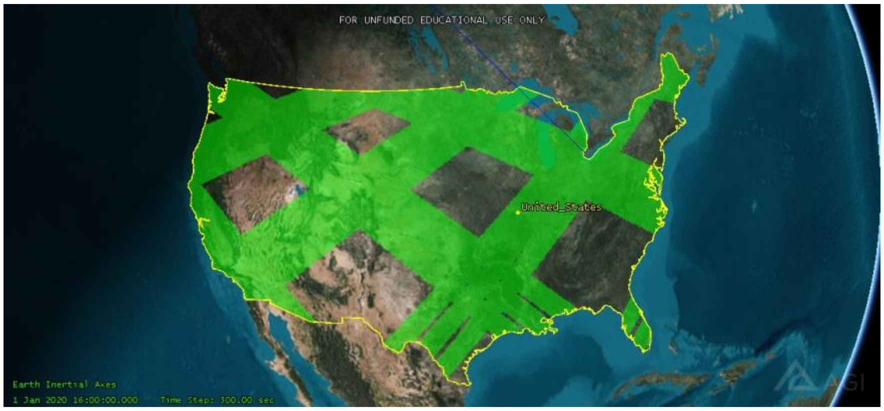

FOR UNFUNDED EDUCATIONAL USE ONLY
United_States
Earth Inertial Axes
1 Jan 2020 16:00:00.000 Time Step: 300.00 sec
AGI

Figure 16 - Secondary payload US coverage over 3 days assuming ISS orbit.

More of these coverage images for different periods of time can be found in Appendix C. Additionally, the graph below provides the percent coverage of the US provided by the satellite over time.

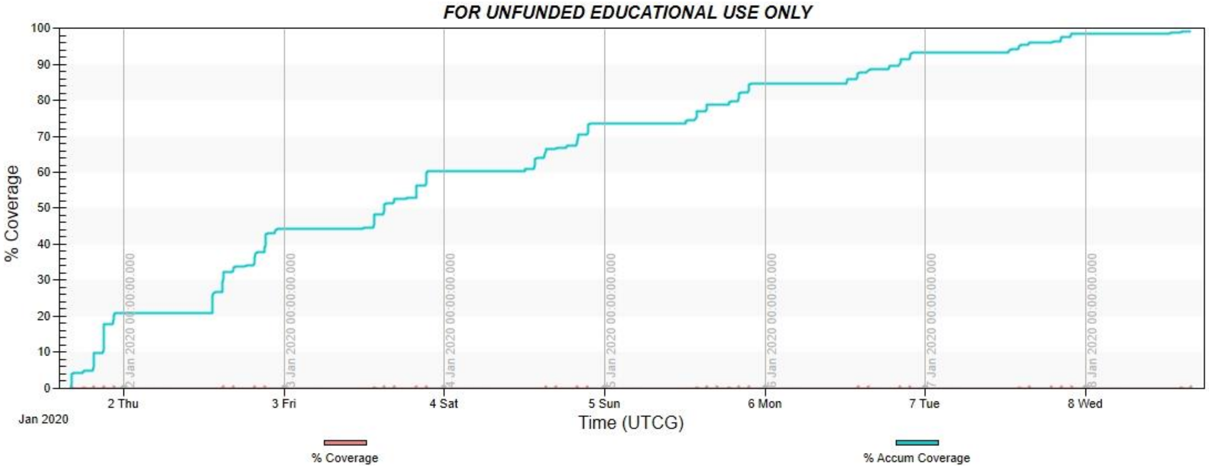


Figure 17 - Secondary payload US percent coverage over time in ISS orbit.

Sun-Synchronous Orbit

Images can be found in the appendix that demonstrate the coverage analysis of the United States for the secondary payload sensor, assuming the specifications specified in the above design description. These calculations assume that the satellite is at 400km altitude and is in ISS orbit.

The graph summarizes these results by providing the percent coverage of the US provided by the satellite over time.

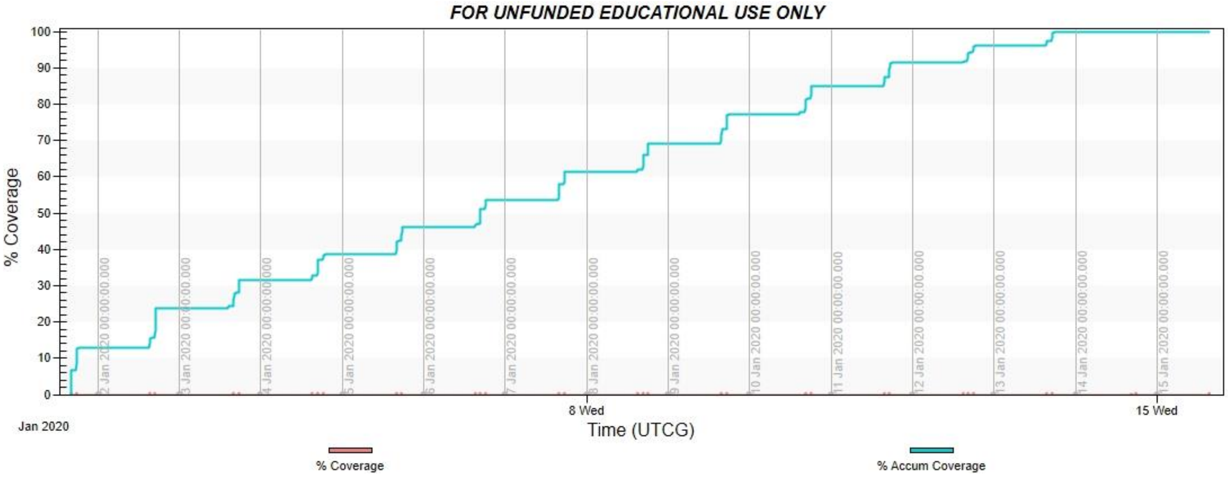


Figure 18 - Secondary payload US percent coverage over time in Sun-Synchronous orbit.

5.2 CDH

[Jon]

To test and validate the software design of the onboard computer or command and data module, a “FlatSat” setup was built and is shown in the below figure.

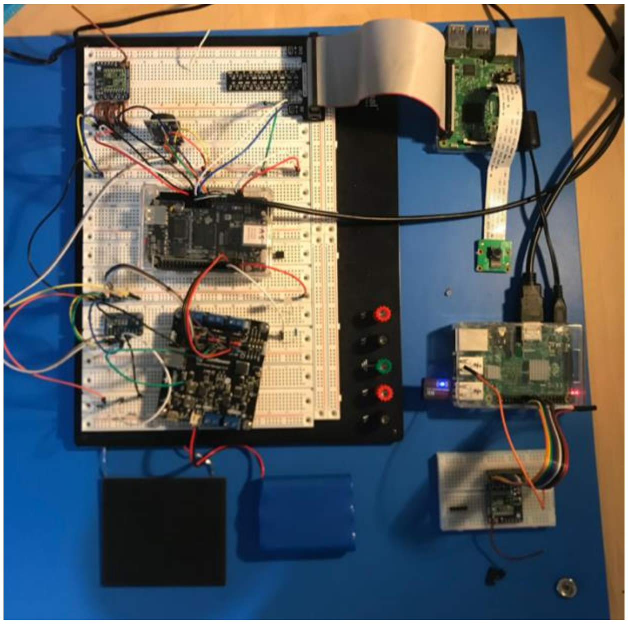


Figure 19 - "Flat-Sat" software integration testing.

This setup includes COTS replacements for all the major components that would be present in the flight model of the satellite. This allowed for demonstration and testing of the software design on the main onboard computer and its functionality and interfacing with other subsystems.

5.2.1 ADCS

[Adam]

Figure 22 shows the testing of the STM32 board bring-up using STM32CubeIDE [15] and testing I2C1 and I2C2 peripherals as slave and master respectively. One Arduino acting as slave (right) and the other as master (left) connected to different I2C peripheral buses in the STM32 board one configured as master (for right Arduino) and the other I2C bus as slave (to the left Arduino).

Figure 23 shows the STM32 board bring-up using STM32duino [16] imported in the Arduino IDE. The NXP 9DoF gyroscope communicated as an I2C slave to visualise the orientation of a model of the CubeSat provided by the structural team. Refer to *Appendix L* for the orientation software.

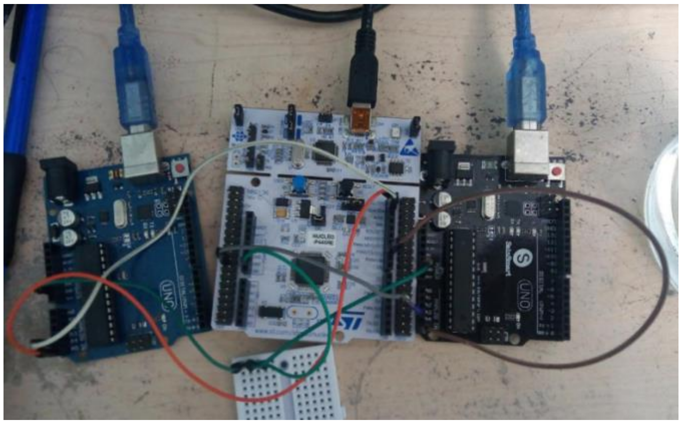


Figure 20 - ADCS controller comms test

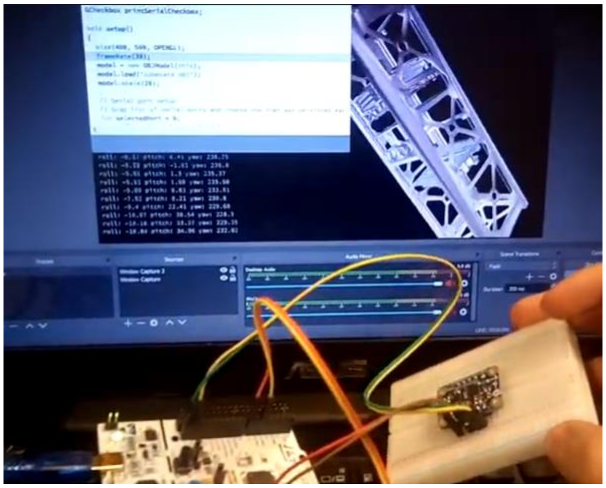


Figure 21 - ACDS controller IMU read test

5.3 Radio

[Pierre]

Considering the simulation results, the antenna deployment mechanism could be designed based on the chosen orientation. The radio frequency division worked alongside the structural team to complete a basic antenna deployment system whereby the antennas would be coiled into the structure. The C brackets provide the appropriate shape for the antennas to coil but present certain challenges as they consume a large amount of the PCB surface area. Upon considering the placement of the two baluns for each dipole antenna, it was important that they be placed on the PCB in such a fashion that the coaxial cables connecting them to the antennas and the transceiver would have easy accessibility. The view of the modeled design seen below does not include the top plate that encloses this structure. Taking this into account, the routes of the coaxial cables are limited by the space between the PCB (green plate) and the top plate of the antenna release mechanism. In addition, the coaxial cables would have to go around the C brackets which finally led to the decision that the baluns should be located directly in between the C brackets thus only requiring one of the coaxial cables to pass in between the C bracket. Finally, the center hole would provide the necessary access for the coaxial cables to connect the balun to the transceiver.

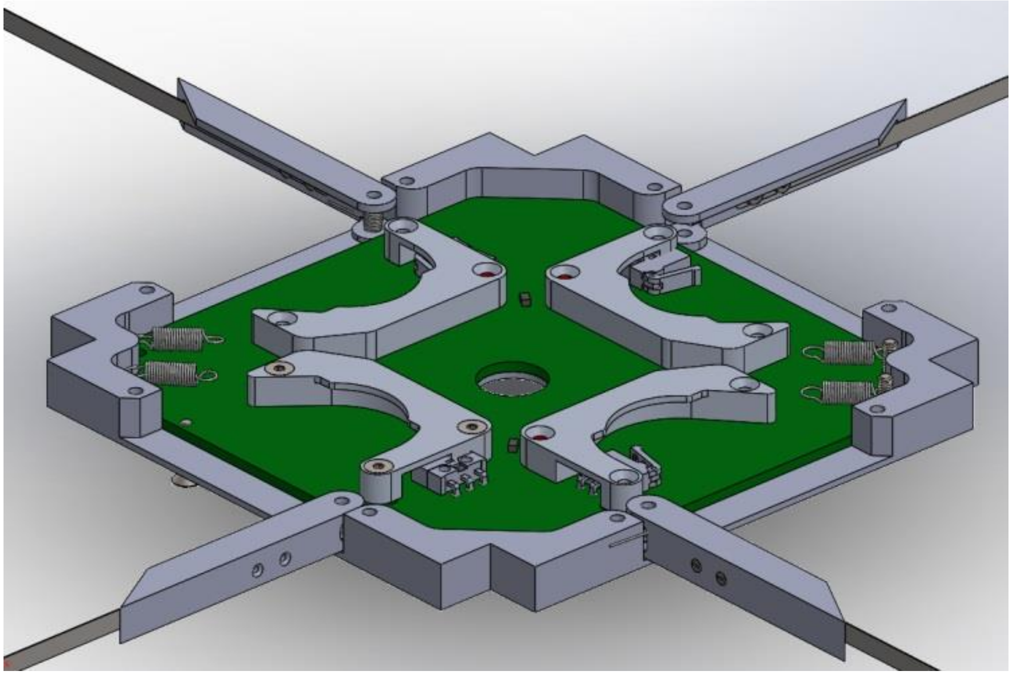


Figure 22 - Antenna Deployment System

Within very close proximity to the balun, U. FL surface mount male connector jacks would be soldered onto the PCB with traces connecting them to the balun. Coaxial cables with female U.FL connectors could then connect the balun circuit to the dipole antennas and to the transceiver. The following figure represents the connectivity of all these components.

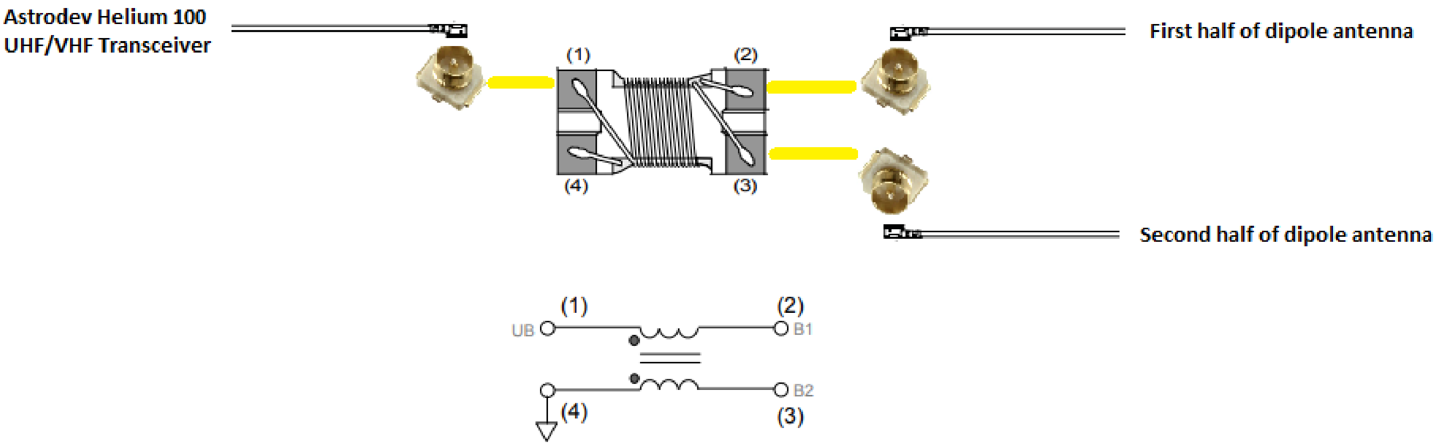


Figure 23 - Balun Connections

Considering this architecture, between any one of these connections or transmissions there is a possibility for the signal to lose its integrity as well as impedance mismatch. Refer to Appendix K for proper clarifications on this terminology. As the connectors, coaxial cables and baluns are components that have been specifically designed for radio frequencies, they could be trusted with relative confidence. However, the PCB substrate that connect the U.FL connectors to the balun should use the appropriate materials, thickness and manufacturing to ensure the proper signal transmission for both the receiving and transmitting frequencies. Assessing these possible trace characteristics was conducted through Advanced Design System and EMPro from Keysight technologies. Using an existing balun design in the ADS environment, different substrates could be tested at both 144MHz and 437 MHz frequencies. Here is a basic three-layer substrate that was tested with two dielectric layers, one conductor, a top cover, and a bottom cover.

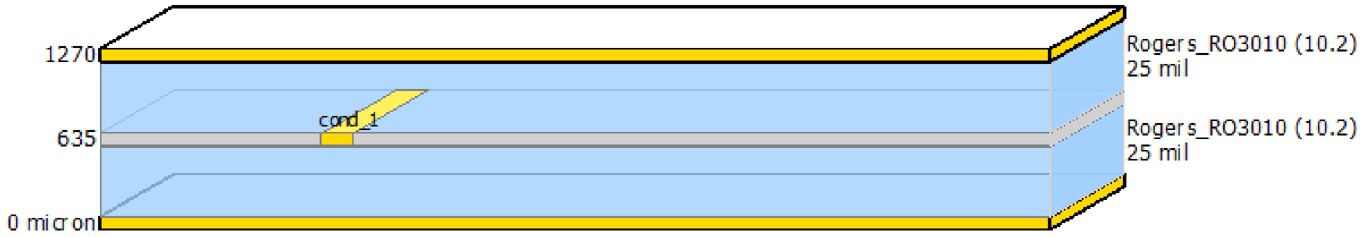


Figure 24 - Substrate Example

Upon specifying the substrate layers in ADS, the balun circuit could be imported into EMPro and simulated for both frequencies. Results would define radiation efficiency, reflection coefficient and voltage standing wave ratio amongst other things. Trace curvature was implemented to fully evaluate the capabilities of each substrate and expose weaknesses.

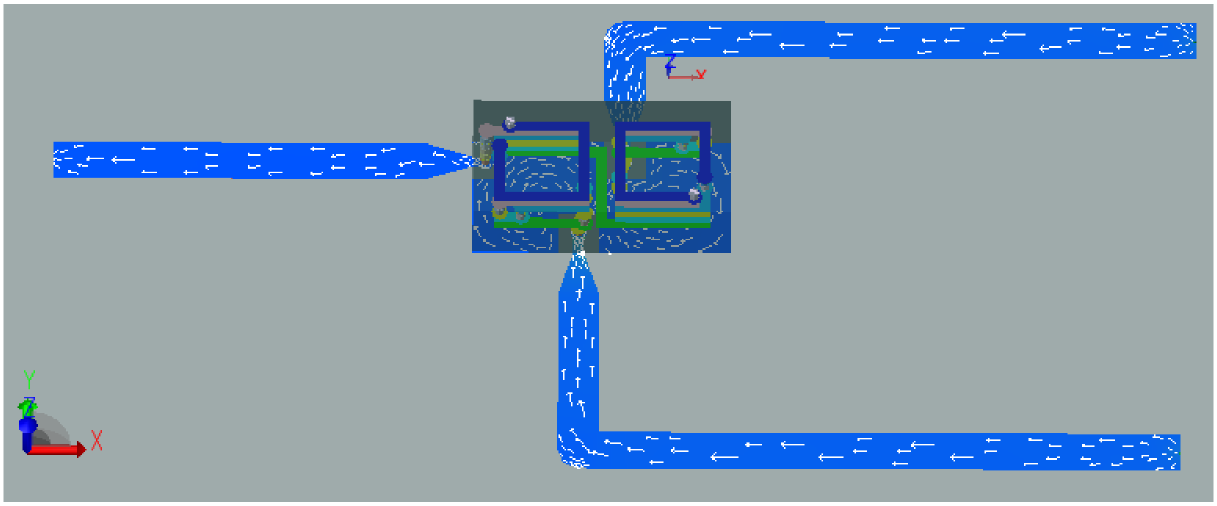


Figure 25 - EMPro Balun Design

Future work for this project will include further testing using this model for various substrate materials and thicknesses. Once sufficient data has been collected, the PCB design will be finalized and sent to be manufactured.

6 Experimental Methods/Model Validation

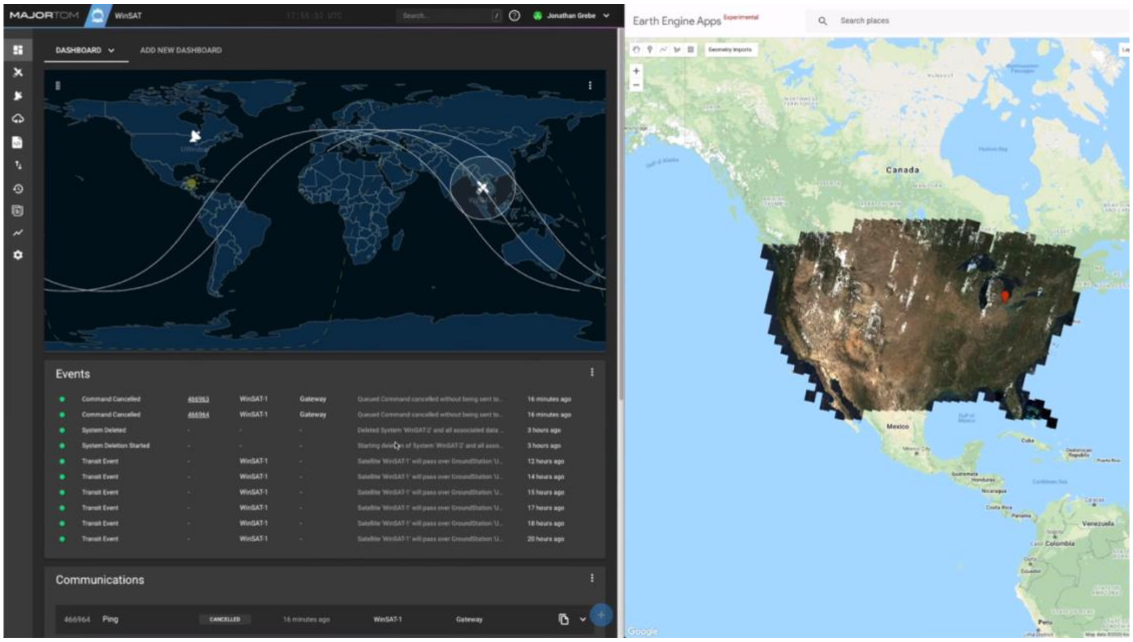
6.1 Software Integration Testing

With the implementation of the “Flat-Sat”, the team performed software unit testing on individuals subsystems of the satellite as well as integration testing with all of the satellite components interfaced together. Mock software models were created to simulate other subsystems of the

6.2 Ground Station Operator and Satellite Image Viewing

To test the satellite image captures and transfers, the team used the KubOS ground station software to communicate with the satellite as if it was in orbit. This ground station software was used to command the “Flat-Sat” model to take coverage images. For testing, the team used sample images from other satellites, placed them on the payload module, and then downlinked those images to ground station through the radio. Then, the coverage images were modified and stitched together and then uploaded to web application that provides access to viewing these images through a public link: https://winsat.ca/payload

The image below is a snapshot of what the ground station operator may see while the satellite is in orbit. Both the coverage images from the web application and the ground station operator form KubOS are shown.



Events

Command Cancelled	466963	WinSAT-1	Gateway	Queued Command cancelled without being sent to...	16 minutes ago
Command Cancelled	466964	WinSAT-1	Gateway	Queued Command cancelled without being sent to...	16 minutes ago
System Deleted	-	-	-	Deleted System 'WinSAT-2' and all associated data ...	3 hours ago
System Deletion Started	-	-	-	Starting deletion of System 'WinSAT-2' and all asso...	3 hours ago
Transit Event	-	WinSAT-1	-	Satellite 'WinSAT-1' will pass over GroundStation 'U...	12 hours ago
Transit Event	-	WinSAT-1	-	Satellite 'WinSAT-1' will pass over GroundStation 'U...	14 hours ago
Transit Event	-	WinSAT-1	-	Satellite 'WinSAT-1' will pass over GroundStation 'U...	15 hours ago
Transit Event	-	WinSAT-1	-	Satellite 'WinSAT-1' will pass over GroundStation 'U...	17 hours ago
Transit Event	-	WinSAT-1	-	Satellite 'WinSAT-1' will pass over GroundStation 'U...	18 hours ago
Transit Event	-	WinSAT-1	-	Satellite 'WinSAT-1' will pass over GroundStation 'U...	20 hours ago

Communications

466964	Ping	CANCELLED	16 minutes ago	WinSAT-1	Gateway

Figure 26 - KubOS ground station operator and WinSAT web application for satellite imagery viewing.

6.3 Radio Antenna Deployment

To facilitate deployment, the antennas were designed to be fastened to spring loaded armatures that would only release the antennas once the holding wire was burnt. In outer space, materials have certain out gassing properties which limit the types of materials that can be used. Original designs included a nichrome wire tied to the spring-loaded armature which then was wrapped around a wire wound resistor that would heat up and burn the wire to release the spring-loaded armatures thus releasing the antennas. Multiple gauges of nichrome wire and different value resistors were used to burn the wire as fast as possible and with the least amount of voltage. Various tests exposed the difficulty of burning the nichrome wire and the large resistors needed to do so. After some revision, a better method was proposed which would use the nichrome wire as a resistive element and a nylon fishing wire would be used to hold the spring-loaded armature. A contact switch would then apply a voltage to the nichrome wire for the required time needed for the nichrome wire to burn the fishing line and release the spring-loaded doors. Testing different nichrome wire gauges provided different burn times for 10lbs test fishing line.

7 Design Specifications and Evaluation Matrix

7.1 CDH

[Adam]

I2C Communication Frequency

Standard I2C mode is at 100KHz communication; the standard configuration uses an RC (resistor capacitor) circuit on the evaluation board (HSI – high speed internal[11]) at 16MHz to drive the peripheral bus. [11]

Anything greater than 100KHz is considered FMPI2C (fast mode plus I2C) where the data and clock lines can go up to 400KHz. An external crystal oscillator (HSE – high speed external) is on the breakout board the runs at 26MHz to drive the peripheral bus. [11]

Refer to *Appendix F.2* for clock tree to go from HSI or HSE to APB1 where I2C peripherals are.

Pull-Up Resistors

In *Figure 11* there is shown to be two different Vcc sources and pull-up resistor values. The physical build shown in *Figure 16* uses the STM32's internal pull-up resistors that is soldered on the breakout board. Therefore, the build also uses voltage from the STM32 board for Vcc1 and Vcc2.

The GPIO pin configuration has been set so that the internal pull-up resistor is an option. Future WinSAT teams will be adding more devices on the STM32 I2C2 bus which adds more parasitic capacitance. At some point there may be too many devices for the internal pull-up resistor to be enough to hit the voltage logic threshold. In this case, an external pull-up resistor must be used for this bus and Vcc2 will be supplied by the EPS module of the CubeSat.

The maximum allowed bus capacitance is 400pF but Fast Mode Plus I2C (FMPI2C) allows 550pF. The maximum allowed rise time for the SDA and SCL is 1000ns or 120ns for FMPI2C. [17]. Refer to *Appendix F. 3* for pull-up resistor equation and GPIO configuration for I2C pins.

7.2 Radio

[Pierre]

Link Budgets

When considering radio frequency link margins over long distance, there are several variables that can improve or degrade transmission and reception. These constraints were input into link budget excel spread sheets and compiled to produce a final link margin. Primary values input into the link budgets consisted of the total rf power delivered to the antenna, system noise temperature, antenna gain, pointing losses, antenna polarization losses, atmospheric and ionospheric losses as well as the modulation and demodulation method. In addition, values were determined for the losses attributed to cables types, lengths, connections, as well as the signal balancing balun insertion loss. The antenna gains and pointing losses were dependent on the antenna type, length, and frequency. Regarding atmospheric and ionospheric losses, the parameters considered were the minimum elevation angle value along with the uplink and downlink frequencies as they define the distance the signal must travel through and the effect that the ionosphere and atmosphere have on signals at certain frequencies. Specific to the uplink budget, antenna/noise temperatures, spacecraft temperature and the low noise amplifier gain and temperature were considered.

Ultimately, the uplink and downlink budgets for both the Sun-Synchronous and ISS orbits were calculated and used to evaluate the chosen/designed satellite transceiver, ground station and antennas. The link budgets generated respective uplink and downlink link margins defining overall performance with one final value in decibels. All tabulated parameters and final link margins can be found in Appendix D.

Based on the RF Link Budget, the final ground station and satellite properties were calculated. These values would later be used in STK orbital simulations.

Table 2 - Ground Station Parameters

Ground Station Parameters	Power	Link Margin Threshold	Bandwidth	Data Rate	Modulator/ Demodulator	Antenna to LNA Line Loss	System Noise Temp.	LNA Gain	LNA to Receiver Line Loss
Receiver Values	N/A	6.8 dB	N/A	N/A	FSK	3.3 dB	724 Kelvin	23.5 dB	2.3 dB
Transmitter Values	4.36 W	N/A	0.0192 MHz	1200 bps	FSK	N/A	N/A	N/A	N/A

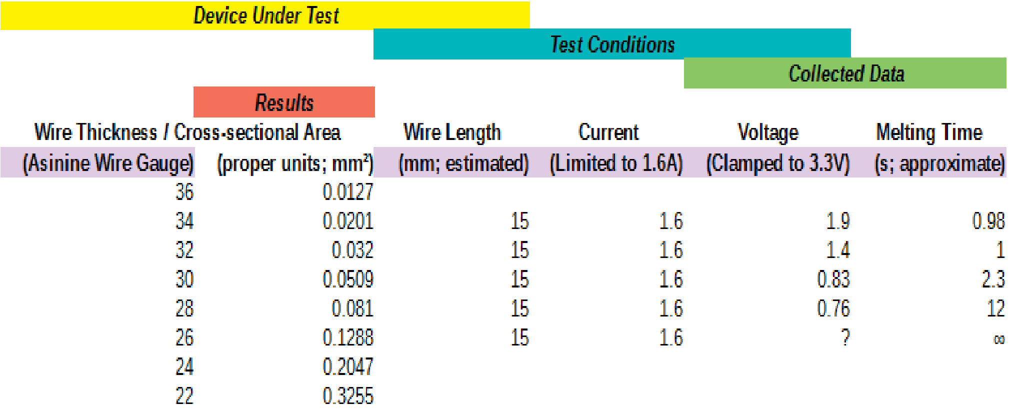
Table 3 - Satellite Parameters

Satellite Parameters	Power	Link Margin Threshold	Bandwidth	Data Rate	Modulator/ Demodulator	Antenna to LNA Line Loss	System Noise Temp.	LNA Gain	LNA to Receiver Line Loss
Receiver Values	N/A	31 dB	N/A	N/A	FSK	1 dB	510 Kelvin	20 dB	N/A
Transmitter Values	-0.73 W	N/A	0.0192 MHz	9600 bps	FSK	N/A	N/A	N/A	N/A

Based on the different antenna orientation patterns, various performance characteristics could be assessed for both the ISS and Sun-Synchronous orbits. Scenario times were conducted over several months to ensure that all possible communication link permutations were considered. Plotting the link margins and bit error rates against all elevation angles provided graphical results for all three antenna orientations. All antenna orientations provided a very low bit error rate which meant that the link margin values would determine the chosen configuration. It is important to note that for all transmitting antenna orientations tested, the receiving antenna was oriented perpendicularly. The vertical transmitting orientation provided most of its link margins within a relatively good link margin. The 45-degree transmitting orientation provided very inconsistent link margins. Finally, the horizontal transmitting orientation provided consistent link margins at high values. Comparing the results for both orbits provided sufficient evidence to conclude that a horizontal transmitting radiation pattern would provide optimal performance. Refer to Appendix E for graphical results.

Referring to the antenna deployment described in the Experimental Methods/Model Validation section, the following table lists the different burn times for 10lbs test fishing line from various nichrome wire gauges.

Table 4 - Fishing Line Burn Time Test Results



Device Under Test					
		Test Conditions			
				Collected Data	
	Results				
Wire Thickness / Cross-sectional Area		Wire Length	Current	Voltage	Melting Time
(Asinine Wire Gauge)	(proper units; mm^2)	(mm; estimated)	(Limited to 1.6A)	(Clamped to 3.3V)	(s; approximate)
36	0.0127				
34	0.0201	15	1.6	1.9	0.98
32	0.032	15	1.6	1.4	1
30	0.0509	15	1.6	0.83	2.3
28	0.081	15	1.6	0.76	12
26	0.1288	15	1.6	?	∞
24	0.2047				
22	0.3255				

Considering both the voltage drop and melting time, the 34-gauge nichrome wire was selected as the most practical option. Although the 36-gauge wire would theoretically provide a faster melting time, the applied voltage would melt the thin nichrome wire. Other voltages were not considered as the onboard EPS module only supports 3.3V and 5V.

8 Budget

[Pierre]

The finalized budget accounts for all subsystem components that our team has developed over the course of this project. Component prices are listed as the proposed designs have all been finalized. Previous WINSAT funding will be used for larger expenses as future members choose to continue this project. Currently, most components have been purchased but have not been used for physical construction as Covid-19 has made this process unfeasible.

Table 5 - Tabulated Budget

Components	Quantity	Unit Price (CAD)	Subtotal (CAD)	Tax (HST) (Estimated)	Total (CAD)
Astrodev Helium 100 UHF/VHF Transceiver	1	$6697.22	$6697.22	$870.64	$7567.86
U.FL (UMCC) Connector Jack, Male Pin 50Ohm Surface Mount Solder	6	$0.72	$4.32	$0.65	$4.97
Cable Assembly Coaxial U.FL (UMCC) 1.37mm OD Coaxial Cable 7.874" (200.00mm)	4	$3.35	$13.4	$1.75	$15.15
RF Balun - 50/50 Ohm, 100MHz ~ 1GHz IL 1.0dB	2	$1.88	$3.76	$0.48	$4.24
Nichrome 80 Wire Sample Pack 22,24,26,28,30,32,34,36 Gauge	1	$35.2	$35.2	$4.58	$39.78

RF Deployment - Antenna Release Switches	4	$3.8	$15.2	$2	$17.2
Pumpkin Motherboard Module 2 (MBM2)	1	$7354.49	$7354.49	$956.08	$8310.57
Raspberry Pi Compute Stick	1	$132.64	$132.64	$17.24	$149.88
Arducam 2MP Mini Plus Camera Module	2	$38.29	$76.58	$9.96	$86.53
Edmunds 6mm Lens	1	$76.52	$76.52	$9.95	$86.47
Edmunds 16mm Lens	1	$49.50	$49.50	$6.44	$55.94
STM32-F446RE	1	$22.00	$22.00	$2.79	$24.79
NXP 9-DOF	1	$23.99	$23.99	$3.12	$27.11

The budget is within what was set during the initial design phase and initial rounds of funding and fits within the WinSAT team budget.

9 Conclusions

The team was able to design, develop, build, and test three subsystems onboard the WinSAT CubeSat satellite – Payload, CDH, and RF. Using both physical implementations as well as simulation models (when a physical model was unachievable), the design for the subsystems was verified according to the original CSDC competition guidelines [1].

With the design of the Payload subsystem, it has been verified through satellite simulation the optical design and parameters and software implementation are able to meet the mission requirements with regard to imaging. The primary payload optical design allows for the acquisition of a single image, covering between an area of 40kmx40km and 100kmx100km as specified in the design requirements.

The CDH subsystem has been developed and designed to provide successful communication between all subsystems on the satellite. The software design was tested using the "Flat-Sat" bench setup, including integration with other subsystems. The satellite is able to retrieve and execute both immediate and time-tagged commands from ground station. As well, the satellite successfully time-tags all onboard telemetry and then is able to downlink those to the ground station within a 12-hour period.

Altogether, the RF communications link satisfies the required guidelines defined by the CSDS. Using the 9600-bps downlink and 1200-bps uplink data rates, antenna selection, design and basic physical construction was finalized. The required 6 dB link margin at minimum 10º elevation angle was verified within the link budget and orbital simulations.

All of this combined work contributes to completing and fulfilling the main mission goal of WinSAT satellite – complete the concept of operations for acquiring a single "Selfie-Sat" image when commanded by an ARO. The design for the RF, CDH, and Payload subsystems meets the design requirements and has been verified to adequately facilitate the main mission goal.

10 References

[1] C. Satellite and D. Challenge, “The Canadian Satellite Design Challenge General Rules & Requirements,” no. 3, 2014.

[2] D. J. F. Miranda, M. Ferreira, F. Kucinskis, and D. McComas, “A comparative survey on flight software frameworks for ‘new space’ nanosatellite missions,” *J. Aerosp. Technol. Manag.*, vol. 11, 2019, doi: 10.5028/jatm.v11.1081.

[3] B. Dunbar and Y. Kovo, “Command and Data Handling | NASA,” *National Aeronautics and Space Administration*, 2020. https://www.nasa.gov/smallsat-institute/sst-soa/command-and-data-handling (accessed Aug. 07, 2020).

[4] M. Lankinen and E. Kallio, “Design and Testing of Antenna Deployment System for Aalto-1 Satellite Title: Design and Testing of Antenna Deployment System for Aalto-1 Satellite,” 2015.

[5] S. B. M. Zaki, M. H. Azami, T. Yamauchi, S. Kim, H. Masui, and M. Cho, “Design, Analysis and Testing of Monopole Antenna Deployment Mechanism for BIRDS-2 CubeSat Applications,” *J. Phys. Conf. Ser.*, vol. 1152, no. 1, 2019, doi: 10.1088/1742-6596/1152/1/012007.

[6] S. Marholm, “Antenna Systems for NUTS,” no. July 2012, 2012.

[7] Z. J. Leffke, R. W. Mcgwier, D. G. Sweeney, and S. S. Bailey, “Distributed Ground Station Network For CubeSat Communications,” 2013.

[8] C. B. Crail and M. Herson, “Ranking CubeSat Communication Systems Using a Value-centric Framework,” 2007.

[9] Arducam, *ArduCAM-M-2MP Camera Shield 2MP SPI Camera User Guide*. 2015.

[10] Kubos Corporation, “Kubos Documentation,” 2020. https://docs.kubos.com/1.21.0/index.html (accessed Aug. 05, 2020).

[11] “RM0390 Reference manual STM32F446xx advanced Arm ®-based 32-bit MCUs For information on the Arm ® Cortex ®-M4 with FPU core, refer to the Cortex ®-M4 Technical Reference Manual. For information on the Cortex ®-M4 with FPU, refer to the STM32F3xx/F4xxx Cortex ®-M4 with FPU programming manual (PM0214),” 2018. [Online]. Available: www.st.com:

[12] “Mbed OS | Mbed.” https://os.mbed.com/mbed-os/ (accessed Aug. 07, 2020).

[13] “CubeSat KitTM Motherboard Module (MBM) 2,” vol. 2, no. March. Pumpkin, pp. 1–23, 2019, [Online]. Available: http://www.pumpkininc.com/space/datasheet/710-01362-E_DS_MBM_2.pdf.

[14] “STM32 IDEs - STMicroelectronics.” https://www.st.com/en/development-tools/stm32-ides.html (accessed Aug. 07, 2020).

[15] “STM32duino.” https://github.com/stm32duino (accessed Aug. 07, 2020).

[16] N. Semiconductors, “UM10204 I 2 C-bus specification and user manual Rev. 6-4 April 2014 User manual Document information Info Content.” [Online]. Available: http://www.nxp.com.

11 Appendices

APPENDIX A

Payload Optical, Orbital, and Sensor Viewing Parameters Calculations

A-1 Orbital Parameters

The following figure is a visual of some of the orbital and sensor viewing parameters that will be calculated that supplements the calculations for better understanding.

Assuming orbital altitude of H=400km and the radius of the Earth Re=6378.14km, calculate the orbital period (P):

$$P = 1.658669\text{e-}4\ x\left(\Re + H\right)^{3/2}$$

$$P = 92.56\,min$$

Compute the ground track velocity (Vg):

$$Vg = \left(2\pi Re\right)/P$$

$$Vg = \left(2\pi\ x\ 6378.14\right)/92.56$$

$$Vg = 7.216\,km/s$$

Compute the node shift (L) or the change in the angular position of each consecutive orbit:

$$\Delta L = \left(\frac{P}{1436\,min} 360\,deg\right)$$

$$\Delta L = \left(\frac{92.56\,min}{1436\,min} 360\,deg\right)$$

$$\Delta L = \left(\frac{92.56\,min}{1436\,min} 360\,deg\right)$$

$$\Delta L = 23.2\,degrees$$

A-2 Sensor Viewing Parameters

Assuming a maximum off-nadir angle (n) of 25 degrees or 0.4363 rad as specified in the competition rules:

Compute the earth angular radius (p) as seen and observed from the satellite:

$$\sin(p)=\frac{\Re}{\Re+H}$$

$$\sin(p)=\frac{6378.14\,km}{6378.14+400\,km}$$

$$p=1.23\text{rad}$$

Compute the maximum distance to the horizon from the satellite (Dmax):

$$Dmax=\left[(\Re+H)^2-Re^2\right]^{1/2}$$

$$Dmax=\left[(6378.14+400)^2-6378.14^2\right]^{1/2}$$

$$Dmax=2294\,km$$

Compute the minimum elevation angle (e), the angle between the ground and the satellite from the target location. This is the elevation angle at maximum off nadir angle of 25 deg.

$$\cos(e)=\frac{\sin(n)}{\sin(p)}$$

$$\cos(e)=\frac{\sin(0.4363\,rad)}{\sin(1.23\,rad)}$$

$$e = 1.11 \text{ rad}$$

Compute the max incidence angle (IA) or incidence angle at maximum off nadir angle of 25 deg:

$$IA=\frac{\pi}{2}-e_{min}$$

$$IA=\frac{\pi}{2}-1.11\,rad$$

$$IA=0.466\,rad$$

Computer maximum earth central angle () or earth central angle at max off nadir angle:

$$n+\lambda+e=\pi/2$$

$$\lambda=0.0294\,rad$$

Finally, compute the slant range (Rs) or the distance from the satellite to the earth's surface at the maximum nadir angle (n):

$$Rs=\Re\left(\frac{\sin(\lambda)}{\sin(n)}\right)$$

$$Rs=6378.14\left(\frac{\sin(0.0294)}{\sin(0.4363)}\right)$$

$$Rs=444\,km$$

A-3. Sensor Optics

Assuming a detector width of d=2.2um (from camera specifications), calculate the required focal length to meet the spatial resolution of 40m at 400km altitude:

$$f=\frac{hxd}{X}$$

$$f=\frac{400\,km\,x\,2.2\,um}{40\,m}$$

$$f=22\,mm$$

Assume a lens of focal length of 22mm.

Calculate field of view, image coverage, and spatial resolution when using a 22mm lens and the OV2640 camera chip:

$$Spatial\,Resolution=\frac{hxd}{f}$$

$$Spatial\,Resolution=\frac{400\,km\,x\,2.2\,um}{22\,mm}$$

$$Spatial\,Resolution=40\,m$$

$$\mathit{Horiz\ Field\ of\ View} = 2\tan^{-1}\left(\frac{\left(\frac{\mathit{imageWidth}}{2}\right)}{f}\right)$$

$$\mathit{Horiz\ Field\ of\ View} = 2\tan^{-1}\left(\frac{\left(\frac{2.2\,\mathit{um}\ x\ 1600}{2}\right)}{22\,\mathit{mm}}\right)$$

$$\mathit{Horiz\ Field\ of\ View} = 10.06\,\mathit{deg}$$

$$\mathit{Vert\ Field\ of\ View} = \mathit{Horiz\ Field\ of\ View}\ x\ 3/4$$

$$\mathit{Vert\ Field\ of\ View} = 7.54\ \mathit{deg}$$

$$\mathit{Horiz\ Image\ Coverage} = 2\ x\ h\ x\ \tan\left(\frac{\mathit{HFOV}}{2}\right)$$

$$\mathit{Horiz\ Image\ Coverage} = 2\ x\ 400\,\mathit{km}\ x\ \tan\left(\frac{10.06}{2}\right)$$

$$\mathit{Horiz\ Image\ Coverage} = 70.4\ \mathit{km}$$

$$\mathit{Vert\ Image\ Coverage} = 2\ x\ h\ x\ \tan\left(\frac{\mathit{VFOV}}{2}\right)$$

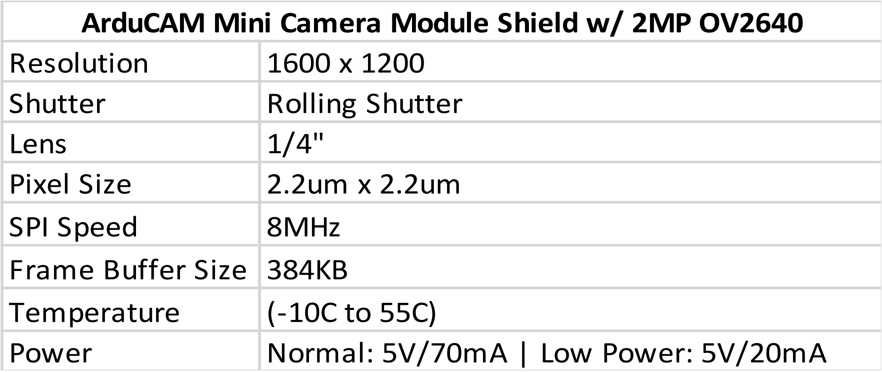
$$\mathit{Vert\ Image\ Coverage} = 2\ x\ 400\,\mathit{km}\ x\ \tan\left(\frac{7.54}{2}\right)$$

$$\mathit{Vert\ Image\ Coverage} = 2\ x\ 400\,\mathit{km}\ x\ \tan\left(\frac{7.54}{2}\right)$$

$$\mathit{Vert\ Image\ Coverage} = 52.7\ \mathit{km}$$

A-3 Payload optical tables and figures.

Table A. 1 - Arducam Mini Camera Module Specifications [9].



ArduCAM Mini Camera Module Shield w/ 2MP OV2640	
Resolution	1600 x 1200
Shutter	Rolling Shutter
Lens	1/4"
Pixel Size	2.2um x 2.2um
SPI Speed	8MHz
Frame Buffer Size	384KB
Temperature	(-10C to 55C)
Power	Normal: 5V/70mA | Low Power: 5V/20mA

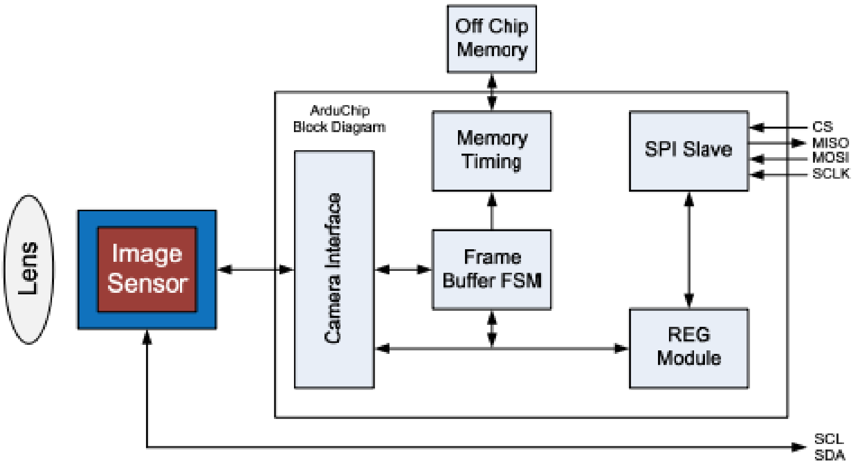
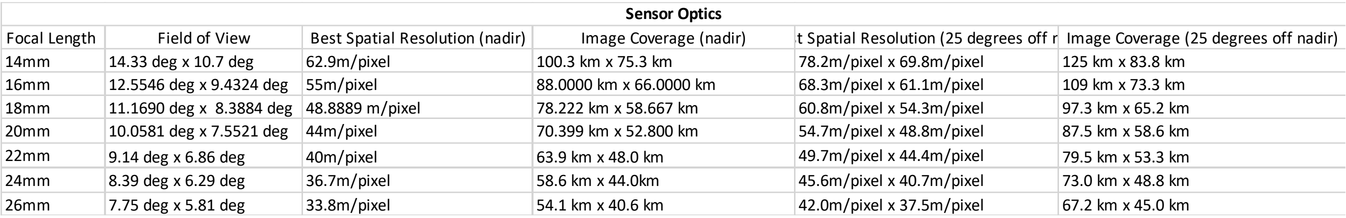


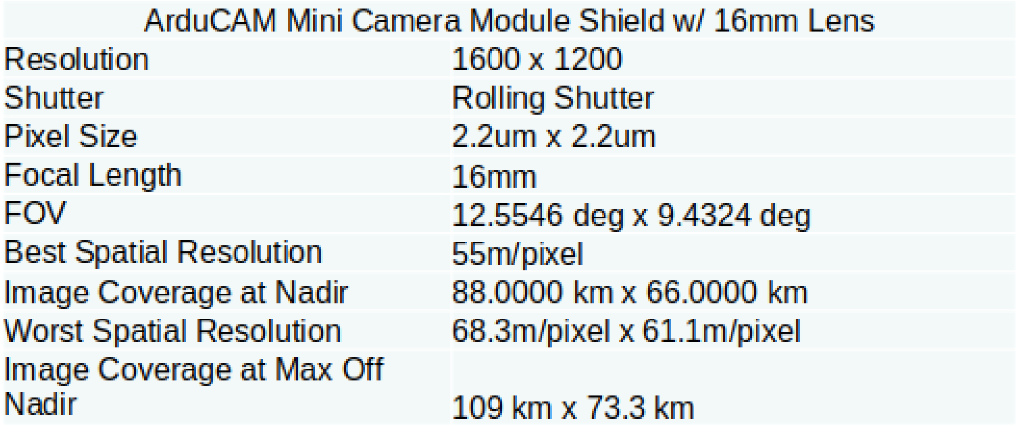
Figure A. 1 - ArduCam Mini Shield Block Diagram [9].

Table A. 2 - Lens comparison on OV2640 image sensor for single-point “Space-Selfie” image acquisition primary objective.



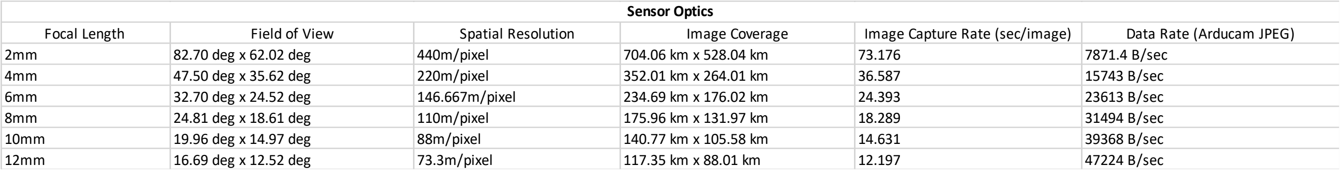
Sensor Optics					
Focal Length	Field of View	Best Spatial Resolution (nadir)	Image Coverage (nadir)	t Spatial Resolution (25 degrees off r	Image Coverage (25 degrees off nadir)
14mm	14.33 deg x 10.7 deg	62.9m/pixel	100.3 km x 75.3 km	78.2m/pixel x 69.8m/pixel	125 km x 83.8 km
16mm	12.5546 deg x 9.4324 deg	55m/pixel	88.0000 km x 66.0000 km	68.3m/pixel x 61.1m/pixel	109 km x 73.3 km
18mm	11.1690 deg x 8.3884 deg	48.8889 m/pixel	78.222 km x 58.667 km	60.8m/pixel x 54.3m/pixel	97.3 km x 65.2 km
20mm	10.0581 deg x 7.5521 deg	44m/pixel	70.399 km x 52.800 km	54.7m/pixel x 48.8m/pixel	87.5 km x 58.6 km
22mm	9.14 deg x 6.86 deg	40m/pixel	63.9 km x 48.0 km	49.7m/pixel x 44.4m/pixel	79.5 km x 53.3 km
24mm	8.39 deg x 6.29 deg	36.7m/pixel	58.6 km x 44.0km	45.6m/pixel x 40.7m/pixel	73.0 km x 48.8 km
26mm	7.75 deg x 5.81 deg	33.8m/pixel	54.1 km x 40.6 km	42.0m/pixel x 37.5m/pixel	67.2 km x 45.0 km

Table A. 3 - Primary payload optical sensor parameters with selected 16 mm lens.



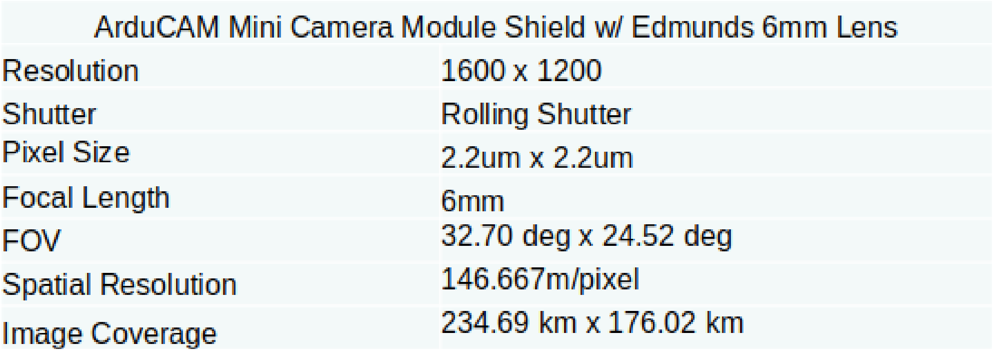
ArduCAM Mini Camera Module Shield w/ 16mm Lens	
Resolution	1600 x 1200
Shutter	Rolling Shutter
Pixel Size	2.2um x 2.2um
Focal Length	16mm
FOV	12.5546 deg x 9.4324 deg
Best Spatial Resolution	55m/pixel
Image Coverage at Nadir	88.0000 km x 66.0000 km
Worst Spatial Resolution	68.3m/pixel x 61.1m/pixel
Image Coverage at Max Off Nadir	109 km x 73.3 km

Table A. 4 - Lens comparison on OV2640 image sensor for secondary area/global coverage objective.



Sensor Optics					
Focal Length	Field of View	Spatial Resolution	Image Coverage	Image Capture Rate (sec/image)	Data Rate (Arducam JPEG)
2mm	82.70 deg x 62.02 deg	440m/pixel	704.06 km x 528.04 km	73.176	7871.4 B/sec
4mm	47.50 deg x 35.62 deg	220m/pixel	352.01 km x 264.01 km	36.587	15743 B/sec
6mm	32.70 deg x 24.52 deg	146.667m/pixel	234.69 km x 176.02 km	24.393	23613 B/sec
8mm	24.81 deg x 18.61 deg	110m/pixel	175.96 km x 131.97 km	18.289	31494 B/sec
10mm	19.96 deg x 14.97 deg	88m/pixel	140.77 km x 105.58 km	14.631	39368 B/sec
12mm	16.69 deg x 12.52 deg	73.3m/pixel	117.35 km x 88.01 km	12.197	47224 B/sec

Table A. 5 - Secondary payload optical sensor parameters with selected 6 mm lens.



ArduCAM Mini Camera Module Shield w/ Edmunds 6mm Lens	
Resolution	1600 x 1200
Shutter	Rolling Shutter
Pixel Size	2.2um x 2.2um
Focal Length	6mm
FOV	32.70 deg x 24.52 deg
Spatial Resolution	146.667m/pixel
Image Coverage	234.69 km x 176.02 km

APPENDIX B

Optical Sensor Comparison

Table B. 1 - Comparison of camera modules for optical payload selection.

	Xcam	**ov2640**	**e-CAM20_CU0230_MOD**	**C329-SPI**	**AR0135AT**	**NOIS1SM1000A**	**UCAM-III**
Sensor	1.3 MP CMOS	CMOS	2 MP CMOS	CMOS	1.2 MP CMOS	CMOS	CMOS
Resolution	1280x1024	1600x1200	1920x1080	640x480	1280x960	1024x1024	656x496
Focal Length	9.6mm		6.72mm				
Sensor size		1/4"				1"	
Lens		s-mount	various	various	DFM 27UR0135-ML (Camera and Sensor)	Various	Various
Wavelength (nm)	400-650		400-650				
Dimensions	9.5x9.1x2.7		30mmx30mm	20x28mm	30mmx30mmx10mm	26.8mmx26.8mm	32mmx32mmx21mm
Mass	85g				15g		~6g
Operating Temp	-25><65	-30><70			-40><105	-40><85	-30><85
Survival Temp	-35><75		-40><105		-40><120		-40><105
Data Format	8 bit	8, 10 bit	10, 12 bit	8 bit	8,12 bit	10 bit	8,16 bit
Image Compression	JPEG	JPEG	RAW	JPEG	RAW	RAW	JPEG
Data Interface	I2C/SPI	SCCB/I2C/SPI	I2C	SPI	Parallel Pixel	SPI	
Power Consumption	0.845 W	140mW	0.7 W		1.25 W	0.4 W	
Voltage		1.7-3.3V	1.8-2.8V	3.3V	4.75V - 5.25V	5V	5V
Flight Heritage	Y	Y	N	Y	N	Y	Y
Data Sheet	Xcam Data	ov2640 Data	eCom Data	C329 Data	AR0 Data	NOIS Data	uCAM Data

APPENDIX C

STK Image and Coverage Analysis

C-1. Ground Track Figures

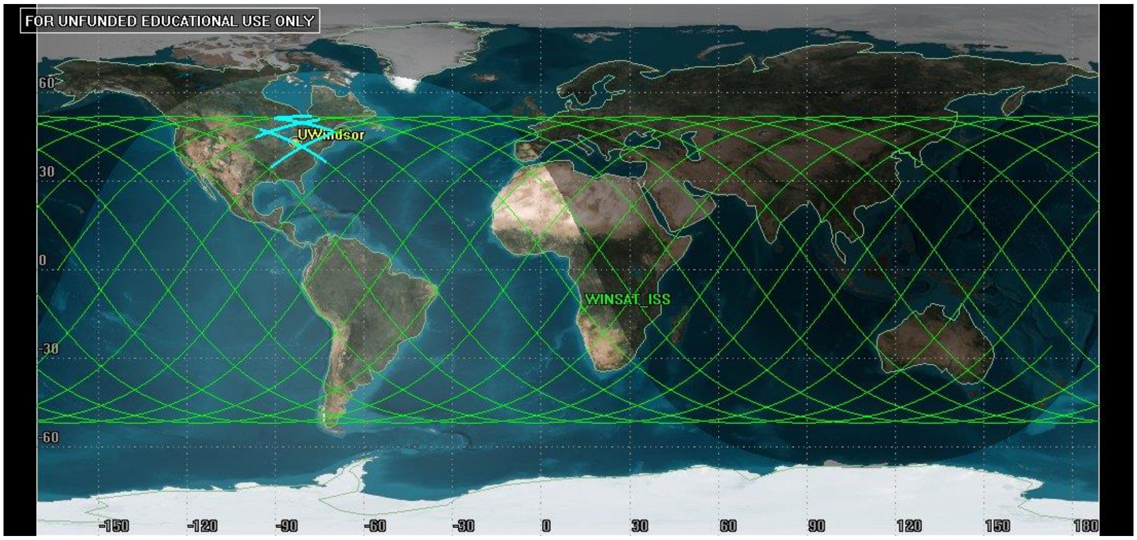


Figure C. 1 - ISS orbit mission ground track.

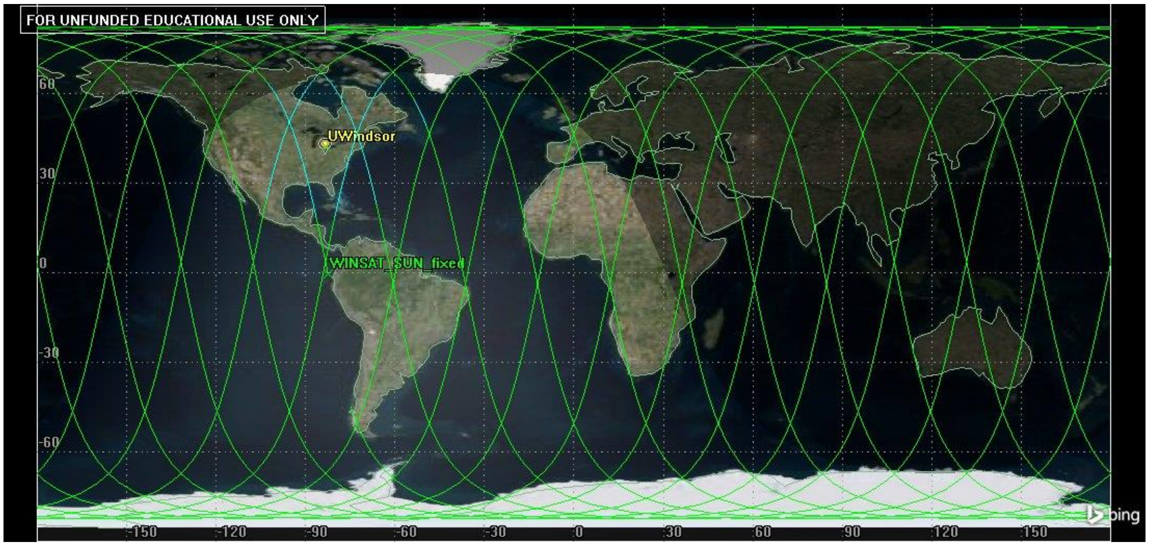


Figure C. 2 - Sun Synchronous orbit mission ground track.

C-2. Primary Payload Point Access Analysis

Primary Payload Access Statistics and Results

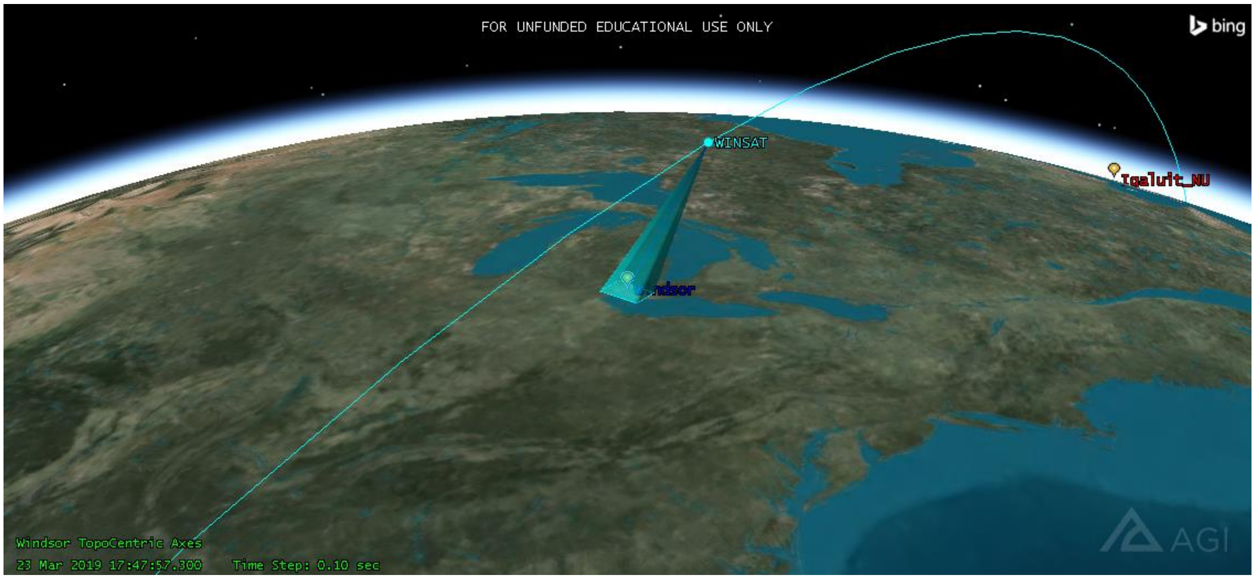


Figure C. 3 - Payload Windsor access.

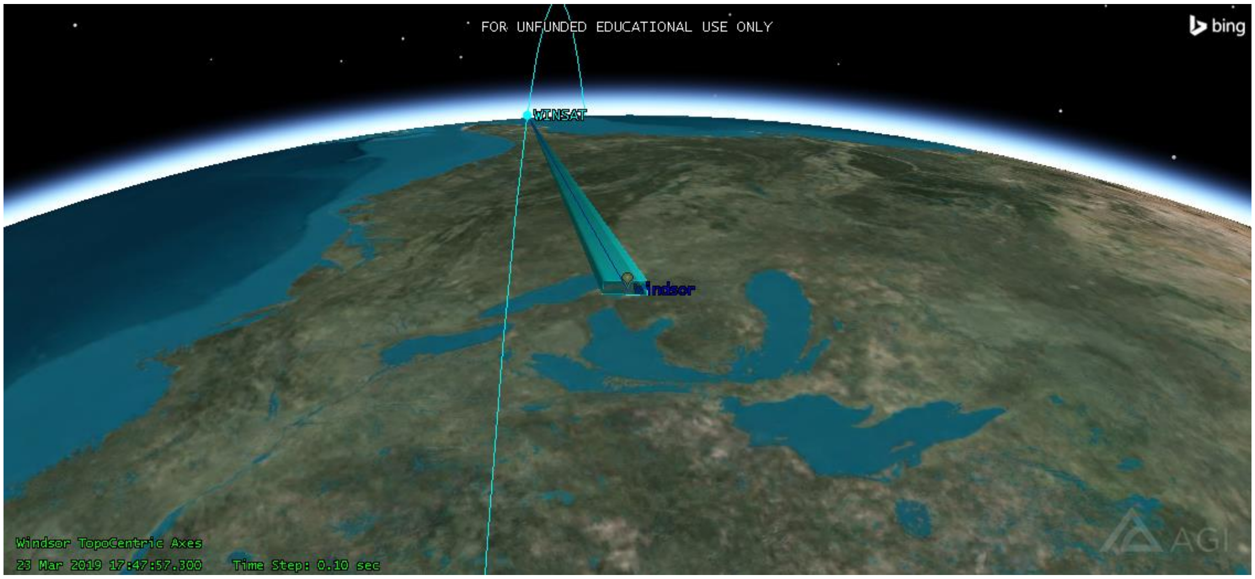
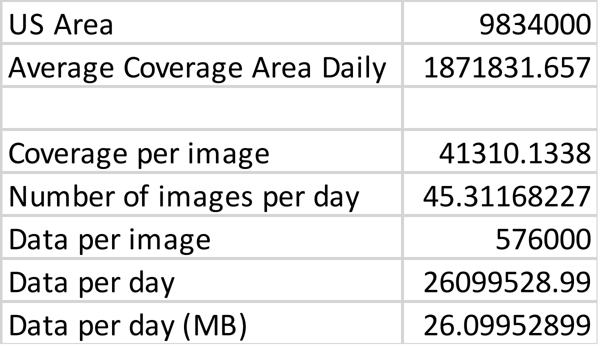


Figure C. 4 - Payload Windsor access.

C-3. Secondary Payload Coverage Analysis

Table C. 1 - Coverage statistics of secondary payload in sun synchronous orbit at 400 km altitude.



US Area	9834000
Average Coverage Area Daily	1871831.657
Coverage per image	41310.1338
Number of images per day	45.31168227
Data per image	576000
Data per day	26099528.99
Data per day (MB)	26.09952899

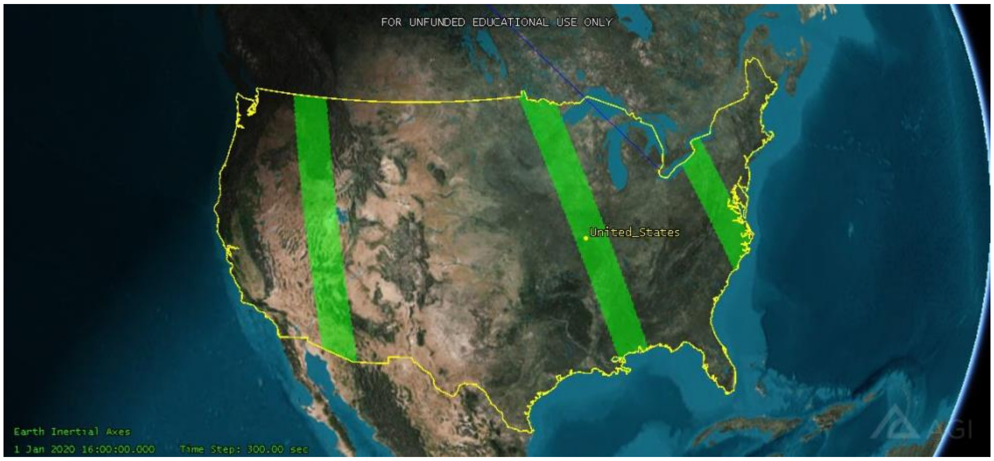


Figure C. 5 - US coverage analysis after 1 day.

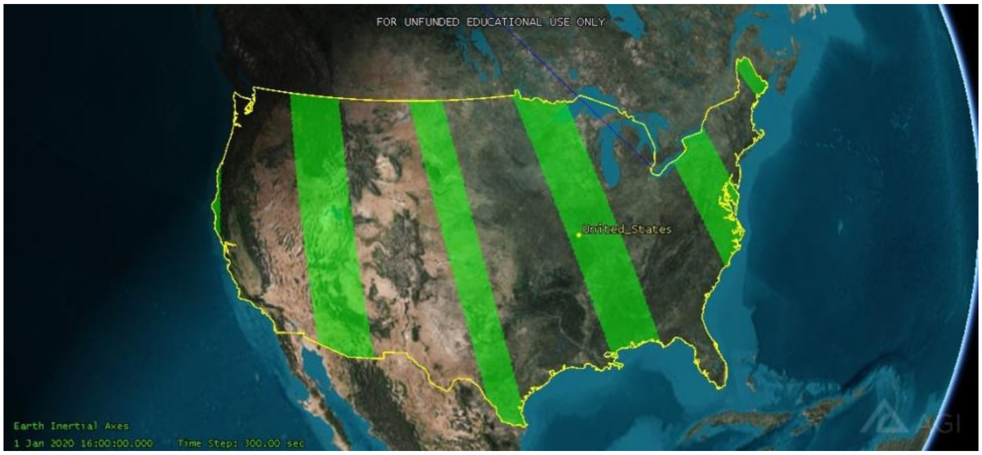


Figure C. 6 - US coverage analysis after 3 days.

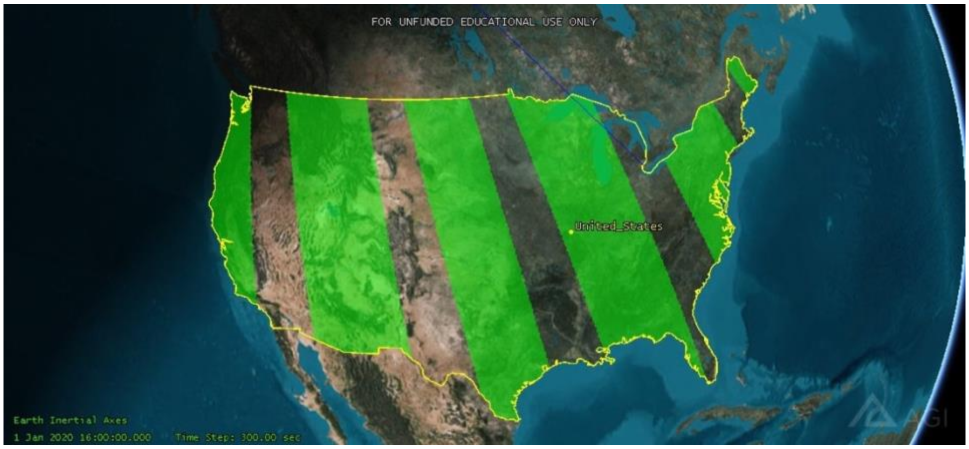


Figure C. 7 - US coverage analysis after 7 days.

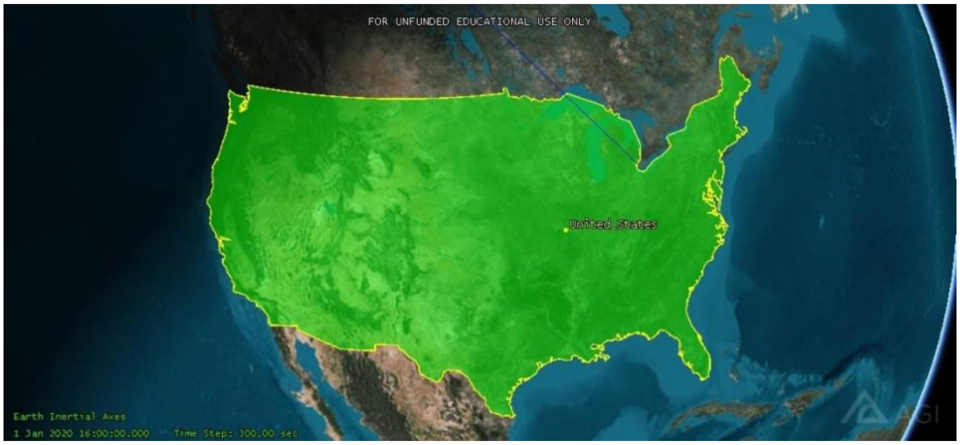
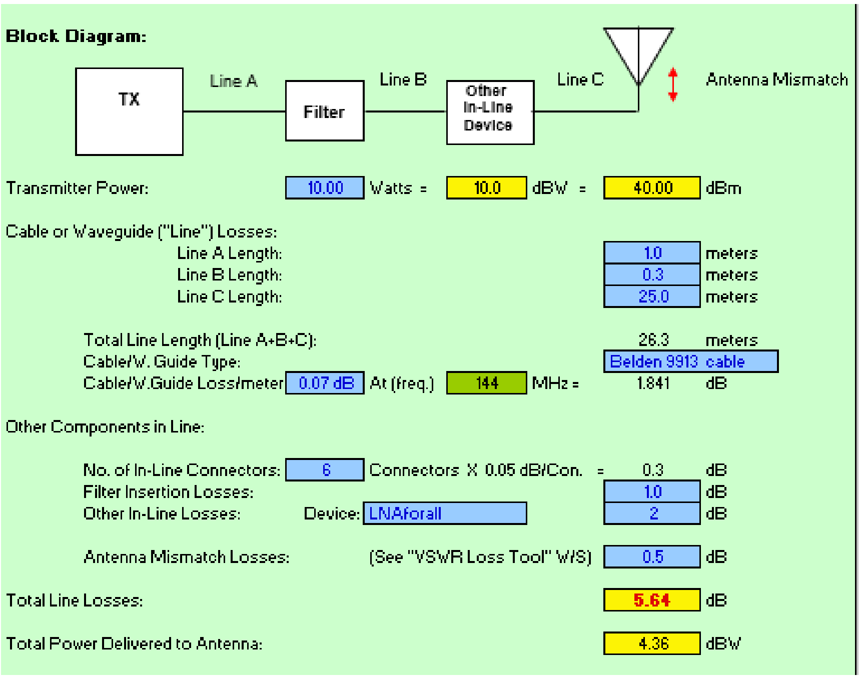


Figure C. 8 - US coverage analysis after 14 days.

APPENDIX D

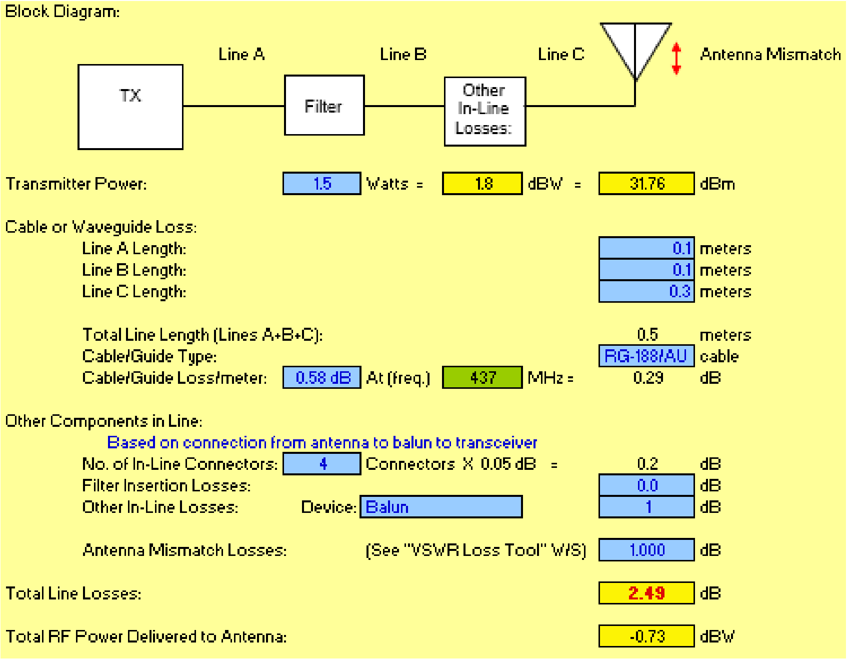


Block Diagram:

TX — Line A — Filter — Line B — Other In-Line Device — Line C — Antenna Mismatch

Transmitter Power:	10.00	Watts =	10.0	dBW =	40.00	dBm
Cable or Waveguide ("Line") Losses:						
Line A Length:					1.0	meters
Line B Length:					0.3	meters
Line C Length:					25.0	meters
Total Line Length (Line A+B+C):					26.3	meters
Cable/W. Guide Type:					Belden 9913 cable	
Cable/W.Guide Loss/meter	0.07 dB	At (freq.)	144	MHz =	1.841	dB
Other Components in Line:						
No. of In-Line Connectors:	6	Connectors X 0.05 dB/Con. =			0.3	dB
Filter Insertion Losses:					1.0	dB
Other In-Line Losses:	Device:	LNAforall			2	dB
Antenna Mismatch Losses:		(See "VSWR Loss Tool" W/S)			0.5	dB
Total Line Losses:					5.64	dB
Total Power Delivered to Antenna:					4.36	dBW

APPENDIX D. 1 - Uplink Transmitter System (At Ground Station)

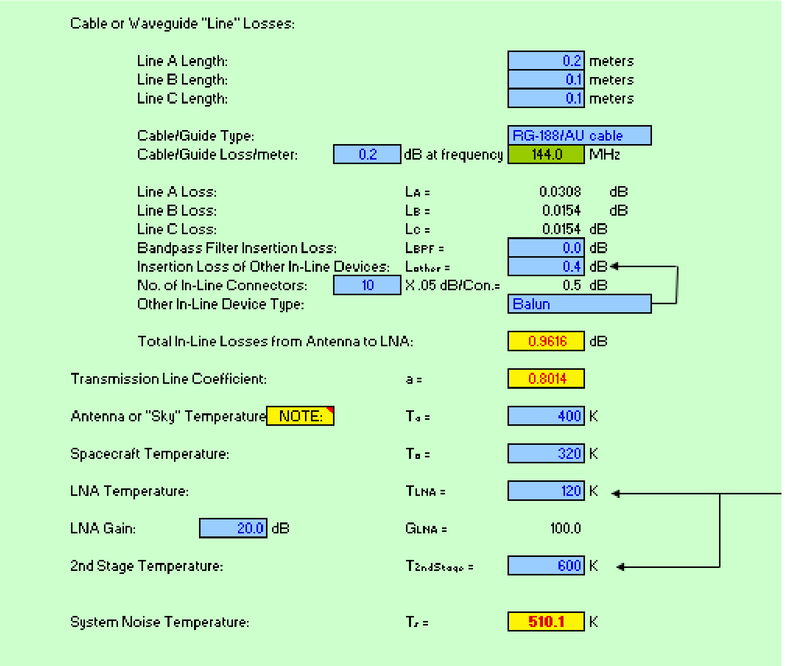


Block Diagram:

TX — Line A — Filter — Line B — Other In-Line Losses: — Line C — Antenna Mismatch

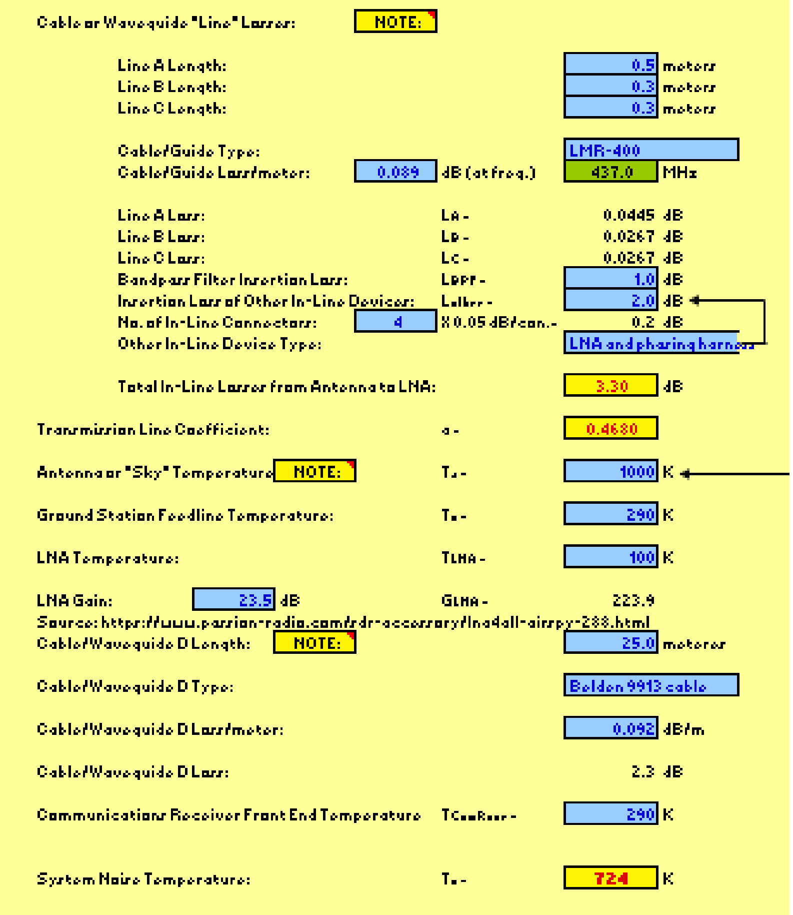
Transmitter Power:	1.5	Watts =	1.8	dBW =	31.76	dBm
Cable or Waveguide Loss:						
Line A Length:					0.1	meters
Line B Length:					0.1	meters
Line C Length:					0.3	meters
Total Line Length (Lines A+B+C):					0.5	meters
Cable/Guide Type:					RG-188/AU	cable
Cable/Guide Loss/meter:	0.58 dB	At (freq.)	437	MHz =	0.29	dB
Other Components in Line:						
Based on connection from antenna to balun to transceiver						
No. of In-Line Connectors:	4	Connectors X 0.05 dB =			0.2	dB
Filter Insertion Losses:					0.0	dB
Other In-Line Losses:	Device:	Balun			1	dB
Antenna Mismatch Losses:		(See "VSWR Loss Tool" W/S)			1.000	dB
Total Line Losses:					2.49	dB
Total RF Power Delivered to Antenna:					-0.73	dBW

APPENDIX D. 2 - Downlink Transmitter System (At Spacecraft)



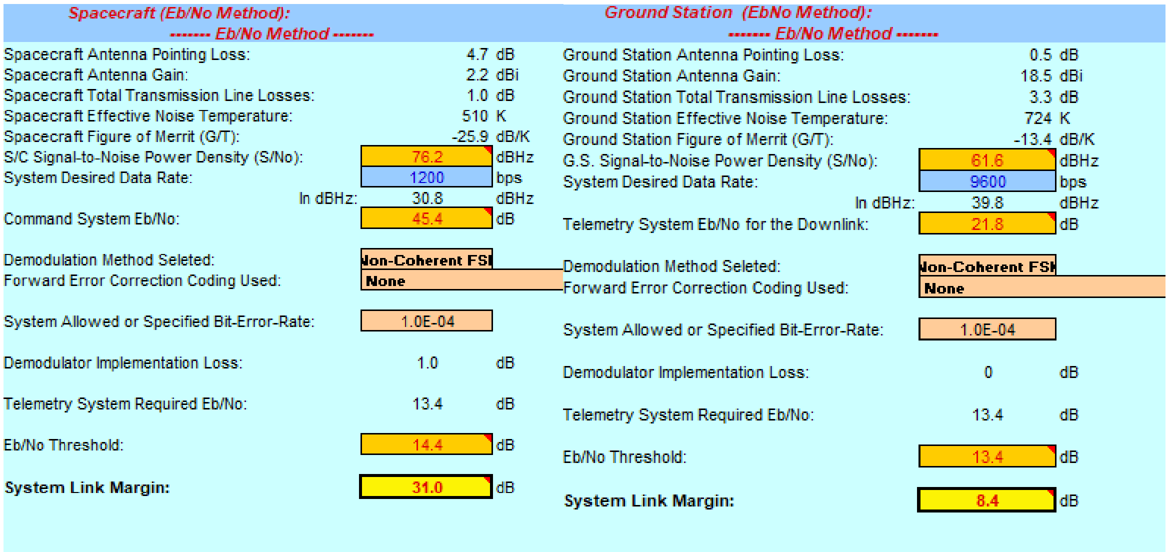
Cable or Waveguide "Line" Losses:					
Line A Length:				0.2	meters
Line B Length:				0.1	meters
Line C Length:				0.1	meters
Cable/Guide Type:				RG-188/AU cable	
Cable/Guide Loss/meter:	0.2	dB at frequency		144.0	MHz
Line A Loss:			L_A =	0.0308	dB
Line B Loss:			L_B =	0.0154	dB
Line C Loss:			L_C =	0.0154	dB
Bandpass Filter Insertion Loss:			L_{BPF} =	0.0	dB
Insertion Loss of Other In-Line Devices:			L_{other} =	0.4	dB
No. of In-Line Connectors:	10		X .05 dB/Con.=	0.5	dB
Other In-Line Device Type:				Balun	
Total In-Line Losses from Antenna to LNA:				0.9616	dB
Transmission Line Coefficient:			a =	0.8014	
Antenna or "Sky" Temperature	NOTE:		T_a =	400	K
Spacecraft Temperature:			T_0 =	320	K
LNA Temperature:			T_{LNA} =	120	K
LNA Gain:	20.0	dB	G_{LNA} =	100.0	
2nd Stage Temperature:			$T_{2ndStage}$ =	600	K
System Noise Temperature:			T_s =	510.1	K

APPENDIX D. 3 - Uplink Receiver System (At Spacecraft)



Cable or Waveguide "Line" Losses:	NOTE:				
Line A Length:				0.5	meters
Line B Length:				0.3	meters
Line C Length:				0.3	meters
Cable/Guide Type:				LMR-400	
Cable/Guide Loss/meter:	0.089	dB (at freq.)		437.0	MHz
Line A Loss:			L_A =	0.0445	dB
Line B Loss:			L_B =	0.0267	dB
Line C Loss:			L_C =	0.0267	dB
Bandpass Filter Insertion Loss:			L_{BPF} =	1.0	dB
Insertion Loss of Other In-Line Devices:			L_{other} =	2.0	dB
No. of In-Line Connectors:	4		X 0.05 dB/con.=	0.2	dB
Other In-Line Device Type:				LNA and phasing harness	
Total In-Line Losses from Antenna to LNA:				3.30	dB
Transmission Line Coefficient:			a =	0.4680	
Antenna or "Sky" Temperature	NOTE:		T_a =	1000	K
Ground Station Feedline Temperature:			T_0 =	290	K
LNA Temperature:			T_{LNA} =	100	K
LNA Gain:	23.5	dB	G_{LNA} =	223.9	
Source: https://www.passion-radio.com/sdr-accessory/lna4all-airspy-288.html					
Cable/Waveguide D Length:	NOTE:			25.0	meteres
Cable/Waveguide D Type:				Belden 9913 cable	
Cable/Waveguide D Loss/meter:				0.092	dB/m
Cable/Waveguide D Loss:				2.3	dB
Communications Receiver Front End Temperature			$T_{ComRecr}$ =	290	K
System Noise Temperature:			T_s =	724	K

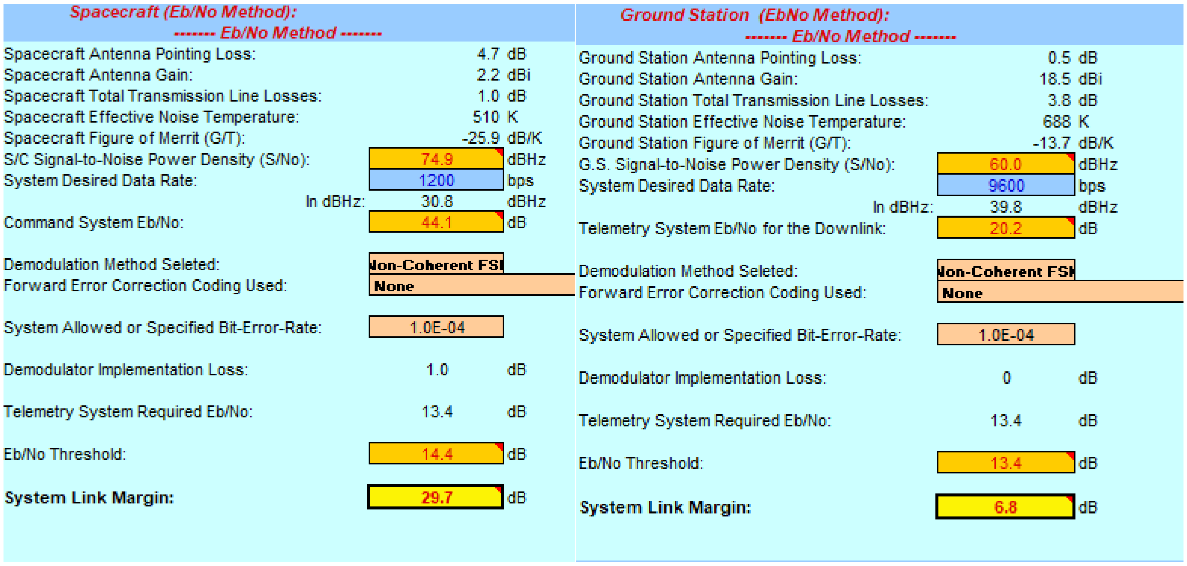
APPENDIX D. 4 - Downlink Receiver System (At Ground Station)



Spacecraft (Eb/No Method):		
------- Eb/No Method -------		
Spacecraft Antenna Pointing Loss:	4.7	dB
Spacecraft Antenna Gain:	2.2	dBi
Spacecraft Total Transmission Line Losses:	1.0	dB
Spacecraft Effective Noise Temperature:	510	K
Spacecraft Figure of Merrit (G/T):	-25.9	dB/K
S/C Signal-to-Noise Power Density (S/No):	76.2	dBHz
System Desired Data Rate:	1200	bps
In dBHz:	30.8	dBHz
Command System Eb/No:	45.4	dB
Demodulation Method Seleted:	**Non-Coherent FSI**	
Forward Error Correction Coding Used:	**None**	
System Allowed or Specified Bit-Error-Rate:	1.0E-04	
Demodulator Implementation Loss:	1.0	dB
Telemetry System Required Eb/No:	13.4	dB
Eb/No Threshold:	14.4	dB
System Link Margin:	**31.0**	dB

Ground Station (EbNo Method):		
------- Eb/No Method -------		
Ground Station Antenna Pointing Loss:	0.5	dB
Ground Station Antenna Gain:	18.5	dBi
Ground Station Total Transmission Line Losses:	3.3	dB
Ground Station Effective Noise Temperature:	724	K
Ground Station Figure of Merrit (G/T):	-13.4	dB/K
G.S. Signal-to-Noise Power Density (S/No):	61.6	dBHz
System Desired Data Rate:	9600	bps
In dBHz:	39.8	dBHz
Telemetry System Eb/No for the Downlink:	21.8	dB
Demodulation Method Seleted:	**Non-Coherent FSK**	
Forward Error Correction Coding Used:	**None**	
System Allowed or Specified Bit-Error-Rate:	1.0E-04	
Demodulator Implementation Loss:	0	dB
Telemetry System Required Eb/No:	13.4	dB
Eb/No Threshold:	13.4	dB
System Link Margin:	**8.4**	dB

APPENDIX D. 5 - Uplink ISS Orbit (Left) and Downlink ISS Orbit (Right)

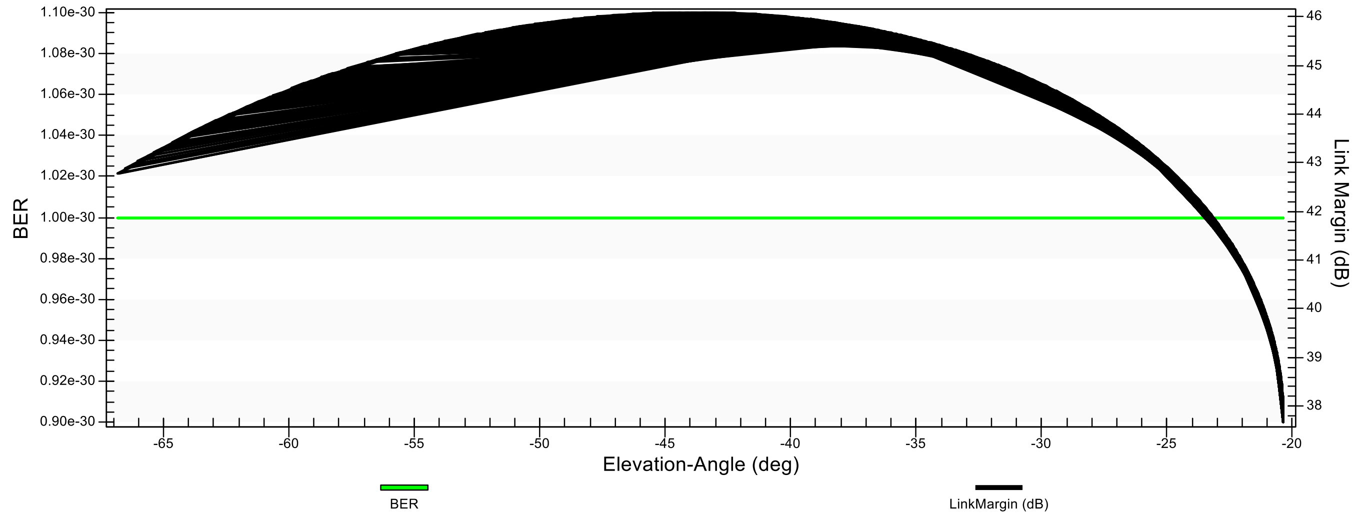


Spacecraft (Eb/No Method):		
------- Eb/No Method -------		
Spacecraft Antenna Pointing Loss:	4.7	dB
Spacecraft Antenna Gain:	2.2	dBi
Spacecraft Total Transmission Line Losses:	1.0	dB
Spacecraft Effective Noise Temperature:	510	K
Spacecraft Figure of Merrit (G/T):	-25.9	dB/K
S/C Signal-to-Noise Power Density (S/No):	74.9	dBHz
System Desired Data Rate:	1200	bps
In dBHz:	30.8	dBHz
Command System Eb/No:	44.1	dB
Demodulation Method Seleted:	**Non-Coherent FSI**	
Forward Error Correction Coding Used:	**None**	
System Allowed or Specified Bit-Error-Rate:	1.0E-04	
Demodulator Implementation Loss:	1.0	dB
Telemetry System Required Eb/No:	13.4	dB
Eb/No Threshold:	14.4	dB
System Link Margin:	**29.7**	dB

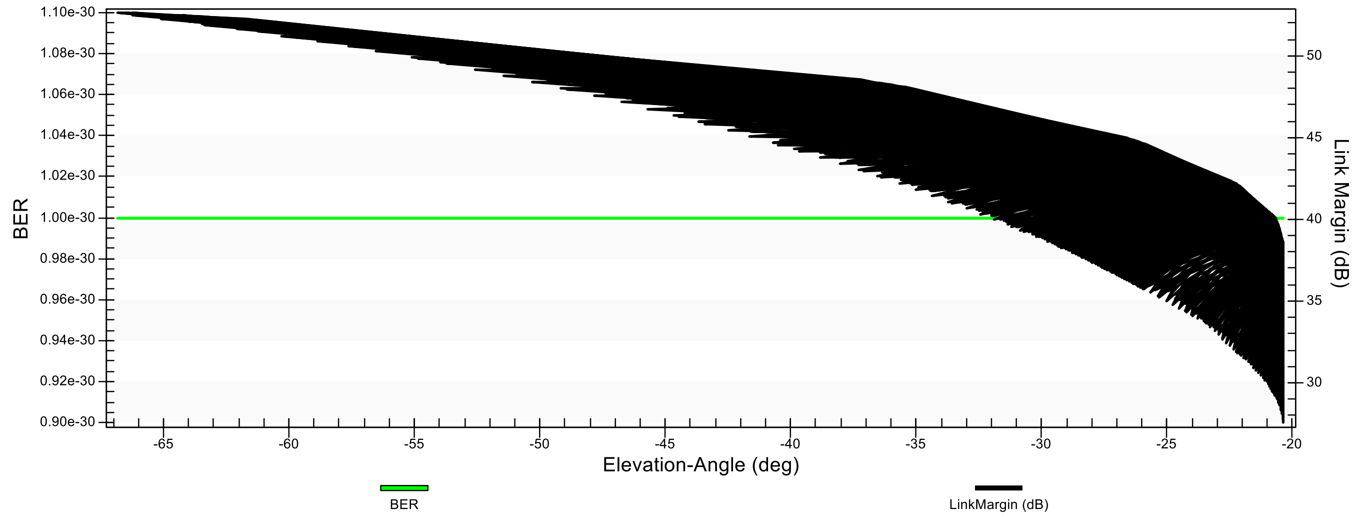
Ground Station (EbNo Method):		
------- Eb/No Method -------		
Ground Station Antenna Pointing Loss:	0.5	dB
Ground Station Antenna Gain:	18.5	dBi
Ground Station Total Transmission Line Losses:	3.8	dB
Ground Station Effective Noise Temperature:	688	K
Ground Station Figure of Merrit (G/T):	-13.7	dB/K
G.S. Signal-to-Noise Power Density (S/No):	60.0	dBHz
System Desired Data Rate:	9600	bps
In dBHz:	39.8	dBHz
Telemetry System Eb/No for the Downlink:	20.2	dB
Demodulation Method Seleted:	**Non-Coherent FSK**	
Forward Error Correction Coding Used:	**None**	
System Allowed or Specified Bit-Error-Rate:	1.0E-04	
Demodulator Implementation Loss:	0	dB
Telemetry System Required Eb/No:	13.4	dB
Eb/No Threshold:	13.4	dB
System Link Margin:	**6.8**	dB

APPENDIX D. 6 - Uplink Sun-Synchronous Orbit (Left) and Downlink Sun-Synchronous Orbit (Right)

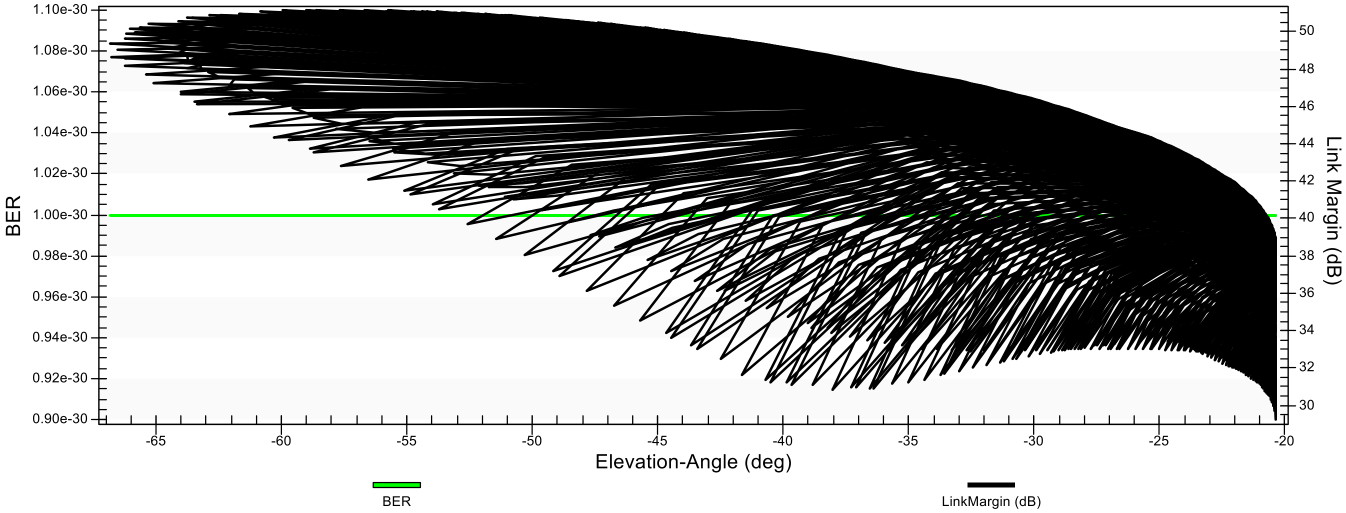
APPENDIX E



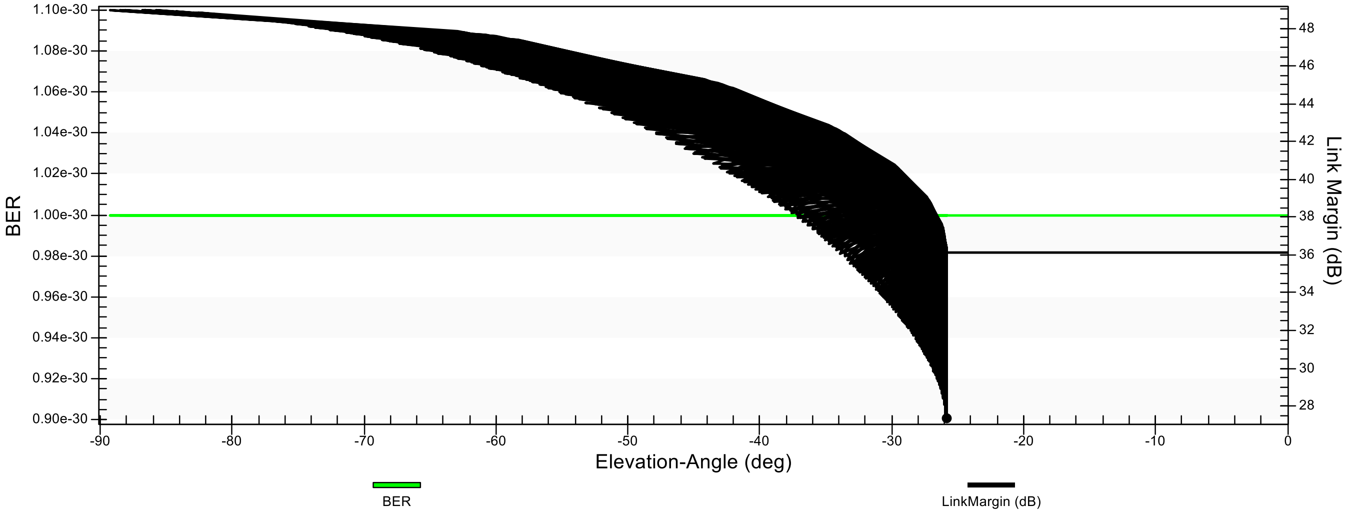
APPENDIX E. 1 - ISS Horizontal Transmitting Radiation Pattern Link Margin & Bit Error Rate vs. Elevation Angle



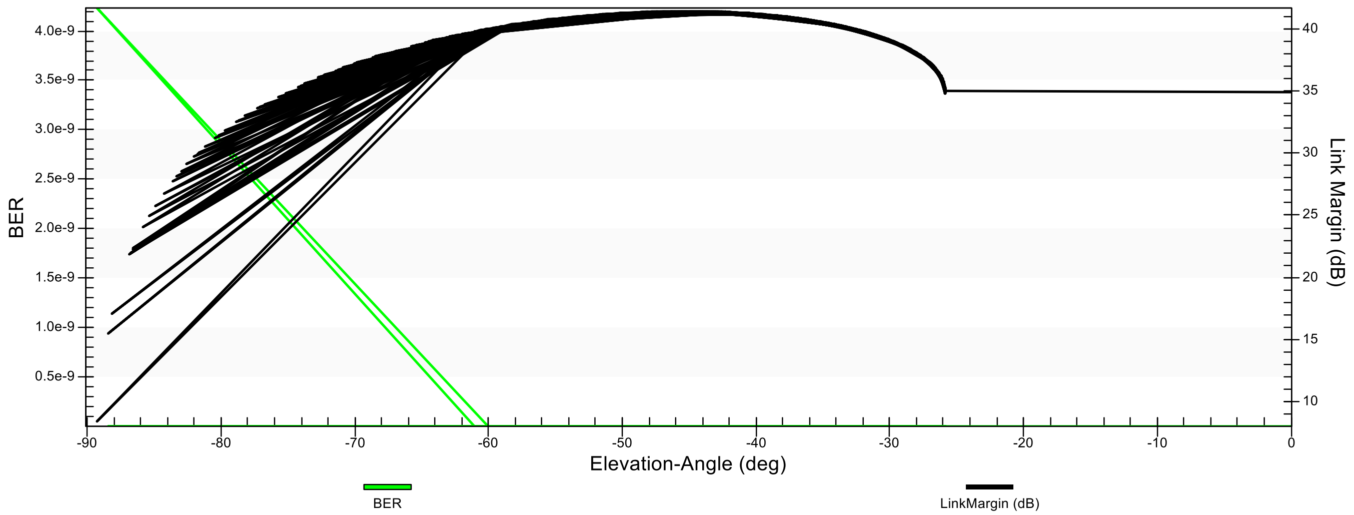
APPENDIX E. 2 - ISS Vertical Transmitting Radiation Pattern Link Margin & Bit Error Rate vs. Elevation Angle



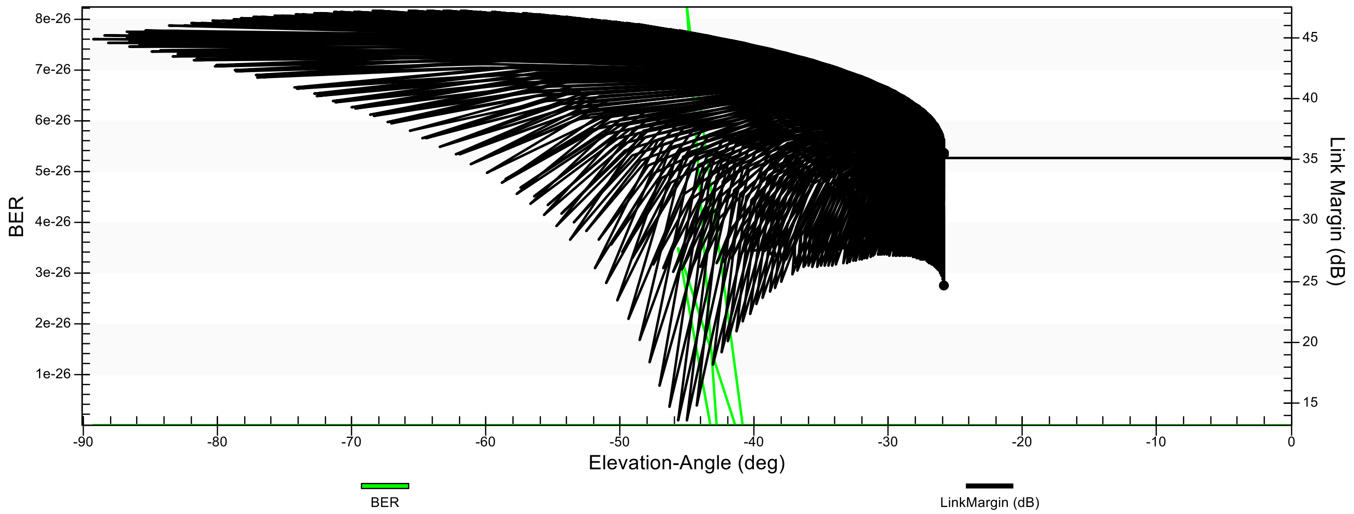
APPENDIX E. 3 - ISS 45 deg. Transmitting Radiation Pattern Link Margin & Bit Error Rate vs. Elevation Angle



APPENDIX E. 4 - Sun-Sync Horizontal transmitting Radiation Pattern Link Margin & BER vs. Elevation Angle



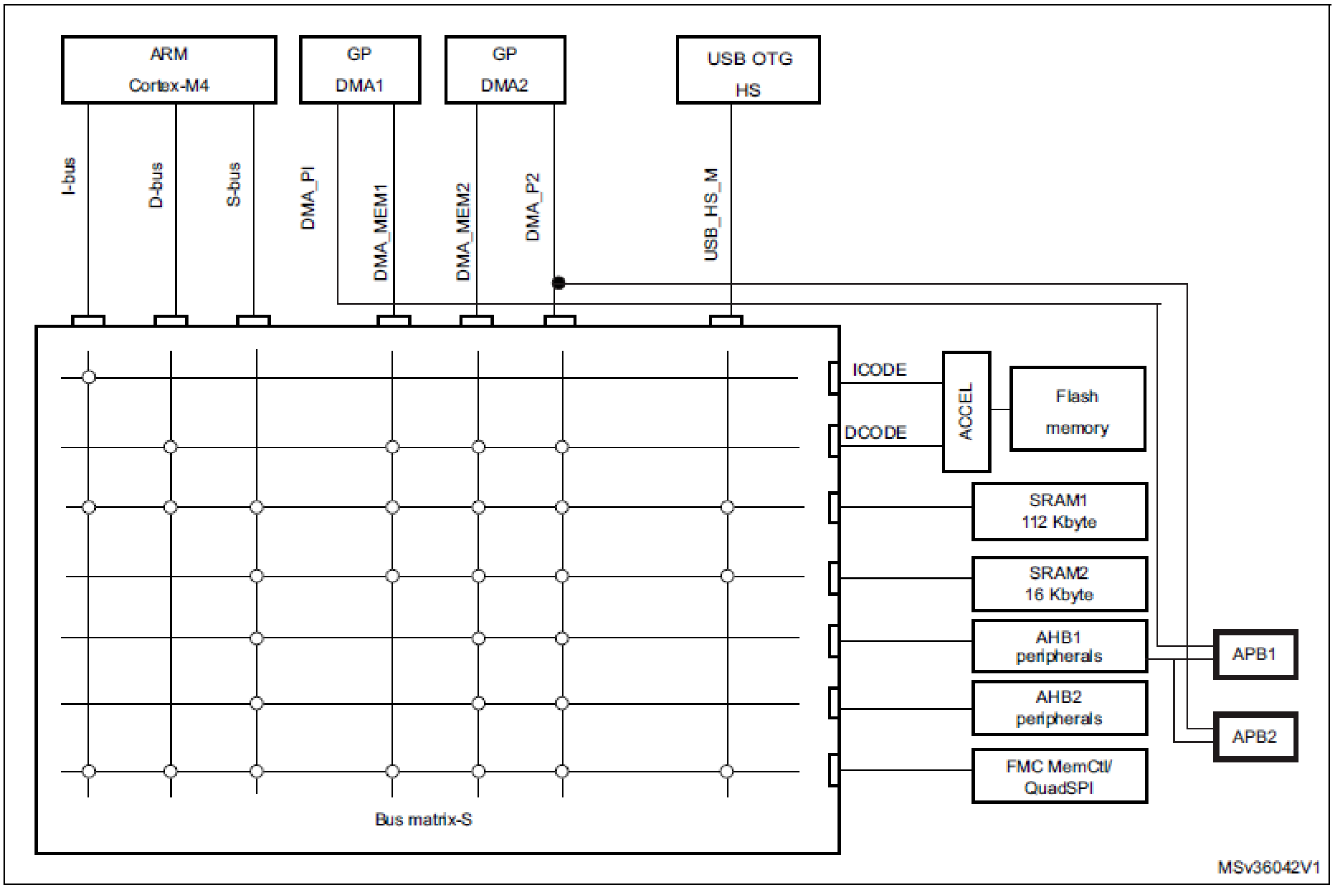
APPENDIX E. 5 - Sun-Sync Vertical transmitting Radiation Pattern Link Margin & BER vs. Elevation Angle



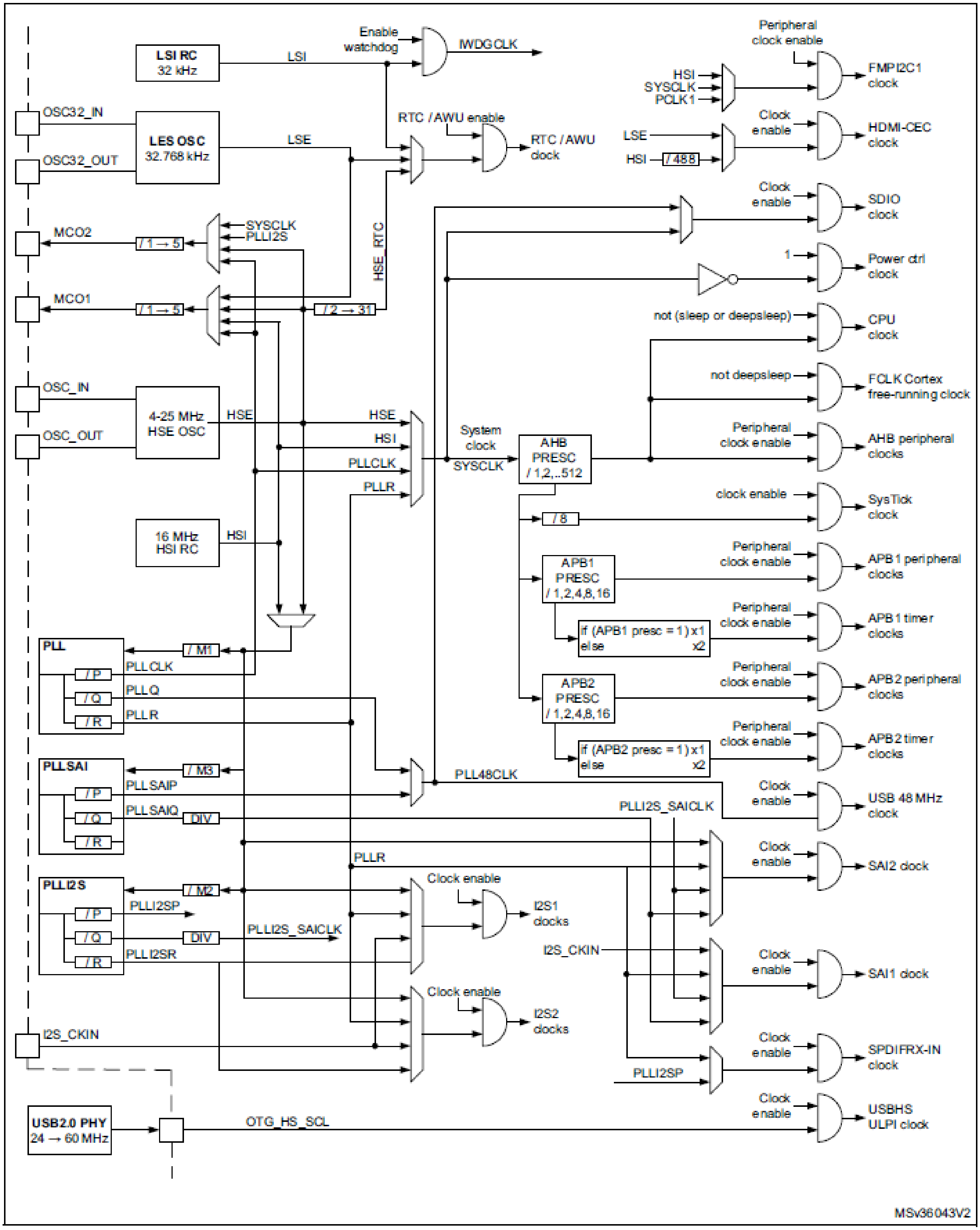
APPENDIX E. 6 - Sun-Sync 45 deg. transmitting Radiation Pattern Link Margin & BER vs. Elevation Angle

APPENDIX F

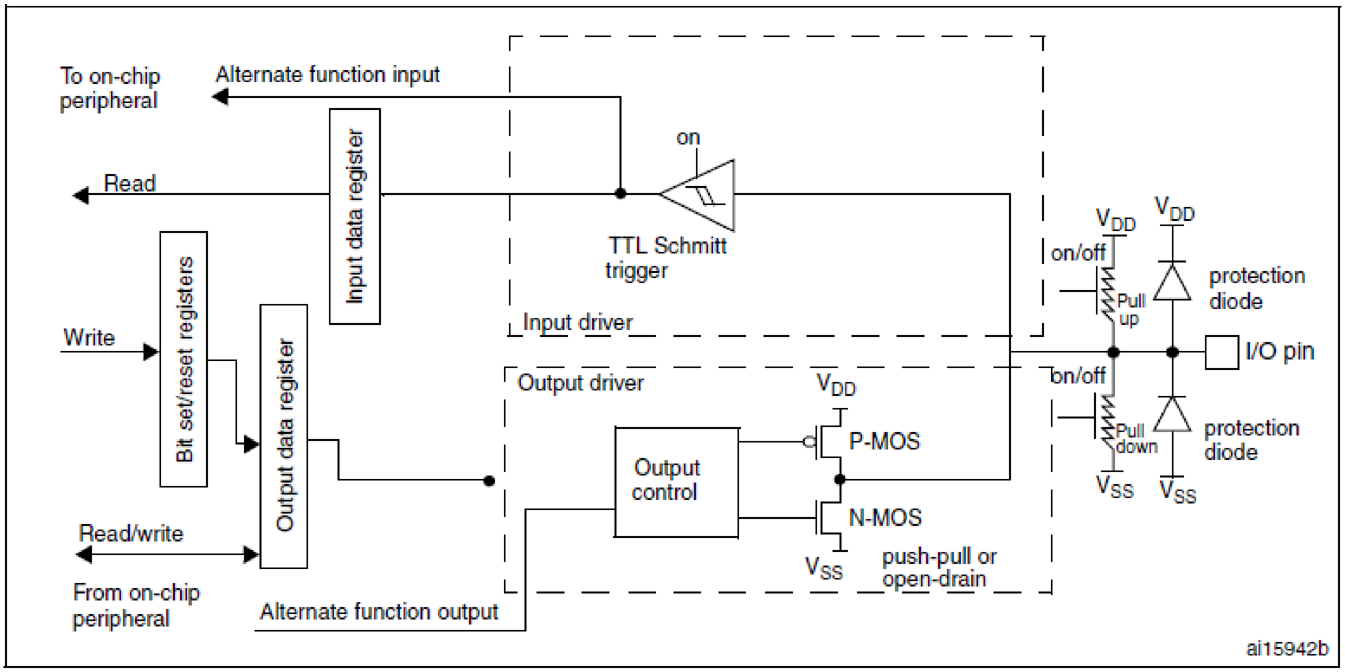
ADCS Controller – STM32 F446RE



Appendix F. 1 – STM32-F446RE Memory Busses



Appendix F. 2 – STM32 F446RE Clock Tree



Appendix F. 3 - GPIO Alternate Function Configuration

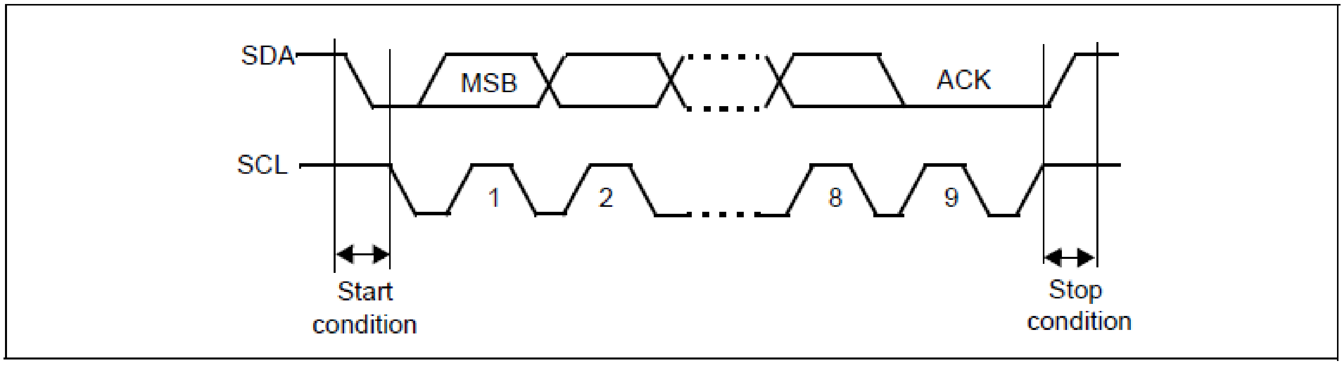
I2C pins are in open-drain configuration, this means the GPIO is set to alternate function mode:

$$Rp_{min} = \frac{V_{cc2} - V_{OL(MAX)}}{I_{OL}}$$

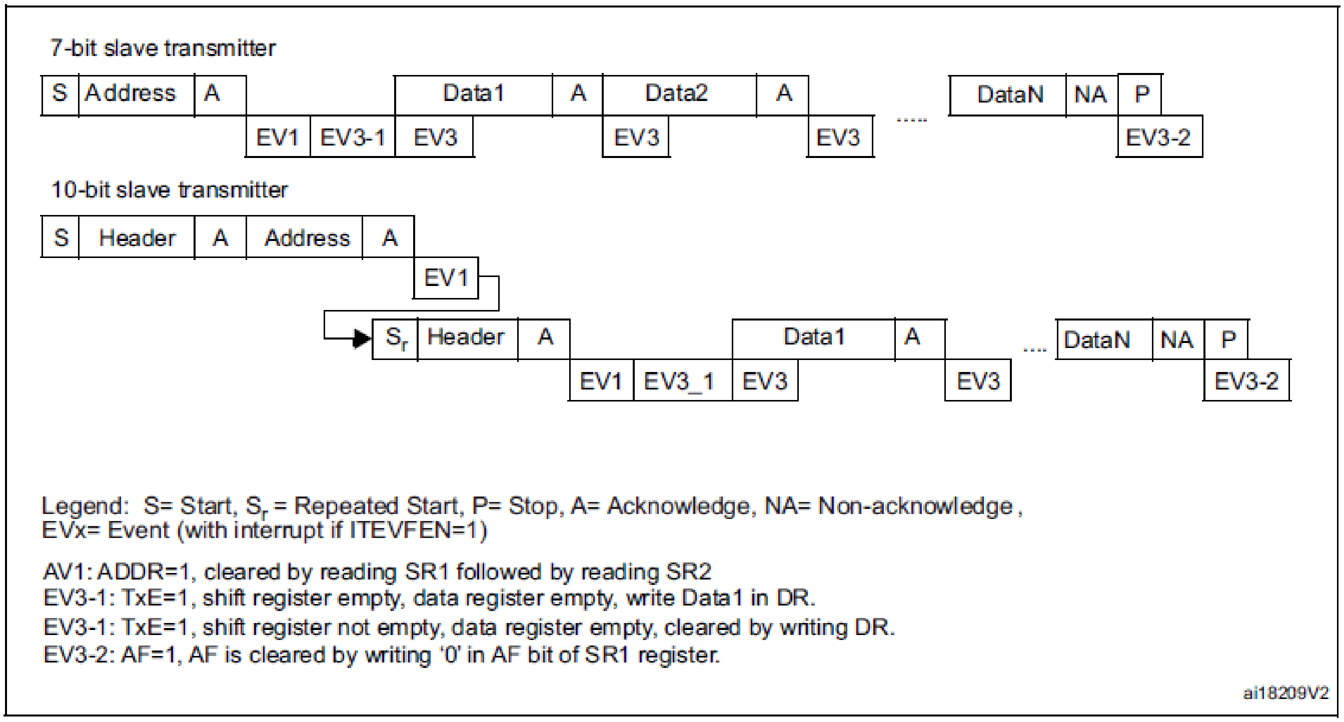
“OL” subscripts stand for low-level output for voltage (V) and current (I).

$V_{OL(MAX)}$ is 0.4V [17]

I_{OL} is 3mA [17]



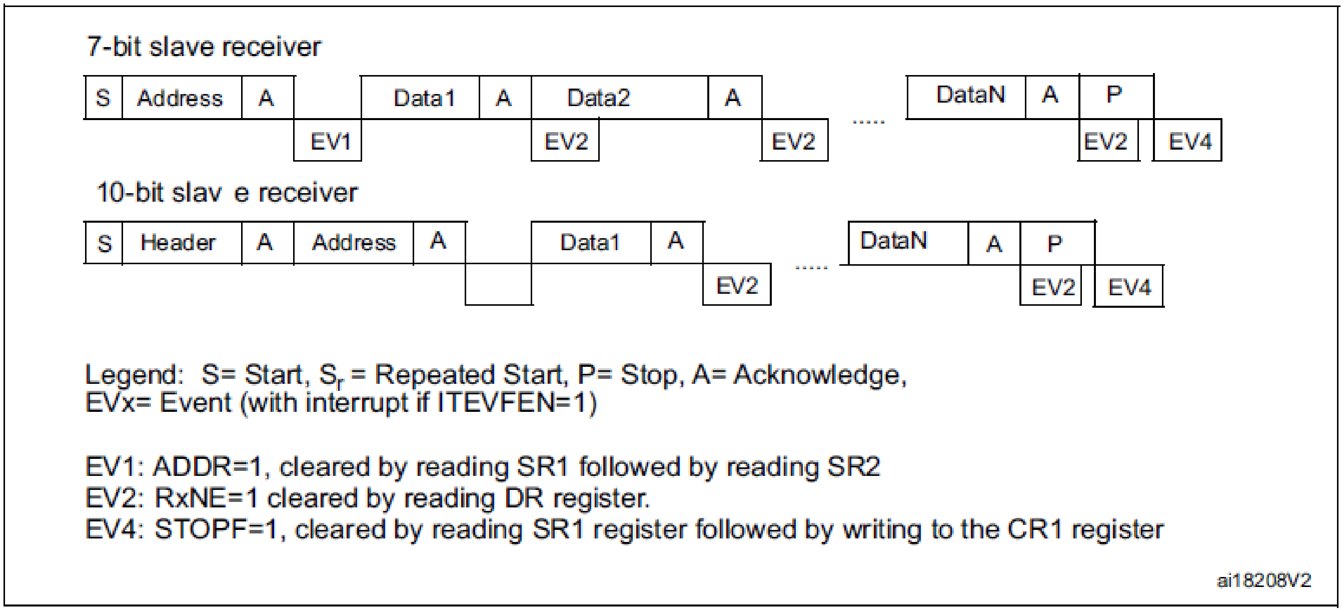
Appendix F. 4 - I2C 7-bit address with ACK timing diagram



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge,
EVx= Event (with interrupt if ITEVFEN=1)

AV1: ADDR=1, cleared by reading SR1 followed by reading SR2
EV3-1: TxE=1, shift register empty, data register empty, write Data1 in DR.
EV3-1: TxE=1, shift register not empty, data register empty, cleared by writing DR.
EV3-2: AF=1, AF is cleared by writing '0' in AF bit of SR1 register.

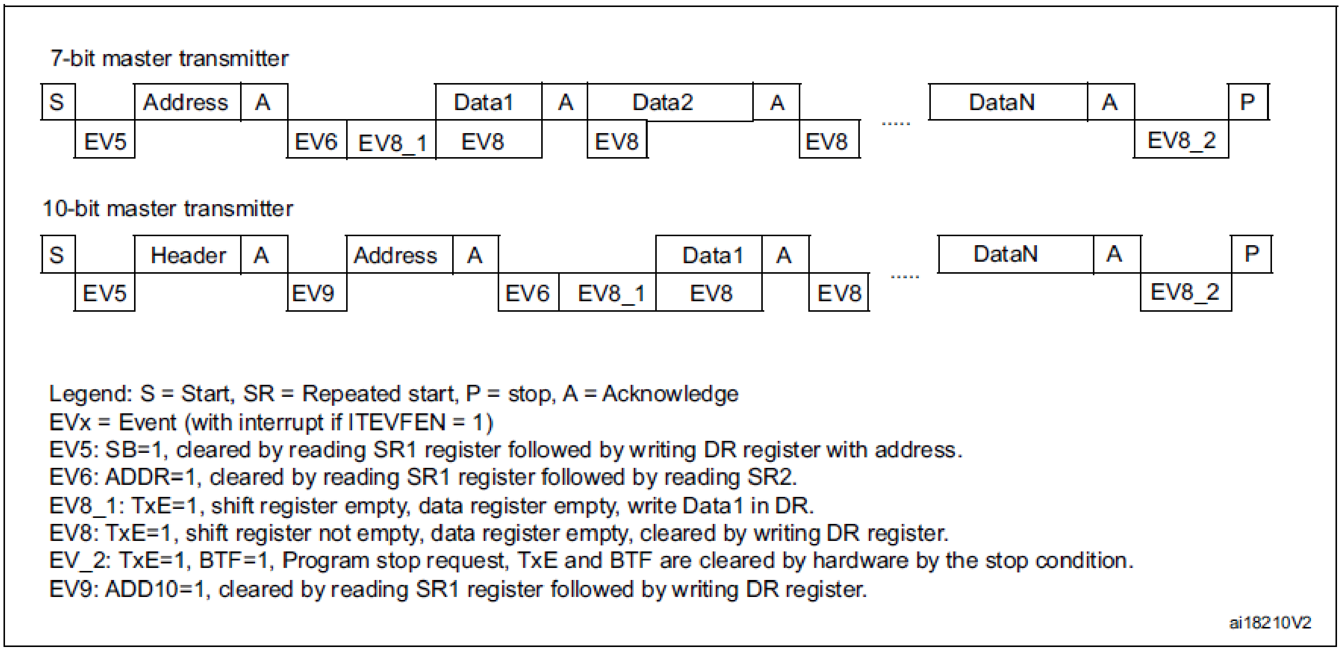
Appendix F. 5 - I2C 7-bit Slave Send Data



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge,
EVx= Event (with interrupt if ITEVFEN=1)

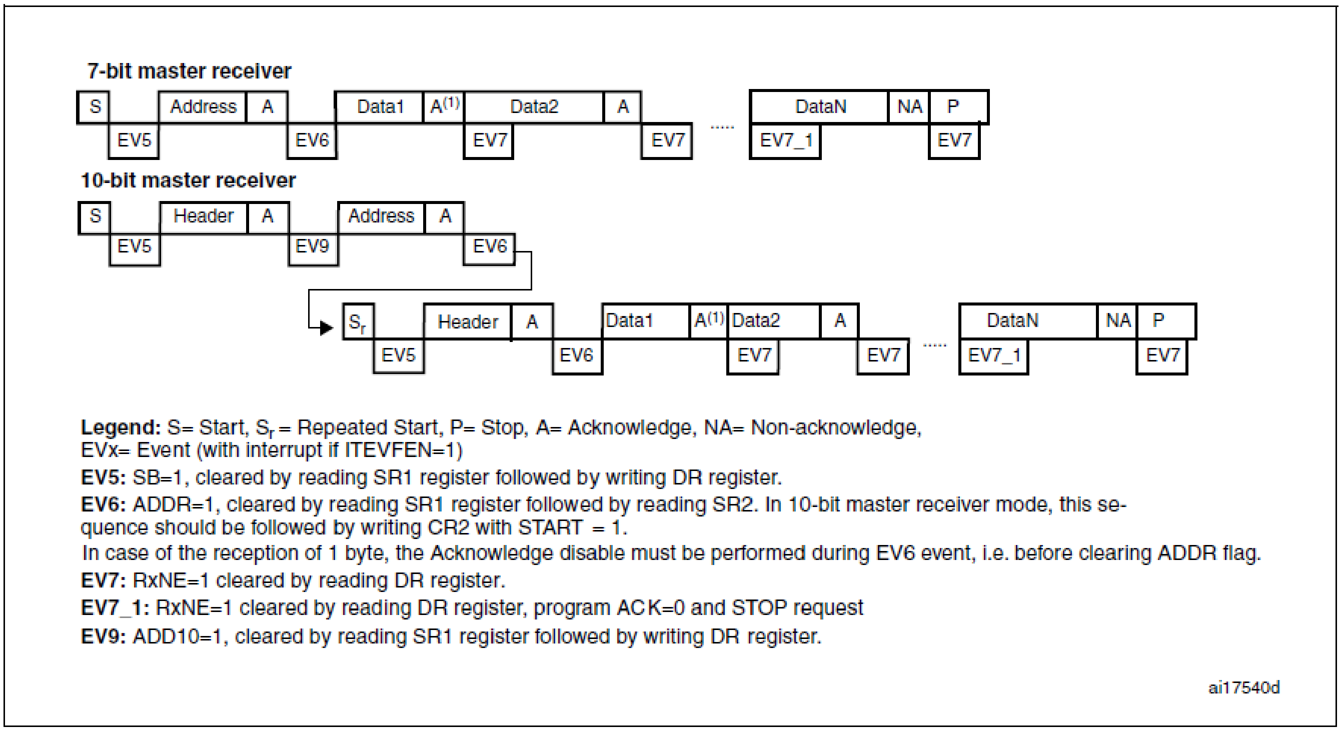
EV1: ADDR=1, cleared by reading SR1 followed by reading SR2
EV2: RxNE=1 cleared by reading DR register.
EV4: STOPF=1, cleared by reading SR1 register followed by writing to the CR1 register

Appendix F. 6 - I2C 7-bit Slave Receive Data



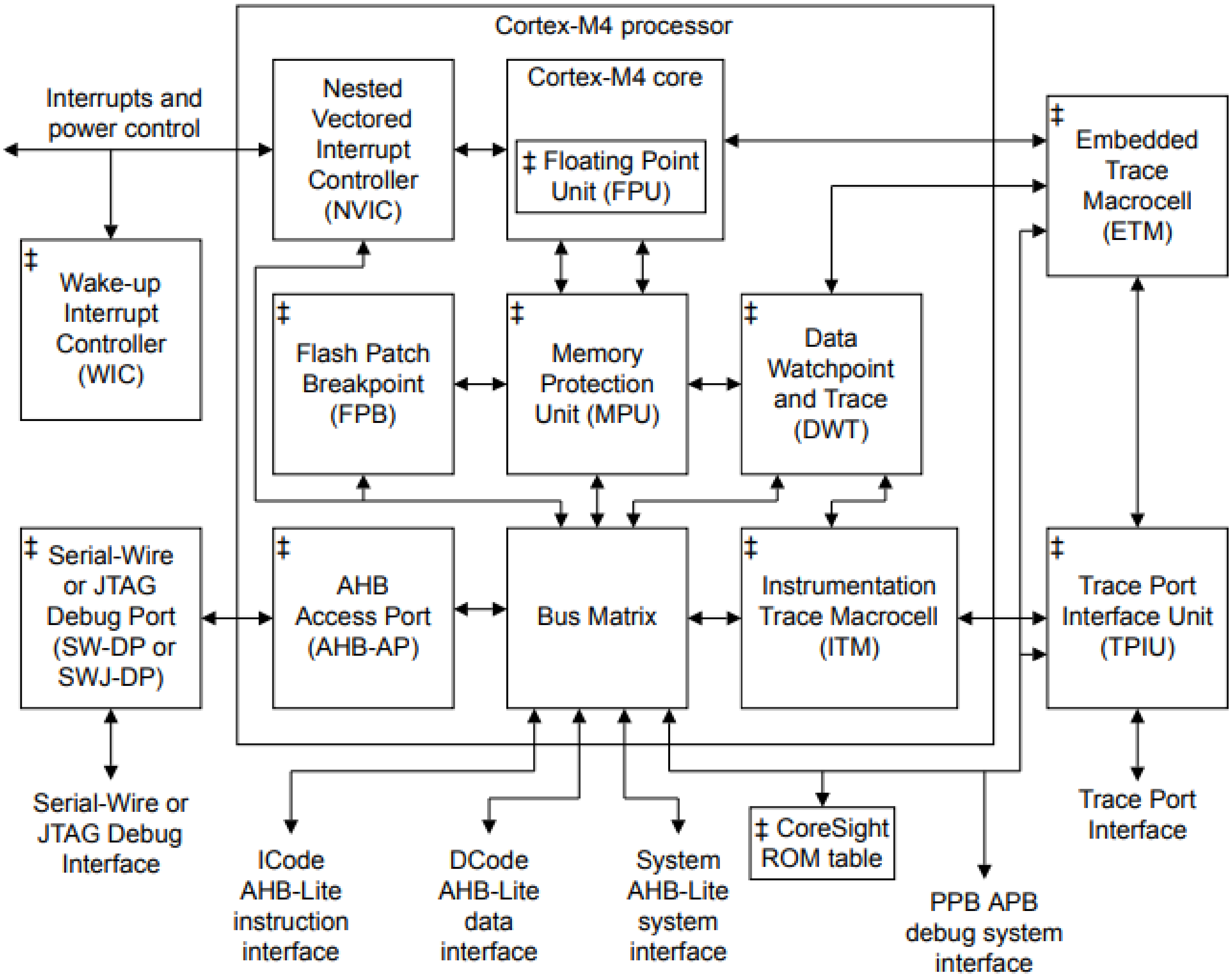
Legend: S = Start, SR = Repeated start, P = stop, A = Acknowledge
EVx = Event (with interrupt if ITEVFEN = 1)
EV5: SB=1, cleared by reading SR1 register followed by writing DR register with address.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2.
EV8_1: TxE=1, shift register empty, data register empty, write Data1 in DR.
EV8: TxE=1, shift register not empty, data register empty, cleared by writing DR register.
EV_2: TxE=1, BTF=1, Program stop request, TxE and BTF are cleared by hardware by the stop condition.
EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

Appendix F. 7 - I2C 7-bit Master Send Data



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge,
EVx= Event (with interrupt if ITEVFEN=1)
EV5: SB=1, cleared by reading SR1 register followed by writing DR register.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.
In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.
EV7: RxNE=1 cleared by reading DR register.
EV7_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request
EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

Appendix F. 8 - I2C 7-bit Master Receive Data


Cortex-M4 processor
Interrupts and power control
Nested Vectored Interrupt Controller (NVIC)
Cortex-M4 core
‡ Floating Point Unit (FPU)
‡ Embedded Trace Macrocell (ETM)
‡ Wake-up Interrupt Controller (WIC)
‡ Flash Patch Breakpoint (FPB)
‡ Memory Protection Unit (MPU)
‡ Data Watchpoint and Trace (DWT)
‡ Serial-Wire or JTAG Debug Port (SW-DP or SWJ-DP)
‡ AHB Access Port (AHB-AP)
Bus Matrix
‡ Instrumentation Trace Macrocell (ITM)
‡ Trace Port Interface Unit (TPIU)
Serial-Wire or JTAG Debug Interface
ICode AHB-Lite instruction interface
DCode AHB-Lite data interface
System AHB-Lite system interface
‡ CoreSight ROM table
PPB APB debug system interface
Trace Port Interface

Appendix F. 9 - ARM CORTEX M4 Block Diagram

APPENDIX G

ADCS_comms/drivers/Inc/ mcu.h

```
/*
 * mcu.h
 *
 * MCU specific header file for STM32-F446RE
 *
 * Refer to reference manual to see memory, register maps, and control bits:
 *     RM0390 Reference Manual - STM32F446xx advanced ARM-based 32-bit MCUs
 *

 */

#ifndef DRIVERS_INC_MCU_H_
#define DRIVERS_INC_MCU_H_

#include <stdint.h>

/************* General Macros **************/
#define TRUE  1
#define FALSE 0

/************* AHB/APB Bridges *************/

/* base address of APB1 (Advanced Peripheral Bus)
* that contains the CPU's I2C registers in it's memory map
```

```
*/
#define APB1 0x40000000U

/* base address of AHB1 (Advanced High-performance Bus)
 * that contains the CPU's RCC registers in it's memory map
 */
#define AHB1 0x40020000U
/*******************************************/

/************** Memory Maps ***************/

/* Base address of RCC (Reset and Clock Control)
* control registers in the CPU memory map
*/
#define RCC_ADDR (AHB1 + 0x3800U)

/* Base address of GPIO ports on the AHB1 bus
 * that can be configures as I2C pins
 */
#define GPIOA_ADDR (AHB1 + 0x0000)
#define GPIOB_ADDR (AHB1 + 0x0400)

/* Base addresses of I2C (Inter-Integrated Circuit)
 * control registers in the CPU memory
 */
#define I2C1_ADDR (APB1 + 0x5400U)
#define I2C2_ADDR (APB1 + 0x5800U)
```

```
#define I2C3_ADDR (APB1 + 0x5C00U)
/*********************************************/


/************** Register Maps ****************/


// RCC register map
typedef struct {
        volatile uint32_t RCC_CR;
        volatile uint32_t RCC_PLLCFGR;
        volatile uint32_t RCC_CFGR;
        volatile uint32_t RCC_CIR;
        volatile uint32_t RCC_AHB1RSTR;
        volatile uint32_t RCC_AHB2RSTR;
        volatile uint32_t RCC_AHB3RSTR;
        uint32_t Reserved_1C;
        volatile uint32_t RCC_APB1RSTR;
        volatile uint32_t RCC_APB2RSTR;
        uint32_t Reserved_28;
        uint32_t Reserved_2C;
        volatile uint32_t RCC_AHB1ENR;
        volatile uint32_t RCC_AHB2ENR;
        volatile uint32_t RCC_AHB3ENR;
        volatile uint32_t RCC_APB1ENR; // use
        volatile uint32_t RCC_APB2ENR;
        uint32_t Reserved_48;
        uint32_t Reserved_4C;
        volatile uint32_t RCC_AHB1LPENR;
        volatile uint32_t RCC_AHB2LPENR;
        volatile uint32_t RCC_AHB3LPENR;
        uint32_t Reserved_5C;
        volatile uint32_t RCC_APB1LPER;
        volatile uint32_t RCC_APB2LPENR;
        uint32_t Reserved_68;
        uint32_t Reserved_6C;
        volatile uint32_t RCC_BDCR;
        volatile uint32_t RCC_CSR;
        uint32_t Reserved_78;
        uint32_t Reserved_7C;
        volatile uint32_t RCC_SSCGR;
        volatile uint32_t RCC_PLLI2SCFGR;
        volatile uint32_t RCC_PLLSAICFGR;
        volatile uint32_t RCC_DCKCFGR;
        volatile uint32_t RCC_CKGATENR;
        volatile uint32_t RCC_DCKCFGR2;
}RCC_regs_t;

// GPIO register map
typedef struct {
        volatile uint32_t GPIO_MODER; // port mode
        volatile uint32_t GPIO_OTYPER; // port output type
        volatile uint32_t GPIO_OSPEEDR; // port output speed
        volatile uint32_t GPIO_PUPDR; // port pull-up/pull-down
        volatile uint32_t GPIO_IDR; // port input data
        volatile uint32_t GPIO_ODR; // port output data
        volatile uint32_t GPIO_BSRR; // port bit set/reset
        volatile uint32_t GPIO_LCKR; // port configuration lock
        volatile uint32_t GPIO_AFRL; // alternate function low
        volatile uint32_t GPIO_AFRH; // alternate function high
}GPIO_regs_t;

// I2C register map
typedef struct {
        volatile uint32_t CR1; // use
        volatile uint32_t CR2; // use
        volatile uint32_t OAR1;
        volatile uint32_t OAR2;
        volatile uint32_t DR;
        volatile uint32_t SR1; // use
        volatile uint32_t SR2; // use
        volatile uint32_t CCR;  // use
        volatile uint32_t TRISE;
        volatile uint32_t FLTR;
}I2C_regs_t;

/* Register map pointers to register
 * map structures in memory
 */
#define RCC   ((RCC_regs_t*)RCC_ADDR)
#define GPIOA ((GPIO_regs_t*)GPIOA_ADDR)
#define GPIOB ((GPIO_regs_t*)GPIOB_ADDR)
#define I2C1  ((I2C_regs_t*)I2C1_ADDR)
```

```
#define I2C2  ((I2C_regs_t*)I2C2_ADDR)
#define I2C3  ((I2C_regs_t*)I2C3_ADDR)
/*********************************************/

/******** I2C registers bit positions ********/

// I2C_CR1 (control register 1) bit positions
#define I2C_CR1_PE        0
#define I2C_CR1_NOSTRETCH 7
#define I2C_CR1_START     8
#define I2C_CR1_STOP      9
#define I2C_CR1_ACK       10
#define I2C_CR1_SWRST     15

// I2C_CR2 bit position
#define I2C_CR2_FREQ    0
#define I2C_CR2_ITERREN 8
#define I2C_CR2_ITEVTEN 9
#define I2C_CR2_ITBUFEN 10

// I2C_OAR1 bit position
#define I2C_OAR1_RESERVED    14
#define I2C_OAR1_OFFSET_ADD0 1

// I2C_SR1 (status register) bit position
#define I2C_SR1_SB      0
#define I2C_SR1_ADDR    1
```

```
#define I2C_SR1_BTF     2
#define I2C_SR1_ADD10   3
#define I2C_SR1_STOPF   4
#define I2C_SR1_RXNE    6
#define I2C_SR1_TXE     7
#define I2C_SR1_BERR    8
#define I2C_SR1_ARLO    9
#define I2C_SR1_AF      10
#define I2C_SR1_OVR     11
#define I2C_SR1_TIMEOUT 14

// I2C_SR2 bit positions
#define I2C_SR2_MSL     0
#define I2C_SR2_BUSY    1
#define I2C_SR2_TRA     2
#define I2C_SR2_GENCALL 4
#define I2C_SR2_DUALF   7

// I2C_CCR (clock control register) bit positions
#define I2C_CCR_CCR  0
#define I2C_CCR_DUTY 14
#define I2C_CCR_FS   15
/*********************************************/

#endif /* DRIVERS_INC_MCU_H_ */
```

ADCS_comms/drivers/Inc/rcc.h

```c
/*
 * rcc.h
 *
 */

#ifndef DRIVERS_INC_RCC_H_
#define DRIVERS_INC_RCC_H_

#include "mcu.h"

/*
 * Peripheral Clock enable for I2C peripheral on APB1 bus
 *  - APB1 is default 16MHz here as we use the HSI (High Speed Internal)
 *    clock, that is the RC (resister capacitor) circuit to generate square wave
 *
 *    This is done in macro functions so we don't pass RCC
 *    memory map through user space
 */
#define GPIOA_CLK_ENABLE() (RCC->RCC_AHB1ENR |= (1 << 0)) // set GPIOAEN bit
#define GPIOB_CLK_ENABLE() (RCC->RCC_AHB1ENR |= (1 << 1)) // set GPIOBEN bit

#define I2C1_CLK_ENABLE() (RCC->RCC_APB1ENR |= (1 << 21)) // set I2C1EN bit
#define I2C2_CLK_ENABLE() (RCC->RCC_APB1ENR |= (1 << 22)) // set I2C2EN bit
#define I2C3_CLK_ENABLE() (RCC->RCC_APB1ENR |= (1 << 23)) // set I2C3EN bit

uint32_t RCC_PCLK1_get(void);
```

```
#endif /* DRIVERS_INC_RCC_H_ */
```

ADCS_comms/drivers/Inc/i2c.h

```
/*
          * i2c.h
          *
          *       i2c driver header file
```

```
 *
 *      Author: Adam Al-Khazraji
 *
 */

#ifndef DRIVERS_INC_I2C_H_
#define DRIVERS_INC_I2C_H_

#include "mcu.h"

typedef struct {
       uint32_t I2C_SCL; // SCL speed or frequency
       uint8_t I2C_DeviceAddress; // device address on I2C bus
       uint8_t I2C_ACK; // ACK after every message
       uint16_t I2C_FM; // Duty Cycle when FM in (Fast Mode)
}I2C_config_t;

typedef struct {
       I2C_regs_t* i2c_regs; // i2c register structure
       I2C_config_t config; // options for i2c comm
}I2C_control_t;

#define SCL_DEFAULT 100000 // SCL default to 100KHz
#define SCL_FMPI2C  400000 // Fast Mode Plus I2C is between 100KHz to 400KHz

// ACK control bit is bit 10 of I2C CR1 register
#define I2C_ACK_ENABLE  1
#define I2C_ACK_DISABLE 0 // default

// DUTY control bit is bit 14 of I2C CCR register
#define FMPI2C_DUTY_CYCLE_2     0
#define FMPI2C_DUTY_CYCLE_16_9  1

/* I2C_SR1 flags*/
#define I2C_SR1_FLAG_SB      (1 << I2C_SR1_SB)
#define I2C_SR1_FLAG_ADDR    (1 << I2C_SR1_ADDR)
#define I2C_SR1_FLAG_BTF     (1 << I2C_SR1_BTF)
#define I2C_SR1_FLAG_ADD10   (1 << I2C_SR1_ADD10)
#define I2C_SR1_FLAG_STOPF   (1 << I2C_SR1_STOPF)
#define I2C_SR1_FLAG_RXNE    (1 << I2C_SR1_RXNE)
#define I2C_SR1_FLAG_TXE     (1 << I2C_SR1_TXE)
#define I2C_SR1_FLAG_BERR    (1 << I2C_SR1_BERR)
#define I2C_SR1_FLAG_ARLO    (1 << I2C_SR1_ARLO)
#define I2C_SR1_FLAG_AF      (1 << I2C_SR1_AF)
```

```
#define I2C_SR1_FLAG_OVR     (1 << I2C_SR1_OVR)
#define I2C_SR1_FLAG_TIMEOUT (1 << I2C_SR1_TIMEOUT)

// I2C peripheral clock setup
void I2C_CLK_Enable(I2C_regs_t *i2c_regs, uint8_t enable);

void I2C_Init(I2C_control_t *i2c_control);
void I2C_Close(I2C_regs_t *i2c_regs);

// IRQ - interrupt requests
void I2C_IRQ_Config(uint8_t IRQ, uint8_t enable);
void I2C_IRQ_Priority(uint8_t IRQ, uint32_t priority);

void I2C_Enable_Disable(I2C_regs_t *i2c_regs, uint8_t enable);
uint8_t I2C_GetStatus(I2C_regs_t *i2c_regs, uint32_t flag);

void I2C_Callback(I2C_control_t *i2c_control, uint8_t app_event);

void I2C_MasterSend(I2C_control_t *i2c_control, uint8_t *tx_buf, uint32_t len, uint8_t
slave_addr);

#endif /* DRIVERS_INC_I2C_H_ */
```

ADCS_comms/drivers/Inc/gpio.h

```
/*
 * gpio.h
 *
 */

#ifndef DRIVERS_INC_GPIO_H_
#define DRIVERS_INC_GPIO_H_

#include "mcu.h"

typedef struct {
	uint8_t GPIO_Pin; // pin number
	uint8_t GPIO_Mode; // can be both interrupt or output based
	uint8_t GPIO_Speed;
	uint8_t GPIO_PUPD; // pill up pull down
	uint8_t GPIO_Output;
	uint8_t GPIO_AltFunc;
}GPIO_config_t;

typedef struct {
        GPIO_regs_t* gpio_regs;
        GPIO_config_t config;
}GPIO_control_t;

// GPIO pins
#define GPIO_PIN_0    0
#define GPIO_PIN_1    1
#define GPIO_PIN_2    2
#define GPIO_PIN_3    3
#define GPIO_PIN_4    4
#define GPIO_PIN_5    5
#define GPIO_PIN_6    6
#define GPIO_PIN_7  7
#define GPIO_PIN_8  8
#define GPIO_PIN_9    9
#define GPIO_PIN_10   10
#define GPIO_PIN_11   11
#define GPIO_PIN_12   12
#define GPIO_PIN_13 13
#define GPIO_PIN_14 14
#define GPIO_PIN_15 15

// pin modes
// two lsb of GPIO_MODER (port mode register)
#define GPIO_MODE_INPUT      0
#define GPIO_MODE_OUTPUT     1
#define GPIO_MODE_ALTFUNC    2
#define GPIO_MODE_ANALOG     3

// pin interrupt modes
#define GPIO_MODE_IT_FT     4 // falling edge
#define GPIO_MODE_IT_RT     5 // rising
#define GPIO_MODE_IT_RFT    6 // rising falling edge

// output type
// GPIO_OTYPER (port output type register) bits 15:0
// look at GPIO circuit in reference manual for more information
#define GPIO_OUTPUT_PP   0 // push-pull register
#define GPIO_OUTPUT_OD   1 // open drain

// speed
// GPIO_OSPEEDR (port output speed register) bits 1:0
#define GPIO_SPEED_LOW    0
#define GPIO_SPEED_MEDIUM 1
#define GPIO_SPEED_FAST   2
#define GPOI_SPEED_HIGH   3

// pull-up/pull-down resistor
/*
 * we use the MCU internal PuPd resistor for this project
 * Check reference manual for external resistor calculations
 * that would be needed when more devices are connected to the
 * peripheral bus
 */
// GPIO_PUPDR (port pull-up/pull-down register) bits 1:0
#define GPIO_NO_PUPD 0
#define GPIO_PIN_PU  1
#define GPIO_PIN_PD  2
// 3 is reserved in the reference manual

// alternate function
// use these macros for the high and low registers
// GPIO_AFHR GPIO_AFLR
#define GPIO_AF0  0
#define GPIO_AF1  1
#define GPIO_AF2  2
#define GPIO_AF3  3
#define GPIO_AF4  4
#define GPIO_AF5  5
#define GPIO_AF6  6
#define GPIO_AF7  7
#define GPIO_AF8  8
#define GPIO_AF9  9
#define GPIO_AF10 10
#define GPIO_AF11 11
#define GPIO_AF12 12
#define GPIO_AF13 13
#define GPIO_AF14 14
#define GPIO_AF15 15

/* Only use init as we are using the GPIO ports
 * to configure them as I2C pins
 */
void GPIO_ClkEnable(GPIO_regs_t* gpio_regs, uint8_t enable);
void GPIO_Init(GPIO_control_t* gpio);
```

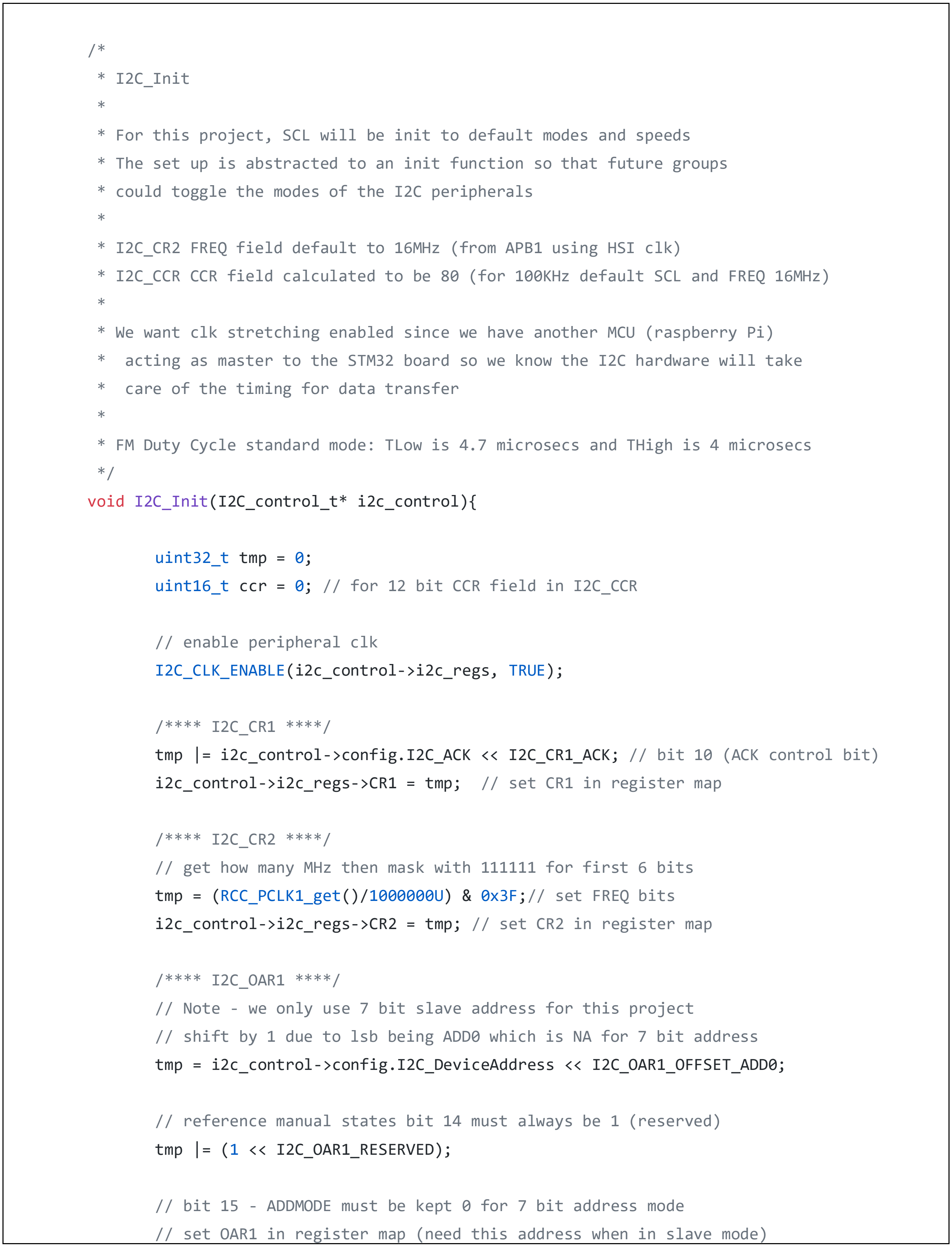
```
#endif /* DRIVERS_INC_GPIO_H_ */
```

ADCS_comms/drivers/Src/rcc.c

```
/*
 * rcc.c
 *
 *      Author: Adam
 */


#include "../Inc/rcc.h"

/*
 * RCC_PCLK1_get
 * return the clk speed used for APB1 in this case
 *
 * This should be abstracted to some RCC.c file for other peripherals to use
 *
 * Refer to RCC_CFGR (clock configuration register)
 *        SWS (system clock switch status) - bits 2 and 3
 *        HPRE (AHP Prescaler) - bits 4 to 7
 *        PPRE1 (APB1 prescaler) - bits 10 to 12
 */
uint32_t RCC_PCLK1_get(void){
        uint32_t pclk1, freq, ahb_clk_div, apb1_clk_div;
        uint8_t sws, hpre, ppre1;

        /*
         * sws:   bits 2 to 3   r-shift 2  then mask with 00000011
         * ppre1: bits 10 to 12 r-shift 10 then mask with 00000111
         * hre:   bits 4 to 7   r-shift 4  then mask with 00001111
         */
        sws   = (RCC->RCC_CFGR >> 2)  & 0x3;
        ppre1 = (RCC->RCC_CFGR >> 10) & 0x7;
        hpre  = (RCC->RCC_CFGR >> 4)  & 0xF;

        if (sws == 0) // HSI
                freq = 1600000;
        else if (sws == 1) // HSE
                // HSE (High Speed External) uses X2 crystal oscillator on evaluation board
                freq = 800000;
        // PLL is option 2, NA for our project
    // option 3 is NA in reference manual
        else return 0; // error

        // AHB clk div
        if (hpre < 8) ahb_clk_div = 1; // no clk divider
        else if (hpre == 8) ahb_clk_div = 2;
        else if (hpre == 9) ahb_clk_div = 4;
        else if (hpre == 10) ahb_clk_div = 8;
        else if (hpre == 11) ahb_clk_div = 16;
        else if (hpre == 12) ahb_clk_div = 32;
        else if (hpre == 13) ahb_clk_div = 64;
        else if (hpre == 14) ahb_clk_div = 128;
        else if (hpre == 15) ahb_clk_div = 512;
        else return 0; // error

        if (ppre1 < 4) apb1_clk_div = 1; // no clk divider
        else if (ppre1 == 4) apb1_clk_div = 2;
        else if (ppre1 == 5) apb1_clk_div = 4;
        else if (ppre1 == 6) apb1_clk_div = 8;
        else if (ppre1 == 7) apb1_clk_div = 16;
        else return 0; // error

        pclk1 = (freq / ahb_clk_div) / apb1_clk_div;
        return pclk1;
}
```

ADCS_comms/drivers/Src/i2c.c

```
/*
 * i2c.c
 *
 *   i2c driver source code
 *
 *      Author: Adam Al-Khazraji
 */

#include "../Inc/i2c.h"
#include "../Inc/rcc.h"

/******* local function declarations *******/
static void I2C_Start(I2C_regs_t* i2c_regs);
static void I2C_Stop(I2C_regs_t* i2c_regs);
static void I2C_SendAddr(I2C_regs_t* i2c_regs, uint8_t slave_addr);
static void I2C_ClearADDRFlag(I2C_regs_t* i2c_regs);

/*
 * I2C_Enable_Disable
 * toggle peripheral enable bit I2C_CR1_PE (bit 0) of I2C_CR1 (control reg)
 */
void I2C_Enable_Disable(I2C_regs_t* i2c_regs, uint8_t enable)
{
        if (enable == TRUE)
                i2c_regs->CR1 |= (1 << I2C_CR1_PE);
        else
                i2c_regs->CR1 &= ~(1 << I2C_CR1_PE);
}

/*
 * I2C_CLK_ENABLE
 *
 * set I2CEN bit for the I2C peripheral
 *  - for I2C1, I2C2, and I2C3: RCC_APB1ENR bit at 21, 22, and 23 respectively
 */
void I2C_CLK_ENABLE(I2C_regs_t* i2c_regs, uint8_t enable)
{
        if (enable == TRUE){
                if (i2c_regs == I2C1)
                        I2C1_CLK_ENABLE();
                else if (i2c_regs == I2C2)
                        I2C2_CLK_ENABLE();
                else if (i2c_regs == I2C3)
                        I2C3_CLK_ENABLE();
                else
                        return;
        }
        else
                return;
}

uint8_t I2C_GetStatus(I2C_regs_t* i2c_regs, uint32_t flag)
{
        if(i2c_regs->SR1 & flag)
                return TRUE;
        else return FALSE;
}
```



```c
/*
 * I2C_Init
 *
 * For this project, SCL will be init to default modes and speeds
 * The set up is abstracted to an init function so that future groups
 * could toggle the modes of the I2C peripherals
 *
 * I2C_CR2 FREQ field default to 16MHz (from APB1 using HSI clk)
 * I2C_CCR CCR field calculated to be 80 (for 100KHz default SCL and FREQ 16MHz)
 *
 * We want clk stretching enabled since we have another MCU (raspberry Pi)
 *  acting as master to the STM32 board so we know the I2C hardware will take
 *  care of the timing for data transfer
 *
 * FM Duty Cycle standard mode: TLow is 4.7 microsecs and THigh is 4 microsecs
 */
void I2C_Init(I2C_control_t* i2c_control){

        uint32_t tmp = 0;
        uint16_t ccr = 0; // for 12 bit CCR field in I2C_CCR

        // enable peripheral clk
        I2C_CLK_ENABLE(i2c_control->i2c_regs, TRUE);

        /**** I2C_CR1 ****/
        tmp |= i2c_control->config.I2C_ACK << I2C_CR1_ACK; // bit 10 (ACK control bit)
        i2c_control->i2c_regs->CR1 = tmp;  // set CR1 in register map

        /**** I2C_CR2 ****/
        // get how many MHz then mask with 111111 for first 6 bits
        tmp = (RCC_PCLK1_get()/1000000U) & 0x3F;// set FREQ bits
        i2c_control->i2c_regs->CR2 = tmp; // set CR2 in register map

        /**** I2C_OAR1 ****/
        // Note - we only use 7 bit slave address for this project
        // shift by 1 due to lsb being ADD0 which is NA for 7 bit address
        tmp = i2c_control->config.I2C_DeviceAddress << I2C_OAR1_OFFSET_ADD0;

        // reference manual states bit 14 must always be 1 (reserved)
        tmp |= (1 << I2C_OAR1_RESERVED);

        // bit 15 - ADDMODE must be kept 0 for 7 bit address mode
        // set OAR1 in register map (need this address when in slave mode)
        i2c_control->i2c_regs->OAR1 = tmp;

        /**** I2C_CCR ****/
    // CCR field is bit 0 to 11
        // control bit 15 F/S is 0 for standard mode used for this project
        tmp = 0;
        if(i2c_control->config.I2C_SCL <= SCL_DEFAULT)
        {
                ccr = RCC_PCLK1_get()/(2 * i2c_control->config.I2C_SCL); // multiply by 2
from standard mode ccr formula
                tmp |= (ccr & 0xFFF); // first 12 bits
        }
        else{ // Fast Mode
                tmp |= (1 << I2C_CCR_FS); // set F/S to Fast Mode (bit 15)
                tmp |= (i2c_control->config.I2C_FM << I2C_CCR_DUTY); // set DUTY (bit 14) to
given FM duty cycle
                if(i2c_control->config.I2C_FM == FMPI2C_DUTY_CYCLE_2)
                        ccr = RCC_PCLK1_get()/(3 * i2c_control->config.I2C_SCL); // DUTY is 2
                else
                        ccr = RCC_PCLK1_get()/(25 * i2c_control->config.I2C_SCL); // DUTY is
16/9

                // set CCR field and mask to first 12 bits
                tmp |= (ccr & 0xFFF);
        }
        // set CCR is register map
        i2c_control->i2c_regs->CCR = tmp;

        /**** I2C_TRISE ****/
        if(i2c_control->config.I2C_SCL <= SCL_DEFAULT)
        {
                tmp = (RCC_PCLK1_get() / 1000000U) + 1; // add 1 from reference manual
        }
        else{ // fast mode
                tmp = ((RCC_PCLK1_get() * 300) / 1000000000U) + 1;
        }

        i2c_control->i2c_regs->TRISE = (tmp & 0x3F); // 6 bit mask

        return;
}

void I2C_MasterSend(I2C_control_t* i2c_control, uint8_t* tx_buf, uint32_t len, uint8_t
slave_addr)
```

```
{
        // start condition
        I2C_Start(i2c_control->i2c_regs);

        // wait for SB (start bit)
        while(!(I2C_GetStatus(i2c_control->i2c_regs, I2C_SR1_FLAG_SB)));

        I2C_SendAddr(i2c_control->i2c_regs, slave_addr);

        I2C_ClearADDRFlag(i2c_control->i2c_regs);

        // send data until len is 0
        for (; len > 0; len--){
                while(!(I2C_GetStatus(i2c_control->i2c_regs, I2C_SR1_FLAG_TXE))); // wait
for TxE
                i2c_control->i2c_regs->DR = *tx_buf; // dereference for value
                tx_buf++; // increment position
        }

        // wait for TxE and BTF in I2C_SR1 then set STOP to 1 in I2C_CR1
        while(!(I2C_GetStatus(i2c_control->i2c_regs, I2C_SR1_FLAG_TXE)));
        while(!(I2C_GetStatus(i2c_control->i2c_regs, I2C_SR1_FLAG_BTF)));
        I2C_Stop(i2c_control->i2c_regs);

        return;
}

// i2c start condition
static void I2C_Start(I2C_regs_t* i2c_regs){
        /* START bit 8 in I2C_CR1
         * set to 1 for repeated start generation
         * if PE is 0, the i2c hardware will clear START bit
         */
        i2c_regs->CR1 |= (1 << I2C_CR1_START);

        return;
}

// i2c stop condition
static void I2C_Stop(I2C_regs_t* i2c_regs){
        // I2C_CR1 STOP (bit 9)
        // when bit set to 1:
        //    "Stop generation after the current byte transfer or after the current Start
condition is sent"
```

```
        i2c_regs->CR1 |= (1 << I2C_CR1_STOP);
}

// send address with r/w bit set to 0
static void I2C_SendAddr(I2C_regs_t* i2c_regs, uint8_t slave_addr)
{
        slave_addr = (slave_addr << 1); // shift left for r/w_ bit
        slave_addr &= ~(1); // set r/w_ bit to 0

        /* I2C_DR (data register) has DR field from bits 0 to 7
         * "Transmitter mode: Byte transmission starts automatically when a byte is written in the DR register" - 24.6.5
         */
        i2c_regs->DR = slave_addr;

        return;
}

/* ADDR bit is SR1
 * "This bit is cleared by software reading SR1 register followed reading SR2
 * or by hardware when PE=0" - 24.6.6
 */
static void I2C_ClearADDRFlag(I2C_regs_t* i2c_regs){
        uint32_t foo = i2c_regs->SR1;
        foo = i2c_regs->SR2;
        (void)foo;
}
```

ADCS_comms/drivers/Src/gpio.h

```
/*
 * gpio.c
 *
 */

#include "../Inc/gpio.h"
#include "../Inc/rcc.h"

/* Only use init as we are using the GPIO ports
 * to configure them as I2C pins
 */
void GPIO_ClkEnable(GPIO_regs_t* gpio_regs, uint8_t enable)
{
        if(enable == TRUE)
        {
                if(gpio_regs == GPIOA)
                        GPIOA_CLK_ENABLE();
                else if (gpio_regs == GPIOB)
                        GPIOB_CLK_ENABLE();
        }
        else return;
}

void GPIO_Init(GPIO_control_t* gpio)
{
        uint32_t tmp = 0;
        //enable the peripheral clock

        GPIO_ClkEnable(gpio->gpio_regs, TRUE);

        // pin mode
        // non IT modes
        if(gpio->config.GPIO_Mode <= GPIO_MODE_ANALOG)
        {
                // multiply by 2 since each pin mode is a two bit field in GPIO_MODER
                tmp = (gpio->config.GPIO_Mode << (2 * gpio->config.GPIO_Pin ) );
                gpio->gpio_regs->GPIO_MODER &= ~( 0x3 << (2 * gpio->config.GPIO_Pin));
//clearing
                gpio->gpio_regs->GPIO_MODER |= tmp; //setting
        }
        // else is IT iterrupt modes which is unused for this
        // project as we are focusing on I2C driver

        // speed
        tmp = (gpio->config.GPIO_Speed << (2 * gpio->config.GPIO_Pin));
        gpio->gpio_regs->GPIO_OSPEEDR &= ~( 0x3 << ( 2 * gpio->config.GPIO_Pin));
//clearing
        gpio->gpio_regs->GPIO_OSPEEDR |= tmp;

        //pull-up/pull-down resistor
        tmp = (gpio->config.GPIO_PUPD << ( 2 * gpio->config.GPIO_Pin) );
        gpio->gpio_regs->GPIO_PUPDR &= ~( 0x3 << ( 2 * gpio->config.GPIO_Pin)); //clearing
        gpio->gpio_regs->GPIO_PUPDR |= tmp;

        //output type
        tmp = (gpio->config.GPIO_Output << gpio->config.GPIO_Pin);
        gpio->gpio_regs->GPIO_OTYPER &= ~( 0x1 << gpio->config.GPIO_Pin); //clearing
        gpio->gpio_regs->GPIO_OTYPER |= tmp;

        //5. configure the alt functionality
        if(gpio->config.GPIO_Mode == GPIO_MODE_ALTFUNC)
        {
            uint8_t high_reg =  gpio->config.GPIO_Pin / 8;
            tmp =  gpio->config.GPIO_Pin % 8;

            if(high_reg)
            {
                gpio->gpio_regs->GPIO_AFRH &= ~(0xF << (4 * tmp)); //clear
                gpio->gpio_regs->GPIO_AFRH |= (gpio->config.GPIO_AltFunc << (4 *
    tmp));
            }
            else { // alternate function reg low
                gpio->gpio_regs->GPIO_AFRL &= ~(0xF << (4 * tmp)); //clear
                gpio->gpio_regs->GPIO_AFRL|= (gpio->config.GPIO_AltFunc << (4 *
    tmp));
            }
        }

    }
```

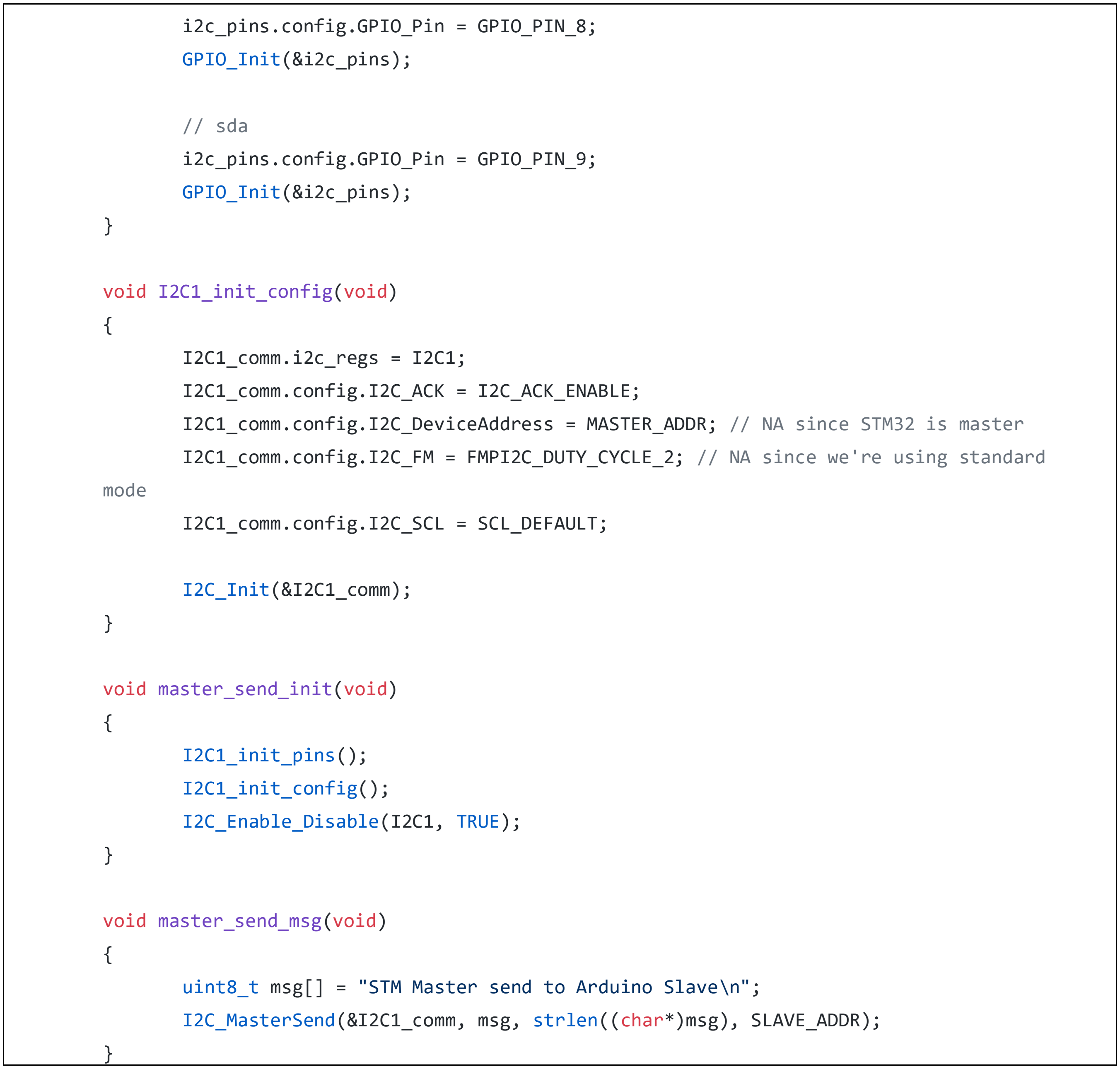
ADCS_comms/Inc/master_send.h

```
/*
     * master_send.h
     *
     *      Author: Adam
     *
     *      This is to test master send data
     *      The STM32 communicates to an Arudino i2c slave
     */

    /* Pins:
     * SCL - PB8
     * SDA - PB9
     */
    #ifndef INC_MASTER_SEND_H_
    #define INC_MASTER_SEND_H_

    #define MASTER_ADDR 0x61 // STM addr is NA
    #define SLAVE_ADDR 0x68 // Arduino slave address
```

```
void master_send_init(void);
void master_send_msg(void);

#endif /* INC_MASTER_SEND_H_ */
```

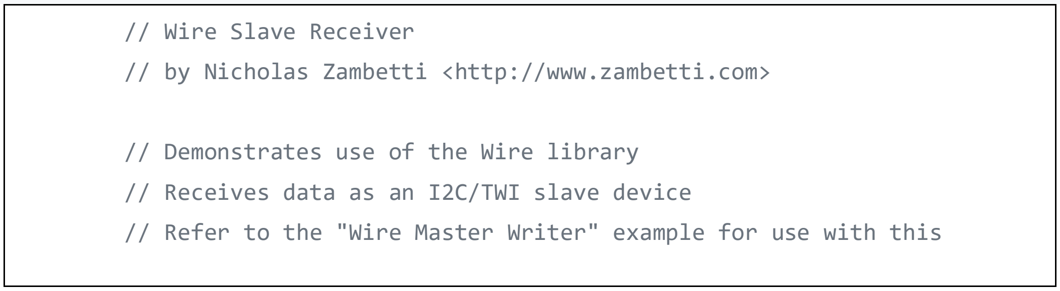
ADCS_comms/Src/master_send.c

```
/*
 * master_send_test.c
 *
 *      Author: Adam
 *
 *      This is to test master send data
 *      The STM32 communicates to an Arudino i2c slave
 */

/* Pins:
 * SCL - PB8
 * SDA - PB9
 */

#include <stdio.h>
#include <string.h>
#include "../drivers/Inc/mcu.h"
#include "../drivers/Inc/gpio.h"
#include "../drivers/Inc/i2c.h"
#include "../Inc/master_send.h"

I2C_control_t I2C1_comm;

void I2C1_init_pins(void)
{
	GPIO_control_t i2c_pins;

	i2c_pins.gpio_regs = GPIOB;
	i2c_pins.config.GPIO_Mode = GPIO_MODE_ALTFUNC; // alternating function type
	i2c_pins.config.GPIO_Output = GPIO_OUTPUT_OD; // open drain output type
	i2c_pins.config.GPIO_PUPD = GPIO_PIN_PU; // internal pullup resistor
	i2c_pins.config.GPIO_AltFunc = GPIO_AF4; // alternate function mode is 4
	i2c_pins.config.GPIO_Speed = GPIO_SPEED_FAST;

	// scl
```



```
                i2c_pins.config.GPIO_Pin = GPIO_PIN_8;
                GPIO_Init(&i2c_pins);

                // sda
                i2c_pins.config.GPIO_Pin = GPIO_PIN_9;
                GPIO_Init(&i2c_pins);
        }

        void I2C1_init_config(void)
        {
                I2C1_comm.i2c_regs = I2C1;
                I2C1_comm.config.I2C_ACK = I2C_ACK_ENABLE;
                I2C1_comm.config.I2C_DeviceAddress = MASTER_ADDR; // NA since STM32 is master
                I2C1_comm.config.I2C_FM = FMPI2C_DUTY_CYCLE_2; // NA since we're using standard
        mode
                I2C1_comm.config.I2C_SCL = SCL_DEFAULT;

                I2C_Init(&I2C1_comm);
        }

        void master_send_init(void)
        {
                I2C1_init_pins();
                I2C1_init_config();
                I2C_Enable_Disable(I2C1, TRUE);
        }

        void master_send_msg(void)
        {
                uint8_t msg[] = "STM Master send to Arduino Slave\n";
                I2C_MasterSend(&I2C1_comm, msg, strlen((char*)msg), SLAVE_ADDR);
        }
```

Arduino_ide/slave_receiver_2/slave_receiver_2.ino [18]



```
        // Wire Slave Receiver
        // by Nicholas Zambetti <http://www.zambetti.com>

        // Demonstrates use of the Wire library
        // Receives data as an I2C/TWI slave device
        // Refer to the "Wire Master Writer" example for use with this
```

```
// Created 29 March 2006

// This example code is in the public domain.


#include <Wire.h>
#define MY_ADDR   0x68

void setup() {
  Wire.begin(MY_ADDR);                // join i2c bus with address #8
  Wire.onReceive(receiveEvent); // register event
  Serial.begin(9600);           // start serial for output

  Serial.println("Slave is ready : Address 0x68");
  Serial.println("Waiting for data from master");
}

void loop() {

}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void receiveEvent(int howMany) {
  while (1 < Wire.available()) { // loop through all but the last
    char c = Wire.read(); // receive byte as a character
    Serial.print(c);         // print the character
  }
  int x = Wire.read();    // receive byte as an integer
  Serial.println(x);         // print the integer
}
```

APPENDIX H

Payload Module Software

Systemd service file – payload.service

```
[Unit]
Description=Payload main application
After=multi-user.target
Conflicts=getty@tty1.service

[Service]
Type=simple
ExecStart=/usr/bin/python3 /usr/local/bin/run.py
StandardInput=tty-force

[Install]
WantedBy=multi-user.target
```

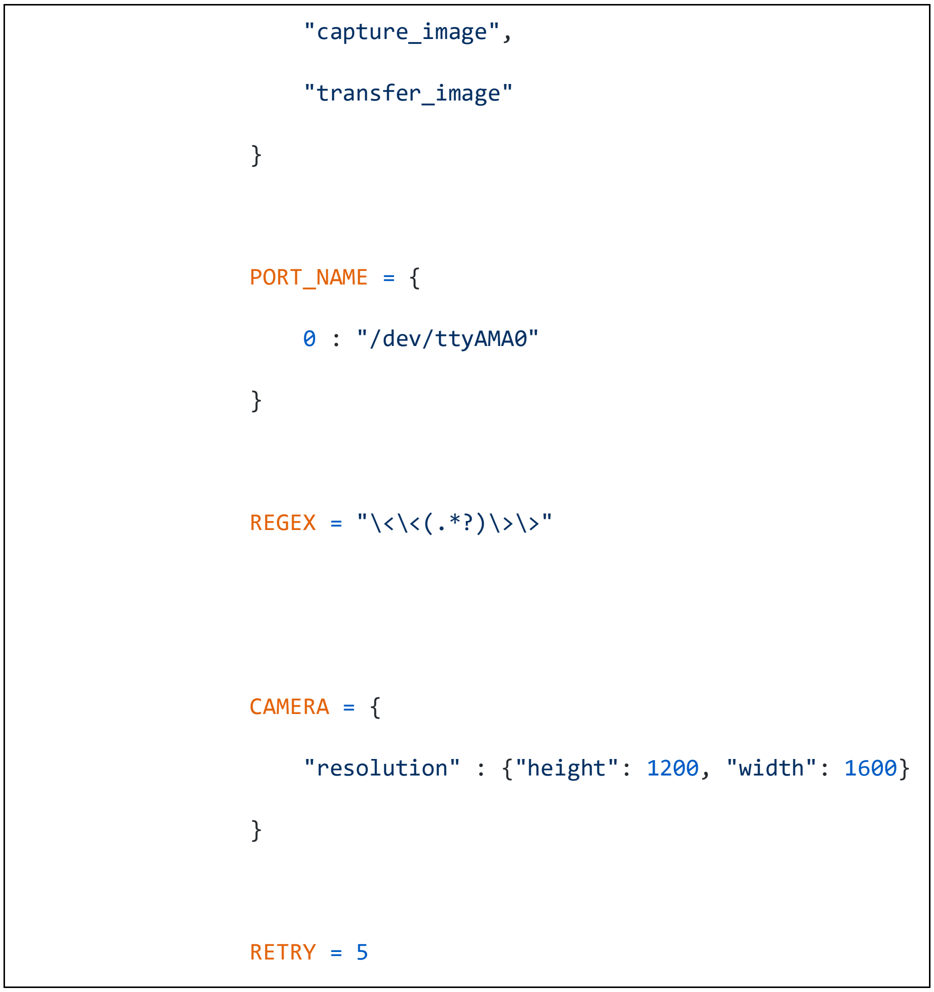
Main file – run.py

```
#!/usr/bin/env python3

import argparse

# get arguments
parser = argparse.ArgumentParser()
```

```
    parser.add_argument('--debug', '-d', dest='debug', action='store_const',
                        const=True, default=False,
                        help='run application in debug mode')
    parser.add_argument('--uart', '-u', dest='uart', action='store_const',
                        const=False, default=True,
                        help='run application using real uart port - defaults to mock uart connection')
    args = parser.parse_args()

    # Start payload application
    from app import main
    main.run(debug=args.debug, uart=args.uart)
```

Config File – config.py

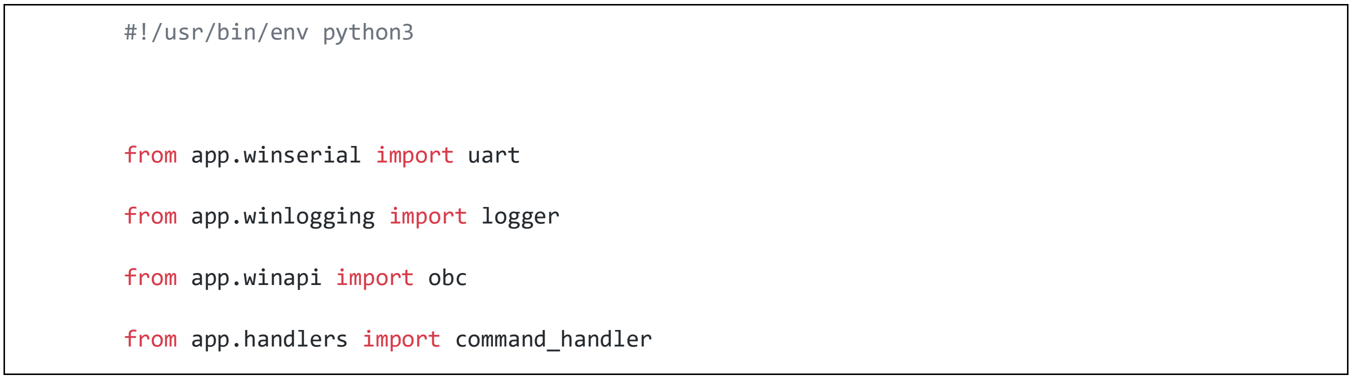
```
#!/usr/bin/env python3

from app.handlers import command_handler

# RETURN CODES
RETURN_CODE = {
    0 : "OK",
    1 : "ERR",
    2 : "DONE",
    3 : "FAIL"
}

COMMANDS = {
    "ping",
```



```
    "capture_image",

    "transfer_image"

}


PORT_NAME = {

    0 : "/dev/ttyAMA0"

}


REGEX = "\<\<(.*?)\>\>"




CAMERA = {

    "resolution" : {"height": 1200, "width": 1600}

}


RETRY = 5
```

PAYLOAD APPLICATION

main.py



```
#!/usr/bin/env python3



from app.winserial import uart

from app.winlogging import logger

from app.winapi import obc

from app.handlers import command_handler
```

```
import time
import serial

# setup logger
logger = logger.Logger("main")

# setup global objects
OBC = obc.OBC()
ch = command_handler.CommandHandler()

def run(debug, uart):

    logger.info("Trying to initiate connection with OBC...")
    OBC.connect(uart) # this will loop until a connection is made

    logger.info("Initiated connection with OBC. Waiting for messages...")
    # start main system loop
    while True:
        try:
            command = OBC.read()
            if command == None:
                continue

            if (OBC.check_command(command)):
                OBC.status(True)
                success, response = ch.handle(command, OBC)
                if success:
                    logger.info("Successful handling command: {}. Sending back reponse: {}".format(command, response))
                    OBC.write(response)
                else:
                    logger.info("Unsuccessful handling command: {}. Sending back error status...".format(command))
                    OBC.status(False)
            else:
                logger.info("Command received is invalid: {}. Sending back error status...".format(command))
                OBC.status(False)

        except Exception as e:
            logger.warn("Exception {}:{}".format(type(e).__name__, str(e)))
            # pass to error handler here?
            # error_handler.handle(error)

        finally:
            time.sleep(0.01)
            # kick watchdog here

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

logger.py

```
#!/usr/bin/env python3
import logging
```

```
class Logger():

    def __init__(self, logger_name, log_level=logging.DEBUG):
        # setup logging
        self.logger = logging.getLogger(logger_name)
        self.logger.setLevel(log_level)

        # setup log file handler
        fh = logging.FileHandler('/var/log/app/{}.log'.format(logger_name))
        fh.setLevel(log_level)

        # setup log console hanlder
        ch = logging.StreamHandler()
        ch.setLevel(log_level)

        # set log format
        formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
        fh.setFormatter(formatter)
        ch.setFormatter(formatter)

        # add log handlers to logger
        self.logger.addHandler(fh)
        self.logger.addHandler(ch)

    def info(self, message):
        self.logger.info(message)

    def debug(self, message):
        self.logger.debug(message)


    def warn(self, message):

        self.logger.warn(message)


    def error(self, message):

        self.logger.error(message)
```

OBC API – obc.py

```
#!/usr/bin/env python3


'''

API for interacting with OBC

'''


import time

import re


from app.winlogging import logger

from app.winserial import uart

from app import config


class OBC():


    def __init__(self):

        self.logger = logger.Logger("OBC")

        self.UART = None
        self.use_uart = None

    # get UART port
    def connect(self, use_uart):
        self.use_uart = use_uart
        if self.use_uart:
            while True:
                try:
                    self.UART = uart.UART(0)
                    break
                except Exception as e:
                    self.logger.error("FATAL ERROR: Unable to open UART port {}:{}. No communication with OBC. Retrying in 10 seconds...".format(type(e).__name__, str(e)))
                    # maybe reboot here after a while? -> pass to error handler for that?
                    time.sleep(10)
        else:
            self.logger.info("Setup fake connection with OBC for testing.")

    # read from UART serial port
    def read(self):
        if self.use_uart:
            success, message = self.UART.read()
            if success:
                return self.unpack(message)
            return None
        else:
            return self.unpack(input("Enter fake serial input:"))
    # check if command is valid
    def check_command(self, command):
        if command in config.COMMANDS:
            return True
        else:
            return False

    def pack(self, message):
        return "<<" + message + ">>"

    def unpack(self, message):
        commands = re.findall(config.REGEX, message)
        if len(commands) != 0:
            return commands[0]
        else:
            return None

    # write respone back to OBC
    def write(self, message):
        if self.use_uart:
            self.UART.write(self.pack(message))
        else:
            print(self.pack(message))

    # write status back to OBC
    def status(self, success):
        if self.use_uart:
            if success:
                self.UART.write(self.pack(config.RETURN_CODE[0]))
            else:
                self.UART.write(self.pack(config.RETURN_CODE[1]))

        else:
            if success:
                print(self.pack(config.RETURN_CODE[0]))
            else:
                print(self.pack(config.RETURN_CODE[1]))

    def send_image(self, filename):
        if self.use_uart:
            return self.UART.transfer_image(filename)
        else:
            return True
```

Handlers

command_handler.py

```
#!/usr/bin/env python3

'''
Class for handling commands received from OBC
'''
```

```
import time
from app.winlogging import logger
from app.handlers import image_handler
from app.handlers import ping_handler
import re
from app import config

image_handler = image_handler.ImageHandler()
ping_handler = ping_handler.PingHandler()
#error_handler = error_handler.ErrorHandler()

class CommandHandler():

    def __init__(self):
        # setup logger
        self.logger = logger.Logger("command-handler")

    def handle(self, command, OBC):
        # send the command to the appropriate handler

        # command: ping
        if command == "ping":
            success, response = ping_handler.handle_ping()

        # command: image_capture
        elif command == "capture_image":
            success, response = image_handler.handle_capture()
        # command: image_transfer
        elif command == "transfer_image":
            success, response = image_handler.handle_transfer(OBC)


        else:
            # should never get here
            self.logger.warn("FATAL: Command {} is in valid commands but doesn't have handler.".format(command))
            success = False


        return success, response
```

image_handler.py

```
#!/usr/bin/env python3


'''
Class for interacting with images (transfer, read/write, captures, etc.)
'''



import os
import time
from datetime import datetime
from app.winserial import uart
from app.winlogging import logger
from app import config
```

```python
# pi camera stuff
from picamera import PiCamera
from picamera.array import PiRGBArray
import time
import cv2

class ImageHandler():

    def __init__(self):
        # setup logger
        self.logger = logger.Logger("image-handler")

    def handle_capture(self):
        # capture image here
        success = self.capture_image()
        if success:
            return success, config.RETURN_CODE[2]
        else:
            return success, config.RETURN_CODE[3]

    def handle_transfer(self, OBC):
        # start image transfer
        success = self.transfer_image(OBC)
        if success:
            return success, config.RETURN_CODE[2]
        else:
            return success, config.RETURN_CODE[3]
    def capture_image(self):
        try:
            # initialize the camera and grab a reference to the raw camera capture
            camera = PiCamera()
            camera.resolution = (config.CAMERA["resolution"]["width"],
config.CAMERA["resolution"]["height"])
            rawCapture = PiRGBArray(camera)

            # allow the camera to warmup
            time.sleep(0.1)

            # grab an image from the camera
            camera.capture(rawCapture, format="bgr")
            image = rawCapture.array

            # save image with timestamp
            ts = datetime.now()
            filename = '/images/{}-{}-{}.{}:{}:{}.jpg'.format(ts.year, ts.month, ts.day,
ts.hour, ts.minute, ts.second)
            cv2.imwrite(filename, image)

            camera.close()

            self.logger.info("Successful image capture. Image saved at:
{}".format(filename))
            return True

        except Exception as e:
            self.logger.warn("Unable to capture image: {} | {}".format(type(e).__name__,
str(e)))
```

```
            return False


    def transfer_image(self, OBC):
        try:
            # wait for the OBC to tell us to start
            start = False
            for i in range(config.RETRY):
                command = OBC.read()
                if command == "START":
                    self.logger.debug("Got START command. Starting image transfer...")
                    start = True
                    break
                else:
                    self.logger.debug("Waiting for START for image transfer. Got:
{}".format(command))
                    start = False


            if start:
                filename = None
                latest_time = 0
                with os.scandir('/images/') as entries:
                    for entry in entries:
                        info = entry.stat()
                        if info.st_mtime > latest_time:
                            filename = entry.name
                            latest_time = info.st_mtime


                if filename == None:
                    return False
                filepath = '/images/{}'.format(filename)

                # open up stream to start image transfer


                return OBC.send_image(filepath)

            else:

                self.logger.debug("Aborting image transfer. Never got START command.")

                return False


        except Exception as e:

            self.logger.warn("Unable to transfer image name: {} error:
{}|{}".format(filename, type(e).__name__, str(e)))

            return False
```

ping_handler.py

```
#!/usr/bin/env
python3
```

```
'''

Class for handling ping command received from OBC

'''
```

```
import time
```

```
from app.winlogging import logger


class PingHandler:


    def __init__(self):

        # setup logger

        self.logger = logger.Logger("ping-handler")


    def handle_ping(self):

        return True, "pong"
```

queue_handler.py

```
#!/usr/bin/env
python3
```

```
'''

Class for handling input serial queue received from OBC

'''


import time

from app.winlogging import logger

import re

from threading import Thread

import queue




class QueueHandler(Thread):
    def __init__(self):
        # setup logger
        self.logger = logger.Logger("queue-handler")

    def check_command(self, buffer):
        command = self.parse(buffer)
        for key in COMMANDS:
            if command in key:
                return True
        return False

    def handle(self, buffer):
        # send the command to the appropriate handler
        if command in COMMANDS["PING"]:
            success, response = ping_handler.handle(command)
        elif command in COMMANDS["IMAGE"]:
            success, response = image_handler.handle(command)
        else:
            # should never get here
            self.logger.warn("FATAL: Got invalid command from OBC: {}. PREVIOUS
CHECK FOR THIS FAILED.".format(command))
            success = False
            response = INVALID

        response = format(response)
        return success, response

    # gets input from UART buffer, parses it, and returns commands
```

```python
    def parse(self, buffer):
        commands = re.findall(REGEX, buffer)
        if commands is not None:
            return True, commands
        else:
            return False, None

    # format responses to be send over serial back to OBC
    def format(self, command):
        return "<<" + command + ">>"
```

APPENDIX I

KubOS OBC Software

KUBOS SERVICES

SYSTEMD SERVICE INIT SCRIPTS

adcs-service.init

```
#!/bin/sh

# Start the ADCS service in the background
# passing in the location of the config.toml file
# (in the same directory as the service)
export PYTHONPATH=$PYTHONPATH:/home/kubos/winlib
python /home/kubos/adcs/adcs-service/service.py -c /etc/kubos-config.toml &

exit 0
```

eps-service.init

```
#!/bin/sh

# Start the EPS service in the background
# passing in the location of the config.toml file
# (in the same directory as the service)
export PYTHONPATH=$PYTHONPATH:/home/kubos/winlib
python /home/kubos/eps/eps-service/service.py -c /etc/kubos-config.toml &

exit 0
```

rtc-service.init

```
#!/bin/sh


# Start the RTC service in the background

# passing in the location of the config.toml file

# (in the same directory as the service)

export PYTHONPATH=$PYTHONPATH:/home/kubos/winlib

python /home/kubos/rtc/rtc-service/service.py -c /etc/kubos-config.toml &


exit 0
```

radio-service.init

```
#!/bin/sh


# Start the EPS service in the background

# passing in the location of the config.toml file

# (in the same directory as the service)

export PYTHONPATH=$PYTHONPATH:/home/kubos/winlib

python /home/kubos/radio/radio-service/service.py -c /etc/kubos-config.toml &


exit 0
```

payload-service.init

```
#!/bin/sh


# Start the payload service in the background
```

```
             # passing in the location of the config.toml file

             # (in the same directory as the service)

             export PYTHONPATH=$PYTHONPATH:/home/kubos/winlib

             python /home/kubos/payload/payload-service/service.py -c /etc/kubos-config.toml &


             exit 0
```

MONIT SERVICE SCRIPTS

adcs-service.monit

```
CHECK PROCESS
adcs-service
PIDFILE
/var/run/adcs-
service.pid
                        START PROGRAM = "/home/system/etc/init.d/S01adcs-service start"

                        IF 3 RESTART WITHIN 10 CYCLES THEN TIMEOUT
```

eps-service.monit

```
CHECK PROCESS
eps-service
PIDFILE
/var/run/eps-
service.pid
                       START PROGRAM = "/home/system/etc/init.d/S01eps-service start"

                       IF 3 RESTART WITHIN 10 CYCLES THEN TIMEOUT
```

rtc-service.monit

```
        CHECK PROCESS rtc-service PIDFILE /var/run/rtc-service.pid

               START PROGRAM = "/home/system/etc/init.d/S01rtc-service start"

               IF 3 RESTART WITHIN 10 CYCLES THEN TIMEOUT
```

radio-service.monit

```
CHECK PROCESS radio-service PIDFILE /var/run/radio-service.pid
       START PROGRAM = "/home/system/etc/init.d/S01radio-service start"
       IF 3 RESTART WITHIN 10 CYCLES THEN TIMEOUT
```

payload-service.monit

```
CHECK PROCESS payload-service PIDFILE /var/run/payload-service.pid
       START PROGRAM = "/home/system/etc/init.d/S01payload-service start"
       IF 3 RESTART WITHIN 10 CYCLES THEN TIMEOUT
```

ADCS SERVICE

service.py

```
#!/usr/bin/env python3

"""
Boilerplate service code which reads the config file and starts up the
GraphQL/HTTP endpoint. (should not need to much modification)
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
```

```
__license__ = "MIT"

import logging

from service import schema
from kubos_service.config import Config
from logging.handlers import SysLogHandler
import sys

config = Config("adcs-service")

# Setup logging
logger = logging.getLogger("adcs-service")
logger.setLevel(logging.DEBUG)
handler = SysLogHandler(address='/dev/log', facility=SysLogHandler.LOG_DAEMON)
formatter = logging.Formatter('adcs-service: %(message)s')
handler.formatter = formatter
logger.addHandler(handler)

# Set up a handler for logging to stdout
stdout = logging.StreamHandler(stream=sys.stdout)
stdout.setFormatter(formatter)
logger.addHandler(stdout)

from kubos_service import http_service
# Start an http service
http_service.start(config, schema.schema)
```

```
#from kubos_service import udp_service


# Start a udp service with optional context
# udp_service.start(config, schema, {'bus': '/dev/ttyS3'})


# Start a udp service
#udp_service.start(logger, config, schema)
```

app.py

```
#!/usr/bin/env python3


"""
Boilerplate Flask setup for service application (should not be modified)
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"


from flask import Flask
from flask_graphql import GraphQLView
from .schema import schema


def create_app():
    """
    Creates graphql and graphiql endpoints
    """
    app = Flask(__name__)
    app.debug = True

    app.add_url_rule(
        '/',
        view_func=GraphQLView.as_view(
            'graphql',
            schema=schema,
            graphiql=False
        )
    )

    app.add_url_rule(
        '/graphiql',
        view_func=GraphQLView.as_view(
            'graphiql',
            schema=schema,
            graphiql=True
        )
    )

    return app
```

models.py

```python
#!/usr/bin/env python3

"""
Graphene ObjectType classes for subsystem modeling.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import graphene
import serial

# POWER
class PowerStateEnum(graphene.Enum):
    OFF = 0
    ON = 1
    RESET = 2

class PowerState(graphene.ObjectType):
    state = graphene.Field(PowerStateEnum)

# MODE
class ModeStateEnum(graphene.Enum):
    IDLE = 0
    DETUMBLE = 1
    POINTING = 2
```

```
class ModeState(graphene.ObjectType):
    state = graphene.Field(ModeStateEnum)

# ORIENTATION
class Orientation(graphene.ObjectType):
    x = graphene.Float()
    y = graphene.Float()
    z = graphene.Float()
    yaw = graphene.Float()
    pitch = graphene.Float()
    roll = graphene.Float()

# SPIN
class Spin(graphene.ObjectType):
    x = graphene.Float()
    y = graphene.Float()
    z = graphene.Float()

# Mutation Result
class Result(graphene.ObjectType):
    errors = graphene.List(graphene.String)
    success = graphene.Boolean()

# Control power mutation input
class ControlPowerInput(graphene.InputObjectType):
    power = graphene.Field(PowerStateEnum)

# Set ADCS mode mutation input
```

```
class SetModeInput(graphene.InputObjectType):
    mode = graphene.Field(ModeStateEnum)

class Telemetry(graphene.ObjectType):
    # telemetry items for general status of hardware
    mode = graphene.Field(ModeState)
    power = graphene.Field(PowerState)
    orientation = graphene.Field(Orientation)
    spin = graphene.Field(Spin)
```

schema.py

```
#!/usr/bin/env python3

"""
Graphene schema setup to enable queries.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import graphene
from .models import *
from obcapi import adcs
```

```
# Local subsystem instance for tracking state

# May not be neccesary when tied into actual hardware

_adcs = adcs.ADCS()


############## QUERIES ################


'''

type Query {

    ping(): String

    power(): PowerState

    mode(): ModeState

    orientation(): Orientation

    spin(): Spin

    telemetry(): Telemetry

}

'''

class Query(graphene.ObjectType):


    '''

    query {

        ping

    }

    '''

    ping = graphene.String()

    def resolve_ping(self, info):

        return _adcs.ping()


    '''
```

```
query {
    power { state }
}
'''
power = graphene.Field(PowerState)
def resolve_power(self, info):
    state = _adcs.power()
    return PowerState(state=state)

'''
query {
    mode { state }
}
'''
mode = graphene.Field(ModeState)
def resolve_mode(self, info):
    state = _adcs.mode()
    return ModeState(state=state)

'''
query {
    orientation { x y z yaw pitch roll }
}
'''
orientation = graphene.Field(Orientation)
def resolve_orientation(self, info):
    orient = _adcs.orientation()
```

```
        return Orientation(x=orient[0],y=orient[1],z=orient[2],yaw=orient[3],pitch=orient[4],roll=orient[5])

    '''
    query {
        spin { x y z }
    }
    '''
    spin = graphene.Field(Spin)
    def resolve_spin(self, info):
        spin = _adcs.spin()
        return Spin(x=spin[0],y=spin[1],z=spin[2])

    '''
    query {
        telemetry {
            orientation { x y z yaw pitch roll }
            spin { x y z }
            mode { state }
            power { state }
        }
    }
    '''
    telemetry = graphene.Field(Telemetry)
    def resolve_telemetry(self, info):
        mode = _adcs.mode()
        power = _adcs.power()
        o = _adcs.orientation()
        orientation = Orientation(x=o[0],y=o[1],z=o[2],yaw=o[3],pitch=o[4],roll=o[5])

        spin = _adcs.spin()
        spin = Spin(x=spin[0],y=spin[1],z=spin[2])

        return Telemetry(   ModeState(state=mode),
                            PowerState(state=power),
                            orientation,
                            spin
        )

############## MUTATIONS ################

'''
mutation {
    controlPower(controlPowerInput: {power: OFF}) {
        success
        errors
        }
    }
'''
class ControlPower(graphene.Mutation):
    class Arguments:
        controlPowerInput = ControlPowerInput()

    Output = Result
    def mutate(self, info, controlPowerInput):
        success, errors = _adcs.controlPower(controlPowerInput)
        return Result(success=success, errors=errors)


'''
mutation {
    setMode(setModeInput: {mode: DETUMBLE}) {
        errors
        success
    }
}
'''
class SetMode(graphene.Mutation):
    class Arguments:
        setModeInput = SetModeInput()


    Output = Result
    def mutate(self, info, setModeInput):
        success, errors = _adcs.setMode(setModeInput)
        return Result(success=success, errors=errors)


'''
type Mutation {
    controlPower(
        input: ControlPowerInput!
    ): ControlPower
    setMode(
        input: SetModeInput!
    ): SetMode
}
'''

class Mutation(graphene.ObjectType):
    controlPower = ControlPower.Field()
    setMode = SetMode.Field()


schema = graphene.Schema(query=Query, mutation=Mutation)
```

EPS SERVICE

service.py

```
#!/usr/bin/env python3

"""
Boilerplate service code which reads the config file and starts up the
GraphQL/HTTP endpoint. (should not need to much modification)
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import logging

from service import schema
from kubos_service.config import Config
from logging.handlers import SysLogHandler
import sys
```

```python
config = Config("eps-service")

# Setup logging
logger = logging.getLogger("eps-service")
logger.setLevel(logging.DEBUG)
handler = SysLogHandler(address='/dev/log', facility=SysLogHandler.LOG_DAEMON)
formatter = logging.Formatter('eps-service: %(message)s')
handler.formatter = formatter
logger.addHandler(handler)

# Set up a handler for logging to stdout
stdout = logging.StreamHandler(stream=sys.stdout)
stdout.setFormatter(formatter)
logger.addHandler(stdout)

from kubos_service import http_service
# Start an http service
http_service.start(config, schema.schema)

#from kubos_service import udp_service

# Start a udp service with optional context
# udp_service.start(config, schema, {'bus': '/dev/ttyS3'})

# Start a udp service
#udp_service.start(logger, config, schema)
```

app.py

```
#!/usr/bin/env
python3

"""
Boilerplate Flask setup for service application (should not be modified)
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

from flask import Flask
from flask_graphql import GraphQLView
from .schema import schema



def create_app():
    """
    Creates graphql and graphiql endpoints
    """

    app = Flask(__name__)
    app.debug = True

    app.add_url_rule(
        '/',
        view_func=GraphQLView.as_view(
            'graphql',
            schema=schema,
            graphiql=False
        )
    )

    app.add_url_rule(
        '/graphiql',
        view_func=GraphQLView.as_view(
            'graphiql',
            schema=schema,
            graphiql=True
        )
    )

    return app
```

models.py

```
#!/usr/bin/env python3

"""
Graphene ObjectType classes for subsystem modeling.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"
```

```
import graphene
import serial

class PowerEnum(graphene.Enum):
    OFF = 0
    ON = 1

class PortEnum(graphene.Enum):
    PORT1 = 1
    PORT2 = 2
    PORT3 = 3

class PowerState(graphene.ObjectType):
    power1 = graphene.Field(PowerEnum)
    power2 = graphene.Field(PowerEnum)
    power3 = graphene.Field(PowerEnum)

class ControlPortInput(graphene.InputObjectType):
    power = graphene.Field(PowerEnum)
    port = graphene.Field(PortEnum)

class Result(graphene.ObjectType):
    errors = graphene.List(graphene.String)
    success = graphene.Boolean()

class Telemetry(graphene.ObjectType):
    power = graphene.Field(PowerState)
```

```
        battery = graphene.Float()
```

schema.py

```
#!/usr/bin/env python3

"""
Graphene schema setup to enable queries.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import graphene
from .models import *

from obcapi import eps

_eps = eps.EPS()

'''
type Query {
    ping(): pong
    power(): PowerState
    telemetry(): Telemetry
    battery(): Float
}
```

```
'''
class Query(graphene.ObjectType):

    '''
    query {
        ping
    }
    '''
    ping = graphene.String()
    def resolve_ping(self, info):
        return _eps.ping()

    '''
    query {
        battery
    }
    '''
    battery = graphene.Int()
    def resolve_battery(self, info):
        return _eps.battery()

    '''
    query {
        power {
            power1
            power2
            power3
        }
    }
    '''
    power = graphene.Field(PowerState)
    def resolve_power(self, info):
        power1, power2, power3 = _eps.power()
        return PowerState(power1=power1, power2=power2, power3=power3)

    '''
    query {
        telemetry {
            power {
                power1
                power2
                power3
            }
            battery
        }
    }
    '''
    telemetry = graphene.Field(Telemetry)
    def resolve_telemetry(self, info):
        power1, power2, power3 = _eps.power()
        battery_level = _eps.battery()
        return Telemetry(power=PowerState(power1=power1, power2=power2,
power3=power3), battery=battery_level)

############## MUTATIONS ################
'''
mutation {
    controlPort(controlPortInput: {
                power: ON
                port: 1 })
    {
    errors
    success
    }
}
'''
class ControlPort(graphene.Mutation):
    class Arguments:
        controlPortInput = ControlPortInput()

    Output = Result
    def mutate(self, info, controlPortInput):
        if (_eps.controlPort(controlPortInput)):
            return Result(success=True, errors=[])
        else:
            return Result(success=False, errors=["Invalid port number: {}".format(controlPortInput.port)])

####################################################################
'''
type Mutation {
    controlPort(
        input: ControlPowerInput!
    ): Result
}
'''
```

```
class Mutation(graphene.ObjectType):

    controlPort = ControlPort.Field()


schema = graphene.Schema(query=Query, mutation=Mutation)
```

PAYLOAD SERVICE

service.py

```
#!/usr/bin/env
python3
```

```
"""

Boilerplate service code which reads the config file and starts up the

GraphQL/HTTP endpoint. (should not need to much modification)

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import logging


from service import schema

from logging.handlers import SysLogHandler

import sys

#import toml


#print("hello")

from kubos_service.config import Config
```

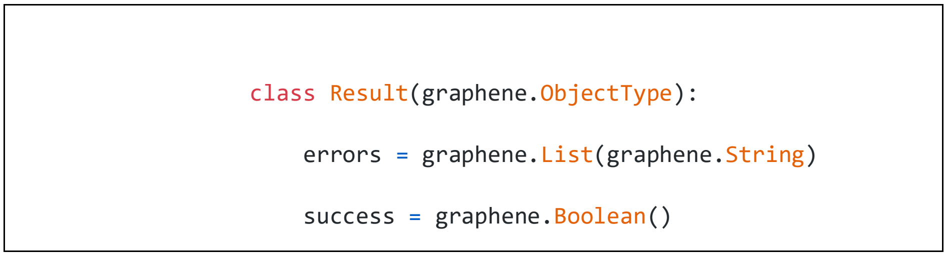
```python
config = Config("payload-service")
#print(toml.dumps(config))

# Setup logging
logger = logging.getLogger("payload-service")
logger.setLevel(logging.DEBUG)
handler = SysLogHandler(address='/dev/log', facility=SysLogHandler.LOG_DAEMON)
formatter = logging.Formatter('payload-service: %(message)s')
handler.formatter = formatter
logger.addHandler(handler)

# Set up a handler for logging to stdout
stdout = logging.StreamHandler(stream=sys.stdout)
stdout.setFormatter(formatter)
logger.addHandler(stdout)

from kubos_service import http_service
# Start an http service
http_service.start(config, schema.schema)

#from kubos_service import udp_service

# Start a udp service with optional context
# udp_service.start(config, schema, {'bus': '/dev/ttyS3'})

# Start a udp service
#udp_service.start(logger, config, schema)
```

app.py

```
#!/usr/bin/env
python3

"""
Boilerplate Flask setup for service application (should not be modified)
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

from flask import Flask
from flask_graphql import GraphQLView
from .schema import schema


def create_app():
    """
    Creates graphql and graphiql endpoints
    """

    app = Flask(__name__)
    app.debug = True

    app.add_url_rule(
        '/',
        view_func=GraphQLView.as_view(
            'graphql',
            schema=schema,
            graphiql=False
        )
    )

    app.add_url_rule(
        '/graphiql',
        view_func=GraphQLView.as_view(
            'graphiql',
            schema=schema,
            graphiql=True
        )
    )

    return app
```

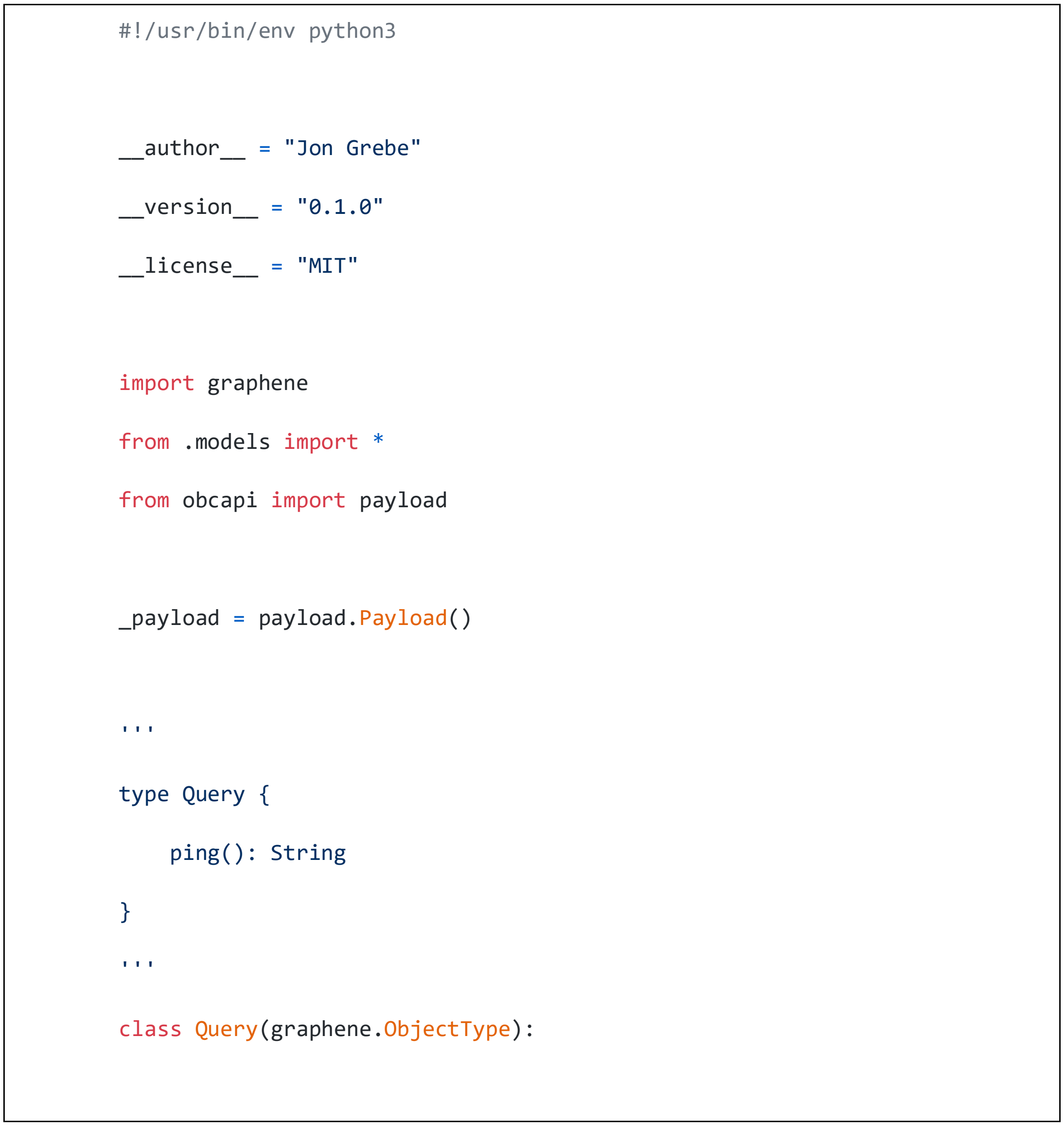
models.py

```
#!/usr/bin/env python3

"""
Graphene ObjectType classes for subsystem modeling.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import graphene
```



```
class Result(graphene.ObjectType):
    errors = graphene.List(graphene.String)
    success = graphene.Boolean()
```

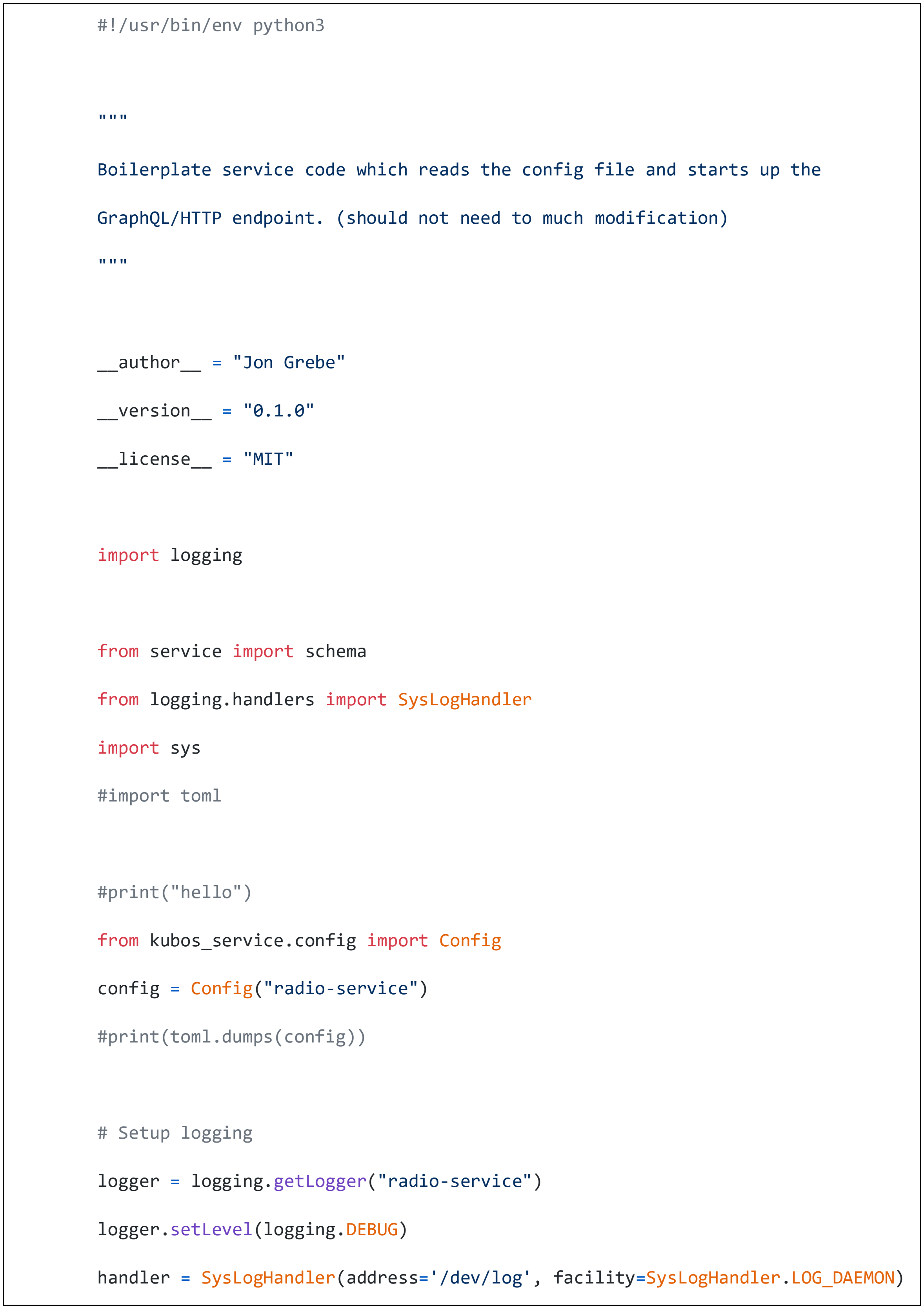
schema.py



```
#!/usr/bin/env python3

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import graphene
from .models import *
from obcapi import payload

_payload = payload.Payload()

'''
type Query {
    ping(): String
}
'''
class Query(graphene.ObjectType):
    '''
    {
        ping
    }
    '''
    ping = graphene.String()
    def resolve_ping(self, info):
        return _payload.ping()


'''
mutation {
    image_capture() {
        success
        errors
    }
}
'''
class ImageTransfer(graphene.Mutation):
    Output = Result
    def mutate(self, info):
        # should send hardware command to payload to start image transfer
        success, errors = _payload.image_transfer()
        # return results
        return Result(success=success, errors=errors)


'''
mutation {
    image_transfer() {
        success
        errors
    }
}
'''
class ImageCapture(graphene.Mutation):
    Output = Result
    def mutate(self, info):
        # should send hardware command to payload to start image capture
        success, errors = _payload.image_capture()
        # return results
        return Result(success=success, errors=errors)

'''
type Mutation {
    imageCapture(): Result
    imageTransfer(): Result
}
'''
class Mutation(graphene.ObjectType):
    imageCapture = ImageCapture.Field()
    imageTransfer = ImageTransfer.Field()

schema = graphene.Schema(query=Query, mutation=Mutation)
```

RADIO SERVICE

service.py



```
#!/usr/bin/env python3

"""
Boilerplate service code which reads the config file and starts up the
GraphQL/HTTP endpoint. (should not need to much modification)
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import logging

from service import schema
from logging.handlers import SysLogHandler
import sys
#import toml

#print("hello")
from kubos_service.config import Config
config = Config("radio-service")
#print(toml.dumps(config))

# Setup logging
logger = logging.getLogger("radio-service")
logger.setLevel(logging.DEBUG)
handler = SysLogHandler(address='/dev/log', facility=SysLogHandler.LOG_DAEMON)
```

```
    formatter = logging.Formatter('radio-service: %(message)s')
    handler.formatter = formatter
    logger.addHandler(handler)

    # Set up a handler for logging to stdout
    stdout = logging.StreamHandler(stream=sys.stdout)
    stdout.setFormatter(formatter)
    logger.addHandler(stdout)

    from kubos_service import http_service
    # Start an http service
    http_service.start(config, schema.schema)

    #from kubos_service import udp_service

    # Start a udp service with optional context
    # udp_service.start(config, schema, {'bus': '/dev/ttyS3'})

    # Start a udp service
    #udp_service.start(logger, config, schema)
```

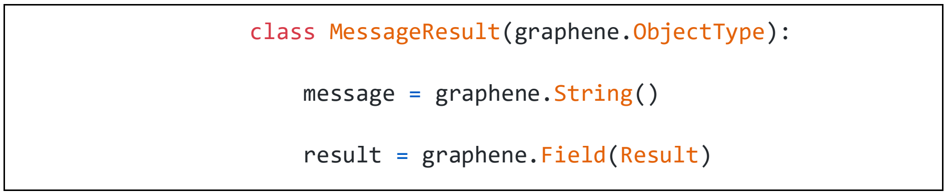
app.py

```
#!/usr/bin/env python3

"""
Boilerplate Flask setup for service application (should not be modified)
"""
```

```python
__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

from flask import Flask
from flask_graphql import GraphQLView
from .schema import schema


def create_app():
    """
    Creates graphql and graphiql endpoints
    """

    app = Flask(__name__)
    app.debug = True

    app.add_url_rule(
        '/',
        view_func=GraphQLView.as_view(
            'graphql',
            schema=schema,
            graphiql=False
        )
    )

    app.add_url_rule(
        '/graphiql',
        view_func=GraphQLView.as_view(
            'graphiql',
            schema=schema,
            graphiql=True
        )
    )

    return app
```

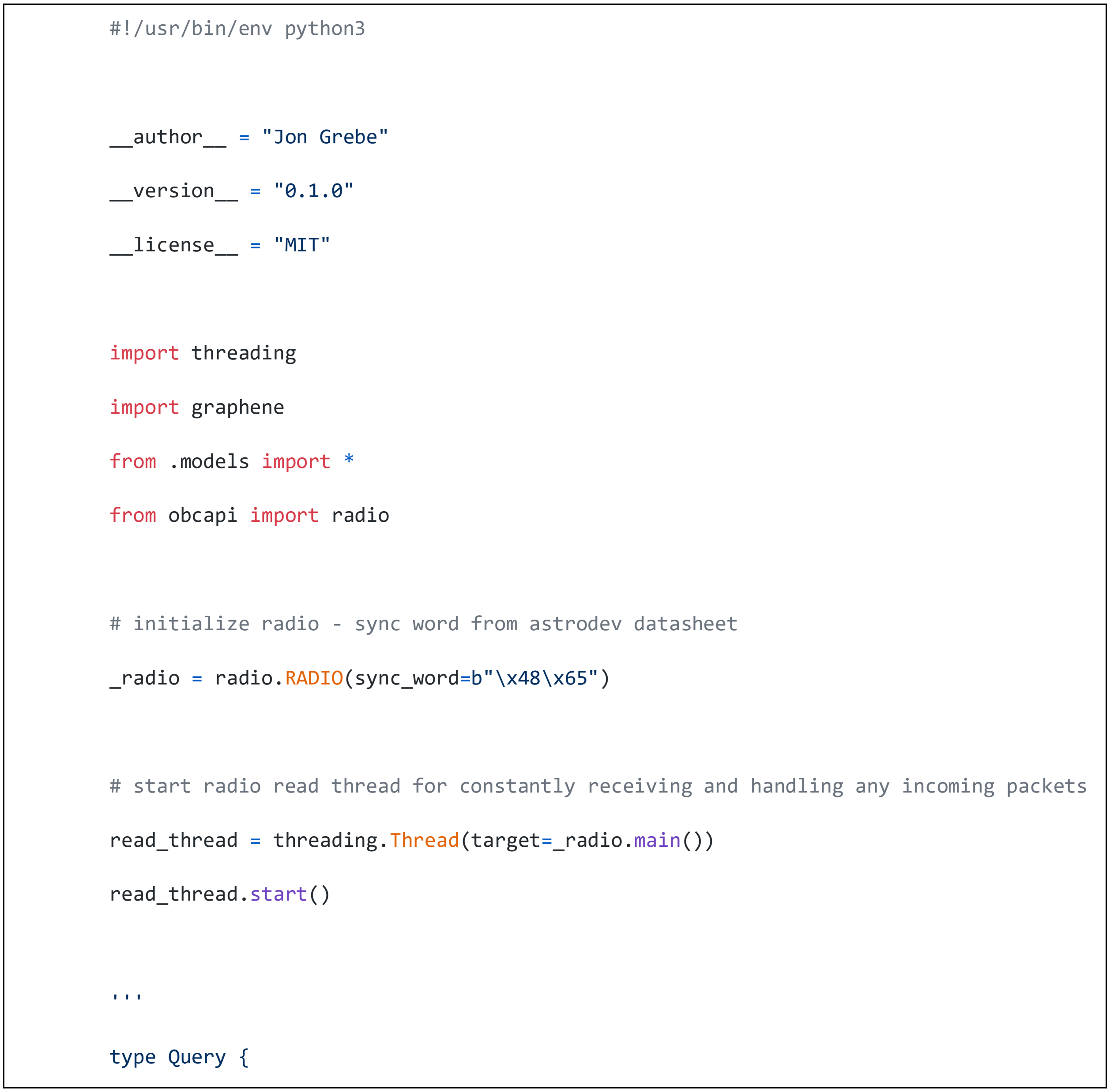
models.py

```
#!/usr/bin/env python3

"""
Graphene ObjectType classes for subsystem modeling.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import graphene

class Result(graphene.ObjectType):
    errors = graphene.List(graphene.String)
    success = graphene.Boolean()
```



```
class MessageResult(graphene.ObjectType):

    message = graphene.String()

    result = graphene.Field(Result)
```

schema.py



```
#!/usr/bin/env python3


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import threading

import graphene

from .models import *

from obcapi import radio


# initialize radio - sync word from astrodev datasheet

_radio = radio.RADIO(sync_word=b"\x48\x65")


# start radio read thread for constantly receiving and handling any incoming packets

read_thread = threading.Thread(target=_radio.main())

read_thread.start()


'''

type Query {
    ping(): String
}
'''
class Query(graphene.ObjectType):

    '''
    {
        ping
    }
    '''
    ping = graphene.String()
    def resolve_ping(self, info):
        return _radio.ping()

    '''
    {
        read {
            buffer
            success
            errors
        }
    }
    '''
    read = graphene.Field(MessageResult)
    def resolve_read(self, info):
        # should send hardware a ping and expect a pong back
        buffer = _radio.receive()
        success = True
        errors = []
        # return results
        result = Result(success=success, errors=errors)
        return MessageResult(message=buffer, result=result)

'''
mutation {
    image_transfer() {
        success
        errors
    }
}
'''
class ImageTransfer(graphene.Mutation):
    Output = Result
    def mutate(self, info):
        # should send hardware command to payload to start image transfer
        success, errors = _radio.image_transfer()
        # return results
        return Result(success=success, errors=errors)

'''
type Mutation {
    imageTransfer(): Result
}
'''
class Mutation(graphene.ObjectType):
    imageTransfer = ImageTransfer.Field()
```

```
schema = graphene.Schema(query=Query, mutation=Mutation)
```

IMU SERVICE

service.py

```
#!/usr/bin/env
python3

"""

Boilerplate service code which reads the config file and starts up the

GraphQL/HTTP endpoint. (should not need to much modification)

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import logging


from service import schema

from kubos_service.config import Config

from logging.handlers import SysLogHandler

import sys


config = Config("imu-service")


# Setup logging

logger = logging.getLogger("imu-service")
```

```
logger.setLevel(logging.DEBUG)

handler = SysLogHandler(address='/dev/log', facility=SysLogHandler.LOG_DAEMON)

formatter = logging.Formatter('imu-service: %(message)s')

handler.formatter = formatter

logger.addHandler(handler)


# Set up a handler for logging to stdout

stdout = logging.StreamHandler(stream=sys.stdout)

stdout.setFormatter(formatter)

logger.addHandler(stdout)


from kubos_service import http_service

# Start an http service

http_service.start(config, schema.schema)


#from kubos_service import udp_service


# Start a udp service with optional context

# udp_service.start(config, schema, {'bus': '/dev/ttyS3'})


# Start a udp service

#udp_service.start(logger, config, schema)
```

app.py

```
#!/usr/bin/env
python3
```

```
"""
```

```
Boilerplate Flask setup for service application (should not be modified)
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

from flask import Flask
from flask_graphql import GraphQLView
from .schema import schema


def create_app():
    """
    Creates graphql and graphiql endpoints
    """

    app = Flask(__name__)
    app.debug = True

    app.add_url_rule(
        '/',
        view_func=GraphQLView.as_view(
            'graphql',
            schema=schema,
            graphiql=False
        )
    )
    app.add_url_rule(
        '/graphiql',
        view_func=GraphQLView.as_view(
            'graphiql',
            schema=schema,
            graphiql=True
        )
    )

    return app
```

models.py

```
#!/usr/bin/env python3
"""
Graphene ObjectType classes for subsystem modeling.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import graphene
import serial
import time
import app_api
```

```python
import struct
from winserial import i2c
import logging
import smbus


class Result(graphene.ObjectType):
    errors = graphene.List(graphene.String)
    success = graphene.Boolean()


class Accelerometer(graphene.ObjectType):
    x = graphene.Float()
    y = graphene.Float()
    z = graphene.Float()


class Magnetometer(graphene.ObjectType):
    x = graphene.Float()
    y = graphene.Float()
    z = graphene.Float()


class Gyroscope(graphene.ObjectType):
    x = graphene.Float()
    y = graphene.Float()
    z = graphene.Float()


class Quaternion(graphene.ObjectType):
    q1 = graphene.Float()
    q2 = graphene.Float()
    q3 = graphene.Float()
    q4 = graphene.Float()


class AccelerometerResult(graphene.ObjectType):

    errors = graphene.List(graphene.String)

    success = graphene.Boolean()

    accData = graphene.Field(Accelerometer)


class MagnetometerResult(graphene.ObjectType):

    errors = graphene.List(graphene.String)

    success = graphene.Boolean()

    magData = graphene.Field(Magnetometer)


class GyroscopeResult(graphene.ObjectType):

    errors = graphene.List(graphene.String)

    success = graphene.Boolean()

    gyrData = graphene.Field(Gyroscope)


class QuaternionResult(graphene.ObjectType):

    errors = graphene.List(graphene.String)

    success = graphene.Boolean()

    quaData = graphene.Field(Quaternion)
```

schema.py

```
#!/usr/bin/env python3


__author__ = "Jon Grebe"

__version__ = "0.1.0"
```

```
__license__ = "MIT"


import graphene
from .models import *
from winapi import imu


_imu = imu.IMU(bus=2)


'''
type Query {
    mag(): MagResult
    acc(): AccResult
    gyr(): GyrResult
    qua(): QuaResult
}
'''
class Query(graphene.ObjectType):


    '''
    query {
        mag
    }
    '''
    mag = graphene.Field(MagnetometerResult)
    def resolve_mag(self, info):
        success, errors, x, y, z = _imu.mag()
        return MagnetometerResult(success=success, errors=errors, magData=Magnetometer(x=x, y=y, z=z))
    '''
    query {
        acc
    }
    '''
    acc = graphene.Field(AccelerometerResult)
    def resolve_acc(self, info):
        success, errors, x, y, z = _imu.acc()
        return AccelerometerResult(success=success, errors=errors,
accData=Accelerometer(x=x, y=y, z=z))

    '''
    query {
        gyr
    }
    '''
    gyr = graphene.Field(GyroscopeResult)
    def resolve_gyr(self, info):
        success, errors, x, y, z = _imu.gyr()
        return GyroscopeResult(success=success, errors=errors,
gyrData=Gyroscope(x=x, y=y, z=z))

    '''
    query {
        qua
    }
    '''
    qua = graphene.Field(QuaternionResult)
    def resolve_qua(self, info):
        success, errors, q1, q2, q3, q4 = _imu.qua()
        return QuaternionResult(success=success, errors=errors,
quaData=Quaternion(q1=q1,q2=q2,q3=q3,q4=q4))


schema = graphene.Schema(query=Query)
```

RTC SERVICE

service.py

```
#!/usr/bin/env python3

"""
Boilerplate service code which reads the config file and starts up the
GraphQL/HTTP endpoint. (should not need to much modification)
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import logging

from service import schema
from logging.handlers import SysLogHandler
import sys
```

```
#import toml


#print("hello")
from kubos_service.config import Config
config = Config("rtc-service")
#print(toml.dumps(config))


# Setup logging
logger = logging.getLogger("rtc-service")
logger.setLevel(logging.DEBUG)
handler = SysLogHandler(address='/dev/log', facility=SysLogHandler.LOG_DAEMON)
formatter = logging.Formatter('rtc-service: %(message)s')
handler.formatter = formatter
logger.addHandler(handler)


# Set up a handler for logging to stdout
stdout = logging.StreamHandler(stream=sys.stdout)
stdout.setFormatter(formatter)
logger.addHandler(stdout)


from kubos_service import http_service
# Start an http service
http_service.start(config, schema.schema)


#from kubos_service import udp_service


# Start a udp service with optional context
# udp_service.start(config, schema, {'bus': '/dev/ttyS3'})
```

```
    # Start a udp service

    #udp_service.start(logger, config, schema)
```

app.py

```
#!/usr/bin/env python3

"""

Boilerplate Flask setup for service application (should not be modified)

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


from flask import Flask

from flask_graphql import GraphQLView

from .schema import schema




def create_app():

    """

    Creates graphql and graphiql endpoints

    """


    app = Flask(__name__)

    app.debug = True
    app.add_url_rule(
        '/',
        view_func=GraphQLView.as_view(
            'graphql',
            schema=schema,
            graphiql=False
        )
    )

    app.add_url_rule(
        '/graphiql',
        view_func=GraphQLView.as_view(
            'graphiql',
            schema=schema,
            graphiql=True
        )
    )

    return app
```

models.py

```
#!/usr/bin/env python3

"""
Graphene ObjectType classes for subsystem modeling.
"""
```

```
__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import graphene


class Result(graphene.ObjectType):

    errors = graphene.List(graphene.String)

    success = graphene.Boolean()


class RTCDateTime(graphene.ObjectType):

    datetime = graphene.types.datetime.DateTime()

    result = graphene.Field(Result)
```

schema.py

```
#!/usr/bin/env
python3


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import graphene

from .models import *

from obcapi import DS3231


_rtc = DS3231.DS3231(2)
```

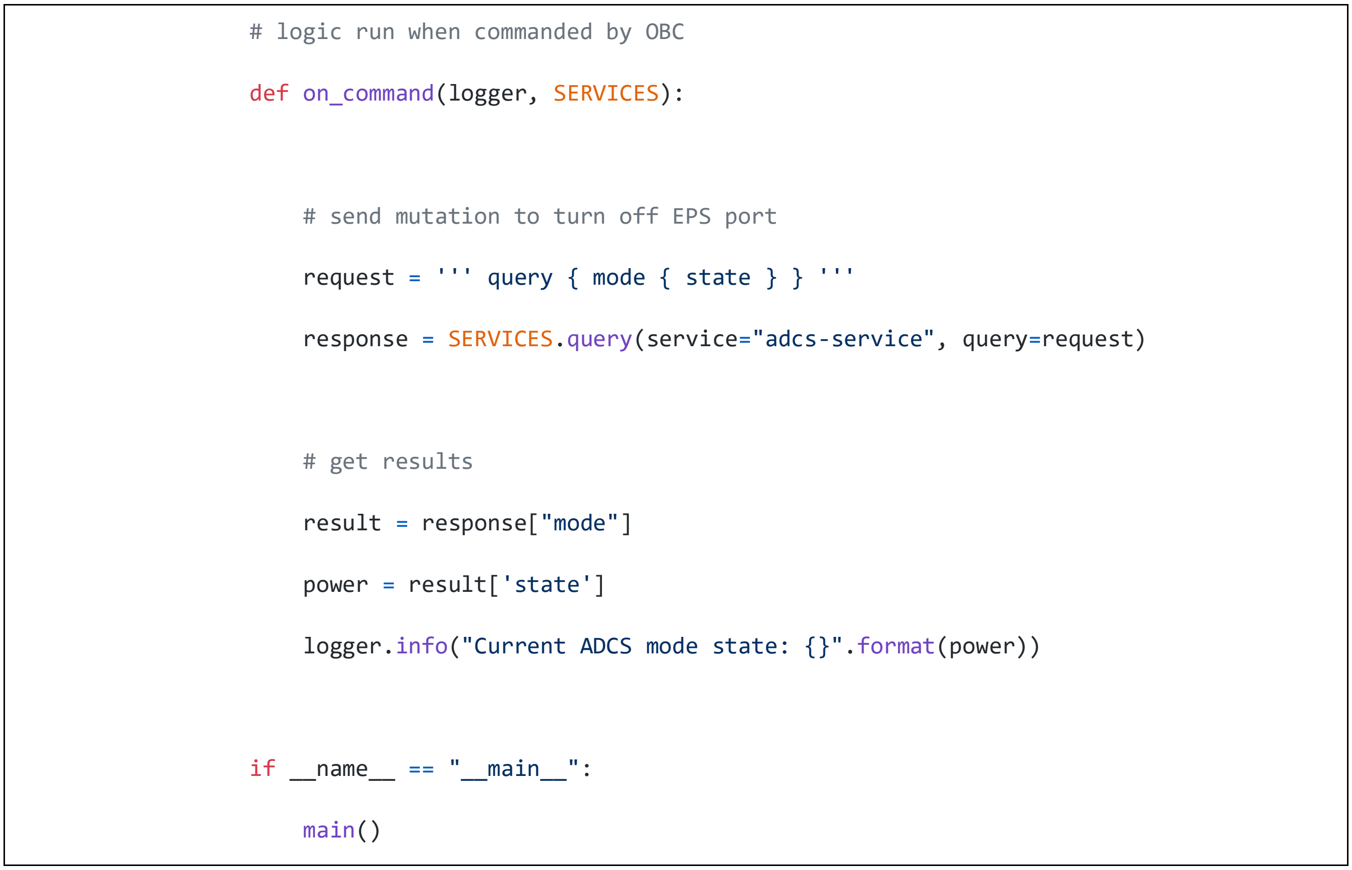
```
class Query(graphene.ObjectType):

    '''
    query {
        ping
    }
    '''
    ping = graphene.String()
    def resolve_ping(self, info):
        return _rtc.ping()

    '''
    type Query {
        dateTime {
            datetime
                result {
                    success
                    errors
            }
        }
    }
    '''
    dateTime = graphene.Field(RTCDateTime)
    def resolve_dateTime(self, info):
        # should send hardware a ping and expect a pong back
        _datetime = _rtc.datetime()
        # set success to true and error to nothing as default for now
        _success = True
        _errors = []

        # return results
        return RTCDateTime(result=Result(success=_success, errors=_errors),
datetime=_datetime)

schema = graphene.Schema(query=Query)
```

KUBOS APPLICATIONS

ADCS Applications

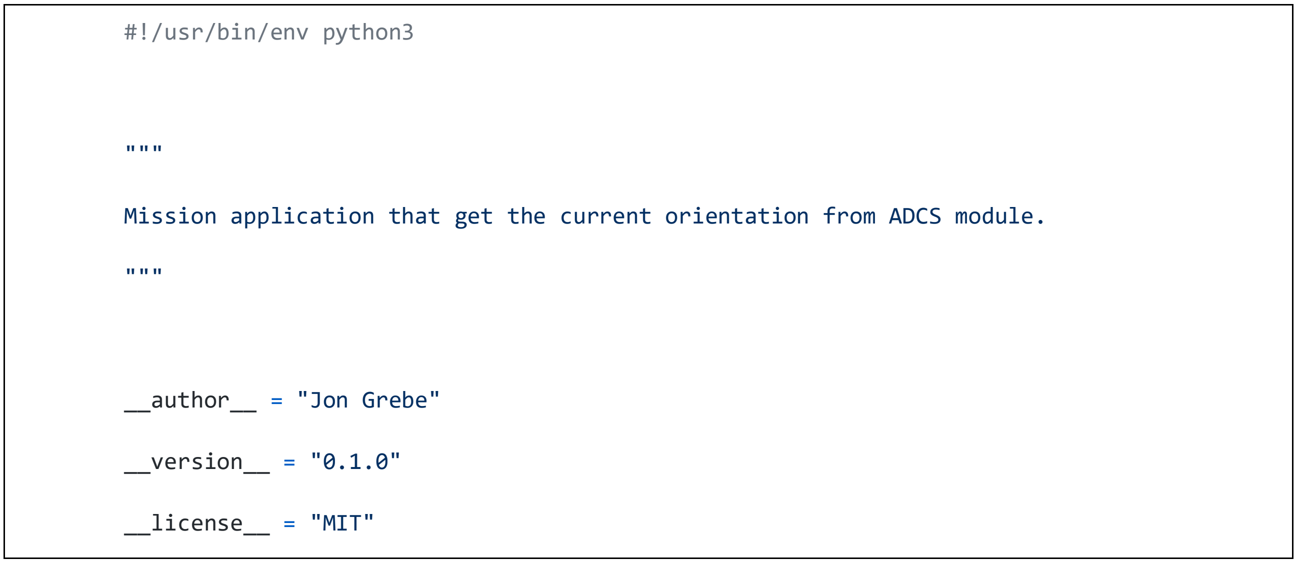
get-adcs-mode.py

```
#!/usr/bin/env
python3

"""
Mission application that get the current power mode of ADCS (IDLE, DETUMBLE,
POINTING).
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():
    logger = app_api.logging_setup("get-adcs-mode")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass
```



```python
# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send mutation to turn off EPS port
    request = ''' query { mode { state } } '''
    response = SERVICES.query(service="adcs-service", query=request)

    # get results
    result = response["mode"]
    power = result['state']
    logger.info("Current ADCS mode state: {}".format(power))

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

get-adcs-orientation.py



```python
#!/usr/bin/env python3

"""
Mission application that get the current orientation from ADCS module.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"
```

```python
import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("get-adcs-orientation")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send mutation to turn off EPS port
    request = ''' query { orientation { x y z yaw pitch roll } } '''
    response = SERVICES.query(service="adcs-service", query=request)

    # get results
    result = response["orientation"]

    x = result['x']
    y = result['y']
    z = result['z']
    yaw = result['yaw']
    pitch = result['pitch']
    roll = result['roll']

    logger.info("Current orientation ({},{},{},{},{},{})".format(x,y,z,yaw,pitch,roll))

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

get-adcs-mode.py

```
#!/usr/bin/env
python3

"""

Mission application that get the current power state of ADCS (ON, OFF, RESET).

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import app_api

import argparse

import sys


def main():


    logger = app_api.logging_setup("get-adcs-power")


    # parse arguments for config file and run type

    parser = argparse.ArgumentParser()

    parser.add_argument('--run', '-r', nargs=1)

    parser.add_argument('--config', '-c', nargs=1)

    args = parser.parse_args()


    if args.config is not None:

        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send mutation to turn off EPS port
    request = ''' query { power { state } } '''
    response = SERVICES.query(service="adcs-service", query=request)

    # get results
    result = response["power"]
    power = result['state']
    logger.info("Current ADCS power state: {}".format(power))


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

get-adcs-spin.py

```
#!/usr/bin/env python3

"""
Mission application that get the current spin from ADCS module.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("get-adcs-spin")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send mutation to turn off EPS port
    request = ''' query { spin { x y z } } '''
    response = SERVICES.query(service="adcs-service", query=request)

    # get results
    result = response["spin"]

    x = result['x']
    y = result['y']
    z = result['z']

    logger.info("Current spin: ({},{},{})".format(x,y,z))

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

get-adcs-telemetry.py

```
#!/usr/bin/env
python3
```

```
"""
Mission application that collects the current telemetry from ADCS module.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
```

```python
import sys


def main():


    logger = app_api.logging_setup("get-adcs-spin")


    # parse arguments for config file and run type

    parser = argparse.ArgumentParser()

    parser.add_argument('--run', '-r', nargs=1)

    parser.add_argument('--config', '-c', nargs=1)

    args = parser.parse_args()


    if args.config is not None:

        # use user config file if specified in command line

        SERVICES = app_api.Services(args.config[0])

    else:

        # else use default global config file

        SERVICES = app_api.Services("/etc/kubos-config.toml")


    # run app onboot or oncommand logic

    if args.run is not None:

        if args.run[0] == 'OnBoot':

            on_boot(logger, SERVICES)

        elif args.run[0] == 'OnCommand':

            on_command(logger, SERVICES)

    else:

        on_command(logger, SERVICES)
```

```
# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass


# logic run when commanded by OBC
def on_command(logger, SERVICES):


    # send mutation to turn off EPS port
    request = '''
    query {
        telemetry {
            orientation { x y z yaw pitch roll }
            spin { x y z }
            mode { state }
            power { state }
        }
    } '''
    response = SERVICES.query(service="adcs-service", query=request)


    # get results
    result = response["telemetry"]


    spin = result['spin']
    x_spin = spin['x']
    y_spin = spin['y']
    z_spin = spin['z']


    orientation = result['orientation']
    x = orientation['x']

    y = orientation['y']

    z = orientation['z']

    yaw = orientation['yaw']

    pitch = orientation['pitch']

    roll = orientation['roll']


    mode = result['mode']

    mode = mode['state']


    power = result['power']

    power = power['state']


    logger.info("Got ADCS telemetry:\n power={} mode={}\n
orientation=({},{},{},{},{}.{})\n spin=({},{},{})".format(

                power,

                mode,

                x,y,z,yaw,pitch,roll,

                x_spin,y_spin,z_spin

    ))


    logger.info("Storing telemetry in ADCS database...")


    # timestamp is optional and defaults to current system time

    request = '''

    mutation {

        insertBulk(entries: [

            { subsystem: "ADCS", parameter: "mode", value: "%s" },

            { subsystem: "ADCS", parameter: "power", value: "%s" },
            { subsystem: "ADCS", parameter: "x", value: "%s" },
            { subsystem: "ADCS", parameter: "y", value: "%s" },
            { subsystem: "ADCS", parameter: "z", value: "%s" },
            { subsystem: "ADCS", parameter: "yaw", value: "%s" },
            { subsystem: "ADCS", parameter: "pitch", value: "%s" },
            { subsystem: "ADCS", parameter: "roll", value: "%s" },
            { subsystem: "ADCS", parameter: "spin_x", value: "%s" },
            { subsystem: "ADCS", parameter: "spin_y", value: "%s" },
            { subsystem: "ADCS", parameter: "spin_z", value: "%s" }
        ])
        {
            success
            errors
        }
    }
    ''' % (mode,power,x,y,z,yaw,pitch,roll,x_spin,y_spin,z_spin)
    response = SERVICES.query(service="telemetry-service", query=request)

    # get results
    result = response["insertBulk"]
    success = result["success"]
    errors = result["errors"]

    if success:
        logger.info("Logged telemetry in database.")
    else:
        logger.warn("Unable to log telemetry to database: {}".format(errors))
```

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

ping-adcs.py

```
#!/usr/bin/env python3

"""
Mission application that pings the ADCS service.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("ping-adcs")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()
    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    request = ''' query { ping } '''
    response = SERVICES.query(service="adcs-service", query=request)

    # get results
    result = response["ping"]

    # check results
    if "pong" == result:
        logger.info("Success ping to ADCS service")
    else:
        logger.warn("Unable to ping ADCS service")

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

set-adcs-mode.py

```
#!/usr/bin/env
python3

"""
Mission application that sets the mode of the ADCS module (IDLE, DETUMBLE,
POINTING).
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():
```

```python
    logger = app_api.logging_setup("set-adcs-mode")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    parser.add_argument('mode')
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES, args.mode)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES, args.mode)
    else:
        on_command(logger, SERVICES, args.mode)

# logic run for application on OBC boot
def on_boot(logger, SERVICES, mode):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES, mode):

    # send mutation to turn off EPS port
    request = '''
    mutation {
        setMode(setModeInput: {mode: %s}) {
            errors
            success
        }
    }
    ''' % (mode)
    response = SERVICES.query(service="adcs-service", query=request)

    # get results
    result = response["setMode"]
    success = result["success"]
    errors = result["errors"]

    # check results
    if success:
        logger.info("Set ADCS mode={}.".format(mode))
    else:
        logger.warn("Unable to set ADCS mode={}: {}.".format(mode, errors))

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

set-adcs-power.py

```
#!/usr/bin/env python3

"""
Mission application that sets the power state of the ADCS module (ON, OFF, RESET).
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("set-adcs-power")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    parser.add_argument('power')
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES, args.power)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES, args.power)
    else:
        on_command(logger, SERVICES, args.power)

# logic run for application on OBC boot
def on_boot(logger, SERVICES, power):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES, power):

    # send mutation to turn off EPS port
    request = '''
    mutation {
        controlPower(controlPowerInput: {power: %s}) {
            success
            errors
        }
    }
    ''' % (power)
    response = SERVICES.query(service="adcs-service", query=request)

    # get results
    result = response["controlPower"]
    success = result["success"]
    errors = result["errors"]

    # check results
    if success:
        logger.info("Set ADCS power={}.".format(power))
    else:
        logger.warn("Unable to set ADCS power={}: {}.".format(power, errors))

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

EPS Applications

get-eps-telemetry.py

```
#!/usr/bin/env python3
```

```
"""
Mission application that queries all telemetry from EPS module.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("get-eps-telemetry")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")
    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # collect EPS telemetry
    try:
        # send mutation to turn off EPS port
        request = '''
        query {
            telemetry {
                power {
                    power1
                    power2
                    power3
                }
                battery

            }

        }

        '''

        response = SERVICES.query(service="eps-service", query=request)


        # get results

        telemetry = response["telemetry"]


        power = telemetry["power"]

        power1 = power["power1"]

        power2 = power["power2"]

        power3 = power["power3"]

        battery = telemetry["battery"]


        telemetry = [("power1", power1), ("power2", power2), ("power3", power3),
("battery", battery)]

        logger.info("Got EPS telemetry - Current power states: 1 = {} 2 = {} 3 =
{} and battery level = {}. Storing in database...".format(power1, power2, power3,
battery))


    except Exception as e:

        logger.error("Error collecting EPS telemetry:
{}:{}".format(type(e).__name__,str(e)))

        sys.exit(1)


    # add EPS telemetry to database

    try:

        # timestamp is optional and defaults to current system time

        subsystem = "EPS"
        for item in telemetry:
            name = item[0]
            value = item[1]

            request = '''
            mutation {
                insert(subsystem: "%s", parameter: "%s", value: "%s") {
                    success
                    errors
                }
            }
            ''' % (subsystem, name, value)
            response = SERVICES.query(service="telemetry-service", query=request)

            # get results
            result = response["insert"]
            success = result["success"]
            errors = result["errors"]

            if success:
                logger.info("Logged telemetry {}:{}={} in
database.".format(subsystem, name, value))
            else:
                logger.error("Unable to log telemetry {}:{}={} in database:
{}.".format(subsystem, name, value, errors))

    except Exception as e:
        logger.error("Error logging information in telemetry database:
{}:{}".format(type(e).__name__,str(e)))
        sys.exit(1)


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

query-battery-level.py

```
#!/usr/bin/env python3

"""
Mission application that queries the battery level (percentage) on EPS module.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("query-battery-level")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send mutation to turn off EPS port
    request = '''
    query {
        battery
    }
    '''
    response = SERVICES.query(service="eps-service", query=request)

    # get results
    result = response["battery"]

    # check results
    logger.info("Current battery level: {}.".format(result))

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

query-power-state.py

```
#!/usr/bin/env
python3
```

```
"""
Mission application that queries the power state of the ports on EPS module.
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"
```

```python
import app_api
import argparse
import sys


def main():

    logger = app_api.logging_setup("query-power-state")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send mutation to turn off EPS port
    request = '''
    query {
        power {
            power1
            power2
            power3
        }
    }
    '''
    response = SERVICES.query(service="eps-service", query=request)

    # get results
    result = response["power"]
    power1 = result["power1"]
    power2 = result["power2"]
    power3 = result["power3"]

    # check results
    logger.info("Current power states: 1 = {} 2 = {} 3 = {}.".format(power1,
power2, power3))


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

turn-port-off.py

```
#!/usr/bin/env
python3
```

```
"""
Mission application that turns off a output port on EPS module.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("turn-port-off")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)

    parser.add_argument('port', type=int)

    args = parser.parse_args()


    if args.config is not None:

        # use user config file if specified in command line

        SERVICES = app_api.Services(args.config[0])

    else:

        # else use default global config file

        SERVICES = app_api.Services("/etc/kubos-config.toml")


    # run app onboot or oncommand logic

    if args.run is not None:

        if args.run[0] == 'OnBoot':

            on_boot(logger, SERVICES)

        elif args.run[0] == 'OnCommand':

            on_command(logger, SERVICES, args.port)

    else:

        on_command(logger, SERVICES, args.port)


# logic run for application on OBC boot

def on_boot(logger, SERVICES):

    pass


# logic run when commanded by OBC

def on_command(logger, SERVICES, port):


    # send mutation to turn off EPS port
    request = '''
    mutation {
        controlPort(controlPortInput: {
                    power: OFF
                    port: PORT%d })
        {
        errors
        success
        }
    }
    ''' % (port)
    response = SERVICES.query(service="eps-service", query=request)

    # get results
    result = response["controlPort"]
    success = result["success"]
    errors = result["errors"]

    # check results
    if success:
        logger.info("Turned off port {}.".format(port))
    else:
        logger.warn("Uable to turn off port {}: {}.".format(port, errors))

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

turn-port-on.py

```
#!/usr/bin/env
python3

"""

Mission application that turns on a output port on EPS module.

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import app_api

import argparse

import sys


def main():


    logger = app_api.logging_setup("turn-port-on")


    # parse arguments for config file and run type

    parser = argparse.ArgumentParser()

    parser.add_argument('--run', '-r', nargs=1)

    parser.add_argument('--config', '-c', nargs=1)

    parser.add_argument('port', type=int)

    args = parser.parse_args()


    if args.config is not None:

        # use user config file if specified in command line

        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES, args.port)
    else:
        on_command(logger, SERVICES, args.port)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES, port):

    # send mutation to turn on EPS port
    request = '''
    mutation {
        controlPort(controlPortInput: {
                    power: ON
                    port: PORT%d })
        {
        errors
        success
        }
    }
    ''' % (port)
    response = SERVICES.query(service="eps-service", query=request)

    # get results
    result = response["controlPort"]
    success = result["success"]
    errors = result["errors"]

    # check results
    if success:
        logger.info("Turned on port {}.".format(port))
    else:
        logger.warn("Uable to turn on port {}: {}.".format(port, errors))

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

PAYLOAD Applications

payload-image-capture-transfer.py

```
#!/usr/bin/env python3

"""
Mission application that requests an image capture and then transfers it from payload subsystem.
```

```
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time

def main():

    logger = app_api.logging_setup("payload-image-capture-transfer")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/home/kubos/kubos/local_config.toml")
    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Starting image capture..")

    time_1 = time.time()

    # send request for image capture
    request = '''
    mutation {
        imageCapture {
            success
            errors
        }
    }
    '''
    response = SERVICES.query(service="payload-service", query=request)

    time_2 = time.time()

    # get results
    result = response["imageCapture"]
    success = result["success"]
    errors = result["errors"]

    # check results
    if success:
        logger.info("Payload completed successful image capture.")
    else:
        logger.warn("Unsuccessful image capture request to payload: {}.".format(errors))
        sys.exit(1)

    time.sleep(1)

    logger.info("Starting image transfer..")

    time_3 = time.time()

    # send request for image transfer
    request = '''
    mutation {
        imageTransfer {
            success
            errors
        }
    }
    '''
    response = SERVICES.query(service="payload-service", query=request, timeout=100)

    time_4 = time.time()

    # get results
    result = response["imageTransfer"]
    success = result["success"]
    errors = result["errors"]

    # check results
    if success:
        logger.warn("Successful image transfer with payload.")
    else:
        logger.warn("Unable to complete image transfer with payload: {}.".format(errors))
        sys.exit(1)

    # debug
    logger.debug("Time for image capture: {}".format(time_2 - time_1))
    logger.debug("Time for image transfer: {}".format(time_4 - time_3))

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

payload-image-capture.py

```
#!/usr/bin/env
python3

"""
Mission application that requests an image capture from payload subsystem.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("payload-image-capture")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/home/kubos/kubos/local_config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Starting image capture..")

    # send request for image capture
    request = '''
    mutation {
        imageCapture {
            success
            errors
        }
    }
    '''
    response = SERVICES.query(service="payload-service", query=request)

    # get results
    result = response["imageCapture"]
    success = result["success"]
    errors = result["errors"]

    # check results
    if success:
        logger.info("Payload completed successful image capture.")
    else:
        logger.warn("Unsuccessful image capture request to payload: {}.".format(errors))


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

payload-image-transfer.py

```
#!/usr/bin/env python3

"""
Mission application that requests an image transfer from payload subsystem to OBC.
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
```

```python
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("payload-image-transfer")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/home/kubos/kubos/local_config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send request for image capture
    request = '''
    mutation {
        imageTransfer {
            success
            errors
        }
    }
    '''
    response = SERVICES.query(service="payload-service", query=request)

    # get results
    result = response["imageTransfer"]
    success = result["success"]
    errors = result["errors"]

    # check results
    if success:

        logger.warn("Successful image transfer with payload.")

    else:

        logger.warn("Unable to complete image transfer with payload:
{}.".format(errors))

        sys.exit(1)



if __name__ == "__main__":

    main()
```

payload-ping.py

```
#!/usr/bin/env python3



"""

Mission application that pings the payload subsystem to check for successful connection.

"""



__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"



import app_api

import argparse

import sys



def main():
```

```python
    logger = app_api.logging_setup("payload-ping")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass
```

```
# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Starting nominal operation for payload subsystem...")

    # pinging payload subsystem
    request = ''' {
        ping
    }
    '''
    response = SERVICES.query(service="payload-service", query=request)

    # get results
    result = response["ping"]

    # check results
    if "pong"==result:
        logger.info("Successful ping connection to payload.")
    else:
        logger.warn("Unsuccessful ping connection to payload: {}.".format(errors))

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

RADIO Applications

radio-image-transfer.py

```
#!/usr/bin/env python3
```

```
"""
Mission application that transfers selfie-sat image to ground station using
radio.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

from obcapi import radio

def main():

    logger = app_api.logging_setup("radio-ping")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    filename = "/home/kubos/image.jpg"

    logger.info("Sending image {} to ground station...".format(filename))

    r = radio.RADIO()

    r.downlink_image(filename)
```

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

radio-ping.py

```
#!/usr/bin/env python3

"""
Mission application that sends a ping to the ground station over radio.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("radio-ping")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()
```

```python
    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Sending ping to ground station...")

    # send request for image capture
    request = '''
    query {
        ping
    }
    '''
    response = SERVICES.query(service="radio-service", query=request)


    # get results
    result = response["ping"]
    success = result["success"]
    errors = result["errors"]


    # check results
    if success:
        logger.info("SUCCESS: Sent ping to ground station.")
    else:
        logger.warn("ERROR sending ping to ground station: {}.".format(errors))


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

radio-read.py

```
#!/usr/bin/env python3


"""
Mission application that sends reads incoming messages from the ground station over radio.
"""


__author__ = "Jon Grebe"
```

```python
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time

def main():

    logger = app_api.logging_setup("radio-read")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Reading messages from ground station...")

    while 1:
        request = '''
        query {
            read {
                result {
                    success
                    errors
                }
                message
            }
        }
        '''
        response = SERVICES.query(service="radio-service", query=request)
        # get results
        read = response["read"]
        result = read["result"]
        success = result["success"]
        errors = result["errors"]
        message = read["message"]

        # check results
        if success:
            if message:
                logger.info("SUCCESS: Got message from ground station:
{}".format(message))
            else:
                logger.info("Did not get anything from ground station. Trying
again...")
        else:
            logger.warn("ERROR: Trying to read from ground station:
{}.".format(errors))

        time.sleep(1)

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

DEPLOYMENT Applications

deployment-app.py

```
#!/usr/bin/env python3


"""

Deployment application that handles the deployment sequence. The Deployment

sequence is made of 4 steps:

    1) Keeping track of hold time required by launch provider

    2) Deployment of deployables (solar panels, antenna, etc.)

    3) Powering on radio and configuring appropriately for initial contact

    4) Detumbling and stabilization of spacecraft

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import argparse

import app_api

import sys


def on_boot(logger):

    pass


def on_command(logger):

    pass


    '''''''''''''''''''''' STEP 1 - HOLD TIME TRACKING '''''''''''''''''''

    Hold time tracking done by U-boot environment variables. Two variables are used:

        1) deployed: boolean True if satellite deployment already complete
        2) deploy_start: string in seconds since unix epoch
    (The U-Boot environment is a block of memory that is kept on persistent storage
    and copied to RAM when U-Boot starts. It is used to store environment variables
    which can be used to configure the system.)
    '''
        # set system time from real-time clock from OBC

        # if (deployed)
        #   complete recurring boot tasks
        # else
        #   check "deploy_start" has a value
        #   yes:
        #       resume from "deploy_start" time
        #   no:
        #       set "deploy_start"
        #       begin timer
        #   once timer ends...
        #   check deployment tasks
        #   set "deployed" to True if successful

def main():

    logger = app_api.logging_setup("deployment-app")

    parser = argparse.ArgumentParser()

    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    parser.add_argument('cmd_args', nargs='*')

    args = parser.parse_args()

    if args.config is not None:
        global SERVICES
        SERVICES = app_api.Services(args.config[0])
    else:
        SERVICES = app_api.Services()

    if args.run[0] == 'OnBoot':
        on_boot(logger)
    elif args.run[0] == 'OnCommand':
        on_command(logger)
    else:
        logger.error("Unknown run level specified")
        sys.exit(1)

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

HEALTH Applications

health-mem-check.py

```
#!/usr/bin/env python3

"""
```

```
Checks current memory usage. If memory usage more than certain amount, enter
critical mode
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"


import app_api
import argparse
import sys
import time


def main():

    logger = app_api.logging_setup("health-mem-check")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()
    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    # retrieve memory telemetry from database
    try:
        '''
        query {
            telemetry(timestampGe: Float, timestampLe: Float, subsystem: String, parameter: String, parameters: [String], limit: Integer): [{
                timestamp
                subsystem
                parameter
                value
            }]
        }
        timestampGe = return entries with timestamps on or after given value
        timestampLe = return entries with timestamps on or before given value

        subsystem = return entries that match subsystem name

        parameter = return entries which match given parameter name (mutually
exclusive with parameters)

        parameters = returns entries which match given parameter names (mutually
exclusive with parameter)

        limit = return only first n entries found

        '''



        request = '''

        {

            telemetry(parameter: "free_memory_percentage") {

                timestamp

                subsystem

                parameter

                value

            }

        }

        '''

        response = SERVICES.query(service="telemetry-service", query=request)

        # get results

        result = response["telemetry"]

        for item in result:

            timestamp = item["timestamp"]

            subsystem = item["subsystem"]

            parameter = item["parameter"]

            value = item["value"]
            #logger.debug("Got telemetry:{} | timestamp:{} | subsystem:{} |
value:{}".format(parameter, timestamp, subsystem, value))

            threshold = 10.0
            if (float(value) < threshold):
                logger.error("Available memory: {} at {} below threshold: {}.
Enter critical mode.".format(value, timestamp, threshold))

                # send mutation to activate critical mode
                mode = "critical"
                request = '''
                mutation {
                    activateMode(name: "%s") {
                        success
                        errors
                    }
                }''' % (mode)
                response = SERVICES.query(service="scheduler-service",
query=request)

                # get results
                response = response["activateMode"]
                success = response["success"]
                errors = response["errors"]

                if success:
                    logger.info("Activtated mode named: {}.".format(mode))
                else:
                    logger.warning("Could not activate {} mode: {}.".format(mode,
errors))
```

```
                break

    except Exception as e:
        logger.error("Unable to retrieve memory telemetry from database: {}{}".format(type(e).__name__,str(e)))

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

health-mem-query.py

```
#!/usr/bin/env python3

"""
Mission application that retrieves current memory usage.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time

def main():
    logger = app_api.logging_setup("health-mem-query")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass
```

```
# logic run when commanded by OBC
def on_command(logger, SERVICES):
    # get current memory usage
    try:
        logger.info("Getting current memory usage...")

        # total = total usable RAM
        # free = toal amount of free memory (includes lowFree)
        # available = estimate how much memory is available for starting new
applications, without swapping
        # lowFree = amount of free memory which can be used by the kernel
        request = '''
        {
            memInfo {
                total
                free
            }
        }
        '''
        response = SERVICES.query(service="monitor-service", query=request)

        # get results
        result = response["memInfo"]
        total = int(result["total"])
        free = int(result["free"])

        free_memory_percentage = round(free/total * 100, 1)
        logger.info("Total free memory available:
{}%".format(free_memory_percentage))

    except Exception as e:
        logger.error("Error retrieving memory information from monitor service:
{}".format(str(e)))
        sys.exit(1)

    # add memory telemetry to database
    try:
        # timestamp is optional and defaults to current system time
        request = '''
        mutation {
            insert(subsystem: "CDH", parameter: "free_memory_percentage", value:
"%s") {
                success
                errors
            }
        }
        ''' % (free_memory_percentage)
        response = SERVICES.query(service="telemetry-service", query=request)

        # get results
        result = response["insert"]
        success = result["success"]
        errors = result["errors"]

        if success:
            logger.info("Logged memory telemetry in database.")
        else:
            logger.warn("Unable to log memory telemetry to database: {}".format(errors))


    except Exception as e:
        logger.error("Error logging memory information in telemetry database: {}{}".format(type(e).__name__,str(e)))


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

health-ping-service.py

```
#!/usr/bin/env python3

"""
Health mission application that pings every hardware service at intervals to notify of system issues/failures.
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"


import app_api
import argparse
import sys
import time
```

```
def main():

    logger = app_api.logging_setup("health-ping-services")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass


# logic run when commanded by OBC
def on_command(logger, SERVICES):


    # ping ADCS service
    try:
        request = ''' query { ping } '''
        response = SERVICES.query(service="adcs-service", query=request)
        if "pong" == response["ping"]:
            logger.info("Success pinging ADCS service")
        else:
            logger.warn("Unable to ping ADCS service")
    except Exception as e:
        logger.error("Exception trying to ping ADCS service:
{}{}".format(type(e).__name__,str(e)))


    # ping EPS service
    try:
        request = ''' query { ping } '''
        response = SERVICES.query(service="eps-service", query=request)
        if "pong" == response["ping"]:
            logger.info("Success pinginG EPS service")
        else:
            logger.warn("Unable to ping EPS service")
    except Exception as e:
        logger.error("Exception trying to ping EPS service:
{}{}".format(type(e).__name__,str(e)))
```

```
    # ping Payload service
    try:
        request = ''' query { ping } '''
        response = SERVICES.query(service="payload-service", query=request)
        if "pong" == response["ping"]:
            logger.info("Success pinging payload service")
        else:
            logger.warn("Unable to ping payload service")
    except Exception as e:
        logger.error("Exception trying to ping payload service:
{}{}".format(type(e).__name__,str(e)))


    # ping Radio service
    try:
        request = ''' query { ping } '''
        response = SERVICES.query(service="radio-service", query=request)
        if "pong" == response["ping"]:
            logger.info("Success pinging radio service")
        else:
            logger.warn("Unable to ping radio service")
    except Exception as e:
        logger.error("Exception trying to ping radio service:
{}{}".format(type(e).__name__,str(e)))


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

health-ps-query.py

```
#!/usr/bin/env
python3

"""
Mission application that pings the payload subsystem to check for successful
connection.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("payload-ping")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Starting nominal operation for payload subsystem...")

    '''
    {
        ps(pids: [Int!] = null): [
            {
                pid: Int!       - process ID
                uid: Int        - user ID who created the process
                gid: Int        - group ID who created the process
                usr: String     - username associated with UID
                grp: String     - group name associated with the GID
                state: String   - single character indicating process state
(refer to ps state code manual)
                ppid: Int       - process ID of process which started the process
                mem: Int        - virtual memory of process (bytes)
                rss: Int        - current number of pages process has in real
memory
                threads: Int    - number of threads in process
                cmd: String     - full command used to execute the process
(defaults to filename)
            }
        ]
    }
    '''


    # pinging payload subsystem


    request = '''
    {
        ps(pids: [30972]) {
            pid,
            state,
            ppid,
            threads,
            cmd
        }
    }
    '''
    response = SERVICES.query(service="monitor-service", query=request)


    # get results

    print(response)

    result = response["ps"][0]

    pid = result["pid"]

    state = result["state"]

    ppid = result["ppid"]

    threads = result["threads"]

    cmd = result["cmd"]


    logger.info("Got the following information about PID:{} | State:{} | CMD:{} |
Threads:{}".format(pid,state,cmd,threads))


if __name__ == "__main__":

    main()
```

IMU Applications

Imu-read-acc.py

```
#!/usr/bin/env
python3


"""

Mission application that reads accelerometer data from imu

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"
```

```
__license__ = "MIT"

import app_api
import argparse
import sys
import time

def main():

    logger = app_api.logging_setup("imu-read-acc")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    parser.add_argument('cmd_args', nargs='*')
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Querying acceleromter data from IMU...")

    try:
        request = '''
        {
            acc {
                success
                errors
                accData {
                    x
                    y
                    z
                }
            }
        }
        '''
        response = SERVICES.query(service="imu-service", query=request)
        result = response["acc"]
        success = result["success"]
        errors = result["errors"]
        accData = result["accData"]
        x = accData["x"]
        y = accData["y"]
        z = accData["z"]

        if success:
            logger.info('Acceleration (m/s^2): ({}, {}, {})'.format(x, y, z))
        else:
            logger.warn("Unable to retrieve accelerometer data:
{}".format(errors))

    except Exception as e:
        logger.warn("Unsuccessful getting data from accelerometer: {}-
{}".format(type(e).__name__,str(e)))

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

Imu-read-gyr.py

```
#!/usr/bin/env python3

"""
```

```
Mission application that reads gyroscope data from IMU
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time

def main():

    logger = app_api.logging_setup("imu-read-gyr")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    parser.add_argument('cmd_args', nargs='*')
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Querying gyroscope data from FXAS21002...")

    try:
        request = '''
        {
            gyr {
                success
                errors
                gyrData {
                    x
                    y
                    z
                }
            }
        }
        '''
        response = SERVICES.query(service="imu-service", query=request)
        result = response["gyr"]
        success = result["success"]
        errors = result["errors"]
        gyrData = result["gyrData"]
        x = gyrData["x"]
        y = gyrData["y"]
        z = gyrData["z"]

        if success:
            logger.info('Gyroscope (uTesla): ({}, {}, {})'.format(x, y, z))
        else:
            logger.warn("Unable to retrieve gyroscope data: {}".format(errors))

    except Exception as e:
        logger.warn("Unsuccessful getting data from gyroscope: {}-
{}".format(type(e).__name__,str(e)))

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

Imu-read-mag.py

```
#!/usr/bin/env python3

"""
Mission application that reads magnetometer data from IMU
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time

def main():

    logger = app_api.logging_setup("imu-read-mag")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    parser.add_argument('cmd_args', nargs='*')
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line

        SERVICES = app_api.Services(args.config[0])

    else:

        # else use default global config file

        SERVICES = app_api.Services()


    # run app onboot or oncommand logic

    if args.run is not None:

        if args.run[0] == 'OnBoot':

            on_boot(logger, SERVICES)

        elif args.run[0] == 'OnCommand':

            on_command(logger, SERVICES)

    else:

        on_command(logger, SERVICES)


# logic run for application on OBC boot

def on_boot(logger, SERVICES):

    pass


# logic run when commanded by OBC

def on_command(logger, SERVICES):

    logger.info("Querying magnetometer data from FXOS8700...")


    try:

        request = '''

        {

            mag {

                success
                errors
                magData {
                    x
                    y
                    z
                }
            }
        }
        '''
        response = SERVICES.query(service="imu-service", query=request)
        result = response["mag"]
        success = result["success"]
        errors = result["errors"]
        magData = result["magData"]
        x = magData["x"]
        y = magData["y"]
        z = magData["z"]

        if success:
            logger.info('Magnetometer (uTesla): ({}, {}, {})'.format(x, y, z))
        else:
            logger.warn("Unable to retrieve magnetometer data: {}".format(errors))

    except Exception as e:
        logger.warn("Unsuccessful getting data from magnetometer: {}-
{}".format(type(e).__name__,str(e)))

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

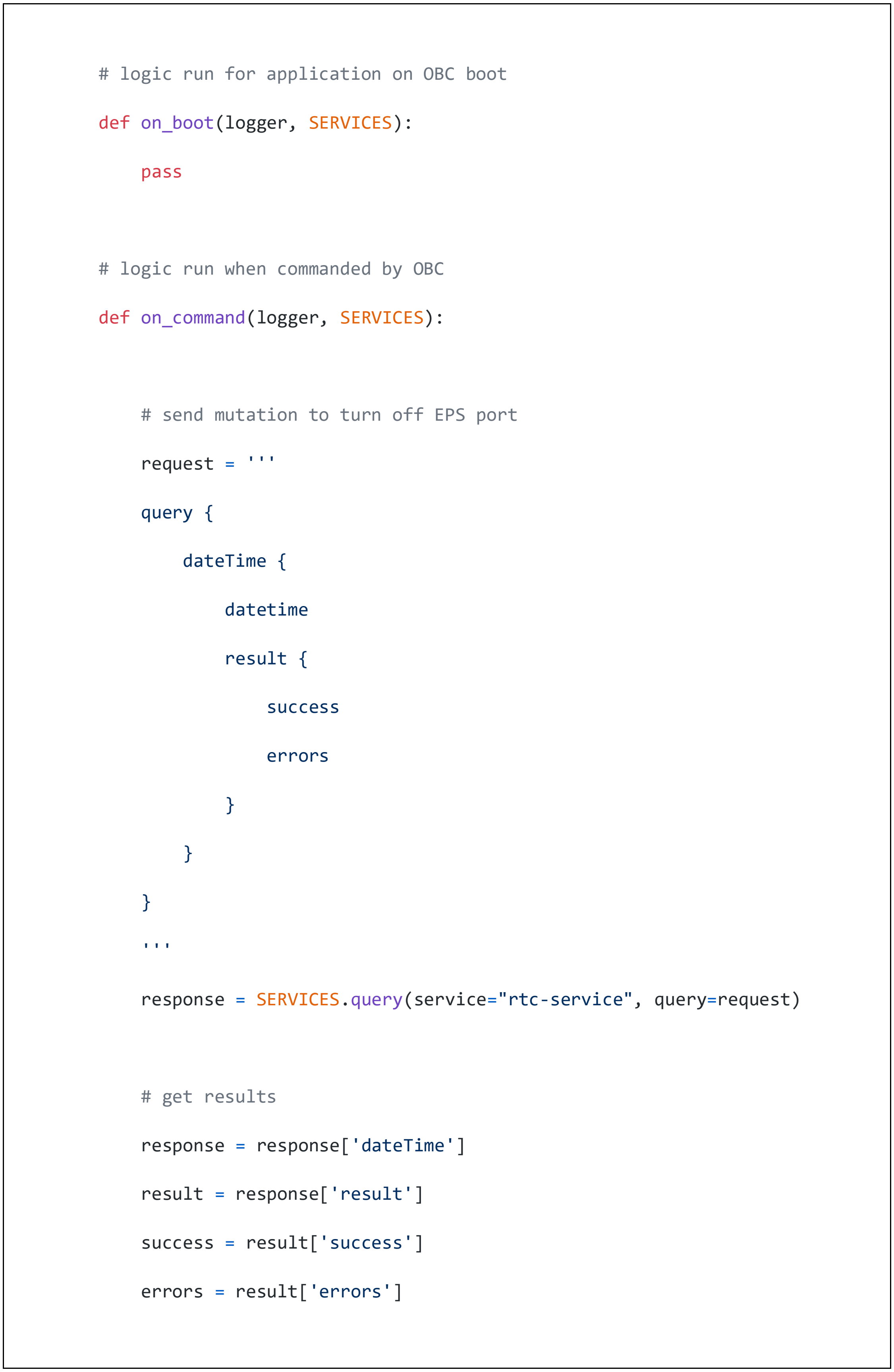
Imu-read-qua.py

```python
#!/usr/bin/env python3

"""
Mission application that reads all data from IMU and returns quaternion values
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time

import serial
serial = serial.Serial(
    port='/dev/ttyS0',
    baudrate=115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=1)

def main():
    logger = app_api.logging_setup("imu-read-qua")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    parser.add_argument('cmd_args', nargs='*')
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass


# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.info("Querying quaternion values from IMU...")
    while True:
        input("pause")
        try:
            request = '''
            {
                qua {
                    success
                    errors
                    quaData {
                        q1
                        q2
                        q3
                        q4
                    }
                }
            }
            '''
            response = SERVICES.query(service="imu-service", query=request)
            result = response["qua"]
            success = result["success"]
            errors = result["errors"]
            quaData = result["quaData"]
            q1 = quaData["q1"]
            q2 = quaData["q2"]
            q3 = quaData["q3"]
            q4 = quaData["q4"]

            if success:
                logger.info('Quaternion: ({}, {}, {}, {})'.format(q1,q2,q3,q4))
            else:
                logger.warning("Unable to retrieve quaternion data:
{}".format(errors))

            try:
                # open uart port
                serial.close()
                serial.open()

                if serial.isOpen():
                    message = (q1,q2,q3,q4)
                    # if uart port is open, try to send encoded string message
                    serial.write(str(message).encode('utf-8'))
                    serial.close()
                    logger.debug("UART port is open. Sent message:
{}".format(str(message)))
                else:
                    # if could not open uart port, return failure
                    serial.close()
                    logger.warn("Could not open serial port")

            # return failure if exception during write/encoding
            except Exception as e:
                logger.warn("Error sending message {} over uart port:
{}".format(str(message), str(e)))



        except Exception as e:
            logger.warning("Unsuccessful getting data from quaternion: {}-
{}".format(type(e).__name__,str(e)))


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

RTC Applications

get-system-time.py

```
#!/usr/bin/env python3


"""
Mission application that queries and prints current date and time from RTC
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"


import app_api
import argparse
import sys
import datetime

def main():

    logger = app_api.logging_setup("print-rtc-datetime")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)
```



```
# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass


# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send mutation to turn off EPS port
    request = '''
    query {
        dateTime {
            datetime
            result {
                success
                errors
            }
        }
    }
    '''
    response = SERVICES.query(service="rtc-service", query=request)

    # get results
    response = response['dateTime']
    result = response['result']
    success = result['success']
    errors = result['errors']
    # check results
    if success:
        datetime = response['datetime']
        logger.info("Got datetime from RTC: {}.".format(datetime))
    else:
        logger.warn("Unable to get datetime from RTC: {}.".format(errors))

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

set-system-time.py

```
#!/usr/bin/env
python3
```

```
"""
Mission application that queries time from RTC and sets BBB accordingly
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import os

def main():
    logger = app_api.logging_setup("set-system-time")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass
```

```
# logic run when commanded by OBC
def on_command(logger, SERVICES):

    # send mutation to turn off EPS port
    request = '''
    query {
        dateTime {
            datetime
            result {
                success
                errors
            }
        }
    }
    '''
    response = SERVICES.query(service="rtc-service", query=request)

    # get results
    response = response['dateTime']
    result = response['result']
    success = result['success']
    errors = result['errors']

    # check results
    try:
        if success:
            datetime = response['datetime']
            datetime = datetime.replace('T',' ')
            print(datetime)
            logger.info("Got datetime from RTC: {}. Setting system time...".format(datetime))
            os.system("date -s \"%s\"" % datetime)
        else:
            logger.warn("Unable to get datetime from RTC: {}.".format(errors))
    except Exception as e:
        logger.warn("Unable to set datetime from RTC: {}.".format(str(e)))


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

SCHEDULER Applications

activate-mode.py

```
#!/usr/bin/env python3

"""
Activate mode in scheduler.
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"


import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("activate-mode")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)
```

```python
# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass


# logic run when commanded by OBC
def on_command(logger, SERVICES):
    name = str(input("Enter mode name: "))


    # send mutation to activate mode
    request = '''
    mutation {
        activateMode(name: "%s") {
            success
            errors
        }
    }''' % (name)
    response = SERVICES.query(service="scheduler-service", query=request)


    # get results
    response = response["activateMode"]
    success = response["success"]
    errors = response["errors"]


    if success:
        logger.info("Activtated mode named: {}.".format(name))
    else:
        logger.warning("Could not activate {} mode: {}.".format(name, errors))
```

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

activate-safe-mode.py

```
#!/usr/bin/env python3

"""
Activate safe mode in scheduler.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("activate-safe-mode")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()


    if args.config is not None:

        # use user config file if specified in command line

        SERVICES = app_api.Services(args.config[0])

    else:

        # else use default global config file

        SERVICES = app_api.Services("/etc/kubos-config.toml")


    # run app onboot or oncommand logic

    if args.run is not None:

        if args.run[0] == 'OnBoot':

            on_boot(logger, SERVICES)

        elif args.run[0] == 'OnCommand':

            on_command(logger, SERVICES)

    else:

        on_command(logger, SERVICES)


# logic run for application on OBC boot

def on_boot(logger, SERVICES):

    pass


# logic run when commanded by OBC

def on_command(logger, SERVICES):


    # send mutation to activate safe mode

    request = '''

    mutation {
        safeMode {
            success
            errors
        }
    }'''
    response = SERVICES.query(service="scheduler-service", query=request)

    # get results
    response = response["safeMode"]
    success = response["success"]
    errors = response["errors"]

    if success:
        logger.info("Activated safe mode.")
    else:
        logger.warning("Could not activate safe mode: {}".format(errors))


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

check-active-mode.py

```
#!/usr/bin/env
python3

"""
Mission application that checks current active mode (safe, idle, critical).
"""
```

```python
__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("check-active-mode")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.debug("Requesting current active mode..")

    # send query to check current active mode
    request = '''
    query {
        activeMode {
            name
            path
            lastRevised
            active
            schedule {
                filename
                path
                timeImported
                tasks {
                    description
                    delay
                    time
                    period
                    app {
                        name
                        args
                        config
                    }
                }
            }
        }
    }
    '''
    response = SERVICES.query(service="scheduler-service", query=request)

    # get results
    response = response["activeMode"]
    name = response["name"]
    path = response["path"]
    lastRevised = response["lastRevised"]
    active = response["active"]
    schedule = response["schedule"]

    logger.info("Currently in {} mode.".format(name))

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

check-available-modes.py

```
#!/usr/bin/env python3

"""
Mission application that checks current available modes (safe, idle, critical).
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("check-available-modes")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    logger.debug("Requesting current available modes..")

    # send query for available modes
    request = '''
    query {
        availableModes {
            name
            path
            lastRevised
            active
            schedule {
                filename
                path
                timeImported
                tasks {
                    description
                    delay
                    time
                    period
                    app {
                        name
                        args
                        config
                    }
                }
            }
        }
    }
    '''
    response = SERVICES.query(service="scheduler-service", query=request)
    # get results
    response = response["availableModes"]
    modes = []
    for mode in response:
        name = mode["name"]
        path = mode["path"]
```

```
        lastRevised = mode["lastRevised"]

        active = mode["active"]

        schedule = mode["schedule"]


        modes.append(name)



    logger.info("Currently available modes: {}.".format(modes))


if __name__ == "__main__":

    main()
```

create-mode.py

```
#!/usr/bin/env python3


"""

Create mode in scheduler.

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import app_api

import argparse
import sys

def main():

    logger = app_api.logging_setup("create-mode")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)
```

```
# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass


# logic run when commanded by OBC
def on_command(logger, SERVICES):
    name = str(input("Enter mode name: "))

    # send mutation to create mode in scheduler
    request = '''
    mutation {
        createMode(name: "%s") {
            success
            errors
        }
    }''' % (name)
    response = SERVICES.query(service="scheduler-service", query=request)

    # get results
    response = response["createMode"]
    success = response["success"]
    errors = response["errors"]

    if success:
        logger.info("Created empty mode named: {}.".format(name))
    else:
        logger.warning("Could not create {} mode: {}.".format(name, errors))
```

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

Import-task-list.py

```
#!/usr/bin/env python3

"""
Add a task list to a mode in the scheduler. If the targeted mode is active, all
tasks in the task list will be immediately scheduled.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("import-task-list")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):

    path = "task_path"
    name = "task_name"
    mode = "mode_name"
```

```
    # send mutation to add task list to mode in scheduler

    request = '''

    mutation {

        importTaskList(path: "%s", name: "%s", mode: "%s") {

            success

            errors

        }

    }

    ''' % (path, name, mode)

    response = SERVICES.query(service="scheduler-service", query=request)


    # get results

    response = response["importTaskList"]

    success = response["success"]

    errors = response["errors"]


    if success:

        logger.info("Added task list {} at {} to mode {}.".format(name, path,
mode))

    else:

        logger.warning("Could not add task list {} at {} to mode {}:
{}.".format(name, path, mode, errors))



if __name__
```

remove-mode.py

```
#!/usr/bin/env python3

"""

Remove mode in scheduler.

"""


__author__ = "Jon Grebe"

__version__ = "0.1.0"

__license__ = "MIT"


import app_api

import argparse

import sys


def main():


    logger = app_api.logging_setup("remove-mode")


    # parse arguments for config file and run type

    parser = argparse.ArgumentParser()

    parser.add_argument('--run', '-r', nargs=1)

    parser.add_argument('--config', '-c', nargs=1)

    args = parser.parse_args()


    if args.config is not None:

        # use user config file if specified in command line

        SERVICES = app_api.Services(args.config[0])

    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    name = str(input("Enter mode name: "))

    # send mutation to remove mode from scheduler
    request = '''
    mutation {
        removeMode(name: "%s") {
            success
            errors
        }
    }''' % (name)
    response = SERVICES.query(service="scheduler-service", query=request)

    # get results
    response = response["removeMode"]
    success = response["success"]
    errors = response["errors"]

    if success:
        logger.info("Removed mode named: {}.".format(name))
    else:
        logger.warning("Could not remove {} mode: {}.".format(name, errors))


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

remove-task-list.py

```
#!/usr/bin/env python3

"""
Remove a task list from a mode in the scheduler. If the mode is active, all tasks in the task list will be removed from the schedule.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"
```

```python
import app_api
import argparse
import sys

def main():

    logger = app_api.logging_setup("remove-task-list")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services("/etc/kubos-config.toml")

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)


# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass


# logic run when commanded by OBC
def on_command(logger, SERVICES):


    name = "task_name"
    mode = "mode_name"


    # send mutation to remove task list from mode in scheduler
    request = '''
    mutation {
        removeTaskList(name: "%s", mode: "%s") {
            success
            errors
        }
    }
    ''' % (name, mode)
    response = SERVICES.query(service="scheduler-service", query=request)


    # get results
    response = response["removeTaskList"]
    success = response["success"]
    errors = response["errors"]
    if success:
        logger.info("Removed task list {} from mode {}.".format(name, mode))
    else:
        logger.warning("Could not remove task list {} from mode {}: {}.".format(name, mode, errors))



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

TELEMETRY Applications

add-entry-database.py

```
#!/usr/bin/env python3

"""
Add entry to telemetry database
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time
```

```
def main():

    logger = app_api.logging_setup("add-entry-database")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass


# logic run when commanded by OBC
def on_command(logger, SERVICES):
    name = str(input("Enter telemetry item name: "))
    subsystem = str(input("Enter telemetry item subsystem: "))
    value = str(input("Enter telemetry item value: "))

    # add telemetry to database
    try:
        # timestamp is optional and defaults to current system time
        request = '''
        mutation {
            insert(subsystem: "%s", parameter: "%s", value: "%s") {
                success
                errors
            }
        }
        ''' % (subsystem, name, value)
        response = SERVICES.query(service="telemetry-service", query=request)

        # get results
        result = response["insert"]
        success = result["success"]
        errors = result["errors"]

        if success:
            logger.info("Logged telemetry in database.")
        else:
            logger.warn("Unable to log telemetry to database: {}".format(errors))

    except Exception as e:
        logger.error("Error logging information in telemetry database: {}{}".format(type(e).__name__,str(e)))

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

clear-database.py

```
#!/usr/bin/env python3

"""
Mission application that cleans out telemetry database
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time

def main():
```

```python
    logger = app_api.logging_setup("health-clear-database")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass
```

```
# logic run when commanded by OBC
def on_command(logger, SERVICES):
    # delete all telemetry entries before a certain time
    timestamp = time.time() - 86400
    try:
        '''
        mutation {
            delete(timestampGe: Float, timestampLe: Float, subsystem: String, parameter: String): [{
                success: Boolean!,
                errors: String!,
                entriesDeleted: Integer
            }]
        }
        timestampGe = delete entries with timestamps on or after given value
        timestampLe = delete entries with timestamps on or before given value
        subsystem = delete entries that match subsystem name
        parameter = delete entries which match given parameter name (mutually exclusive with parameters)
        '''

        request = '''
        mutation {
            delete(timestampLe: %f) {
                success
                errors
                entriesDeleted
            }
        }
        ''' % (timestamp) # delete entries more than a day old
        response = SERVICES.query(service="telemetry-service", query=request)

        # get results
        result = response["delete"]
        success = result["success"]
        errors = result["errors"]
        entriesDeleted = result["entriesDeleted"]

        if success:
            logger.debug("Deleted {} entries from database before timestamp:
{}".format(entriesDeleted, timestamp))
        else:
            logger.warn("Unable to delete entries beforee timestamp:
{}".format(timestamp))

    except Exception as e:
        logger.error("Exception trying to clean telemetry database:
{}{}".format(type(e).__name__,str(e)))

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

mem-database-query.py

```
#!/usr/bin/env python3
```

```
"""
Mission application that retrieves current memory usage.
"""

__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import app_api
import argparse
import sys
import time

def main():

    logger = app_api.logging_setup("health-mem-query")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass

# logic run when commanded by OBC
def on_command(logger, SERVICES):
    # retrieve memory telemetry from database
    try:
        '''
        query {
            telemetry(timestampGe: Float, timestampLe: Float, subsystem: String, parameter: String, parameters: [String], limit: Integer): [{
                timestamp
                subsystem
                parameter
                value
            }]
        }
        timestampGe = return entries with timestamps on or after given value
        timestampLe = return entries with timestamps on or before given value
        subsystem = return entries that match subsystem name
        parameter = return entries which match given parameter name (mutually exclusive
with parameters)
        parameters = returns entries which match given parameter names (mutually exclusive
with parameter)
        limit = return only first n entries found
        '''

        request = '''
        {
            telemetry(parameter: "free_memory_percentage") {
                timestamp
                subsystem
                parameter
                value
            }
        }
        '''
        response = SERVICES.query(service="telemetry-service", query=request)

        # get results
        result = response["telemetry"]
        for item in result:
            timestamp = item["timestamp"]
            subsystem = item["subsystem"]
            parameter = item["parameter"]
            value = item["value"]
            logger.debug("Got telemetry:{} | timestamp:{} | subsystem:{} |
value:{}".format(parameter, timestamp, subsystem, value))


    except Exception as e:
        logger.error("Unable to retrieve memory telemetry from database:
{}{}".format(type(e).__name__,str(e)))


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

print-database.py

```
#!/usr/bin/env python3

"""
Mission application that prints out values in telemetry database
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"


import app_api
import argparse
import sys
import time


def main():
    logger = app_api.logging_setup("health-print-database")

    # parse arguments for config file and run type
    parser = argparse.ArgumentParser()
    parser.add_argument('--run', '-r', nargs=1)
    parser.add_argument('--config', '-c', nargs=1)
    args = parser.parse_args()

    if args.config is not None:
        # use user config file if specified in command line
        SERVICES = app_api.Services(args.config[0])
    else:
        # else use default global config file
        SERVICES = app_api.Services()

    # run app onboot or oncommand logic
    if args.run is not None:
        if args.run[0] == 'OnBoot':
            on_boot(logger, SERVICES)
        elif args.run[0] == 'OnCommand':
            on_command(logger, SERVICES)
    else:
        on_command(logger, SERVICES)

# logic run for application on OBC boot
def on_boot(logger, SERVICES):
    pass
```

```
# logic run when commanded by OBC
def on_command(logger, SERVICES):
    # retrieve memory telemetry from database
    try:
        '''
        query {
            telemetry(timestampGe: Float, timestampLe: Float, subsystem: String,
parameter: String, parameters: [String], limit: Integer): [{
                timestamp
                subsystem
                parameter
                value
            }]
        }
        timestampGe = return entries with timestamps on or after given value
        timestampLe = return entries with timestamps on or before given value
        subsystem = return entries that match subsystem name
        parameter = return entries which match given parameter name (mutually
exclusive with parameters)
        parameters = returns entries which match given parameter names (mutually
exclusive with parameter)
        limit = return only first n entries found
        '''

        request = '''
        {
            telemetry(timestampLe: %f) {
                timestamp
                subsystem
                parameter
                value
            }
        }
        ''' % (time.time())
        response = SERVICES.query(service="telemetry-service", query=request)

        # get results
        result = response["telemetry"]
        for item in result:
            timestamp = item["timestamp"]
            subsystem = item["subsystem"]
            parameter = item["parameter"]
            value = item["value"]

            logger.debug("Got telemetry:{} | timestamp:{} | subsystem:{} |
value:{}".format(parameter, timestamp, subsystem, value))

    except Exception as e:
        logger.error("Unable to retrieve memory telemetry from database:
{}|{}".format(type(e).__name__,str(e)))

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

SERIAL Libraries

gpio.py

```
#!/usr/bin/env python3


"""
Winsat gpio library for interacting with BBB GPIO pins
"""


__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import time
import app_api
import os

class GPIO:

    def __init__(self, pin):
        self.pin = pin
        self.logger = app_api.logging_setup("gpio-service-pin{}".format(self.pin))

    def attach(self):
        try:
            if os.system("echo {} > /sys/class/gpio/export".format(self.pin)) == 0:
                return True
            else:
                self.logger.warning("Error attaching GPIO pin {}. Using fake
GPIO...".format(self.pin))
```

```
                return False
        except Exception as e:
            self.logger.warning("Error attaching GPIO pin {}: {}. Using fake
GPIO...".format(self.pin), str(e))
            return False

    def release(self):
        try:
            os.system("echo {} > /sys/class/gpio/unexport".format(self.pin))
            return True
        except Exception as e:
            self.logger.warning("Error releasing GPIO pin {}: {}".format(self.pin), str(e))
            return False

    def on(self):
        try:
            os.system("echo 1 > /sys/class/gpio/gpio{}/value".format(self.pin))
            return True
        except Exception as e:
            self.logger.warning("Error turning on GPIO pin {}: {}".format(self.pin),
str(e))
            return False

    def off(self):
        try:
            os.system("echo 0 > /sys/class/gpio/gpio{}/value".format(self.pin))
            return True
        except Exception as e:
            self.logger.warning("Error turning off of GPIO pin {}: {}".format(self.pin),
str(e))

            return False


    def direction(self, direction):

        if direction:

            try:

                os.system("echo out > /sys/class/gpio/gpio{}/direction".format(self.pin))

                return True

            except Exception as e:

                self.logger.warning("Error setting direction of GPIO pin {}:
{}".format(self.pin), str(e))

                return False

        else:

            try:

                os.system("echo in > /sys/class/gpio/gpio{}/direction".format(self.pin))

                return True

            except Exception as e:

                self.logger.warning("Error setting direction of GPIO pin {}:
{}".format(self.pin), str(e))

                return False
```

I2c.py

```
#!/usr/bin/env
python3

"""

Winsat i2c library built on top of built-in Kubos i2c library

"""
```

```python
__author__ = "Jon Grebe"
__version__ = "0.1.0"
__license__ = "MIT"

import graphene
import serial
import time
import app_api
import smbus

class I2C:

    def __init__(self, bus, slave_address):
        self.slave_address = slave_address
        self.bus = smbus.SMBus(bus)

    def write(self, register_address, data):
           return self.bus.write_byte_data(self.slave_address, register_address,
data)

    def read(self, register_address):
        return self.bus.read_byte_data(self.slave_address, register_address)

class I2C_fake:

    def __init__(self, bus, slave_address):
        pass
    def write(self, register_address, data):
           return


    def read(self, register_address):
        return 0x00
```

uart.py

```
#!/usr/bin/env python3


import serial
import time
from obcserial import config
from xmodem import XMODEM
import app_api


class UART:
    def __init__(self, port_number):
        self.port = config.PORT_NAME[port_number]
        self.serial = serial.Serial(
            port=self.port,
            baudrate=115200,
            parity=serial.PARITY_NONE,
            stopbits=serial.STOPBITS_ONE,
            bytesize=serial.EIGHTBITS,
            timeout=1)


        # setup xmodem for image transfers
        self.modem = XMODEM(self.getc, self.putc)

        self.logger = app_api.logging_setup("UART")

    def getc(self, size, timeout=1):
        return self.serial.read(size) or None

    def putc(self, data, timeout=1):
        return self.serial.write(data) # note that this ignores the timeout

    def transfer_image(self, filename):
        try:
            # open uart port
            self.serial.close()
            self.serial.open()

            if self.serial.isOpen():
                self.logger.debug("UART port {} is open. Waiting for file...".format(self.port))
                stream = open(filename, 'wb+')
                result = self.modem.recv(stream)
                self.serial.close()
                return result
            else:
                self.serial.close()
                self.logger.warn("Could not open serial port for file transfer: {}".format(self.port))
                return False
        except Exception as e:
            self.logger.warn("Exception trying to read file: {} from xmodem 
stream: {}|{}".format(filename, type(e).__name__,str(e)))
            return False


    # send message to hardware over uart
    def write(self, message):
        try:
            # open uart port
            self.serial.close()
            self.serial.open()


            if self.serial.isOpen():
                # if uart port is open, try to send encoded string message
                self.serial.write(str(message + '\r\n').encode('utf-8'))
                self.serial.close()
                self.logger.debug("UART port {} is open. Sent message: 
{}".format(self.port, str(message)))
                return True
            else:
                # if could not open uart port, return failure
                self.serial.close()
                self.logger.warn("Could not open serial port: 
{}".format(self.port))
                return False


        # return failure if exception during write/encoding
        except Exception as e:
            self.logger.warn("Error sending message {} over uart port {}: 
{}".format(str(message), self.port, str(e)))
```

```
            self.serial.close()
            return False

    # get message from hardware over uart
    def read(self):
        try:
            message = None
            self.serial.close()
            self.serial.open()
            if self.serial.isOpen():
                # if uart port is open, try to read something
                message = self.serial.readline()
                message = message.decode('utf-8')
                self.logger.debug("Uart port {} is open. Read line: 
{}".format(self.port,message))
                self.serial.close()
                return True, message
            else:
                # if could not open uart port, return failure
                self.serial.close()
                self.logger.warn("Could not open serial port: 
{}".format(self.port))
                return False, None

        # return failure if exception during read/decoding
        except Exception as e:
            self.logger.warn("Error reading message: {} over uart port {}: 
{}".format(message, self.port, str(e)))
            self.serial.close
        return False, None
```

HARDWARE APIs

ADC – ADS1115.py

```
# Copyright (c) 2016 Adafruit Industries
# Author: Tony DiCola
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
import time
import Adafruit_GPIO.I2C as I2C


# Register and other configuration values:
ADS1x15_DEFAULT_ADDRESS        = 0x48
ADS1x15_POINTER_CONVERSION     = 0x00
ADS1x15_POINTER_CONFIG         = 0x01
ADS1x15_POINTER_LOW_THRESHOLD  = 0x02
ADS1x15_POINTER_HIGH_THRESHOLD = 0x03
ADS1x15_CONFIG_OS_SINGLE       = 0x8000
ADS1x15_CONFIG_MUX_OFFSET      = 12
# Maping of gain values to config register values.
ADS1x15_CONFIG_GAIN = {
    2/3: 0x0000,
    1:   0x0200,
    2:   0x0400,
    4:   0x0600,
    8:   0x0800,
    16:  0x0A00
}
ADS1x15_CONFIG_MODE_CONTINUOUS  = 0x0000
ADS1x15_CONFIG_MODE_SINGLE      = 0x0100


# Mapping of data/sample rate to config register values for ADS1115 (slower).
ADS1115_CONFIG_DR = {
    8:    0x0000,
    16:   0x0020,
    32:   0x0040,
    64:   0x0060,
    128:  0x0080,
    250:  0x00A0,
    475:  0x00C0,
    860:  0x00E0
}
ADS1x15_CONFIG_COMP_WINDOW      = 0x0010
ADS1x15_CONFIG_COMP_ACTIVE_HIGH = 0x0008
ADS1x15_CONFIG_COMP_LATCHING    = 0x0004
ADS1x15_CONFIG_COMP_QUE = {
    1: 0x0000,
    2: 0x0001,
    4: 0x0002
}
ADS1x15_CONFIG_COMP_QUE_DISABLE = 0x0003


class ADS1115():
    """ADS1115 16-bit analog to digital converter instance."""

    def __init__(self, address=ADS1x15_DEFAULT_ADDRESS, bus=2):
        self._device = I2C.get_i2c_device(address, bus)

    def _data_rate_default(self):
        """Retrieve the default data rate for this ADC (in samples per second).
        """
        # Default from datasheet page 16, config register DR bit default.
        return 128

    def _data_rate_config(self, data_rate):
        """Return a 16-bit value
        that can be OR'ed with the config register to set the specified
        data rate.  If a value of None is specified then a default data_rate
        setting should be returned.  If an invalid or unsupported data_rate is
        provided then an exception should be thrown.
        """
        if data_rate not in ADS1115_CONFIG_DR:
            raise ValueError('Data rate must be one of: 8, 16, 32, 64, 128, 250, 475,
860')
        return ADS1115_CONFIG_DR[data_rate]


    def _conversion_value(self, low, high):
        """Takes the low and high byte of a conversion result and returns a signed
integer value."""
        # Convert to 16-bit signed value.
        value = ((high & 0xFF) << 8) | (low & 0xFF)
        # Check for sign bit and turn into a negative value if set.
        if value & 0x8000 != 0:
            value -= 1 << 16
        return value


    def _read(self, mux, gain, data_rate, mode):
        """Perform an ADC read with the provided mux, gain, data_rate, and mode
        values.  Returns the signed integer result of the read.
        """
        config = ADS1x15_CONFIG_OS_SINGLE  # Go out of power-down mode for
conversion.
        # Specify mux value.
        config |= (mux & 0x07) << ADS1x15_CONFIG_MUX_OFFSET
        # Validate the passed in gain and then set it in the config.
        if gain not in ADS1x15_CONFIG_GAIN:
            raise ValueError('Gain must be one of: 2/3, 1, 2, 4, 8, 16')
        config |= ADS1x15_CONFIG_GAIN[gain]
        # Set the mode (continuous or single shot).
        config |= mode
        # Get the default data rate if none is specified (default differs between
        # ADS1015 and ADS1115).
        if data_rate is None:
            data_rate = self._data_rate_default()
        # Set the data rate (this is controlled by the subclass as it differs
        # between ADS1015 and ADS1115).
        config |= self._data_rate_config(data_rate)
        config |= ADS1x15_CONFIG_COMP_QUE_DISABLE  # Disble comparator mode.
        # Send the config value to start the ADC conversion.
        # Explicitly break the 16-bit value down to a big endian pair of bytes.
        self._device.writeList(ADS1x15_POINTER_CONFIG, [(config >> 8) & 0xFF, config & 0xFF])
        # Wait for the ADC sample to finish based on the sample rate plus a
        # small offset to be sure (0.1 millisecond).
        time.sleep(1.0/data_rate+0.0001)
        # Retrieve the result.
        result = self._device.readList(ADS1x15_POINTER_CONVERSION, 2)
        return self._conversion_value(result[1], result[0])


    def _read_comparator(self, mux, gain, data_rate, mode, high_threshold,
                         low_threshold, active_low, traditional, latching,
                         num_readings):
        """Perform an ADC read with the provided mux, gain, data_rate, and mode
        values and with the comparator enabled as specified.  Returns the signed
        integer result of the read.
        """
        assert num_readings == 1 or num_readings == 2 or num_readings == 4, 'Num readings must be 1, 2, or 4!'
        # Set high and low threshold register values.
        self._device.writeList(ADS1x15_POINTER_HIGH_THRESHOLD, [(high_threshold >> 8) & 0xFF, high_threshold & 0xFF])
        self._device.writeList(ADS1x15_POINTER_LOW_THRESHOLD, [(low_threshold >> 8) & 0xFF, low_threshold & 0xFF])
        # Now build up the appropriate config register value.
        config = ADS1x15_CONFIG_OS_SINGLE  # Go out of power-down mode for conversion.
        # Specify mux value.
        config |= (mux & 0x07) << ADS1x15_CONFIG_MUX_OFFSET
        # Validate the passed in gain and then set it in the config.
        if gain not in ADS1x15_CONFIG_GAIN:
            raise ValueError('Gain must be one of: 2/3, 1, 2, 4, 8, 16')
        config |= ADS1x15_CONFIG_GAIN[gain]
        # Set the mode (continuous or single shot).
        config |= mode
        # Get the default data rate if none is specified (default differs between
        # ADS1015 and ADS1115).
        if data_rate is None:
            data_rate = self._data_rate_default()
        # Set the data rate (this is controlled by the subclass as it differs
        # between ADS1015 and ADS1115).
        config |= self._data_rate_config(data_rate)
        # Enable window mode if required.
        if not traditional:
            config |= ADS1x15_CONFIG_COMP_WINDOW
        # Enable active high mode if required.
        if not active_low:
            config |= ADS1x15_CONFIG_COMP_ACTIVE_HIGH
        # Enable latching mode if required.
        if latching:
            config |= ADS1x15_CONFIG_COMP_LATCHING
        # Set number of comparator hits before alerting.
        config |= ADS1x15_CONFIG_COMP_QUE[num_readings]
        # Send the config value to start the ADC conversion.
        # Explicitly break the 16-bit value down to a big endian pair of bytes.
        self._device.writeList(ADS1x15_POINTER_CONFIG, [(config >> 8) & 0xFF, config
& 0xFF])
        # Wait for the ADC sample to finish based on the sample rate plus a
        # small offset to be sure (0.1 millisecond).
        time.sleep(1.0/data_rate+0.0001)
        # Retrieve the result.
        result = self._device.readList(ADS1x15_POINTER_CONVERSION, 2)
        return self._conversion_value(result[1], result[0])

    def read_adc(self, channel, gain=1, data_rate=None):
        """Read a single ADC channel and return the ADC value as a signed integer
        result.  Channel must be a value within 0-3.
        """
        assert 0 <= channel <= 3, 'Channel must be a value within 0-3!'
        # Perform a single shot read and set the mux value to the channel plus
        # the highest bit (bit 3) set.
        return self._read(channel + 0x04, gain, data_rate,
ADS1x15_CONFIG_MODE_SINGLE)
```

```
    def read_adc_difference(self, differential, gain=1, data_rate=None):
        """Read the difference between two ADC channels and return the ADC value
        as a signed integer result.  Differential must be one of:
          - 0 = Channel 0 minus channel 1
          - 1 = Channel 0 minus channel 3
          - 2 = Channel 1 minus channel 3
          - 3 = Channel 2 minus channel 3
        """
        assert 0 <= differential <= 3, 'Differential must be a value within 0-3!'
        # Perform a single shot read using the provided differential value
        # as the mux value (which will enable differential mode).
        return self._read(differential, gain, data_rate, ADS1x15_CONFIG_MODE_SINGLE)

    def start_adc(self, channel, gain=1, data_rate=None):
        """Start continuous ADC conversions on the specified channel (0-3). Will
        return an initial conversion result, then call the get_last_result()
        function to read the most recent conversion result. Call stop_adc() to
        stop conversions.
        """
        assert 0 <= channel <= 3, 'Channel must be a value within 0-3!'
        # Start continuous reads and set the mux value to the channel plus
        # the highest bit (bit 3) set.
        return self._read(channel + 0x04, gain, data_rate,
ADS1x15_CONFIG_MODE_CONTINUOUS)

    def start_adc_difference(self, differential, gain=1, data_rate=None):
        """Start continuous ADC conversions between two ADC channels. Differential
        must be one of:
          - 0 = Channel 0 minus channel 1
          - 1 = Channel 0 minus channel 3
          - 2 = Channel 1 minus channel 3
          - 3 = Channel 2 minus channel 3
        Will return an initial conversion result, then call the get_last_result()
        function continuously to read the most recent conversion result.  Call
        stop_adc() to stop conversions.
        """
        assert 0 <= differential <= 3, 'Differential must be a value within 0-3!'
        # Perform a single shot read using the provided differential value
        # as the mux value (which will enable differential mode).
        return self._read(differential, gain, data_rate,
ADS1x15_CONFIG_MODE_CONTINUOUS)

    def start_adc_comparator(self, channel, high_threshold, low_threshold,
                             gain=1, data_rate=None, active_low=True,
                             traditional=True, latching=False, num_readings=1):
        """Start continuous ADC conversions on the specified channel (0-3) with
        the comparator enabled.  When enabled the comparator to will check if
        the ADC value is within the high_threshold & low_threshold value (both
        should be signed 16-bit integers) and trigger the ALERT pin.  The
        behavior can be controlled by the following parameters:
          - active_low: Boolean that indicates if ALERT is pulled low or high
                        when active/triggered.  Default is true, active low.
          - traditional: Boolean that indicates if the comparator is in traditional
                         mode where it fires when the value is within the threshold,
                         or in window mode where it fires when the value is _outside_
                         the threshold range.  Default is true, traditional mode.
          - latching: Boolean that indicates if the alert should be held until
```

```
                      get_last_result() is called to read the value and clear
                      the alert.  Default is false, non-latching.
          - num_readings: The number of readings that match the comparator before
                          triggering the alert.  Can be 1, 2, or 4.  Default is 1.
        Will return an initial conversion result, then call the get_last_result()
        function continuously to read the most recent conversion result.  Call
        stop_adc() to stop conversions.
        """
        assert 0 <= channel <= 3, 'Channel must be a value within 0-3!'
        # Start continuous reads with comparator and set the mux value to the
        # channel plus the highest bit (bit 3) set.
        return self._read_comparator(channel + 0x04, gain, data_rate,
                                     ADS1x15_CONFIG_MODE_CONTINUOUS,
                                     high_threshold, low_threshold, active_low,
                                     traditional, latching, num_readings)


    def start_adc_difference_comparator(self, differential, high_threshold,
low_threshold,
                                        gain=1, data_rate=None, active_low=True,
                                        traditional=True, latching=False,
num_readings=1):
        """Start continuous ADC conversions between two channels with
        the comparator enabled.  See start_adc_difference for valid differential
        parameter values and their meaning.  When enabled the comparator to will
        check if the ADC value is within the high_threshold & low_threshold value
        (both should be signed 16-bit integers) and trigger the ALERT pin.  The
        behavior can be controlled by the following parameters:
          - active_low: Boolean that indicates if ALERT is pulled low or high
                        when active/triggered.  Default is true, active low.
          - traditional: Boolean that indicates if the comparator is in traditional
                         mode where it fires when the value is within the threshold,
                         or in window mode where it fires when the value is _outside_
                         the threshold range.  Default is true, traditional mode.
          - latching: Boolean that indicates if the alert should be held until
                      get_last_result() is called to read the value and clear
                      the alert.  Default is false, non-latching.
          - num_readings: The number of readings that match the comparator before
                          triggering the alert.  Can be 1, 2, or 4.  Default is 1.
        Will return an initial conversion result, then call the get_last_result()
        function continuously to read the most recent conversion result.  Call
        stop_adc() to stop conversions.
        """
        assert 0 <= differential <= 3, 'Differential must be a value within 0-3!'
        # Start continuous reads with comparator and set the mux value to the
        # channel plus the highest bit (bit 3) set.
        return self._read_comparator(differential, gain, data_rate,
                                     ADS1x15_CONFIG_MODE_CONTINUOUS,
                                     high_threshold, low_threshold, active_low,
                                     traditional, latching, num_readings)

    def stop_adc(self):
        """Stop all continuous ADC conversions (either normal or difference mode).
        """
        # Set the config register to its default value of 0x8583 to stop
        # continuous conversions.
        config = 0x8583
        self._device.writeList(ADS1x15_POINTER_CONFIG, [(config >> 8) & 0xFF, config
& 0xFF])
```

```
def get_last_result(self):
    """Read the last conversion result when in continuous conversion mode.
    Will return a signed integer value.
    """
    # Retrieve the conversion register value, convert to a signed int, and
    # return it.
    result = self._device.readList(ADS1x15_POINTER_CONVERSION, 2)
    return self._conversion_value(result[1], result[0])
```

RTC – DS3231.py

```
#!/usr/bin/env
python3
```

```
from obcserial import i2c
import app_api
import smbus
import datetime

DS3231_ADDRESS = 0x68

class DS3231:
    """Interface to the DS3231 RTC."""

    def __init__(self, bus):
        """
        Sets I2C bus number and address
        """
```

```
            self.fake = False
            try:
                self.i2cfile = i2c.I2C(bus=bus, slave_address=DS3231_ADDRESS)
            except Exception as e:
                # exception trying to open i2c bus, run fake version
                self.fake = True

        def bcd2bin(self, value):
            """
            Convert binary coded decimal (bcd) to binary
            """
            return value - 6 * (value >> 4)

        def bin2bcd(self, value):
            """
            Convert binary value to binary coded decimal (bcd)
            """
            return value + 6 * (value // 10)

        def datetime(self):
            """
            Get current date and time from RTC.
            """
            if (self.fake):
                return datetime.datetime(1970, 1, 1, 0, 0, 0)

            sec = self.bcd2bin(self.i2cfile.read(0x00) & 0x7F)
            minute = self.bcd2bin(self.i2cfile.read(0x01))
```

```
                    hour = self.bcd2bin(self.i2cfile.read(0x02))
                    wday = self.bcd2bin(self.i2cfile.read(0x03) - 1)
                    mday = self.bcd2bin(self.i2cfile.read(0x04))
                    month = self.bcd2bin(self.i2cfile.read(0x05))
                    year = self.bcd2bin(self.i2cfile.read(0x06)) + 2000

                    return datetime.datetime(year, month, mday, hour, minute, sec)
```

IMU – IMU.py

```
import
smbus
        import logging
        import graphene
        from winserial import i2c
        import struct
        from math import sqrt, atan2, asin, degrees, radians
        import time
        from deltat import DeltaT


        _FXOS8700_ADDRESS                = 0x1F   # 0011111
        _FXOS8700_ID                     = 0xC7   # 1100 0111
        _FXOS8700_REGISTER_STATUS        = 0x00
        _FXOS8700_REGISTER_OUT_X_MSB     = 0x01
        _FXOS8700_REGISTER_OUT_X_LSB     = 0x02
        _FXOS8700_REGISTER_OUT_Y_MSB     = 0x03
        _FXOS8700_REGISTER_OUT_Y_LSB     = 0x04
        _FXOS8700_REGISTER_OUT_Z_MSB     = 0x05
        _FXOS8700_REGISTER_OUT_Z_LSB    = 0x06
        _FXOS8700_REGISTER_WHO_AM_I     = 0x0D   # 11000111   r
        _FXOS8700_REGISTER_XYZ_DATA_CFG = 0x0E
        _FXOS8700_REGISTER_CTRL_REG1    = 0x2A   # 00000000   r/w
        _FXOS8700_REGISTER_CTRL_REG2    = 0x2B   # 00000000   r/w
        _FXOS8700_REGISTER_CTRL_REG3    = 0x2C   # 00000000   r/w
        _FXOS8700_REGISTER_CTRL_REG4    = 0x2D   # 00000000   r/w
        _FXOS8700_REGISTER_CTRL_REG5    = 0x2E   # 00000000   r/w
        _FXOS8700_REGISTER_MSTATUS      = 0x32
        _FXOS8700_REGISTER_MOUT_X_MSB   = 0x33
        _FXOS8700_REGISTER_MOUT_X_LSB   = 0x34
        _FXOS8700_REGISTER_MOUT_Y_MSB   = 0x35
        _FXOS8700_REGISTER_MOUT_Y_LSB   = 0x36
        _FXOS8700_REGISTER_MOUT_Z_MSB   = 0x37
        _FXOS8700_REGISTER_MOUT_Z_LSB   = 0x38
        _FXOS8700_REGISTER_MCTRL_REG1   = 0x5B   # 00000000   r/w
        _FXOS8700_REGISTER_MCTRL_REG2   = 0x5C   # 00000000   r/w
        _FXOS8700_REGISTER_MCTRL_REG3   = 0x5D   # 00000000   r/w


        _FXAS21002C_ADDRESS       = 0x21  # 0100001
        _FXAS21002C_ID            = 0xD7       # 1101 0111
        _GYRO_REGISTER_STATUS     = 0x00
        _GYRO_REGISTER_OUT_X_MSB  = 0x01
        _GYRO_REGISTER_OUT_X_LSB  = 0x02
        _GYRO_REGISTER_OUT_Y_MSB  = 0x03
        _GYRO_REGISTER_OUT_Y_LSB  = 0x04
        _GYRO_REGISTER_OUT_Z_MSB  = 0x05
```

```
_GYRO_REGISTER_OUT_Z_LSB  = 0x06

_GYRO_REGISTER_WHO_AM_I   = 0x0C   # 11010111   r

_GYRO_REGISTER_CTRL_REG0  = 0x0D  # 00000000   r/w

_GYRO_REGISTER_CTRL_REG1  = 0x13  # 00000000   r/w

_GYRO_REGISTER_CTRL_REG2  = 0x14  # 00000000   r/w

_GYRO_SENSITIVITY_250DPS  = 0.0078125    # Table 35 of datasheet

_GYRO_SENSITIVITY_500DPS  = 0.015625     # ..

_GYRO_SENSITIVITY_1000DPS = 0.03125     # ..

_GYRO_SENSITIVITY_2000DPS = 0.0625      # .


_MAG_UT_LSB                     = 0.1

_ACCEL_MG_LSB_2G                = 0.000244

_SENSORS_GRAVITY_STANDARD       = 9.80665


logger = logging.getLogger('imu-service')

class IMU(graphene.ObjectType):

    declination = 0

    def __init__(self, bus, timediff=None):


        timediff = lambda start, end : (start-end)/1000000

        self.magbias = (0, 0, 0)

        self.q = [1.0, 0.0, 0.0, 0.0]

        GyroMeasError = radians(40)

        self.beta = sqrt(3.0 / 4.0) * GyroMeasError

        self.deltat = DeltaT(timediff)


        try:

            self.fxos_i2c = i2c.I2C(bus=bus, slave_address=_FXOS8700_ADDRESS)
            self.fxas_i2c = i2c.I2C(bus=bus, slave_address=_FXAS21002C_ADDRESS)
        except Exception as e:
            logger.info("Unable to open I2C bus {} to device: {}. Will use fake i2c for testing.".format(bus, _FXOS8700_ADDRESS))
            self.fxos_i2c = i2c.I2C_fake(bus=bus, slave_address=_FXOS8700_ADDRESS)
            self.fxas_i2c = i2c.I2C_fake(bus=bus, slave_address=_FXAS21002C_ADDRESS)
            return

        if self.fxos_i2c.read(_FXOS8700_REGISTER_WHO_AM_I) != _FXOS8700_ID:
            logger.info('Failed to find FXOS8700, check wiring!. Using fake i2c for now')
            self.fxos_i2c = i2c.I2C_fake(bus=bus, slave_address=_FXOS8700_ADDRESS)

        if self.fxas_i2c.read(_GYRO_REGISTER_WHO_AM_I) != _FXAS21002C_ID:
            logger.info('Failed to find FXAS21002C, check wiring!. Using fake i2c for now')
            self.fxas_i2c = i2c.I2C_fake(bus=bus, slave_address=_FXAS21002C_ADDRESS)

        # Set to standby mode (required to make changes to this register)
        self.fxos_i2c.write(_FXOS8700_REGISTER_CTRL_REG1, 0)
        # set accel range to 2G
        self.fxos_i2c.write(_FXOS8700_REGISTER_XYZ_DATA_CFG, 0x00)
        # High resolution
        self.fxos_i2c.write(_FXOS8700_REGISTER_CTRL_REG2, 0x02)
        # Active, Normal Mode, Low Noise, 100Hz in Hybrid Mode
        self.fxos_i2c.write(_FXOS8700_REGISTER_CTRL_REG1, 0x15)
        # Configure the magnetometer
        # Hybrid Mode, Over Sampling Rate = 16
        self.fxos_i2c.write(_FXOS8700_REGISTER_MCTRL_REG1, 0x1F)
        # Jump to reg 0x33 after reading 0x06
        self.fxos_i2c.write(_FXOS8700_REGISTER_MCTRL_REG2, 0x20)

        self.fxas_i2c.write(_GYRO_REGISTER_CTRL_REG0, 0x03) # Set sensitivity
        self.fxas_i2c.write(_GYRO_REGISTER_CTRL_REG1, 0x0E)     # Active
        time.sleep(0.1) # 60 ms + 1/ODR

    # return magnetometer data
    def mag(self):
        try:
            x_MSB = self.fxos_i2c.read(_FXOS8700_REGISTER_MOUT_X_MSB)
            x_LSB = self.fxos_i2c.read(_FXOS8700_REGISTER_MOUT_X_LSB)

            y_MSB = self.fxos_i2c.read(_FXOS8700_REGISTER_MOUT_Y_MSB)
            y_LSB = self.fxos_i2c.read(_FXOS8700_REGISTER_MOUT_Y_LSB)

            z_MSB = self.fxos_i2c.read(_FXOS8700_REGISTER_MOUT_Z_MSB)
            z_LSB = self.fxos_i2c.read(_FXOS8700_REGISTER_MOUT_Z_LSB)

            BUFFER = bytearray()
            BUFFER = [x_MSB, x_LSB, y_MSB, y_LSB, z_MSB, z_LSB]
            BUFFER = bytes(BUFFER)
            x = struct.unpack_from('>H', BUFFER[0:2])[0]
            y = struct.unpack_from('>H', BUFFER[2:4])[0]
            z = struct.unpack_from('>H', BUFFER[4:6])[0]

            x = self._twos_comp(x >> 2, 14)
            y = self._twos_comp(y >> 2, 14)
            z = self._twos_comp(z >> 2, 14)
            x = x * _MAG_UT_LSB
            y = y * _MAG_UT_LSB
            z = z * _MAG_UT_LSB

            success = True
            errors = []
            return success, errors, x, y, z

        except Exception as e:
            success = False
            errors = ["{}:{}".format(type(e).__name__,str(e))]
            return success, errors, None, None, None

    # return accelerometer x,y,z values
    def acc(self):
        try:
            x_MSB = self.fxos_i2c.read(_FXOS8700_REGISTER_OUT_X_MSB)
            x_LSB = self.fxos_i2c.read(_FXOS8700_REGISTER_OUT_X_LSB)

            y_MSB = self.fxos_i2c.read(_FXOS8700_REGISTER_OUT_Y_MSB)
            y_LSB = self.fxos_i2c.read(_FXOS8700_REGISTER_OUT_Y_LSB)

            z_MSB = self.fxos_i2c.read(_FXOS8700_REGISTER_OUT_Z_MSB)
            z_LSB = self.fxos_i2c.read(_FXOS8700_REGISTER_OUT_Z_LSB)

            BUFFER = bytearray()
            BUFFER = [x_MSB, x_LSB, y_MSB, y_LSB, z_MSB, z_LSB]
```

```
                BUFFER = bytes(BUFFER)
                x = struct.unpack_from('>H', BUFFER[0:2])[0]
                y = struct.unpack_from('>H', BUFFER[2:4])[0]
                z = struct.unpack_from('>H', BUFFER[4:6])[0]

                x = self._twos_comp(x >> 2, 14)
                y = self._twos_comp(y >> 2, 14)
                z = self._twos_comp(z >> 2, 14)

                x = x * _ACCEL_MG_LSB_2G * _SENSORS_GRAVITY_STANDARD
                y = y * _ACCEL_MG_LSB_2G * _SENSORS_GRAVITY_STANDARD
                z = z * _ACCEL_MG_LSB_2G * _SENSORS_GRAVITY_STANDARD

                success = True
                errors = []
                return success, errors, x, y, z

            except Exception as e:
                success = False
                errors = ["{}:{}".format(type(e).__name__,str(e))]
                return success, errors, None, None, None

        # return gyroscope x,y,z values
        def gyr(self):
            try:
                x_MSB = self.fxas_i2c.read(_GYRO_REGISTER_OUT_X_MSB)
                x_LSB = self.fxas_i2c.read(_GYRO_REGISTER_OUT_X_LSB)
                y_MSB = self.fxas_i2c.read(_GYRO_REGISTER_OUT_Y_MSB)

                y_LSB = self.fxas_i2c.read(_GYRO_REGISTER_OUT_Y_LSB)


                z_MSB = self.fxas_i2c.read(_GYRO_REGISTER_OUT_Z_MSB)

                z_LSB = self.fxas_i2c.read(_GYRO_REGISTER_OUT_Z_LSB)


                BUFFER = bytearray()

                BUFFER = [x_MSB, x_LSB, y_MSB, y_LSB, z_MSB, z_LSB]

                BUFFER = bytes(BUFFER)

                # Parse out the gyroscope data as 16-bit signed data.

                x = struct.unpack_from('>h', BUFFER[0:2])[0]

                y = struct.unpack_from('>h', BUFFER[2:4])[0]

                z = struct.unpack_from('>h', BUFFER[4:6])[0]


                x = x * _GYRO_SENSITIVITY_250DPS

                y = y * _GYRO_SENSITIVITY_250DPS

                z = z * _GYRO_SENSITIVITY_250DPS


                success = True

                errors = []

                return success, errors, x, y, z


            except Exception as e:

                success = False

                errors = ["{}:{}".format(type(e).__name__,str(e))]

                return success, errors, None, None, None


        def qua(self, ts=None):
            ts = time.time()
            success, errors, x, y, z = self.mag()
            mag = (x,y,z)
            success, errors, x, y, z = self.acc()
            accel = (x,y,z)
            success, errors, x, y, z = self.gyr()
            gyro = (x,y,z)

            mx, my, mz = (mag[x] - self.magbias[x] for x in range(3)) # Units irrelevant (normalised)
            ax, ay, az = accel                  # Units irrelevant (normalised)
            gx, gy, gz = (radians(x) for x in gyro)  # Units deg/s
            q1, q2, q3, q4 = (self.q[x] for x in range(4))   # short name local variable for readability
            # Auxiliary variables to avoid repeated arithmetic
            _2q1 = 2 * q1
            _2q2 = 2 * q2
            _2q3 = 2 * q3
            _2q4 = 2 * q4
            _2q1q3 = 2 * q1 * q3
            _2q3q4 = 2 * q3 * q4
            q1q1 = q1 * q1
            q1q2 = q1 * q2
            q1q3 = q1 * q3
            q1q4 = q1 * q4
            q2q2 = q2 * q2
            q2q3 = q2 * q3
            q2q4 = q2 * q4
            q3q3 = q3 * q3
            q3q4 = q3 * q4
            q4q4 = q4 * q4

            # Normalise accelerometer measurement
            norm = sqrt(ax * ax + ay * ay + az * az)
            if (norm == 0):
                return True, [], self.q[0], self.q[1], self.q[2], self.q[3]
            norm = 1 / norm                     # use reciprocal for division
            ax *= norm
            ay *= norm
            az *= norm

            # Normalise magnetometer measurement
            norm = sqrt(mx * mx + my * my + mz * mz)
            if (norm == 0):
                return True, [], self.q[0], self.q[1], self.q[2], self.q[3]
            norm = 1 / norm                     # use reciprocal for division
            mx *= norm
            my *= norm
            mz *= norm

            # Reference direction of Earth's magnetic field
            _2q1mx = 2 * q1 * mx
            _2q1my = 2 * q1 * my
            _2q1mz = 2 * q1 * mz
            _2q2mx = 2 * q2 * mx
            hx = mx * q1q1 - _2q1my * q4 + _2q1mz * q3 + mx * q2q2 + _2q2 * my * q3 + _2q2 * mz * q4 - mx * q3q3 - mx * q4q4
            hy = _2q1mx * q4 + my * q1q1 - _2q1mz * q2 + _2q2mx * q3 - my * q2q2 + my * q3q3 + _2q3 * mz * q4 - my * q4q4
            _2bx = sqrt(hx * hx + hy * hy)
            _2bz = -_2q1mx * q3 + _2q1my * q2 + mz * q1q1 + _2q2mx * q4 - mz * q2q2 + _2q3 * my * q4 - mz * q3q3 + mz * q4q4
            _4bx = 2 * _2bx
            _4bz = 2 * _2bz

            # Gradient descent algorithm corrective step
            s1 = (-_2q3 * (2 * q2q4 - _2q1q3 - ax) + _2q2 * (2 * q1q2 + _2q3q4 - ay) - _2bz * q3 * (_2bx * (0.5 - q3q3 - q4q4)
                 + _2bz * (q2q4 - q1q3) - mx) + (-_2bx * q4 + _2bz * q2) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my)
                 + _2bx * q3 * (_2bx * (q1q3 + q2q4) + _2bz * (0.5 - q2q2 - q3q3) - mz))

            s2 = (_2q4 * (2 * q2q4 - _2q1q3 - ax) + _2q1 * (2 * q1q2 + _2q3q4 - ay) - 4 * q2 * (1 - 2 * q2q2 - 2 * q3q3 - az)
                 + _2bz * q4 * (_2bx * (0.5 - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (_2bx * q3 + _2bz * q1) * (_2bx * (q2q3 - q1q4)
                 + _2bz * (q1q2 + q3q4) - my) + (_2bx * q4 - _4bz * q2) * (_2bx * (q1q3 + q2q4) + _2bz * (0.5 - q2q2 - q3q3) - mz))

            s3 = (-_2q1 * (2 * q2q4 - _2q1q3 - ax) + _2q4 * (2 * q1q2 + _2q3q4 - ay) - 4 * q3 * (1 - 2 * q2q2 - 2 * q3q3 - az)
                 + (-_4bx * q3 - _2bz * q1) * (_2bx * (0.5 - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx)
                 + (_2bx * q2 + _2bz * q4) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my)
                 + (_2bx * q1 - _4bz * q3) * (_2bx * (q1q3 + q2q4) + _2bz * (0.5 - q2q2 - q3q3) - mz))

            s4 = (_2q2 * (2 * q2q4 - _2q1q3 - ax) + _2q3 * (2 * q1q2 + _2q3q4 - ay) + (-_4bx * q4 + _2bz * q2) * (_2bx * (0.5 - q3q3 - q4q4)
                  + _2bz * (q2q4 - q1q3) - mx) + (-_2bx * q1 + _2bz * q3) * (_2bx * (q2q3 -
    q1q4) + _2bz * (q1q2 + q3q4) - my)
                  + _2bx * q2 * (_2bx * (q1q3 + q2q4) + _2bz * (0.5 - q2q2 - q3q3) - mz))

            norm = 1 / sqrt(s1 * s1 + s2 * s2 + s3 * s3 + s4 * s4)    # normalise step
    magnitude
            s1 *= norm
            s2 *= norm
            s3 *= norm
            s4 *= norm

            # Compute rate of change of quaternion
            qDot1 = 0.5 * (-q2 * gx - q3 * gy - q4 * gz) - self.beta * s1
            qDot2 = 0.5 * (q1 * gx + q3 * gz - q4 * gy) - self.beta * s2
            qDot3 = 0.5 * (q1 * gy - q2 * gz + q4 * gx) - self.beta * s3
            qDot4 = 0.5 * (q1 * gz + q2 * gy - q3 * gx) - self.beta * s4

            # Integrate to yield quaternion
            deltat = self.deltat(ts)
            q1 += qDot1 * deltat
            q2 += qDot2 * deltat
            q3 += qDot3 * deltat
            q4 += qDot4 * deltat
            norm = 1 / sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4)    # normalise quaternion
            self.q = q1 * norm, q2 * norm, q3 * norm, q4 * norm
            self.heading = self.declination + degrees(atan2(2.0 * (self.q[1] * self.q[2] +
    self.q[0] * self.q[3]),
                self.q[0] * self.q[0] + self.q[1] * self.q[1] - self.q[2] * self.q[2] -
    self.q[3] * self.q[3]))
```

```
        self.pitch = degrees(-asin(2.0 * (self.q[1] * self.q[3] - self.q[0] *
self.q[2])))

        self.roll = degrees(atan2(2.0 * (self.q[0] * self.q[1] + self.q[2] * self.q[3]),

            self.q[0] * self.q[0] - self.q[1] * self.q[1] - self.q[2] * self.q[2] +
self.q[3] * self.q[3]))


        return True, [], self.q[0], self.q[1], self.q[2], self.q[3]



    def _twos_comp(self, val, bits):

        # Convert an unsigned integer in 2's compliment form of the specified bit

        # length to its signed integer value and return it.

        if val & (1 << (bits - 1)) != 0:

            return val - (1 << bits)

        return val
```

RADIO – RFM69.py

```
# The MIT License (MIT)

#

# Copyright (c) 2017 Tony DiCola for Adafruit Industries

#

# Permission is hereby granted, free of charge, to any person obtaining a copy

# of this software and associated documentation files (the "Software"), to deal

# in the Software without restriction, including without limitation the rights

# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

# copies of the Software, and to permit persons to whom the Software is

# furnished to do so, subject to the following conditions:
```

```
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
"""
`adafruit_rfm69`
====================================================
CircuitPython RFM69 packet radio module. This supports basic RadioHead-compatible sending and
receiving of packets with RFM69 series radios (433/915Mhz).
.. warning:: This is NOT for LoRa radios!
.. note:: This is a 'best effort' at receiving data using pure Python code--there is not interrupt
    support so you might lose packets if they're sent too quickly for the board to process them.
    You will have the most luck using this in simple low bandwidth scenarios like sending and
    receiving a 60 byte packet at a time--don't try to receive many kilobytes of data at a time!
* Author(s): Tony DiCola, Jerry Needell
Implementation Notes
--------------------
**Hardware:**
```

```
* Adafruit `RFM69HCW Transceiver Radio Breakout - 868 or 915 MHz - RadioFruit
  <https://www.adafruit.com/product/3070>`_ (Product ID: 3070)
* Adafruit `RFM69HCW Transceiver Radio Breakout - 433 MHz - RadioFruit
  <https://www.adafruit.com/product/3071>`_ (Product ID: 3071)
* Adafruit `Feather M0 RFM69HCW Packet Radio - 868 or 915 MHz - RadioFruit
  <https://www.adafruit.com/product/3176>`_ (Product ID: 3176)
* Adafruit `Feather M0 RFM69HCW Packet Radio - 433 MHz - RadioFruit
  <https://www.adafruit.com/product/3177>`_ (Product ID: 3177)
* Adafruit `Radio FeatherWing - RFM69HCW 900MHz - RadioFruit
  <https://www.adafruit.com/product/3229>`_ (Product ID: 3229)
* Adafruit `Radio FeatherWing - RFM69HCW 433MHz - RadioFruit
  <https://www.adafruit.com/product/3230>`_ (Product ID: 3230)
**Software and Dependencies:**
* Adafruit CircuitPython firmware for the ESP8622 and M0-based boards:
  https://github.com/adafruit/circuitpython/releases
* Adafruit's Bus Device library:
https://github.com/adafruit/Adafruit_CircuitPython_BusDevice
"""
import time
import random

from micropython import const

import adafruit_bus_device.spi_device as spidev



__version__ = "0.0.0-auto.0"
__repo__ = "https://github.com/adafruit/Adafruit_CircuitPython_RFM69.git"
```

```
# Internal constants:

_REG_FIFO = const(0x00)

_REG_OP_MODE = const(0x01)

_REG_DATA_MOD = const(0x02)

_REG_BITRATE_MSB = const(0x03)

_REG_BITRATE_LSB = const(0x04)

_REG_FDEV_MSB = const(0x05)

_REG_FDEV_LSB = const(0x06)

_REG_FRF_MSB = const(0x07)

_REG_FRF_MID = const(0x08)

_REG_FRF_LSB = const(0x09)

_REG_VERSION = const(0x10)

_REG_PA_LEVEL = const(0x11)

_REG_RX_BW = const(0x19)

_REG_AFC_BW = const(0x1A)

_REG_RSSI_VALUE = const(0x24)

_REG_DIO_MAPPING1 = const(0x25)

_REG_IRQ_FLAGS1 = const(0x27)

_REG_IRQ_FLAGS2 = const(0x28)

_REG_PREAMBLE_MSB = const(0x2C)

_REG_PREAMBLE_LSB = const(0x2D)

_REG_SYNC_CONFIG = const(0x2E)

_REG_SYNC_VALUE1 = const(0x2F)

_REG_PACKET_CONFIG1 = const(0x37)

_REG_FIFO_THRESH = const(0x3C)

_REG_PACKET_CONFIG2 = const(0x3D)

_REG_AES_KEY1 = const(0x3E)
```

```
_REG_TEMP1 = const(0x4E)
_REG_TEMP2 = const(0x4F)
_REG_TEST_PA1 = const(0x5A)
_REG_TEST_PA2 = const(0x5C)
_REG_TEST_DAGC = const(0x6F)

_TEST_PA1_NORMAL = const(0x55)
_TEST_PA1_BOOST = const(0x5D)
_TEST_PA2_NORMAL = const(0x70)
_TEST_PA2_BOOST = const(0x7C)

# The crystal oscillator frequency and frequency synthesizer step size.
# See the datasheet for details of this calculation.
_FXOSC = 32000000.0
_FSTEP = _FXOSC / 524288

# RadioHead specific compatibility constants.
_RH_BROADCAST_ADDRESS = const(0xFF)
# The acknowledgement bit in the FLAGS
# The top 4 bits of the flags are reserved for RadioHead. The lower 4 bits are reserved
# for application layer use.
_RH_FLAGS_ACK = const(0x80)
_RH_FLAGS_RETRY = const(0x40)

# User facing constants:
SLEEP_MODE = 0b000
STANDBY_MODE = 0b001
FS_MODE = 0b010
```

```
TX_MODE = 0b011
RX_MODE = 0b100


# Disable the silly too many instance members warning.  Pylint has no knowledge
# of the context and is merely guessing at the proper amount of members.  This
# is a complex chip which requires exposing many attributes and state.  Disable
# the warning to work around the error.
# pylint: disable=too-many-instance-attributes



class RFM69:
    """Interface to a RFM69 series packet radio.  Allows simple sending and
    receiving of wireless data at supported frequencies of the radio
    (433/915mhz).
    :param busio.SPI spi: The SPI bus connected to the chip.  Ensure SCK, MOSI, and MISO
are
        connected.
    :param ~digitalio.DigitalInOut cs: A DigitalInOut object connected to the chip's
CS/chip select
        line.
    :param ~digitalio.DigitalInOut reset: A DigitalInOut object connected to the chip's
RST/reset
        line.
    :param int frequency: The center frequency to configure for radio transmission and
reception.
        Must be a frequency supported by your hardware (i.e. either 433 or 915mhz).
    :param bytes sync_word: A byte string up to 8 bytes long which represents the
syncronization
        word used by received and transmitted packets. Read the datasheet for a full
understanding
```

```
        of this value! However by default the library will set a value that matches the
RadioHead

        Arduino library.

    :param int preamble_length: The number of bytes to pre-pend to a data packet as a
preamble.

        This is by default 4 to match the RadioHead library.

    :param bytes encryption_key: A 16 byte long string that represents the AES
encryption key to use

        when encrypting and decrypting packets.  Both the transmitter and receiver MUST
have the

        same key value! By default no encryption key is set or used.

    :param bool high_power: Indicate if the chip is a high power variant that supports
boosted

        transmission power.  The default is True as it supports the common RFM69HCW
modules sold by

        Adafruit.

    .. note:: The D0/interrupt line is currently unused by this module and can remain
unconnected.

    Remember this library makes a best effort at receiving packets with pure Python
code.  Trying

    to receive packets too quickly will result in lost data so limit yourself to simple
scenarios

    of sending and receiving single packets at a time.

    Also note this library tries to be compatible with raw RadioHead Arduino library
communication.

    This means the library sets up the radio modulation to match RadioHead's default of
GFSK

    encoding, 250kbit/s bitrate, and 250khz frequency deviation. To change this requires
explicitly

    setting the radio's bitrate and encoding register bits. Read the datasheet and study
the init

    function to see an example of this--advanced users only! Advanced RadioHead features
like
```

```
    address/node specific packets or "reliable datagram" delivery are supported however
due to the
    limitations noted, "reliable datagram" is still subject to missed packets but with
it, the
    sender is notified if a packe has potentially been missed.
    """

    # Global buffer for SPI commands.
    _BUFFER = bytearray(4)

    class _RegisterBits:
        # Class to simplify access to the many configuration bits avaialable
        # on the chip's registers.  This is a subclass here instead of using
        # a higher level module to increase the efficiency of memory usage
        # (all of the instances of this bit class will share the same buffer
        # used by the parent RFM69 class instance vs. each having their own
        # buffer and taking too much memory).

        # Quirk of pylint that it requires public methods for a class.  This
        # is a decorator class in Python and by design it has no public methods.
        # Instead it uses dunder accessors like get and set below.  For some
        # reason pylint can't figure this out so disable the check.
        # pylint: disable=too-few-public-methods

        # Again pylint fails to see the true intent of this code and warns
        # against private access by calling the write and read functions below.
        # This is by design as this is an internally used class.  Disable the
        # check from pylint.
        # pylint: disable=protected-access
        def __init__(self, address, *, offset=0, bits=1):
            assert 0 <= offset <= 7
            assert 1 <= bits <= 8
            assert (offset + bits) <= 8
            self._address = address
            self._mask = 0
            for _ in range(bits):
                self._mask <<= 1
                self._mask |= 1
            self._mask <<= offset
            self._offset = offset

        def __get__(self, obj, objtype):
            reg_value = obj._read_u8(self._address)
            return (reg_value & self._mask) >> self._offset

        def __set__(self, obj, val):
            reg_value = obj._read_u8(self._address)
            reg_value &= ~self._mask
            reg_value |= (val & 0xFF) << self._offset
            obj._write_u8(self._address, reg_value)

    # Control bits from the registers of the chip:
    data_mode = _RegisterBits(_REG_DATA_MOD, offset=5, bits=2)
    modulation_type = _RegisterBits(_REG_DATA_MOD, offset=3, bits=2)
    modulation_shaping = _RegisterBits(_REG_DATA_MOD, offset=0, bits=2)
    temp_start = _RegisterBits(_REG_TEMP1, offset=3)
```

```
    temp_running = _RegisterBits(_REG_TEMP1, offset=2)
    sync_on = _RegisterBits(_REG_SYNC_CONFIG, offset=7)
    sync_size = _RegisterBits(_REG_SYNC_CONFIG, offset=3, bits=3)
    aes_on = _RegisterBits(_REG_PACKET_CONFIG2, offset=0)
    pa_0_on = _RegisterBits(_REG_PA_LEVEL, offset=7)
    pa_1_on = _RegisterBits(_REG_PA_LEVEL, offset=6)
    pa_2_on = _RegisterBits(_REG_PA_LEVEL, offset=5)
    output_power = _RegisterBits(_REG_PA_LEVEL, offset=0, bits=5)
    rx_bw_dcc_freq = _RegisterBits(_REG_RX_BW, offset=5, bits=3)
    rx_bw_mantissa = _RegisterBits(_REG_RX_BW, offset=3, bits=2)
    rx_bw_exponent = _RegisterBits(_REG_RX_BW, offset=0, bits=3)
    afc_bw_dcc_freq = _RegisterBits(_REG_AFC_BW, offset=5, bits=3)
    afc_bw_mantissa = _RegisterBits(_REG_AFC_BW, offset=3, bits=2)
    afc_bw_exponent = _RegisterBits(_REG_AFC_BW, offset=0, bits=3)
    packet_format = _RegisterBits(_REG_PACKET_CONFIG1, offset=7, bits=1)
    dc_free = _RegisterBits(_REG_PACKET_CONFIG1, offset=5, bits=2)
    crc_on = _RegisterBits(_REG_PACKET_CONFIG1, offset=4, bits=1)
    crc_auto_clear_off = _RegisterBits(_REG_PACKET_CONFIG1, offset=3, bits=1)
    address_filter = _RegisterBits(_REG_PACKET_CONFIG1, offset=1, bits=2)
    mode_ready = _RegisterBits(_REG_IRQ_FLAGS1, offset=7)
    rx_ready = _RegisterBits(_REG_IRQ_FLAGS1, offset=6)
    tx_ready = _RegisterBits(_REG_IRQ_FLAGS1, offset=5)
    dio_0_mapping = _RegisterBits(_REG_DIO_MAPPING1, offset=6, bits=2)
    packet_sent = _RegisterBits(_REG_IRQ_FLAGS2, offset=3)
    payload_ready = _RegisterBits(_REG_IRQ_FLAGS2, offset=2)

    def __init__(
        self,
        spi,
        cs,
        reset,
        frequency,
        *,
        sync_word=b"\x2D\xD4",
        preamble_length=4,
        encryption_key=None,
        high_power=True,
        baudrate=5000000
    ):
        self._tx_power = 13
        self.high_power = high_power
        # Device support SPI mode 0 (polarity & phase = 0) up to a max of 10mhz.
        self._device = spidev.SPIDevice(spi, cs, baudrate=baudrate, polarity=0, phase=0)
        # Setup reset as a digital output that's low.
        self._reset = reset
        self._reset.switch_to_output(value=False)
        self.reset()  # Reset the chip.
        # Check the version of the chip.
        version = self._read_u8(_REG_VERSION)
        if version != 0x24:
            raise RuntimeError(
                "Failed to find RFM69 with expected version, check wiring!"
            )
        self.idle()  # Enter idle state.
        # Setup the chip in a similar way to the RadioHead RFM69 library.
        # Set FIFO TX condition to not empty and the default FIFO threshold to 15.
        self._write_u8(_REG_FIFO_THRESH, 0b10001111)
        # Configure low beta off.
        self._write_u8(_REG_TEST_DAGC, 0x30)
        # Disable boost.
        self._write_u8(_REG_TEST_PA1, _TEST_PA1_NORMAL)
        self._write_u8(_REG_TEST_PA2, _TEST_PA2_NORMAL)
        # Set the syncronization word.
        self.sync_word = sync_word
        self.preamble_length = preamble_length  # Set the preamble length.
        self.frequency_mhz = frequency  # Set frequency.
        self.encryption_key = encryption_key  # Set encryption key.
        # set radio configuration parameters
        self._configure_radio()
        # initialize last RSSI reading
        self.last_rssi = 0.0
        """The RSSI of the last received packet. Stored when the packet was received.
           This instantaneous RSSI value may not be accurate once the
           operating mode has been changed.
        """
        # initialize timeouts and delays delays
        self.ack_wait = 0.5
        """The delay time before attempting a retry after not receiving an ACK"""
        self.receive_timeout = 0.5
        """The amount of time to poll for a received packet.
           If no packet is received, the returned packet will be None
        """
        self.xmit_timeout = 2.0
        """The amount of time to wait for the HW to transmit the packet.
```

```
           This is mainly used to prevent a hang due to a HW issue
        """
        self.ack_retries = 5
        """The number of ACK retries before reporting a failure."""
        self.ack_delay = 0.2
        """The delay time before attemting to send an ACK.
           If ACKs are being missed try setting this to .1 or .2.
        """
        # initialize sequence number counter for reliabe datagram mode
        self.sequence_number = 0
        # create seen Ids list
        self.seen_ids = bytearray(256)
        # initialize packet header
        # node address - default is broadcast
        self.node = _RH_BROADCAST_ADDRESS
        """The default address of this Node. (0-255).
           If not 255 (0xff) then only packets address to this node will be accepted.
           First byte of the RadioHead header.
        """
        # destination address - default is broadcast
        self.destination = _RH_BROADCAST_ADDRESS
        """The default destination address for packet transmissions. (0-255).
           If 255 (0xff) then any receiving node should accept the packet.
           Second byte of the RadioHead header.
        """
        # ID - contains seq count for reliable datagram mode
        self.identifier = 0
        """Automatically set to the sequence number when send_with_ack() used.
```

```
           Third byte of the RadioHead header.
        """
        # flags - identifies ack/reetry packet for reliable datagram mode
        self.flags = 0
        """Upper 4 bits reserved for use by Reliable Datagram Mode.
           Lower 4 bits may be used to pass information.
           Fourth byte of the RadioHead header.
        """

    def _configure_radio(self):
        # Configure modulation for RadioHead library GFSK_Rb250Fd250 mode
        # by default.  Users with advanced knowledge can manually reconfigure
        # for any other mode (consulting the datasheet is absolutely
        # necessary!).
        self.data_mode = 0b00  # Packet mode
        self.modulation_type = 0b00  # FSK modulation
        self.modulation_shaping = 0b01  # Gaussian filter, BT=1.0
        self.bitrate = 250000  # 250kbs
        self.frequency_deviation = 250000  # 250khz
        self.rx_bw_dcc_freq = 0b111  # RxBw register = 0xE0
        self.rx_bw_mantissa = 0b00
        self.rx_bw_exponent = 0b000
        self.afc_bw_dcc_freq = 0b111  # AfcBw register = 0xE0
        self.afc_bw_mantissa = 0b00
        self.afc_bw_exponent = 0b000
        self.packet_format = 1  # Variable length.
        self.dc_free = 0b10  # Whitening
        self.crc_on = 1  # CRC enabled
        self.crc_auto_clear = 0  # Clear FIFO on CRC fail
        self.address_filtering = 0b00  # No address filtering
        # Set transmit power to 13 dBm, a safe value any module supports.
        self.tx_power = 13

    # pylint: disable=no-member
    # Reconsider this disable when it can be tested.
    def _read_into(self, address, buf, length=None):
        # Read a number of bytes from the specified address into the provided
        # buffer.  If length is not specified (the default) the entire buffer
        # will be filled.
        if length is None:
            length = len(buf)
        with self._device as device:
            self._BUFFER[0] = address & 0x7F  # Strip out top bit to set 0
            # value (read).
            device.write(self._BUFFER, end=1)
            device.readinto(buf, end=length)

    def _read_u8(self, address):
        # Read a single byte from the provided address and return it.
        self._read_into(address, self._BUFFER, length=1)
        return self._BUFFER[0]

    def _write_from(self, address, buf, length=None):
        # Write a number of bytes to the provided address and taken from the
        # provided buffer.  If no length is specified (the default) the entire
        # buffer is written.
        if length is None:
            length = len(buf)
        with self._device as device:
            self._BUFFER[0] = (address | 0x80) & 0xFF  # Set top bit to 1 to
            # indicate a write.
            device.write(self._BUFFER, end=1)
            device.write(buf, end=length)  # send data

    def _write_fifo_from(self, buf, length=None):
        # Write a number of bytes to the transmit FIFO and taken from the
        # provided buffer.  If no length is specified (the default) the entire
        # buffer is written.
        if length is None:
            length = len(buf)
        with self._device as device:
            self._BUFFER[0] = (_REG_FIFO | 0x80) & 0xFF  # Set top bit to 1 to
            # indicate a write.
            self._BUFFER[1] = length & 0xFF  # Set packt length
            device.write(self._BUFFER, end=2)  # send address and lenght)
            device.write(buf, end=length)  # send data

    def _write_u8(self, address, val):
        # Write a byte register to the chip.  Specify the 7-bit address and the
        # 8-bit value to write to that address.
        with self._device as device:
            self._BUFFER[0] = (address | 0x80) & 0xFF  # Set top bit to 1 to
            # indicate a write.
            self._BUFFER[1] = val & 0xFF
            device.write(self._BUFFER, end=2)


    def reset(self):
        """Perform a reset of the chip."""
        # See section 7.2.2 of the datasheet for reset description.
        self._reset.value = True
        time.sleep(0.0001)  # 100 us
        self._reset.value = False
        time.sleep(0.005)  # 5 ms


    def idle(self):
        """Enter idle standby mode (switching off high power amplifiers if
necessary)."""
        # Like RadioHead library, turn off high power boost if enabled.
        if self._tx_power >= 18:
            self._write_u8(_REG_TEST_PA1, _TEST_PA1_NORMAL)
            self._write_u8(_REG_TEST_PA2, _TEST_PA2_NORMAL)
        self.operation_mode = STANDBY_MODE


    def sleep(self):
        """Enter sleep mode."""
        self.operation_mode = SLEEP_MODE


    def listen(self):
        """Listen for packets to be received by the chip.  Use :py:func:`receive` to
listen, wait
           and retrieve packets as they're available.
        """
        # Like RadioHead library, turn off high power boost if enabled.
        if self._tx_power >= 18:
            self._write_u8(_REG_TEST_PA1, _TEST_PA1_NORMAL)
            self._write_u8(_REG_TEST_PA2, _TEST_PA2_NORMAL)
        # Enable payload ready interrupt for D0 line.
        self.dio_0_mapping = 0b01
        # Enter RX mode (will clear FIFO!).
        self.operation_mode = RX_MODE

    def transmit(self):
        """Transmit a packet which is queued in the FIFO.  This is a low level function
for
           entering transmit mode and more.  For generating and transmitting a packet of
data use
           :py:func:`send` instead.
        """
        # Like RadioHead library, turn on high power boost if enabled.
        if self._tx_power >= 18:
            self._write_u8(_REG_TEST_PA1, _TEST_PA1_BOOST)
            self._write_u8(_REG_TEST_PA2, _TEST_PA2_BOOST)
        # Enable packet sent interrupt for D0 line.
        self.dio_0_mapping = 0b00
        # Enter TX mode (will clear FIFO!).
        self.operation_mode = TX_MODE

    @property
    def temperature(self):
        """The internal temperature of the chip in degrees Celsius. Be warned this is
not
           calibrated or very accurate.
           .. warning:: Reading this will STOP any receiving/sending that might be
happening!
        """
        # Start a measurement then poll the measurement finished bit.
        self.temp_start = 1
        while self.temp_running > 0:
            pass
        # Grab the temperature value and convert it to Celsius.
        # This uses the same observed value formula from the Radiohead library.
        temp = self._read_u8(_REG_TEMP2)
        return 166.0 - temp

    @property
    def operation_mode(self):
        """The operation mode value.  Unless you're manually controlling the chip you
shouldn't
           change the operation_mode with this property as other side-effects are
required for
           changing logical modes--use :py:func:`idle`, :py:func:`sleep`,
:py:func:`transmit`,
           :py:func:`listen` instead to signal intent for explicit logical modes.
        """
        op_mode = self._read_u8(_REG_OP_MODE)
        return (op_mode >> 2) & 0b111

    @operation_mode.setter
    def operation_mode(self, val):
        assert 0 <= val <= 4
        # Set the mode bits inside the operation mode register.
        op_mode = self._read_u8(_REG_OP_MODE)
        op_mode &= 0b11100011
        op_mode |= val << 2
        self._write_u8(_REG_OP_MODE, op_mode)
        # Wait for mode to change by polling interrupt bit.
        while not self.mode_ready:
            pass


    @property
    def sync_word(self):
        """The synchronization word value.  This is a byte string up to 8 bytes long (64 bits)
           which indicates the synchronization word for transmitted and received packets. Any
           received packet which does not include this sync word will be ignored. The default value
           is 0x2D, 0xD4 which matches the RadioHead RFM69 library. Setting a value of None will
           disable synchronization word matching entirely.
        """
        # Handle when sync word is disabled..
        if not self.sync_on:
            return None
        # Sync word is not disabled so read the current value.
        sync_word_length = self.sync_size + 1  # Sync word size is offset by 1
        # according to datasheet.
        sync_word = bytearray(sync_word_length)
        self._read_into(_REG_SYNC_VALUE1, sync_word)
        return sync_word


    @sync_word.setter
    def sync_word(self, val):
        # Handle disabling sync word when None value is set.
        if val is None:
            self.sync_on = 0
        else:
            # Check sync word is at most 8 bytes.
            assert 1 <= len(val) <= 8
            # Update the value, size and turn on the sync word.
            self._write_from(_REG_SYNC_VALUE1, val)
            self.sync_size = len(val) - 1  # Again sync word size is offset by
            # 1 according to datasheet.
            self.sync_on = 1

    @property
    def preamble_length(self):
        """The length of the preamble for sent and received packets, an unsigned 16-bit
value.
           Received packets must match this length or they are ignored! Set to 4 to
match the
           RadioHead RFM69 library.
        """
        msb = self._read_u8(_REG_PREAMBLE_MSB)
        lsb = self._read_u8(_REG_PREAMBLE_LSB)
        return ((msb << 8) | lsb) & 0xFFFF

    @preamble_length.setter
    def preamble_length(self, val):
        assert 0 <= val <= 65535
        self._write_u8(_REG_PREAMBLE_MSB, (val >> 8) & 0xFF)
        self._write_u8(_REG_PREAMBLE_LSB, val & 0xFF)


    @property
    def frequency_mhz(self):
        """The frequency of the radio in Megahertz. Only the allowed values for your
radio must be
           specified (i.e. 433 vs. 915 mhz)!
        """
        # FRF register is computed from the frequency following the datasheet.
        # See section 6.2 and FRF register description.
        # Read bytes of FRF register and assemble into a 24-bit unsigned value.
        msb = self._read_u8(_REG_FRF_MSB)
        mid = self._read_u8(_REG_FRF_MID)
        lsb = self._read_u8(_REG_FRF_LSB)
        frf = ((msb << 16) | (mid << 8) | lsb) & 0xFFFFFF
        frequency = (frf * _FSTEP) / 1000000.0
        return frequency


    @frequency_mhz.setter
    def frequency_mhz(self, val):
        assert 290 <= val <= 1020
        # Calculate FRF register 24-bit value using section 6.2 of the datasheet.
        frf = int((val * 1000000.0) / _FSTEP) & 0xFFFFFF
        # Extract byte values and update registers.
        msb = frf >> 16
        mid = (frf >> 8) & 0xFF
        lsb = frf & 0xFF
        self._write_u8(_REG_FRF_MSB, msb)
        self._write_u8(_REG_FRF_MID, mid)
        self._write_u8(_REG_FRF_LSB, lsb)


    @property
    def encryption_key(self):
        """The AES encryption key used to encrypt and decrypt packets by the chip. This
can be set
           to None to disable encryption (the default), otherwise it must be a 16 byte
long byte
           string which defines the key (both the transmitter and receiver must use the
same key
           value).
        """
        # Handle if encryption is disabled.
        if self.aes_on == 0:
            return None
        # Encryption is enabled so read the key and return it.
        key = bytearray(16)
        self._read_into(_REG_AES_KEY1, key)
        return key


    @encryption_key.setter
    def encryption_key(self, val):
        # Handle if unsetting the encryption key (None value).
        if val is None:
            self.aes_on = 0
        else:
            # Set the encryption key and enable encryption.
            assert len(val) == 16
            self._write_from(_REG_AES_KEY1, val)
            self.aes_on = 1


    @property
    def tx_power(self):
        """The transmit power in dBm. Can be set to a value from -2 to 20 for high power
devices
           (RFM69HCW, high_power=True) or -18 to 13 for low power devices. Only integer
power
           levels are actually set (i.e. 12.5 will result in a value of 12 dBm).
        """
        # Follow table 10 truth table from the datasheet for determining power
        # level from the individual PA level bits and output power register.
        pa0 = self.pa_0_on
        pa1 = self.pa_1_on
        pa2 = self.pa_2_on
        if pa0 and not pa1 and not pa2:
            # -18 to 13 dBm range
            return -18 + self.output_power
        if not pa0 and pa1 and not pa2:
            # -2 to 13 dBm range
            return -18 + self.output_power
        if not pa0 and pa1 and pa2 and not self.high_power:
            # 2 to 17 dBm range
            return -14 + self.output_power
        if not pa0 and pa1 and pa2 and self.high_power:
            # 5 to 20 dBm range
            return -11 + self.output_power
        raise RuntimeError("Power amplifiers in unknown state!")
    @tx_power.setter
    def tx_power(self, val):
        val = int(val)
        # Determine power amplifier and output power values depending on
        # high power state and requested power.
        pa_0_on = 0
        pa_1_on = 0
        pa_2_on = 0
        output_power = 0
        if self.high_power:
            # Handle high power mode.
            assert -2 <= val <= 20
            if val <= 13:
                pa_1_on = 1
                output_power = val + 18
            elif 13 < val <= 17:
                pa_1_on = 1
                pa_2_on = 1
                output_power = val + 14
            else:  # power >= 18 dBm
                # Note this also needs PA boost enabled separately!
                pa_1_on = 1
                pa_2_on = 1
                output_power = val + 11
        else:
            # Handle non-high power mode.
            assert -18 <= val <= 13
            # Enable only power amplifier 0 and set output power.
            pa_0_on = 1
            output_power = val + 18
        # Set power amplifiers and output power as computed above.
        self.pa_0_on = pa_0_on
        self.pa_1_on = pa_1_on
        self.pa_2_on = pa_2_on
        self.output_power = output_power
        self._tx_power = val

    @property
    def rssi(self):
        """The received strength indicator (in dBm).
           May be inaccuate if not read immediatey. last_rssi contains the value read
immediately
           receipt of the last packet.
        """
        # Read RSSI register and convert to value using formula in datasheet.
        return -self._read_u8(_REG_RSSI_VALUE) / 2.0

    @property
    def bitrate(self):
        """The modulation bitrate in bits/second (or chip rate if Manchester encoding is
enabled).
           Can be a value from ~489 to 32mbit/s, but see the datasheet for the exact
supported
           values.
        """
        msb = self._read_u8(_REG_BITRATE_MSB)
        lsb = self._read_u8(_REG_BITRATE_LSB)
        return _FXOSC / ((msb << 8) | lsb)

    @bitrate.setter
    def bitrate(self, val):
        assert (_FXOSC / 65535) <= val <= 32000000.0
        # Round up to the next closest bit-rate value with addition of 0.5.
        bitrate = int((_FXOSC / val) + 0.5) & 0xFFFF
        self._write_u8(_REG_BITRATE_MSB, bitrate >> 8)
        self._write_u8(_REG_BITRATE_LSB, bitrate & 0xFF)

    @property
    def frequency_deviation(self):
        """The frequency deviation in Hertz."""
        msb = self._read_u8(_REG_FDEV_MSB)
        lsb = self._read_u8(_REG_FDEV_LSB)
        return _FSTEP * ((msb << 8) | lsb)

    @frequency_deviation.setter
    def frequency_deviation(self, val):
        assert 0 <= val <= (_FSTEP * 16383)  # fdev is a 14-bit unsigned value
        # Round up to the next closest integer value with addition of 0.5.
        fdev = int((val / _FSTEP) + 0.5) & 0x3FFF
        self._write_u8(_REG_FDEV_MSB, fdev >> 8)
        self._write_u8(_REG_FDEV_LSB, fdev & 0xFF)

    def send(
        self,
        data,
        *,
        keep_listening=False,
        destination=None,
        node=None,
        identifier=None,
        flags=None
    ):
        """Send a string of data using the transmitter.
           You can only send 60 bytes at a time
           (limited by chip's FIFO size and appended headers).
           This appends a 4 byte header to be compatible with the RadioHead library.
           The header defaults to using the initialized attributes:
           (destination,node,identifier,flags)
           It may be temporarily overidden via the kwargs - 
destination,node,identifier,flags.
           Values passed via kwargs do not alter the attribute settings.
           The keep_listening argument should be set to True if you want to start 
listening
           automatically after the packet is sent. The default setting is False.
           Returns: True if success or False if the send timed out.
        """
        # Disable pylint warning to not use length as a check for zero.
        # This is a puzzling warning as the below code is clearly the most
        # efficient and proper way to ensure a precondition that the provided
        # buffer be within an expected range of bounds.  Disable this check.
        # pylint: disable=len-as-condition
        assert 0 < len(data) <= 60
        # pylint: enable=len-as-condition
        self.idle()  # Stop receiving to clear FIFO and keep it clear.
        # Fill the FIFO with a packet to send.

        # Combine header and data to form payload

        payload = bytearray(4)

        if destination is None:  # use attribute

            payload[0] = self.destination

        else:  # use kwarg

            payload[0] = destination

        if node is None:  # use attribute

            payload[1] = self.node

        else:  # use kwarg

            payload[1] = node

        if identifier is None:  # use attribute

            payload[2] = self.identifier

        else:  # use kwarg

            payload[2] = identifier

        if flags is None:  # use attribute

            payload[3] = self.flags

        else:  # use kwarg

            payload[3] = flags

        payload = payload + data

        # Write payload to transmit fifo

        self._write_fifo_from(payload)

        # Turn on transmit mode to send out the packet.

        self.transmit()

        # Wait for packet sent interrupt with explicit polling (not ideal but

        # best that can be done right now without interrupts).

        start = time.monotonic()

        timed_out = False
```

```python
        while not timed_out and not self.packet_sent:
            if (time.monotonic() - start) >= self.xmit_timeout:
                timed_out = True
        # Listen again if requested.
        if keep_listening:
            self.listen()
        else:  # Enter idle mode to stop receiving other packets.
            self.idle()
        return not timed_out

    def send_with_ack(self, data):
        """Reliable Datagram mode:
           Send a packet with data and wait for an ACK response.
           The packet header is automatically generated.
           If enabled, the packet transmission will be retried on failure
        """
        if self.ack_retries:
            retries_remaining = self.ack_retries
        else:
            retries_remaining = 1
        got_ack = False
        self.sequence_number = (self.sequence_number + 1) & 0xFF
        while not got_ack and retries_remaining:
            self.identifier = self.sequence_number
            self.send(data, keep_listening=True)
            # Don't look for ACK from Broadcast message
            if self.destination == _RH_BROADCAST_ADDRESS:
                got_ack = True
            else:
                # wait for a packet from our destination
                ack_packet = self.receive(timeout=self.ack_wait, with_header=True)
                if ack_packet is not None:
                    if ack_packet[3] & _RH_FLAGS_ACK:
                        # check the ID
                        if ack_packet[2] == self.identifier:
                            got_ack = True
                            break
            # pause before next retry -- random delay
            if not got_ack:
                # delay by random amount before next try
                time.sleep(self.ack_wait + self.ack_wait * random.random())
            retries_remaining = retries_remaining - 1
            # set retry flag in packet header
            self.flags |= _RH_FLAGS_RETRY
        self.flags = 0  # clear flags
        return got_ack

    # pylint: disable=too-many-branches
    def receive(
        self, *, keep_listening=True, with_ack=False, timeout=None, with_header=False
    ):
        """Wait to receive a packet from the receiver. If a packet is found the payload
bytes
           are returned, otherwise None is returned (which indicates the timeout elapsed
with no
           reception).
           If keep_listening is True (the default) the chip will immediately enter
listening mode

           after reception of a packet, otherwise it will fall back to idle mode and
ignore any

           future reception.

           All packets must have a 4 byte header for compatibilty with the

           RadioHead library.

           The header consists of 4 bytes (To,From,ID,Flags). The default setting will
strip

           the header before returning the packet to the caller.

           If with_header is True then the 4 byte header will be returned with the
packet.

           The payload then begins at packet[4].

           If with_ack is True, send an ACK after receipt (Reliable Datagram mode)

        """

        timed_out = False

        if timeout is None:

            timeout = self.receive_timeout

        if timeout is not None:

            # Wait for the payload_ready signal.  This is not ideal and will

            # surely miss or overflow the FIFO when packets aren't read fast

            # enough, however it's the best that can be done from Python without

            # interrupt supports.

            # Make sure we are listening for packets.

            self.listen()

            start = time.monotonic()

            timed_out = False

            while not timed_out and not self.payload_ready:

                if (time.monotonic() - start) >= timeout:

                    timed_out = True
        # Payload ready is set, a packet is in the FIFO.
        packet = None
        # save last RSSI reading
        self.last_rssi = self.rssi
        # Enter idle mode to stop receiving other packets.
        self.idle()
        if not timed_out:
            # Read the length of the FIFO.
            fifo_length = self._read_u8(_REG_FIFO)
            # Handle if the received packet is too small to include the 4 byte
            # RadioHead header and at least one byte of data --reject this packet and
ignore it.
            if fifo_length > 0:  # read and clear the FIFO if anything in it
                packet = bytearray(fifo_length)
                self._read_into(_REG_FIFO, packet)
            if fifo_length < 5:
                packet = None
            else:
                if (
                    self.node != _RH_BROADCAST_ADDRESS
                    and packet[0] != _RH_BROADCAST_ADDRESS
                    and packet[0] != self.node
                ):
                    packet = None
                # send ACK unless this was an ACK or a broadcast
                elif (
                    with_ack
                    and ((packet[3] & _RH_FLAGS_ACK) == 0)
                    and (packet[0] != _RH_BROADCAST_ADDRESS)
                ):
                    # delay before sending Ack to give receiver a chance to get ready
                    if self.ack_delay is not None:
                        time.sleep(self.ack_delay)
                    # send ACK packet to sender
                    data = bytes("!", "UTF-8")
                    self.send(
                        data,
                        destination=packet[1],
                        node=packet[0],
                        identifier=packet[2],
                        flags=(packet[3] | _RH_FLAGS_ACK),
                    )
                    # reject Retries if we have seen this idetifier from this source
before
                    if (self.seen_ids[packet[1]] == packet[2]) and (
                        packet[3] & _RH_FLAGS_RETRY
                    ):
                        packet = None
                    else:  # save the packet identifier for this source
                        self.seen_ids[packet[1]] = packet[2]
                if (
                    not with_header and packet is not None
                ):  # skip the header if not wanted
                    packet = packet[4:]
        # Listen again if necessary and return the result packet.
        if keep_listening:
            self.listen()
        else:
```

```
    # Enter idle mode to stop receiving other packets.

    self.idle()

return packet
```

RADIO – radio.py

```
#!/usr/bin/env
python3

'''

API for interacting with Radio subsystem

'''



import time

import re

import logging

import math


# Adafruit

import board

import busio

import digitalio

import adafruit_rfm69


COMMANDS = {

    "ping": '\x00'


}
```

```
class RADIO:

    def __init__(self, sync_word=b"\x2D\xD4", frequency=915.0):
        """ Initialize radio and serial connection """

        self.logger = logging.getLogger("radio-service")
        self.logger.setLevel(logging.DEBUG)

        self.frequency = frequency
        self.sync_word = sync_word

        self.timeout_sec = 60

        #self.baud_rate = 9600

        # set pins
        self.cs = digitalio.DigitalInOut(board.P9_25)
        self.reset = digitalio.DigitalInOut(board.P9_27)
        self.led = digitalio.DigitalInOut(board.P8_8)

        # set pin direction
        self.cs.direction = digitalio.Direction.OUTPUT
        self.reset.direction = digitalio.Direction.OUTPUT
        self.led.direction = digitalio.Direction.OUTPUT

        # setup spi
        self.spi = busio.SPI(board.SCLK_1, board.MOSI_1, board.MISO_1)
        # using Adafruit rfm 69 radio module

        self.radio = adafruit_rfm69.RFM69(self.spi, self.cs, self.reset,
self.frequency, sync_word=self.sync_word)


        # set encryption key - must be same on other end at other transceiver

        self.radio.encryption_key =
(b"\x01\x02\x03\x04\x05\x06\x07\x08\x01\x02\x03\x04\x05\x06\x07\x08")


        self.logger.debug("Finished radio initialization.")


    def ping(self):

        """ For testing communications service connection """

        return "pong"


    def timeout(self, start_time):

        if (time.time() - start_time > self.timeout_sec):

            return True

        else:

            return False


    def write(self, frame):

        start_time = time.time()

        # keep sending frame until we get confirmed returned acknowledgement

        while (self.radio.send_with_ack(frame) == False):

            # check for timeout

            if self.timeout(start_time):

                self.logger.warn("Timeout trying to write packet to radio.")

                return False
            else:
                continue

        self.logger.debug("Wrote packet to radio: {}".format(frame))
        return True

    def read(self):
        start_time = time.time()
        while (True):
            frame = self.radio.receive()

            # check for timeout
            if self.timeout(start_time):
                self.logger.warn("Timeout trying to read a packet.")
                return False, []

            if frame is None:
                continue
            else:
                break

        self.logger.debug("Read packet: {}".format(frame))
        return True, frame


    def downlink_image(self, filename):

        with open(filename, "rb") as image:
            f = image.read()
            b = bytearray(f)

            print(b)

            size = 60
            num = math.ceil(len(b)/60)

            # split image into separate frames
            frames = []
            for i in range(num):
                frames.append(b[(i*size):((i+1)*size)])
                print("Frame {}: {}".format(i, frames[i]))

            for n, frame in enumerate(frames):
                while self.radio.send_with_ack(frame) is False:
                    continue
                print("Frame {}/{}".format(n, len(frames)))
                time.sleep(0.1)

            self.radio.send_with_ack(bytearray(b'\x00\x01'))
            print("Sent {}".format(bytearray(b'\x00\x01')))

    # decode radio protocol (AX_25)
    def decode(self, frame):
        opcode = bytes(frame[:1]).decode('utf-8')
        payload = bytes(frame[1:]).decode('utf-8')
        #opcode = opcode.decode('utf-8')

        #payload = opcode.decode('utf-8')


        self.logger.debug("Decoded packet into opcode: {} payload:
{}".format(opcode, payload))

        return opcode, payload


    # encode message into radio protocol for sending (AX_25)

    def encode(self, opcode, payload):

        frame = bytearray()

        frame.extend(bytearray(opcode, 'utf-8'))

        frame.extend(bytearray(payload, 'utf-8'))


        self.logger.debug("Encoded opcode: {} and payload: {} into frame:
{}".format(opcode, payload, frame))

        return frame


    def main(self):


        self.logger.info("Starting main loop on radio for receiving packets")

        while (True):

            success, frame = self.read()


            if not success:

                continue


            opcode, payload = self.decode(frame)


            # received ping, send back acknowledgement
            if (opcode == COMMANDS['ping']):
                self.logger.debug("Got 'ping' command from ground station. 
Returning acknowledgement")
                frame = self.encode(opcode, '\x00')
            else:
                self.logger.warn("Received command/opcode that is not in valid 
opcodes: {}.".format(opcode))
                continue

            # return acknowledgement
            self.write(frame)

    def __exit__(self):
        pass
```

ADCS – adcs.py

```
#!/usr/bin/env
python3

'''
API for interacting with ADCS subsystem
'''

from obcserial import i2c
import logging

class ADCS:

    def __init__(self):
        self.logger = logging.getLogger("adcs-service")

        # telemetry defaults
        self._power = 0  # (OFF=0,ON=1,RESET=2)
        self._mode = 0   # (IDLE=0,DETUMBLE=1,POINTING=2)
        self._orientation = [1.0, 1.0, 1.0, 0.0, 0.0, 0.0]
        self._spin = [1.0, 1.0, 1.0]

        # ADCS initialize stuff here

################ queries ###################
    def ping(self):
        # should send hardware a ping and expect a pong back
        return "pong"

    def power(self):
        #############################################################
        # TODO: Get power state (ON=1, OFF=0, RESET=2) from ADCS here
        #############################################################
        return self._power

    def mode(self):
        ####################################################################
        # TODO: Get mode state (IDLE=1,DETUMBLE=0,POINTING=2) from ADCS here
        ####################################################################
        return self._mode
    def orientation(self):
        ##############################################################
        # TODO: Get orientation (x,y,z,yaw,pitch,roll) from ADCS here
        ##############################################################
        return self._orientation

    def spin(self):
        #######################################
        # TODO: Get spin (x,y,z) from ADCS here
        #######################################
        return self._spin

################ mutations #################
    # Controls the power state of the ADCS
    def controlPower(self, controlPowerInput):
        success = False
        errors = []
        try:
            self.logger.info("Sending new power state to ADCS...")
            power=controlPowerInput.power
            ####################################################################
            # TODO: Set mode state (IDLE=1,DETUMBLE=0,POINTING=2) from ADCS here
            ####################################################################
            success = True
            errors = []
            self._power = power

            if success:
                self.logger.info("Set ADCS power state={}".format(power))
            else:
                self.logger.error("Unable to set ADCS power state={}".format(power))

        except Exception as e:
            self.logger.error("Exception trying to set ADCS power state={} : {}".format(power, str(e)))
            success = False
            errors = [str(e)]

        finally:
            return success, errors

    def setMode(self, setModeInput):
        success = False
        errors = []
        try:
            self.logger.info("Sending new mode to ADCS...")
            mode=setModeInput.mode
            ####################################################################
            # TODO: Set mode state (IDLE=1,DETUMBLE=0,POINTING=2) from ADCS here
            ####################################################################
            success = True
            errors = []
            self._mode = mode

            if success:
                self.logger.info("Set ADCS mode={}".format(mode))
            else:

                self.logger.error("Unable to set ADCS mode={}".format(mode))


        except Exception as e:

            self.logger.error("Exception trying to set ADCS mode={} :
{}".format(mode, str(e)))

            success = False

            errors = [str(e)]


        finally:

            return success, errors
```

EPS – eps.py

```
#!/usr/bin/env python3


'''

API for interacting with EPS subsystem

'''



import time

import re

import logging


# for controlling power ports

from obcserial import gpio

# for reading battery levels through ADC
```

```
from obcapi import ADS1115

class EPS:

    def __init__(self):

        self.logger = logging.getLogger("eps-service")

########################### SUNFLOWER SOLAR POWER MANAGER ##############################
        # initialize GPIO pins
        self.PORT1 = gpio.GPIO(66)
        self.PORT2 = gpio.GPIO(69)
        self.PORT3 = gpio.GPIO(45)

        self.fake_GPIO = True

        self.PORT1.release()
        self.PORT2.release()
        self.PORT3.release()

        # attach GPIO pins
        if self.PORT1.attach():
            if self.PORT2.attach():
                if self.PORT3.attach():
                    self.fake_GPIO = False

        if self.fake_GPIO == False:
            # set GPIO direction
            self.PORT1.direction(1)
            self.PORT2.direction(1)
            self.PORT3.direction(1)

            # make sure everything is off
            self.PORT1.off()
            self.PORT2.off()
            self.PORT3.off()

        self.power1 = False # power to output 1 off
        self.power2 = False # power to output 2 off
        self.power3 = False # power to output 3 off

############################# ADAFRUIT ADS1115 ADC ######################################

        # using Adafruit ADS1115 16-bit, 4 channel ADC
        self.adc = ADS1115.ADS1115()

        # Start continuous ADC conversions on channel 0 using the previously set gain
        # value.  Note you can also pass an optional data_rate parameter, see the
simpletest.py
        # example and read_adc function for more infromation.

        # Choose a gain of 1 for reading voltages from 0 to 4.09V.
        # Or pick a different gain to change the range of voltages that are read:
        #  - 2/3 = +/-6.144V
        #  -   1 = +/-4.096V
        #  -   2 = +/-2.048V
        #  -   4 = +/-1.024V
```

```python
        #  -   8 = +/-0.512V
        #  -  16 = +/-0.256V
        self.adc.start_adc(channel=0, gain=1)

        # Once continuous ADC conversions are started you can call get_last_result() to
        # retrieve the latest result, or stop_adc() to stop conversions.

        # adc reading on full battery
        # usually 2^16/2 but using a voltage divider
        self.max_adc_reading = 24490

        # max voltage from sunflower solar manager that means full battery
        self.full_battery_voltage = 4.2

    # test service connection
    def ping(self):
        return "pong"

    # control power of ports
    def controlPort(self, controlPortInput):
        if self.fake_GPIO:
            if (controlPortInput.power == 1):
                self.power1 = True
            elif (controlPortInput.power == 2):
                self.power2 = True
            elif (controlPortInput.power == 3):
                self.power3 = True
            return True
```

```
if (controlPortInput.port == 1):

    if (controlPortInput.power == 1):

        self.PORT1.on()

        self.power1 = True

    else:

        self.PORT1.off()

        self.power1 = False

    return True

elif (controlPortInput.port == 2):

    if (controlPortInput.power == 1):

        self.PORT2.on()

        self.power2 = True

    else:

        self.PORT2.off()

        self.power2 = False

    return True

elif (controlPortInput.port == 3):

    if (controlPortInput.power == 1):

        self.PORT3.on()

        self.power3= True

    else:

        self.PORT3.off()

        self.power3 = False

    return True

else:

    return False
    # get power state of all ports
    def power(self):
        return self.power1, self.power2, self.power3


    # get current battery level as percentage
    def battery(self):


        # read ADC value
        value = self.adc.get_last_result()
        # get percentage of highest reading
        if value < self.max_adc_reading:
            percentage = float(value)/float(self.max_adc_reading)
        else:
            percentage = 1.0


        # convert to voltage
        voltage = percentage * self.full_battery_voltage


        return int(percentage * 100)


    # release GPIO pins
    def __exit__(self):
        if self.fake_GPIO == False:
            self.PORT1.release()
            self.PORT2.release()
            self.PORT3.release()


        # stop adc
```

```
            self.adc.stop_adc()
```

PAYLOAD – payload.py

```
#!/usr/bin/env python3

'''
API for interacting with Payload subsystem
'''


import time
import re
from obcserial import uart
from obcapi import config
import app_api


class Payload:


    def __init__(self):


        self.logger = app_api.logging_setup("payload-api")
        self.use_uart = False
        try:
            self.UART = uart.UART(1)
            self.use_uart = True
        except Exception as e:
            self.logger.error("FATAL ERROR: Unable to open UART port {}:{}. No communication with payload. Using fake connection...".format(type(e).__name__, str(e)))
    def read(self):
        if self.use_uart:
            success, message = self.UART.read()
            if success:
                return self.unpack(message)
            return None
        else:
            return self.unpack(input("Enter fake serial input:"))


    def send_command(self, command):
        if self.write(command):
            for i in range(config.RETRY):
                response = self.read()
                if response == config.RETURN_CODE[0]:
                    self.logger.info("Got back {} message from payload sending command: {}.".format(response, command))
                    return True
                elif response == config.RETURN_CODE[1]:
                    self.logger.warn("Got back {} message from payload sending command: {}.".format(response, command))
                    return False
            return False
        else:
            self.logger.warn("Error writing {} over UART.".format(command))
            return False


    def read_result(self, expected_response):
        for i in range(config.RETRY):
            response = self.read()
            if response == expected_response:
                self.logger.info("Got back successful {} message from
payload.".format(response))
                return True
            elif response == config.RETURN_CODE[1]:
                self.logger.warn("Got back {} message from payload sending
command.".format(response))
                return False
            else:
                self.logger.warn("Got back {} message from payload but expecting:
{}.".format(response, expected_response))
        return False

    def pack(self, message):
        return "<<" + message + ">>"

    def unpack(self, message):
        commands = re.findall(config.REGEX, message)
        if len(commands) != 0:
            return commands[0]
        else:
            return None

    def write(self, command):
        if self.use_uart:
            return self.UART.write(self.pack(command))
        else:
            print(self.pack(command))
            return True


    def read_image(self, filename):
        return self.UART.transfer_image(filename)



######## QUERIES AND MUTATIONS ################
    def ping(self):
        return "pong"
        # should send hardware a ping and expect a pong back
        #try:
        #    command = "ping"
        #    response = "pong"
        #    if self.send_command(command):
        #        if self.read_result(response):
        #            return True, []
        #        else:
        #            return False, ["Payload did not send back sucessful result
from {}".format(command)]
        #    return False, ["Could not send payload successful
{}".format(command)]
        #
        #except Exception as e:
        #    return False, [str(e)]


    def image_capture(self):
        # should send request to capture an image
        try:
            command = "capture_image"
            response = config.RETURN_CODE[2]
            if self.send_command(command):
                if self.read_result(response):
                    return True, []
                else:
                    return False, ["Payload did not send back successful result
from {}".format(command)]
            return False, ["Could not send payload successful
{}".format(command)]


        except Exception as e:
            return False, [str(e)]


    def image_transfer(self):
        # should send request to start image transfer and then open a stream to
read image
        try:
            command = "transfer_image"
            response = config.RETURN_CODE[2]
            if self.send_command(command):
                time.sleep(1)
                self.write("START")
                if self.read_image("/home/kubos/images/image.jpg"):
                    if self.read_result(response):
                        return True, []
                    else:
                        return False, ["Payload did not send back successful
result from {}".format(command)]
                else:
                    return False, ["Error trying to read image from Payload."]
            return False, ["Could not send payload successful
{}".format(command)]


        except Exception as e:

            return False, [str(e)]
```

APPENDIX J

Cross-Polarized Yagi-Uda Antenna Design - 144MHz Uplink Antenna

```
%Based on M2 Antenna Systems, Inc. - 2MCP8A
%https://www.m2inc.com/FGLEOPACK
%https://www.m2inc.com/FG2MCP8A

% create a yagiUda object called "yagiAntenna1Horizontal"
yagiAntenna1Horizontal = yagiUda;
% create a yagiUda object called "yagiAntenna1Vertical"
yagiAntenna1Vertical = yagiUda;

% set the exciters of the antennas to a dipoleFolded object
yagiAntenna1Horizontal.Exciter = dipoleFolded;
yagiAntenna1Vertical.Exciter = dipoleFolded;
% set the width of the exciters
yagiAntenna1Horizontal.Exciter.Width =
cylinder2strip(convlength(3/16,'in','m')/2);
yagiAntenna1Vertical.Exciter.Width =
cylinder2strip(convlength(3/16,'in','m')/2);
% set the lengths of the exciters (311 mm from the datasheet)
yagiAntenna1Horizontal.Exciter.Length = convlength(39.063,'in','m');
yagiAntenna1Vertical.Exciter.Length = convlength(39.25,'in','m');
% set the spacings of the exciters
%yagiAntenna1XXXX.Exciter.Length/50
yagiAntenna1Horizontal.Exciter.Spacing = 0.0197;
yagiAntenna1Vertical.Exciter.Spacing = 0.0198;
% set the number of directors (13 from the datasheet)
yagiAntenna1Horizontal.NumDirectors = 2;
yagiAntenna1Vertical.NumDirectors = 2;
% set the director lengths (D1 - D13 from the datasheet)
yagiAntenna1Horizontal.DirectorLength = convlength([37.562,
36.00],'in','m');
yagiAntenna1Vertical.DirectorLength = convlength([37.562, 36.00],'in','m');
% set the director spacings (8.375 in from the datasheet)
horizontalOffset = [13.0, 24.5];
verticalOffset = [33.188, 44.688];
yagiAntenna1Horizontal.DirectorSpacing = convlength([24.5, 42.5]-
horizontalOffset,'in','m');
yagiAntenna1Vertical.DirectorSpacing = convlength([44.688, 62.688]-
verticalOffset,'in','m');
% set the reflector lengths (330 mm from the datasheet)
yagiAntenna1Horizontal.ReflectorLength = convlength(41.250,'in','m');
yagiAntenna1Vertical.ReflectorLength = convlength(41.250,'in','m');
```

```
% set the reflector spacings (136.75 mm from the datasheet)
yagiAntenna1Horizontal.ReflectorSpacing = convlength(13.0-7.0,'in','m');
yagiAntenna1Vertical.ReflectorSpacing = convlength(33.188-27.188,'in','m');

% set the tilt of the first antenna
yagiAntenna1Horizontal.Tilt = 45;
% set the tilt axis of the first antenna
yagiAntenna1Horizontal.TiltAxis = 'Z';
% set the tilt of the second antenna
yagiAntenna1Vertical.Tilt = -45;
% set the tilt axis of the second antenna
yagiAntenna1Vertical.TiltAxis = 'Z';

% create a conformal array to represent the composite antenna
crossPolYagiAntenna = conformalArray;
% fill the conformal array with the antennas
crossPolYagiAntenna.Element = [yagiAntenna1Horizontal yagiAntenna1Vertical];
% put the first element at the origin
crossPolYagiAntenna.ElementPosition(1,:) = [0 0 0];
% offset the antenna in the other plane
crossPolYagiAntenna.ElementPosition(2,:) = [0 0 convlength(27.188-
7.0,'in','m')];
% apply a 90 degree phase shift between the two antennas
crossPolYagiAntenna.PhaseShift = [0 90];

% show the Cross-Polarized Yagi-Uda antenna
figure;
show(crossPolYagiAntenna);

% setup variables for analysis
frequency = 144e6;
azimuth_range = 0:1:360;
elevation_range = -90:1:90;
% show the antenna radiation pattern at 435 MHz
figure;
pattern(crossPolYagiAntenna,frequency,azimuth_range,elevation_range);
title("GS Yagi Uplink Antenna - 144MHz");

% azimuth for yagiuda
figure;
patternAzimuth(crossPolYagiAntenna, frequency)
% elevation for yagiuda
figure;
patternElevation(crossPolYagiAntenna, frequency)

% output the radiation pattern into variables
[gain,phi,theta] =
pattern(crossPolYagiAntenna,frequency,azimuth_range,elevation_range);

% computer the front-to-back-ratio
d_max = pattern(crossPolYagiAntenna,frequency,0,90);
d_back = pattern(crossPolYagiAntenna,frequency,0,-90);
fb_ratio = d_max - d_back
```

```
% computer the eplane and hplane beamwidths
eplane_beamwidth = beamwidth(crossPolYagiAntenna,frequency,0,1:1:360)
hplane_beamwidth = beamwidth(crossPolYagiAntenna,frequency,90,1:1:360)

saveToSTK('crossPolarizedYagiGS_144MHzUplink_New.txt', phi, theta, gain);
```

Cross-Polarized Yagi-Uda Antenna Design - 437MH DownUplink Antenna

```
%Based on M2 Antenna Systems, Inc. - 436CP16
%https://www.m2inc.com/FGLEOPACK
%https://www.m2inc.com/FG436CP16

% create a yagiUda object called "yagiAntenna1Horizontal"
yagiAntenna1Horizontal = yagiUda;
% create a yagiUda object called "yagiAntenna1Vertical"
yagiAntenna1Vertical = yagiUda;

% set the exciters of the antennas to a dipoleFolded object
yagiAntenna1Horizontal.Exciter = dipoleFolded;
yagiAntenna1Vertical.Exciter = dipoleFolded;
% set the width of the exciters
yagiAntenna1Horizontal.Exciter.Width =
cylinder2strip(convlength(3/16,'in','m')/2);
yagiAntenna1Vertical.Exciter.Width =
cylinder2strip(convlength(3/16,'in','m')/2);
% set the lengths of the exciters (311 mm from the datasheet)
yagiAntenna1Horizontal.Exciter.Length = convlength(13.625,'in','m');
yagiAntenna1Vertical.Exciter.Length = convlength(13.625,'in','m');
% set the spacings of the exciters
%yagiAntenna1Horizontal.Exciter.Length/50
yagiAntenna1Horizontal.Exciter.Spacing = 0.0068;%convlength(1.122,'in','m');
yagiAntenna1Vertical.Exciter.Spacing = 0.0068;%convlength(1.122,'in','m');
% set the number of directors (13 from the datasheet)
yagiAntenna1Horizontal.NumDirectors = 6;
yagiAntenna1Vertical.NumDirectors = 6;
% set the director lengths (D1 - D13 from the datasheet)
yagiAntenna1Horizontal.DirectorLength = convlength([12.625, 12.250, 11.937,
11.750, 11.531, 11.375],'in','m');
yagiAntenna1Vertical.DirectorLength = convlength([12.625, 12.250, 11.937,
11.750, 11.531, 11.375],'in','m');
% set the director spacings (8.375 in from the datasheet)
horizontalOffset = [14.562, 17.0, 23.313, 31.875, 41.813, 51.563];
verticalOffset = [21.312, 23.750, 30.063, 38.625, 48.563, 58.313];
yagiAntenna1Horizontal.DirectorSpacing = convlength([17.0, 23.313, 31.875,
41.813, 51.563, 61.000]-horizontalOffset,'in','m');
yagiAntenna1Vertical.DirectorSpacing = convlength([23.750, 30.063, 38.625,
48.563, 58.313, 67.750]-verticalOffset,'in','m');
% set the reflector lengths (330 mm from the datasheet)
yagiAntenna1Horizontal.ReflectorLength = convlength(13.687,'in','m');
yagiAntenna1Vertical.ReflectorLength = convlength(13.687,'in','m');
% set the reflector spacings (136.75 mm from the datasheet)
```

```
yagiAntenna1Horizontal.ReflectorSpacing = convlength(14.562-
12.000,'in','m');
yagiAntenna1Vertical.ReflectorSpacing = convlength(21.312-18.750,'in','m');

% set the tilt of the first antenna
yagiAntenna1Horizontal.Tilt = 45;
% set the tilt axis of the first antenna
yagiAntenna1Horizontal.TiltAxis = 'Z';
% set the tilt of the second antenna
yagiAntenna1Vertical.Tilt = -45;
% set the tilt axis of the second antenna
yagiAntenna1Vertical.TiltAxis = 'Z';

% create a conformal array to represent the composite antenna
crossPolYagiAntenna = conformalArray;
% fill the conformal array with the antennas
crossPolYagiAntenna.Element = [yagiAntenna1Horizontal yagiAntenna1Vertical];
% put the first element at the origin
crossPolYagiAntenna.ElementPosition(1,:) = [0 0 0];
% offset the antenna in the other plane
crossPolYagiAntenna.ElementPosition(2,:) = [0 0 convlength(18.750-
12.0,'in','m')];
% apply a 90 degree phase shift between the two antennas
crossPolYagiAntenna.PhaseShift = [0 90];

% show the Cross-Polarized Yagi-Uda antenna
figure;
show(crossPolYagiAntenna);

% setup variables for analysis
frequency = 437e6;
azimuth_range = 0:1:360;
elevation_range = -90:1:90;
% show the antenna radiation pattern at 435 MHz
figure;
pattern(crossPolYagiAntenna,frequency,azimuth_range,elevation_range);

% azimuth for yagiuda
figure;
patternAzimuth(crossPolYagiAntenna, frequency)
% elevation for yagiuda
figure;
patternElevation(crossPolYagiAntenna, frequency)

% output the radiation pattern into variables
[gain,phi,theta] =
pattern(crossPolYagiAntenna,frequency,azimuth_range,elevation_range);

% computer the front-to-back-ratio
d_max = pattern(crossPolYagiAntenna,frequency,0,90);
d_back = pattern(crossPolYagiAntenna,frequency,0,-90);
fb_ratio = d_max - d_back
% computer the eplane and hplane beamwidths
```

```
eplane_beamwidth = beamwidth(crossPolYagiAntenna,frequency,0,1:1:360)
hplane_beamwidth = beamwidth(crossPolYagiAntenna,frequency,90,1:1:360)

saveToSTK('crossPolarizedYagiGS_437MHzDownlink_New.txt', phi, theta, gain);
```

Dipole 144MHz

```
%% Antenna Properties
% Design antenna at frequency 144000000Hz
antennaObject = design(dipole,144000000);
% Update load properties
antennaObject.Load.Impedance = 50;
azimuth_range = 0:1:360;
elevation_range = -90:1:90;

%% Antenna Analysis
% Define plot frequency
plotFrequency = 144e6;
% Define frequency range
freqRange = (129.6:1.44:158.4) * 1e6;
% show for dipole
figure;
show(antennaObject)
% calculate the beamwidth of the antenna. Antenna beamwidth is the angular
measure of the antenna pattern coverage.
% Beamwidth angle is measured in plane containing the direction of main lobe
of the antenna.
[bw, angles] = beamwidth(antennaObject,plotFrequency,0,1:1:360)

% pattern for dipole
figure;
[gain,phi,theta] = pattern(antennaObject,
plotFrequency,azimuth_range,elevation_range)
saveToSTK('Real_SATDipole_144MHz.txt', phi, theta, gain)

% impedance for dipole
figure;
impedance(antennaObject, freqRange)
% s11 for dipole
figure;
s = sparameters(antennaObject, freqRange);
rfplot(s)
% current for dipole
figure;
current(antennaObject, plotFrequency)
% azimuth for dipole
figure;
patternAzimuth(antennaObject, plotFrequency)
% elevation for dipole
figure;
patternElevation(antennaObject, plotFrequency)
% Right-Hand Circularly Polarized(RHCP) radiation pattern
```

```
figure;
pattern(antennaObject,plotFrequency,'Polarization','RHCP')
figure;
pattern(antennaObject,plotFrequency,'Polarization','LHCP')

% calculate and plot the return loss of the helix antenna. Antenna return
loss is a measure of the
% effectiveness of power delivery from a transmission line to a load such as
antenna.
% The calculations are displayed in logscale.
impedenceValue = 50.0
figure;
returnLoss(antennaObject,freqRange,impedenceValue)

% calculate and plot the VSWR of the helix antenna.
% The antenna VSWR is another measure of impedance matching between
transmission line and antenna.
figure;
vswr(antennaObject,freqRange,impedenceValue)
```

Dipole 437MHz

```
%% Antenna Properties
% Design antenna at frequency 437000000Hz
antennaObject = design(dipole,437000000);
% Update load properties
antennaObject.Load.Impedance = 50;
azimuth_range = 0:1:360;
elevation_range = -90:1:90;

%% Antenna Analysis
% Define plot frequency
plotFrequency = 437000000;
% Define frequency range
freqRange = (391.5:4.35:478.5) * 1e6;
% show for dipole
figure;
show(antennaObject)
% calculate the beamwidth of the antenna. Antenna beamwidth is the angular
measure of the antenna pattern coverage.
% Beamwidth angle is measured in plane containing the direction of main lobe
of the antenna.
[bw, angles] = beamwidth(antennaObject,plotFrequency,0,1:1:360)

% pattern for dipole
figure;
[gain,phi,theta] = pattern(antennaObject,
plotFrequency,azimuth_range,elevation_range)
saveToSTK('Real_SATDipole_437MHz_Downlink.txt', phi, theta, gain)
% impedance for dipole

figure;
impedance(antennaObject, freqRange)
```

```
% s11 for dipole
figure;
s = sparameters(antennaObject, freqRange);
rfplot(s)
% current for dipole
figure;
current(antennaObject, plotFrequency)
% azimuth for dipole
figure;
patternAzimuth(antennaObject, plotFrequency)
% elevation for dipole
figure;
patternElevation(antennaObject, plotFrequency)
% Right-Hand Circularly Polarized(RHCP) radiation pattern
figure;
pattern(antennaObject,plotFrequency,'Polarization','RHCP')
figure;
pattern(antennaObject,plotFrequency,'Polarization','LHCP')

% calculate and plot the return loss of the helix antenna. Antenna return
loss is a measure of the
% effectiveness of power delivery from a transmission line to a load such as
antenna.
% The calculations are displayed in logscale.
impedenceValue = 50.0
figure;
returnLoss(antennaObject,freqRange,impedenceValue)

% calculate and plot the VSWR of the helix antenna.
% The antenna VSWR is another measure of impedance matching between
transmission line and antenna.
figure;
vswr(antennaObject,freqRange,impedenceValue)
```

MATLAB to STK Radiation Pattern Conversion

```
%Writing to STK Antenna File
% open a new file for writing
function y = saveToSTK(fileName, phi, theta, gain)
    output_file = fopen(fileName,'w');
    % write the external antenna pattern file header
    fprintf(output_file, 'stk.v.11.1.0\n');
    fprintf(output_file, 'PhiThetaPattern\n');
    fprintf(output_file, 'AngleUnit Degrees\n');
    fprintf(output_file, 'NumberOfPoints 65341\n');
    fprintf(output_file, 'PatternData\n');

    % loop through the azimuth, elevation, and gain values and write to file
    for i = 1:1:361
        k = 181;
        for j = 1:1:181
            if i == 361 && j == 181
```

```
                fprintf(output_file, '%f\t%f\t%f', phi(i),
(theta(k) + 90), gain(j,i));
            else
                fprintf(output_file, '%f\t%f\t%f\n', phi(i),
(theta(k) + 90), gain(j,i));
            end
        k = k - 1;
        end
    end
    % close the file
    fclose(output_file);
    disp('done saving')
end
```

APPENDIX K

[Justin]

The radio transceivers aboard the satellite are equipped with 50 Ω ports for the RF (antenna) receive/input and transmit/output, in the form of an *unbalanced* coaxial transmission line. But a Dipole antenna normally exhibits approximately 72 Ω as its natural impedance. Connecting the satellite transceivers directly to such an antenna would cause a signal reflection at the point of the impedance mismatch, sending some of the signal power backwards along the transmission line to its source, instead of efficiently delivering it to the destination.

The formula for calculating the reflection coefficient is:

$\Gamma = \sqrt{[(R-Z_0)_2 + j_2]} / \sqrt{[(R+Z_0)_2 + j_2]}$

…where:

- Γ is the reflection coefficient
- R is the Real part of the *load* impedance
- j is the Imaginary part of the *load* impedance
- Z_0 is the *source* impedance.

From the reflection coefficient, the Voltage Standing Wave Ratio can be calculated as follows.

VSWR = (1 + Γ) / (1 – Γ)

Ideally, a VSWR = 1 would represent a perfect coupling. To get *close* to this, an impedance matching circuit must be inserted in the antenna feedlines.

But because the Dipole antennas are *balanced*, and the transcievers send and receive *balanced* signals, a device known as a *balun* (portmanteau of “BALanced” = “UNbalanced”) must be inserted in the feedlines.

In a Dipole antenna, the radial opposite of the “main” radial acts as the *counterpoise* – carrying a *displacement current* equal and opposite to the main radial. (Hence the term “balanced”.) Everything else should, ideally, be electromagnetically neutral to the antenna.

In a (single-core) coaxial transmission line, only the displacement current along the core is considered the active signal; technically the *shield* does carry the current, but the signal is always measured or processed *relative* to this, so it is effectively neutral. And with most radio transceivers, the RF shield is common to the chassis and Earth Ground (or “Spacecraft Ground”, in this case).

If a Dipole was to be connected directly to the coaxial feedlines, the entire satellite chassis (“Spacecraft Ground”) would effectively become the *counterpoise*, and the antenna radial connected to the feedline core would act more like a Monopole. Clearly not the desired operation!

The balun contains an autotransformer to “electromagnetically isolate” the counterpoise of the balanced antenna from the unbalanced feedline shield (and spacecraft chassis, in this case), while also bringing the impedance to a closer match – 75 Ω. But the balun needs to pass the constraints and criteria imposed by the application (cubesat spacecraft, in this case), and a compatible balun should be selected with return loss figures as low as possible (ideally).

Appendix L

Gyro Demo [19] [20]

arduino_ide/calibrated_orientation/calibrated_orientation.ino

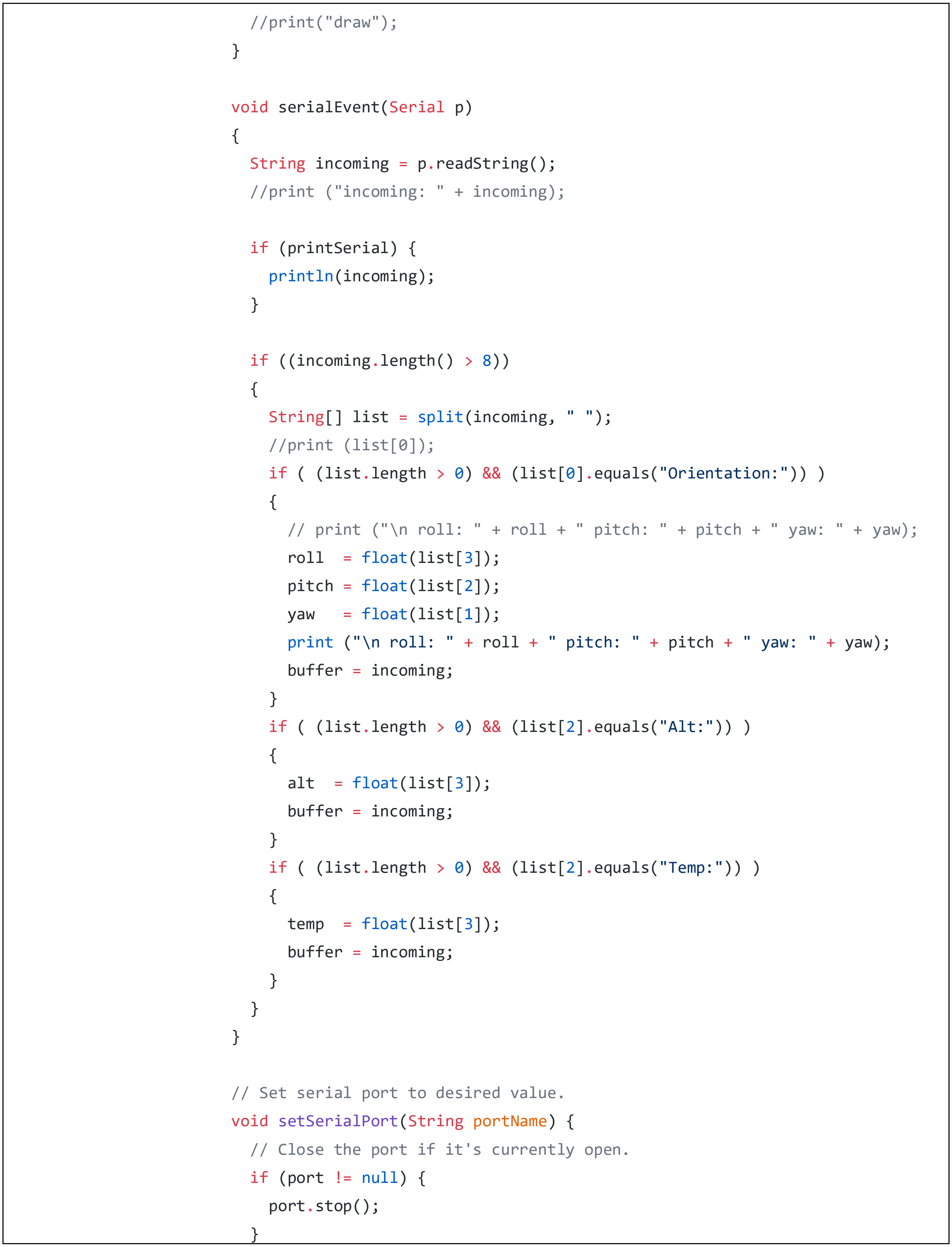
```
// Full orientation sensing using NXP/Madgwick/Mahony and a range of 9-DoF
// sensor sets.
// You *must* perform a magnetic calibration before this code will work.
//
// To view this data, use the Arduino Serial Monitor to watch the
// scrolling angles, or run the OrientationVisualiser example in Processing.
// Based on  https://github.com/PaulStoffregen/NXPMotionSense with adjustments
// to Adafruit Unified Sensor interface

#include <Adafruit_Sensor_Calibration.h>
#include <Adafruit_AHRS.h>

Adafruit_Sensor *accelerometer, *gyroscope, *magnetometer;

// uncomment one combo 9-DoF!
//#include "LSM6DS_LIS3MDL.h"  // can adjust to LSM6DS33, LSM6DS3U, LSM6DSOX...
//#include "LSM9DS.h"           // LSM9DS1 or LSM9DS0
#include "NXP_FXOS_FXAS.h"  // NXP 9-DoF breakout

// pick your filter! slower == better quality output
//Adafruit_NXPSensorFusion filter; // slowest
//Adafruit_Madgwick filter;  // faster than NXP
Adafruit_Mahony filter;  // fastest/smalleset

#if defined(ADAFRUIT_SENSOR_CALIBRATION_USE_EEPROM)
  Adafruit_Sensor_Calibration_EEPROM cal;
#else
  Adafruit_Sensor_Calibration_SDFat cal;
#endif

#define FILTER_UPDATE_RATE_HZ 100
#define PRINT_EVERY_N_UPDATES 10
```

```
//#define AHRS_DEBUG_OUTPUT

uint32_t timestamp;

void setup() {
  Serial.begin(9600);
  while (!Serial) yield();

  if (!cal.begin()) {
    Serial.println("Failed to initialize calibration helper");
  } else if (! cal.loadCalibration()) {
    Serial.println("No calibration loaded/found");
  }

  if (!init_sensors()) {
    Serial.println("Failed to find sensors");
    while (1) delay(10);
  }

  accelerometer->printSensorDetails();
  gyroscope->printSensorDetails();
  magnetometer->printSensorDetails();

  setup_sensors();
  filter.begin(FILTER_UPDATE_RATE_HZ);
  timestamp = millis();

  Wire.setClock(400000); // 400KHz
}


void loop() {
  float roll, pitch, heading;
  float gx, gy, gz;
  static uint8_t counter = 0;

  if ((millis() - timestamp) < (1000 / FILTER_UPDATE_RATE_HZ)) {
    return;
  }
  timestamp = millis();
  // Read the motion sensors
  sensors_event_t accel, gyro, mag;
  accelerometer->getEvent(&accel);
  gyroscope->getEvent(&gyro);
  magnetometer->getEvent(&mag);
#if defined(AHRS_DEBUG_OUTPUT)
  Serial.print("I2C took "); Serial.print(millis()-timestamp);
Serial.println(" ms");
#endif

  cal.calibrate(mag);
  cal.calibrate(accel);
  cal.calibrate(gyro);
  // Gyroscope needs to be converted from Rad/s to Degree/s
  // the rest are not unit-important
  gx = gyro.gyro.x * SENSORS_RADS_TO_DPS;
  gy = gyro.gyro.y * SENSORS_RADS_TO_DPS;
  gz = gyro.gyro.z * SENSORS_RADS_TO_DPS;

  // Update the SensorFusion filter
  filter.update(gx, gy, gz,
                accel.acceleration.x, accel.acceleration.y,
accel.acceleration.z,
                mag.magnetic.x, mag.magnetic.y, mag.magnetic.z);
#if defined(AHRS_DEBUG_OUTPUT)
  Serial.print("Update took "); Serial.print(millis()-timestamp);
Serial.println(" ms");
#endif

  // only print the calculated output once in a while
  if (counter++ <= PRINT_EVERY_N_UPDATES) {
    return;
  }
  // reset the counter
  counter = 0;

#if defined(AHRS_DEBUG_OUTPUT)
  Serial.print("Raw: ");
  Serial.print(accel.acceleration.x, 4); Serial.print(", ");
  Serial.print(accel.acceleration.y, 4); Serial.print(", ");
  Serial.print(accel.acceleration.z, 4); Serial.print(", ");
  Serial.print(gx, 4); Serial.print(", ");
  Serial.print(gy, 4); Serial.print(", ");
  Serial.print(gz, 4); Serial.print(", ");
  Serial.print(mag.magnetic.x, 4); Serial.print(", ");
  Serial.print(mag.magnetic.y, 4); Serial.print(", ");
  Serial.print(mag.magnetic.z, 4); Serial.println("");
#endif
```

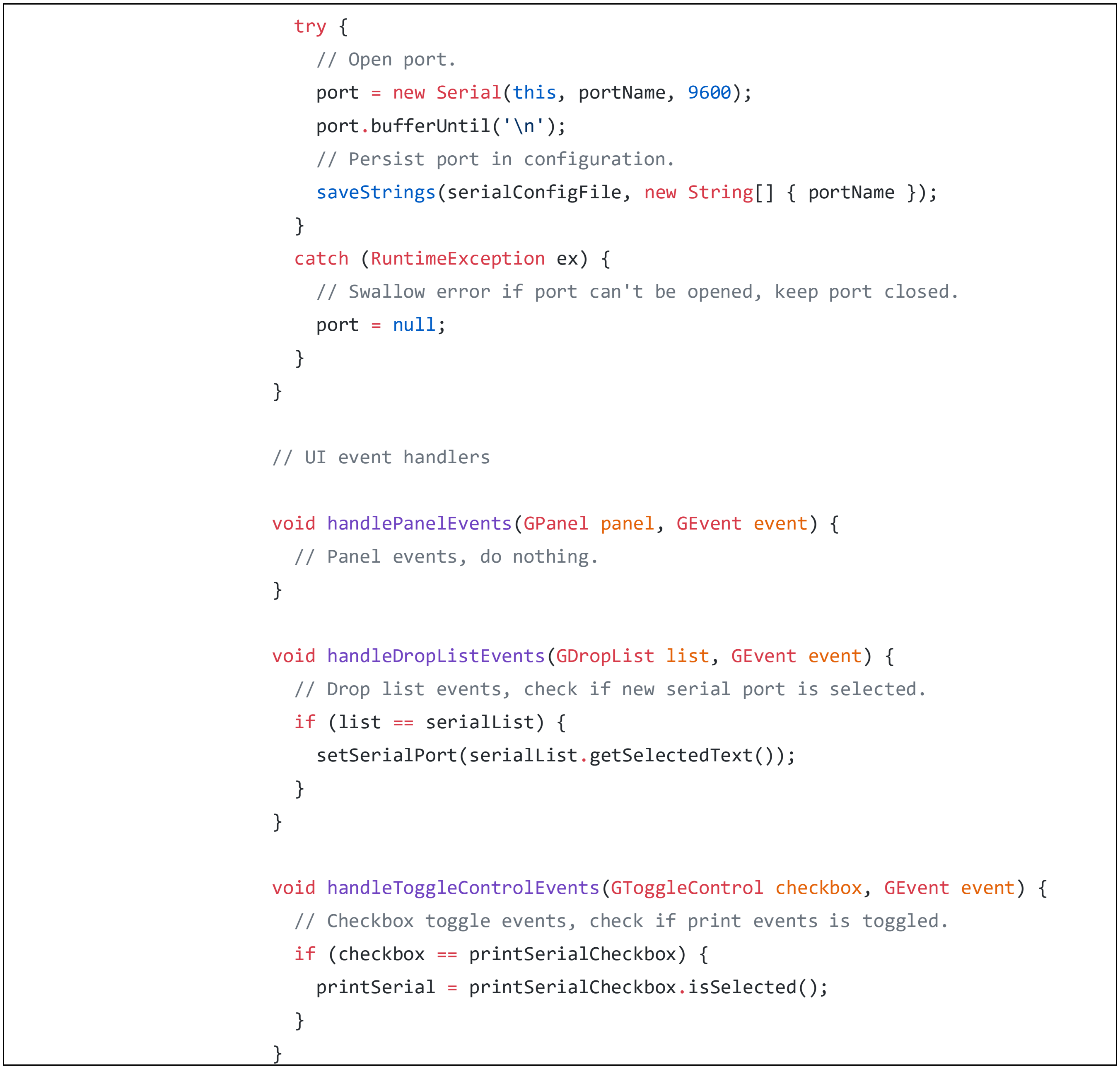
```
  // print the heading, pitch and roll
  roll = filter.getRoll();
  pitch = filter.getPitch();
  heading = filter.getYaw();
  Serial.print("Orientation: ");
  Serial.print(heading);
  Serial.print(" ");
  Serial.print(pitch);
  Serial.print(" ");
  Serial.println(roll);

  float qw, qx, qy, qz;
  filter.getQuaternion(&qw, &qx, &qy, &qz);
  Serial.print("Quaternion: ");
  Serial.print(qw, 4);
  Serial.print(", ");
  Serial.print(qx, 4);
  Serial.print(", ");
  Serial.print(qy, 4);
  Serial.print(", ");
  Serial.println(qz, 4);

#if defined(AHRS_DEBUG_OUTPUT)
  Serial.print("Took "); Serial.print(millis()-timestamp); Serial.println("
ms");
#endif
}
```

arduino_ide/Adafruit_AHRS/processing/bunnyrotate_ahrs_fusion_usb/bunny_ahrs_fusion_usb.pde

```
import
processing.serial.*;
import java.awt.datatransfer.*;
import java.awt.Toolkit;
import processing.opengl.*;
import saito.objloader.*;
import g4p_controls.*;

float roll  = 0.0F;
float pitch = 0.0F;
float yaw   = 0.0F;
float temp  = 0.0F;
float alt   = 0.0F;

OBJModel model;

// Serial port state.
Serial       port;
String       buffer = "";
final String serialConfigFile = "serialconfig.txt";
boolean      printSerial = false;

// UI controls.
GPanel    configPanel;
GDropList serialList;
GLabel    serialLabel;
GCheckbox printSerialCheckbox;

void setup()
{
  size(400, 500, OPENGL);
  frameRate(30);
  model = new OBJModel(this);
  model.load("cubesat4.obj");
  model.scale(20);

  // Serial port setup.
  // Grab list of serial ports and choose one that was persisted earlier or
default to the first port.
  int selectedPort = 0;
  String[] availablePorts = Serial.list();
  if (availablePorts == null) {
    println("ERROR: No serial ports available!");
    exit();
  }
  String[] serialConfig = loadStrings(serialConfigFile);
  if (serialConfig != null && serialConfig.length > 0) {
    String savedPort = serialConfig[0];
    // Check if saved port is in available ports.
    for (int i = 0; i < availablePorts.length; ++i) {
      if (availablePorts[i].equals(savedPort)) {
        selectedPort = i;
      }
    }
  }
  // Build serial config UI.
  configPanel = new GPanel(this, 10, 10, width-20, 90, "Configuration (click to hide/show)");
  serialLabel = new GLabel(this,  0, 20, 80, 25, "Serial port:");
  configPanel.addControl(serialLabel);
  serialList = new GDropList(this, 90, 20, 200, 200, 6);
  serialList.setItems(availablePorts, selectedPort);
  configPanel.addControl(serialList);
  printSerialCheckbox = new GCheckbox(this, 5, 50, 200, 20, "Print serial data");
  printSerialCheckbox.setSelected(printSerial);
  configPanel.addControl(printSerialCheckbox);
  // Set serial port.
  setSerialPort(serialList.getSelectedText());
}

void draw()
{
  background(0,0, 0);

  // Set a new co-ordinate space
  pushMatrix();

  // Simple 3 point lighting for dramatic effect.
  // Slightly red light in upper right, slightly blue light in upper left, and white light from behind.
  pointLight(255, 200, 200,  400, 400,  500);
  pointLight(200, 200, 255, -400, 400,  500);
  pointLight(255, 255, 255,    0,   0, -500);

  // Displace objects from 0,0
  translate(200, 350, 0);

  // Rotate shapes around the X/Y/Z axis (values in radians, 0..Pi*2)
  rotateX(radians(roll));
  rotateZ(radians((-1)*pitch));
  rotateY(radians(yaw));

  pushMatrix();
  noStroke();
  model.draw();
  popMatrix();
  popMatrix();
```



```
  //print("draw");
}

void serialEvent(Serial p)
{
  String incoming = p.readString();
  //print ("incoming: " + incoming);

  if (printSerial) {
    println(incoming);
  }

  if ((incoming.length() > 8))
  {
    String[] list = split(incoming, " ");
    //print (list[0]);
    if ( (list.length > 0) && (list[0].equals("Orientation:")) )
    {
      // print ("\n roll: " + roll + " pitch: " + pitch + " yaw: " + yaw);
      roll  = float(list[3]);
      pitch = float(list[2]);
      yaw   = float(list[1]);
      print ("\n roll: " + roll + " pitch: " + pitch + " yaw: " + yaw);
      buffer = incoming;
    }
    if ( (list.length > 0) && (list[2].equals("Alt:")) )
    {
      alt  = float(list[3]);
      buffer = incoming;
    }
    if ( (list.length > 0) && (list[2].equals("Temp:")) )
    {
      temp  = float(list[3]);
      buffer = incoming;
    }
  }
}

// Set serial port to desired value.
void setSerialPort(String portName) {
  // Close the port if it's currently open.
  if (port != null) {
    port.stop();
  }
```



```
  try {
    // Open port.
    port = new Serial(this, portName, 9600);
    port.bufferUntil('\n');
    // Persist port in configuration.
    saveStrings(serialConfigFile, new String[] { portName });
  }
  catch (RuntimeException ex) {
    // Swallow error if port can't be opened, keep port closed.
    port = null;
  }
}

// UI event handlers

void handlePanelEvents(GPanel panel, GEvent event) {
  // Panel events, do nothing.
}

void handleDropListEvents(GDropList list, GEvent event) {
  // Drop list events, check if new serial port is selected.
  if (list == serialList) {
    setSerialPort(serialList.getSelectedText());
  }
}

void handleToggleControlEvents(GToggleControl checkbox, GEvent event) {
  // Checkbox toggle events, check if print events is toggled.
  if (checkbox == printSerialCheckbox) {
    printSerial = printSerialCheckbox.isSelected();
  }
}
```

APPENDIX M - Goals

Radio Frequency Communication Link Budget

[Pierre] February

A link budget was completed to prepare for STK orbital analysis. This includes the satellite's transceiver power and losses as well as ground station transceiver power and losses. Antenna parameters are also included. Final uplink and downlink budgets will be used for initial STK orbital analysis. This will determine if the satellite has good enough communication with the groundstation based on antenna configuration as well as properties of the transceivers.

ADCS Controller

[Adam] February

- Stm32 f401re board has been decided to be used for the ADCS subsystem
- Board has been proven to be space operative so long as operation is under a year
- There is no cpu lock stepping as processor is single core ARM Cortex-M4
- Workstation has been configured to allow the board to be used with the Arduino IDE which rapidly speeds up development since there are many adafruit libraries that can be used for sensors and motor control
- Adafruit 9Dof sensor includes magnetometer, gyroscope, and accelerometer has now been configured to interface with the f401re through i2c
- roll, pitch, and yaw can be read from the 9Dof and mapped to an Obj file for a real time display of the sensor's orientation and this will be used when an Obj file of the satellite is made
- Watchdog time has been tested on the f401re

TO DO:

- Soft error handling on the f401re (ie. bit flipping)
- configure daughter board that will sit on top of the f401re to control motors
- access low hardware configurations of f401re while using arduino ide (i.e. control bits to make gpio port SDA)

Payload-OBC Communications

[Jon] February

- initial software application has been written and installed on the payload module (raspberry pi)
- the OBC (beaglebone board) can now ping the payload module for successful connection, request an immediate capture, and then transfer the image from the payload module to the OBC storage
- however, the image transfer is quite slow over serial UART, so currently investigating other options for the image data transfer between the payload and OBC
- SPI was going to be used for the image data transfer, but unfortunately it was discovered that linux does not have driver support to act as an SPI or I2C slave, only SPI and I2C masters. The payload module and OBC are both running versions of linux and SPI connection can only happen between an SPI master and SPI slave. As a result I2C and SPI communication will not work.
- this interface may also be used for loading new software versions from the OBC to payload module so this also will be taken into consideration when selecting another interface.

TODO:

- research, select and test new, faster interface for image data transfer between the payload module and OBC.
- starting setup RF connection with OBC and simulated ground station (arduino) to be able to be achieve full image capture on payload module and full image transfer to OBC and then to ground station over RF.

Payload-CDH Update

[Jon] April

- due to the recent circumstances of COVID-19, the project scope has changed a bit related to payload and the CDH development.
- Payload will mostly continue as normal, with simulations being done using STK. Payload software will be developed and testing on a RPi Zero with a pi camera as the image sensor.
- CDH software development will continue with KubOS Linux on the beaglebone black, once the board is ordered and received (the team has lost access to the lab and all of the components there so the minimum amount of components will be ordered for at home development and testing). The flight software will be simulated on the BBB as much as possible, without attaching any the ADCS, EPS, or RF hardware as these components are unavailable. These will be simply simulated in the software, returning fake or mock telemetry. The goal is to be able to interface the BBB (CDH) and the RPi Zero (Payload) for commanding image captures and performing image transfers.
- Note: from the previous update, we were having issues with the interface between the payload module and the OBC. Since both are running Linux, no master-slave protocol could be used so we reverted to using UART. UART is alright for sending small amounts of data, but image transfers were taking way too long. The KubOS Linux team has now added support for the USB interface on the BBB. Once the BBB has been received, we are hoping that the BBB will act as the USB host and the RPi zero as the USB device to perform faster image transfers from the payload to the OBC.

TODO:

- CDH software development and testing
- Payload software development and testing
- Image sensor integration
- Payload (RPi Zero) and CDH (BBB) integration

RF Antenna Deployment System and Ground Station

[Pierre] April

Revised Plan:

Due to the capstone project being limited to working from home, the antenna deployment system will not be constructed. However, we were previously able to get licensing from Dr. Rashidzadeh for EM Pro and will simulate our RF PCB design for the antenna release mechanism, signal conversion and power. In addition, more attention will directed towards the ground station design. This will include basic configuration and possibly simulating these parameters in EM Pro.

CDH Update – ADCS subsystem

[Adam] April

Due to the impact of COVID-19, the ADCS system will no longer be done with space grade systems.
Instead a flat sat will be made using off the shelf, hobbyist boards.

TO DO:

Instead of the STM32 F401RE board, the ADCS system will be an Arduino UNO at its core and a Raspberry Pi 2 will be used as the OBC.
The main thing that will be implemented with the Arduino board is multitasking, low power modes, I2C communication to the Pi and to hobby sensors, and any other embedded system standards (ie. error handling, memory dump on crash).

Payload/CDH Update
[Jon] July

Command and Data Handling Updates:

- CDH team has been working to finalize initial prototype of flight software to run on the main Onboard Computer as well as working on the the software and integration with the ADCS submodule.
- Satellite has been setup at home as a "flat-sat" where all space-flight components have been replaced with COTS parts for testing and proof of design, layed out flat. This is to simulate as best as possible the actual satellite for software testing.
- The flight software now includes nominal operation procedures, error handling, hardware communications with all other subsystems (EPS, ADCS, Payload, and RF).
- All testing has been done with COTS components to prove software functionality, although not the exact same with actual space-flight components. Further integration and testing would need to be done once actual components for flight model have been received.
- Simple, software radio functionality has been implemented and tested to where no the user can send the satellite various commands over RF (turn things on/off, operate subsystems, downlink images/telemetry, etc..). The more control and more commands we have to the satellite from the ground station, the safer and more reliable the mission will be.

TODO:

- finish as much of software development/testing of other additional features/functionality of the satellite.

- look into ground station software for monitoring passes and automated data downlink.

Payload Updates:

- Payload team has been working through simulations, software design of payload module that is responsible for taking images, doing any necessary filtering, processing, encryption, and preparing them for downlink.
- Work has been done to verify both satellite mission objectives: single, pointed image acquisition and image coverage acquisition.
- The team has also been developing a simple web app that will receive images from the ground station when downlinked from the satellite and automatically overlay them over a global map for viewing by users. This will allow for viewing of satellite coverage of certain areas.

TODO:

- Finalize simulations, calculations, and other software to prepare for final report and presentation.

Radio Frequency Update
[Pierre] July

ADS and EMPro Simulations:

- The radio frequency division has been working on simulating the balun circuit that will be used to convert the balanced/unbalanced signal for receiving data from the ground station and for down linking data to the ground station.
- ADS (from keysight technologies) has been used to modify an existing example of a balun circuit. The final designs have been exported to EMPro for better simulation results including graphs and 3D imagery.
- ADS and EMPro provide VSWR, E-field radiation, reflection coefficient and power characteristics that help to design the traces on the PCB for optimal signal preservation. Comparisons have been made between configurations.
- The simulations also consider different dielectric materials for the substrate of the traces. Previously space tested PCB designs have been used as primary examples for their choices of materials, thickness, etc. Provided with the opportunity, the final design would have been built and tested with the antennas.
- All testing configurations consider the satellite's size restrictions as this design constraint is the reason for simulating different orientations and curvatures for the PCB traces.

TO DO:

- Test as many types of trace curvatures to minimize signal reflection.
- Provide a method of observing performance with a disturbance signal to determine noise pollution
- Research possible ground station setups

CDH - ADCS Controller Update

[Adam] July

- Hardware abstraction for STM32 F446RE was created and board was brought up using STM32CubeIDE

TO DO:

- Configure and test ADCS controller as I2C slave and I2C master