CHARMS:

Control of **H**ardware **A**ttitude using **R**eliable **M**agnetorquers **S**atellite

Isaac Brej, Sarah Kopfer, Peter Gibbons, Aidan Oblepias

University of Notre Dame – EE Senior Design

Professor Mike Schafer

May 7th, 2025

In Collaboration with **NearSpace Education** and the **Dream Big Program**

- **Table of Contents**

1 Introduction

Satellites are a cornerstone of modern communication, navigation, and Earth observation, yet their utility is immediately challenged after deployment due to accumulating momentum across all three rotational axes. This uncontrolled angular velocity renders precise orientation—and therefore effective charging, communication, imaging, and sensing—impossible without a capable Attitude Determination and Control System (ADCS) that can critically *detumble* the spacecraft. If a satellite cannot slow its spin within a short timeframe and under strict power constraints, it risks entering an unrecoverable "spinning death," terminating its mission before any meaningful operation can occur.

For small satellites, which are increasingly prevalent in both academic and commercial missions, traditional detumbling and attitude control systems pose significant integration challenges. Most standard ADCS configurations rely on a combination of reaction wheels and magnetorquers, but these systems are often bulky, power-intensive, and require complex interfaces—making them difficult or impossible to incorporate into constrained form factors like 0.5U. While some magnetorquer-only systems for small satellites do exist, they are prohibitively expensive and often deliver limited performance.

The **C**ontrol of **H**ardware **A**ttitude using **R**eliable **M**agnetorquer **S**atellite (**CHARMS**) is an innovative response to this challenge. Designed as a modular, low-power, and low-cost magnetorquer-only ADCS, CHARMS supports satellites as small as 0.5U (10 cm × 10 cm × 5 cm). It offers a plug-and-play solution requiring only power and serial communication with the host satellite bus, with no dependence on system-external inputs. Its reduced yet sufficient sensor suite, compact architecture, and autonomous operation make it scalable, practical, and highly adaptable. CHARMS represents a critical advancement in democratizing access to space, enabling the next generation of small satellites to achieve precise attitude control without compromising size, cost, or power and ensuring mission viability beyond the moment of deployment.

1.1 Problem Statement

When satellites are deployed into low Earth orbit (LEO), they inherit significant angular momentum on all three axes from their separation event. This uncontrolled rotation renders essential operations such as communication, imaging, data collection, and power management impossible. To begin functional operations, satellites must first stabilize their orientation through a process known as *detumbling*, which applies counteracting torques to reduce spin. Successful detumbling is a critical prerequisite for mission success.

Small satellites, constrained by tight limits on power, volume, and mass, often struggle to accommodate traditional detumbling systems. Solutions that rely on reaction wheels, thrusters, or extensive sensor suites are generally infeasible due to their size, complexity, and power demands, rendering them especially impractical for compact form factors like 0.5U satellites. Even after successful detumbling, maintaining precise attitude control remains essential for operations such as sensor alignment, data collection, and solar charging. This capability, provided by an Attitude Determination and Control System (ADCS), is critical to mission success.

The problem is particularly sensitive during the earliest phases of a satellite's mission. A significant number of small satellite failures occur shortly after deployment due to power depletion, failure to detumble, or instability introduced during detumbling. These early-stage vulnerabilities highlight the need for ADCS solutions that are lightweight, low-power, reliable, and simple to integrate. Without such systems, many small satellites risk becoming nonfunctional within minutes of deployment before they can fulfill their mission objectives.

Unfortunately, many existing ADCS offerings for small satellites do not meet these needs. Reaction wheels consume considerable power and introduce mechanical failure modes. Thrusters are generally impractical for small platforms. Even commercially available magnetorquer-only systems are either underperforming or prohibitively expensive. As a result, there is a clear and pressing need for an ADCS architecture that minimizes complexity, maximizes efficiency, and remains cost-effective all without sacrificing core functionality.

The challenge, then, is to design a system that can perform robust detumbling and basic attitude control using minimal hardware and computational resources. Such a solution would not only improve mission reliability but also expand opportunities for scientific and commercial operations on compact satellite platforms.

1.2 The CHARMS Solution

To address the challenge of low-cost, low-power satellite stabilization, the IrishSat Senior Design team developed *Control of Hardware Attitude using Reliable Magnetorquers Satellite (**CHARMS**)*. CHARMS is a magnetorquer-only Attitude Determination and Control System (ADCS) designed specifically for compact platforms as small as 0.5U (10 cm × 10 cm × 5 cm) and capable of interfacing with any standard unit CubeSat. It provides a fully autonomous, plug-and-play solution requiring only power and serial communication with the host satellite bus, with no reliance on GPS, star trackers, or external data sources. The CHARMS system is divided into four key subsystems: the **PCB/Electronics Subsystem**, the **Magnetorquer Subsystem**, the **RTOS/Software Subsystem**, and the **Structural/Thermal Subsystem**.

1.2.1 PCB/Electronics Subsystem

At the core of CHARMS is a custom-designed printed circuit board that integrates all control computation, sensor input, and actuation output. The CHARMS's PCB employs a suite of sensors with outputs which provide the data which aid in determining the satellite's orientation. These include:

- **Dual ICM-20948 Inertial Measurement Units (IMUs):** Provide redundant gyroscopic and acceleration data to support real-time detumbling feedback.
- **Magnetometer**: Measures Earth's local magnetic field to assist the magnetorquer control torque algorithms during detumbling and attitude control.
- **Dual MLX90640 Infrared Cameras:** Orthogonally mounted to perform horizon sensing and sun sensing following image processing. Captures basic images to be relayed. Provides data on the satellite's relative orientation compared to Earth.

    - These sensors will be used in future iterations of CHARMS to do more complex control, including Earth and Sun pointing.

In its current configuration, CHARMS is configured to be a technology demonstration of the most critical step of satellite deployment, detumbling. No other controls are ready for operation at this time.

This reduced sensor suite was chosen to ensure accurate attitude awareness while minimizing volume, power, and complexity. Additional onboard hardware includes:

- **ESP32-S3 MCU:** Executes all real-time software tasks, including control, telemetry packaging, and communication.
- **H-bridge Motor driver ICs:** Regulate bidirectional current flow through the magnetorquers.
- **20-pin bus interface:** Supports 3.3 V, 5 V, ground, and full-duplex UART communication with the NSL flight computer.

All payload functions are handled on this single PCB, which also acts as the physical interface wall between CHARMS and the bus.

1.2.2 Magnetorquer Subsystem

Detumbling and attitude control in satellites can be achieved through various methods, including reaction wheels and thrusters. However, these approaches are generally unsuitable for small satellites due to their significant size, weight, power consumption, and mechanical complexity. Instead, CHARMS relies exclusively on magnetorquers.

> These passive, lightweight, low-power devices are ideal for constrained platforms. By optimizing design choices, these simple solenoids can provide significant control capabilities. Magnetorquers work by generating a magnetic moment that interacts with Earth's magnetic field to produce a reactionary torque ($\boldsymbol{\tau} = \mathbf{m} \times \mathbf{B}$), enabling both detumbling and coarse attitude control.

The CHARMS system employs three custom-built magnetorquers, one per axis, to achieve full three-axis control without exceeding power or volume constraints.

- **X and Y axes:** utilize compact ferromagnetic-core (mu-metal) rods, which produce high dipole moments in small volumes.
- **Z axis:** features a large-diameter air-core magnetorquer wrapped around the CHARMS structure to maximize its magnetic moment.

All magnetorquers are precision-machined in-house, with windings applied using an automated wrapper to ensure uniformity and optimal performance. This design enables precise orientation adjustments for both detumbling and pointing, while remaining well within the satellite's SWaP (Size, Weight, and Power) budget.

1.2.3 RTOS/Software Subsystem

The software architecture for CHARMS's ADCS is designed to achieve precise, efficient, and autonomous control of the satellite's attitude with high system reliability. The system is built around a real-time operating system (FreeRTOS) enabling deterministic task scheduling and modular operation, supporting seamless execution of multiple mission-critical functions.

Key components of the software stack include:

- **Sensor polling and data filtering** for IMUs, magnetometer, and IR cameras via I²C
- **Implementation of the B-dot detumble algorithm** to calculate magnetic moment commands
- **Generation of actuation signals** (PWM and direction) for the magnetorquer drivers
- **Real-time task management** and prioritization of mission-critical processes
- **Telemetry packaging and bus communication** over UART
- **Interpretation of ground commands** and in-flight reconfiguration

1.2.3.1 Controls Operation

The CHARMS detumble software is designed to autonomously stabilize the satellite from its initial spin state using only magnetorquers. The heart of the real-time control architecture system is the **B-dot control algorithm**, which calculates the satellite's angular velocity by measuring the rate of change of the local magnetic field (**dB/dt**) via onboard sensors. Using this information, the algorithm computes the ideal magnetic moment to generate torque and counteract the spin. This desired moment is translated into direction and PWM signals that drive the magnetorquers, thereby reducing angular velocity on all axes.

Sensor data acquisition and filtering form the foundation of the control loop. A dedicated RTOS task polls the onboard IMUs, magnetometer, and infrared cameras every 10 milliseconds. The data is passed through low-pass filters to reduce noise and is continuously updated in memory for use by the control algorithm. All sensor interfacing is performed over I²C and optimized for low power usage and real-time responsiveness.

The CHARMS control system operates in **three defined modes** (**IDLE**, **DETUMBLE**, and **SAFETY**) each triggered via serial commands from the satellite bus. These modes define how sensor data is processed and how actuation signals are applied to the magnetorquers. DETUMBLE mode actively executes the B-dot algorithm while IDLE mode maintains power-saving sensor monitoring without actuation.

1.2.3.2 Firmware Operation

In addition to sensor data handling and control, the RTOS enables deterministic task scheduling and resource allocation. It ensures that high-priority tasks (such as magnetorquer actuation and error handling) are executed promptly, while maintaining seamless execution of lower-priority tasks like data logging and telemetry.

CHARMS is designed to interface seamlessly with **NearSpace Launch's proprietary communications protocol** over a full-duplex UART connection with the satellite bus. The firmware supports this interface by:

- **Constructing data packets** in a format recognizable and parsable to the NSL bus processor.
- **Sending requests** to the bus when information like health and safety information, battery levels, or data downlinks through Iridium comms networks is needed.
- **Receiving and interpreting responses** from the bus and extracting useful information from the packets sent to CHARMS over the serial connection, including receiving ground commands to change the configuration or parameters of the system.

All packet formatting, transmission, and parsing are performed autonomously, ensuring robust and reliable communication throughout the mission lifecycle. This interface is essential to the payload's ability to operate and adapt in orbit.

Execution on the **ESP32-S3 microcontroller** allows the firmware to leverage its built-in hardware features, including I²C and UART peripherals, dual-core processing, and PWM outputs, to support real-time, low-latency operation. This tight integration between hardware and software results in a self-contained, autonomous ADCS capable of performing all essential detumbling, telemetry, and configuration tasks in orbit without dependence on external processors or manual intervention.

1.2.4 Structures and Thermal Subsystem

The CHARMS payload must be structurally robust to withstand launch loads and maintain secure mechanical integration with the NSL bus throughout the mission. Mechanical design considerations ensure compatibility with the 0.5U ThinSat frame and compliance with interface standards specified by NearSpace Launch. Thermal reliability is equally critical; the internal environment must remain within the operating temperature ranges of all electronic components. This is achieved through passive thermal design and strategic component placement to minimize thermal hotspots.

Structural integration and detailed thermal management are primarily handled by the broader IrishSat team, with the Senior Design team's contributions focused on ensuring mechanical compatibility and thermal awareness within the electrical design.

1.3 Expectations and Results

At the outset of the CHARMS project, the team defined a clear set of expectations encompassing system performance, learning outcomes, and compliance with industry standards. These expectations informed every phase of development, from early architecture decisions to subsystem tradeoffs and final integration. Our primary objective was to deliver a low-power, magnetorquer-only ADCS capable of autonomous detumbling on a 0.5U satellite platform that could operate independently, within a strict power budget, and remain compatible with the NearSpace Launch (NSL) ThinSat bus. While not every target was perfectly met, the system

performed exceptionally in nearly all categories, achieving core functionality and demonstrating both technical viability and engineering excellence.

1.3.1 System Performance Objectives and Outcomes

CHARMS was evaluated against a set of high-level operational criteria:

Objective	Target Specification	CHARMS Performance	Status
Detumbling Success	Reduce spin to < 0.5°/s	~2°/s achieved after 15°/s initial	Partial Success
Detumbling Time	< 75 minutes	17 minutes minimum	Exceeded
Precise Actuation	Produce expected B-field from magnetorquers	~4000 magnitude achieved	Met
Magnetorquers Functionality	All three magnetorquers operate	All 3 verified and functional	Met
Sensor Suite Performance	Accurate magnetometer and gyro readings	All axes accurately read, decreasing ω detected	Met
Reduced Sensor Suite	Use minimal sensors for functionality	Magnetometer & IMU only	Met
Image Processing	IR cameras perform sun/horizon tracking	Horizon images successfully captured	Met
Dipole Strength (MuMetal)	> 5 Am²	~1.06 Am²	Partial Success
Dipole Strength (Air Core)	> 0.1 Am²	~1.67 Am²	Exceeded
Power Consumption	< 7 Wh	~0.2087 Wh during detumble	Exceeded
PWM Control	PWM scaled based on sensor feedback	Gain-varying PWM confirmed	Met
UART Communication	Interface with NSL serial packet structure	Verified with emulator	Met

Telemetry and Commanding	Send/receive data and commands over UART	Confirmed autonomous packet exchange	Met
Autonomous Operation	Self-contained RTOS + control architecture	Fully autonomous detumble mode	Met
Cost Reduction	Significantly lower than COTS solutions	~$550 total, ~1/40 of comparable COTS systems	Exceeded
Power Efficiency	Operates under tight energy budget	~0.2087 Wh detumble (<< 7 Wh)	Exceeded
Software Robustness	Real-time scheduling, modular RTOS	FreeRTOS running multi-mode architecture	Met
System Independence	Minimal reliance on bus for control	Operates fully on onboard ESP32-S3	Met

CHARMS met or exceeded nearly all performance objectives, especially in power efficiency, hardware integration, and autonomous control.

1.3.1.1 Physical Testing and Experimental Validation

Testing was conducted using a custom-built Helmholtz cage to simulate magnetic field environments. For isolated magnetorquer validation, the magnetic center was zeroed and field strength measured across multiple axis-specific configurations. To evaluate full-system performance, PySol was used to recreate low-Earth orbit (LEO) magnetic field conditions within the cage.

The CHARMS payload was mounted atop a hemispherical bowling ball resting on an air bearing, enabling low-friction, multi-axis rotational movement. The system was externally powered via 5V and 3.3V rails, mass-matched to reflect the full NSL bus, and manually spun prior to operation. Telemetry and health data were transmitted via Bluetooth throughout each test.

A range of trials were conducted with varying gains, timing intervals, and control configurations. Notably, in a control-disabled trial, the system took over 90 minutes to decelerate from 20°/s to an asymptotic ~7°/s, highlighting the natural damping baseline of the testbed.

The plot included in **Section 4.1.2.5.1 Successful Autonomous Detumble in Simulated Orbit** in **Figure 44** compares angular velocity reduction for two test conditions—active detumbling (blue) and idle (red)—both beginning at 15°/s. Under active control, CHARMS reduced spin to ~2°/s in under 18 minutes, while the idle condition plateaued around 7°/s after more than 90 minutes. The testbed itself induces some passive damping, but CHARMS's control system clearly accelerates the detumble process.

While the target spin rate of <0.5°/s was not reached, the experiment validated functional detumbling and energy efficiency (~0.2087 Wh). The shortfall may be due to sensor noise from the low-cost IMU, residual torques from the air bearing (which, though low-friction, is not torque-neutral), limitations in the current proportional control implementation, or other constraints inherent to Earth-based simulation. Still, the results substantiate CHARMS's capability to deliver real-time detumbling under aggressive SWaP limitations.

1.3.1.2 Learning and Development Goals

CHARMS was conceived as both a technical demonstrator and a learning platform. The team worked across disciplines to design a novel system under real-world aerospace standards, achieving the following development goals:

- **Designing a complete, low-power ADCS**
  → Implemented a fully functional, magnetorquer-only detumble system operating at ~0.2087 Wh.

- **Developing a space-capable, single-board PCB**
  → Created, assembled, and tested an integrated flight computer optimized for thermal and spatial efficiency.

- **Creating robust autonomous software**
  → Built a FreeRTOS-based architecture supporting real-time task execution, telemetry handling, and motor control.

- **Collaborating with a commercial launch provider**
  → Worked directly with NSL, interfacing with their bus protocol and following their integration pipeline.

- **Working interdisciplinarily across subsystems**
  → Integrated electrical, software, mechanical, and systems components into a cohesive, flight-ready payload.

- **Meeting stringent aerospace requirements**
  → Followed NSL's ICD guidelines for power, firmware, mechanical, and thermal standards.

- **Designing and fabricating novel magnetorquers**
  → Produced in-house magnetorquers using automated winding tools, surpassing COTS alternatives.

- **Delivering within a fixed budget**
  → Final cost was ~$550—roughly 1/40th the price of commercial equivalents.

- **Enabling future extensibility**
  → Built a modular platform expandable for full attitude pointing using onboard IR sensors and vision algorithms.

Each goal was achieved or exceeded. The project provided the team with experience in aerospace design, embedded systems, and spaceflight hardware development—preparing them for future technical leadership in the space sector.

1.3.1.3 Testing and Compliance

CHARMS addressed all NSL-defined requirements through a structured testing and review pipeline:

- Passed Integration Readiness Review (IRR), including power, continuity, and communication checks
- Conducted mode-based functionality tests with emulator integration
- Verified power resilience (0V startup, watchdog timers, power cycling)
- Developed an environmental testing plan aligned with GEVS guidelines (TVAC, bakeout, vibration); post-environmental tests are in progress

While CHARMS did not meet its most ambitious spin-rate target, it unequivocally succeeded in its core mission: delivering a fully autonomous, low-power, magnetorquer-only ADCS for small satellites. The system met rigorous performance standards and integration constraints, while offering a scalable, cost-effective alternative to commercial systems. Perhaps most significantly, it equipped the team with firsthand experience in end-to-end aerospace system design—balancing innovation, precision, and practicality to deliver a flight-ready solution.

2 System Requirements

This project is in collaboration with a commercial partner, NearSpace Launch (NSL), whose education branch, NearSpace Education (NSE), sponsors the CHARMS project. The other half of the thinsat, the satellite bus, is built by NSL. Because the payload is one half of an integrated satellite system CHARMS had to meet both internally defined engineering goals and externally imposed integration, interface, and qualification requirements outlined by NSL. This section details the requirements framework that guided CHARMS's development, spanning performance expectations, risk mitigation, and compliance with launch provider standards.

2.1 Internal Requirements

The internal system requirements for CHARMS were derived from mission objectives, subsystem design constraints, and innovation goals. These requirements shaped system architecture, drove design tradeoffs, and informed testing protocols.

2.1.1 Required Features and Performance Goals

CHARMS's mission was structured around three core objectives: data generation, team learning, and technical innovation. These drive many of the flow-down requirements of the full system and subsystems. The high-level objectives for the team are:

Data Objectives

- Demonstrate torque authority and control fidelity using onboard gyroscopes, accelerometers, and magnetometers.
- Construct and transmit operational telemetry packets from LEO using the Iridium communications network.

Learning Goals

- Design and validate a low-power ADCS.
- Explore optimal material and geometry choices for magnetorquer design.
- Implement a robust, autonomous software architecture for real-time control.

Innovation Goals

- Develop a custom, in-house magnetorquer architecture using novel form factors and materials.
- Deliver an ultra-low-power, low-cost attitude control system operable within strict SWaP constraints.

High-level features that realize these objectives of the CHARMS mission are as follows:

1. **CHARMS's ability to "detumble"** or reduce angular velocity along all three axes, autonomously slowing rotation to below 0.5°/s.

2. **Precise actuation of the custom-built magnetorquers** to produce expected magnetic dipole moments and achieve commanded B-fields using **less than 7 Wh** of energy.
3. **A real-time control algorithm (B-dot) that calculates and outputs** direction and PWM signals based on filtered sensor input, enabling adaptive control through detumble and pointing.
4. **Interface with NSL's novel packet structure and serial communication protocol** to request data from the bus and send information to Iridium comms (this is done through an emulator given to us by NSL).
5. **Sensor data acquisition and filtering pipeline** that collects IMU, magnetometer, and IR sensor data, enabling both B-dot control and horizon/sun tracking through image processing.
6. **Embedded software capable of autonomous operation across multiple modes** (IDLE, DETUMBLE, SAFETY), each managing power consumption and control behavior accordingly.

CHARMS has performance goals relating to particular features. Performance goals, set at the beginning of the CHARMS program, are as follows:

Table 1. Measurement Performance Goals, Tiered

Precision for Measurements	**Base Tier**	**Middle Tier**	**Stretch Tier**
Mu-metal Dipole Moment	5 Am^2	10 Am^2	20 Am^2
Air Core Dipole Moment	0.1 Am^2	0.2 Am^2	0.3 Am^2

Table 2. Detumble Performance Goals, Tiered

Operational Metrics	**Base Tier**	**Middle Tier**	**Stretch Tier**
Detumble Power Consumption	5 Wh	2 Wh	1 Wh
Detumble Time (mins)	1 hr 15 mins	45 mins	30 mins

Final Angular Velocity	+/- 2 deg/s	+/- 1 deg/s	+/- 0.5 deg/s

Reference **Section 4.1.2 Functional Testing** to find deeper analysis of CHARMS's empirical performance compared to these goals.

2.1.2 Risk Mitigation Requirements

In order to be proactive in the design phase of the system, the CHARMS team performed a comprehensive risk analysis and developed different mitigation measures to assist the mission in meeting its requirements and mission objectives.

Table 3. CHARMS Risk Analysis and Corresponding Design Requirements

Risk	Potential Effect	Design Requirement
MEMS Sensor Helium Effect	Sensor performance drift or failure due to helium permeation	Use helium-resistant packaging options or select alternative sensors less susceptible to helium exposure. Test and monitor sensor performance during environmental testing.
Sensor anomaly due to radiation	Each sensor could be a single point of failure	Implement ground commands allowing the payload to switch between redundant sensors to continue operation.
Structural Failure	Physical damage to the board, connections, or magnetorquers during launch or operation	Conduct extensive vibration testing to aerospace standards. Use robust design and secure mounting techniques to avoid physical breakage.
Circuitry Incorrectness	Non-functional or suboptimal performance limiting mission success	Validate circuits through breadboarding, logic analyzers, and in-depth testing with test pads before final assembly to ensure performance.
Power Overconsumption	Exceeding the power budget leading to mission failure	Perform detailed power budgeting, implement low-power modes, power cycling, and optimize component usage to stay within power constraints.

Sensor Data Inaccuracy	Inaccurate sensor readings affecting ADCS performance	Apply sensor calibration before flight, and implement filtering algorithms (e.g., low-pass filters) to improve sensor data in real-time.
Software Bugs	Unexpected software behavior leading to mission degradation	Thorough testing of software in simulation and HIL environments.
Communication Failure	Loss of link between payload and bus	Periodic communication checks and self-reset protocols to restore the communications connection.

2.2 External NearSpace Launch Requirements

CHARMS is hosted aboard and must interface with NSL's "0.5U" ThinSat platform, it must comply with interface, integration, and test requirements provided by NSL through their Interface Control Document (ICD) [1]. A significant amount of system and subsystem requirements span mechanical compatibility, electrical specifications, firmware standards, and environmental qualification. to ensure that the satellite as a whole remains operational through all phases of the mission. They are as follows:

2.2.1 Mechanical Requirements

Table 4. Mechanical/Workmanship Requirements from NSL ICD [1]

ID	Payload Requirement	Verification Criteria	Verification Method			
			I	A	D	T
R1.1	Payload shall be clean and free from dust, contaminants, deposits, and any material which could impact outgassing or operation	Visually inspect payload with magnifying glass or microscope and blacklight	X			
R1.2	Payload shall be composed solely of materials which have a TML of less than 1% and CVCM of less than 0.1% per the NASA outgassing database or other reputable source, with deviations agreed to by NSL	Inspect BOM for nonconformance	X			
R1.3	Payload shall contain no prohibited materials: cadmium parts or cadmium plating; titanium; zinc plating; mercury or compounds containing mercury; pure tin or tin electroplate (except when alloyed with lead, antimony, or bismuth); silicone rubber or RTV silicones; magnets; other materials as identified by NSL	Inspect BOM for prohibited materials	X			
R1.4	Payload shall contain no pyrotechnics, propulsion, radio frequency transmitters or receivers, externally-facing cameras, or anything else that requires explicit approval from FCC, ITU, NOAA, FAA, or other governing agency, except where agreed to by NSL and included in appropriate licensing application documents	Inspect system block diagram, BOM, and delivered Payload for prohibited systems	X			
R1.5	Payload shall contain no energy storage devices including pressure vessels, batteries, springs, etc., except where approved by NSL	Inspect system block diagram, BOM, and delivered Payload	X			
R1.6	Payload shall be designed such that no space debris, intentional or unintentional, are generated during launch, separation, commissioning, or operation, and to survive high-vibration launch environment	Inspect Payload for weak or mechanical connections, thin structural members, etc. Optional: perform vibe test	X			
R1.7	Payload shall be designed to operate in temperatures ranging from -30 to +60 Celsius and survive temperatures ranging from -40 to +80 Celsius	Analyze manufacturer specs for materials and parts Optional: perform TVAC test		X		
R1.8	Payload shall comply with the volumetric constraints presented in Section 3	Inspect outside w/ calipers Demonstrate fit check into ThinSat (top and bottom)	X		X	
R1.9	Payload shall be rigidly mounted to the four primary connections points identified in Section 3 via #2-56 stainless steel machine screws with thread depth into the Bus of no less than 3 mm and no greater than 5 mm	Inspect thread depth with calipers Demonstrate bolting into ThinSat with clean alignment	X		X	
R1.10	Payload shall have primary electrical connector placed as specified in Section 3.2	Demonstrate fit check with ThinSat Visually inspect pins during insertion for deflecting or misalignment	X		X	
R1.11	Payload may not exceed 250g	Analyze the payload CAD model and verify. Demonstrate that the Payload is 250g or less.		X	X	

2.2.2 Electrical Requirements

Table 5. Electrical Requirements from NSL ICD [1]

ID	Payload Requirement	Verification Criteria	Verification Method			
			I	A	D	T
R2.1	Payload shall be powered solely by the Bus 3.3v, 5v, and 6-9v BUSS+ rails	Demonstrate near-zero potential with voltmeter on various Payload PCB test points with Bus powered but all Payload switches off			X	
R2.2	Payload shall be tolerant to +-5% deviations from nominal voltage on both the 3.3v and 5v rails, and the full 6v to 9v range on the BUSS+ rail	Analyze BOM components for sensitivity to input voltage Demonstrate payload operation at min (3.1v, 4.7v, 6v), max (3.5v, 5.3v, 9v), and any other specifically concerning combinations based on intended Payload design and operation		X	X	
R2.3	Payload shall draw inrush current of no greater than 5 amps per rail for no longer than 50ms	Test inrush current for each rail and switch by measuring supply voltage and current through a shunt resistor with two oscilloscope channels triggered by the voltage increase and sampling for at least 100ms				X
R2.4	Payload shall draw no more than 1 amp peak on the 3.3v rail (across both switches), after inrush	Demonstrate the current draw from a power supply in the highest-draw state is less than 1 amp			X	
R2.5	Payload shall draw no more than 1 amp peak on the 5v rail (across both switches), after inrush	Demonstrate the current draw from a power supply in the highest-draw state is less than 1 amp			X	
R2.6	Payload shall draw no more than 2 amps peak on the 6-9v BUSS+ rail, after inrush	Demonstrate the current draw from a power supply in the highest-draw state is less than 2 amps			X	
R2.7	Payload shall include clearly labeled electrical test points for all nets of the Primary Connector used by the Payload that can be accessed from the top or sides after integration with the ThinSat base	Inspect labels for readability Demonstrate probe access to each test point with the payload installed in the ThinSat base	X		X	
R2.8	Payload shall comply with the Primary Connector pinouts specified in Section 4.1	Inspect schematic Demonstrate electrical continuity with a multimeter between each connector pin and the labeled test point	X		X	
R2.9	Payload shall be designed such that the voltage from all Payload outputs (analog, RTS, and TX pins) cannot exceed the voltage of the 5v rail, ignoring minor transient effects	Inspect schematic for nonconformance Analyze using simulation tools if basic inspection cannot deterministically provide verification	X	X		
R2.10	Payload shall have primary processor be powered solely by Switch 1 (3.3v and 5v)	Demonstrate Payload can turn on Switch 2/3/4 (if used) via Bus request when starting with just Switch 1 on			X	
R2.11	Payload shall be tolerant to abrupt power loss on all rails at any time	Analyze worst-case effect based on known code procedures, mutex locks, triplication, etc. Demonstrate abrupt removal of power from all rails at arbitrary time and successful power-up afterwards with no adverse effects		X	X	
R2.12	Payload shall be designed to draw no more than 0.5 watts (nadir) or 3 watts (sun-pointing) OAP	Analyze OAP of all operational modes based on measured power draw, pointing configuration, and duty cycling		X		
R2.13	Payload shall not produce EMI or RF radiation at levels that interfere with bus operations	Analyze possible noise sources and verify through similarity and/or discussions with NSL Demonstrate nominal Bus operations with Payload powered during day-in-the-life test Optional, or if required by NSL: perform RE102		X	X	

2.2.3 Firmware/Protocol Requirements

Table 6. Firmware/Protocol Requirements from NSL ICD [1]

ID	Payload Requirement	Verification Criteria	Verification Method			
			I	A	D	T
R3.1	Payload shall be able to send downlinks, fetch uplinks from buffer, and request self-reboots	Demonstrate successful downlink, uplink, and self-reboot commands to the Bus			X	
R3.2	Payload shall be designed to have useful and relevant data even in limited throughput scenarios of less than ten 200-byte packets daily	Test Payload against the NSL emulator for a 24-hour continuous DITL with the emulator in limited-throughput mode, and document usefulness/relevance of collected data				X

2.2.4 Testing Requirements

In addition to payload specifications and requirements given to us from NSL, they have also given us testing requirements with which to create a testing plan for the finished payload before integration. They are given below:

Table 7. Testing Requirements from NSL ICD [1]

ID	Payload Requirement	Verification Criteria	Verification Method			
			I	A	D	T
R4.1	Payload shall pass pre-environmental functional testing	Test payload functionality in all operational modes and document thoroughly				X
R4.2	Payload shall undergo thermal vacuum bakeout at no less than 60° Celcius and no greater than 1E-4 Torr, for no less than 6 hours	Test and document temperature and pressure throughout full test				X
R4.3	Payload shall pass post-environmental functional testing	Test using same as R4.1				X

Beyond the required testing, NSL also had recommended testing that the teams should pursue, but is not necessary [1]:

- Perform random vibration testing at NASA GEVS Qualification levels (14.1 GRMS) on an EM unit or Acceptance levels (10.0 GRMS) on the FM unit.
- Perform TVAC cycling between -30 and +60 °C or beyond; 1 hour dwell at each extreme; at least 4 hot/cold cycles.
- Perform MIL-STD-461 RE102 radiated emissions test.

From these requirements, the CHARMS team was able to construct a comprehensive test plan for the payload:

Pre-environmental testing:

- Test all operational modes of the satellite
    - Idle, Detumble, and Safety Modes
- Interface with the **emulator** and test data interface with the bus in all modes

TVAC testing:

- Use the **UND Nanofabrication TVAC system** and run the payload through TVAC cycling meeting ICD testing requirements
    - 60 °C, 1E-4 Torr, 6 hours (Bake out)

Vibration testing:

- Use NASA GEVS Qual levels (14.1 GRMS) on EM units and Acceptance levels (10.0 GRMS) on FM units
- Utilize the NearSpace Launch vibration table or the IrishSat vibration tabl**e** for testing.

Post-environmental testing:

- After all environmental testing, make sure the system still functions. Repeat pre-environmental testing.

This test plan was found to be sufficient, covering all testing requirements, in a Critical Design Review and Integration Readiness Review with NSL.

3 Detailed Project Description

3.1 System Theory of Operation

CHARMS functions as an autonomous detumbling system for satellite stabilization, built around a closed-loop sensor-to-actuation feedback architecture. It operates by continuously collecting sensor data, performing fusion and filtering, executing control algorithms, and driving magnetic actuation accordingly. At the core of the system is a real-time operating system (RTOS), which manages deterministic task scheduling and ensures reliable execution of time-critical operations.

The system can be conceptually divided into three core functional domains: **input**, **processing**, and **output**. The **input** domain consists of the sensor suite, which includes IMUs, a magnetometer, and infrared cameras; these sensors collect real-time environmental data such as angular velocity, acceleration, and Earth's magnetic field. The **processing** domain is managed by the onboard microcontroller, which executes sensor fusion, applies calibration, and runs the B-dot control algorithm to determine required magnetic actuation. The **output** domain includes the custom magnetorquers, which produce controlled magnetic dipole moments to generate torque via interaction with Earth's magnetic field.

CHARMS interfaces with its host satellite exclusively through a 20-pin connection to the NearSpace Launch (NSL) ThinSat bus. This connection provides regulated 5 V and 3.3 V power lines as well as UART-based full-duplex serial communication. All mission telemetry, ground commands, and power are handled through this single bus interface. The bus itself is responsible for Iridium communication and power regulation for the entire spacecraft, while CHARMS remains a self-contained, plug-and-play payload module responsible solely for attitude control.

3.2 System Block Diagram

As illustrated in **Figure 1**, the CHARMS architecture is divided into three core physical subsystems: the **Sensor Suite** (input), the **Command and Data Handling (C&DH) PCB** (processing), and the **Magnetorquer System** (output). These subsystems work together in a closed-loop control configuration to achieve autonomous detumbling and support future attitude pointing functionalities.

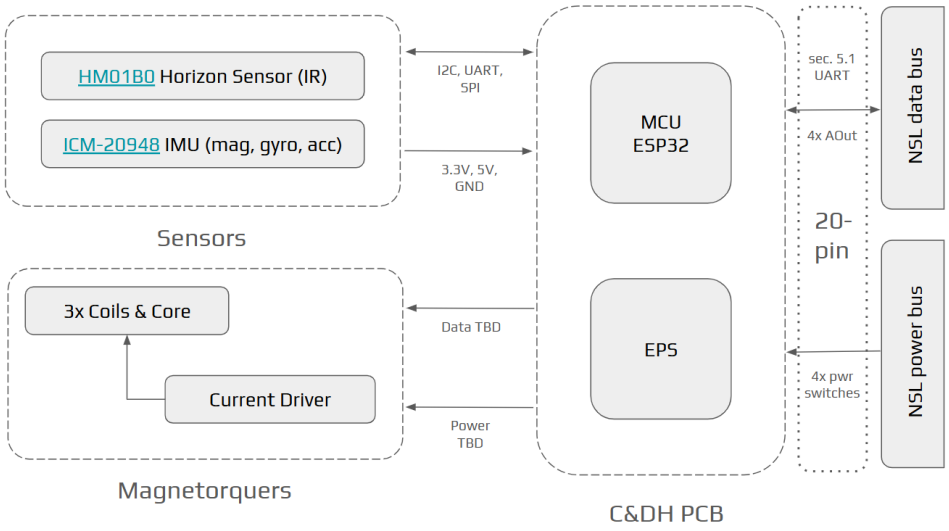

Figure 1. Overall System Block Diagram

- **Sensor Suite:** includes the ICM-20948 Inertial Measurement Units (IMUs) and HM01B0 infrared horizon sensors. The IMUs provide critical gyroscopic, accelerometer, and magnetometer data used in detumbling. The IR sensors, while not used in the current detumble routine, are integrated for future horizon detection and Earth imaging tasks. All sensors communicate with the microcontroller over I²C and are powered by the regulated 3.3 V and 5 V lines from the power subsystem.
- **C&DH PCB:** hosts the **ESP32-S3 microcontroller** and power distribution circuitry (**EPS**). The microcontroller executes the RTOS-based control software, including the B-dot algorithm, sensor polling, telemetry generation, and packet communication with the NSL bus. The EPS handles regulated power delivery to all onboard components and includes features such as inrush current protection and debugging support.
- **Magnetorquer System:** consists of three custom-wound torquers (two with mu-metal cores and one air-core), driven by dedicated current driver ICs. These are controlled via PWM and GPIO signals from the microcontroller, and their power is sourced and gated via the EPS.

The CHARMS system connects to the NSL ThinSat via a **20-pin interface**, which provides:

1. Dual-voltage power lines (3.3 V and 5 V)
2. Full-duplex UART for serial communication with the NSL bus
3. Four analog outputs (currently unused)
4. Four discrete power control lines (used for power gating and debugging)

This streamlined interface design supports CHARMS's **plug-and-play integration philosophy**, minimizing bus-side dependencies while maximizing modularity, electrical isolation, and ease of system replacement.

3.3 Detailed Design and Operation of Subsystem 1: PCB

The CHARMS Printed Circuit Board (PCB) is the foundational platform upon which all subsystem functionality is built. It integrates sensing, computation, power regulation, magnetorquer actuation, and communication into a single, compact form factor that also serves as the structural and electrical interface to the NSL bus. Every subsystem—sensor input, control processing, motor driving, telemetry generation, and bus interfacing—relies on the reliable performance of this board, making its design and validation one of the most critical aspects of the CHARMS development process.

To ensure maximum functionality, correctness, and fault tolerance, the team went through four full hardware design iterations, each fabricated, assembled, and tested. Each version incorporated lessons learned from prior tests—ranging from schematic-level signal corrections and sensor routing adjustments to major architectural changes like inrush current limiting, voltage regulation refinements, and modular debugging enhancements. The final iteration represents a highly optimized system that satisfies all power, size, and communication constraints imposed by the mission and the NSL Interface Control Document (ICD).

The board supports dual-voltage operation (3.3 V and 5 V), houses all sensors required for detumbling, provides PWM-based current control for three custom magnetorquers, and includes UART communication lines for full integration with the NSL data bus. Mechanically, it also serves as the inner dividing wall between the CHARMS payload and the satellite bus, with precisely positioned mounting holes and edge clearances that support plug-and-play operation while maintaining electrical isolation.

The following subsections break down the detailed design, functionality, and engineering rationale for each major component of the PCB.

3.3.1 MCU

At the heart of the CHARMS system is the **ESP32-S3-WROOM-1**, a dual-core microcontroller that balances high processing performance with low power consumption. The ESP32-S3 supports both **I²C and UART** interfaces, enabling seamless communication with CHARMS's sensor suite and the NSL satellite bus. It also includes integrated **Wi-Fi and Bluetooth Low Energy (BLE)** capabilities, which the team leveraged during development for wireless debugging and data logging. Programming is performed over **USB 2.0**, which was selected for its simplicity and sufficient bandwidth, as throughput is not a bottleneck for CHARMS's operations.

The microcontroller was selected after evaluating a wide range of candidates, with power efficiency, computational performance, and radiation resilience being key criteria. While the ESP32-S3 is not radiation-hardened, its cost-effectiveness, development simplicity, and strong ecosystem made it the most suitable COTS solution for CHARMS's rapid development and educational mission.

A radiation-hardened alternative, the SAM3X8ERT from Microchip, was considered. However, the cost, complexity, and development time associated with its integration would have compromised the CHARMS system's goals of being low-cost, accessible, and modular. Since

CHARMS's mission lifetime is relatively short and will take place in Low Earth Orbit (LEO)—a radiation environment that is less severe than deep space—the team opted to accept this tradeoff in favor of rapid prototyping and deployment. To mitigate the ESP32-S3's susceptibility to radiation-induced faults, CHARMS employs several design safeguards including fault-tolerant software and hardware redundancy. This design choice also mirrors the broader hardware philosophy of NearSpace Launch, which frequently deploys COTS components to reduce cost and broaden access to space experimentation.

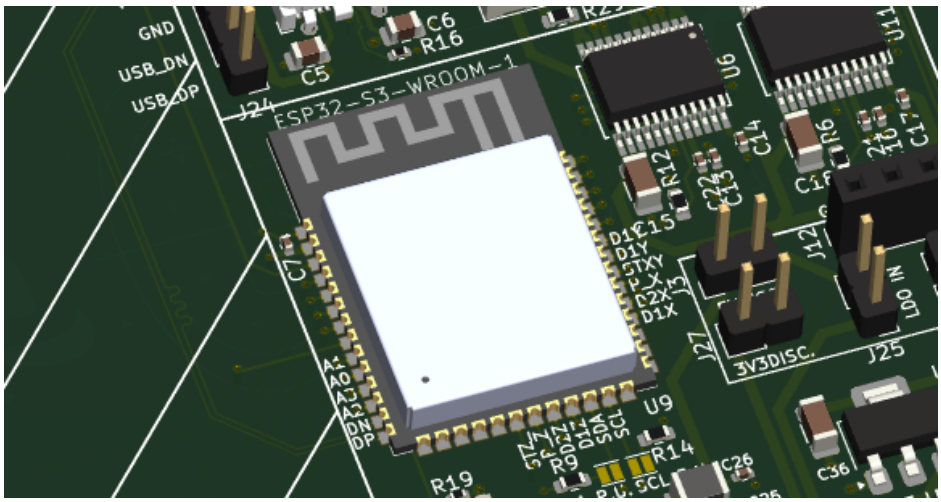

Figure 2. ESP32-S3-WROOM-1 placed on CHARMS PCB

3.3.2 Sensors

3.3.2.1 Magnetometer

CHARMS uses a dedicated LIS2MDL 3-axis magnetometer as an independent reference for Earth's magnetic field. This external sensor can supplement IMU readings and provides more flexibility over physical mounting location (important since CHARMS's magnetorquers output high magnitude fields). The LIS2MDL was chosen for its low power consumption, high availability, and qualification in reading strong and precise field measurements.

The magnetometer is critical to the operation of the satellite and needs to provide accurate readings, so a calibration process is required to ensure proper operation. CHARMS applies hard iron correction (which subtracts offsets based on experimentally collected fields) and soft iron correction (a 3x3 transformation matrix which offsets field shape distortions). These calculations are computed using a FOSS tool called MotionCal created by PJRC. MotionCal was meant to be used with a specific PJRC motion sensor PCBA, so serial data must be logged in a specific manner to ensure proper calibration. Rotating the ICM and printing data in a specific format (iterated in the tool's Github) creates a 3D map of the distortions present in the sensor's readings and maps these data points to a perfect sphere. **Figure 3,** below, shows an example image of a sensor's hard iron (magnetic offset) and soft iron (magnetic mapping).

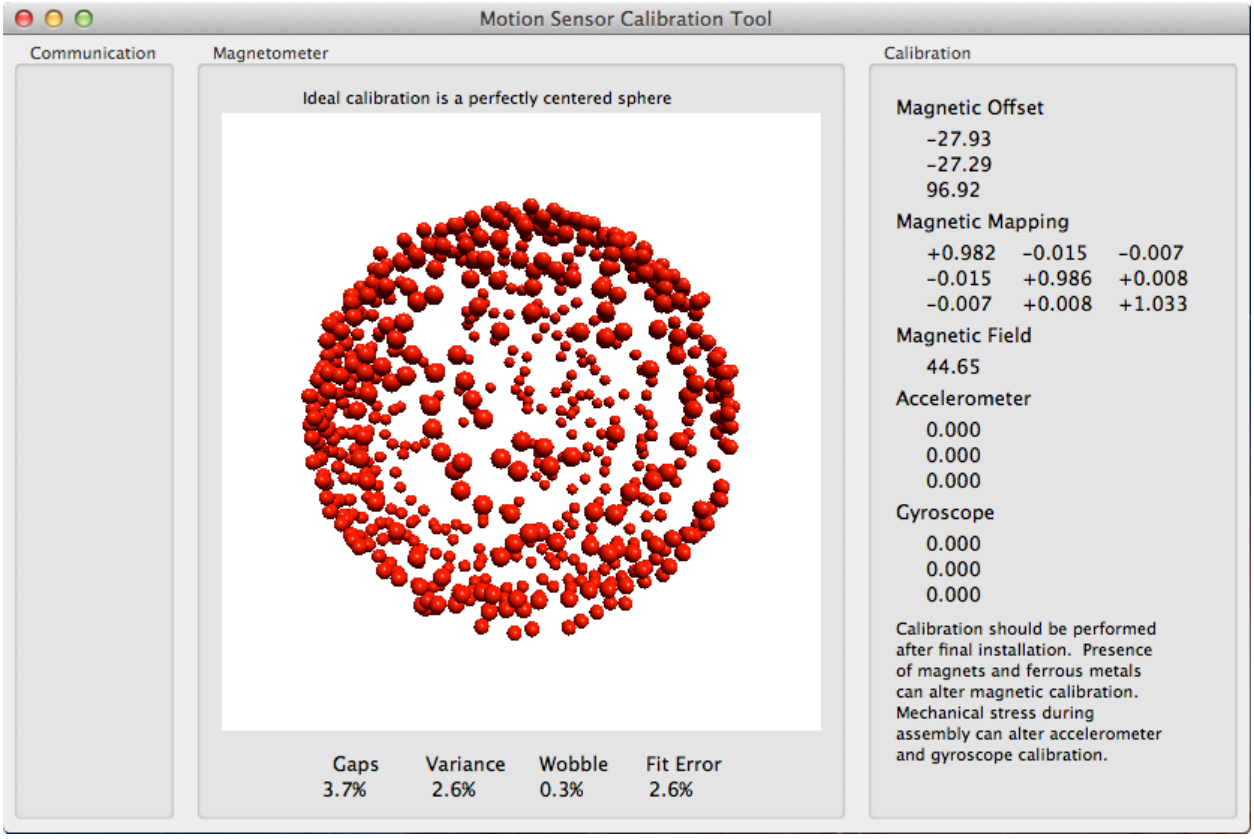

Figure 3. MotionCal Calibration Tool

Once the hard and soft iron offsets are calculated, they can be stored in the software as indicated:

```
float lisHardIron[3] = {6.47, 5.87, 20.30};
float lisSoftIron[3][3] =  {     {1.021, -0.013, -0.041},
                                 {-0.013, 0.958, -0.011},
                                       {-0.041, 0.07, 1.024} }};
```

Once calibrated values are determined, the magnetometer data is used in CHARMS's detumbling algorithm, where the magnetic field (B) and then the cross product ($m = -k\ (\omega \times B)$) is calculated.

The magnetometer wiring schematic is straightforward as depicted in its datasheet. The LIS2MDL supports both I2C and SPI communication, but I2C was selected since communication speed is not critical. The sensor is paired with a wakeup signal and provided with 3.3V power and decoupling capacitors to ensure clean input. **Figure 4** shows the wiring schematic for the magnetometer.

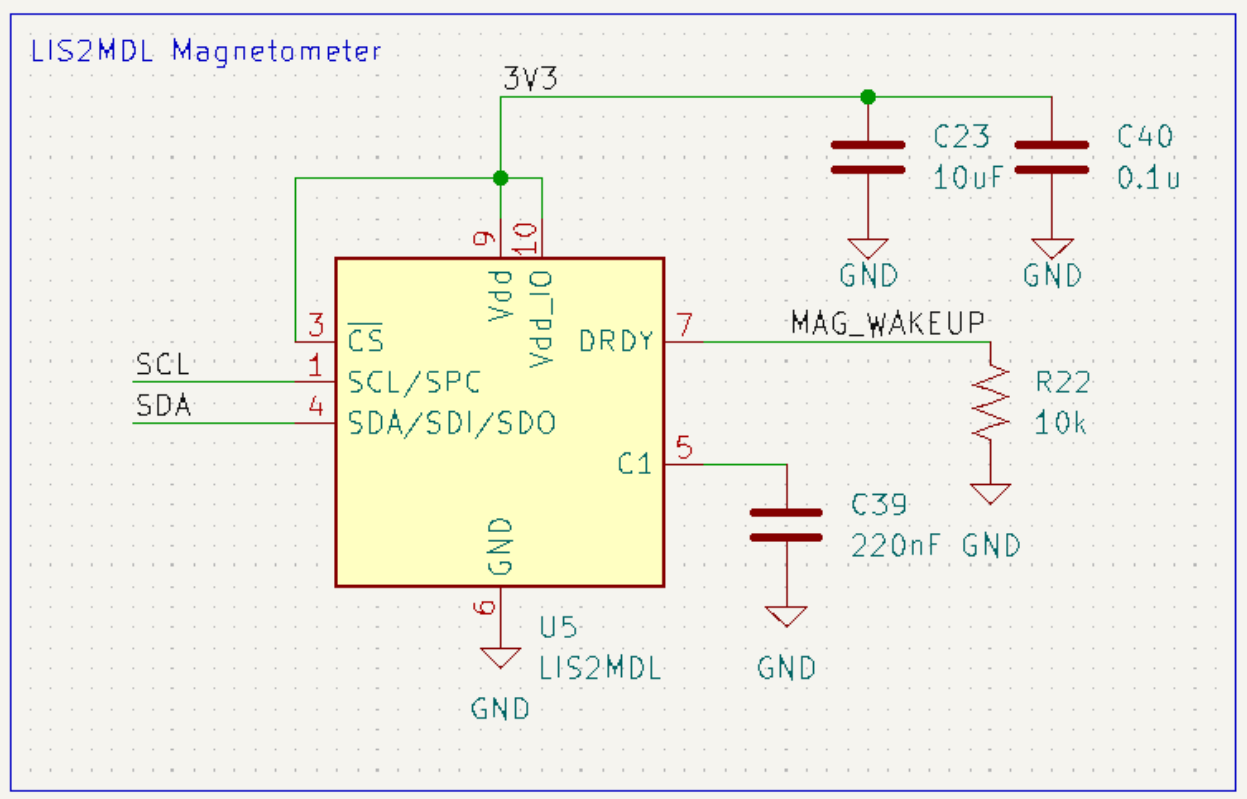

Figure 4. LIS2MDL Magnetometer Schematic

3.3.2.2 IMUs

CHARMS is equipped with a dual-IMU setup featuring two ICM-20948s from TDK. These dual sensors ensure that CHARMS can implement motion tracking even if one IMU is damaged in flight. These IMUs also have built-in magnetometers, meaning that the critical magnetic field sensing is triple-redundant (along with the external dedicated LIS2MDL magnetometer).

The ICM-20948 has a three-axis gyroscope and a three-axis accelerometer that interface over I2C to the ESP32-S3 and provide the main angular velocity required for the B-dot detumble algorithm. A calibration process similar to the magnetometer is used for accelerometer and gyro data to ensure accurate readings. This calibration process, performed for both IMUs, is applied at runtime before any control logic, ensuring calibrated vectors are used for attitude determination.

```
float icmOffsets[3][2] = {      {-0.503, -0.119, 0.096},        // Gyro X,Y,Z
                                {22.146, 7.236, 22.211}};       // Accel X,Y,Z
```

Once calibrated values are determined, the IMU data is used in CHARMS's detumbling algorithm, where the angular velocity (ω) and then the cross product ($m = -k\ (\omega \times B)$) is calculated.

The ICM-20948s are wired as indicated in their datasheets and in **Figure 5**, with decoupling capacitors placed at all power inputs. The ICM-20948 requires 1.8V input power (contributing to its ultra-low power consumption), so CHARMS employs a 3.3V-1.8V LDO onboard to provide

power to these critical components (shown in **Figure 6**). Transistor-based level shifters are also used onboard to ensure that data is sent at the proper 1.8V.

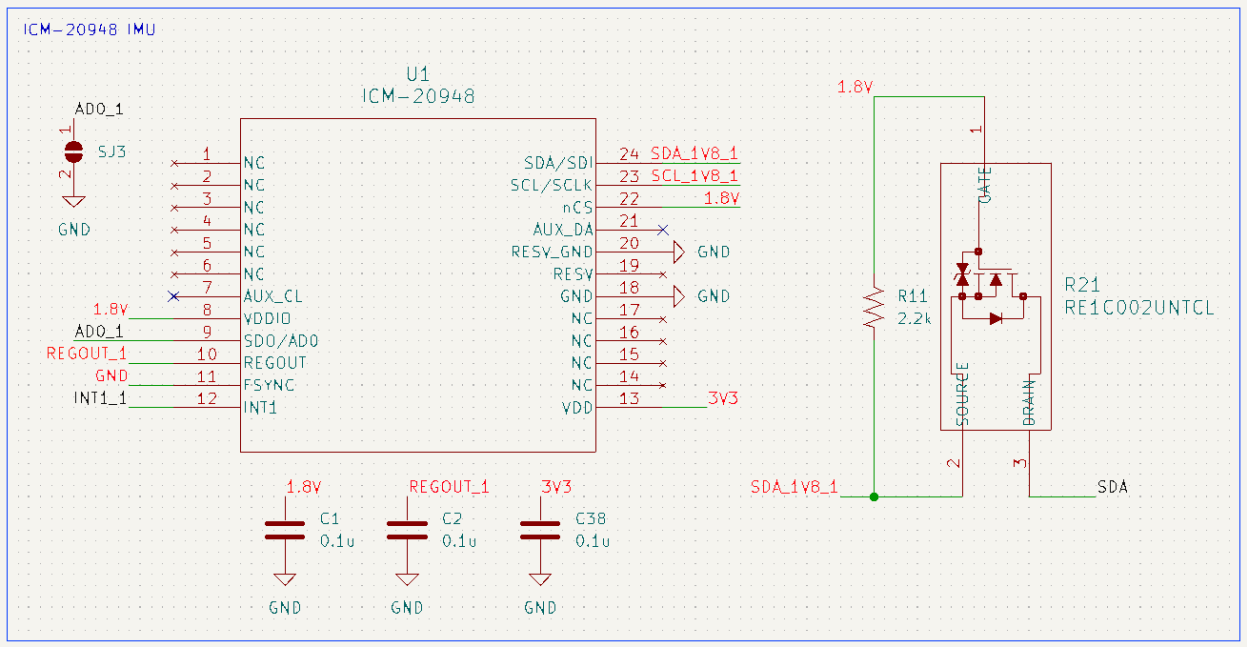

Figure 5. ICM-20948 IMU Schematic with SDA Level Shifter

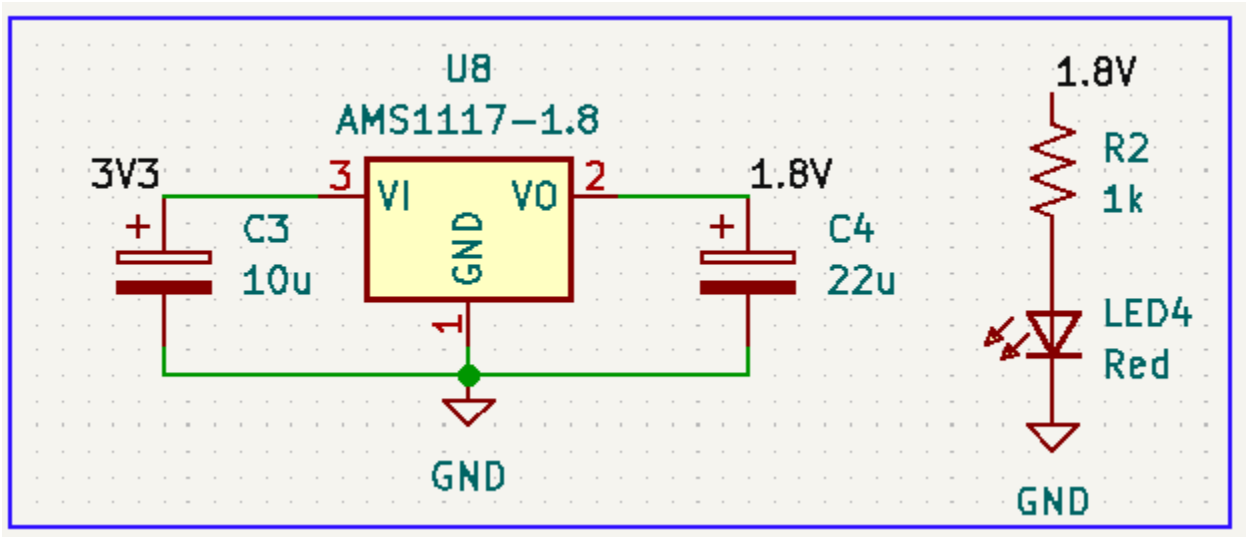

Figure 6. 3.3V -> 1.8V LDO Schematic

See **Figure 7** below for the placement of the sensors. These were intentionally placed physically near each other to ensure that measured fields would be as close to each other as possible.

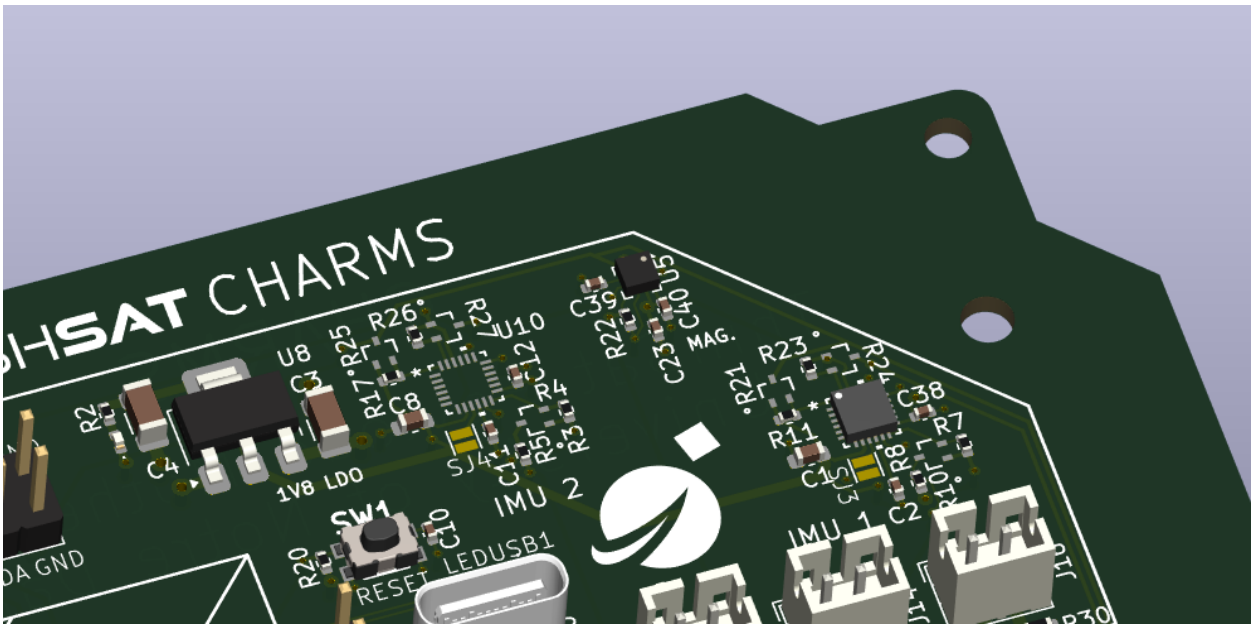

Figure 7. IMUs and Magnetometer with 1.8V LDO placed on CHARMS PCB

3.3.2.3 IR Cameras

The CHARMS board includes support for dual MLX90640 IR cameras, which serve as non-critical payload instruments for Earth observation (and eventually nadir pointing horizon detectors). These cameras are interfaced through generic JST I2C ports, meaning that external I2C accessories can be swapped in place of the cameras. This provides additional flexibility if CHARMS hardware requires another external sensor in future iterations, allowing previous board iterations to be retrofitted as necessary. I2C was selected as the communication protocol for these cameras since the resolution is only 24x32, meaning that only low throughput is necessary. For higher resolution cameras, faster protocols should be employed.

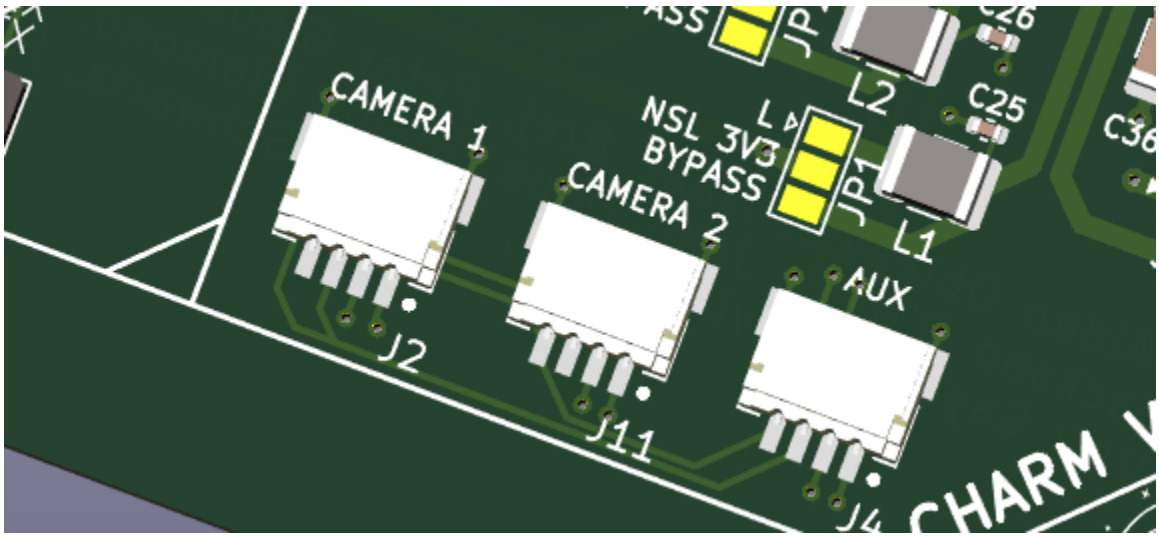

Figure 8. IR Camera Breakouts on CHARMS PCB

3.3.3 Current Drivers

CHARMS's magnetorquers are essentially coils of wire with low resistances (on the order of tens of ohms), so any solution to power them should be able to provide high currents. Previous iterations of CHARMS utilized a homemade H-bridge with power MOSFETs, allowing bidirectional control, but shoot-through current on startup caused many problems with unnecessarily high power consumption and difficulties with overloading the 5V rail, which CHARMS reserves for just the magnetorquers. Instead, CHARMS uses a dedicated TB6612FNG motor driver, which provides bidirectional current control, dual channel access, built-in protective diodes, and built-in power gating to minimize static draw. This proved to be an effective solution, providing simple PWM control and allowing any arbitrary current to be sent to the torquers. As a protective feature, 0-ohm resistors were added to the current path in case the torquer resistances were lower than expected and current draw was too high, but these did not need to be replaced. **Figure 9**, below, shows the schematic for the TB6612FNG. Note that this is a dual-channel chip, so one was used for the X and Y direction, and another was used just for the Z.

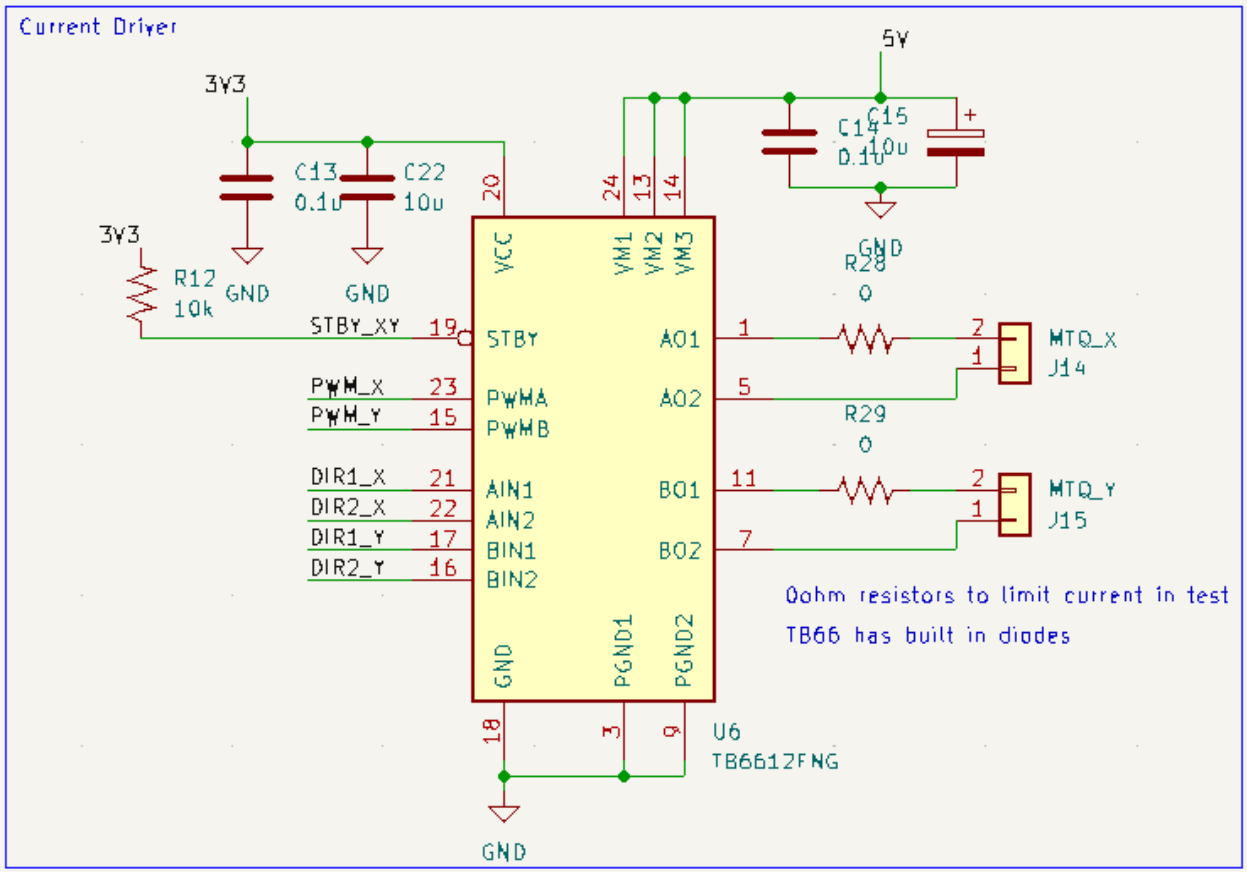

Figure 9. TB6612FNG Magnetorquer Driver IC

Interfacing with the torquers happens through 2-pin JST connectors, which were chosen because of their standard size and high current rating. The direction of each torquer is controlled by two opposite polarity GPIO signals, and the magnitude is controlled by a PWM signal. The

TB6612FNG handles all protection and isolation, including protective diodes to stop voltage spikes often found in inductive loads like motors and magnetorquers.

Layout for the drivers is shown in **Figure 10**, where large traces were selected for high current torquer power (up to 300 mA) and small traces were used for GPIO signals and 3.3V power (used just for logic level comparison). Note the large decoupling capacitors to ensure consistent input power to the torquers.

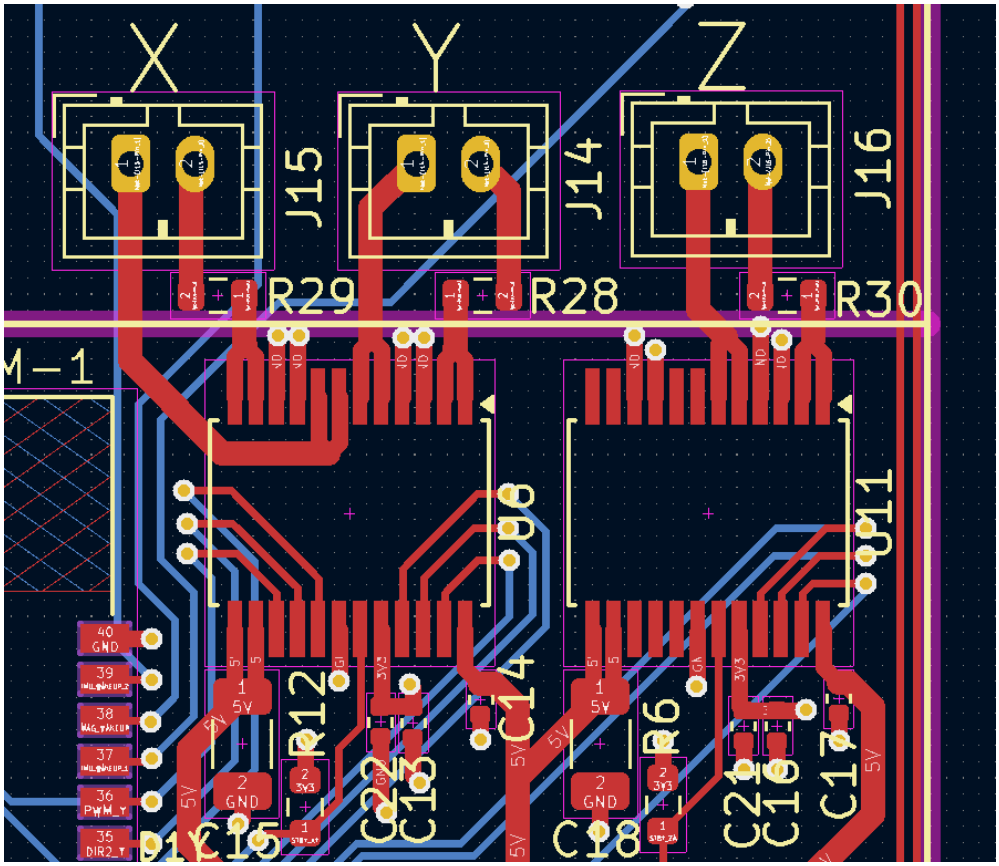

Figure 10. TB6612FNG Magnetorquer Driver IC Layout

3.3.4 Power

Since CHARMS is powered through the NSL 20-pin connector, the power path is greatly simplified. All the battery management and filtering hardware exists on the NSL bus, but CHARMS still has several power-related considerations to simplify board bring-up. CHARMS has both 5V and 3.3V power inputs available, but for early board iterations and software tests, it is inconvenient to simulate the NSL bus and its two power supplies. CHARMS has a physically isolated 5V -> 3.3V LDO so that the entire board can be powered from a single USB-C 5V input. This LDO is enabled with jumpers during debugging and can be physically unplugged to prevent reverse current flow or unnecessary static power consumption.

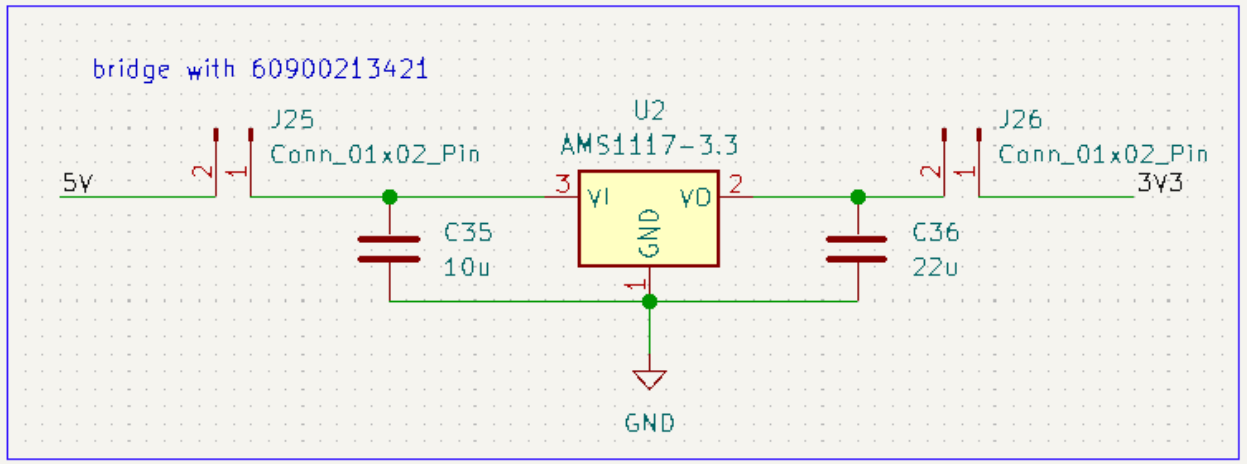

Figure 11. CHARMS Debug 5V -> 3.3V LDO

Another important hardware consideration occurred after Integration Readiness Review (IRR) testing. From the IRR requirements, listed in prior sections, CHARMS cannot draw more than 5A on either power rail during the first 100 ms after being powered on. On the 3.3V rail, CHARMS experienced a peak current of over 7A for around 50 ns. To counteract this inrush current, an inductor rated for the proper current and voltage was added in series with the power path, ensuring that the change in current is resisted and CHARMS is able to limit its strain on the bus. After adding this inductor, the inrush current dropped to 4.3A, meeting the requirement. **Figure 12** shows the schematic for the protective circuit.

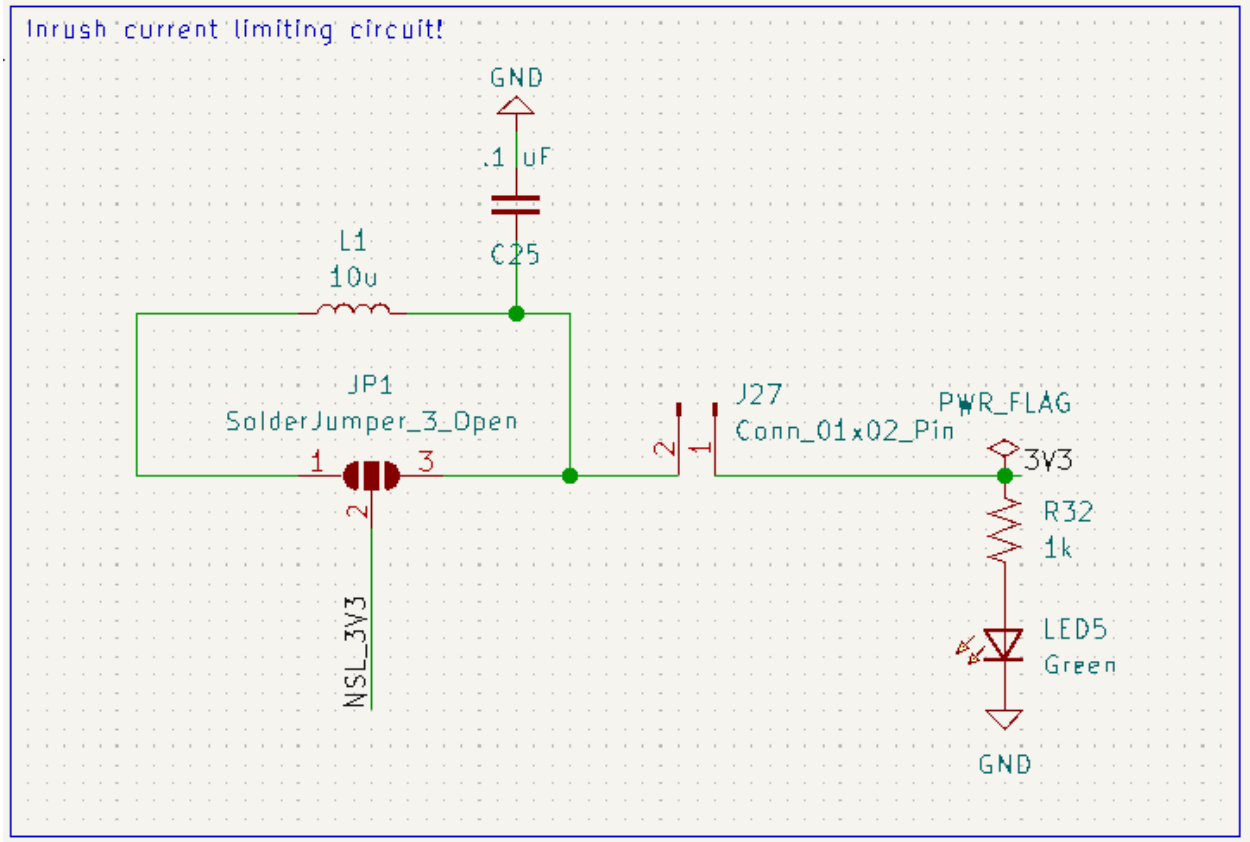

Figure 12. Inrush Current Limiting Circuit

3.3.5 Bus Interface

As detailed in previous sections, CHARMS interfaces with the rest of the NSL satellite through a 20-pin connector. This enables easy plug-and-play capability and simple replacement of the CHARMS module. CHARMS uses a 5V power supply for the magnetorquers, a 3.3V power supply for the ESP32-S3 and for digital communication, and the designated TTL UART pins to send and receive information packets to and from the rest of the system. **Figure 13** shows the 20-pin connector routed on the CHARMS PCB. Note that there are four analog lines that are routed to GPIO pins on the bus, but these are not used and are included simply for potential future expansion.

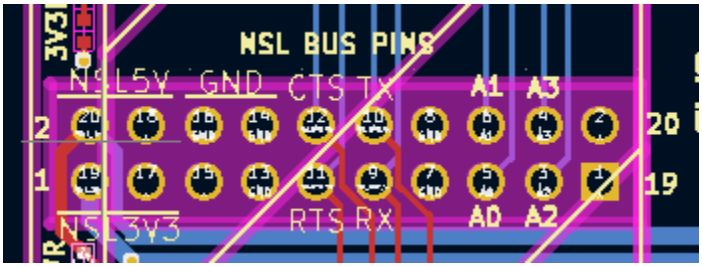

Figure 13. NSL 20-pin connector routing

3.3.6 Mechanical Considerations

The mechanical design of the CHARMS PCB was developed to fit precisely within the payload volume allocated by NearSpace Launch (NSL) and to interface seamlessly with both the bus and the structural components of the CHARMS system. The board conforms to a compact rectangular outline with rounded corners and precise edge clearance to prevent shorting against the aluminum chassis. Critically, the PCB also serves as the physical inner wall between the CHARMS payload and the NSL bus, forming a structural and electrical barrier that simplifies integration and ensures clean modular separation.

Four primary mounting holes at the board's corners enable rigid attachment to the NSL satellite frame, while four additional threaded holes provide secure connection to the CHARMS aluminum chassis. The 20-pin bus connector is placed in exact accordance with NSL's Interface Control Document (ICD), allowing direct alignment with the satellite's bus interface for both power and serial communication.

Component layout was guided by mechanical clearance constraints and signal integrity requirements. Tall components—such as debug headers—were placed away from interference-prone zones, avoiding conflicts with external hardware like the IR camera mounts and magnetorquer supports. A designated keepout zone on the board's left side, featuring a 0.1 mm clearance margin, was reserved to accommodate the large Z-axis magnetorquer. Despite the reduction in usable board area, careful design allowed full routing and placement of all required components.

The board follows a 4-layer stackup with a signal–ground–power–signal configuration. This provides robust power delivery and grounding, while maintaining simple, manufacturable

geometry. The mechanical and electrical interfaces were verified through physical fit checks and finalized during the Integration Readiness Review (IRR).

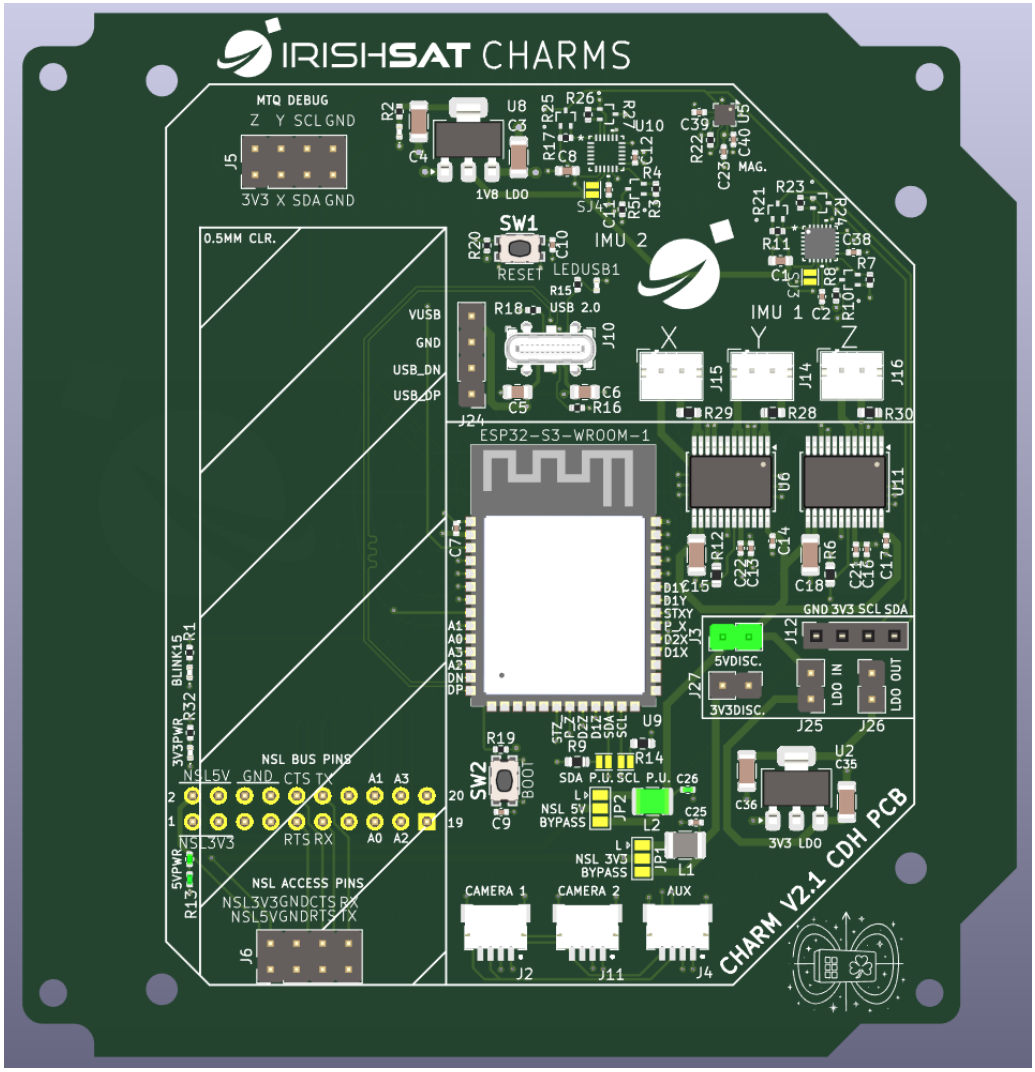

Figure 14. CHARMS PCB Physical Layout

3.4 Detailed Design and Operation of Subsystem 2: RTOS/Software

The software for the CHARMS was developed in C++ using the arduino development platform provided by PlatformIO. Using this development tool, the CHARMS team developed the CHARMS Real-Time Operating System (RTOS) which manages all tasks that CHARMS needs to carry out during operations. Additionally, state variables are used in addition to RTOS task management to control the state of the payload and ensure smooth operation.

Please refer to this file path to review the full software implementation. There are too many files to provide a full code listing in **Appendix C, D, and E**:

"...\CHARM Senior Design Code\Payload RTOS\CHARMS RTOS v2"

This contains the PlatformIO project, which the CHARMS payload runs as flight software. These files are available for download on the CHARMS Senior Design website.

3.4.1 High-level Architecture

Figure 15 shows the basic architecture of the CHARMS RTOS.

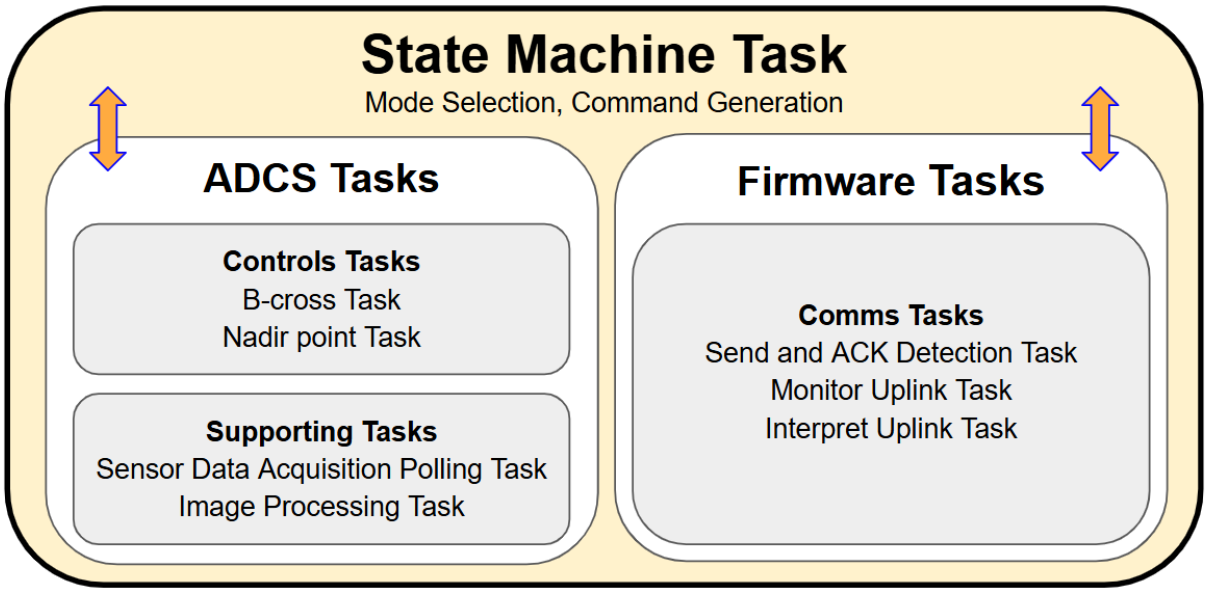

Figure 15. CHARMS Software Architecture

The State Machine Task manages the payload state as well as performing state start-up routines. This means that when the payload switches between states, some actions need to happen once at the beginning of that state, actions that are carried out by the State Machine Task. The next level down describes the separation between control tasks and firmware tasks. The ADCS tasks are those needed to adjust the orientation of the satellite, whereas the Firmware tasks are those that facilitate communication between the CHARMS MCU and the NSL bus flight computer. In-depth descriptions of each task can be found in the following sections.

The tasks within the RTOS all reference shared data, all of which either reside in C++ structs or freeRTOS queues. This allows for the accurate and safe transfer of data between different tasks that the CHARMS MCU is performing.

3.4.1.1 Structs

Many structs exist in the RTOS that hold critical information for proper operation. **Table 8** outlines every struct's name and their function within the RTOS.

Table 8. Description of Data Structs in the CHARMS RTOS

Name of C++ Struct	Description of Struct Function

StateVars	Contains all state variables governing the operation of the RTOS, including task delay times, uplink and downlink request times, packet sequence numbers, the RTOS start time, current and previous state information, booleans controlling which tasks are running, booleans controlled through ground commands to turn sensors on and off as well as flip control voltages, and track if the last Iridium downlink was successful.
TaskHandles	Contains all task handles needed to initialize each of the 7 RTOS tasks.
sensorOffsets	Contains calibration values for both on-board IMUs and the magnetometer to enable more accurate sensor readings.
tempOffsetCalcHolder	Pre-allocates space for intermediate values during the calibration process to avoid stack and/or heap overflows.
IMUData	Contains all available IMU data, including gyro spin rate readings, accelerometer readings, and magnetic field readings for the X, Y, and Z axes.
MagnetometerData	Contains all available magnetometer data, including magnetic field data for the X, Y, and Z axes.
IRCameraData	Contains pixels from the last captured IR image from the IR camera.
SensorData	Conglomerates all sensor data, including two IMU structs, one magnetometer struct, and two IR Camera structs, to store and update all sensor data available from the payload PCB.
RGB	Contains pixel information during the IR camera image acquisition process.

3.4.1.2 Queues

Several queues exist in the RTOS that hold critical information for proper operation. **Table 9** outlines every queue's name and their function within the RTOS.

Table 9. Description of Data Queues in the CHARMS RTOS

Name of RTOS Queue	Description of Queue Function

iridiumDownlinkBufQueue	Contains packetized information that is to be sent to the Iridium communications network and ultimately to the ground, adhering to the packet structure outlined by the NSL communications protocol as well as a CHARMS payload operational data packet structure so that we can decode and analyze operational data when the packet is received.
downlinkBufQueue	Contains packetized information that is to be sent to the NSL flight computer, adhering to the packet structure outlined by the NSL communications protocol.
uplinkBufQueue	Contains uplinked packets from the NSL flight computer to the CHARMS MCU. It can contain acknowledgement information, time information, GPS information, latency information relating to Iridium downlinks, ground commands to reconfigure the CHARMS payload, and a whole host of other information that can be requested from the flight computer.
sensorDataQueue	Contains most up-to-date sensor readings. This is accessed by the control tasks to calculate desired magnetorquer actuation. This is also accessed by the RTOS state machine to detect critical situations like a spin rate that is too high and/or increasing due to CHARMS operation.

Ground commands are available to reset all data queues into a known state if misconfiguration is thought to be an issue during operations.

3.4.1.3 Setup Routine

On power-up, the C++ scripts run setup routines which initialize critical systems on the CHARMS PCB as well as in the CHARMS RTOS.

First, the software initializes all critical communications links including the UART connection between the CHARMS MCU and the NSL flight computer as well as the I2C bus to communicate with all on-board sensors.

After, communications is successfully initialized, the software initializes all output pins including the Request-To-Send (RTS) pin, the PWM pins which drive the magnetorquers, and the magnetorquer current direction pins to configure the current driver to create positive or negative B-fields in a desired axis. Initial states, either HIGH or LOW, are set for all output pins to put the CHARMS payload into a known state. The only input pin, the Clear-To-Send (CTS) pins, is also configured.

Next, all sensors are initialized and communicated with over the I2C communications bus. This includes two MLX90640 IR Cameras, one LIS2MDL Magnetometer, and two ICM20948 IMUs, all of which are used during operations to calculate and control satellite attitude.

Lastly, all RTOS objects including queues and tasks are initialized before the RTOS begins. To initialize the RTOS queues, you must set these two quantities:

- Queue total size

- Queue individual item size

To initialize a task, you must set these four quantities and objects:

- Task name
- Task stack allocation size
- Task priority level
- Task handle

The most critical parameters in a proper RTOS function is making sure that your stack size can support all task activities and also making sure that task priority is set properly to generate the correct task order when multiple tasks are ready to run at the same time. To avoid any memory issues, all memory is statically allocated outside of the RTOS tasks.

3.4.1.4 State Machine Task

The State Machine Task is the last software implementation which governs the high level architecture of the CHARMS payload. The State Machine Flow Chart can be found in **Figure 16**.

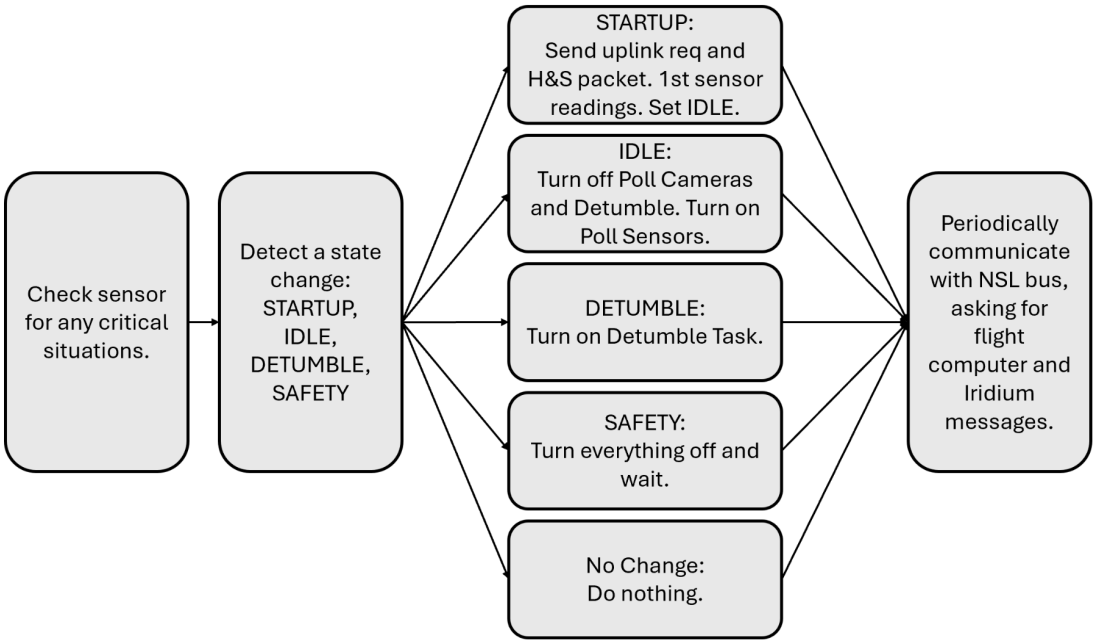

Figure 16. State Machine Task Flow Chart

By periodically communicating with the NSL bus, the CHARMS payload stays up to date in ground commands present on the Iridium network as well as other messages from the NSL flight computer. Additionally, if the CHARMS MCU does not communicate with the NSL bus at least once in 4 minutes, the bus will power cycle the payload. This avoids any power cycling during nominal operation.

3.4.2 Firmware Design

The firmware design handles all communications between the CHARMS payload and the NSL flight computer, adhering to the proprietary communications protocol provided by NSL. Tasks were written to receive commands from the serial monitor for testing purposes, to send packets waiting in the downlink buffers and wait for NSL acknowledgements, and to interpret any messages that NSL sends to the CHARMS payload.

3.4.2.1 Monitor Commands Task

This task is strictly for testing purposes and enables the software designer to send commands over the serial monitor to the CHARMS payload to test certain operations while the RTOS is running. **Figure 17** shows the flow chart for this task.

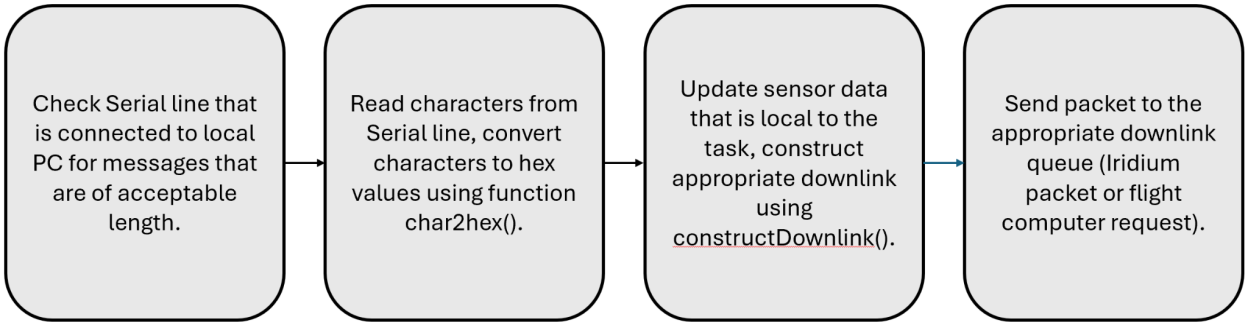

Figure 17. Monitor Commands Task Flow Chart

The constructDonwlink() function is integral to proper operation of the CHARMS payload. This function takes in the RTOS state struct, the command request, and the most current sensor data to construct the appropriate packet which adheres to the NSL packet structure. This includes health and safety packets containing the current state of the satellite, detumble packets which contain operational IMU data, bus radio reconfiguration, NSL bus uplink checks, and iridium network uplink checks. The NSL packet structure can be found in **Table 10**.

Table 10. NSL packet structure for proper communication [1]

Payload to S4	Message Format (205 bytes)				
Description	**Header**	**Function Byte**	**Data**	**BUSY Time**	**Notes**
Send Payload Data to Ground	50 50 50	F5	Length + 200 payload/ filler bytes	< 35 s	S4 will ACK if good, then attempt transmission for 35s, then reply with PASS/FAIL
Request Configuration	50 50 50	F3	201 filler bytes	< 1 s	Requests current S4 parameters.
Change Configuration	50 50 50	F4	201 data + bytes	< 1 s	Reconfigures S4 parameters. Resets on power-off
Check Network for Uplink Data	50 50 50	47	201 filler bytes	~ 15 s	S4 connects and checks queue, returns after 15 s.
Check Buffer for Uplink Data	50 50 50	48	201 filler bytes	< 1 s	Returns buffer data
Check Last Serial Packet Status	50 50 50	49	201 filler bytes	< 1 s	Returns last serial packet status and latency
Request UTC Time	50 50 50	4A	201 filler bytes	< 1 s	Returns current UTC time from modem (if available)
Request Health and Safety	50 50 50	F1	201 filler bytes	< 1 s	
Request GPS Geodetic Packet	50 50 50	F7	201 filler bytes	< 1 s	Optional
Request GPS Cartesian Packet	50 50 50	F8	201 filler bytes	< 1 s	Optional
Set Power Switch List	50 50 50	0B	4 switch bytes + 197 filler bytes	< 1 s	S4 sends switch list to EPS, EPS responds with confirmation
Request EPS Health and Safety	50 50 50	0C	201 filler bytes	< 1 s	S4 sends request to EPS, EPS responds with its H&S packet

The most frequently used commands by the CHARMS payload is “0xF5” to downlink data to the ground, “0x47” to check for ground commands from the CHARMS operators, and “0x48” to check for messages from the NSL flight computer.

3.4.2.2 Send and Acknowledge Task

The Send and Acknowledge Task is another critical firmware task which manages sending constructed downlink packets (iridium or flight computer) over the UART lines and interpreting NSL’s ACK or NACK messaging. If this task detects that the response from NSL is something other than ACK or NACK, it will send the packet to the Interpret Uplinks Task for further processing. The flow diagram for this task can be found in **Figure 18**.

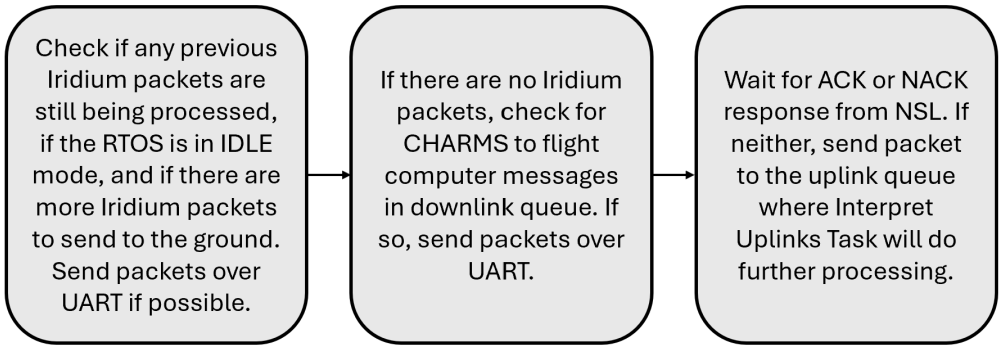

Figure 18. Send and Acknowledge Task Flow Chart

3.4.2.3 Interpret Uplinks Task

This task decodes all messages from NSL which are not ACK or NACK messages. For example, some messages from NSL will tell you some sort of information that you can decode such as GPS coordinates, time stamps, radio configurations, and so on and so forth. The most common uplink message that CHARMS waits to decode is a ground command from the Iridium network. Beyond the NSL-defined header and functional byte protocol, CHARMS adds another layer to the packet structure with its own set of functional bytes to carry out payload reconfiguration. The first six bytes of any uplink containing ground information are dedicated to the NSL packet structures, and are outlined in **Table 11**.

Table 11. NSL packet structure, first 6 bytes of message from ground

Header	**Function Byte**	**Ground Command Sequence Number**	**Number of Bytes in Message**
Bytes 0-2	**Byte 3**	**Byte 4**	**Byte 5**
0x50 0x50 0x50	0x48	0x__	0x__

After the functional byte 0x48 is decoded as a ground message by the Interpret Uplinks Task, it will begin to look for which ground command it received by decoding the byte located at Byte 6. Available ground commands are described in **Table 12**.

Table 12. Ground commands for CHARMS payload

Ground Command name	Function Byte	Description
Downlink	0x30	Manually request a downlink packet. This will result in a health and safety packet being sent to the ground by the CHARMS payload containing state information, the most recent IMU data, and a timestamp.
Detumble	0x31	This will change the payload from any state into the DETUMBLE state. This essentially turns on the control algorithm and the actuation signals which drive the magnetorquers.
Nadir Point	0x32	This command will turn on a separate control algorithm which points the satellite's normal vector directly at Earth. This algorithm is not fully implemented yet, so this ground command is not currently supported.
Safety	0x33	This puts the payload into the SAFETY state, turning everything off and waiting to be power cycled by the bus for a full-system reset.
Stop	0x34	This takes the payload out of any state and places it in the IDLE state. This will manually turn off the magnetorquers and control scripts.
Flip Voltage X	0x35	In case of backwards hardware installation, it is possible that our magnetorquers speed up the spin of the satellite instead of slow it down. This allows for a software fix in case there are hardware problems.
Flip Voltage Y	0x36	Same as 0x35, different axis.
Flip Voltage Z	0x37	Same as 0x35, different axis.
Flip IMU	0x38	There are 2 redundant IMUs on the CHARMS PCB. This allows the operator to toggle between either IMU in case one malfunctions due to radiation.
Reset Queues	0x39	This resets all RTOS queues in case they become misconfigured during operation.

Interpret Uplinks Task Flow Chart can be found in **Figure 19**.

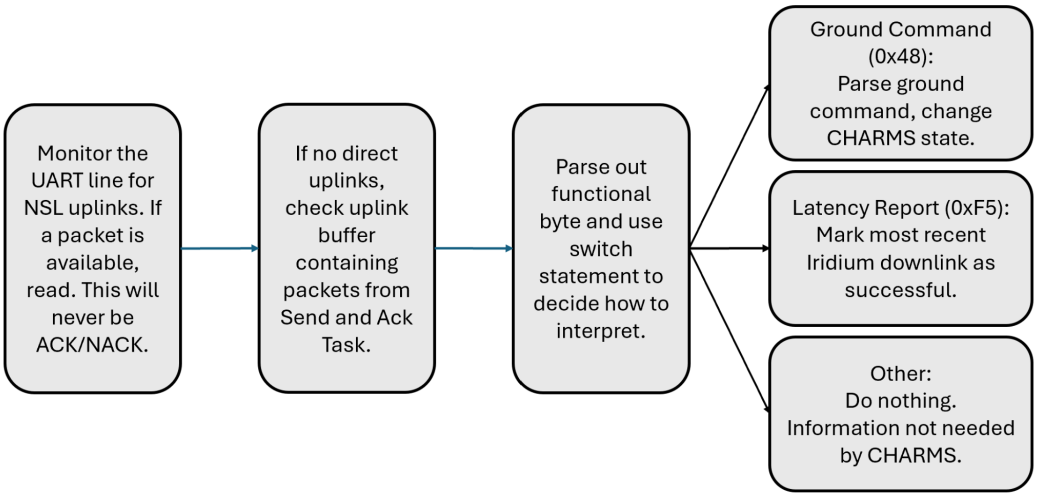

Figure 19. Interpret Uplinks Task Flow Chart

3.4.3 Control Design

The control design handles all sensor data acquisition and filtering. It then uses this data as inputs to a B-cross detumble algorithm which attempts to stop the spin of the satellite on all three axes by actuating the magnetorquers.

3.4.3.1 Poll Sensors Task

This task takes in sensor readings from 2 IMUs and 1 magnetometer, calibrates the data using calibration coefficients including hard and soft iron offsets for the magnetic field readings, filters the data using a low-pass filter algorithm, and stores this corrected data as a shared resource for all tasks and functions to access.

This task is critical for providing accurate, low-noise data for accurate and precise control. Because of poor IMU selection, the data is very noisy and makes fine-pointing with this first prototype of CHARMS impossible. Regardless, we do everything possible we can to reduce the noise of the sensors. The calibration process used to correct for sensor offsets can be found in **Section 3.3.2.1 Magnetometer.** Additionally, we low pass filter the data to avoid anomalous data spikes to affect the control algorithm. The flow chart for this task can be found in **Figure 20**.

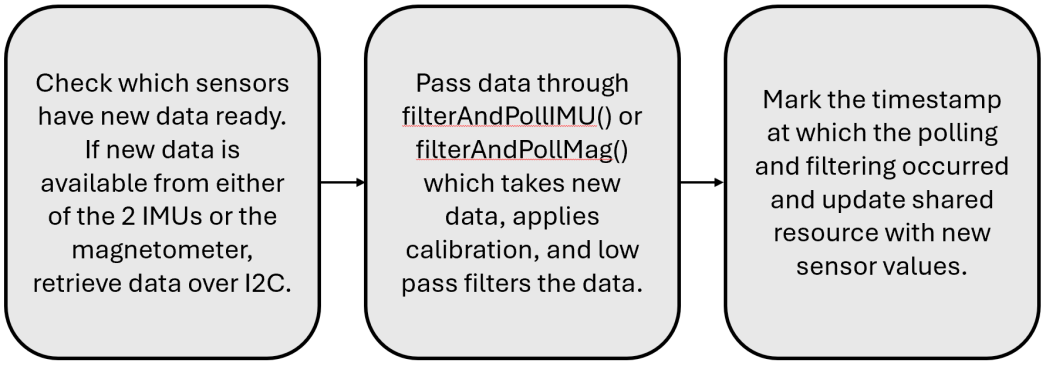

Figure 20. Poll Sensors Task Flow Chart

3.4.3.2 Poll Cameras Task

This task takes in image data from the 2 IR cameras, normalizes and grayscales the temperature readings for each pixel, and saves this data in a shared resource struct for all tasks and functions to access. This sensor data is used for the Nadir Point algorithm which is not a part of this iteration of the CHARMS payload. As such, this portion of the RTOS is never turned on for operations. However, it may be used to capture low resolution images of the Earth from orbit to take Notre Dame's first pictures from space.

This task was separated from the Poll Sensors task because it takes significantly longer to poll the MLX90640 cameras. For better control algorithm performance, we want to poll the IMU and magnetometer every 10 milliseconds. However, polling the IR cameras can take up to 1 second and was causing delays in updating the IMU and magnetometer data. The flow chart for this task can be found in **Figure 21**.

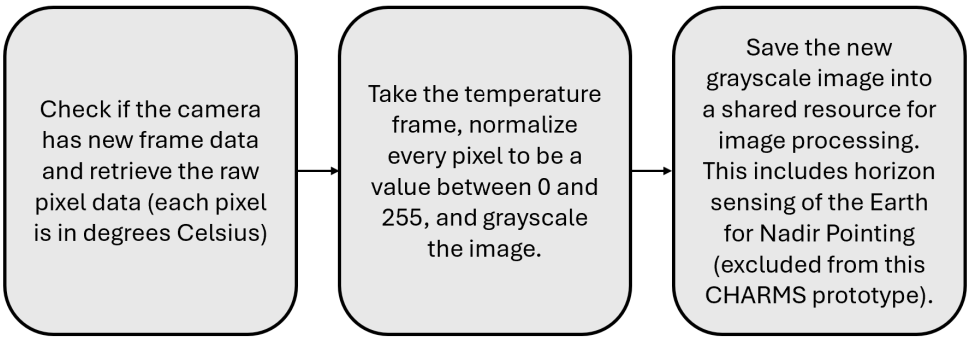

Figure 21. Poll Cameras Task Flow Chart

3.4.3.3 Detumble Task

The detumble task attempts to stop the spin of the satellite on all three axes. It does this by reading the magnetic field that is present relative to the body frame of the satellite, finding the desired torque which will oppose the spin of the satellite, and then calculating the desired magnetic moment using this equation in **Figure 22**:

$$\overrightarrow{\tau}_{magnetic} = \overrightarrow{\mu} \times \overrightarrow{B}$$

Figure 22. Guiding equation for B-cross detumble algorithm

After finding the desired magnetic moment, μ, the detumble algorithm finds the duty cycle to drive the magnetorquers on the X, Y, and Z axis to produce that magnetic moment.

Additionally, during operations the CHARMS payload periodically buffers operational data (about every 3 seconds) for downlink through the Iridium network to the ground after operations are complete. The payload waits to downlink data to the ground until after operations because the magnetorquers produce significant magnetic fields which could interfere with transmission of data through the Iridium radio. Because of this, data is buffered during operation and sent after the detumble is completed.

A full flow chart for the detumble algorithm is included in **Figure 23**.

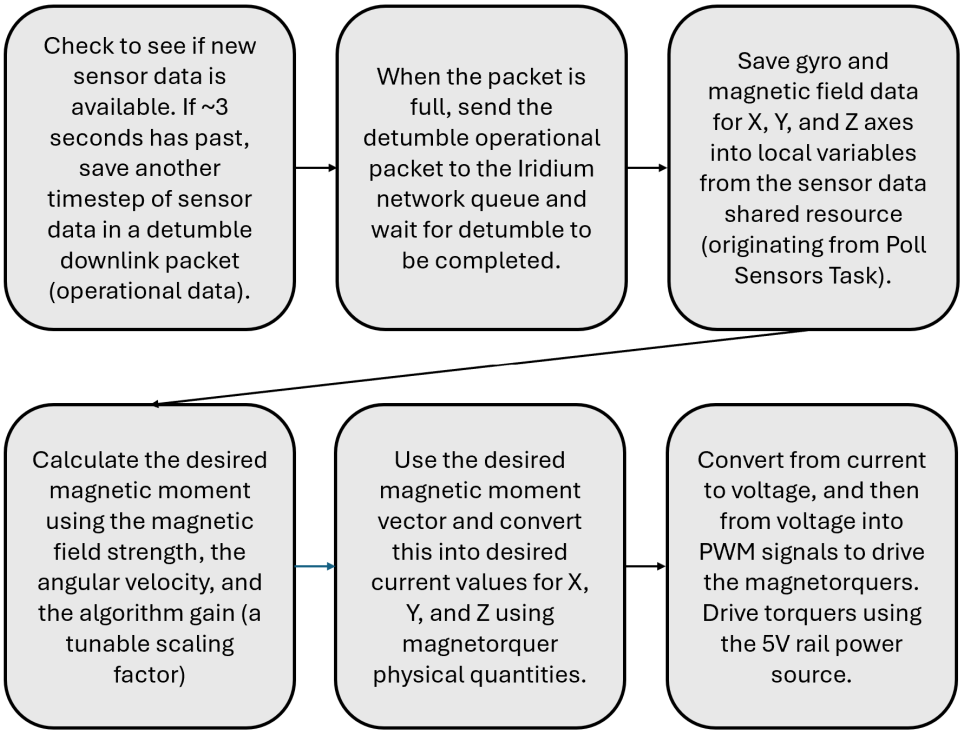

Figure 23. Detumble Task Flow Chart

After the detumble algorithm reaches a desired spin rate which is low enough to signal a successful detumble or the payload receives a ground command to exit the detumble state, all buffered operational data will be sent through the Iridium network for analysis on the ground. This will allow the operator to confirm that the CHARMS payload is working as intended.

3.5 Detailed Design and Operation of Subsystem 3: Magnetorquers

The CHARMS Attitude Determination and Control System (ADCS) utilizes a fully magnetorquer-based actuation architecture, employing three custom-designed, in-house-fabricated magnetorquers to provide torque in the X, Y, and Z axes. These magnetorquers operate by generating a magnetic dipole moment through current-carrying wire coils, which interacts with Earth's magnetic field to produce torque according to the Lorentz force:

$$\vec{\tau} = \vec{m} \times \vec{B}$$

Figure 24. Lorentz Force

3.5.1 Design Overview

Two magnetorquers aligned along the X and Y axes feature **mu-metal ferromagnetic cores** with high permeability to amplify dipole strength within compact volumes. The third, aligned along the Z-axis, is an **air-core solenoid** wrapped around the payload's rectangular perimeter to maximize cross-sectional area and leverage the full spacecraft volume for increased dipole moment.

This configuration was selected for its:

- Fully passive operation (no moving parts, low failure risk)
- Compact, CubeSat-compatible form factor
- Low power draw under constrained 5V/0.4A conditions

Table 13. Core and Air-Core Magnetorquer Specifications

Parameter	**Mu-metal Core**	**Air-Core**
Core Radius	0.32 cm	4.3 cm
Core Length	7.0 cm	8.5 x 9.7 cm
Relative Permeability	10,000	1 (vacuum)
Number of Layers	9	4
Total Turns	2,270	338
Resistance	21 Ω	91 Ω

Max Current	240 mA	55 mA
Magnetic Dipole	1.06 A·m²	1.67 A·m²
Max Magnetic Field	~1300 μT	~1300 μT

3.5.2 Optimization Strategy

The magnetic dipole moment was maximized while adhering to sizing, resistance, and power constraints through:

- Parametric sweeps using Python simulation scripts
- Qualitative modeling of magnetic dipole vs. wire count, current, and core dimensions
- Maximizing length-to-radius ratio for improved efficiency
- Selecting 30 AWG wire to balance packing density and resistance

Key insights included:

- More wire turns = stronger dipole until power limits
- Longer, narrower cores improved power efficiency
- Core permeability directly boosted B-field strength

3.5.3 Fabrication Process

All magnetorquers were fabricated in-house using a **custom-built automated coil-winding machine**. The machine employed a rotating core driven by a stepper motor to guide side-to-side threading with tightly packed coils and tight layer spacing via a tension-controlled wire feed. Wrapping counts were monitored and verified. The final cores were checked for tightness, continuity, and resistance.

3.5.4 Testing and Characterization

To validate and characterize the behavior of each magnetorquer independently, we conducted a series of isolated single-axis tests inside a zeroed Helmholtz cage. Each magnetorquer was placed individually at the magnetic center of the cage, and its B-field output was measured in all three axes while ramping PWM input and applying demagnetization procedures. These tests served five key purposes:

1. Quantify directional B-field contributions per axis
2. Verify that magnetic response is linear, symmetric, and smooth

3. Identify best core configurations and quantitatively determine their performance
4. Evaluate demagnetization effectiveness on mu-metal core hysteresis
5. Provide controls team with data for their algorithms

3.5.4.1 PWM vs. B-field Characterization

To understand the torque-generating capacity of the magnetorquers, PWM signals were swept across the full ±100% duty cycle while logging B-field strength along X, Y, and Z axes using precision magnetometers.

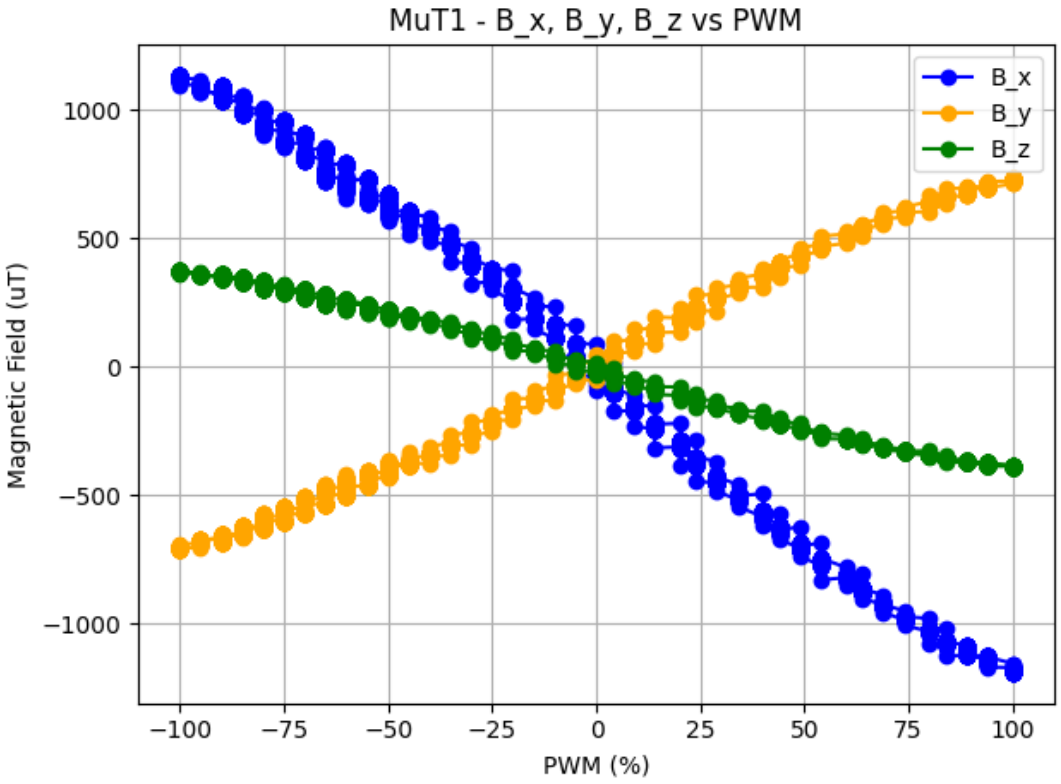

Figure 25. Magnetic Fields on All Axes for Single Magnetorquer vs PWM

This plot shows a strong, clean magnetic field response that varies with PWM duty cycle for each axis. The blue (B_x) and orange (B_y) components (corresponding to the axes most aligned with the magnetorquer's winding direction) show a symmetric, monotonic trend, indicating well-formed dipole generation and minimal cross-axis interference. The green trace (B_z) remains relatively flat, confirming that the Z component is much less decoupled to this magnetorquer, as expected.

- **Key insight**: Mu-metal cores produce strong and consistent fields in their aligned axis, and are relatively insensitive in the orthogonal directions. This validates axis isolation and directional torque authority.

3.5.4.2 B-field Over Time During PWM Cycling

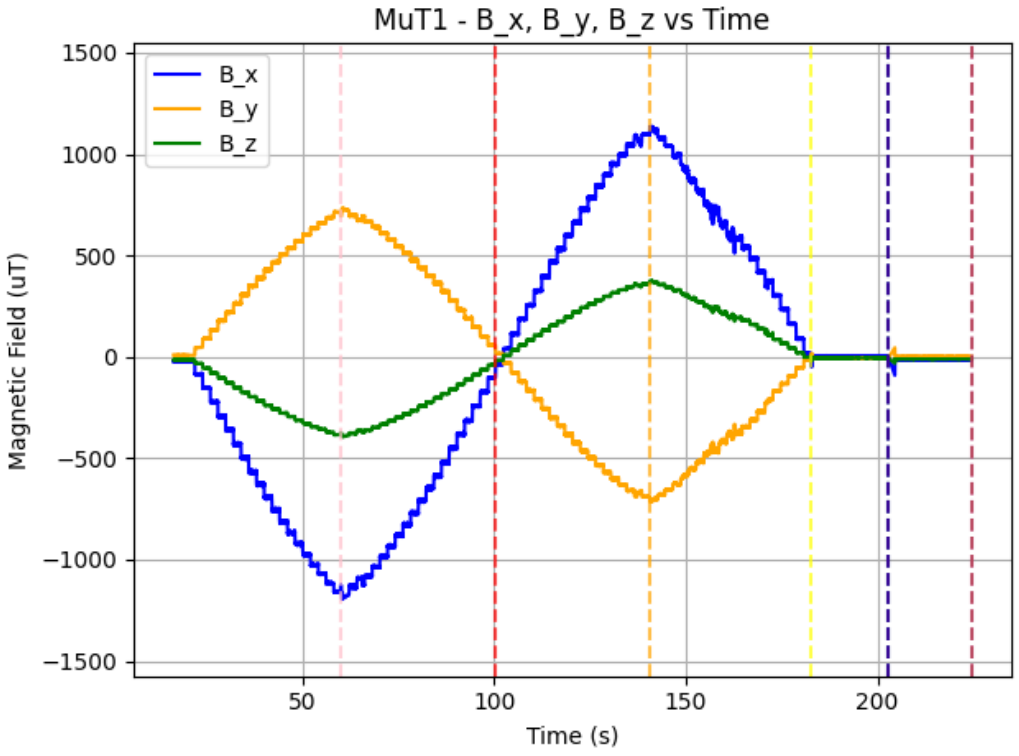

Figure 26. Magnetic Field on All Axes for Single Magnetorquer over Time

This time-based plot visualizes the magnetorquer's response to a sequential PWM ramp and reversal cycle. Each vertical dashed line marks a change in actuation. B_x and B_y, and B_z follow a predictable sinusoidal pattern, with B_z reaching the lowest magnitude of B field.

- **Interpretation**: This confirms that the field is not only proportional to PWM but also tracks time-varying inputs with minimal lag or hysteresis.
- **Notable result**: The clean reversal of magnetic field vectors during duty cycle inversion confirms reliable bidirectional control and minimal delay in magnetic response, validating closed-loop usability.

3.5.4.3 Residual Magnetization and Demagnetization Testing

To investigate hysteresis and retained field issues associated with the magnetization of mu-metal cores, each magnetorquer was tested for residual B-field after actuation, both before and after a demagnetization sequence.

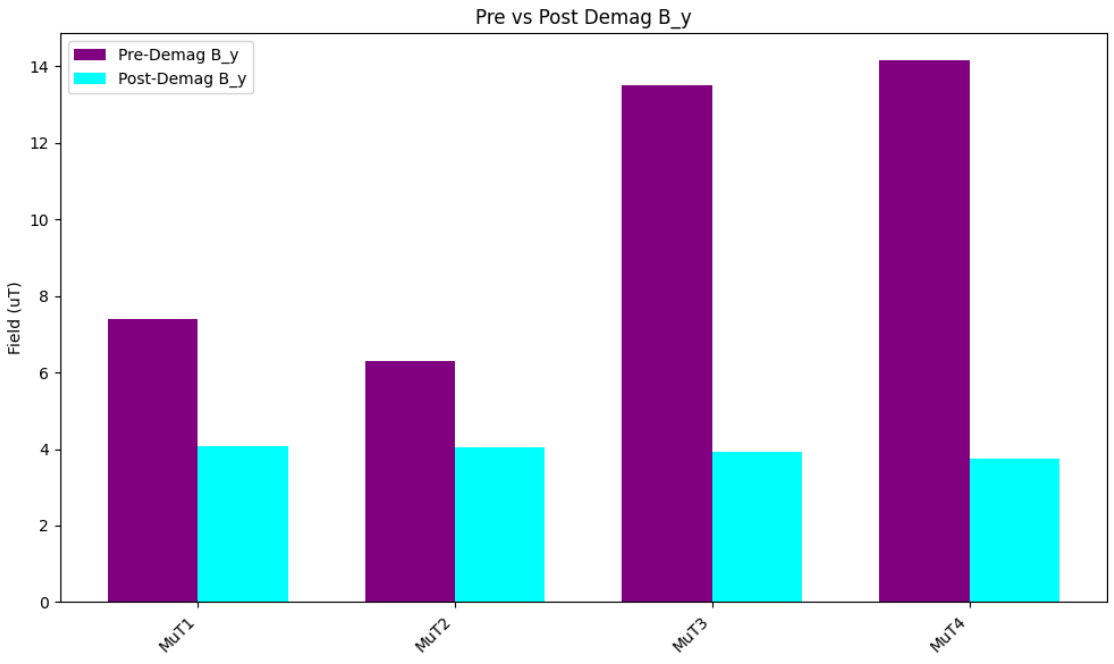

Figure 27. Residual Magnetic Field on Y Axis Before and After Demagnetization

This bar graph compares the residual B_y values before and after demag. All four torquers showed a measurable drop in retained magnetic field following demagnetization.

- **Key conclusion**: While all units retained some residual magnetization, the demag procedure successfully reduced it by ~40–70%, indicating partial but consistent effectiveness.
- **Design implication**: The inclusion of a software-controlled demag routine is justified for flight, especially after sustained actuation periods.

3.5.4.4 Comparative Maximum Field Strengths

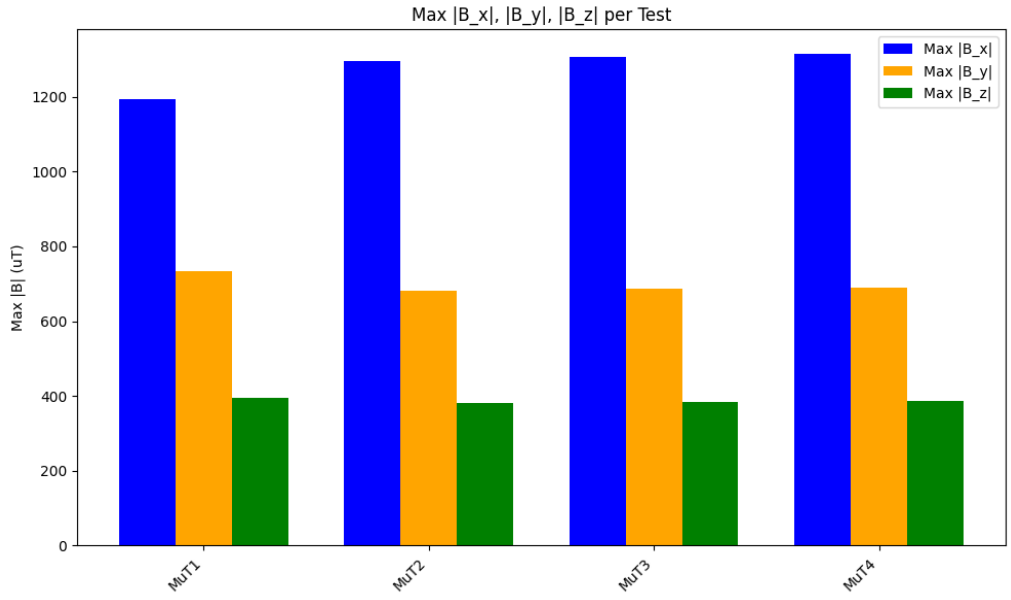

Figure 28. Amplitude of Magnetic Field on All Three Axes

This summary plot visualizes the peak field strengths recorded for each torquer during full PWM application.

- **Blue (B_x)** peaks dominate, followed by orange (B_y), and green (B_z), matching expected dipole orientations based on torquer alignment.
- **Implication**: The system exhibits predictable field magnitudes in the intended directions, with little deviation across the four torquers, demonstrating manufacturing consistency and axis alignment.

3.5.4.5 Summary

These individual magnetorquer tests affirm that:

- Each torquer delivers axis-specific, nearly linear, and reversible field responses.
- Residual magnetization can be effectively mitigated through onboard demag routines.
- Directional outputs match modeled expectations and support reliable torque control.

Together, these results reinforce the validity of CHARMS's hardware-in-the-loop control assumptions and confirm that the magnetic actuation system is ready for mission deployment.

The CHARMS magnetorquer subsystem provides reliable, directionally controllable torque with minimal power draw. Through iterative simulation, precision fabrication, and in-depth Helmholtz testing, the team successfully optimized each torquer for peak dipole performance under tight form factor and power constraints, demonstrating an effective solution for magnetorquer-only CubeSat ADCS control.

3.6 Detailed Design and Operation of Subsystem 4: Structures

The structures subsystem was beyond the scope of the Senior Design team's work on this project. All of the structural design work was completed by members of the IrishSat CubeSat Structures team. However, the structural design was driven by CHARMS magnetorquer subsystem functional requirements. Because CHARMS is required to have 3 magnetorquers, one on each major axis, the structures team must accommodate securing 2 rod magnetorquers and manufacturing one air core magnetorquer structure. In **Figure 29**, we can see the designed aluminum structure to fit system requirements. Circled in red are the 2 rod magnetorquers. Boxed in blue is the Z-axis air core magnetorquer.

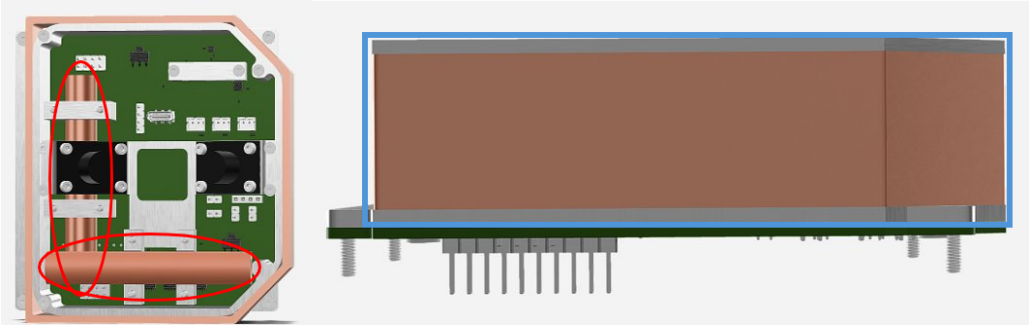

Figure 29. Magnetorquer Structural Design

See Section 5.1 Payload Assembly for more detailed information on the mechanical assembly of the CHARMS payload. Additionally, the CHARMS PCB team, driven by requirements from NSL, required the mechanical team to provide access holes for the USB and for signal line test points which allow for troubleshooting during integration with the rest of the satellite. These access points are seen in **Figure 30.** Also included are holes in the top plate so that the IR cameras can view outside of the satellite.

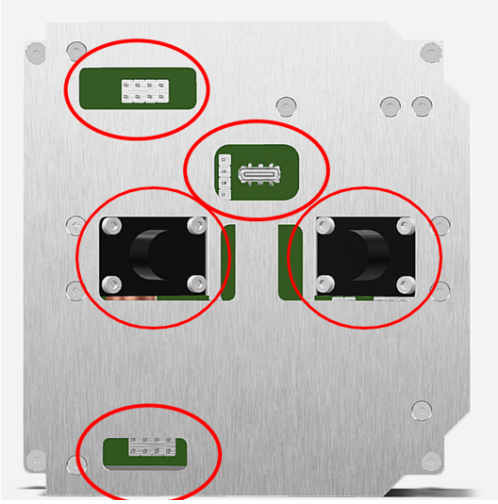

Figure 30. Access points and camera holes for troubleshooting and picture taking

Another requirement, driven by NSL requirements, is an external mechanical interface so that the full satellite can be assembled properly. These mounting points are seen in **Figure 31**. The

red circles indicate external mounting points and the blue circles indicate PCB mechanical integration with the air core structure.

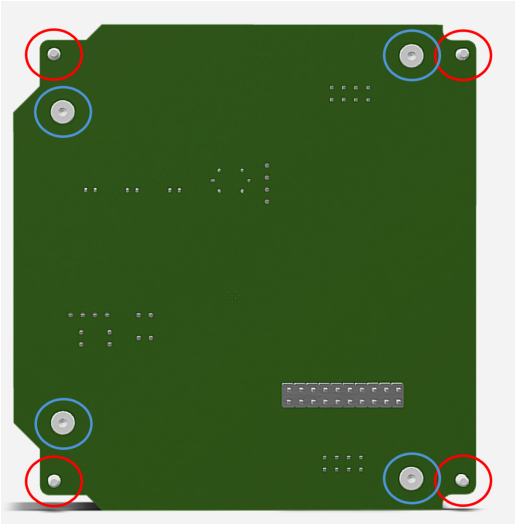

Figure 31. Mechanical integration of CHARMS

The structures team was also required to create a camera mounting bracket which extends through the middle of the payload space to hold the IR cameras brackets. The structural mounting points of the IR camera are seen **Figure 32**.

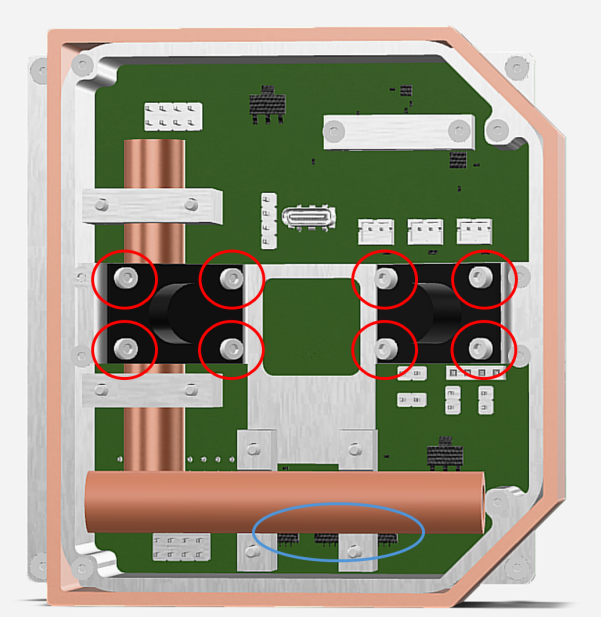

Figure 32. Attachment of IR cameras using screws (red) and JST board connection (blue)

Additional mounting points were added onto the aluminum top plate to support the connection of the IR camera bracket and wire clamps required for the magnetorquer cabling. This is seen in **Figure 33**.

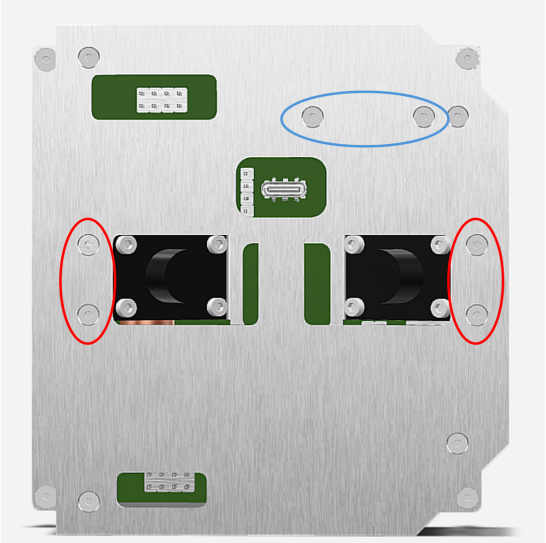

Figure 33. Attachment of IR cameras brackets (red) and wire clamp (blue) to top plate

This mechanical design by the IrishSat team directly supported the CHARMS electrical and operational functionality. A more detailed assembly process of the electrical and mechanical structures are included in **Section 5.1 Payload Assembly**.

3.7 Subsystem Integration

The CHARMS payload has very simple subsystem integration. All sensors chosen for the PCB are I2C enabled, making communication with all sensors very simple within the software subsystem. Additionally, all sensors have Adafruit development boards with accompanying libraries for sensor communication, making software and hardware integration seamless.

Integration of the magnetorquer subsystem is also very simple and uses 3 JST connectors which are mounted onto the PCB to interface with magnetorquer cabling. This cabling is mechanically clamped down to avoid damage due to the harsh vibrational environment during a space system launch.

4 System Integration Testing

4.1 Full Integration Testing

4.1.1 IRR Testing

Integration Readiness Review (IRR) testing was conducted by the CHARMS team to empirically validate that we met all NearSpace Launch requirements outlined in **Section 2.2 External NearSpace Launch Requirements**. The CHARMS team had to demonstrate to the NearSpace team that our payload was space-ready and with certain operational limits.

Valparaiso University which is partnered with NearSpace Launch on the Dream Big program, just like the CHARMS team, designed and fabricated and distributed an emulation PCB which is designed to act exactly like the NearSpace bus. This means that it directly emulates external power supplies, as well as containing an Arduino Mega board running emulator software. The Arduino Mega allows for seamless testing of the communications between the payload PCB and the NSL flight computer. It also allowed the CHARMS team to test electrical and mechanical characteristics of the payload, confirming that it fit within NSL requirements. This IRR testing setup is seen in **Figure 34**.

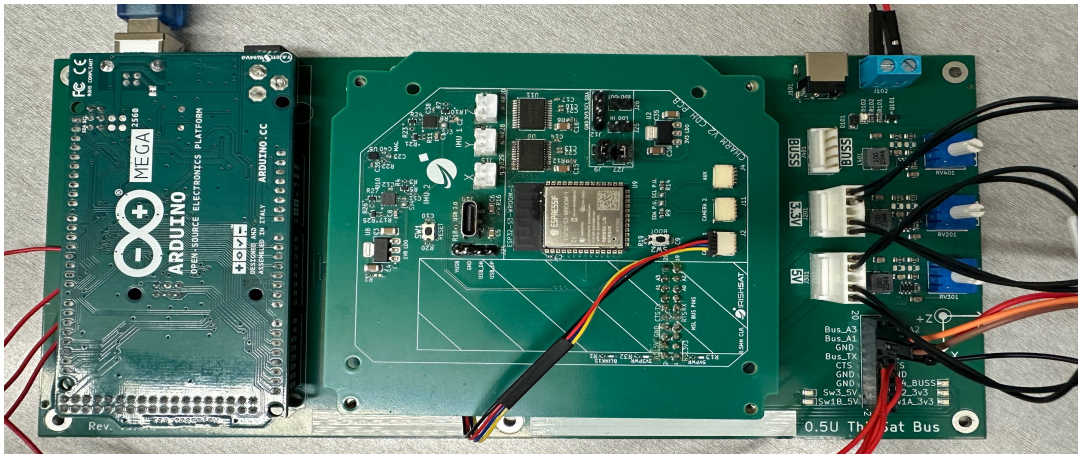

Figure 34. IRR testing setup using Valparaiso emulator board.

See the CHARMS IRR Tracker information in **Appendix A** to see detailed descriptions of all testing done for IRR.

4.1.1.1 Mechanical Testing

Reference **Table 4** from **Section 2.2.1** for detailed requirements listing.

Many of the mechanical requirements from NearSpace Launch were done through BOM inspection. Find the CHARMS BOM in **Appendix B**. These requirements include:

- Outgassing requirements for all materials and components (TML and CVCM values)
- Prohibited items in space, including energy storage devices and radio emitters

- Component operational temperature range fitting with space standards

These BOM inspection items were confirmed to be met by the CHARMS team. In Addition to BOM inspections, the team was required to carry out visual inspection of the payload to ensure that the payload is free of dust and contaminants, controlled by alcohol cleaning of all payload surfaces.

The rest of mechanical requirements were met by examining our payload CAD's integration with the NSL bus CAD. Certain items that needed to be confirmed through CAD inspection were:

- Fit the payload space described in **Figure 35**
- Rigid mounting at specified locations by NSL
- Electrical bus connection at specified location

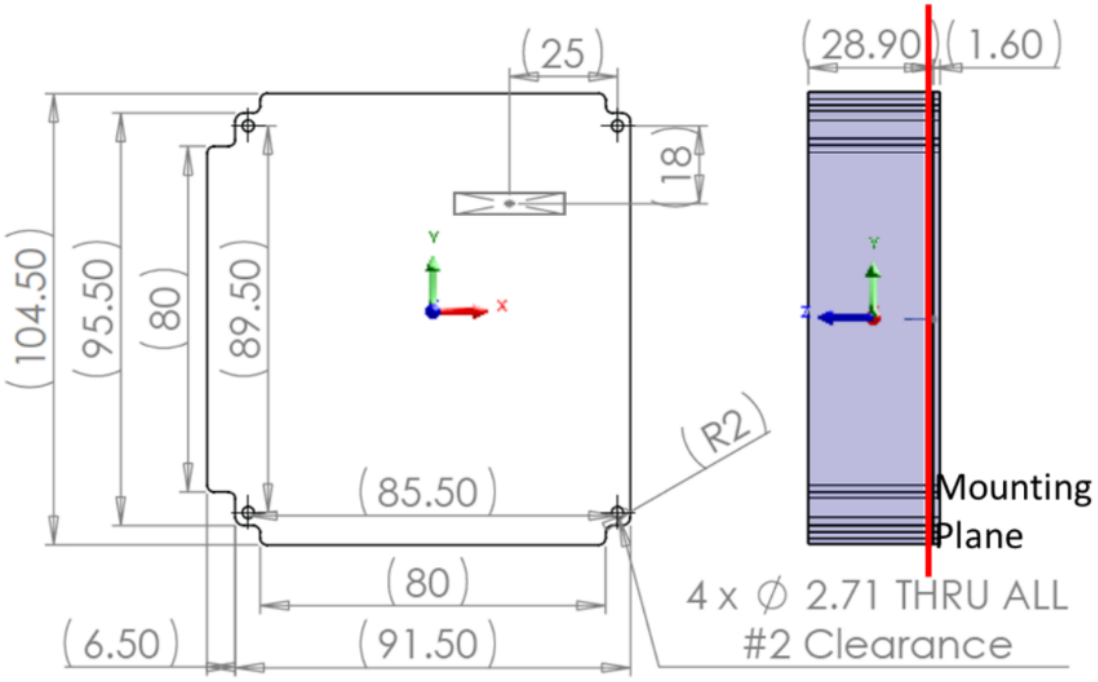

Figure 35. Student payload space on the satellite

Additional mechanical testing such as characterizing the payload mass and confirming that it is less than 500 grams was performed. Additional verification that the CHARMS payload meets mechanical integration requirements was done by successfully integrating mechanically and electrically with the Valparaiso emulator board, which is built to NSL specifications.

See the BOM in **Appendix B** and CHARMS IRR tracker in **Appendix A** for more detailed information.

4.1.1.2 Electrical Testing

Reference **Table 5** from **Section 2.2.2** for detailed requirements listing.

Electrical testing was almost entirely complete using the Valparaiso emulator board. This emulator board allowed us to show that our payload met these requirements:

- Powered by bus (external) power only, remains functional
- Can survive voltage ripples
- Below the required inrush current limits set by NSL
- Pulls less than maximum steady state current on power rails
- Payload is tolerant to abrupt power loss

To carry out payload testing, the CHARMS team used the Schlafly Lab at Notre Dame for access to current sensors and oscilloscopes to take the appropriate measurements. The testing setup used to check all electrical requirements is seen in **Figure 36** below.

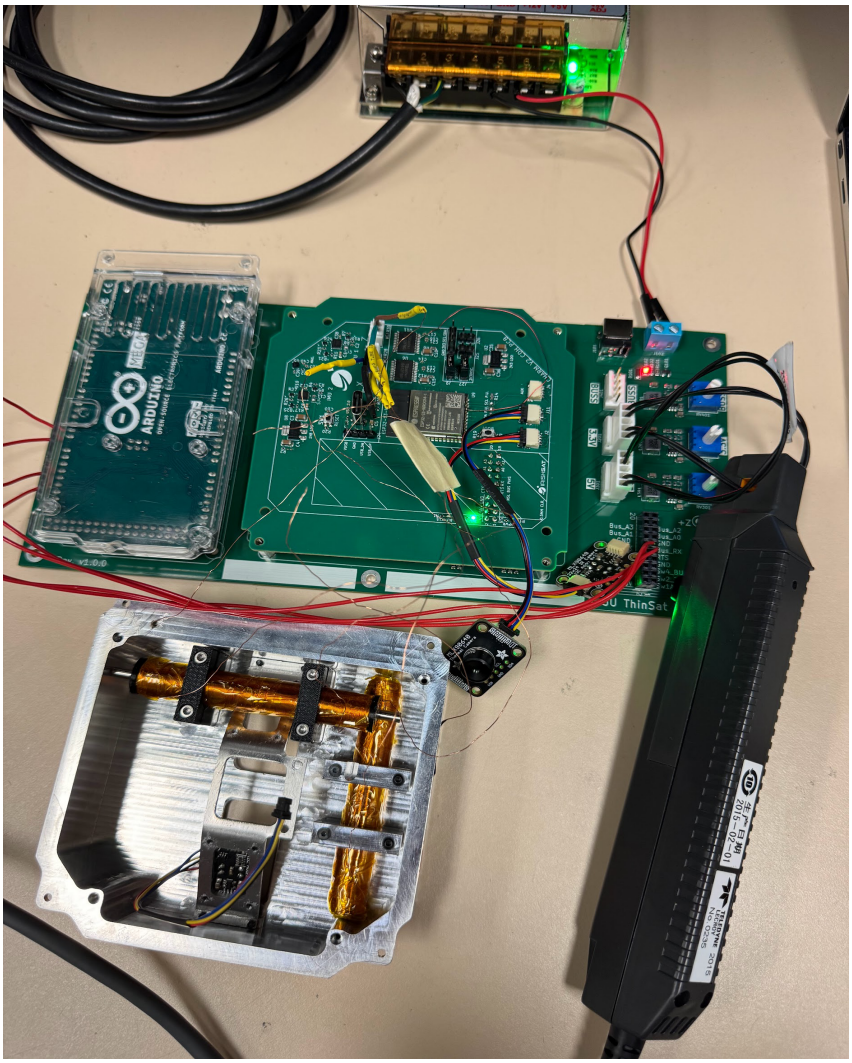

Figure 36. Schlafly Lab IRR electrical testing setup

In this figure, the Arduino Mega is on the left and is used to run the emulator software. The CHARMS PCB is in the middle and runs the full CHARMS software suite. The emulator board power systems are on the right and are connected to a 12V power supply picture on the top of the figure. A current probe is seen connected to the external 3.3V power rail, used to measure current draw and inrush current. Finally, you see the CHARMS PCB wired up to the magnetorquer subsystem which is running at full power during testing.

The only requirement that the CHARMS payload did not meet was an inrush current on the 3.3V line that was too high on start up. As such, a series inductor was added to the 3.3V line to constrain the inrush current to below 5 amps.

Additionally, a test was conducted to cut off power to CHARMS and then powered back on to observe how the software and hardware reacted to random power cycling. The CHARMS code takes into account that it may be randomly power cycled while in any state including IDLE and DETUMBLE. This testing verified that power cycling during IDLE, DETUMBLE, or any other state simply places the payload back into the IDLE state, requiring another ground command to be sent to resume operational testing. This is because a power cycle causes CHARMS to reinitialize all hardware and the RTOS.

See CHARMS IRR tracker in **Appendix A** for more detailed information.

4.1.1.3 Firmware Testing

Reference **Table 6** from **Section 2.2.3** for detailed requirements listing.

Firmware requirements state that the payload should be able to downlink flight computer requests and also downlink useful data through the Iridium network for analysis on the ground. A detailed description of how the software is configured to do this is described in **Section 3.4 Detailed Design and Operation of Subsystem 2: RTOS/Software**.

Using the Valparaiso emulator board, the CHARMS team was able to demonstrate the smooth operation of the firmware. To view the information included in both health and safety packets and also operational detumble packets, view the code listing in **Appendix E**. This information is included in the constructDownlink() function.

4.1.1.4 Environmental Testing

Reference **Table 7** from **Section 2.2.4** for detailed requirements listing.

Using the testing setup seen in **Figure 37**, the CHARMS team was able to verify the pre-environmental functionality of the payload. This setup includes an air bearing, which allows for a low friction environment similar to that experienced in orbit. Additionally, the Helmholtz cage controls the magnetic environment within the cage to simulate the B-fields that would be experienced in orbit. See **Section 4.1.2.2 Empirical Verification** for full, pre-environmental testing results.

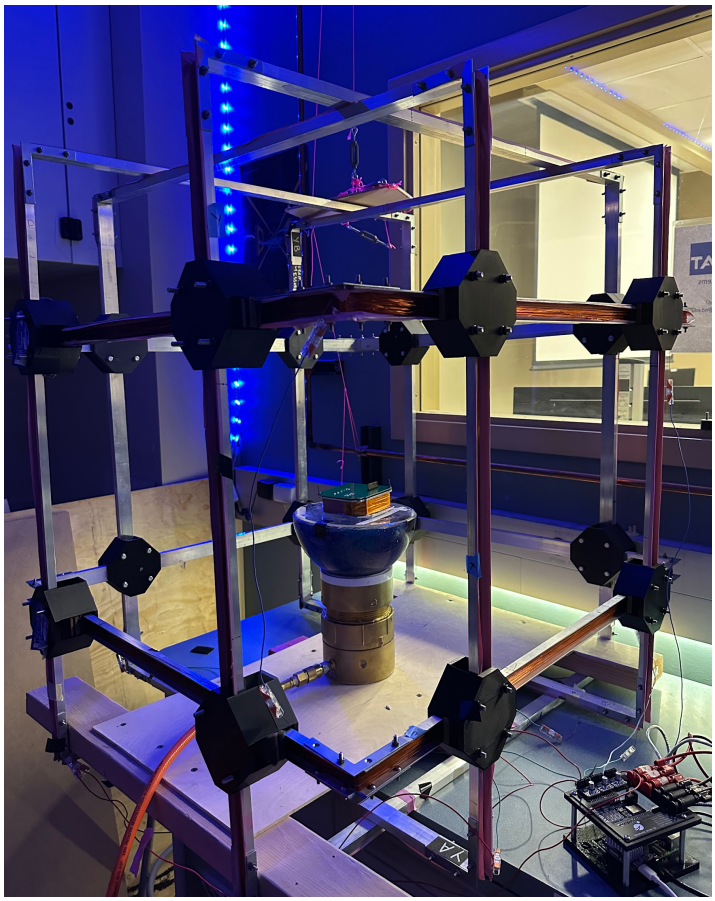

Figure 37. Empirical testing setup for verification of CHARMS functionality

The CHARMS team has yet to carry out the environmental testing plan outlined in **Section 2.2.4 Testing Requirements**. Testing plans for TVAC bakeout have been arranged with the Notre Dame Nanofabrication Lab and will happen in the near future. Additionally, IrishSat's testing equipment team has created a vibration table with which the CHARMS team will perform extended vibrational testing to induce any mechanical failures that the CHARMS payload may be susceptible to.

After performing this environmental testing, the team will carry out the same exact testing as in pre-environmental testing so that testing results can be compared.

4.1.2 Functional Testing

Functional testing was conducted to verify that all integrated subsystems of the CHARMS payload (hardware, software, and the magnetorquer assembly) operated as intended under representative operational and environmental conditions. These tests confirmed that the system responded accurately to user input, operated within expected performance bounds, and fulfilled mission-level requirements across power, timing, sensing, and control domains. Testing encompassed both manual command execution and autonomous state-based operations to validate transitions, sensor responsiveness, and real-time actuation logic.

4.1.2.1 Functional Test Setup

Two primary environments were used for functional testing: a bench-level emulator configuration and a high-fidelity orbital simulation platform.

During software, firmware, and communication-focused testing, the CHARMS PCB interfaced with the NSL bus emulator via the Valparaiso University emulator board. This setup allowed comprehensive emulation of the satellite bus's electrical behavior and command infrastructure. The configuration enabled validation that the FreeRTOS-based software stack properly initialized, interpreted incoming commands, and formatted outgoing telemetry for transmission via NSL's Iridium network.

For full integration testing, the complete CHARMS payload was assembled, with all electrical and mechanical interfaces in place. Inputs and outputs were connected to the PCB, the structure was fully enclosed and fastened, and the 20-pin connector was wired to a mass-matched power pack emulating 3.3V and 5V rail delivery from the NSL bus.

The full functional testing environment included:

- **Helmholtz Cage:** Used to null Earth's magnetic field and inject controlled magnetic fields emulating those encountered in Low Earth Orbit (LEO), using the PySol field simulation platform.
- **Air Bearing System:** A hemispherical bowling ball platform allowed near-frictionless movement—360° rotation on the Z-axis and ±120° on the X and Y axes—enabling realistic spacecraft dynamics.
- **BLE Telemetry:** Provided real-time data visualization of orientation, PWM outputs, magnetometer readings, and gyroscope measurements.

These tools together enabled comprehensive, closed-loop Hardware-in-the-Loop (HIL) testing of the CHARMS flight system, with emphasis on detumble responsiveness and magnetorquer behavior under flight-representative conditions.

4.1.2.2 Functional Test Protocols

Functional testing was executed in two key phases.

4.1.2.2.1 Emulator-Based Testing with Valparaiso Board

These tests focused on validating firmware functionality, system state handling, and telemetry reliability:

- Startup Routine Test
  Upon power-up via the emulator board, CHARMS reliably entered the IDLE state. All peripherals including the IMU, magnetometer, and magnetorquer control circuitry initialized successfully. LED indicators and telemetry packets confirmed system readiness.
- Ground Command and State Transition Tests
  Commands successfully transitioned the payload between operational states (e.g., IDLE → DETUMBLE → IDLE). The RTOS parsed incoming commands, updated

the internal state machine, and triggered the correct routines, including sensor logging and magnetorquer activation.

- Power Cycle Recovery Tests
  Power to the payload was abruptly cut while in various states. Upon restart, the system returned to the IDLE state, reinitialized peripherals, and awaited commands. This confirmed system robustness and fault-tolerant behavior.

- Data Logging and Downlink Tests
  Collected data was reviewed to verify accurate formatting and storage. The constructDownlink() function consistently produced correctly structured telemetry packets, suitable for downstream transmission over the simulated Iridium interface.

4.1.2.2.2 Helmholtz Cage and Air Bearing Testing

These tests validated full-system dynamic response in an environment simulating LEO magnetic and rotational conditions:

- Detumble Magnetorquer Activation Tests
  When in the DETUMBLE state, the control algorithm activates the X, Y, and Z magnetorquers in real time. PWM signals were generated based on angular velocity inputs, and sensor readings confirmed successful actuation and system responsiveness.

- PWM Response Tests
  PWM output varied dynamically in response to angular rate values from the filter and detumble logic. Signal profiles ramped accurately matching expected control behavior. Induced load remained within constraints.

- Sensor Feedback Verification
  Data from the IMU and magnetometer sensors was logged continuously. The Helmholtz cage enabled ground-truth validation of magnetometer accuracy, confirming system reliability in a controlled test environment.

4.1.2.3 Functional Test Procedure

To ensure a consistent and replicable detumble test, the air bearing and Helmholtz cage were standardized and each full system CHARMS test was performed with the following process:

1. Ensure air bearing is set to nominal pressure. Too much or too little airflow would lead to an unstable system or unnecessary friction, accordingly. **Figure 39** shows the standardized air pressure of the CHARMS air bearing compressor.

Figure 39. CHARMS air bearing pressures

2. Connect CHARMS to power. Since it would be impractical to power using an exact bus emulator, simply directly powering the 20-pin connector via a USB battery pack proved sufficient. The battery pack shown below in **Figure 40** has dual outputs, where each 5V from USB is broken out to a custom testing perf board. One output goes straight to CHARMS's 5V input and the other passes through a 5V -> 3.3V LDO before it goes to the 3.3V pin on CHARM.

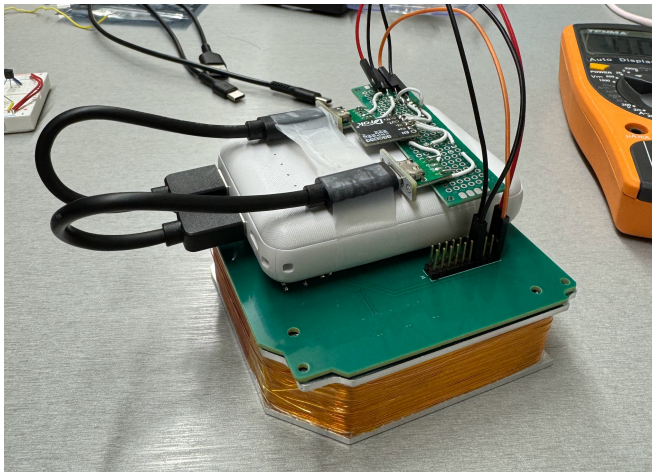

Figure 40. CHARMS with testing battery pack and power perf board

3. Pair CHARMS with the logging computer and ensure data is collected. This involves running a Python script that searches for the custom CHARMS hardware address. This script then begins to log data to a .CSV, where it can be plotted and analyzed. **Figure 41** outlines the data collected.

```
# Write CSV headers
with open(CSV_FILENAME, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow([
        "timestamp", "dutyX", "dutyY", "dutyZ",
        "magX", "magY", "magZ",
        "accX", "accY", "accZ",
        "gyroX", "gyroY", "gyroZ"
    ])
```

Figure 41. Data collected and logged over BLE

4. Center CHARMS on the air bearing, making sure that the center of mass is aligned with the center of the bearing. This is often difficult, since the low-friction air bearing is quite sensitive.

5. Place CHARMS and the air bearing in the center of the Helmholtz cage, since induced magnetic fields are only guaranteed at the origin of the cage. Previously, the cage was calibrated to the exact center, but the CHARMS payload with the air bearing is a bit shorter than this point. This meant that the Helmholtz cage had to be recalibrated, with the new origin being the point where the payload would sit. This ensures that CHARMS experiences the proper magnetic field and can react to an environment simulating the Earth's LEO field.

6. Enable the LEO simulation. This is controlled from the Helmholtz cage's computer, a Raspberry Pi 5. The cage uses three power supplies and a custom current driver circuit (similar to the one found on CHARM!) to ensure the proper current and field are induced in the cage.

7. Induce a spin. This is a procedure that took some practice, since early attempts to spin CHARMS added a lot of additional vertical variations. The best practice became spinning CHARMS to a higher than desired velocity, then slowing it down gradually until the desired rotation rate was logged over BLE.

8. Data was collected over the course of CHARMS's rotation and analyzed after each run.

4.1.2.4 Functional Test Adjustments

As expected, initial full-system test data did not initially meet the requirements set in the HLD. The CHARMS B-dot algorithm includes many adjustable constants to ensure proportional control over the response seen from the magnetorquers. These constants are used in the B-dot control algorithm to provide an accurate characterization of the inputs and ensure the level of processing is appropriate to induce the desired output. **Figure 42** shows a few constants used by the CHARMS control system.

```
#define NUM_MAGS
#define RESIS_AIR_MAG
#define RESIS_FERRO_MAG

#define DT
#define K
#define RELATIVE_PERM_MM
#define FERRO_LENGTH
#define FERRO_NUM_TURNS
#define FERRO_ROD_RADIUS
#define FERRO_AREA
#define FERRO_RADIUS
#define FERRO_DEMAG_FACTOR
#define FERRO_EPSILON

#define AIR_NUM_TURNS
#define AIR_AREA
#define AIR_EPSILON

#define FULL_VOLTAGE
```

Figure 42. Some IrishSatRTOS.h detumble constants

Many of these constants are representative of the physical CHARMS system and should not be adjusted, like the resistance of each magnetometer (used to calculate the amount of current and proportionally the torque response seen from each axis). However, the B-dot algorithm provides one particular constant, *k*, as a way to incorporate a detumbling constant gain. This constant was initially calculated using the expected dipole moment (found from the torquer physical characteristics) and the expected magnetic field (found from experimentally collected data) in the equation:

$$k = \alpha \cdot (m_{max} / \|\mathbf{B}\|).$$

In satellite detumble systems, this relationship is simply used as a starting point and an additional scaling constant, α, is often introduced to account for other forces present that may be difficult to analytically determine. Initial CHARMS runs used a k value of *1e-5*, but this small value led to long detumble times and low duty cycle utilization percentages even at high angular velocity. This means that CHARMS is not applying aggressive enough magnetic fields and the torques experienced were smaller than optimal. **Figure 43** shows an early detumble test, where the duty cycle (from 0 to 255) rarely reaches an optimal point. Note that due to the circular nature of a rotating field, the optimal value will oscillate from -255 to 255, but the infrequent peaks seen in **Figure 43** mean that k should be increased.

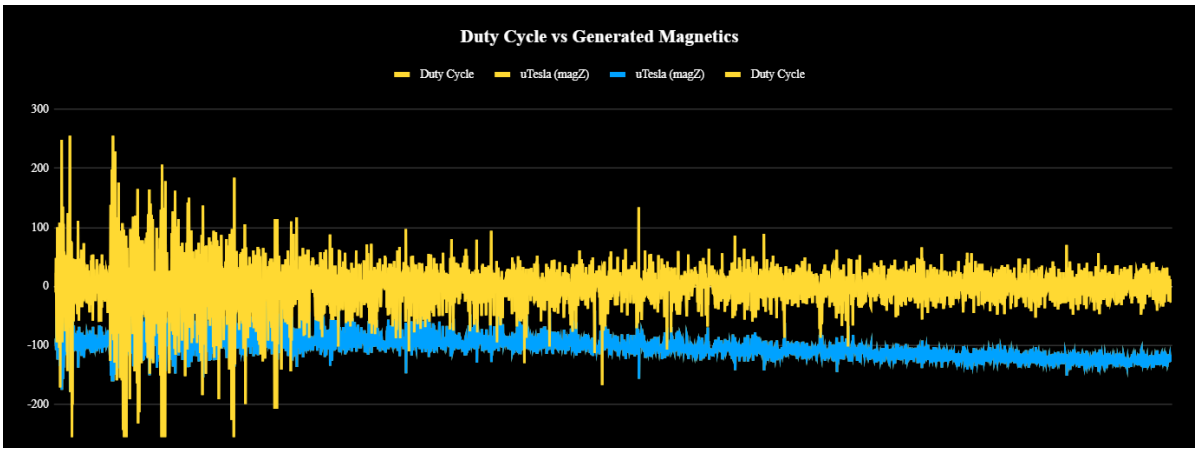

Figure 43. An early CHARMS detumble test result

In future runs, the k value was increased to 10e-5, meaning that the detumble product

$$m = -k\,(\omega \times B)$$

is ten times higher. This led to successful results and a faster detumble, as seen in **Figure 44** in the Functional Test Outcomes section below. Attempts to raise k even more (to ensure full power is applied to each magnetorquer for as long as possible) actually resulted in longer times to detumble as well as poorer power consumption, since any variation in gyroscope readings led to near-maximum duty cycle application and an unnecessarily large force being applied.

4.1.2.5 Functional Test Outcomes

All functional tests confirmed that the CHARMS payload met or exceeded its mission requirements for reliable actuation, sensor feedback, power efficiency, and software robustness. The system demonstrated stable performance across multiple subsystems and under flight-like environmental conditions.

4.1.2.5.1 Successful Autonomous Detumble in Simulated Orbit

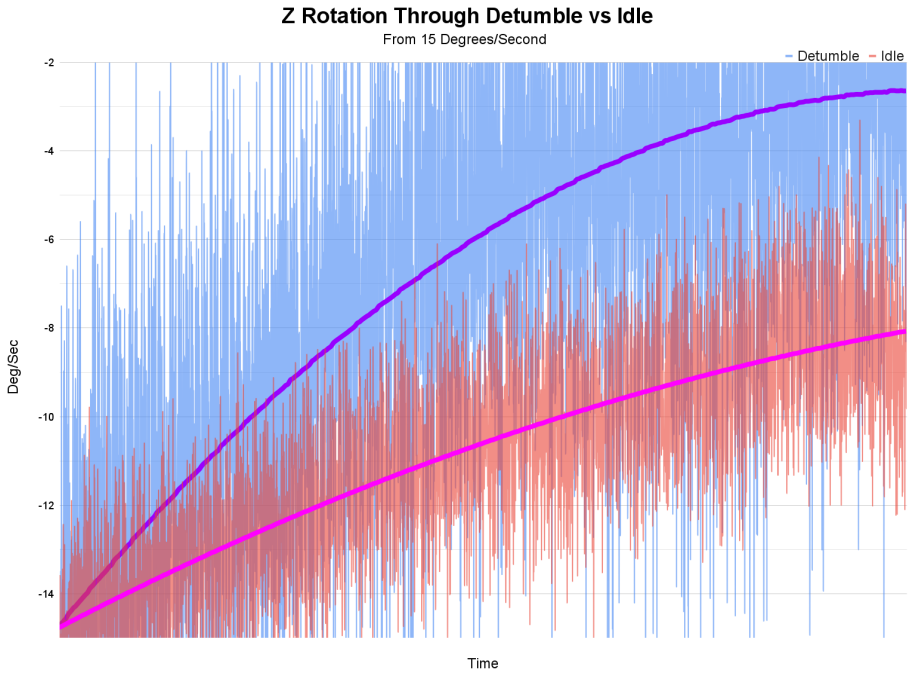

Figure 44. Detumble on Z axis of CHARMS vs IDLE System

In a closed-loop Helmholtz cage test replicating full LEO magnetic field conditions, CHARMS reduced angular velocity from 15°/s to approximately 2°/s in 17 minutes and 38 seconds. Total energy consumption during detumble was 0.2087 Wh, well below the 7 Wh performance threshold. This behavior directly confirmed functional magnetorquer actuation and closed-loop control execution.

4.1.2.5.2 Validated Control Algorithm and State Machine

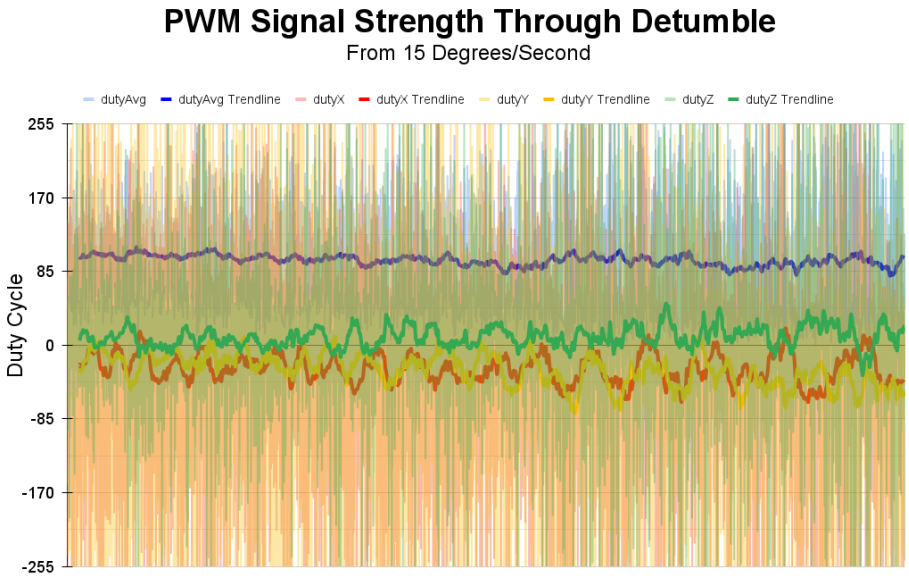

Figure 45. PWM Signal Strength of CHARMS Through Detumble

Real-time telemetry showed expected PWM outputs from the detumble controller in response to angular velocity and magnetic field measurements. All three magnetorquers were actuated using a gain-scaled B-dot control law implemented in the RTOS task scheduler.

4.1.2.5.3 Clear Separation from Passive Damping Behavior

A control-disabled comparison trial, where magnetorquers were unplugged but the system was otherwise identical, resulted in asymptotic decay to ~7°/s over 75+ minutes. The active control test achieved over 4x faster spin rate reduction, confirming that active magnetorquer control (not natural damping) was the stabilizing agent.

4.1.2.5.4 Robust Sensor Performance and System Feedback

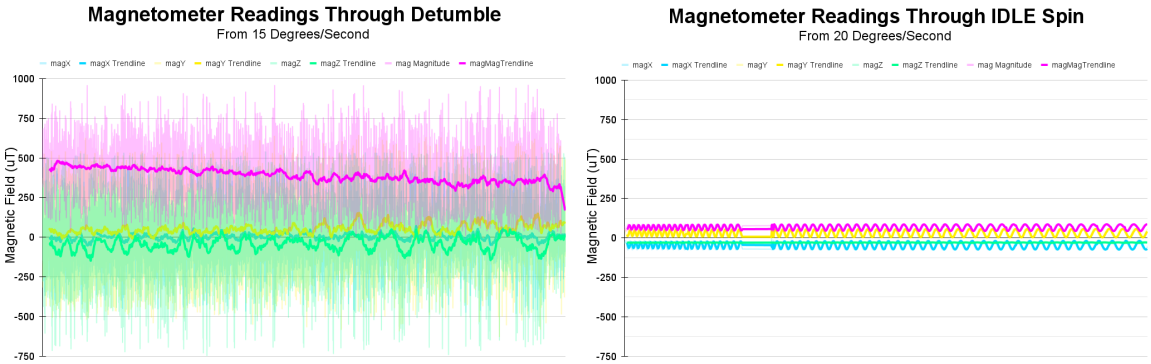

Figure 46. Magnetometer Readings onboard CHARMS Through Detumble

IMU, magnetometer, and IR sensors performed within expected ranges throughout testing. Logged sensor data closely aligned with predicted field inputs from PySol, and real-time graphs (**Figure 46**) demonstrated consistent measurement trends, including decreasing gyro magnitudes and B-field stabilization over time.

4.1.2.5.5 Power and Current Compliance

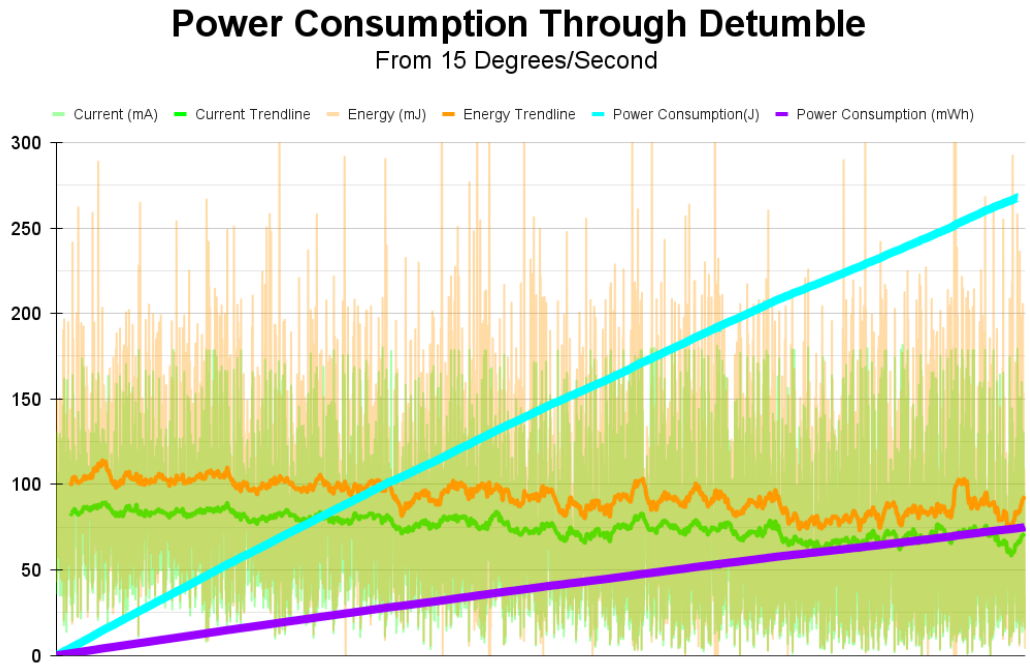

Figure 47. Power Consumption of CHARMS Through Detumble

Current draw remained well within NSL's 0.4 A rail limit throughout operation. PWM signals followed trendlines aligned with model expectations (**Figure 47**), and power analysis confirmed that thermal and electrical budgets were not exceeded. **Figure 47** illustrates a stable current trend of ~60–100 mA, with energy accumulation peaking under 0.21 Wh.

4.1.2.5.6 Stable Re-Initialization After Power Loss

The payload reliably re-entered IDLE mode after each abrupt power interruption. All peripherals re-initialized without failure, validating the system's fault-tolerant behavior and confirming that no undefined states or corruption occurred due to power loss.

4.1.2.5.7 Consistent Telemetry and Logging

Data packets were correctly generated, formatted, and transmitted, both in emulator-based tests and real-time trials. Log files and BLE terminal output included full sensor snapshots, state transitions, and control inputs with accurate time correlation.

4.1.2.5.8 Flight-Ready Integration and Bus Compatibility

End-to-end validation with NSL-Valparaiso emulator board and bus emulation power hardware confirmed that CHARMS is compatible with NSL's packet structure, command format, mechanical mounting, and electrical interface. The system responded appropriately to NSL-style inputs including power, bus activity, and signal commands.

The CHARMS team will repeat tests conducted previously after environmental qualification to ensure post-TVAC and vibration survivability.

The functional testing outcomes, supported by visualizations in **Figures 39-47**, show that CHARMS is not only functionally complete, but mission ready. System behavior aligned tightly with simulations, validating the design approach and confirming the efficacy of the onboard control system under real testbed conditions.

4.1.2.6 Extended Functional Test Outcomes

To further showcase CHARMS's readiness, the system was operated continuously during a 3-hour live demonstration on Demo Day using the full hardware-in-the-loop setup. Over this extended period, the payload:

- Successfully performed ten partial autonomous detumble cycles, each beginning from high initial angular velocities and multiaxis rotational wobble.
- Demonstrated proportional PWM intensity scaling, as shown in real-time telemetry, with actuator response closely matching expected angular deceleration profiles.
- Maintained strong and consistent magnetic dipole output, evidenced by the magnetometer readings, which showed clear directional field generation corresponding to commanded actuation.
- Retained stable power consumption, with no performance degradation or overdraw despite extended operation.
- Maintained unbroken communication and telemetry output, with valuable sensor data and command acknowledgments logged throughout the full session.

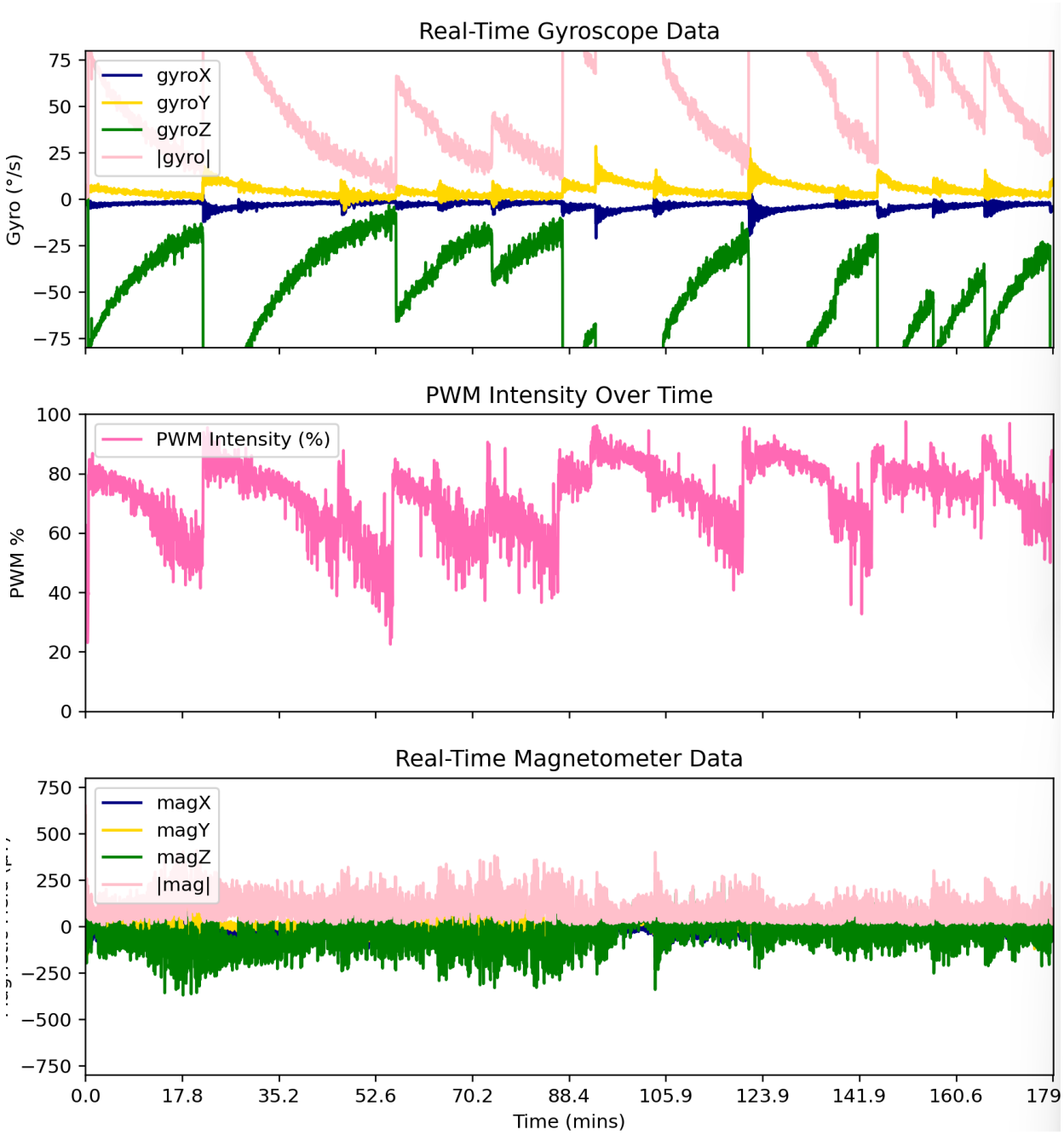

Figure 48. 3 Hour Demonstration of Detumble Operations Results

This demonstration not only reaffirmed subsystem durability but also emphasized the repeatability and endurance of CHARMS under sustained control loop stress, adding further confidence in its flight robustness.

4.2 Alignment of Testing Data with Design Requirements

Objective	Target Specification	CHARMS Performance	Status
Detumbling Success	Reduce spin to < 0.5°/s	~2°/s achieved after 15°/s initial	Partial Success
Detumbling Time	< 75 minutes	17 minutes minimum	Exceeded
Precise Actuation	Produce expected B-field from magnetorquers	~4000 magnitude achieved	Met
Sensor Suite Performance	Accurate magnetometer and gyro readings	All axes accurately read, decreasing ω detected	Met
Dipole Strength (MuMetal)	> 5 Am²	~1.06 Am²	Partially Met
Dipole Strength (Air Core)	> 0.1 Am²	~1.67 Am²	Exceeded
Power Consumption	< 7 Wh	~0.2087 Wh during detumble	Exceeded
PWM Control	PWM scaled based on sensor feedback	Gain-varying PWM confirmed	Met
Autonomous Operation	Self-contained RTOS + control architecture	Fully autonomous detumble mode	Met
Power Efficiency	Operates under tight energy budget	~0.2087 Wh detumble (<< 7 Wh)	Exceeded

As iterated above, CHARMS met or exceeded nearly all of the initial requirements. Above are the testing-related metrics of success. One notable failed metric is the <0.5 degrees/second spin rate. This can be attributed to the imperfect proportion-based control system, the low-cost and noisy (even after calibration) gyroscope, and the imperfect air bearing potentially introducing some external torque from differing pressure zones. Overall, CHARMS's testing process proved overwhelmingly successful.

5 Installation Manual for Mission Operations

The CHARMS payload was designed for ease of use, with simple construction, installation, and verification in mind. This allows customers to buy a CHARMS module and have confidence in its serviceability and functionality.

5.1 Payload Assembly

The CHARMS payload assembly process is relatively simple and can be completed in an hour or less. This simple construction also allows easy maintenance access. The assembly kit will have 6 main components: the CHARMS PCB, the air core magnetorquer, the aluminum top plate, 2 rod magnetorquers, and several mechanical clamps to hold the rod magnetorquers and wire assemblies within the payload. Pictures of the full payload are included in **Figure 49** below.

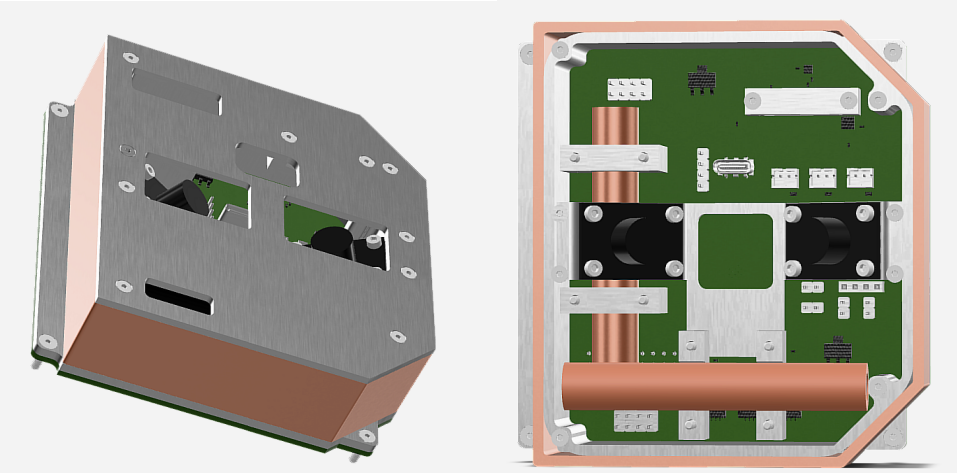

Figure 49. Isometric top view (left) and internal assembly (right) of the CHARMS payload

Step 1 of Payload Assembly – PCB Attachment

Since the part selection ensured the use of commonly used components, the PCB fabrication and assembly house will fully assemble all PCBs for the CHARMS payload.

The CHARMS payload has 8 mounting screws which are to be attached through the PCB: 4 screws to hold the PCB into the air core magnetorquer structure and 4 screws to integrate mechanically with the rest of the CubeSat. In this prototype's case, the mechanical integration happens with the NSL bus to complete the 0.5U satellite. The attachment points are highlighted below in **Figure 50**. The red circles indicate mechanical integration points with external systems. The blue circles indicate mechanical integration with the air core structure.

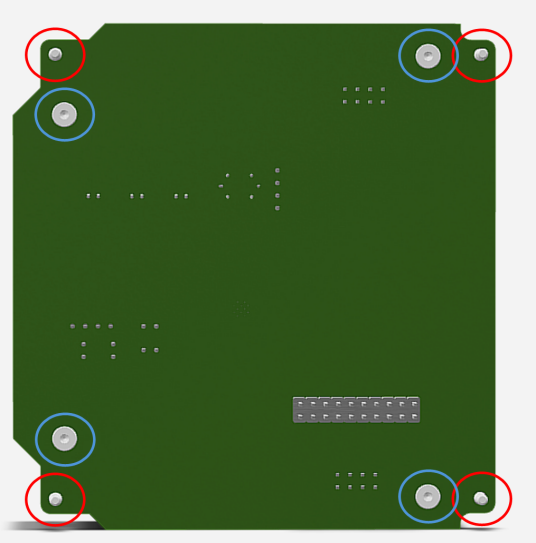

Figure 50. Mechanical integration of CHARMS PCB with air core structure (bottom view)

Step 2 of Payload Assembly – Magnetorquer Attachment and Wiring

The next step of CHARMS assembly is to install and wire the magnetorquers. The rod magnetorquers, which are internal to the payload, are held in place by mechanical clamps that attach to the side wall of the air core structure. **Figure 51** shows the location of these clamps for the proper installation of the X and Y axis magnetorquers.

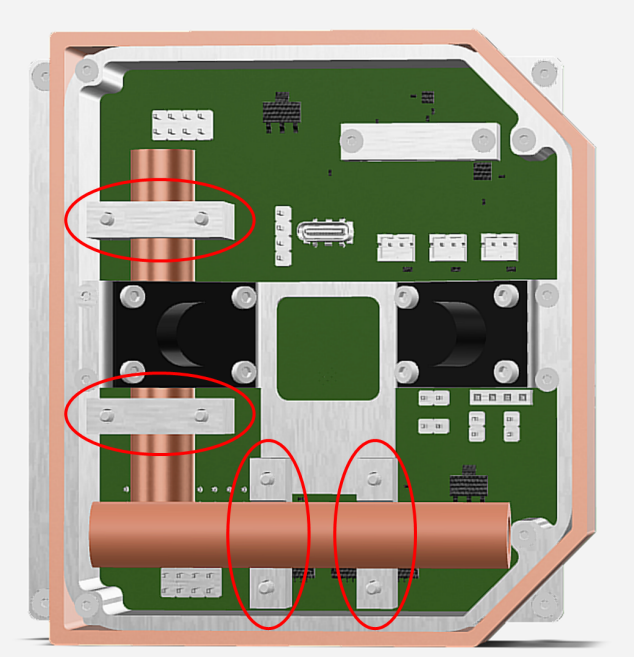

Figure 51. Magnetorquer structural clamps (top view, no top plate)

The next step is to wire the X, Y, and Z axis magnetorquers using the JST connectors circled in red in **Figure 52**. These wires coming from the JST connectors are then clamped using the

mechanical clamp circled in blue, which is attached to the top plate using screws. Do not attach the top plate yet. This completes the magnetorquer assembly.

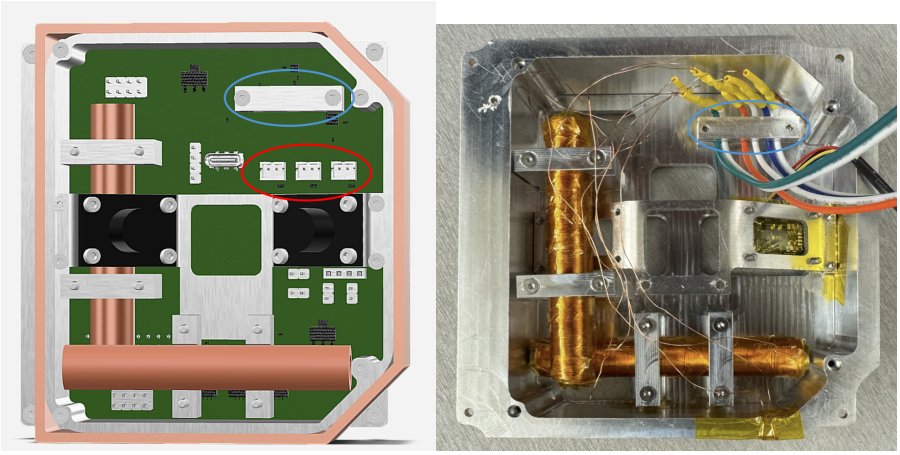

Figure 52. Magnetorquer wiring clamp (blue) and JST connectors (red)

Step 3 of Payload Assembly – IR Camera Installation

Next, the IR camera brackets and the IR camera will be assembled and attached to the top plate, similar to the wire clamp. First, screw in the cameras to the aluminum camera bracket, as seen in **Figure 53**. After securing the cameras to the IR camera bracket, plug in the JST connector in to connect them to the I2C bus, circled in blue underneath the magnetorquer. This figure is repeated from **Section 3.6 Detailed Design and Operation of Subsystem 4: Structures** for reader ease.

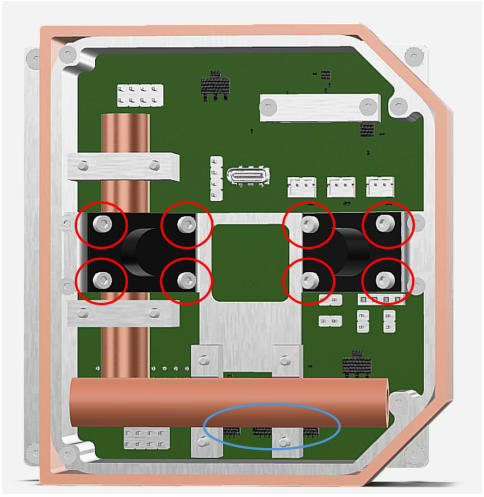

Figure 53. Attachment of IR cameras using screws (red) and JST board connection (blue)

After securing the cameras to the IR camera bracket and plugging in the IR cameras, attach the IR camera bracket to the top plate, circled in red in **Figure 54**. Circled in blue is the mechanical clamp for the wiring. This figure is also repeated from **Section 3.6 Detailed Design and Operation of Subsystem 4: Structures** for reader ease.

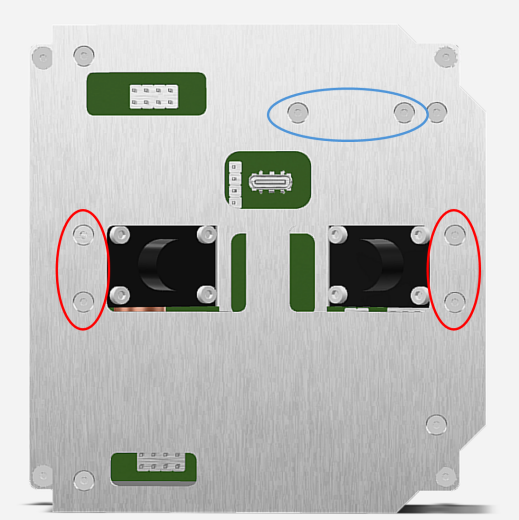

Figure 54. Attachment of IR cameras brackets (red) and wire clamp (blue) to top plate

Step 4 of Payload Assembly – Secure the Top Plate

The final part of the CHARMS assembly process is the attachment of the top plate. The top plate has 4 screws to secure it to the air core structure, as seen in **Figure 55**.

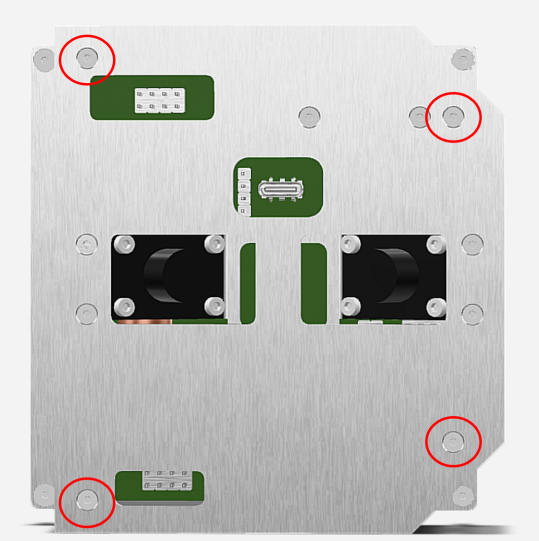

Figure 55. Attachment of top plate to the air core structure

After this step, the CHARMS module is fully assembled and ready for testing and operation.

5.2 Installation

Installation of the CHARMS payload is designed to be extremely easy, using one 20-pin connector which extrudes from the bottom of the CHARMS payload as seen in **Figure 56**. This bus connector includes external 3.3V, 5V, and 6-9V power rails, UART serial Tx and Rx pins, Ready-To-Send (RTS) and Clear-To-Send (CTS) signalling pins, and 3 analog GPIO pins for powering and information exchange between the CHARMS MCU and the external system.

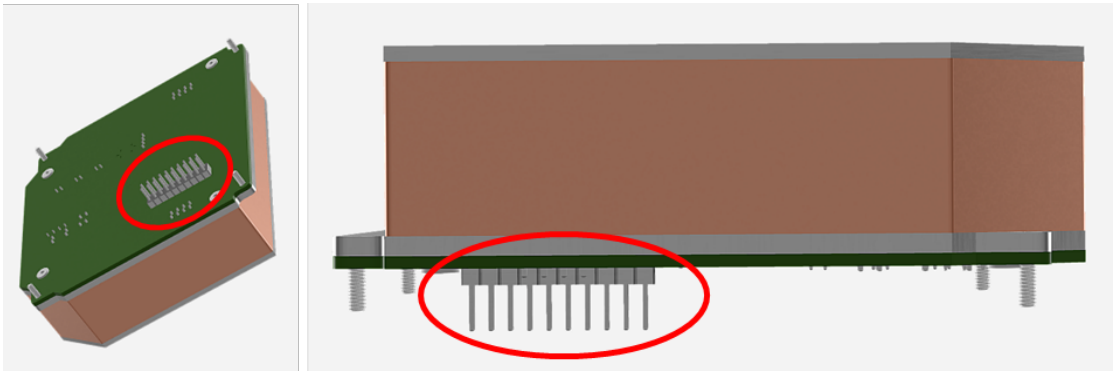

Figure 56. 20-pin bus connector for external power and communication with CHARMS

Figure 57 shows a top view of the CHARMS 20-pin connector with pin numbers visible on the silkscreen of the PCB. **Table 14** shows the pinout assignments for the CHARMS payload.

Table 14. 20-pin bus connector pinout assignments [1]

Pin	Payload Pin	Dir	Bus Pin	Required?	Notes
1	Power 1.A	←	Switch 1.A	x (at least 1 of 2 for primary)	3.3v
2	Power 1.B	←	Switch 1.B		5v
3	Power 2	←	Switch 2		3.3v
4	Power 3	←	Switch 3		5v
5	Power 4	←	Switch 4		BUSS+ 6-9v
6	Ground	-	Ground	x	
7	Ground	-	Ground	x	
8	Ground	-	Ground	x	
9	RTS	→	RTS	x	5v active-high
10	CTS	←	CTS	x	5v active-high
11	Payload TX	→	Bus RX	x	5v 8N1 38400
12	Payload RX	←	Bus TX	x	5v 8N1 38400
13	Ground	-	Ground	x	
14	Ground	-	Ground	x	
15	Analog Output0	→	Analog Input0		0-5v
16	Analog Output1	→	Analog Input1		0-5v
17	Analog Output2	→	Analog Input2		0-5v
18	Analog Output3	→	Analog Input3		0-5v
19	NC				
20	NC				

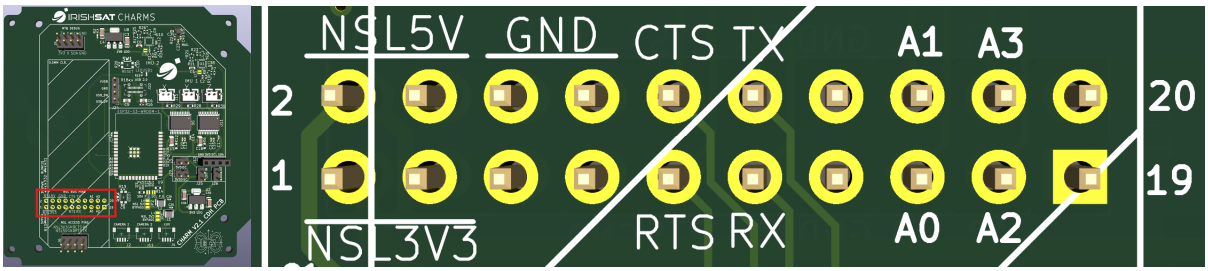

Figure 57. 20-pin bus connector pin numbering

5.3 Initialization

To initialize the payload, connect the 20-pin connector to the external system, which is configured properly to externally power and communicate with CHARMS. When the board is being powered, you can now reprogram CHARMS if you desire. Use the accessibility holes in the top plate to access the USB for reprogramming, as well as the signal line header breakouts for debugging. See **Figure 58** to locate accessibility holes.

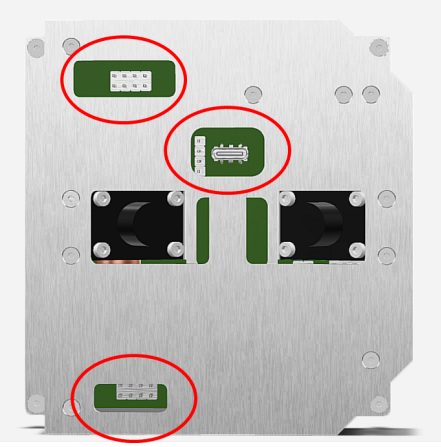

Figure 58. CHARMS accessibility holes in top plate for USB and signal probing

After CHARMS is running the proper software, the USB connection is no longer needed. CHARMS will solely run off the 20-pin connector.

5.4 Verification and Troubleshooting of System Functionality

To verify that CHARMS is working properly, the accessibility holes seen in **Figure 58** are very helpful. The USB can be utilized for serial monitor debugging. Add printouts throughout the code to verify that everything is working as expected. Additionally, connecting a digital analyzer to the signal breakout headers or probing voltage breakouts with a multimeter will confirm CHARMS's functionality. The pinouts for the top header breakout, circled in red, and the bottom header breakout, circled in blue, can be found in **Figure 59**.

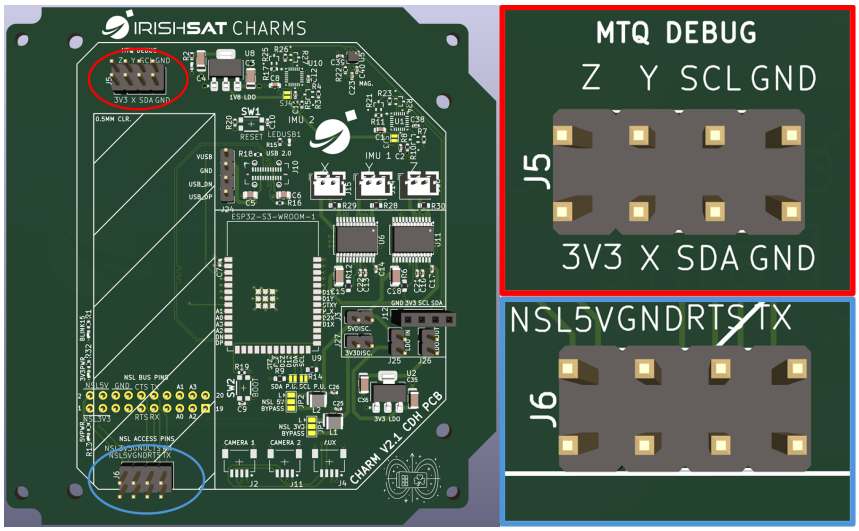

Figure 59. CHARMS signal probing pinouts

Beyond probing, verifying the functionality of CHARMS can be tested by gathering empirical data. First, set up a Helmholtz cage to run PySol, a program that simulates the magnetic fields that a satellite would experience during one orbit in LEO. Place CHARMS inside the cage on top of an air bearing and upload Bluetooth-enabled code for wireless data transmission. This will allow you to gather and plot testing data of CHARMS during its detumble state. This empirical testing setup can be seen in **Figure 60**. See more information in **Section 4.1.2 Functional Testing**. This figure is repeated from **Section 4.1.1.4 Environmental Testing** for reader ease.

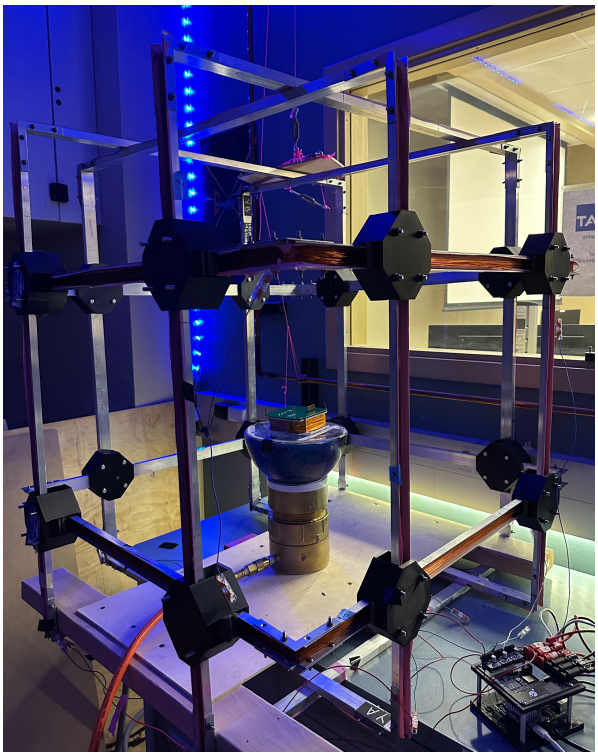

Figure 60. Empirical testing setup for verification of CHARMS functionality

6 Design Changes Required for Commercial Marketability

- As the CHARMS team wraps up the prototype and testing of this magnetorquer-only ADCS, there are a few design changes that would improve the commercial viability of the product for other CubeSat missions.

- Observed in testing results, we find that our system performance decreases significantly once the spin rate reduces down to around 2 degrees per second on all three axes. We believe that this reduction in performance is due to many factors which are described in detail in **Table 15**.

- **Table 15**. Design changes to improve CHARMS performance in future iterations

Current Design	Design Improvement
Proportional Controller	CHARMS could employ significantly better controls which take into account previous states to guide future actuation, something that a proportional controller does not do. For example, CHARMS could use a Proportional Integral Derivative (PID) controller which would result in more accurate and effective actuation.
Low-Pass Filtering	CHARMS could use a more advanced filtering technique called Kalman filtering to improve sensor data. With noisy sensor data heavily impacting system performance, improving our filtering algorithm would lead to better system performance.
Low-Cost Sensor Suite	CHARMS could use higher accuracy, lower noise sensors on the PCB. This would have a similar effect over performance that improving our filtering would have.

Additionally, ADCS on CubeSats are expected to do more than just detumble the satellite. ADCS are also expected to be able to do some sort of pointing, especially sun pointing, which allows the satellite to carry out critical functions. For example, sun pointing is a required function of a satellite ADCS because the satellite needs to charge its batteries using its solar panels. Without sun pointing, the satellite quickly loses power and becomes inoperable.

Other pointing algorithms like Earth pointing are very helpful for satellites doing Earth communication and Earth science. This is another pointing algorithm which will need to be developed for the CHARMS payload for it to become a viable commercial product.

One other design improvement which could improve CHARMS commercial viability is reduction of the module size to take up less space on the satellite. In small satellites like CubeSat, space is hard to come by when designing. Smaller modules allow you to improve other critical bus systems as well as increasing the available payload space for research and technology demonstrations. This can be achieved by reducing the size and wrappings in our magnetorquer

design, trading speed and strength for reduced profile. If ADCS is able to detumble and sun point before running low on power, having a small module would improve our marketability.

7 Conclusions

- The CHARMS (Control of Hardware Attitude using Reliable Magnetorquers Satellite) payload represents a major milestone in accessible, autonomous attitude control for small satellite platforms. Developed to operate within the minimal space and power constraints of an 0.5U Cubesat, CHARMS successfully represents a technology demonstration of a magnetorquer-only GPS-free ADCS architecture - the first of its kind at this price point.

Through design, simulation, validation, and testing, the CHARMS team delivered a plug-and-play system that integrates a control and data handling PCBA, a custom real time operating system, and in-house machined and wrapped magnetorquers. The system achieved angular velocity reductions from 15 degrees/second to 2 degrees/second in just 17 minutes, consuming less than 0.21 Wh, surpassing expectations for both energy efficiency and actuation capability.

Extensive hardware-in-the-loop testing validated sensor performance, firmware reliability and robustness, and closed-loop full system control under accurate Earth LEO simulated conditions. These results prove the validity of a proportion-based B-dot control algorithm and prove CHARMS's viability as a standalone detumbling system.

CHARMS lays the groundwork for much more capability beyond detumbling, including sun and nadir pointing algorithms with the already-included IR cameras. With specific hardware and software improvements like higher fidelity sensors and more advanced sensor filtering, CHARMS could eventually evolve into a fully functional, general purpose ADCS for commercial scalability and deployment.

Most importantly, CHARMS embodies the broader mission of democratizing space. It proves that reliable ADCS functionality can be achieved at a fraction of the power and cost of commercial systems, meeting its technical goals and also providing hands-on experience to the CHARMS team with real-world flight hardware and systems engineering. As a plug-and-play, low cost, low power, open-architecture ADCS, CHARMS promises to be a launch-ready solution for educational programs, low-budget smallsats, and commercial tech demonstrators seeking low-cost, effective detumbling.

8 Appendices

Appendix A – Modified IRR Tracker Document

Non-critical columns were removed from the IRR tracker document and placed here for readability. The full IRR Tracker document is linked here and also provided on request from the CHARMS team..

Requirement Reference & Documentation	Test Complete (Requirement Met)	Responsible Engineer(s) In Charge of Req.
R1.1	Final cleaning will take place after post-env. testing	Jackson O'Neill
R1.2	TRUE	Aidan, Peter
R1.3	TRUE	Peter Gibbons
R1.4	TRUE	Peter Gibbons
R1.5	TRUE	Peter Gibbons
R1.6	???	Jackson O'Neill
R1.7	TRUE	Peter, Aidan, Sarah
R1.8	20 pin connector black spacers are 1 mm out of spec.	Jackson O'Neill
R1.9	Waiting for correct parts to arrive	Jackson O'Neill
R1.10	TRUE	Peter Gibbons
R1.11	Full assembly almost complete... almost certainly will meet this req	Jackson O'Neill
R2.1	TRUE	Aidan
R2.2	TRUE	Aidan
R2.3	In-rush too high, next iteration will have filtering inductor	Aidan, Peter
R2.4	TRUE	Aidan, Peter
R2.5	TRUE	Aidan, Peter
R2.6	TRUE	Aidan, Peter
R2.7	Next iteration of board will contain all test points. Ordering board in a couple of days	Aidan
R2.8	TRUE	Peter Gibbons, Aidan
R2.9	TRUE	Aidan
R2.10	TRUE	Aidan
R2.11	TRUE	Isaac
R2.12	Full functional test for verifying power consumption happening in next week	isaac
R2.13	???	Sarah, Isaac
R3.1	TRUE	Isaac

R3.2	**Ongoing**: Still need to complete this SW implementation	Isaac
R4.1	**Ongoing**: Will be completed withing 3 days of IRR	Isaac
R4.2	**Pending**: Can be done whenever ready	Isaac
R4.3	**Pending:** Done whenever TVAC happens	Isaac

Appendix B – CHARMS BOM Document

This is a modified BOM which significantly reduced the number of column so that it only contains limited information. This is done so that some form of the BOM could be represented on this document. To view to full BOM, go here.

Name/PN	Description	Qty (#)	Part Mass (g)
CHARMS	**Complete Integrated Payload**	**1**	**N/A**
Chassis Assembly	**Structural Assembly of Payload**	**1**	**N/A**
N1-1-008_A	Payload Roof Plate	1	48
N1-1-007_A	Air Core Shell	1	68.1
N1-1-011_A	Camera Mount Bracket	1	7.1
N1-1-006_A	Magnetorquer Mounting Bracket	2	3.2
N1-1-006_B	Magnetorquer Mounting Bracket	2	1.5
N1-1-015_A	Magnetorquer Mounting Bracket	2	3.1
N1-1-003_A	Magntorquer Rod Core	2	19.7
N2-1-002_A	Magnetorquer Rod Wire Segments	2	28.35
N2-1-003_A	Air Core Wire Segment	1	56.7
N1-1-016_A	Magnetorquer Wire Clamp	1	0.6
N1-1-017_A	Magnetorquer Wire Clamp	1	0.5
90318A411	M2 Shoulder Bolts for PCB Mounting	4	0.04
91292A831	M2 x 6mm Socket Head for Camera Mounting	8	0.02
92125A052	M2 x 6mm CS	8	0.02
92196A078	#2-56 x 5/16" Socket Head for Payload Mounting	4	0.03
92125A611	M2 x 20mm CS	4	0.06
92125A059	M2 x 16mm CS	4	0.05
92125A056	M2 x 10mm CS	2	0.03
MLX90640	Infared Camera	2	3
PCB Asm 1	**Primary PCB Assembly**	**1**	**N/A**
C10,C9	1uF	2	
R9,R14	4.7k	2	
R7,R23,R11,R26,R3,R17,R10,R5	2.2k	8	
C17,C38,C16,C7,C14,C2,C11,C40,C13,C12	0.1uF	10	
J11,J4,J2	JST_SH_4	3	
C8,C1	0.1uF	**2**	

R19,R22,R20	10k	3	
J9,J27,J25,J26	Conn_01x02_Pin	4	
C21,C22	10uF	2	
SJ1,SJ2,SJ3,SJ4	DNP	4	
R29,R28,R30	0 ohm	3	
R6,R12	10k	2	
C4,C36	22uF	2	
R25,R24,R4,R27,R8,R21	RE1C002UNTCL (Mosfet)	6	
C35,C3,C15,C18	10uF	4	
J12	Conn_01x04_Socket	1	
J16	JST_PH_B2B	1	
U8	AMS1117-1.8 (LDO)	1	
J10	USB_C_Receptacle	1	
J24	Conn_01x04_Pin	1	
J15	JST_PH_B2B	1	
C5,C6	100n	2	
U2	AMS1117-3.3 (LDO)	1	
R2,R1,R32,R13	1k	4	
U9	ESP32-S3-WROOM-1-N16R2	1	
U5	LIS2MDL (Magnetometer)	1	
C39	220nF	1	
R16,R18	5.1k	2	
LED4	Red LED	1	
LED1,LED5	Green LED	2	
U1,U10	ICM-20948 (9-Axis Accelerometer)	2	
SW2,SW1	KMR231GLFS (Button)	2	
U11,U6	TB6612FNG (H-bridge)	2	
J14	JST_PH_B2B	1	
C23	10uF	1	
LED2	Blue LED	1	
J1	Conn_02x10_Odd_Even	1	
L1, L2	10uH	2	

Appendix C – IrishSat_RTOS_main.cpp Code Listing

```
#include "IrishSatRTOS.h"

TaskHandles taskHandles;

QueueHandle_t iridiumDownlinkBufQueue;
QueueHandle_t downlinkBufQueue;
QueueHandle_t uplinkBufQueue;
QueueHandle_t sensorDataQueue;

HardwareSerial Serial3(2);
Adafruit_MLX90640 mlx;
ICM_20948_I2C myICM1;
ICM_20948_I2C myICM2;
Adafruit_LIS2MDL lis2mdl = Adafruit_LIS2MDL(12345);
sensors_event_t event;

StateVars RTOSstateVars = { {100},                      // State Machine
Delay
                            {100 / portTICK_PERIOD_MS}, // State Machine
Delay Ticks
                            {100},                      // Poll Sensor Delay
                            {100 / portTICK_PERIOD_MS}, // Poll Sensor Delay
Ticks
                            {1000},                      // Poll Camera Delay
                            {1000 / portTICK_PERIOD_MS}, // Poll Camera Delay
Ticks
                            {100},                      // Interpret Uplink
Delay
                            {100 / portTICK_PERIOD_MS}, // Interpret Uplink
Delay Ticks
                            {100},                      // Detumble Delay
                            {100 / portTICK_PERIOD_MS}, // Detumble Delay
Ticks
                            {100},                      // Send and Ack Delay
                            {100 / portTICK_PERIOD_MS}, // Send and Ack Delay
Ticks
                            {100},                      // Monitor Commands
Delay
                            {100 / portTICK_PERIOD_MS}, // Monitor Commands
Delay Ticks
                            {60000},                     // Iridium Request
Time
                            {5000},                      // Uplink Request
Time
                            {15000},                     // Iridium High
Frequency Packet Creation Time
                            {180000},                    // Iridium Low
Frequency Packet Creation Time
```

```
                              {0},                          // Packet Sequence
Number
                              {0},                          // Detumble Sequence
Number
                              {0},                          // RTOS Start Time
                              {START_UP},                   // Current State
                              {-1},                         // Previous State
                              {true},                       // New State?
                              {true},                       // Start Up?
                              {true},                       // Poll Sensors?
                              {false},                      // Poll Cameras?
                              {false},                      // Detumble?
                              {false},                      // Flip X Voltage?
                              {false},                      // Flip Y Voltage?
                              {false},                      // Flip Z Voltage?
                              {false},
                              {false} };                    // Use IMU2?

sensorOffsets sensorOffsetBoard2 = {
  //lis2mdl
  {-17, -20,  42.21},     // Mag X,Y, and
  //ICM1
  { {0.346 , -0.379, -0.117},   // Gyro X,Y, and Z
  {17.828, -8.817,  6.472},   // Accel X,Y, and Z
  {-15.27,  21.58,  31.54}  }, // Mag X,Y, and Z
  //ICM2
  { {0.973 , -0.576, -0.518},   // Gyro X,Y, and Z
  {28.249, -0.630,  29.002},   // Accel X,Y, and Z
  {6.76 ,  5.32,  57.80} }, // Mag X,Y, and Z
  //lis2mdl soft iron
  { {0.965 ,  0.012, -0.007},
    {0.012 ,  1.040, -0.014},
    {-0.007, -0.014,  0.997} },
  //ICM1 soft iron
  { {1.071 , -0.009,  0.028},
  {-0.009,  0.986, -0.013},
  {0.028 , -0.013,  0.948} },
  //ICM2 soft iron
  { {1.025 , -0.015,  0.014},
  {-0.015,  0.982,  0.006},
  {0.014 ,  0.006,  0.994} },
};

tempOffsetCalcHolder corrValStruct;

// the setup function runs once when you press reset or power the board
void setup() {

  delay(2000);

  Serial.begin(38400); // to the computer
```

```
  Serial3.begin(38400, SERIAL_8N1, RX_PIN, TX_PIN); // to the NSL FC
  Serial.read(); // Clear Serial Bus
  Serial3.read(); // Clear Serial3 Bus

  // Initialize I2C on specified pins
  Wire.begin(SDA_PIN, SCL_PIN);

  Serial.println("Serial Bus Initialized.");

  pinMode(CTS_PIN, INPUT); // Clear-to-Send pin set high or low by NSL FC
  pinMode(RTS_PIN, OUTPUT); // Request-to-Send pin set high or low by
IrishSat
  digitalWrite(RTS_PIN, LOW); // Initial state LOW

  // Set PWM Pin States
  pinMode(DIR1X, OUTPUT);
  pinMode(DIR2X, OUTPUT);
  pinMode(PWMX, OUTPUT);
  digitalWrite(DIR1X, LOW);
  digitalWrite(DIR2X, HIGH);

  pinMode(DIR1Y, OUTPUT);
  pinMode(DIR2Y, OUTPUT);
  pinMode(PWMY, OUTPUT);
  digitalWrite(DIR1Y, LOW);
  digitalWrite(DIR2Y, HIGH);

  pinMode(DIR1Z, OUTPUT);
  pinMode(DIR2Z, OUTPUT);
  pinMode(PWMZ, OUTPUT);
  digitalWrite(DIR1Z, LOW);
  digitalWrite(DIR2Z, HIGH);

  //Serial.println("Pin Modes Set.");

  if (! mlx.begin(MLX_ADDR, &Wire)) {
    Serial.println("MLX90640 not found!");
    while (! mlx.begin(MLX_ADDR, &Wire)) delay(10);
  }

  if (! myICM1.begin(Wire, IMU1_AD0_VAL)) {
    Serial.println("ICM20948-1 not found! Trying Again...");
    while (myICM1.status != ICM_20948_Stat_Ok){
      myICM1.begin(Wire, IMU1_AD0_VAL);
      delay(500);
    }
  }

  if (! myICM2.begin(Wire, IMU2_AD0_VAL)) {
    Serial.println("ICM20948-2 not found! Trying Again...");
    while (myICM2.status != ICM_20948_Stat_Ok){
```

```
    myICM2.begin(Wire, IMU2_AD0_VAL);
    delay(500);
  }
 }

 /* Enable auto-gain */
 lis2mdl.enableAutoRange(true);
 if (!lis2mdl.begin(0x1E, &Wire)) {  // I2C mode
  Serial.println("Ooops, no LIS2MDL detected ... Check your wiring!");
  while (!lis2mdl.begin(0x1E, &Wire)){
    delay(500);
  }
 }

 Serial.println("Found MLX90640, ICM20948-1, ICM20948-2, LIS2MDL");

 iridiumDownlinkBufQueue = xQueueCreate(200, sizeof( uint8_t
)*UART_PACKET_SIZE);
 downlinkBufQueue = xQueueCreate(10, sizeof( uint8_t )*UART_PACKET_SIZE);
 uplinkBufQueue = xQueueCreate(10, sizeof( uint8_t )*UART_PACKET_SIZE);
 sensorDataQueue = xQueueCreate(1, sizeof( SensorData ));

 //Serial.println("Queues Created.");

 // Now set up two tasks to run independently.
 xTaskCreate(
   TaskStateMachine
   ,  "State Machine"
   ,  16384  // Stack size
   ,  NULL
   ,  1  // Priority
   ,  &taskHandles.taskStateMachineHandle );

 //Serial.println("State Machine Created.");


 xTaskCreate(
   TaskPollSensors
   ,  "Poll Sensors"
   ,  16384  // Stack size
   ,  NULL
   ,  2  // Priority
   ,  &taskHandles.taskPollSensorsHandle );

 //Serial.println("Poll Sensors Created.");

 xTaskCreate(
   TaskPollCameras
   ,  "Poll Cameras"
   ,  16384  // Stack size
   ,  NULL
```

```
    ,  3  // Priority
    ,  &taskHandles.taskPollCamerasHandle );

  //Serial.println("Poll Sensors Created.");

  xTaskCreate(
    TaskInterpretUplink
    ,  "Interpret Uplink"
    ,  16384  // Stack size
    ,  NULL
    ,  4  // Priority
    ,  &taskHandles.taskInterpretUplinkHandle );

  //Serial.println("Interpret Uplink Created.");

  xTaskCreate(
    TaskDetumble
    ,  "Detumble with bcross"
    ,  16384  // Stack size
    ,  NULL
    ,  5  // Priority
    ,  &taskHandles.taskDetumbleHandle );

  //Serial.println("Detumble Created.");

  xTaskCreate(
    TaskSendAndAck
    ,  "Send and Ack"   // A name just for humans
    ,  4096  // This stack size can be checked & adjusted by reading the
Stack Highwater
    ,  NULL
    ,  6  // Priority, with 3 (configMAX_PRIORITIES - 1) being the highest,
and 0 being the lowest.
    ,  &taskHandles.taskSendAndAckHandle );

  //Serial.println("Send and Ack Created.");

  xTaskCreate(
    TaskMonitorCommands
    ,  "Monitor Commands"
    ,  16384  // Stack size
    ,  NULL
    ,  7  // Priority
    ,  &taskHandles.taskMonitorCommandsHandle );

  //Serial.println("Monitor Commands Created.");

  //Serial.println("Setup Complete!");

  // Now the task scheduler, which takes over control of scheduling
individual tasks, is automatically started.
```

```
}

void loop()
{
  // Empty. Things are done in Tasks.
}

void TaskStateMachine(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  uint8_t downlinkPacket[UART_PACKET_SIZE]; // Used as buffer to send stuff
to NSL FC
  uint8_t uplinkPacket[UART_PACKET_SIZE]; // Used as buffer to send stuff to
NSL FC
  uint8_t netReqCmd = 0x47;
  uint8_t checkUplinkCmd = 0x48;
  uint64_t loopCounter = 0;
  int numBytesAvailable;
  SensorData sensorData; // Used as buffer to send stuff to NSL FC

  float pixel;

  unsigned long previousTime; //track how long you are in the current state
  unsigned long switchTime = millis();
  unsigned long totalStateTime;

  unsigned long totalDetumbleTime;

  RTOSstateVars.RTOSStarttime = millis();

  for(;;)
  {
    if((xQueuePeek(sensorDataQueue, &sensorData, 10) == pdPASS) &&
!RTOSstateVars.startUp)
    {
      //printSensorData(sensorData, pixel);
    }

    //Serial.println("State Machine");
    delay(1);
    if(RTOSstateVars.newState){

      RTOSstateVars.newState = false;
/*
      previousTime = switchTime; // Save last state switch time
      switchTime = millis(); //track
      totalStateTime = switchTime - previousTime;
*/
      // Turn off torquers
      digitalWrite(DIR1X, 0 >= 0 ? LOW : HIGH);
```

```
      digitalWrite(DIR2X, 0 >= 0 ? HIGH : LOW);
      analogWrite(PWMX, abs(0));

      digitalWrite(DIR1Y, 0 >= 0 ? LOW : HIGH);
      digitalWrite(DIR2Y, 0 >= 0 ? HIGH : LOW);
      analogWrite(PWMY, abs(0));

      digitalWrite(DIR1Z, 0 >= 0 ? LOW : HIGH);
      digitalWrite(DIR2Z, 0 >= 0 ? HIGH : LOW);
      analogWrite(PWMZ, abs(0));

      if(RTOSstateVars.prevState == DETUMBLE){
        // Save the total detumble time
        totalDetumbleTime = totalStateTime;
      }

      switch (RTOSstateVars.state){

        case START_UP:
        {
          Serial.println("RTOS start up!");
          // Get payload into a known state, all other tasks always running
          RTOSstateVars.startUp = true;

          RTOSstateVars.prevState = RTOSstateVars.state;
          RTOSstateVars.state = IDLE;
          RTOSstateVars.newState = true;

          // Check uplink buffer immediately on start up
          initBufAsZeros(downlinkPacket);
          constructDownlink(RTOSstateVars, CHKUPLINK, downlinkPacket,
CHKUPLINK, sensorData);
          xQueueSend(downlinkBufQueue, &downlinkPacket, portMAX_DELAY);
          delay(5);

          // Check IMU1 Data
          if (myICM1.dataReady()) {
            RTOSstateVars.sensorUpdate = true; // update made to sensor data
            myICM1.getAGMT();

            fliterAndPollIMU(myICM1, sensorData, sensorOffsetBoard2,
corrValStruct, RTOSstateVars.startUp, IMU1_AD0_VAL);
          }
          else{
            Serial.println("Failed to poll ICM20948-1.");
          }

          // Check IMU2 Data
          if (myICM2.dataReady()) {
            RTOSstateVars.sensorUpdate = true; // update made to sensor data
            myICM2.getAGMT();
```

```
            fliterAndPollIMU(myICM2, sensorData, sensorOffsetBoard2,
corrValStruct, RTOSstateVars.startUp, IMU2_AD0_VAL);
          }
          else{
            Serial.println("Failed to poll ICM20948-2.");
          }

          // Check Magnetometer Data
          if(lis2mdl.getEvent(&event)){
            RTOSstateVars.sensorUpdate = true; // update made to sensor data
            fliterAndPollMag(event, sensorData, sensorOffsetBoard2,
corrValStruct, RTOSstateVars.startUp);
          }
          else{
            Serial.println("Failed to poll LIS2MDL.");
          }

          if(RTOSstateVars.sensorUpdate){
            sensorData.timeStamp = millis()-RTOSstateVars.RTOSStarttime;  //
milliseconds since the beginning of operation
            xQueueReset(sensorDataQueue); // Clear the queue
            xQueueSend(sensorDataQueue, &sensorData, portMAX_DELAY); //
replace with new sensor data
            RTOSstateVars.sensorUpdate = false; // reset the control variable
            //Serial.println("Sensor Data updated.");
          }

          // Downlink H&S Packet on start up
          initBufAsZeros(downlinkPacket);
          constructDownlink(RTOSstateVars, SEND_DOWNLINK, downlinkPacket,
HSPACKET, sensorData);
          RTOSstateVars.packetSeqNum += 1; // Iterate Sequence number for
iridium packets
          xQueueSend(iridiumDownlinkBufQueue, &downlinkPacket,
portMAX_DELAY);
          delay(5);

        }break;

        case IDLE:
        {
          RTOSstateVars.pollSensors = true;
          RTOSstateVars.pollCameras = false;
          RTOSstateVars.detumble = false;
          RTOSstateVars.IridiumReqTime = 60000; // in ms
          RTOSstateVars.UplinkReqTime = 5000; // in ms
        }break;

        case DETUMBLE:
        {
          RTOSstateVars.IridiumReqTime = 60000; // in ms
```

```
          RTOSstateVars.UplinkReqTime = 5000; // in ms
          RTOSstateVars.detumble = true;
        }break;
/*
        case NADIR_POINT:
        {

        }
*/
        case SAFETY:
        {
          RTOSstateVars.pollSensors = false;
          RTOSstateVars.pollCameras = false;
          RTOSstateVars.detumble = false;

          RTOSstateVars.IridiumReqTime = INT32_MAX; // in ms
          RTOSstateVars.UplinkReqTime = INT32_MAX; // in ms
        }break;

        default:
        {

        }break;
      }
    }

    loopCounter += 1;

    if(int(fmod(loopCounter,
(RTOSstateVars.UplinkReqTime/RTOSstateVars.StateMachineDelay))) == 0){
      initBufAsZeros(downlinkPacket);
      constructDownlink(RTOSstateVars, CHKUPLINK, downlinkPacket, CHKUPLINK,
sensorData);
      xQueueSend(downlinkBufQueue, &downlinkPacket, portMAX_DELAY);
    }

    if(int(fmod(loopCounter,
(RTOSstateVars.IridiumReqTime/RTOSstateVars.StateMachineDelay))) == 0){ //
requests an uplink uplink about every netReqTime milliseconds
      initBufAsZeros(downlinkPacket);
      constructDownlink(RTOSstateVars, NETREQ, downlinkPacket, NETREQ,
sensorData);
      xQueueSend(downlinkBufQueue, &downlinkPacket, portMAX_DELAY);
    }

    //Serial.print("State: ");
    //Serial.println(state);
    //Serial.println();

    vTaskDelay( RTOSstateVars.StateMachineDelayTicks );
  }
```

```
}

void TaskPollSensors(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  SensorData sensorData; // Used as buffer to send stuff to NSL FC

  float magPrevVals[3];

  for(;;)
  {
    if(RTOSstateVars.pollSensors){
      //Serial.println("Poll Sensors");

      // Check IMU1 Data
      if (myICM1.dataReady()) {
        RTOSstateVars.sensorUpdate = true; // update made to sensor data
        myICM1.getAGMT();

        fliterAndPollIMU(myICM1, sensorData, sensorOffsetBoard2,
corrValStruct, RTOSstateVars.startUp, IMU1_AD0_VAL);
      }
      else{
        Serial.println("Failed to poll ICM20948-1.");
      }

      // Check IMU2 Data
      if (myICM2.dataReady()) {
        RTOSstateVars.sensorUpdate = true; // update made to sensor data
        myICM2.getAGMT();
        fliterAndPollIMU(myICM2, sensorData, sensorOffsetBoard2,
corrValStruct, RTOSstateVars.startUp, IMU2_AD0_VAL);
      }
      else{
        Serial.println("Failed to poll ICM20948-2.");
      }

      // Check Magnetometer Data
      if(lis2mdl.getEvent(&event)){
        RTOSstateVars.sensorUpdate = true; // update made to sensor data
        fliterAndPollMag(event, sensorData, sensorOffsetBoard2,
corrValStruct, RTOSstateVars.startUp);
      }
      else{
        Serial.println("Failed to poll LIS2MDL.");
      }

      if(RTOSstateVars.sensorUpdate){
        sensorData.timeStamp = millis()-RTOSstateVars.RTOSStarttime;  //
milliseconds since the beginning of operation
```

```
        xQueueReset(sensorDataQueue); // Clear the queue
        xQueueSend(sensorDataQueue, &sensorData, portMAX_DELAY); // replace
with new sensor data
        RTOSstateVars.sensorUpdate = false; // reset the control variable
        //Serial.println("Sensor Data updated.");
      }
    }

    vTaskDelay( RTOSstateVars.PollSensorsDelayTicks ); // wait for
PollSensorsDelay ms
  }
}

void TaskPollCameras( void *pvParameters )
{
  (void) pvParameters;

  SensorData sensorData; // Used as buffer to send stuff to NSL FC

  float minVal = INT_MAX;
  float maxVal = 0;

  // MLX IR Cam
  float frame[32*24]; // buffer for full frame of temperatures
  float pixel;

  bool update = false;

  for(;;)
  {
    if(RTOSstateVars.pollCameras){
      //Serial.println("Poll Cameras");

      // Reset the min and max val for next image
      minVal = INT_MAX;
      maxVal = 0;

      // Check MLX IR Cam data
      if (mlx.getFrame(frame) != 0) {
        Serial.println("Failed to get MLX frame.");
      }
      else{
        update = true; // update made to sensor data
        for(int i=0; i<24; i++){
          for(int j=0; j<32; j++){
            sensorData.cam1Data.Image[i][j] = frame[i*32 + j];
            if (sensorData.cam1Data.Image[i][j] < minVal) {
              minVal = sensorData.cam1Data.Image[i][j];
            }
            if (sensorData.cam1Data.Image[i][j] > maxVal) {
              maxVal = sensorData.cam1Data.Image[i][j];
```

```
            }
          }
        }

        // Convert temperature data to pixel values data
        temp_to_image(sensorData.cam1Data.Image, WIDTH, HEIGHT, minVal,
maxVal);
      }

      if(update){
        xQueueReset(sensorDataQueue); // Clear the queue
        xQueueSend(sensorDataQueue, &sensorData, portMAX_DELAY); // replace
with new sensor data
        update = false; // reset the control variable
        Serial.println("Sensor Data updated.");
      }
    }

    vTaskDelay( RTOSstateVars.PollCamerasDelayTicks ); // wait for
PollSensorsDelay ms
  }
}

void TaskInterpretUplink(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  uint8_t downlinkPacket[UART_PACKET_SIZE]; // Used as buffer to send stuff
to NSL FC
  uint8_t detumbleDownlinkPacket[UART_PACKET_SIZE]; // Used as buffer to send
stuff to NSL FC
  initBufAsZeros(detumbleDownlinkPacket);

  uint8_t uplinkBuf[UART_PACKET_SIZE]; // Used as buffer to send stuff to NSL
FC
  uint8_t functionByte;
  uint16_t busTempBytes;
  float busTemp;
  uint16_t busVoltageBytes;
  float busVoltage;

  // S4 Configuration Settings
  uint8_t HSPacketTXPeriod = 0x15; // 15 minutes
  uint8_t GPSPacketTXPeriod = 0x15; // 15 minutes
  uint8_t UplinkQueueCheckPeriod = 0x15; // 15 minutes

  int iridiumLatency = -1; // how long did it take to send your packet over
iridium S4 radio

  bool uplinkReceived = false;
```

```
  SensorData sensorData;
  uint8_t numCmdSent;
  uint8_t numBytesSent;
  uint8_t gndCmd;
  uint8_t packetType;
  bool newGndCmd = false;

  for(;;)
  {
    initBufAsZeros(uplinkBuf);
    //Serial.println("Interpret Uplink");
    //Read in queue and check if the known packet header is present
    if (Serial3.available()) {
      delay(50); // allow whole transmission to send

      int numBytesAvailable = Serial3.available();
      if(numBytesAvailable > 0){
        // Read in the available data on Serial 3. This only happens when
there is an NSL uplink which is not prompted by us
        for(int i=0; i<numBytesAvailable; i++){
          uplinkBuf[i] = Serial3.read();
        }

        uplinkReceived = true;
      }
    }
    else if (xQueueReceive(uplinkBufQueue, &uplinkBuf, 10) == pdPASS) {
      uplinkReceived = true;
    }


    if(uplinkReceived){

      uplinkReceived = false; // reset control var

      if(uplinkBuf[0] == 0x50 && uplinkBuf[1] == 0x50 && uplinkBuf[2] ==
0x50){
        functionByte = uplinkBuf[3];
      }
      else{
        functionByte = uplinkBuf[0];
      }

      switch(functionByte) {

        case 0x48: // Check uplink buffer from NSL Bus for data
        {
          // NSL keeps track of how many ground commands it has uplinked to
the payload with this byte.
          // We can use it to tell if we have processed a command already or
not.
```

```
          if(RTOSstateVars.startUp){
            numCmdSent = uplinkBuf[4];
            RTOSstateVars.startUp = false;
          }

          if(numCmdSent != uplinkBuf[4]){
            numCmdSent = uplinkBuf[4]; // update for new command
            newGndCmd = true;
          }

          if(newGndCmd){
            newGndCmd = false;
            numBytesSent = uplinkBuf[5]; // From emulator code
            gndCmd = uplinkBuf[6]; // Need to ask NSL about this

            switch(gndCmd) {
              // char(zero) in ASCII is 0x30, so sending a "0" over Iridium
will result in 0x30 being uplinked to our payload
              case 0x30: // DOWNLINK ground command
              {
                Serial.println("DOWNLINK command received");

                xQueuePeek(sensorDataQueue, &sensorData, portMAX_DELAY);

                packetType = uplinkBuf[7];
/*
                if(packetType == DETUMBLEHIGHRES){
                  constructDownlink(packetSeqNum, SEND_DOWNLINK,
detumbleDownlinkPacket, packetType, sensorData, detumbleSequenceNum,
useIMU2);
                  packetSeqNum += 1;
                  detumbleSequenceNum += 1;
                  if(detumbleSequenceNum == 8){ // Detect when packet is
full, then write to the buffer. Multiple time steps in one packet
                    detumbleSequenceNum = 0;
                    xQueueSend(iridiumDownlinkBufQueue,
&detumbleDownlinkPacket, portMAX_DELAY);
                    initBufAsZeros(detumbleDownlinkPacket);
                    delay(1);
                  }
                }
*/
                initBufAsZeros(downlinkPacket);
                constructDownlink(RTOSstateVars, SEND_DOWNLINK,
downlinkPacket, packetType, sensorData); // Only supports H&S downlink from
ground cmd, detumble packets should send automatically
                RTOSstateVars.packetSeqNum += 1; // Iterate Sequence number
for iridium packets
                xQueueSend(iridiumDownlinkBufQueue, &downlinkPacket,
portMAX_DELAY);
                delay(1);
```

```
} break;

case 0x31: // DETUMBLE ground command
{
  Serial.println("DETUMBLE command received");
  RTOSstateVars.prevState = RTOSstateVars.state;
  RTOSstateVars.state = DETUMBLE;
  RTOSstateVars.newState = true;
  delay(1);
} break;

case 0x32: // NADIR POINT ground command
{
  Serial.println("NADIR POINT command received");
  RTOSstateVars.prevState = RTOSstateVars.state;
  RTOSstateVars.state = NADIR_POINT;
  RTOSstateVars.newState = true;
  delay(1);
} break;

case 0x33: // SAFETY ground command
{
  Serial.println("SAFETY command received");
  RTOSstateVars.prevState = RTOSstateVars.state;
  RTOSstateVars.state = SAFETY;
  RTOSstateVars.newState = true;
  delay(1);
} break;

case 0x34: // STOP ground command
{
  Serial.println("STOP command received");
  RTOSstateVars.prevState = RTOSstateVars.state;
  RTOSstateVars.state = IDLE;
  RTOSstateVars.newState = true;
  delay(1);
} break;
case 0x35:
{
  Serial.println("FLIP VOLTAGE X command received");
  RTOSstateVars.flipVoltageX = !RTOSstateVars.flipVoltageX;
  Serial.print("flipVoltageX: ");
  Serial.println(RTOSstateVars.flipVoltageX);
}break;
case 0x36:
{
  Serial.println("FLIP VOLTAGE Y command received");
  RTOSstateVars.flipVoltageY = !RTOSstateVars.flipVoltageY;
  Serial.print("flipVoltageY: ");
  Serial.println(RTOSstateVars.flipVoltageY);
}break;
```

```
              case 0x37:
              {
                Serial.println("FLIP VOLTAGE Z command received");
                RTOSstateVars.flipVoltageZ = !RTOSstateVars.flipVoltageZ;
                Serial.print("flipVoltageZ: ");
                Serial.println(RTOSstateVars.flipVoltageZ);
              }break;
              case 0x38:
              {
                Serial.println("FLIP IMU command received");
                RTOSstateVars.useIMU2 = !RTOSstateVars.useIMU2;
                Serial.print("Use IMU2: ");
                Serial.println(RTOSstateVars.useIMU2);
              }break;
              case 0x39:
              {
                Serial.println("RESET QUEUES command received");
                xQueueReset(sensorDataQueue); // Clear the queue
                xQueueReset(iridiumDownlinkBufQueue); // Clear the queue
                xQueueReset(downlinkBufQueue); // Clear the queue
                xQueueReset(uplinkBufQueue); // Clear the queue
              }break;
              default:
              {
                Serial.println("Ground Command received is unable to be
interpretted.");
              }break;

            }
          }
          else{
            Serial.println("Command uplinked has already been processed.");
            delay(1);
          }
        }
        break;

        case 0xF1: // Check Health and Safety of Bus
        {
          Serial.println("Health and Safety Information:");
          delay(5);
          serPrintHex(uplinkBuf, UART_PACKET_SIZE);
          Serial.println();
          delay(5);

          busTempBytes = ((uint16_t)uplinkBuf[5]<<8) | uplinkBuf[6];
          busTemp = short(busTempBytes)/16.0;
          busVoltageBytes = ((uint16_t)uplinkBuf[7]<<8) | uplinkBuf[8];
          busVoltage = busVoltageBytes * ( 5.0 / ((1<<10)-1.0) ) * 7.652; //
conversion given on the NSL ICD
          Serial.print("Bus Temp: ");
```

```
          Serial.println(busTemp);
          delay(5);

        }
        break;

        case 0xF3: // Request S4 params
        {
          Serial.println("S4 Parameters:");
          delay(5);
          serPrintHex(uplinkBuf, UART_PACKET_SIZE);
          Serial.println();
          delay(5);

          char temp[2]; // To help extract decimal minutes from hex
transmission
          uint8_t recFuncByte = uplinkBuf[3];
          // NSL sending decimal values as hex, extract decimal value by
converting hex to chars and then to ints
          sprintf(temp, "%x", uplinkBuf[4]);
          HSPacketTXPeriod = atoi(temp);
          sprintf(temp, "%x", uplinkBuf[5]);
          GPSPacketTXPeriod = atoi(temp);
          sprintf(temp, "%x", uplinkBuf[6]);
          UplinkQueueCheckPeriod = atoi(temp);

          Serial.print("H&S Tx Period: ");
          delay(1);
          Serial.println(HSPacketTXPeriod);
          delay(1);
          Serial.print("GPS Tx Period: ");
          delay(1);
          Serial.println(GPSPacketTXPeriod);
          delay(1);
          Serial.print("Uplink Check Period: ");
          delay(1);
          Serial.println(UplinkQueueCheckPeriod);
          delay(1);
        }
        break;

        case 0xF5: // downlink IrishSat data to NSL FC
        {  // Interpret if S4 was successful in transmission, print latency
/*
          Serial.println("Downlink of IrishSat Packet:");
          delay(5);
          serPrintHex(uplinkBuf, UART_PACKET_SIZE);
          Serial.println();
          delay(5);
*/
          if(uplinkBuf[1] == 0xFF){
```

```
            Serial.println("Transmission Failed.");
          }
          else{
            iridiumLatency = uplinkBuf[1];
            Serial.print("Latency: ");
            Serial.print(iridiumLatency);
            Serial.println(" seconds");
            Serial.println("Transmission Successful!");
          }
        }
        break;

        default:
        {
          Serial.println("Invalid functional value OR response not receieved
yet... ");
          serPrintHex(&functionByte, 1);
        }
        break;
      }
    }
    vTaskDelay( RTOSstateVars.InterpretUplinkDelayTicks );
  }
}

void TaskDetumble(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  SensorData sensorData; // Used as buffer to send stuff to NSL FC

  uint8_t detumbleDownlinkPacket[UART_PACKET_SIZE];

  float dB[NUM_MAGS] = {0};
  float voltage_in[NUM_MAGS] = {0};
  float B_magnitude_squared = 0;
  float desiredMagneticMoment[NUM_MAGS] = {0};
  float b_dot_term[NUM_MAGS] = {0};
  float w_sat[3];
  float B_body[3];
  float prevB[3];
  float current_in[NUM_MAGS] = {0};
  //not sure how to make a persistent array of prevB values - possibly a
Queue?
  bool gyro_working = 0;
  float n, a, epsilon = 0;

  int dutyCycleX = 0;
  int dutyCycleY = 0;
  int dutyCycleZ = 0;
```

```
  int demoDutyCycleX = 255;
  int demoDutyCycleY = 255;
  int demoDutyCycleZ = 255;

  long int detumbleLoopCounter = 0;
  int demoSwitchingTime = 10000; // 10 seconds
  bool demo = false;

  for(;;)
  {
    if(RTOSstateVars.detumble){
      detumbleLoopCounter += 1;
      //Serial.println("Detumble");
      if(xQueuePeek(sensorDataQueue, &sensorData, 10) == pdPASS)
      {
        // Use most recent data put into sensor data queue by
TaskPollSensor() to run bcross algo

        xQueuePeek(sensorDataQueue, &sensorData, portMAX_DELAY);

        if(int(fmod(detumbleLoopCounter,
((RTOSstateVars.IridiumPacketHighFreq/8)/RTOSstateVars.StateMachineDelay)))
== 0){
          constructDownlink(RTOSstateVars, SEND_DOWNLINK,
detumbleDownlinkPacket, DETUMBLEHIGHRES, sensorData);
          RTOSstateVars.detumbleSequenceNum += 1;
          if(RTOSstateVars.detumbleSequenceNum == 8){ // Detect when packet
is full, then write to the buffer. Multiple time steps in one packet
            Serial.print("Detumble Packet Sent: Packet #");
            Serial.println(RTOSstateVars.packetSeqNum);
            RTOSstateVars.packetSeqNum += 1;
            RTOSstateVars.detumbleSequenceNum = 0;
            xQueueSend(iridiumDownlinkBufQueue, &detumbleDownlinkPacket,
portMAX_DELAY);
            initBufAsZeros(detumbleDownlinkPacket);
            delay(1);
          }
        }


        if (RTOSstateVars.useIMU2){
          w_sat[0] = sensorData.imu2Data.gyrX * (PI/180); // CHANGED FROM DEG
TO RADIANS
          w_sat[1] = sensorData.imu2Data.gyrY * (PI/180);
          w_sat[2] = sensorData.imu2Data.gyrZ * (PI/180);

          B_body[0] = sensorData.imu2Data.magXField/10e6; // CHANGE FROM uT
TO T
          B_body[1] = sensorData.imu2Data.magYField/10e6;
          B_body[2] = sensorData.imu2Data.magZField/10e6;
        }
```

```
        else{
          w_sat[0] = sensorData.imu1Data.gyrX * (PI/180); // CHANGED FROM DEG
TO RADIANS
          w_sat[1] = sensorData.imu1Data.gyrY * (PI/180);
          w_sat[2] = sensorData.imu1Data.gyrZ * (PI/180);

          B_body[0] = sensorData.imu1Data.magXField/10e6; // CHANGE FfROM uT
TO T
          B_body[1] = sensorData.imu1Data.magYField/10e6;
          B_body[2] = sensorData.imu1Data.magZField/10e6;
        }

        if (w_sat[0]) {
          gyro_working = 1;
        } else {
          gyro_working = 0;
        }

        // Use most recent data put into sensor data queue by
TaskPollSensor() to run bcross algo
        B_magnitude_squared = 0;
        for (int i = 0; i < NUM_MAGS; i++) {
          B_magnitude_squared += pow(B_body[i], 2); // B_body[0]^2 +
B_body[1]^2 + B_body[2]^2
        }

        if (B_magnitude_squared == 0) {
          printf("Magnetic field vector magnitude cannot be zero!\n");
          // should we do something besides this print statement if this
value is 0?
        }

        // use b-cross equation when we have access to magnetic field data
        if (gyro_working) {
            // b_dot_term is B x w, the cross-product of B_body and w_sat
            b_dot_term[0] = B_body[1]*w_sat[2] - B_body[2]*w_sat[1];
            b_dot_term[1] = B_body[2]*w_sat[0] - B_body[0]*w_sat[2];
            b_dot_term[2] = B_body[0]*w_sat[1] - B_body[1]*w_sat[0];

            // desiredMagneticMoment = - (k / ||B||^2) * (B x w)
            for (int j = 0; j < NUM_MAGS; j++) {
                desiredMagneticMoment[j] = - (K / B_magnitude_squared) *
b_dot_term[j];
            }

        } else {
            // if gyroscope is off, estimate derivative of B field
            // Compute magnetic moment using the control law without w: - k *
B'
            // TODO: research methods to smooth noise out: Savitzky-Golay
Filter, low pass filter, kalman, etc
```

```
            if (sizeof(prevB) >= 2*NUM_MAGS) { // OR >= 2?? We want 2+
readings but each reading has 3 components, so check for at least 6 elements
in prevB?
                // perform a linear regression over the last few readings
                // np.arange returns an interval of evenly-spaced values,
from -sizeof(prevB)+1 to 1, but 1 is not included
                // if prevB has 4 readings, then would be DT-sized steps on
interval [-3, 1)
                // find abs val (1 - -3 = 4) then var starts at -3, while
loop incrementing by +DT, while var < 1.
                // use counter? to see how many elements the times array will
need
                //times = np.arange(-sizeof(prevB) + 1, 1) * DT;

            } else { // (if we only have 1 previous B measurement)
                // fall back to basic finite difference if not enough data
                for (int k = 0; k < NUM_MAGS; k++) {
                    dB[k] = B_body[k] - prevB[0] / DT;
                }
            }
            for (int l = 0; l < NUM_MAGS; l++) {
                desiredMagneticMoment[l] = -(K * dB[l]);
            }
        }

        current_in[NUM_MAGS] = {0};
        for (int m = 0; m < NUM_MAGS; m++) {
            n, a, epsilon = 0;
            // find current for 2 ferro and 1 air core magnetorquers using
dipole / nA*epsilon
            if (m == 0 || m == 1) { // ferro
                n =         FERRO_NUM_TURNS;
                a =         FERRO_AREA;
                epsilon =   FERRO_EPSILON;
            } else { // air core (if l == 2)
                n =         AIR_NUM_TURNS;
                a =         AIR_AREA;
                epsilon =   AIR_EPSILON;
            }

            current_in[m] = desiredMagneticMoment[m] / (n * a * epsilon);

            if(m==0 || m==1){ // X or Y axis use torquerods
              voltage_in[m] = current_in[m] * RESIS_FERRO_MAG;
            }
            else if(m==2){ // Z axis uses Air core
              voltage_in[m] = current_in[m] * RESIS_AIR_MAG;
            }

            // Check to see if voltage exceeds maximum of 5V
            if(voltage_in[m] >= FULL_VOLTAGE){
```

```
            voltage_in[m] = FULL_VOLTAGE;
          }
          else if(voltage_in[m] <= (-FULL_VOLTAGE)){
            voltage_in[m] = (-FULL_VOLTAGE);
          }

          // Flip voltage if ground command says so (hardware in backwards
or something)
          if(RTOSstateVars.flipVoltageX && m==0){
            voltage_in[m] = -voltage_in[m];
          }
          else if(RTOSstateVars.flipVoltageY && m==1){
            voltage_in[m] = -voltage_in[m];
          }
          else if(RTOSstateVars.flipVoltageZ && m==2){
            voltage_in[m] = -voltage_in[m];
          }
        }
/*
        Serial.print("Voltage X: ");
        Serial.println(voltage_in[0],10);
        delay(1);
        Serial.print("Voltage Y: ");
        Serial.println(voltage_in[1],10);
        delay(1);
        Serial.print("Voltage Z: ");
        Serial.println(voltage_in[2],10);
        delay(1);
*/
        //Calculate the duty cycles according to algorithm required voltage
        dutyCycleX = int(round((voltage_in[0]/FULL_VOLTAGE)*255.0));
        dutyCycleY = int(round((voltage_in[1]/FULL_VOLTAGE)*255.0));
        dutyCycleZ = int(round((voltage_in[2]/FULL_VOLTAGE)*255.0));

        if((int(fmod(detumbleLoopCounter,
(demoSwitchingTime/RTOSstateVars.DetumbleDelay))) == 0) && demo){
          demoDutyCycleX = -demoDutyCycleX;
          demoDutyCycleY = -demoDutyCycleY;
          demoDutyCycleZ = -demoDutyCycleZ;
        }

        // update magnetorquer outputs
        if(demo){
          digitalWrite(DIR1X, demoDutyCycleX >= 0 ? LOW : HIGH);
          digitalWrite(DIR2X, demoDutyCycleX >= 0 ? HIGH : LOW);
          analogWrite(PWMX, abs(demoDutyCycleX));

          digitalWrite(DIR1Y, demoDutyCycleY >= 0 ? LOW : HIGH);
          digitalWrite(DIR2Y, demoDutyCycleY >= 0 ? HIGH : LOW);
          analogWrite(PWMY, abs(demoDutyCycleY));
```

```
          digitalWrite(DIR1Z, demoDutyCycleZ >= 0 ? LOW : HIGH);
          digitalWrite(DIR2Z, demoDutyCycleZ >= 0 ? HIGH : LOW);
          analogWrite(PWMZ, abs(demoDutyCycleZ));
        }
        else{
          digitalWrite(DIR1X, dutyCycleX >= 0 ? LOW : HIGH);
          digitalWrite(DIR2X, dutyCycleX >= 0 ? HIGH : LOW);
          analogWrite(PWMX, abs(dutyCycleX));

          digitalWrite(DIR1Y, dutyCycleY >= 0 ? LOW : HIGH);
          digitalWrite(DIR2Y, dutyCycleY >= 0 ? HIGH : LOW);
          analogWrite(PWMY, abs(dutyCycleY));

          digitalWrite(DIR1Z, dutyCycleZ >= 0 ? LOW : HIGH);
          digitalWrite(DIR2Z, dutyCycleZ >= 0 ? HIGH : LOW);
          analogWrite(PWMZ, abs(dutyCycleZ));
        }
/*
        Serial.print("Duty Cycle X: ");
        Serial.println(dutyCycleX);
        delay(1);
        Serial.print("Duty Cycle Y: ");
        Serial.println(dutyCycleY);
        delay(1);
        Serial.print("Duty Cycle Z: ");
        Serial.println(dutyCycleZ);
        delay(1);
*/
        //return voltage_in; //not sure what to do with return val - add to a
queue maybe?
      }
    }
    vTaskDelay( RTOSstateVars.DetumbleDelayTicks ); // wait for 1000 ms
  }
}

void TaskSendAndAck(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  uint8_t downlinkBuf[UART_PACKET_SIZE]; // Used as buffer to send stuff to
NSL FC
  uint8_t uplinkBuf[UART_PACKET_SIZE]; // Used as buffer to send stuff to NSL
FC

  int numBytesAvailable = 0;

  bool downlinkReady = false;

  for(;;)
  {
```

```
    initBufAsZeros(downlinkBuf);
    //Serial.println("Send and ACK");
    if(!RTOSstateVars.detumble && xQueueReceive(iridiumDownlinkBufQueue,
&downlinkBuf, 10) == pdPASS) downlinkReady = true;
    else if (xQueueReceive(downlinkBufQueue, &downlinkBuf, 10) == pdPASS)
downlinkReady = true;

    if (downlinkReady){
      downlinkReady = false;

      initBufAsZeros(uplinkBuf);

      digitalWrite(RTS_PIN, HIGH);
      delay(200);
      while(digitalRead(CTS_PIN) == LOW) delay(1); // check every 1ms to see
if NSL FC is ready
      delay(5); // delay before sending packet, as instructed to do by NSL

      Serial3.write(downlinkBuf, sizeof(downlinkBuf));
/*
      Serial.println("Packet Sent to NSL:");
      delay(5);
      serPrintHex(downlinkBuf, UART_PACKET_SIZE);
      Serial.println("");
      delay(5);
*/

      // Wait for NSL to send the whole packet back, then set RTS to LOW
      delay(50);

      // Wait for NSL response...
      while(!Serial3.available()) delay(1);
      delay(50); // Allow full transmission over Serial3

      // Receive NSL ACK or NACK
      numBytesAvailable = Serial3.available();

      digitalWrite(RTS_PIN, LOW);

      // Read the NSL packet
      for(int i=0; i<numBytesAvailable; i++){
        uplinkBuf[i] = Serial3.read();
      }
/*
      Serial.println("Received NSL Response:");
      serPrintHex(uplinkBuf, UART_PACKET_SIZE);
      Serial.println("");
      delay(5);
*/
      if(uplinkBuf[0] == 0xAA && uplinkBuf[1] == 0x05 && uplinkBuf[2] ==
0x00){
```

```
        Serial.println("ACK");
        delay(1);
      }
      else if(uplinkBuf[0] == 0xAA && uplinkBuf[1] == 0x05 && uplinkBuf[2] ==
0xFF){
        Serial.println("NACK");
        delay(1);
      }
      else{
        Serial.println("Not an ACK or NACK. Sent to packet decoder.");
        delay(5);
        xQueueSend(uplinkBufQueue, &uplinkBuf, portMAX_DELAY); // Send
potentially important information to uplink decoding task
      }
    }
    vTaskDelay( RTOSstateVars.SendAndAckDelayTicks );
  }
}

void TaskMonitorCommands(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  uint8_t downlinkPacket[UART_PACKET_SIZE]; // Used as buffer to send stuff
to NSL FC
  char charFuncByte[2];
  uint8_t functionByte = 0x00;
  int numBytesAvailable = 0;

  uint8_t packetType;
  SensorData sensorData;

  for(;;)
  {
    //Serial.println("Monitor Commands");

    if(Serial.available()){
      numBytesAvailable = Serial.available();

      // Read in the available data on Serial 1
      if (numBytesAvailable <= 3){
        for(int i=0; i<numBytesAvailable; i++){
          charFuncByte[i] = Serial.read();
        }
        // Decode the instruction sent over serial monitor
        functionByte = char2hex(charFuncByte, true, 0x00);
        Serial.println("Function Byte: ");
        Serial.println(functionByte);
      }

      xQueuePeek(sensorDataQueue, &sensorData, portMAX_DELAY);
```

```
      initBufAsZeros(downlinkPacket);
      constructDownlink(RTOSstateVars, functionByte, downlinkPacket,
packetType, sensorData);
      xQueueSend(downlinkBufQueue, &downlinkPacket, portMAX_DELAY);
    }
    vTaskDelay( RTOSstateVars.MonitorCommandsDelayTicks ); // wait for 1000
ms
  }
}
```

Appendix D – IrishSatRTOS.h Code List

```
#include <Arduino.h>

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>

#include <semphr.h> // Include semaphore supoport
#include <queue.h> // Include queue support

#include <Adafruit_MLX90640.h>
#include <ICM_20948.h>
#include <Adafruit_LIS2MDL.h>
#include <Adafruit_Sensor.h>

// IMU scalars
#define ACCEL_SCALE 2.0 / 32768.0 // Adjust based on IMU range
#define GYRO_SCALE 250.0 / 32768.0 // Adjust for gyroscope range

//Low Pass Filter Params
#define LPF_SCALAR 0.5
#define TIME_STEP 0.1 // seconds
#define LPF_GAIN LPF_SCALAR*TIME_STEP

#define CTS_PIN 4 // Clear-to-Send pin set high or low by NSL FC
#define RTS_PIN 5 // Request-to-Send pin set high or low by IrishSat
#define RX_PIN 6 // Receive uplink from NSL FC
#define TX_PIN 7 // Send downlink data to NSL FC

#define PRINT_TEMPERATURES
#define MLX_ADDR 0x33
#define TA_SHIFT 8 //Default shift for MLX90640 in open air
#define SDA_PIN 47
#define SCL_PIN 48

// For MLX Photo Frame
#define WIDTH 32
#define HEIGHT 24
```

```
// Needed for ICM20948
#define IMU1_AD0_VAL 0
#define IMU2_AD0_VAL 1

// IMU scalars
#define ACCEL_SCALE 2.0 / 32768.0 // Adjust based on IMU range
#define GYRO_SCALE 250.0 / 32768.0 // Adjust for gyroscope range

// Set up all bytes needed for constructing a packet
#define SYNC_BYTE 0x50 // From ICD
#define UART_PACKET_SIZE 205
#define UART_DATA_SIZE 201
#define DATASIZE 20 // Size of data payload... hardcoded for testing
#define HEADERSIZE 4 // Fixed header size of 205 byte packet
#define SEND_DOWNLINK 0xF5 //Check uplinks from NSL bus
#define S4RECONFIG 0xF4 // Change S4 params
#define CHKUPLINK 0x48 //Check uplinks from NSL bus
#define NETREQ 0x47 // Request that iridium uplinks are checked by NSL bus,
placed in uplink buffer if any message from ground

#define START_UP 1 // Set initial states and stuff
#define IDLE 2 // When payload is turned on, constantly update buffers and
check uplink, wait for ground command
#define SAFETY 3 // Minimal tasks running, not polling sensors, just checking
uplink?
#define DETUMBLE 4 // Run all tasks related to detumble, reduce rate of
uplink check
#define NADIR_POINT 5 // Run all tasks related to Nadir pointing, reduce rate
of uplink check

// Packet Types
#define HSPACKET 0x31
#define DETUMBLEHIGHRES 0x32
#define DETUMBLELOWRES 0x33
#define IMAGE 0x34




//macros and var declarations for detumble (moved from inside loop)

// UPDATE THESE VALS --> Resistance, Relative perm, Num of terms for air and
ferro... spend time checing the rest

#define NUM_MAGS            3
#define RESIS_AIR_MAG       41.9 // ohms
#define RESIS_FERRO_MAG     20.5 // ohms

//#define PI                  M_PI   // pi already defined in Arduino.h
```

```
#define DT                  0.1    // time step for simulation (how long
between each iteration) (s)
#define K                   3e-5    // detumbling constant gain (according to
params.py, if GYRO_WORKING)
#define RELATIVE_PERM_MM    80000.0   // relative permeability of MuMetal
#define FERRO_LENGTH        7.0       // length of the rod (cm)
#define FERRO_NUM_TURNS     2110.0   // number of turns of coil (MuMetal
torquer)
#define FERRO_ROD_RADIUS    0.32    // core rod radius (cm) (MuMetal torquer)
#define FERRO_AREA          PI * pow(FERRO_ROD_RADIUS/100, 2)   // area of
mumetal torquer (m^2)
#define FERRO_RADIUS        FERRO_LENGTH/FERRO_ROD_RADIUS       // length-to-
radius ratio of the cylindrical torquer
#define FERRO_DEMAG_FACTOR  (4 * log(FERRO_RADIUS - 1)) / (FERRO_RADIUS *
FERRO_RADIUS - 4 * log(FERRO_RADIUS))
#define FERRO_EPSILON       (1 + (RELATIVE_PERM_MM - 1)) / (1 +
FERRO_DEMAG_FACTOR * (RELATIVE_PERM_MM - 1))

#define AIR_NUM_TURNS       341     // number of turns of coil (aircore
torquer)
#define AIR_AREA            0.007901   // area of aircore torquer (m^2)
#define AIR_EPSILON         1.0

#define FULL_VOLTAGE        5.0 // Running magnetorquers off of the 5V line


// Define magnetorquer control pins for X direction
#define DIR1X 37
#define DIR2X 38
//#define PWMX 39
#define PWMX 16 // Pin 9, A1

#define DIR1Y 41
#define DIR2Y 42
// #define PWMY 44
#define PWMY 17 // Pin 10, A0

#define DIR1Z 14
#define DIR2Z 21
//#define PWMZ 13
#define PWMZ 18 // Pin 11, A3

// Demag params
#define DEMAG_MAX_PWM 255 // 100% PWM
#define DEMAG_STEP 15     // 15% PWM reduction per step
#define DEMAG_DELAY 50    // .05 seconds

typedef struct {
  //State variables
  int StateMachineDelay; // in ms
```

```
  TickType_t StateMachineDelayTicks; // 1000 ms delay, task runs ~ every
second
  int PollSensorsDelay; // in ms
  TickType_t PollSensorsDelayTicks; // 1000 ms delay, task runs ~ every
second
  int PollCamerasDelay; // in ms
  TickType_t PollCamerasDelayTicks; // 1000 ms delay, task runs ~ every
second
  int InterpretUplinkDelay; // in ms
  TickType_t InterpretUplinkDelayTicks; // 1000 ms delay, task runs ~ every
second
  int DetumbleDelay; // in ms
  TickType_t DetumbleDelayTicks; // 1000 ms delay, task runs ~ every second
  int SendAndAckDelay; // in ms
  TickType_t SendAndAckDelayTicks; // 1000 ms delay, task runs ~ every second
  int MonitorCommandsDelay; // in ms
  TickType_t MonitorCommandsDelayTicks; // 1000 ms delay, task runs ~ every
second
  uint32_t IridiumReqTime; // in ms
  uint32_t UplinkReqTime; // in ms
  uint32_t IridiumPacketHighFreq; // in ms, 4 a minute, 120 in 30 minutes
(960 total data points)
  uint32_t IridiumPacketLowFreq; // in ms, 1 every 3 minutes, 10 in 30
minutes (80 total data points)
  uint16_t packetSeqNum;
  uint8_t detumbleSequenceNum;
  uint32_t RTOSStarttime;
  int state; // startup state
  int prevState;
  bool newState;
  bool startUp;
  bool pollSensors;
  bool pollCameras;
  bool detumble;
  bool flipVoltageX;
  bool flipVoltageY;
  bool flipVoltageZ;
  bool useIMU2;
  bool sensorUpdate;
} StateVars;

typedef struct {
  TaskHandle_t taskSendAndAckHandle;
  TaskHandle_t taskMonitorCommandsHandle;
  TaskHandle_t taskMonitorUplinkHandle;
  TaskHandle_t taskInterpretUplinkHandle;
  TaskHandle_t taskPollSensorsHandle;
  TaskHandle_t taskPollCamerasHandle;
  TaskHandle_t taskDetumbleHandle;
  TaskHandle_t taskStateMachineHandle;
} TaskHandles;
```

```
typedef struct {
  //lis2mdl
  float lisHardIron[3];
  //IMU1
  float icm1Offsets[3][3];
  //IMU2
  float icm2Offsets[3][3];

  // Board 2 Soft iron correction matrices
  float lisSoftIron[3][3];
  float icm1SoftIron[3][3];
  float icm2SoftIron[3][3];
} sensorOffsets;

typedef struct {
  //After hard offsets
  float finalXAccel;
  float finalYAccel;
  float finalZAccel;
  float finalXGyro;
  float finalYGyro;
  float finalZGyro;
  float correctedXMag;
  float correctedYMag;
  float correctedZMag;

  //after soft offsets
  float finalXMag;
  float finalYMag;
  float finalZMag;
} tempOffsetCalcHolder;


typedef struct {
  float magXField; // micro teslas
  float magYField; // micro teslas
  float magZField; // micro teslas

  float accX; // milli g's
  float accY; // milli g's
  float accZ; // milli g's

  float gyrX; // degrees per second
  float gyrY; // degrees per second
  float gyrZ; // degrees per second

  float temp; // degrees celsius
} IMUData;

typedef struct {
```

```
  float magXField; // x-axis raw data
  float magYField; // y-axis raw data
  float magZField; // z-axis raw data
} MagnetometerData;

typedef struct {
  float Image[24][32]; // 24 rows and 32 columns of pixels
} IRCameraData;

typedef struct {
  unsigned long timeStamp;
  IMUData imu1Data;
  IMUData imu2Data;
  MagnetometerData magData;
  IRCameraData cam1Data;
  IRCameraData cam2Data;
} SensorData;

// Structure to represent an RGB color
typedef struct {
  uint8_t r, g, b;
} RGB;

// define two tasks for Blink & AnalogRead
void TaskSendAndAck( void *pvParameters );
void TaskMonitorCommands( void *pvParameters );
void TaskInterpretUplink( void *pvParameters );
void TaskPollSensors( void *pvParameters );
void TaskPollCameras( void *pvParameters );
void TaskDetumble( void *pvParameters );
void TaskStateMachine( void *pvParameters );

// All function definitions
uint8_t char2hex (char c[2], bool firstConversion, uint8_t result);
void serPrintHex(uint8_t *buf, size_t len);
void initBufAsZeros(uint8_t buf[]);
float normalize(float value, float min, float max);
float grayscale_colormap(float normalized_temp);
void temp_to_image(float image[24][32], int width, int height, float
min_temp, float max_temp);
void printSensorData(SensorData &SensorData, float pixel);
void constructDownlink(StateVars &RTOSstateVars, uint8_t cmd, uint8_t
downlinkPacket[], uint8_t packetType, SensorData &sensorData);
void fliterAndPollIMU(ICM_20948_I2C &myICM, SensorData &sensorData,
sensorOffsets &sensorOffsetBoardNum, tempOffsetCalcHolder &corrValStruct,
bool startUp, int AD0val);
void fliterAndPollMag(sensors_event_t &event, SensorData &sensorData,
sensorOffsets &sensorOffsetBoardNum, tempOffsetCalcHolder &corrValStruct,
bool startUp);
```

Appendix E – IrishSatRTOS.cpp Code Listing

```
#include "IrishSatRTOS.h"

uint8_t char2hex (char c[2], bool firstConversion, uint8_t
result){
    for(int i=0; i<2; i++){
        switch(c[i]){
        case 'F':
        case 'f':
            result = result + 0b1111;
            break;
        case 'E':
        case 'e':
            result = result + 0b1110;
            break;
        case 'D':
        case 'd':
            result = result + 0b1101;
            break;
        case 'C':
        case 'c':
            result = result + 0b1100;
            break;
        case 'B':
        case 'b':
            result = result + 0b1011;
            break;
        case 'A':
        case 'a':
            result = result + 0b1010;
            break;
        // Used for hex digits 0-9, cast char as a uint8_t and
grab bottom four bits
        default:
            result = result + ((uint8_t)c[i] & 0x0F); // mask
top four bits of 8 bit character
            break;
        }

        // bit shift the first hex char (representing 4 bits) to
the left (for example, changing 0x0F to 0xF0).
        // This places the first hex value that is decoded in
its proper position.
```

```
        if(firstConversion){
        result = result<<4;
        firstConversion = !firstConversion;
        }
    }
    return result;
}

void serPrintHex(uint8_t buf[], size_t len){
    Serial.print("0x");
    for (int i=0; i<len; i++){
        if (buf[i] <= 0x0F){
        Serial.print("0");
        }
        Serial.print(buf[i], HEX);
        if (i != len-1){
        Serial.print(" ");
        }
    }
}

void initBufAsZeros(uint8_t buf[]){
    //Zero pad the rest of the packet
    for(int i=0; i<UART_PACKET_SIZE; i++){
        buf[i] = 0x00;
    }
}

// Function to normalize a value within a range
float normalize(float value, float min, float max) {
    return (value - min) / (max - min);
}

// Function to map temperature to a grayscale color
float grayscale_colormap(float normalized_temp) {
    float gray_value = (uint8_t)(normalized_temp * 255);
    return gray_value;
}

// Function to convert 2D temperature array to RGB image data
void temp_to_image(float image[24][32], int width, int height,
float min_temp, float max_temp) {
```

```
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float normalized_temp = normalize(image[y][x],
min_temp, max_temp);
            image[y][x] = grayscale_colormap(normalized_temp);
        }
    }
}


void printSensorData(SensorData &sensorData, float pixel){

    Serial.print("\033[2J"); // Clear screen
    Serial.print("\033[H");  // Cursor to home (row=0, col=0)

/*
    //Print image data (for demonstration)
    printf("----------------Image Data----------------\n");
    delay(5);
    for (int y = 0; y < HEIGHT; y++) {
    for (int x = 0; x < WIDTH; x++) {
        pixel = sensorData.cam1Data.Image[y][x];
        printf("%.0f ", pixel);
    }
    printf("\n");
    }
    delay(10);
*/
    Serial.println("----------------IMU1 Data ----------------
");
    Serial.print("Times stamp:");
    delay(1);
    Serial.println(sensorData.timeStamp);
    delay(1);
    Serial.println("Accel (mg):");
    delay(1);
    Serial.print("  X: ");
    delay(1);
    Serial.print(sensorData.imu1Data.accX); // Convert to mg
    delay(1);
    Serial.print(" Y: ");
    delay(1);
```

```
    Serial.print(sensorData.imu1Data.accY);
    delay(1);
    Serial.print(" Z: ");
    delay(1);
    Serial.println(sensorData.imu1Data.accZ);
    delay(1);

    // Gyroscope
    Serial.println("Gyro (dps):");
    delay(1);
    Serial.print("  X: ");
    delay(1);
    Serial.print(sensorData.imu1Data.gyrX); // Convert to
degrees per second
    delay(1);
    Serial.print(" Y: ");
    delay(1);
    Serial.print(sensorData.imu1Data.gyrY);
    delay(1);
    Serial.print(" Z: ");
    delay(1);
    Serial.println(sensorData.imu1Data.gyrZ);
    delay(1);

    // Magnetometer (assuming already in µT)
    Serial.println("Mag (uT):");
    delay(1);
    Serial.print("  X: ");
    delay(1);
    Serial.print(sensorData.imu1Data.magXField);
    delay(1);
    Serial.print(" Y: ");
    delay(1);
    Serial.print(sensorData.imu1Data.magYField);
    delay(1);
    Serial.print(" Z: ");
    delay(1);
    Serial.println(sensorData.imu1Data.magZField);
    delay(1);

    Serial.println("----------------IMU2 Data ----------------
");
```

```
    delay(1);
    Serial.println("Accel (mg):");
    delay(1);
    Serial.print("  X: ");
    delay(1);
    Serial.print(sensorData.imu2Data.accX); // Convert to mg
    delay(1);
    Serial.print(" Y: ");
    delay(1);
    Serial.print(sensorData.imu2Data.accY);
    delay(1);
    Serial.print(" Z: ");
    delay(1);
    Serial.println(sensorData.imu2Data.accZ);
    delay(1);

    // Gyroscope
    Serial.println("Gyro (dps):");
    delay(1);
    Serial.print("  X: ");
    delay(1);
    Serial.print(sensorData.imu2Data.gyrX); // Convert to
degrees per second
    delay(1);
    Serial.print(" Y: ");
    delay(1);
    Serial.print(sensorData.imu2Data.gyrY);
    delay(1);
    Serial.print(" Z: ");
    delay(1);
    Serial.println(sensorData.imu2Data.gyrZ);
    delay(1);

    // Magnetometer (assuming already in µT)
    Serial.println("Mag (uT):");
    delay(1);
    Serial.print("  X: ");
    delay(1);
    Serial.print(sensorData.imu2Data.magXField);
    delay(1);
    Serial.print(" Y: ");
    delay(1);
```

```
    Serial.print(sensorData.imu2Data.magYField);
    delay(1);
    Serial.print(" Z: ");
    delay(1);
    Serial.println(sensorData.imu2Data.magZField);
    delay(1);

    Serial.println("----------------Magnetometer Data ----------
------");
    delay(1);
    Serial.println("Mag (uT):");
    delay(1);
    Serial.print("  X: ");
    delay(1);
    Serial.print(sensorData.magData.magXField); // Convert to mg
    delay(1);
    Serial.print(" Y: ");
    delay(1);
    Serial.print(sensorData.magData.magYField);
    delay(1);
    Serial.print(" Z: ");
    delay(1);
    Serial.println(sensorData.magData.magZField);
    delay(1);
}

void constructDownlink(StateVars &RTOSstateVars, uint8_t cmd,
uint8_t downlinkPacket[], uint8_t packetType, SensorData
&sensorData){

    // Zero out the downlink packet first
    //initBufAsZeros(uplinkPacket);

    // Construct header
    downlinkPacket[0] = SYNC_BYTE;
    downlinkPacket[1] = SYNC_BYTE;
    downlinkPacket[2] = SYNC_BYTE;
    downlinkPacket[3] = cmd;

    switch (cmd){
        case SEND_DOWNLINK:
        {
```

```
            switch (packetType){
                //F5 iridium packets
                case HSPACKET:
                {
                    downlinkPacket[4] = 0x46; // H&S packets
contain 70 out of 200 bytes
                    downlinkPacket[5] =
(uint8_t)((RTOSstateVars.packetSeqNum >> 8) & 0xFF); // high
byte
                    downlinkPacket[6] =
(uint8_t)(RTOSstateVars.packetSeqNum & 0xFF);; // low byte
                    downlinkPacket[7] = HSPACKET;
                    downlinkPacket[8] =
(uint8_t)((sensorData.timeStamp >> 24) & 0xFF); // high byte
                    downlinkPacket[9] =
(uint8_t)((sensorData.timeStamp >> 16) & 0xFF);
                    downlinkPacket[10] =
(uint8_t)((sensorData.timeStamp >> 8) & 0xFF);
                    downlinkPacket[11] =
(uint8_t)(sensorData.timeStamp & 0xFF);; // low byte;
/*
                    Serial.print("Time stamp:");
                    Serial.println(sensorData.timeStamp);
*/
                    if(RTOSstateVars.useIMU2){
                        //Acceleration Conversion
                        downlinkPacket[12] =
(uint8_t)(((int16_t)sensorData.imu2Data.accX >> 8) & 0xFF);
                        downlinkPacket[13] =
(uint8_t)(((int16_t)sensorData.imu2Data.accX) & 0xFF);
                        downlinkPacket[14] =
(uint8_t)(((int16_t)sensorData.imu2Data.accY >> 8) & 0xFF);
                        downlinkPacket[15] =
(uint8_t)(((int16_t)sensorData.imu2Data.accY) & 0xFF);
                        downlinkPacket[16] =
(uint8_t)(((int16_t)sensorData.imu2Data.accZ >> 8) & 0xFF);
                        downlinkPacket[17] =
(uint8_t)(((int16_t)sensorData.imu2Data.accZ) & 0xFF);
                        // Gyro Conversions
                        downlinkPacket[18] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrX*100) >> 8) &
0xFF); // 10s of mdegs/s
```

```
                        downlinkPacket[19] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrX*100)) & 0xFF);
                        downlinkPacket[20] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrY*100) >> 8) &
0xFF);
                        downlinkPacket[21] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrY*100)) & 0xFF);
                        downlinkPacket[22] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrZ*100) >> 8) &
0xFF);
                        downlinkPacket[23] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrZ*100)) & 0xFF);
                        // Mag field Conversion
                        downlinkPacket[24] =
(uint8_t)(((int16_t)(sensorData.imu2Data.magXField) >> 8) &
0xFF); // uT
                        downlinkPacket[25] =
(uint8_t)(((int16_t)sensorData.imu2Data.magXField) & 0xFF);
                        downlinkPacket[26] =
(uint8_t)(((int16_t)(sensorData.imu2Data.magYField) >> 8) &
0xFF);
                        downlinkPacket[27] =
(uint8_t)(((int16_t)sensorData.imu2Data.magYField) & 0xFF);
                        downlinkPacket[28] =
(uint8_t)(((int16_t)(sensorData.imu2Data.magZField) >> 8) &
0xFF);
                        downlinkPacket[29] =
(uint8_t)(((int16_t)sensorData.imu2Data.magZField) & 0xFF);
                    }
                    else{
                        //Acceleration Conversion
                        downlinkPacket[12] =
(uint8_t)(((int16_t)sensorData.imu1Data.accX >> 8) & 0xFF);
                        downlinkPacket[13] =
(uint8_t)(((int16_t)sensorData.imu1Data.accX) & 0xFF);
                        downlinkPacket[14] =
(uint8_t)(((int16_t)sensorData.imu1Data.accY >> 8) & 0xFF);
                        downlinkPacket[15] =
(uint8_t)(((int16_t)sensorData.imu1Data.accY) & 0xFF);
                        downlinkPacket[16] =
(uint8_t)(((int16_t)sensorData.imu1Data.accZ >> 8) & 0xFF);
```

```
                        downlinkPacket[17] =
(uint8_t)(((int16_t)sensorData.imu1Data.accZ) & 0xFF);
/*
                        Serial.print("Acc X: ");

Serial.println(sensorData.imu1Data.accX);
                        Serial.print("Acc Y: ");

Serial.println(sensorData.imu1Data.accY);
                        Serial.print("Acc Z: ");

Serial.println(sensorData.imu1Data.accZ);
                        delay(5);
*/
                        // Gyro Conversions
                        downlinkPacket[18] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrX*100) >> 8) &
0xFF); // 10s of mdegs/s
                        downlinkPacket[19] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrX*100)) & 0xFF);
                        downlinkPacket[20] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrY*100) >> 8) &
0xFF);
                        downlinkPacket[21] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrY*100)) & 0xFF);
                        downlinkPacket[22] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrZ*100) >> 8) &
0xFF);
                        downlinkPacket[23] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrZ*100)) & 0xFF);
/*
                        Serial.print("Gyro X: ");

Serial.println(sensorData.imu1Data.gyrX);
                        Serial.print("Gyro Y: ");

Serial.println(sensorData.imu1Data.gyrY);
                        Serial.print("Gyro Z: ");

Serial.println(sensorData.imu1Data.gyrZ);
                        delay(5);
*/
```

```
                        // Mag field Conversion
                        downlinkPacket[24] =
(uint8_t)(((int16_t)(sensorData.imu1Data.magXField) >> 8) &
0xFF); // uT
                        downlinkPacket[25] =
(uint8_t)(((int16_t)sensorData.imu1Data.magXField) & 0xFF);
                        downlinkPacket[26] =
(uint8_t)(((int16_t)(sensorData.imu1Data.magYField) >> 8) &
0xFF);
                        downlinkPacket[27] =
(uint8_t)(((int16_t)sensorData.imu1Data.magYField) & 0xFF);
                        downlinkPacket[28] =
(uint8_t)(((int16_t)(sensorData.imu1Data.magZField) >> 8) &
0xFF);
                        downlinkPacket[29] =
(uint8_t)(((int16_t)sensorData.imu1Data.magZField) & 0xFF);
                    }

                    // Task State Variables
                    downlinkPacket[30] =
(uint8_t)(((uint16_t)RTOSstateVars.StateMachineDelay >> 8) &
0xFF) ; // in ms
                    downlinkPacket[31] =
(uint8_t)((uint16_t)RTOSstateVars.StateMachineDelay & 0xFF) ;

                    downlinkPacket[32] =
(uint8_t)(((uint16_t)RTOSstateVars.StateMachineDelayTicks >> 8)
& 0xFF) ;
                    downlinkPacket[33] =
(uint8_t)((uint16_t)RTOSstateVars.StateMachineDelayTicks & 0xFF)
;

                    downlinkPacket[34] =
(uint8_t)(((uint16_t)RTOSstateVars.PollSensorsDelay >> 8) &
0xFF) ;
                    downlinkPacket[35] =
(uint8_t)((uint16_t)RTOSstateVars.PollSensorsDelay & 0xFF) ;

                    downlinkPacket[36] =
(uint8_t)(((uint16_t)RTOSstateVars.PollSensorsDelayTicks >> 8) &
0xFF) ;
```

```
                    downlinkPacket[37] =
(uint8_t)((uint16_t)RTOSstateVars.PollSensorsDelayTicks & 0xFF)
;

                    downlinkPacket[38] =
(uint8_t)(((uint16_t)RTOSstateVars.PollCamerasDelay >> 8) &
0xFF) ;
                    downlinkPacket[39] =
(uint8_t)((uint16_t)RTOSstateVars.PollCamerasDelay & 0xFF) ;

                    downlinkPacket[40] =
(uint8_t)(((uint16_t)RTOSstateVars.PollCamerasDelayTicks >> 8) &
0xFF) ;
                    downlinkPacket[41] =
(uint8_t)((uint16_t)RTOSstateVars.PollCamerasDelayTicks & 0xFF)
;

                    downlinkPacket[42] =
(uint8_t)(((uint16_t)RTOSstateVars.InterpretUplinkDelay >> 8) &
0xFF) ;
                    downlinkPacket[43] =
(uint8_t)((uint16_t)RTOSstateVars.InterpretUplinkDelay & 0xFF) ;

                    downlinkPacket[44] =
(uint8_t)(((uint16_t)RTOSstateVars.InterpretUplinkDelayTicks >>
8) & 0xFF) ;
                    downlinkPacket[45] =
(uint8_t)((uint16_t)RTOSstateVars.InterpretUplinkDelayTicks &
0xFF) ;

                    downlinkPacket[46] =
(uint8_t)(((uint16_t)RTOSstateVars.DetumbleDelay >> 8) & 0xFF) ;
                    downlinkPacket[47] =
(uint8_t)((uint16_t)RTOSstateVars.DetumbleDelay & 0xFF) ;

                    downlinkPacket[48] =
(uint8_t)(((uint16_t)RTOSstateVars.DetumbleDelayTicks >> 8) &
0xFF) ;
                    downlinkPacket[49] =
(uint8_t)((uint16_t)RTOSstateVars.DetumbleDelayTicks & 0xFF) ;
```

```
                    downlinkPacket[50] =
(uint8_t)(((uint16_t)RTOSstateVars.SendAndAckDelay >> 8) & 0xFF)
;
                    downlinkPacket[51] =
(uint8_t)((uint16_t)RTOSstateVars.SendAndAckDelay & 0xFF) ;

                    downlinkPacket[52] =
(uint8_t)(((uint16_t)RTOSstateVars.SendAndAckDelayTicks >> 8) &
0xFF) ;
                    downlinkPacket[53] =
(uint8_t)((uint16_t)RTOSstateVars.SendAndAckDelayTicks & 0xFF) ;

                    downlinkPacket[54] =
(uint8_t)(((uint16_t)RTOSstateVars.MonitorCommandsDelay >> 8) &
0xFF) ;
                    downlinkPacket[55] =
(uint8_t)((uint16_t)RTOSstateVars.MonitorCommandsDelay & 0xFF) ;

                    downlinkPacket[56] =
(uint8_t)(((uint16_t)RTOSstateVars.MonitorCommandsDelayTicks >>
8) & 0xFF) ;
                    downlinkPacket[57] =
(uint8_t)((uint16_t)RTOSstateVars.MonitorCommandsDelayTicks &
0xFF) ;


                    // Requesting and Downlink Timing
                    downlinkPacket[58] =
(uint8_t)((RTOSstateVars.IridiumReqTime >> 24) & 0xFF) ;
                    downlinkPacket[59] =
(uint8_t)((RTOSstateVars.IridiumReqTime >> 16) & 0xFF) ;
                    downlinkPacket[60] =
(uint8_t)((RTOSstateVars.IridiumReqTime >> 8) & 0xFF) ;
                    downlinkPacket[61] =
(uint8_t)(RTOSstateVars.IridiumReqTime & 0xFF) ;

                    downlinkPacket[62] =
(uint8_t)((RTOSstateVars.UplinkReqTime >> 24) & 0xFF) ;
                    downlinkPacket[63] =
(uint8_t)((RTOSstateVars.UplinkReqTime >> 16) & 0xFF) ;
                    downlinkPacket[64] =
(uint8_t)((RTOSstateVars.UplinkReqTime >> 8) & 0xFF) ;
```

```
                    downlinkPacket[65] =
(uint8_t)(RTOSstateVars.UplinkReqTime & 0xFF) ;

                    downlinkPacket[66] =
(uint8_t)((RTOSstateVars.IridiumPacketHighFreq >> 24) & 0xFF) ;
                    downlinkPacket[67] =
(uint8_t)((RTOSstateVars.IridiumPacketHighFreq >> 16) & 0xFF) ;
                    downlinkPacket[68] =
(uint8_t)((RTOSstateVars.IridiumPacketHighFreq >> 8) & 0xFF) ;
                    downlinkPacket[69] =
(uint8_t)(RTOSstateVars.IridiumPacketHighFreq & 0xFF) ;

                    downlinkPacket[70] =
(uint8_t)((RTOSstateVars.IridiumPacketLowFreq >> 24) & 0xFF) ;
                    downlinkPacket[71] =
(uint8_t)((RTOSstateVars.IridiumPacketLowFreq >> 16) & 0xFF) ;
                    downlinkPacket[72] =
(uint8_t)((RTOSstateVars.IridiumPacketLowFreq >> 8) & 0xFF) ;
                    downlinkPacket[73] =
(uint8_t)(RTOSstateVars.IridiumPacketLowFreq & 0xFF) ;

                    // Packet Sequencing
                    downlinkPacket[74] =
(uint8_t)((RTOSstateVars.packetSeqNum >> 8) & 0xFF);
                    downlinkPacket[75] =
(uint8_t)((RTOSstateVars.packetSeqNum) & 0xFF);
                    downlinkPacket[76] =
RTOSstateVars.detumbleSequenceNum ;

                    //Other State Variables
                    downlinkPacket[77] =
(uint8_t)((RTOSstateVars.RTOSStarttime >> 24) & 0xFF);
                    downlinkPacket[78] =
(uint8_t)((RTOSstateVars.RTOSStarttime >> 16) & 0xFF);
                    downlinkPacket[79] =
(uint8_t)((RTOSstateVars.RTOSStarttime >> 8) & 0xFF);
                    downlinkPacket[80] =
(uint8_t)((RTOSstateVars.RTOSStarttime) & 0xFF);

                    downlinkPacket[81] =
(uint8_t)RTOSstateVars.state ;
```

```
                    downlinkPacket[82] =
(uint8_t)RTOSstateVars.prevState ;
                    downlinkPacket[83] =
(uint8_t)RTOSstateVars.newState ;
                    downlinkPacket[84] =
(uint8_t)RTOSstateVars.startUp ;
                    downlinkPacket[85] =
(uint8_t)RTOSstateVars.pollSensors ;
                    downlinkPacket[86] =
(uint8_t)RTOSstateVars.pollCameras ;
                    downlinkPacket[87] =
(uint8_t)RTOSstateVars.detumble ;
                    downlinkPacket[88] =
(uint8_t)RTOSstateVars.flipVoltageX ;
                    downlinkPacket[89] =
(uint8_t)RTOSstateVars.flipVoltageY ;
                    downlinkPacket[90] =
(uint8_t)RTOSstateVars.flipVoltageZ ;
                    downlinkPacket[91] =
(uint8_t)RTOSstateVars.useIMU2 ;
/*
                        Serial.print("Mag X: ");

Serial.println(sensorData.imu1Data.magXField);
                        Serial.print("Mag Y: ");

Serial.println(sensorData.imu1Data.magYField);
                        Serial.print("Mag Z: ");

Serial.println(sensorData.imu1Data.magZField);
                        delay(5);
*/

/*
                    downlinkPacket[30] =
sensorData.magData.magXField;
                    downlinkPacket[31] =
sensorData.magData.magYField;
                    downlinkPacket[32] =
sensorData.magData.magZField;
*/
                }break;
```

```
                case DETUMBLEHIGHRES:
                {
                    downlinkPacket[4] = 0xC5; // high res
detumble packets contain 197 out of 200 bytes
                    downlinkPacket[5] =
(uint8_t)((RTOSstateVars.packetSeqNum >> 8) & 0xFF); // high
byte
                    downlinkPacket[6] =
(uint8_t)(RTOSstateVars.packetSeqNum & 0xFF);; // low byte
                    downlinkPacket[7] = DETUMBLEHIGHRES;

                    downlinkPacket[8  +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)((sensorData.timeStamp >> 24) & 0xFF); // high byte
                    downlinkPacket[9  +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)((sensorData.timeStamp >> 16) & 0xFF);
                    downlinkPacket[10 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)((sensorData.timeStamp >> 8) & 0xFF);
                    downlinkPacket[11 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(sensorData.timeStamp & 0xFF);; // low byte;
/*
                    Serial.print("Time stamp:");
                    Serial.println(sensorData.timeStamp);
*/
                    if(RTOSstateVars.useIMU2){
                        //Acceleration Conversion
                        downlinkPacket[12 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.accX >> 8) & 0xFF);
                        downlinkPacket[13 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.accX) & 0xFF);
                        downlinkPacket[14 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.accY >> 8) & 0xFF);
                        downlinkPacket[15 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.accY) & 0xFF);
```

```
                        downlinkPacket[16 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.accZ >> 8) & 0xFF);
                        downlinkPacket[17 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.accZ) & 0xFF);
                        // Gyro Conversions
                        downlinkPacket[18 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrX*100) >> 8) &
0xFF); // 10s of mdegs/s
                        downlinkPacket[19 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrX*100)) & 0xFF);
                        downlinkPacket[20 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrY*100) >> 8) &
0xFF);
                        downlinkPacket[21 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrY*100)) & 0xFF);
                        downlinkPacket[22 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrZ*100) >> 8) &
0xFF);
                        downlinkPacket[23 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.gyrZ*100)) & 0xFF);
                        // Mag field Conversion
                        downlinkPacket[24 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.magXField) >> 8) &
0xFF); // uT
                        downlinkPacket[25 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.magXField) & 0xFF);
                        downlinkPacket[26 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.magYField) >> 8) &
0xFF);
```

```
                        downlinkPacket[27 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.magYField) & 0xFF);
                        downlinkPacket[28 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu2Data.magZField) >> 8) &
0xFF);
                        downlinkPacket[29 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu2Data.magZField) & 0xFF);
                    }
                    else{
                        //Acceleration Conversion
                        downlinkPacket[12 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.accX >> 8) & 0xFF);
                        downlinkPacket[13 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.accX) & 0xFF);
                        downlinkPacket[14 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.accY >> 8) & 0xFF);
                        downlinkPacket[15 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.accY) & 0xFF);
                        downlinkPacket[16 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.accZ >> 8) & 0xFF);
                        downlinkPacket[17 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.accZ) & 0xFF);
/*
                        Serial.print("Acc X: ");

Serial.println(sensorData.imu1Data.accX);
                        Serial.print("Acc Y: ");

Serial.println(sensorData.imu1Data.accY);
                        Serial.print("Acc Z: ");

Serial.println(sensorData.imu1Data.accZ);
                        delay(5);
```

```
*/
                        // Gyro Conversions
                        downlinkPacket[18 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrX*100) >> 8) &
0xFF); // 10s of mdegs/s
                        downlinkPacket[19 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrX*100)) & 0xFF);
                        downlinkPacket[20 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrY*100) >> 8) &
0xFF);
                        downlinkPacket[21 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrY*100)) & 0xFF);
                        downlinkPacket[22 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrZ*100) >> 8) &
0xFF);
                        downlinkPacket[23 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.gyrZ*100)) & 0xFF);
/*
                        Serial.print("Gyro X: ");

Serial.println(sensorData.imu1Data.gyrX);
                        Serial.print("Gyro Y: ");

Serial.println(sensorData.imu1Data.gyrY);
                        Serial.print("Gyro Z: ");

Serial.println(sensorData.imu1Data.gyrZ);
                        delay(5);
*/
                        // Mag field Conversion
                        downlinkPacket[24 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.magXField) >> 8) &
0xFF); // uT
```

```
                        downlinkPacket[25 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.magXField) & 0xFF);
                        downlinkPacket[26 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.magYField) >> 8) &
0xFF);
                        downlinkPacket[27 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.magYField) & 0xFF);
                        downlinkPacket[28 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)(sensorData.imu1Data.magZField) >> 8) &
0xFF);
                        downlinkPacket[29 +
(RTOSstateVars.detumbleSequenceNum* 22)] =
(uint8_t)(((int16_t)sensorData.imu1Data.magZField) & 0xFF);
/*
                        Serial.print("Mag X: ");

Serial.println(sensorData.imu1Data.magXField);
                        Serial.print("Mag Y: ");

Serial.println(sensorData.imu1Data.magYField);
                        Serial.print("Mag Z: ");

Serial.println(sensorData.imu1Data.magZField);
                        delay(5);
*/
                    }
                }break;
/*
                case DETUMBLELOWRES:
                {
                    downlinkPacket[4] = 0xC5; // low res
detumble packets contain 197 out of 200 bytes
                    downlinkPacket[5] = (uint8_t)((packetSeqNum
>> 8) & 0xFF); // high byte
                    downlinkPacket[6] = (uint8_t)(packetSeqNum &
0xFF);; // low byte
                    downlinkPacket[7] = DETUMBLELOWRES;
                }break;
```

```
*/
/*
                case IMAGE:
                {
                    downlinkPacket[4] = 0xC4; // Image packets
contain 196 out of 200 bytes
                    downlinkPacket[5] = (uint8_t)((packetSeqNum
>> 8) & 0xFF); // high byte
                    downlinkPacket[6] = (uint8_t)(packetSeqNum &
0xFF);; // low byte
                    downlinkPacket[7] = IMAGE;
                }break;
*/
                default:
                {
                    Serial.println("Unable to interpret F5
packet type request.");
                }break;
            }
        }break;
        case S4RECONFIG:
        {
            downlinkPacket[4] = 0x30;
            downlinkPacket[5] = 0x60;
            downlinkPacket[6] = 0x6A;
        }break;
        case CHKUPLINK:
        {
            // No further information needs adding
        }break;
        case NETREQ:
        {
            // No further information needs adding
        }break;
        default:
        {
            Serial.println("Unable to interpret packet
request.");
        }break;
    }
}
```

```
void fliterAndPollIMU(ICM_20948_I2C &myICM, SensorData
&sensorData, sensorOffsets &sensorOffsetBoardNum,
tempOffsetCalcHolder &corrValStruct, bool startUp, int AD0val){

    if(AD0val == IMU1_AD0_VAL){
        corrValStruct.finalXGyro = (myICM.gyrX()) -
sensorOffsetBoardNum.icm1Offsets[0][0];
        corrValStruct.finalYGyro = (myICM.gyrY()) -
sensorOffsetBoardNum.icm1Offsets[0][1];
        corrValStruct.finalZGyro = (myICM.gyrZ()) -
sensorOffsetBoardNum.icm1Offsets[0][2];

        corrValStruct.finalXAccel = (myICM.accX()) -
(sensorOffsetBoardNum.icm1Offsets[1][0]);
        corrValStruct.finalYAccel = (myICM.accY()) -
sensorOffsetBoardNum.icm1Offsets[1][1];
        corrValStruct.finalZAccel = (myICM.accZ()) -
sensorOffsetBoardNum.icm1Offsets[1][2];

        corrValStruct.correctedXMag = (myICM.magX()) -
sensorOffsetBoardNum.icm1Offsets[2][0];
        corrValStruct.correctedYMag = (myICM.magY()) -
sensorOffsetBoardNum.icm1Offsets[2][1];
        corrValStruct.correctedZMag = (myICM.magZ()) -
sensorOffsetBoardNum.icm1Offsets[2][2];

        corrValStruct.finalXMag =
sensorOffsetBoardNum.icm1SoftIron[0][0] *
corrValStruct.correctedXMag +

sensorOffsetBoardNum.icm1SoftIron[0][1] *
corrValStruct.correctedYMag +

sensorOffsetBoardNum.icm1SoftIron[0][2] *
corrValStruct.correctedZMag;

        corrValStruct.finalYMag =
sensorOffsetBoardNum.icm1SoftIron[1][0] *
corrValStruct.correctedXMag +

sensorOffsetBoardNum.icm1SoftIron[1][1] *
corrValStruct.correctedYMag +
```

```
sensorOffsetBoardNum.icm1SoftIron[1][2] *
corrValStruct.correctedZMag;

        corrValStruct.finalZMag =
sensorOffsetBoardNum.icm1SoftIron[2][0] *
corrValStruct.correctedXMag +

sensorOffsetBoardNum.icm1SoftIron[2][1] *
corrValStruct.correctedYMag +

sensorOffsetBoardNum.icm1SoftIron[2][2] *
corrValStruct.correctedZMag;

    }
    else if(AD0val == IMU2_AD0_VAL){
        corrValStruct.finalXGyro = (myICM.gyrX()) -
sensorOffsetBoardNum.icm2Offsets[0][0];
        corrValStruct.finalYGyro = (myICM.gyrY()) -
sensorOffsetBoardNum.icm2Offsets[0][1];
        corrValStruct.finalZGyro = (myICM.gyrZ()) -
sensorOffsetBoardNum.icm2Offsets[0][2];

        corrValStruct.finalXAccel = (myICM.accX()) -
(sensorOffsetBoardNum.icm2Offsets[1][0]);
        corrValStruct.finalYAccel = (myICM.accY()) -
sensorOffsetBoardNum.icm2Offsets[1][1];
        corrValStruct.finalZAccel = (myICM.accZ()) -
sensorOffsetBoardNum.icm2Offsets[1][2];

        corrValStruct.correctedXMag = (myICM.magX()) -
sensorOffsetBoardNum.icm2Offsets[2][0];
        corrValStruct.correctedYMag = (myICM.magY()) -
sensorOffsetBoardNum.icm2Offsets[2][1];
        corrValStruct.correctedZMag = (myICM.magZ()) -
sensorOffsetBoardNum.icm2Offsets[2][2];

        corrValStruct.finalXMag =
sensorOffsetBoardNum.icm2SoftIron[0][0] *
corrValStruct.correctedXMag +
```

```
sensorOffsetBoardNum.icm2SoftIron[0][1] *
corrValStruct.correctedYMag +

sensorOffsetBoardNum.icm2SoftIron[0][2] *
corrValStruct.correctedZMag;

        corrValStruct.finalYMag =
sensorOffsetBoardNum.icm2SoftIron[1][0] *
corrValStruct.correctedXMag +

sensorOffsetBoardNum.icm2SoftIron[1][1] *
corrValStruct.correctedYMag +

sensorOffsetBoardNum.icm2SoftIron[1][2] *
corrValStruct.correctedZMag;

        corrValStruct.finalZMag =
sensorOffsetBoardNum.icm2SoftIron[2][0] *
corrValStruct.correctedXMag +

sensorOffsetBoardNum.icm2SoftIron[2][1] *
corrValStruct.correctedYMag +

sensorOffsetBoardNum.icm2SoftIron[2][2] *
corrValStruct.correctedZMag;

    }

    if(startUp){ // No filtering if it is the first time
polling.
        if(AD0val == IMU1_AD0_VAL){
            // Poll Gyro Data (dps):
            sensorData.imu1Data.gyrX = corrValStruct.finalXGyro;
            sensorData.imu1Data.gyrY = corrValStruct.finalYGyro;
            sensorData.imu1Data.gyrZ = corrValStruct.finalZGyro;
            // Poll Acceleration Data (mg):
            sensorData.imu1Data.accX =
corrValStruct.finalXAccel;
            sensorData.imu1Data.accY =
corrValStruct.finalYAccel;
```

```
            sensorData.imu1Data.accZ =
corrValStruct.finalZAccel;
            // Poll Magnet Field (uT):
            sensorData.imu1Data.magXField =
corrValStruct.finalXMag;
            sensorData.imu1Data.magYField =
corrValStruct.finalYMag;
            sensorData.imu1Data.magZField = -
corrValStruct.finalZMag;
        }
        else if(AD0val == IMU2_AD0_VAL){
            // Poll Gyro Data (dps):
            sensorData.imu2Data.gyrX = corrValStruct.finalXGyro;
            sensorData.imu2Data.gyrY = corrValStruct.finalYGyro;
            sensorData.imu2Data.gyrZ = corrValStruct.finalZGyro;
            // Poll Acceleration Data (mg):
            sensorData.imu2Data.accX =
corrValStruct.finalXAccel;
            sensorData.imu2Data.accY =
corrValStruct.finalYAccel;
            sensorData.imu2Data.accZ =
corrValStruct.finalZAccel;
            // Poll Magnet Field (uT):
            sensorData.imu2Data.magXField =
corrValStruct.finalXMag;
            sensorData.imu2Data.magYField =
corrValStruct.finalYMag;
            sensorData.imu2Data.magZField = -
corrValStruct.finalZMag;
        }
    }
    else{
        if(AD0val == IMU1_AD0_VAL){
            // Filter data after correcting for offsets and
store new filtered value
            sensorData.imu1Data.gyrX = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalXGyro) +
sensorData.imu1Data.gyrX);
            sensorData.imu1Data.gyrY = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalYGyro) +
sensorData.imu1Data.gyrY);
```

```
            sensorData.imu1Data.gyrZ = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalZGyro) +
sensorData.imu1Data.gyrZ);

            sensorData.imu1Data.accX = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalXAccel) +
sensorData.imu1Data.accX);
            sensorData.imu1Data.accY = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalYAccel) +
sensorData.imu1Data.accY);
            sensorData.imu1Data.accZ = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalZAccel) +
sensorData.imu1Data.accZ);

            sensorData.imu1Data.magXField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalXMag) +
sensorData.imu1Data.magXField);
            sensorData.imu1Data.magYField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalYMag) +
sensorData.imu1Data.magYField);
            sensorData.imu1Data.magZField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalZMag) +
sensorData.imu1Data.magZField);
        }
        else if(AD0val == IMU2_AD0_VAL){
            // Filter data after correcting for offsets and
store new filtered value
            sensorData.imu2Data.gyrX = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalXGyro) +
sensorData.imu2Data.gyrX);
            sensorData.imu2Data.gyrY = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalYGyro) +
sensorData.imu2Data.gyrY);
            sensorData.imu2Data.gyrZ = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalZGyro) +
sensorData.imu2Data.gyrZ);

            sensorData.imu2Data.accX = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalXAccel) +
sensorData.imu2Data.accX);
```

```
            sensorData.imu2Data.accY = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalYAccel) +
sensorData.imu2Data.accY);
            sensorData.imu2Data.accZ = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalZAccel) +
sensorData.imu2Data.accZ);

            sensorData.imu2Data.magXField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalXMag) +
sensorData.imu2Data.magXField);
            sensorData.imu2Data.magYField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalYMag) +
sensorData.imu2Data.magYField);
            sensorData.imu2Data.magZField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalZMag) +
sensorData.imu2Data.magZField);
        }
    }
}

void fliterAndPollMag(sensors_event_t &event, SensorData
&sensorData, sensorOffsets &sensorOffsetBoardNum,
tempOffsetCalcHolder &corrValStruct, bool startUp){

    corrValStruct.correctedXMag = (event.magnetic.x) -
sensorOffsetBoardNum.lisHardIron[0];
    corrValStruct.correctedYMag = (event.magnetic.y) -
sensorOffsetBoardNum.lisHardIron[1];
    corrValStruct.correctedZMag = (event.magnetic.z) -
sensorOffsetBoardNum.lisHardIron[2];

    corrValStruct.finalXMag =
sensorOffsetBoardNum.lisSoftIron[0][0] *
corrValStruct.correctedXMag +

sensorOffsetBoardNum.lisSoftIron[0][1] *
corrValStruct.correctedYMag +

sensorOffsetBoardNum.lisSoftIron[0][2] *
corrValStruct.correctedZMag;
```

```
    corrValStruct.finalYMag =
sensorOffsetBoardNum.lisSoftIron[1][0] *
corrValStruct.correctedXMag +

sensorOffsetBoardNum.lisSoftIron[1][1] *
corrValStruct.correctedYMag +

sensorOffsetBoardNum.lisSoftIron[1][2] *
corrValStruct.correctedZMag;

    corrValStruct.finalZMag =
sensorOffsetBoardNum.lisSoftIron[2][0] *
corrValStruct.correctedXMag +

sensorOffsetBoardNum.lisSoftIron[2][1] *
corrValStruct.correctedYMag +

sensorOffsetBoardNum.lisSoftIron[2][2] *
corrValStruct.correctedZMag;

    if(startUp){ // No filtering if it is the first time
polling.
        // Poll Acceleration Data (mg):
        sensorData.magData.magXField = corrValStruct.finalXMag;
        sensorData.magData.magYField = corrValStruct.finalYMag;
        sensorData.magData.magZField = corrValStruct.finalZMag;
    }
    else{
        // Poll Acceleration Data (mg):
        sensorData.magData.magXField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalXMag) +
sensorData.magData.magXField);
        sensorData.magData.magYField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(corrValStruct.finalYMag) +
sensorData.magData.magYField);
        sensorData.magData.magZField = (1/(LPF_GAIN + 1)) *
(LPF_GAIN*(-corrValStruct.finalZMag) +
sensorData.magData.magZField);
    }
}
```

References

[1] NearSpace Launch, “0.5U ThinSat Interface Control Document (ICD)”, Dream Big Program satellite ICD, Sept. 2024 [Revised Mar. 2025].