Comprehensive Flight Software Design and Validation for the Harvard CubeSat I Mission

A thesis presented
by

Madison Davis

Presented to the

John A. Paulson School of Engineering and Applied Sciences

in partial fulfillment of the requirements for the degree with honors of

Bachelor of Arts

Harvard College
Cambridge, Massachusetts
March 27, 2026

Table of Contents

Abstract

This thesis analyzes the software architecture, development decisions, and testing procedures of the Harvard Undergraduate CubeSat (HUCSat), a six-year student-led nanosatellite project dedicated to evaluating the economic and practical viability of nitinol shape memory alloys to rotate solar panels. Section 1 provides a contextual background concerning the HUCSat's overall hardware design and layout of circuit boards. Section 2 dives specifically into the Flight Controller Board (FCB) chip, defending how our team progressed through rapid prototypes to our ultimate settlement on the RP2350 microprocessor. Section 3 presents all the major software files that the FCB employs to schedule and operate all necessary tasks. This includes a main loop file, data processing files, finite-state machine (FSM) files, drivers, managerial classes, and configuration files. Some of this work is novel, developed by myself from scratch or re-purposed from open-source development code. Finally, Sections 4-5 underscore the testing and experiment procedure to help ensure robustness and clarity of the mission, respectively. In Sections 1-3, I also detail the design considerations I had to consider, justifying the transition from one variation to the next.

The primary contributions of this thesis are threefold. First, I present the implementation of new low-level software components, including hardware drivers and device managers abstractions tailored to the RP2350 PROVES Kit open-source flight controller. Second, I detail the development of testing methodologies, operational procedures, and experimental designs used to validate system behavior. Overall, such considerations will emphasize the satellite's resilience to concerns such as radiation-induced faults, Low-Earth Orbit (LOE) temperature fluctuations, and efficiency in memory usage under constrained microcontrollers. Third, under the constraints of the FCB, I introduce a dynamic, fault-tolerant, and non-blocking task-scheduling framework for CubeSat operations. Notably, the framework integrates a system-wide finite state machine and data processing class with asynchronous task execution and command-data handling extensions to enable adaptive-decision making based on component health and telemetry data. In addition, the inclusion of a radiation-tolerant Watchdog timer, reboot capability, and backup power systems with health monitoring ensure dynamic fault-tolerance. In sum, these contributions aim to provide a concrete reference to future generations constructing CubeSat flight software.

Introduction

Problem Statement

This thesis focuses on the major development and deployment decisions and justifications for the Harvard Undergraduate CubeSat (HUCSat) software system. CubeSats are a class of nanosatellites based on a standard modularized design. NASA offers a CubeSat program to allow US educational facilities and non-profit organizations to launch scientific research payloads. The Harvard HUCSat is one of these payloads, which was selected in the 2022-2023 round of grants and was given a launch date of 2026. HUCSat encapsulates mechanical, electrical, software, ground-station communication, and scientific elements. Its primary objective is to test the utility and viability of lightweight Shape Memory Alloys (SMAs) compared to larger, heavier motors to move solar panels. This is an important means to reduce space instrumentation complexity and costs [4].

Contribution

The HUCSat development has spanned nearly six years of work over various leadership changes. The leadership team comprises a Chief Engineer, Bus Electrical Lead, Bus Mechanical Lead, Bus Computing Lead, Ground Station Lead, and Payload Lead. My personal contribution spans four of these years, culminating as the Bus Computing lead.

My contribution primarily spans the following three key areas:

1. Instantiating new firmware driver and manager classes for open-source software.
2. Testing development and procedures, as well as experimental design.
3. The software development of the main schedule sequencer, configuration protocols, command-data-handler system add-ons, and finite state machine.

On the software end, the novelty of this work stems around a dynamic, fault-tolerant, and non-blocking task-scheduling framework designed to withstand space's environmental conditions. This solution was developed around the RP2350 PROVES Kit flight controller and makes use of the following key innovations:

1. **Dynamic:** finite state machine logic looks at the system-state as a whole to dynamically decide what state to enter. Specifically, the FSM decides which operation to enter based on the health of its components and telemetry data.
2. **Fault tolerance:** failure detection and recovery within processes in case of radiation ionization such as single-event upsets (SEUs).
3. **Non-blocking:** asynchronous tasks to allow for parallelization and improved performance

4. **Memory efficiency:** to save space, the FCB does not require a large-scale OS and instead utilizes flashed CircuitPython 1.6MB firmware (built on top of MicroPython) to instantiate a compatible runtime environment. As a compatible comparison, resource-constrained embedded systems may use Zephyr OS, which can fit comfortably within 1 MB. This contrasts to more-scaled systems, where even the most stripped software like the Tiny Core Linux OS would operate over 11 MB [16].
5. **Temperature sensitivity:** both hardware and software integrations were utilized to withstand Low-Earth Orbit (LEO) temperature fluctuations. Low power modes and scheduling or heavier operations during sunlit periods are a few examples.

In addition, this work is novel via the addition to open-source software. Without the creation of new firmware drivers, it would not be possible for future missions to be launched using more modernized detumbler and light-sensor technology used for the PROVES Kit. The work presented in this thesis also provides an important milestone for Harvard's undergraduate spatial endeavors, as it is to the best of our knowledge that this is Harvard College's first ever satellite launch supported by the CubeSat program. Therefore, this paper provides useful documentation for future teams to look back upon, defending prototyping decisions in the name of robustness, reliability, and efficiency.

Acknowledgments

The cumulation of this work would not have been possible without the extensive help of peers and outside third parties. This includes, but is not limited to:

1. Current Leadership Team: Christopher Prainito, Katelyn Miller, Milligan Grinstead, Adolfo Balderas, Shriman Jha, Cameryn Neches
2. PROVES Kit at CalPoly Institute: Michael Pham, Nate Gay, Ines Khouider
3. Previous Leadership Teams: David Andrade, Samantha Marks, Ben Fitchenkort
4. Professors: Prof. Woodward Yang, Prof. James Mickens, Prof. Gage Hills, Prof. Paul Horowitz

1. Satellite Hardware Design

The hardware of the satellite can be broken down into several subsystems. The subsystems are as follows: Electrical Power Subsystem (EPS), Attitude Determination and Control subsystem (ADCS), Comms subsystem, Payload subsystem, and Flight Controller Board subsystem (FCB). Figure 1 illustrates how the subsystems are stacked in the CubeSAT in a rack-like structure, which is based on inspiration from Portland's satellite development team [17].

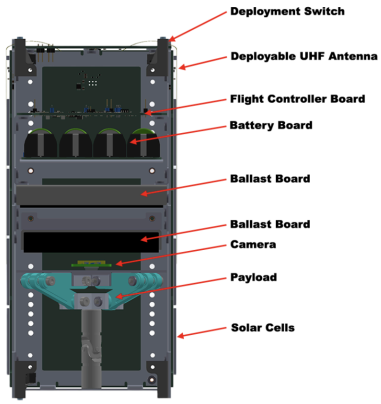

Figure 1A
side-view of CubeSAT Subsystems [10]

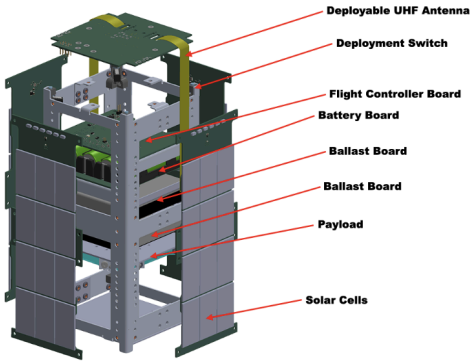

Figure 1B
angular view of CubeSAT Subsystems [10]

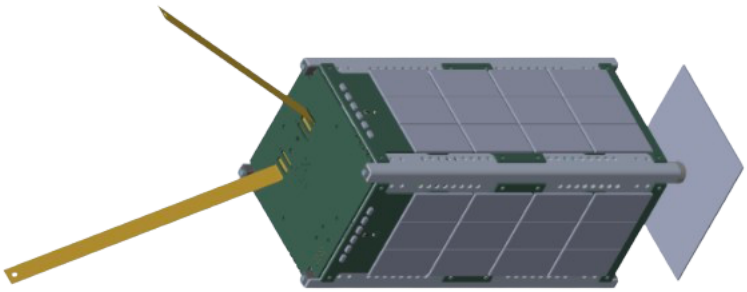

Figure 1C
deployment-view of CubeSAT [10]

Figure 2 provides real-life photos of the actual assembled CubeSAT.

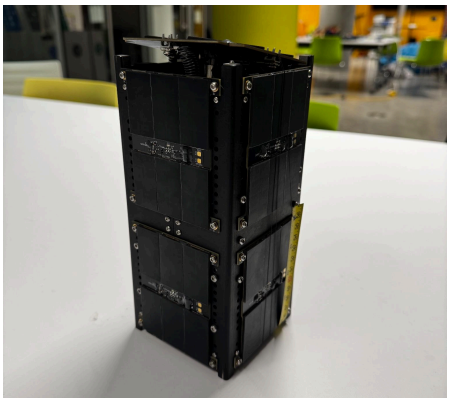

Figure 2A
side-view of CubeSAT Subsystems [10]

Figure 2B
angular view of CubeSAT Subsystems [10]

For the remainder of this section, I will provide a comprehensive overview of all the subsystems. We will proceed in the aforementioned order of subsystems: EPS, ADCS, Comms, Payload, and FCB. The first four subsystems will introduce several hardware components and physics concepts that the reader may find useful for context. The FCB is saved for last because it describes how all the other subsystems are connected to it, therefore relying on the aforementioned context.

EPS: Electrical Power Subsystem

The EPS is what keeps the satellite powered. To collect power, HUCSat contains 9 custom solar panel printed circuit boards (PCBs), 2 on each of the +/-X and +/-Y faces, and 1 on the -Z face. Each solar panel contains six AnySolar/IXYS KXOB101K08TF high-efficiency silicon solar cells. A solar cell is a device that converts sunlight into direct electricity.

The primary storage mechanism on the satellite consists of an ISS-specification approved battery board, which contains four LG INR18650 MJ1 lithium-ion cells in series. Datasheets state that the cells can hold up to 8.0V when fully charged [11]. As an added back-up, the FCB also contains its own coin cell battery. Figure 4 details the real-life visual of the battery board (left) as well as its circuitry schematic (right).

Figure 4A
battery board v5D

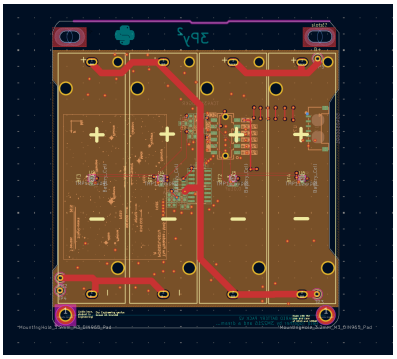

Figure 4B
battery board KiCad Schematic

ADCS: Attitude Determination and Control Subsystem

The ADCS keeps the satellite on a stable trajectory when moving through space. My team went through two versions of the ADCS, as articulated below.

Version 1

Version 1 consisted of both active and passive methods for detumbling the satellite. The active method deals with a custom-made magnetorquer board, which consists of 2 rod magnetorquers and 1 air-coil magnetorquer for the X, Y, and Z directions [10]. The passive method was kept onboard in case of active ADCS malfunction: this included MEMS magnetorquers, a LIS2MDL magnetometer, and a 6-axis IMU (LSM6DOSX) [10].

For background, a magnetorquer is a device that consists of electromagnetic coils that, when energized with a current, creates a magnetic field. The strength and direction of this field is defined as the magnetic moment (M). Mathematically, M = N * I * A, where N is the number of coil turns, I is the strength of the current being passed through the coil (in amps), and A is the cross-sectional coil area (in m^2). Therefore, M has units Am^2. For added intuition, the "N * A" component is referred to as the magnetic gain, and so the physical design of the magnetorquer (the number of coil turns and its cross sectional area) directly determines how much 'gain' or 'strength' the resulting field has.

The reason one needs to create a magnetic field is because Earth has its own magnetic field (B); if current is run through the magnetorquers to create a secondary magnet field, this field interacts with Earth's magnetic field to generate a torque to spin the satellite. Mathematically, the torque is represented as T = M * B. By adjusting the current flowing through the coils, the satellite can control the magnitude and direction of its generated torque. This enables slow but effective attitude adjustments, especially useful for detumbling after deployment. To find out what Earth's magnetic field is at a given location, one can use a magnetometer: a magnetometer is an instrument that can detect the strength and direction of Earth's magnetic field.

Version 2

Version 2 transitioned from using an active set of magnetorquers on its own special magnetorquer board to repurposed DRV2605 haptic drivers for detumbling.[1] The main reason for this was due to the difficulty in manufacturing a reliable active magnetorquer in-house.

Each of the +/- XY solar board panels has one DRV2605 haptic driver mounted on it. Figure 5 details the real-life visual of a singular solar panel board (left) as well as its circuitry schematic (right). For a comprehensive description, each solar panel has the following components:

- 6x AnySolar KXOB101K08TF Solar Cells in a 2s3p configuration
- 1x Microchip MCP9808T-E/MC I2C Temperature Sensor
- 1x TI DRV2605LDGSR I2C Haptic Motor Drivers
- 1x Vishay VEML6301x00 Ambient Light Sensors
- 1x Fun Blue LED
- 1x 5-pin Molex 1.5mm Pitch Picolock for VSolar, 3.3V potential, and I2C connections

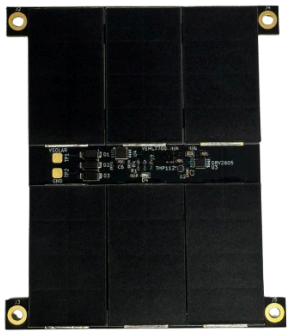

Figure 5A
XY solar board, outward-face

Figure 5B
XY solar board, inward-face

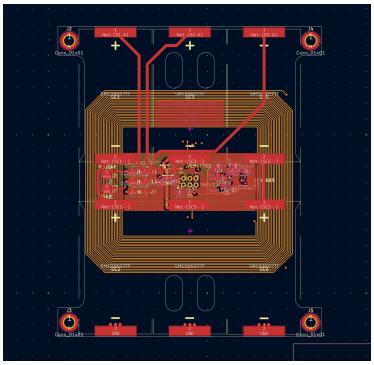

Figure 5C
XY solar board KiCad schematic

[1] Haptic drivers are unlike magnetorquers and don't interact with Earth's magnetic field. Haptic drivers are devices that are composed of three components: a wound-up coil; an integrated circuit (IC) that controls the amount of current going to the coil; and a magnet. When one generates current to go through the coil, the coil creates a magnetic field. This field interacts with the magnet's inherent field to make the magnet react. In ERM haptic drivers, the magnet is attached to an off-center mass that rotates in response to the magnetic field, creating a rotational effect. In LRA haptic drivers, the magnet is usually attached to a spring to move linearly in response to the magnetic field, creating a linear vibration effect. DRV2605 haptic drivers offer both LRA and ERM actuators.

Haptic drivers are very small, and a single driver would not produce any vibration strong enough to detumble the satellite. With this knowledge, it may initially seem confusing as to why haptic drivers are being used in this instance. The key feature is that the satellite is *repurposing* the five haptic drivers (one on each solar panel board) to actuate magnetorquer coils embedded within the solar boards themselves. The +/- X and +/- Y faces have rectangular coils, and the -Z axis has a circular coil. In other words, our team reuses the integrated circuit found within the DRV2605 haptic drivers to control the amount of current going to embedded magnetorquer coils, and this current will be based on the readings from Earth's magnetic field.

The FCB can communicate to each solar panel's DRV2605 haptic driver via a TCA945 sensor multiplexer.[2] For a given face (+/- X, +/- Y, -Z) we wish to communicate to from the FCB, we must make sure it is first able to receive communication by enabling its respective load switch. Because we have many faces, we use a GPIO expander to define pin spots for these load switches.

To determine how much current to send to the haptic drivers, our satellite needs data on its current trajectory and movement. We use an IMU LSM6DSOX to give us data related to angular acceleration and angular velocity (measured in rad/s). We also use a LIS2MDL magnetometer to get magnetic field data (measured in Teslas).

[2] A multiplexer, commonly referred to as a "mux," is an electronic switch that allows a single controller to communicate with multiple devices over the same communication line. In our case, the single controller is the FCB chip. The TCA945 mux allows the FCB to select which DRV2605 haptic driver or VEML6031 light sensor it wants to talk to, one at a time.

Comms Subsystem

The Comms Subsystem is concerned with all inter and intra satellite communication. For communication down to the ground-station, there exist two antennas on board. The first is a UHF RFM9x 437.4 MHz antenna constructed using two tape measurers, each 170cm long. These antennas are deployed after detumbling finishes. The antennas are attached to the antenna board, whose real-life visual (left) and KiCad schematic (right) are depicted in Figure 6. The backup second antenna set is an internal 2.4 GHz SX1280 s-band LoRA antenna that is integrated onto the FCB. The 437.4 MHz is our primary mode of communication to the terminal unit ground-station due to its superiority over long-range distances, compared to the 2.4 GHz antenna.[3]

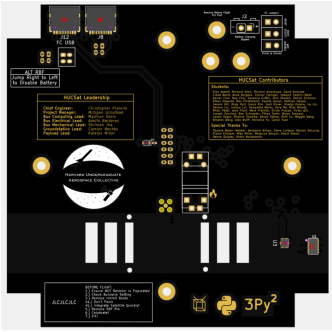

Figure 6A
antenna board

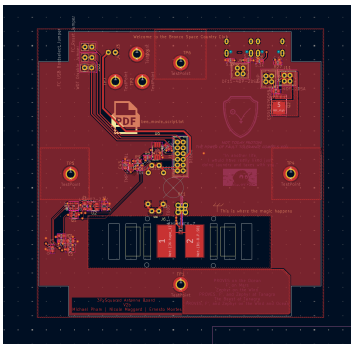

Figure 6B
antenna board KiCad Schematic

Our satellite does not require a direct aim to our ground station to transmit data. Instead, we leverage telemetry systems so that any ground-station that picks up our data will route the data back to our ground-station. This is achieved by having an assigned call-sign embedded in the configuration data. We decided on this approach to reduce the complexity of orienting the satellite.

The ground station consists of 1 modified FCB and a 10-foot antenna structure atop the John Paul SEAS Science and Engineering Complex building. This is where the UHF and s-band radio communication get picked up at. Note that RFM9x and SX1280 have different configuration parameters, including wavelength modulation and transmission power (optimizing for low data rates).

[3] Our team had initially intended to use the 2.4 GHz to allow us to send image data. However, due to NOAA regulations, the chance of being able to acquire a camera for image capturing became very slim, and the team decided to keep the functionality of the s-band antenna for redundancy.

In addition to broad-range communication, there also exists communication mechanisms within the satellite itself. Notably, HUCSat employs 2 SPI buses and 2 I2C buses. SPI allows for full-duplex (communication between two endpoints simultaneously), whereas I2C is simpler and is only half-duplex (analogous to a walkie-talkie, only one direction at a time). In essence, SPI is best for large data transfers across a few devices, I2C is better for small data transfers across many devices. For SPI, the first bus exists to get data between the FCB and UHF radio, and the second bus gets data between the FCB and s-band radio. For I2C, the first bus allows the FCB to talk to two components on two different addresses: the first is the TCA945 sensor multiplexer (subsequently the 5 light sensors and 5 haptic drivers), and the second is the battery power manager. The second bus allows the FCB to talk to three components on three different addresses: the first is the MCP23017 GPIO pin expander, the second is the LSM6DSOX IMU, and third is the LIS2MDL magnetometer. All of these components have been discussed earlier in this section except the MCP23017. The MCP23017 expander is added hardware that simply turns on or off the solar panel enable pins. An enable pin either permits or shuts-off power supply to a certain piece of hardware without shutting down the main power supply (in this case, the enable pins are for the solar board faces).

Payload Subsystem

The Payload Subsystem is the last subsystem and is concerned with orienting and operating the payload. A payload is a scientific experiment on a CubeSAT, and HUCSat's payload is to test the utilization of nitinol springs for solar panel deployment and orientation to maximize energy conservation. For context, nithinol is an alloy composed of nickel and titanium. At low room temperatures, nithinol exists in a martensite phase where it can be easily deformed, twisted, and shaped. From here, if nitinol is heated up at high temperatures for an extended period of time (during a period called *heat training*), it enters a new phase called the austenite phase where the atoms are more ordered and rigid. Here, the atoms remember the shape that the nithinol was in during its martensite phase. For example, if the material was bent like a clover at room temperature, the atoms will remember that structure during heating. After the nitinol cools down again, one can re-apply force or pressure, such as through an electrical current, for the material to snap back to that remembered shape. Following the example above, if one were to pass an electrical current through the material following a heat train and cool down, the material would re-bend to the shape of a clover. The memorized shape can be reset through another round of heat training.

The payload exists on the -Z axis, opposite the UHF antennas. It encompasses two boards connected by four orthogonal legs set up as a Sarraus linkage. The top board is where the -Z axis solar panel sits, and the bottom board is simply used to connect our payload to the rest of the satellite. Figure 7 provides a nice visual representation of the payload's compressed and deployed forms, as well as a real-life visual of the -Z axis board. In addition to the legs, on each corner of the bottom board (not pictured in the figure), there exists a nitinol spring that connects the bottom board to the top board.

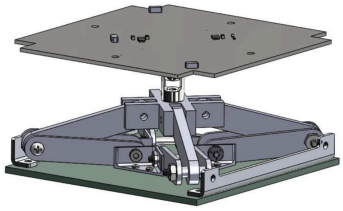

Figure 7A
payload board, compressed [10]

Figure 7B
payload board, deployed [10]

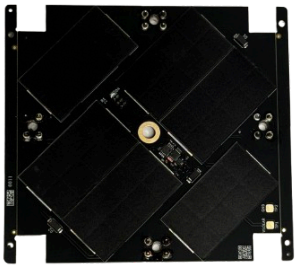

Figure 7C
-Z axis solar panel

We use four VEML6031 light sensors to help us with orientation. The FCB can communicate to each solar board's (+/- X and +/- Y) VEML6031 light sensor via the TCA945 sensor multiplexer, similar to the haptic drivers described in the ADCS section. We first observe the light readings from each sensor to determine which side of the satellite is receiving the most light. For the side that has the most light, we locate the nitinol spring most closely associated with that side. Due to temperature concerns out in space and not wishing to draw too much power, we only select one spring at a time. Therefore, if two adjacent sides have roughly equal amounts of light, we will look at further decimal points to delineate the choice, or choose one side randomly if a tie. After this decision is made, we send current down the chosen nitinol spring via a FCB command. The nitinol springs were heated to a shrunken position: therefore, when heat re-passes through nitinol in the form of current, the coil will shrink, rotating the head of the solar panel in that direction.

The payload, along with the UHF antenna, is deployed via a burnwire after detumbling. The burnwire is literal fishing wire and is attached to the ends of the UHF antennas. The wire wraps around the payload and a resistor on the antenna board that, when heated up, snaps the wire and releases the payload and antenna that were initially held in place. The heater and enable pins for this burnwire deployment are labeled as FIRE_DEPLOY_1A and FIRE_DEPLOY_1B, respectively.

FCB: Flight Controller Board

The FCB refers to the Flight Controller Board, the main processing board that handles the scheduling and operating of all other subsystems. As stated earlier, my discussion here is mostly withheld to be articulated in greater detail for the later sections. Figure 3 details the real-life visual of the FCB (left) as well as its circuitry schematic (right).

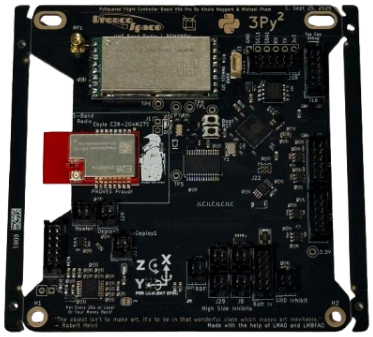

Figure 3A
flight controller board v5D

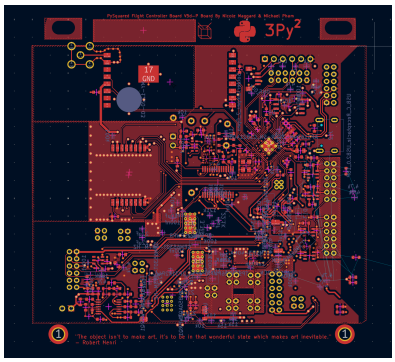

Figure 3B
flight controller board v5D KiCad Schematic

FCB Interactions

The FCB has many components that allow it to interact with the other four subsystems. Figure 10 showcases a labeled diagram of how the FCB connects to the other four aforementioned subsubsystems: EPS, ADCS, Comms, and Payload.

1. The labels display the location of either hardware components/pins (non-italicized labels) or hardware pins' associated software drivers and manager classes (italicized labels) on the board.[4] All the labels for hardware have been described thus far in this Section. Their associated software drivers and manager classes will be described in Section 3.
2. The labels are color-coded based on which of the four subsystems use them: red for EPS, orange for ADCS, yellow for comms, and purple for payload. Several subsystems can use the same underlying pins/drivers/manager classes, as illustrated by multicolored labels.

[4] Drivers can be thought of as "bridges" between the software and hardware, quite literally connecting pin definitions from publicly-released online datasheets to variables defined by the manufacturers and understood by the hardware.

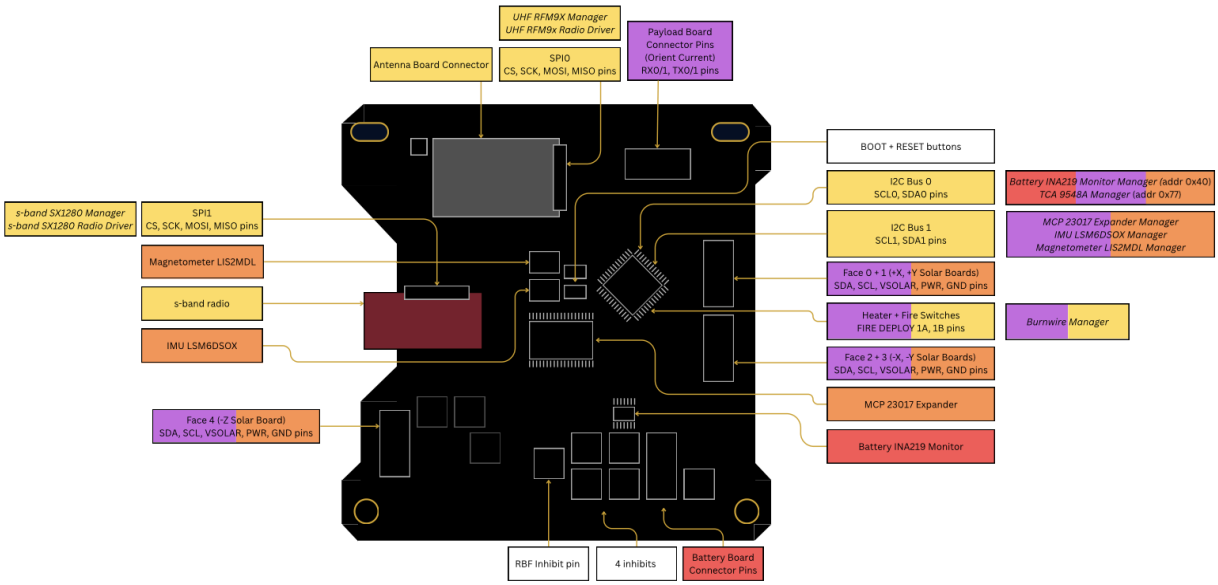

Figure 4A
flight controller board labeled fly-out (front-view)
Legend: (red: EPS) (orange: ADCS) (yellow: comms) (purple: payload)

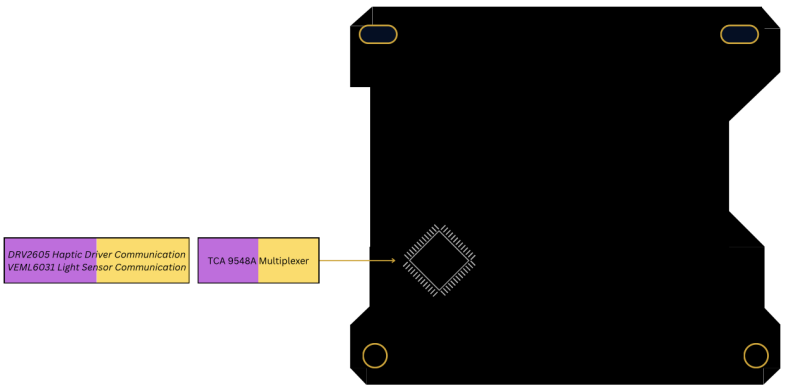

Figure 4B
flight controller board labeled fly-out (back-view)
Legend: (red: EPS) (orange: ADCS) (yellow: comms) (purple: payload)

Design Considerations

During the research of the hardware components, several discussions were paramount in ensuring the reliability and robustness of the satellite. I researched the concerns and my conclusions are articulated below.

Radio Concerns

Deployment Debaucles

Our HUCSat is a 2U satellite, and therefore it is twice as long as it is wide. Subsequently, when the satellite goes to deploy by initiating the burnwire, if there is not enough tension on the wrapped antennas, it is possible for one end to release and the other to snag and stay shut, severely degrading communication.

To ensure a reliable deployment, we integrated several design features: the first was a hardware choice, where we designed the burnwire to not only go around the antennas, but our payload board as well. Since our payload is by default an open-spring contraption, this increased the tension on the antennas. In addition, we kept in the default, fault-safe VL6180 proximity sensing module located underneath the antennas. This sensor detects when the antennas are deployed, and if they are not still deployed post-burn, this may indicate the wire experienced partial fraying via the heated resistor. The finite state machine will then redundantly initiate another burn.

Optimal Antenna Length

For context, antennas are constructed to receive incoming electromagnetic (EM) waves [18]. An EM wave consists of both an oscillating magnetic M- and electric E- field. When the E-field hits the antennas, electrons inside the antenna feel a force and accelerate back and forth, creating an alternating current. This current is later picked up and recovered by a receiver to get the original signal.

The length of the antenna matters crucially for the quality of the signal. Because UHF waves are very short, the E- field changes direction very rapidly. If the antenna is the right size, the electrons along its entire length work together to reinforce the alternating current and produce a strong signal (signal strength is often notated as RSSI). However, if the antenna is not the correct length for the incoming wavelength, different parts of the antenna are out of sync, causing a weaker or non-existent signal.

For our 433 MHz UHF antenna, prior flights flown by CalPoly Institute suggest an antenna length of 170 +/- 5 mm. Given our team had only one set of antennas, the two tape-measure antennas were initially cut to 175 mm to allow for post-assembly trimming and finer adjustments for optimal resonance.

Sharing UHF Frequency

One noteworthy concern was that our licensed UHF of 437.4 MHz is the same frequency as five other CubeSAT satellites flying under the same deployment mission. Fortunately, the downlink communications to a terminal unit would not be a noticeable issue, since we were using LoRA (long-range) communication, and LoRA only supports half-duplex for a very brief transmission time. Therefore, the likelihood of simultaneous transmissions is incredibly low. However,

uplink communication to the satellite would need to be coordinated. Here, HUCSat has several options:

1. Option 1: Designate an exclusive window of 10 seconds for each of the 6 total satellites to do their UHF transmissions. While this sounds simple in practice, clock drift still poses a great issue, even with highly atomized clocks (RTCs) that the CubeSATs operate on.
2. Option 2: Each of the satellites can choose a different LoRA spreading factor. However, increasing the spread factor causes a decrease in the coverable distance. Moreover, it takes almost an entire second for a LoRA with a spreading factor of 8 to move through the ground-stations and properly get decoded.
3. Option 3: Each of the satellites includes a LoRA sync word. Therefore, when a ground station receives a message packet from the satellite, it first checks the packet to see if the sync word exists (and re-route if not for this ground-station). Sync words are a reasonable option because they shrink the time to decoding a packet to a few milliseconds.

Radiation Concerns

The main concern was for the satellite to acquire TID resistance in the event of a SEU. TID resistance is the ability for hardware to cope when it is hit with ionizing radiation. SEU stands for single-event upset, and in the context of space, this occurs most frequently due to radiation.

Solution 1: Watchdog
The Watchdog is a radiant-tolerant piece of equipment based on a previously-designed TLV1704 comparator network from Texas Instruments. The circuit works by expecting a ping (or a "pet") from the microcontroller every 26 seconds, or at the 54 second mark when the FCB has recently undergone a re-boot. The "pet" works by setting a pre-specified pin HIGH then LOW. If no pulse is detected, the board is reset by pulling a RESET line low [3]. In effect, the Watchdog eliminates concern of cosmic rays causing soft failures.

Solution 2: Epoxy
Epoxy is applied to outward-facing chips to shield against radiation and prevent outgassing [20].

Solution 3: Memory
CircuitPython firmware sets apart 1024 KB by default for Non-Volatile Memory (NVM) to ensure the fundamental hardware stays in-tact during a boot reset.

Temperature Concerns

Our orbit is at the same elevation of the ISS (~400 km), and temperature conditions at this orbit range between -65 to 125 C [1]. This leaves two primary concerns for our sub-systems:

Battery Usage

Batteries behave poorly under extreme temperatures. In sub-freezing settings, chemical reactions slow [21], and in excessively warm environments, batteries charge and discharge at a faster pace, leading to shorter lifespans [22]. To protect against the cold, we place a thermal blanket around the battery and provide heating-tapes as an added precaution. Thermal simulations conducted by our Bus Electrical team verify that our batteries have behaved fine in low-temperature conditions.

Nithinol Orientation

Recall we are creating a potential (approximately 3.3V) to the nitinol springs to shrink the coils and orient the payload. However, because space is a vacuum and the primary mode of heat transfer from a heated-coil current is through convection, the nitinol will cool significantly slower than on Earth. To ameliorate this concern, I instantiated the following protocols:

1. During times where the satellite is away from the Sun, we release any current from the nitinol to allow it to cool and check the light sensors every two minutes to see if any one light sensor exceeds a threshold. This threshold symbolizes when we are beginning to come around the Earth into the path of the Sun.
2. When at least one light sensor exceeds the threshold – which we can update from our ground-station at any time – this signifies we are now under partial or full contact with the Sun. At this point, we only make a maximum of two adjustments for current, and we will send up current to at most one nitinol coil at a time. We make two adjustments because one will be at the start of seeing the Sun and one will be at the receding end. Because our orbiting period is 90 minutes, we expect to only have to check our light-sensors at a frequency much longer than 2 minutes when we are in contact with the Sun.
3. We utilize PWM to ensure the current is modularized.

Security Concerns

Finally, I focus on security concerns. Satellites are targeted in one of four main ways: kinetic attacks (such as missiles), non-kinetic attacks (such as EMS lasers), spectrum-styled attacks (such as jamming and spoofing), and software-styled attacks such as authentication theft and command injections. This section focuses on the software-styled attacks, and discusses two places where we enact robust security protocols.

Radio Codes

To prevent an anonymous user who knows our call-sign to send a “self-destruction” command to the satellite, we send for each packet a super-secret code encoded in a 128-character bit string.

OSCAR Commands

Our Command-Data Handler allows the ground-station to send up commands for the satellite to update different configuration thresholds or to do specific tasks. The CDH uses OSCAR, which stands for open source cryptographic architecture. It allows for commands to require HMAC authentication to successfully operate. Specifically, the ground-station will first generate a symmetric HMAC key using a shared secret key known only between itself and the satellite. Then, for each command the ground-station wishes to send up, it will sign the payload (command) packets with the HMAC. The satellite then recalculates the HMAC using its shared secret key to verify the HMAC was a derivation of the shared key. If this is the case, the command is authenticated and executed on the satellite.

2. Flight Controller Board Design

We now deep dive into the FCB Subsystem. This section is primarily focused on the design progression of the FCB, from previous versions to the current in-use version.

FCB Version v1A

We first started with the BeagleBone Black. We quickly diverted from using the BeagleBone Black for several reasons. The first was due to its bulky size and that it could not reliably fit into the 2U satellite restrictions without adjustments to the rest of the rack-like system. In addition, though BeagleBone Black consumes little power, it had very little support from a C/C++ perspective and little ability to modify its internal design once manufactured.

FCB Version v5A

Our first main iteration of the Flight Controller, V4, used a RP2040 chip. This is a Dual-Core ARM cortex-M0+ processor. ARM is a company that designs processor chips and their architectures. The chips are designed for 3 main use-cases: A-profile, R-profile, and M-profile. The A-profile chips are used for laptops and smartphones. The R-profile chips are used for real-time systems, such as robotics or cars. The M-profile chips are used in sensors, embedded systems, and small microcontrollers. All these chips do is simply run instructions. The set of instructions that a chip can understand and operate on the hardware-level is called the Instruction Set Architecture (ISA). The two main branches of ISAs that ARM has developed are the ARM64 and ARM32. ARM64 runs instructions in groups of 64 bits, whereas ARM32 runs them in 32 bits. In recent years, ARM has introduced new variations of these two ISAs that offer more capabilities and optimizations.

The main commercial variations are labeled with version numbers v6-v9. For instance, ARMv7 is an ARM32 (32-bit) that offers better performance at low power consumption. This is ideal for mobile devices. ARMv8 allows you to perform ARM32 (32-bit) and ARM64 (64-bit) instructions on the same chip. This enhancement allows for virtualization. Finally, ARMv9 is an ARM64 (64-bit) that offers enhanced security and better performance for large parallel computations. This is ideal for machine learning applications.

Commercially, ARM groups the many varieties of chips into groups called families. The Cortex family is the ‘core’ set of products that ARM offers. For breadth of content, the Neoverse family encapsulates high-performance chips; the SecurCore family deals with secure chips; and CoreLink family for interconnected chips. The RP2040 chip is one of these Cortex chips and runs on M-profile (so microcontrollers). The chip has two cores on it, hence the ‘dual core’ name, and this allows the chip to run operations in parallel.

The RP2040 has a nice set of design specifications, which we will go into now.

Clock Time

RP2040 has an incredibly fast clock that can run up to 133MHz. This means the chip can execute 133 million cycles a second, and each cycle would take 1/133,000,000 = approximately 7.52 nano-seconds to complete. Not all instructions can be processed within one clock cycle, so the number of instructions would be less than this number threshold. This is considered a moderately-fast processor sufficient for most applications.

Temperature Constraints

Given the complexity of space, cold-temperatures can cause the RP 2040 to reduce in speed. Its operating temperature is between -40 C to 85 C. NASA states that satellites in low-earth orbit (LEO) can experience temperature fluctuations between -65 to 125 [1]. This means we will need thermal shielding and engineer a way to ensure that it works beyond its desired range. RP 2040 has a built-in temperature sensor that we can use for testing purposes.

Power Consumption

RP2040 has five separate pins where power can be supplied to. The chip will work best if one operates it at 3.3V. Some of the pins can be disconnected from the power supply (in other words, can be not used) depending on the application. If opting for a very simple application, one can simply connect all the relevant pins together to a single 3.3V power supply.

1. IOVDD (Digital IO Supply) pin: power supplied here helps with the chip's I/O system. It works at a nominal voltage between 1.8-3.3V.
2. DVDD (Digital Core Supply) pin: power supplied here helps with the chip's core logic. It can either be separately given voltage at 1.1V or hooked up to the same power supply as the IOVDD (1.8-3.3V), as it has its own regulator inside the pin.
3. VREG_VIN (On-Chip Voltage Regulator Input Supply) pin: power supplied here helps with the chip's regulator, power-on-reset, and brown-out detection system. It works between 1.8-3.3V. This must be powered on, regardless if one uses the chip's regulator.
4. USB_VDD (USB PHY Supply) pin: power supplied here helps with the chip's USB PHY system. This system, where PHY stands for physical layer, is responsible for converting the digital signals in the chip to the electrical signals that travel within the USB cable. It must be powered on at 3.3V. One can connect it to the IOVDD assuming IOVDD runs at 3.3V. This pin is optional, and if it is not used, simply tie it back to another pin that is running at 1.8-3.3V.
5. ADC_AVDD (ADC Supply) pin: power supplied here helps the chip's ADC (analog-to-digital signal converter). It can operate between 1.8-3.3V, but is it best above

2.97V, as anything below this will compromise performance and longevity. As with many of the other pins, ADC_AVDD can be hooked up to IOVDD.

Figure 8 details all of the main pins available on the RP2040, which include IOVDD (1/10/22/42), DVDD (23/50), VREG_VIN (44), USB_VDD (46/47), and ADC_AVDD (43).

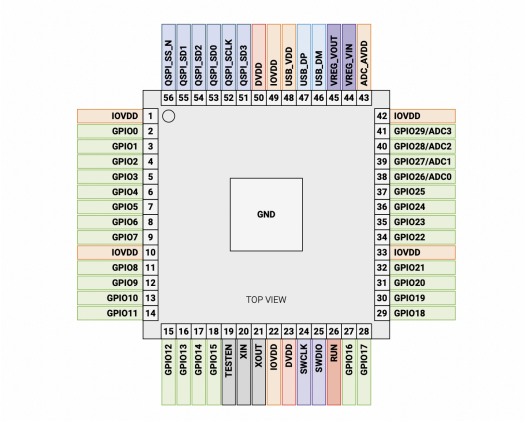

Figure 8
RP2040 Pin-Layout

RP2040 also offers low-power, sleep, and dormant modes. This can be useful for the harsh conditions of space and battery fill-ups.

Memory Constraints

In total, the RP2040 has 284 KB of memory. It has both ROM and SRAM. The 16 KB ROM is fixed at the time of manufacturing and starts at address 0x00. It contains information related to the startup and boot sequence, and basic utility libraries like floating points [2]. Its SRAM comes with 264 KB. Physically, it is divided into six banks, making it easier for master wiring. It offers drag-and-drop programming using USB storage. If the chip's USB port is not hooked up to anything, then that 4 KB of storage can be used as additional memory.

FCB Version v5B-v5D

Several transitions were made from v5A to v5D. Each transition and justification is listed below for a comprehensive design discussion [15]:

v5A to v5B

This transition included the following changes:

1. The initial board used DF13 connectors to connect to outside subsystems. However, these connectors were too easy to accidentally pull off the board, and so a design change was made to use tighter wire connectors.
2. The underlying microcontroller chip RP2040 was exchanged for RP 2350. RP 2350 is all else equal to RP 2040 with the exception of offering 20 extra pins through a MCP GPIO expander. While more expensive, the usage of the MCP allowed the team to offload the solar panel pins and deployment heater ENABLE switches.

v5B to v5C

During the prototyping of v5B, the boards received a design review from CalPoly and Portland State University. The critiques made were mostly non-critical, but their inclusion would provide better reliability. These include:

1. Switching to a 12-16 coin cell battery instead of a supercapacitor. Supercapacitors are more spontaneous in space than a coin cell battery (Voyager wants conservative safety metrics), and the power path was made more efficient under the adoption of the coin cell.
2. There were concerns that because the Watchdog's Load Switch could be manually toggled on and off that it would accidentally be powered off. Without the Watchdog mechanism, the satellite would have less capability of rebooting in the event of a software failure. Therefore, the hardware was changed to keep the Load Switch always on.
3. Instead of just pulling the RST line to reset the board, v5C does a harder reset by pulling the ENABLE pin on the voltage regulator.
4. Finally, the chip added 8 MB of optional external RAM through an SSD card.

v5C to v5D

This transition retired all known bugs. These bugs included the following:

1. Added pull-up resistors that were omitted on previous versions to stop segmentation faults in the underlying software.
2. Added bus protection IC for I2C bus, additionally inserting pull-up resistors here as well.

FCB Final Version: v5D

Our Flight Controller Architecture is based on the PROVES Kit V5 Flight Controller (FC) Board. This board handles the overall commands of the spacecraft and supports all other subsystems [3].

When deciding on the V4 Flight controller, we had considered several design drawbacks. The main concern surrounded the RP2040's core functionality. While the M0 core offers energy-efficiency in its design, it's more well-tuned for basic embedded systems and lacks functionality for more complex multi-streaming systems [8]. We eventually chose a later

development of the Flight Controller that upgraded the RP2040 to an RP2350. Developed also by Raspberry Pi, this microcontroller features the following improvements [6]:

1. **Cores**: The RP2350 has two Cortex-M33 cores. M33 utilizes ARMv8-M compared to RP2040's M0 which uses ARMv6-M. Therefore, unlike M0, which can only handle 32-bit executions, M33 offers execution in both 32-bit and 64-bit architecture. The 64-bit is key, as it allows for more data-handling at once and therefore can allow for better multitasking. For our satellite, which will be dealing with a finite state machine and handling multiple sources of data and transmissions at once under a limited power source, this enhancement is critical [9].
2. **Clock**: Clock speed has been amplified from 133MHz on the 2040 to 150MHz.
3. **Memory**: Its SRAM has been upgraded from 264 KB to 520 KB. In addition, the PROVES Kit does not specify an Operating System (OS). Rather, it is designed to act as an embedded system with firmware to control the hardware. It relies on only 1.6 MB of CircuitPython flashing. Note that the BeagleBone Black *did* operate on a Linux OS, therefore it took up more space (412 MB of RAM/4 GB of flash)!
4. **Voltage**: The RP2040 does not have an on-chip programmable low dropout regulator (LDO) and would require a radiated-protected external regulator to supply a constant 1.1V core voltage to the microcontroller. The RP2350 does offer its own voltage regulator*, meaning that it not only saves space in an already-packed CubeSAT environment, but also allows for dynamic adjustment of the core voltage workload. For instance, if the flight controller were in an idle state requiring less computation, the voltage could be dropped to save power consumption.
5. **Security**: Further research discovered that the RP2040 has no built-in security features like code protection or firmware encryption. In contrast, RP2350 has an expansive security architecture built around the hardware security technology "ARM TrustZone" for M-type architectures. The main features most useful to our mission are its SHA-256 encryption of messages and booting support.
6. **Temperature**: the on-board voltage regulator can allow for over-temperature protection for the microcontroller (for context: a voltage regulator is an electric circuit built from transistors that converts a high voltage into a lower voltage safe for a chip's internal logic). The added protection is achieved by including a built-in temperature sensor near the regulator's transistor that deals with the highest amount of current initially coming in. If the transistor is a BJT, the sensor is usually found near the base terminal, and if it is a FET, it's most usually near the drain terminal. Only when the transistor drops to 20C below a pre-defined temperature threshold is when the board will restart regulation [7].

Figure 9 showcases a more detailed schematic of the RP2350 and its directly connected pins.

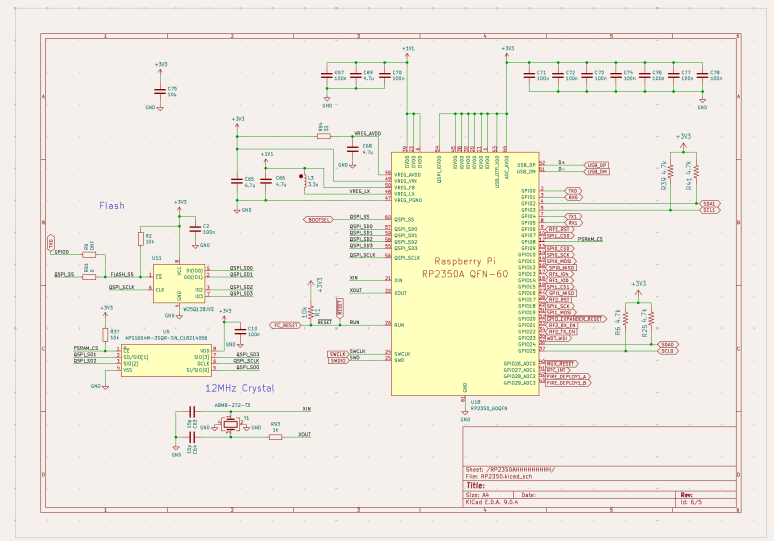

Figure 9

RP2350 Microcontroller Schematic

The RP2350 offers similar performance to the RP2040 in other areas, such as:

1. 30 GPIO Pins, enough for our 5 main subsystems
2. 16 MB of off-chip Flash memory through its QSPI bus.

3. Satellite Software Design

The software system is a customized CircuitPython software platform that extends the capabilities of the open-source PROVES Kit RP2350 v5a CircuitPython Flight Software and Ground Station [5]. On a high level, this code boots up the satellite and initializes software manager classes, collects and aggregates data across sensors, and uses this data to navigate through a pre-defined finite state machine for the satellite. The finite state machine determines what main action the satellite is doing at a given point in time. This is important because the satellite by law needs to follow a strict series of steps for bootup, and likewise needs to be able to handle situations of software failure. The repository setup of the software is as follows:

```
├── src
│   └── flight_software
│       ├── fsm
│       │   ├── data processes
│       │   └── state processes
│       ├── lib
│       │   └── pysquared
│       ├── repl.py
│       └── main.py
```

```
└── typings
    ├── board.pyi
    └── gc.pyi
```

A more technical description of each file is examined below.

Main Functional Loop

The main.py file initiates when we turn on the satellite for the first time or conduct a reboot. It proceeds as follows:

1. Initializes a Real-Time Clock (RTC) object for satellite functionality.
2. Initializes four non-volatile memory (NVM) registers that track the following information: the number of times the satellite has booted up (whether by design or due to a warm/cold reset), the number of errors, whether the satellite has deployed or not, and the number of times the satellite has reset (warm or cold) due to an exception.
3. Starts a 30 minute countdown timer from which all subsequent code will begin. This 30 minute timer is only triggered if we have either not yet deployed or if we have booted less than twice, to avoid unintentional long waiting loops. The 30 minutes is to comply with regulations by NASA for deployment.
4. Initializes Logger and Config objects.

Once these initializations are complete, we define two new functions. The first is a sleep helper object. This function undergoes a safe sleep to act as a guardrail in case of power consumption or technical failure. The second function we define is the main asynchronous loop, which does the following initialization steps:

1. Initializes a Watchdog object to handle the hardware watchdog timer. To not cause a cold reset, it must be petted approximately 26 seconds.
5. Initializes the two SPI buses, which will be used for UHF and s-band radio communication.
6. Initializes the two I2C buses, which will be used for managers such as the battery power manager, TCA sensor multiplexer, and MCP GPIO Expander.
7. Initialize the MCP GPIO Expander to talk to additional sub-components.
8. Initialize the radios, specifically the s-band radio, UHF radio, and UHF packet manager. The package manager module exists between the beacon and radio module objects. On one end, it takes data from the beacon object and further fragments it for sending. On the other end, it reassembles data from the radio module for transmission. The radio module is the bare-bones software that controls the actual operations of the satellite's deployable 437.4 MHz RFM9x radio antennas through a RadioProto interface. RFM9x is a LoRa (long-range) radio that, under dense environments, can work reliably across 2km [12]. However, the vacuum of space allows its information to travel virtually unscathed,

being able to reach our telemetry network of satellites on the ground. The UHF radio mainly deals with data like telemetry, system health, and commands from ground station.

9. Initialize the payload pins, where we will send current through to pull our rotatable solar board in one of four directions at a time.
10. Initialize the TCA sensor multiplexer and light sensors on all four faces (+/- X, +/- Y faces).
11. Initialize the detumbler components, which includes the LSM6DSOX IMU manager [13], LIS2MDL magnetometer manager [14], detumbler manager, and magnetorquer manager.
12. Initialize the CDH and Beacon objects. As a recap, the CDH deals with handling data received by the satellite from radio. It incorporates parsing, validation, and execution of the commands. The Beacon object collects data from many subsystem components, packages them using JSON, and sends them to the UHF radio package manager object.
13. Initialize the battery power manager.
14. Finally, I initialize the finite state machine and data process objects. The FSM is what controls the satellite's every operation and what it should be doing at each point in time during the flight.

For added safety measures, I wrap all initializations besides those dealing with pin-definitions in try-except blocks. If the battery power manager fails to initialize, I do a warm reset by calling microcontroller.reset(). If the s-band radio, UHF radio, UHF packet manager, CDH, or Beacon fail to initialize, I undergo a cold reset by sleeping for a synchronous (blocking) amount of time that supersedes the Watchdog's internal limit (ex. time(30)). This blocked-amount of time is configurable via the Config file in the event we see from the ground-station an inability to conduct a proper cold reset. The reasoning for such is that Watchdog timers can vary in their datasheet's description for the length of time it takes to activate a reset: tests conducted by CalPoly have shown it can take between 23 and 53 seconds, with the majority taking 26 seconds to trigger. Therefore, I wait time(30), and I can easily adjust this number from the ground-station as needed. For further safety, I also define two functions within the main asynchronous loop that can turn on and off all power to the solar board faces. The CDH is defined to have an exec() command that will enable people in the terminal unit to trigger these functions. The exec() command can only be triggered by individuals who have access to a secretive activation code, which is designed to never be pushed to the code repository and known at the minimum only to the Bus Computing Lead and Chief Engineer.

After the initialization steps, the asynchronous function enters into a loop that continues indefinitely unless shut down with power or rebooted for safety, in which case the file will restart from the beginning. This infinite loop incorporates three main actions:

1. Conduct one step of the FSM.
2. Pet the watchdog to let the system know it is still healthy.

3. Conduct a system power check. This checking process incorporates the following steps: first, the FCB clears up any unused memory with a garbage collector; then it periodically sends down a beacon with the satellite's license and a health status report; later, it waits for any commands from the ground station; and finally, it conducts a quick safe sleep to conserve power (analogous to a while True(sleep) loop).

DataProcess and FSM Files

We now dive into the two important file types constructed for this thesis: the data process and the finite state machine files.

DataProcess Files

The DataProcess folder contains one main file that defines the DataProcess() class. This class provides asynchronous functions to grab any and all the data we may need across the different subsystems. The FSM files use an instantiated DataProcess object to make branching decisions.

FSM Files

The FSM software is grouped into a singular folder that contains a main fsm.py file as well as all the states for the finite state machine.

In fsm.py, we define an FSM() class that is utilized by the main.py file, conducting one step in its infinite loop. The FSM() first initializes all the state objects defined in the FSM folder, with the default starting state being "bootup". Only one single state is active at a time and is started up as an asynchronous coroutine. Each state is a class defined by a run function, stop function, and check if done function. The run function is an asynchronous, one-step iteration of the state's actions. Once the actions complete, the function will set a variable indicating that it is done.

When executing a single step of the FSM, we first check the current status of the ongoing state. If the state object indicates it is done, we stop the coroutine by calling the state's stop function, then initialize the next state based on a pre-defined sequence. The sequence is defined below:

Deploy + Time (Minutes)	**Event**
D+0	On-board computer begins with 30-minute delay timer
D+30	**Bootup** where satellite powers on all critical systems
D+35	**Detumble** initialization
D+90	**Deploy** antennas and payload

D+270	**Orient** payload

First is the bootup stage. During the boot-up stage, most CubeSats go through a critical sequence to establish basic functionality, power stability, communication, and readiness for mission operations. For communication, two main types exist:

1. Beacon: satellite sends stuff to ground-station, aka transmits
    a. The beacon function is a wrapper around a lower-level defined function, radio.send()
    b. will ping using LoRA, if any ground-station network of lora picks it up (we need to get in their database of satellites), we'll receive it on our end, so no need to be over Cambridge
    c. We can also keep track of the boot count in this beacons
    d. End your message with a call sign, aka an identifier for our satellite
    e. We'll have on our website (legally required) for how to talk to our satellite
2. Listen: satellite listens to ground-station for stuff, aka receives
    a. The listen function is a wrapper around a lower-level defined function, radio.receive()
    b. Receiving is much less power than transmitting (~1W): beacon has to be under a certain length, short as possible

Next is the detumble stage, at D+35 minutes. The detumbling stage watches the magnetic field and the angular velocity magnitude of the satellite. If the angular velocity is over some threshold determined by experimentation, then the detumbling event is triggered. Namely, the B-dot equation is calculated using the magnetic field and angular velocity to produce a magnetic dipole. The dipole (3D vector) is the mathematical quantity used as input into the magetorquers on board to add velocity to the satellite in one or more of its three axes. The data on the field and resulting velocity is then pulled again. If the new angular velocity is below the threshold, the detumbling stage is over, and the next stage starts.

A noticeable question arises here: how long should we wait between sending a signal to the magnetorquers for an adjustment and measuring the next set of data? This answer exposes a trade-off between reactiveness and physical constraints. The more reactive the system is, the quicker it can correct itself. However, having too quick of a data read may not reflect the changes made by the magnetorquers yet, causing inconsistency in readings and chasing sensor noise. We chose every 2 seconds as a default (yet configurable!) threshold, and here is why:

1. B. Wie, "Space Vehicle Dynamics and Control" (2nd Ed) – a textbook frequently cited in attitude control literature, recommends 1–10 second update intervals for B-dot detumbling using magnetorquers in small satellites, depending on dynamics and field strength.

2. NASA Ames Tech Reports on the PharmaSat and GeneSat CubeSats describe B-dot updates every 4–5 seconds, using rod magnetorquers for detumble.
3. CubeSat Design Specification (Cal Poly) doesn't mandate timing, but many published implementations report 2–5 second loops as optimal for stability. Cal Poly CP6 ADCS design paper has their CP6 mission at 5 seconds; Stanford's Quakesat monitors theirs at 2.5 seconds.

After detumble and proper battery checking is the deploy stage. AT D+90 minutes, we burn the wire for the antennas and payload to be released.

Finally comes the orient stage. At D+270 minutes, controlled electric current will run through each spring selectively. When a spring heats up, it will remember its original "coiled" state from heat training, thereby pulling the payload leg and solar board to its intended side. Once the current stops, the springs will cool, and the legs associated with each spring will extend back out.

Interim each step is communication health-status reports and power consumption monitoring. Namely, during the main.py loop, we check the power and also send down a beacon to the ground-station describing the status of the satellite, waiting for any commands as needed. For the power, if after any step of the FSM execution the power becomes short, our main function immediately goes into critical power operation mode. The watchdog is also repeatedly petted during this iterative process until the power reaches pre-defined normal levels.

Driver and Manager Class Files

There exist a host of driver and manager class files that aim to either communicate and interact with the underlying hardware. Driver files are more low-level and focus on connecting literal hardware component pins to software variables based on publicly-released datasheets. The manager class files allow one to set up usable functions that take advantage of the underlying hardware capabilities. I highlight several notable examples. Examples with an asterisk * are newly created and thus a highlight for this thesis. All files are written in CircuitPython.

Driver Files

*VEML6301 Driver: this driver sets up the VEML6301 light sensors with pin allocations and read/write bus capabilities according to publicly-released datasheets.

*DRV2605 Driver: this driver sets up the DRV2605 haptic drivers with pin allocations and read/write bus capabilities according to publicly-released datasheets.

Manager Class Files

EPS Sub-System

INA219 Manager: in software, we can monitor how our battery system is doing via a INA219 class monitor. This class defines a RTC (real-time-clock) and gives us battery voltage data. When undergoing testing for our specific battery board, 8.4V appears to be most ideal, and anything over > 8.2V should be considered full (higher than the datasheet's 8.0V case). At 5V, the batteries are fully discharged, and the satellite needs to get to emergency saving. At 4.8V, the battery safety system will trigger and all power is cut off. To re-provide power, one would need to jump the circuit via solar cells, but this is as a last-resort.

ADCS Sub-System

*Detumbler Class: this class calculates the satellite's dipole moment based on angular velocity and magnetic field. Magnetic field is expressed in Telsas and angular velocity in rad/s (note: some of this code was re-purposed from older code in the Open Source Space Foundation).

*Magnetorquer Manager: this manager contains several functions that utilize the dipole moment calculated from the Detumbler Class to actually move the haptic drivers. For instance, set_dipole_moment() gives the correct value to the DRV2605 drivers (for rotating) from a series of translational calculations, starting with the dipole moment to current to datasheet-determined DRV2605 driver range values.

Comms Sub-System

Packet Manager: this manager allows one to send and receive packets over radio, where the radio is defined as either RFM9x or SX1280. The send function sends data in bytes using data.encode("utf-8") to ensure that each ASCII is represented as 8-bytes. The send function uses _pack_data under the hood, and _pack_data returns list of packets to send. Each packet includes 1 byte for the packet identifier, 2 bytes for the sequence number (0-based), 2 bytes for the total number of packets, 1 byte for the signal strength of the packet (RSSI), and all remaining bytes refer to the payload.

Beacon Class: this class sends pre-defined status messages periodically via Beacon.send. Underlying the Beacon class is the PacketManager. Specifically the Beacon sends "state" comes from build_state. We can add items to the state via _add_system_info(state) or _add_sensor_data(state). Each message ends with a call sign.

1. The send function will ping using LoRA, if any ground-station network of lora picks it up (we need to get in their database of satellites), we'll receive it on our end, so no need to be over Cambridge. We can also keep track of the boot count in this beacons

2. The listen/receive function makes the satellite listen to ground-station for anything. Receiving is much less power than transmitting (~1W): beacon has to be under a certain length, short as possible

CDH Class: this class defines commands we can send to the satellite and how they're handled/aka implemented. We define for our own use a new value in cdh.py. One can use the update_config function to update the code config changes when a satellite is in space.

Payload Sub-System

Burnwire Manager: this manager allows a user to call a function that initiates a burn to deploy the antennas and payload. One can specify the burn duration as needed.

Configuration File

The configuration file defines variables for our satellite, such as sleep duration, reboot time, thresholds for battery voltage and current, call sign/radio license, and more. The configuration file was primarily utilized to set thresholds via commands from the ground-station post-launch, providing robustness in the event that any threshold needs to be changed for optimal satellite operational performance. These thresholds are defined below:

Threshold Table

Threshold	Description	Default	Changeable
orient_heat_duration	# seconds we heat the designated nitinol spring	10.0	yes
orient_light_threshold	The light threshold from get_light() where we determine if we're at the Sun or not If we are not in Sun, wait 2 minutes before another re-check of light sensors, otherwise, wait [periodic time] minutes	10.0	yes
orient_payload_periodic_time	The amount of minutes we wait before we re-check light sensors and potentially pull down nitinol	25.0	yes
orient_payload_setting	Off: don't pull nitinol, ever On: potentially pull nitinol if we're in the Sun and one light sensor is strongest	0/1 (def: 1)	yes

fsm_batt_threshold_orient	Battery ok to orient	6.0	yes
fsm_batt_threshold_deploy	Battery ok to deploy	7.0	yes
deploy_burn_duration	# seconds to burn the wire	10.0	yes
detumble_adjust_frequency	# seconds will we sleep in between adjustments to detumble	2.0	yes
detumble_stabilize_threshold	At what angular velocity magnitude will we be considered stable	0.5	yes
detumble_max_time	# seconds to try and detumble before stopping	90.0 min (5400.0 sec)	yes
reboot time	# times we will sleep over ‘sleep time’ when reboot up main.py	3.0	no
sleep time	Amount of time in minutes to sleep during main.py boot up	30.0	no, required by law

Prior Design Versions

Several software design changes were made over the course of the project.

First, the first software architecture design once had a single file for each data piece to be collected for the DataProcess. The latest version of the DataProcess file now consolidates all collections of data into one clean class.

In addition, previous designs of the software utilizes processes to promote parallelism. However, since CircuitPython is not compatible with processes, I shifted to asynchronous scheduling for continued concurrency.

Finally, the pre-defined sequence has been simplified significantly since its start. Before, we had nearly seven states: bootup, detumble, deploy antennas, deploy payload, orient payload, comms state, and battery state. The deployment of the antennas and payload is now done in unison, now termed under a single “deploy” state. In addition, the comms and battery state are not singular states in of themselves, but are always enacted at the highest level in the main.py loop. This offers more robustness in the event we need to send an emergency signal or that the battery power degrades significantly in a short span of time.

4. Testing Procedure

The PROVES Kit v5D offers close to 100% unit testing via pysquared. An example of one unit test was to conform code to ensure it remained out-casting safe. However, differences on Python unit tests and the CircuitPython board can make it difficult for verification. In other words, a test could fail on the laptop but succeed on the board. In response to these concerns, the codebase first underwent full rounds of static analysis. Moreover, apart from unit tests, repl.py offers a separate code environment to dynamically specify and execute tests. For context, repl.py is a runtime interpreter that functions as CubeSAT's onboard terminal from the computer. This interpreter enables a user to execute commands and diagnostics to and from the satellite. I utilized the repl.py extensively to test add-on functionalities and baseline systems integrated via the PROVES Kit.

When setting up the board for testing, a cluttered board would need to be initially cleaned for full confidence in subsequent testing protocols. To do so, the FCB would first be plugged into a running laptop using a USB cable. Then, the FCB physical buttons BOOT and RST would be clicked, causing a full reboot of the system. To flash the proper v5D hardware, a .uf2 file would need to be clicked and dragged into the FCB USB drive that appears on the laptop when the laptop is plugged into the FCB. Finally, with the board all cleaned, the required libraries from the CircuitPython GitHub can now be added to the board. To do so, one would navigate via terminal to a locally cloned repository of CircuitPython and execute the command "make install-flight-software". In this command, we specify the flag BOARD_MOUNT_POINT (which is where the installed libraries should go) to the directory that the FCB is mounted on inside the laptop, which typically appears as /Volumes/PYSQUARED.

Once the libraries have been installed on the cleaned board, to run the tests, a user simply needs to screen into the board to run the tests specified within the repl.py file. This can be achieved by executing "screen [make list-tty]", where the bracketed notation refers to the output of the command inside. By screening into the board, the FCB will run the following files in order: boot.py (often blank, so skipped prematurely), main.py, repl.py. In actual orbit, whenever the board has to reboot, such as due to a Watchdog failure, the booting process will restart the main.py file, and when that code file is finished, start up the repl.py file. In real-time orbit, the main.py file will have a while True loop, and so the repl.py test-script file will never be instantiated, as intended.

If changes are made to the testing scripts inside repl.py, a user simply has to press CTRL+D when screened into the board, which will start a soft reboot and update the main.py and repl.py files it needs to execute.

The following set of tests were run over a span of four full months (October 2025 to January 2026) to ensure full robustness of software-hardware integrations. These tests span three main categories: those that test the DataProcess object, the FSM state integration, and non-FSM states.

DataProcess Tests

Test Type	Tested
Magnetometer	1. Can accurately measure magnetometer magnetic field.
IMU angular velocity	1. Can accurately measure angular velocity and angular velocity magnitude. 2. Can accurately measure angular acceleration.
Battery percentage + voltage	1. Can measure voltage on batteries (7.28/8.0 V for batteries, 0.024 for when just plugged into a laptop for source of power).

FSM Tests

Test Type	Tested
FSM	1. All transitions statically tested. 2. Detumble emergency procedure achieved in the highest level of FSM execution. 3. Deploy only triggers if battery voltage > 6 and should stay in detumble otherwise. 4. Orient should only trigger if battery voltage > 5.5 and should stay in detumble or deploy otherwise. 5. Detumble acc mag should be close to zero for it to be marked as done. 6. Detumble properly stops after 90 minutes. 7. In main.py, added a safe sleep mode that pets the watchdog if the battery is too low, either due to transitions or just the fact it is <= 5 volts.
Bootup	1. Technically already working, basic state that we just have w/ no functionality needed so no need to async make a process. 2. Added in main.py a timer to wait 30 minutes.
Deployment	1. Deployment wires properly send down 5V when activated/burned.
Orient	1. Tested state orient's ability to pick up on config change for experimentation. 2. Tested that current goes down the RX#/TX# pins. 3. Lights do light up when we're trying to trigger some current down one of the RX#/TX# pins. 4. Light shows up on solar board when plugged into PWR/GND

	5. Light sensor detects Light and Lux (FACE0_ENABLE). 6. Fully integrated payload board connected to the nithinol works when heated work. Specifically, the setup we had was, from top to bottom: solar board, nithinol springs, payload board. The nitinol board, solar board, and base board are all held at 3.3V. When we set the pin we wish current to go down to HIGH, there is 3.3V potential on the top of the nithinol and 0V at the bottom, and so current is sent downwards.
Detumble	1. DRV drivers operable via play() and stop().

Non-FSM Tests

Test Type	**Tested**
Comms	1. s-band driver communicates properly between two FCBs. 2. The beacon class instantiates properly. 3. The antenna board powers up properly.

Additional tests had to be performed post vibration testing. Vibration testing is the last stage of the environmental testing procedure where the satellite is shaken for several hours, mimicking the pressure it would be under when flown to the ISS in a J-SSOD deployment tube. To pass the test, no hardware should be visibly out-of-place, and the satellite would have to show signs of life post vibration. This includes the following: external LEDs illuminate, software subsystems properly initialize, and internal sensor readings display logical values.

The following initializations and readings were verified to work post-exam: burnwire (heater enable, fire switch, burnwire manager), detumbler (detumbler manager, magnetorquer manager), light sensor (load switches, light manager), sensors (IMU manager, magnetometer manager, battery power manager), logger (logger files, configuration files), communication mechanisms (I2C, SPI, UHF, s-band, Beacon, CDH), and general pins (incl. MCP GPIO expander). Because all were properly initialized, this indicated no loose wiring came undone.

The following readings were logged and were verified to display proper values: light sensors (lux, light magnitude), battery sensor (voltage), IMU (angular acceleration, angular velocity), and magnetometer (magnetic field).

5. Experiment Procedure

The final following section details HUCSat's comprehensive experimental procedure for the payload. Because the launch is scheduled for April 2026, the conclusionary results will not be published until after the submission of this thesis. Nevertheless, the objectives, experimental setup, and hypothesized conclusions will be elaborated on below.

Experiment Objective

HUCSat's experimental objective is the following: how much additional power intensity do we acquire by rotating solar board panels? Does the amount of power draw it takes to contract coils and initiate rotation outweigh the additional power intensity we get from directly facing the Sun?

General Experimental Setup

The experiment needs to have data sufficiently collected within the first 6 months of deployment, since the satellite will later burn up in the Earth's atmosphere.

A controlled experimental framework will be used to compare rotating and non-rotating configurations of the satellite. On designated collection days, the satellite will operate with solar panel rotation enabled, and on other days, the rotation will be disabled to serve as a control. To avoid confounders, power consumption from other subsystems will be accounted for, as variations in system-wide power draw could incorrectly attribute changes in power availability to the solar panel rotator. For example, a day with unusually high subsystem power usage could make the rotating configuration appear less effective due to the heavy power draws placed on other parts of the system. We're fortunate enough to have a logged proxy to detect for this, namely the battery voltage draw.

The flight software allows an individual to command the satellite to either rotate or remain fixed indefinitely. Commands will be sent from the ground station to the satellite via the command and data handling (CDH) system. The command utilizes a structured message format that specifies the satellite name, authentication credentials, the orientation command, and an argument indicating the desired operational mode. As an example, our message sent up in JSON format could be the following, where '0' indicates fully off (no rotation) and '1' indicates full on (plan to rotate whenever in contact with the Sun). The plan would be to set it on for one full week, then off for another week, and repeat this until the end of the satellite's lifespan.

```
msg = { "name": cubesat_name,
        "password": super_secret_code,
        "command": "orient_payload",
        "args": '0'/'1' }
```

Experimental Metrics

Several measurements are key in evaluating the payload's rotational performance. First, we aim to look at solar exposure lux and light intensity. The intuition here is simple: while the satellite experiences approximately the same duration of sunlight for each orbit, the orientation of the solar board relative to the Sun determines the intensity incident on the panel. We will utilize the light sensor manager's get lux and get light functionality to record and average these results over the span of a week.

The second measurement we aim to collect relates to the power draw intensity. We can quantify theoretically how much power draw it will take for the satellite to rotate: 3.3V is applied for a duration of 10 seconds (note the *default* is 10 seconds; this duration threshold can be changed through configuration updates). Nevertheless, we wish to oversee power draw from the batteries as well as the voltage charge of the batteries. The question we seek to answer then by measuring this is as follows: does the average stored voltage charge of the batteries per orbit increase, stay the same, or decrease when under the influence of the rotational panel (exercising the tradeoff between power draw and more light intensity)? Note that there are nuances to this experiment: principally, the observed voltage delta (the difference in voltage from whether we rotate the -Z panel or not) must be analyzed in the context of concurrent power generation from the fixed panels. If on one day the overall satellite sees less light intensity than on a day where there was far more light intensity, a decrease in the battery voltage stored would *not* be indicative of the rotating solar board performing poorly. This is further justification for why we analyze rotating/non-rotating results over the course of a week and plan to repeat the results multiple times throughout the lifespan of the satellite.

The measurements we need will be sent in queued logs down to the ground-station. Specifically, during our main.py loop, the satellite periodically sends down pings that act as health status reports via a beacon class. The aforementioned metrics we wish to measure will be included in those health checks, and they will be stored by the ground-station computer.

Operational Timing

How many times do we check in a singular orbit whether to re-rotate the satellite payload? Note that HUCSat is expected to spend approximately three-fifths of each orbit in sunlight. When in darkness, the FCB will check light intensity via each of the four solar panels' light sensors every two minutes to ensure timely detection of sunlight when the satellite does make its way towards the Sun. Mathematically, the satellite is considered to be in darkness if the light intensities on all of the light sensors remain below a configurable threshold. If the solar intensity from any one board exceeds a predefined and updatable threshold, then that means the satellite has entered in

range of the Sun, and a re-orientation is immediately fired. Then, subsequent re-orientation checks are performed at 25-minute intervals. This equals approximately two adjustments per sunlit period.

Success Criteria

The definition of success for this experiment is not determined by a pre-set threshold. Instead, we wish to define success under the improvement of the two measurements done between control and actual experiment. That is, the operation would be considered a success if we (1) see more light intensity when rotating the payload board and (2) all else equal, see the same or improved voltage storage held by the batteries.

Conclusion

This thesis presented the software architecture, development progression and rationale, validation methodology, and experimental design of the Harvard Undergraduate CubeSat, with a particular focus on the Flight Controller Board (FCB) and its supporting firmware ecosystem.

To begin, I first described a cursory overview of the main subsystems, including the flight controller board, electrical power subsystem, attitude and determination control subsystem, communication subsystem, and payload subsystem. This discussion likewise included alleviating concerns related to radio, radiation, temperature, and security vulnerabilities. From here, I focused specifically on the design of the flight controller board, defending the prototyping progression and ultimate software ecosystem setup.

This work contributed to open-source software through the construction of new firmware drivers and managerial classes. Without this addition, the incorporation of more modern-technical light systems and haptic drivers would not be possible for future missions aimed to be launched through PROVES Kit.

The complexity of this project ultimately stems from its integration of a myriad of topics: security protocols; computer architecture chip design; firmware integration; software system design; unit and integration testing procedures; and additional overlaps with electrical engineering, physics, and material-science. The novelty of this project is that it is, to the best of our team's knowledge, Harvard College's first ever satellite launch to space, under appreciative support of the SEDS program and the CubeSAT initiative.

References

1. Plante, Jeannette, and Brandon Lee. “Environmental Conditions for Space Flight Hardware: A Survey.” *NASA Technical Reports Server*, 2004, https://ntrs.nasa.gov/citations/20060013394.
2. “RP 2040 Technical Specs.” *Raspberry Pi*, https://www.raspberrypi.com/products/rp2040/specifications/.
3. “Flight Control Board.” *The PROVES Kit Documentation*, https://docs.proveskit.space/en/latest/core_documentation/hardware/FC_board/.
4. Prainito, Christopher, and Milligan Grinstead. *HUCSat ISSNR Status*. 13 6 2025.
5. PROVES CubeSat Kit. “PROVES Kit CircuitPython RP2350 v5b Flight Software and Ground Station.” GitHub, https://github.com/proveskit/CircuitPython_RP2350_v5b#.
6. Raspberry Pi. “Silicon RP2350.” *Raspberry Pi*, GitHub, https://www.raspberrypi.com/documentation/microcontrollers/silicon.html.
7. Raspberry Pi Ltd. “RP2350 Datasheet.” *A microcontroller by Raspberry Pi*, Synopsys, Inc., 29 7 2025, https://pip-assets.raspberrypi.com/categories/1214-rp2350/documents/RP-008373-DS-2-rp2350-datasheet.pdf?disposition=inline.
8. Mittal, Aviral. “System on Chip Architecture.” http://www.vlsiip.com/soc/soc_0003.html.
9. Ballejos, Lauren. “64-Bit Computing.” NinjaOne IT, 7 1 2025, https://www.ninjaone.com/it-hub/it-service-management/what-is-64-bit-computing/#:~:text=While%20a%2032%2Dbit%20system,memory%20and%20facilitates%20effective%20multitasking.
10. *Nanoracks Safety Data Template (SDT) Project Name: HUCSat*. Nanoracks Proprietary | Voyager Space, 8 2023.
11. Davis, Madison, et al. “HUCSat Technical Description.” 26 3 2025, https://docs.google.com/document/d/1qcwlxAkl2BrSL1pQ5HLOR2TVnf7ERlAa/edit. lxAkl2BrSL1pQ5HLOR2TVnf7ERlAa/edit.
12. Needell, Gerald, et al. “Adafruit RFM69HCW and RFM9X LoRa Packet Radio Breakouts.” *Adafruit*, 22 1 2025, https://learn.adafruit.com/adafruit-rfm69hcw-and-rfm96-rfm95-rfm98-lora-packet-padio-breakouts/overview.
13. “Datasheet - LSM6DSOX - 6-axis IMU (inertial measurement unit) with embedded AI: always-on 3-axis accelerometer and 3-axis gyro.” *STMicroelectronics*, life.augmented, 3 June 2024, https://www.st.com/resource/en/datasheet/lsm6dsox.pdf.
14. “LIS2DML Digital output magnetic sensor: ultralow-power, high-performance 3-axis magnetometer.” *STMicroelectronics*, life.augmented, 6 7 2024, https://www.st.com/en/mems-and-sensors/lis2mdl.html.
15. Pham, Michael. “Use of Batteries with AP22652 on V5a Can Exceed the Maximum Voltage Ratings.” GitHub, 4 5 2025, https://github.com/proveskit/flight_controller_board/issues/32.
16. Shingledecker, Robert. “Core Project.” *Tiny Core Linux, Micro Core Linux, 12MB Linux GUI Desktop, Live, Frugal, Extendable*, 1 12 2008, http://www.tinycorelinux.net.

17. “ORESAT.” *Bringing Space to Oregon*, PSU Foundation, Portland State University, 27 12 2021, https://www.oresat.org/satellites/oresat.
18. “Understanding Antennas.” *CCNA Enterprise Infrastructure CCNAv1.1*, Network Academy, https://www.networkacademy.io/ccna/wireless/understanding-antennas.
19. Rizvi, Arslan, and Dong Yang. “Atmospheric Attenuation.” *A review and classification of layouts and optimization techniques used in design of heliostat fields in solar central receiver systems*, ScienceDirect, 2021, https://www.sciencedirect.com/topics/engineering/atmospheric-attenuation.
20. Aygün, Hayriye Hale. “Epoxy Composites for Radiation Shielding.” *IntechOpen*, 21 April 2022, https://www.intechopen.com/chapters/81458.
21. “Low Temperature Batteries: How Does Cold Affect Power Sources?” *City Labs*, https://citylabs.net/temperature-control/cold-batteries/.
22. “High Temperature Batteries: How Does Heat Affect Power Sources?” *City Labs*, https://citylabs.net/temperature-control/hot-batteries/.