



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**DESIGN, DEVELOPMENT, AND FLIGHT TESTING
OF A MODULAR AVIONICS AND VEHICLE
MONITORING SYSTEM**

by

Allison D. Adamos

December 2021

Thesis Advisor:
Co-Advisor:

Christopher M. Brophy
Carson R. Vogt

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC, 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 2021	3. REPORT TYPE AND DATES COVERED Master's thesis		
4. TITLE AND SUBTITLE DESIGN, DEVELOPMENT, AND FLIGHT TESTING OF A MODULAR AVIONICS AND VEHICLE MONITORING SYSTEM			5. FUNDING NUMBERS R4Q04	
6. AUTHOR(S) Allison D. Adamos				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) CRUSER Monterey, CA, 93942			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) As more rocket components become commercially available, the feasibility of a low-cost rocket-powered vehicle with capabilities tailored to the needs of the Navy, such as delivering payloads, countering drones, and reliable communications, increases. The avionics bay of such a system is essential as it provides guidance and control to the vehicle and can incorporate a vehicle health monitoring system to transmit vital parameters and document various issues that can arise during flight, which could lead to unexpected flight performance and/or failure. A modular avionics bay was designed to provide flexibility, allow for new hardware requirements and future system improvements, and ensure reliable and proper connections for electronic components to sustain tactical launch and flight vibrations. This effort seeks to give the U.S. Navy more cost-symmetric options and capabilities for a rocket-powered vehicle that adapts to changing mission requirements and technology. A data logging system with a high sampling rate and telemetry capability was also developed to document the forces imposed on the flight system and collect future flight-test data. This data will allow a representative structural vibration spectrum of a delivery vehicle over tactical flight profiles to be constructed, such that future avionics components be certified pre-flight utilizing a shaker table and an established protocol.				
14. SUBJECT TERMS modular avionics, flight performance, vehicle health monitoring, telemetry, cost analysis			15. NUMBER OF PAGES 115	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**DESIGN, DEVELOPMENT, AND FLIGHT TESTING OF A MODULAR
AVIONICS AND VEHICLE MONITORING SYSTEM**

Allison D. Adamos
Lieutenant, United States Navy
BS, University of San Diego, 2016
BA, University of San Diego, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
December 2021**

Approved by: Christopher M. Brophy
Advisor

Carson R. Vogt
Co-Advisor

Garth V. Hobson
Chair, Department of Mechanical and Aerospace Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

As more rocket components become commercially available, the feasibility of a low-cost rocket-powered vehicle with capabilities tailored to the needs of the Navy, such as delivering payloads, countering drones, and reliable communications, increases. The avionics bay of such a system is essential as it provides guidance and control to the vehicle and can incorporate a vehicle health monitoring system to transmit vital parameters and document various issues that can arise during flight, which could lead to unexpected flight performance and/or failure.

A modular avionics bay was designed to provide flexibility, allow for new hardware requirements and future system improvements, and ensure reliable and proper connections for electronic components to sustain tactical launch and flight vibrations. This effort seeks to give the U.S. Navy more cost-symmetric options and capabilities for a rocket-powered vehicle that adapts to changing mission requirements and technology.

A data logging system with a high sampling rate and telemetry capability was also developed to document the forces imposed on the flight system and collect future flight-test data. This data will allow a representative structural vibration spectrum of a delivery vehicle over tactical flight profiles to be constructed, such that future avionics components be certified pre-flight utilizing a shaker table and an established protocol.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	BACKGROUND	1
B.	MOTIVATION	6
1.	Survivability and Fault Tree Analysis	8
C.	OBJECTIVES	11
II.	DESIGN PROCESS.....	13
A.	NPS RPV OVERVIEW	13
B.	AV BAY	19
1.	Approach	19
2.	Avionics Bay Version 1	21
3.	Avionics Bay Version 2	23
4.	Avionics Bay Version 3	26
C.	TELEMETRY/ HEALTH MONITORING SYSTEM.....	29
D.	DATA LOGGER.....	30
III.	RESULTS AND DISCUSSION	35
A.	AV BAY	35
B.	DATA LOGGER.....	39
C.	TELEMETRY/ HEALTH MONITORING SYSTEM.....	40
D.	SHAKER TABLE	42
IV.	COST ANALYSIS	51
V.	SUMMARY AND CONCLUSION	55
VI.	FUTURE WORK RECOMMENDATIONS	57
	APPENDIX A. CONNECTIONS	59
A.	RPI.....	59
B.	DATA LOGGER.....	60
C.	TELEMETRY	60
	APPENDIX B. RPV COMPONENT COSTS	63
	APPENDIX C. ARDUINO CODE	65
A.	GROUND STATION UNO	65

B.	ROCKET TRANSMITTER	66
C.	DATA LOGGER.....	68
D.	BIN TO CSV CONVERSION	76
LIST OF REFERENCES		91
INITIAL DISTRIBUTION LIST		95

LIST OF FIGURES

Figure 1.	2016 NPS RPV concept and design. Source: [6].	2
Figure 2.	2017 NPS RPV concept. Source: [7].	3
Figure 3.	Bomblet SolidWorks model. Source: [8].	4
Figure 4.	2019 NPS RPV concept and design. Source: [7].	4
Figure 5.	Multi-munition concept. Source: [9].	5
Figure 6.	2019 NPS RPV concept and design for munitions. Source: [9].	5
Figure 7.	2016 and 2019 NPS RPV electronics wiring and mounting. Sources: [5], [7].	6
Figure 8.	Previous NPS AV Bay designs	7
Figure 9.	Separation FTA. Adapted from [11].	9
Figure 10.	Data FTA. Adapted from [11].	10
Figure 11.	AV Bay structural integrity FTA. Adapted from [11].	10
Figure 12.	NPS RPV prior to launch	13
Figure 13.	NPS RPV on launch pad, at launch, and recovery	13
Figure 14.	RPV interstage	14
Figure 15.	RPV interstage top and bottom views	14
Figure 16.	Health monitoring system location in the interstage	15
Figure 17.	Telemetry and separation electronics	15
Figure 18.	RPV booster	16
Figure 19.	AV Bay components	18
Figure 20.	AV Bay location in the booster	18
Figure 21.	CubeSat examples. Sources: [12], [13].	19
Figure 22.	Initial rail and AV Bay concept	20

Figure 23.	3D printed rail	20
Figure 24.	Harwin connectors. Sources: [14], [15].	21
Figure 25.	Connection of Harwin connectors	21
Figure 26.	AV Bay flown on 17APR21 launch.....	22
Figure 27.	AV Bay flown on 17JUL21 and 21AUG21 launch.....	23
Figure 28.	RPi GPIO expansion board and RPi HAT. Source: [16].	24
Figure 29.	Female and male six-pin mil-spec connector. Sources: [17], [18].	24
Figure 30.	RPi LAN connection.....	25
Figure 31.	Still shot of forward GoPro footage 21AUG21 launch	25
Figure 32.	AV Bay pieces after impact and reconstructed platters	26
Figure 33.	Electronic components after impact.....	26
Figure 34.	AV Bay flown on 22OCT21 launch	27
Figure 35.	Top view of RPi and Arduino platters with electronics mounted.....	28
Figure 36.	Top view of the two bottom platters with electronics mounted.....	28
Figure 37.	Wire cutouts	28
Figure 38.	RPi heat sink and fan	29
Figure 39.	Reyax Rylr898 LoRa	29
Figure 40.	Ground station receiver and RPV transmitter.....	30
Figure 41.	Data logger.....	32
Figure 42.	Kate receiver (left) & transmitter (right). Source: [24].....	36
Figure 43.	Recovery of the booster with a zoomed in view.....	37
Figure 44.	Arduino and RPi with ADC after impact.....	37
Figure 45.	BNO055 and relay after impact	37
Figure 46.	Damages sustained by the booster and interstage.....	38
Figure 47.	AV bay components pre-flight and post impact	38

Figure 48.	Still shot of forward GoPro footage 22OCT21 launch	39
Figure 49.	Damages between booster and interstage (left) and sled coupler (right)	40
Figure 50.	Telemetry components post impact and damaged pressure transducer	40
Figure 51.	NPS shaker table and data sheet shaker table. Source: [26].	42
Figure 52.	Accelerometer mounted to a platter	43
Figure 53.	AV Bay mounted to shaker table for horizontal testing	44
Figure 54.	ICP accelerometer mounted to AV Bay.....	44
Figure 55.	AV Bay mounted to the shaker table for vertical testing.....	45
Figure 56.	Shaker table test flow. Source: [27].	45
Figure 57.	Shaker table test equipment. Source: [27].	46
Figure 58.	Horizontal and vertical shaker table testing of the AV Bay	46
Figure 59.	Horizontal vibrations test.....	47
Figure 60.	X-axis sine sweep test.....	47
Figure 61.	Y-axis sine sweep test.....	48
Figure 62.	Z-axis sine sweep test	48
Figure 63.	Structural failure for vertical testing.....	49
Figure 64.	Initial vertical sine sweep.....	49
Figure 65.	PC print orientation.....	52
Figure 66.	Tough PLA shell print orientation	53
Figure 67.	Stackable bay SolidWorks concept.....	57
Figure 68.	CubeSat battery matrix. Source: [28].	57

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Flight test log	35
Table 2.	Kate flight data.....	36
Table 3.	LoRa radio vs. Kate	41
Table 4.	Pressure data at separation	41
Table 5.	Random vibrations test specifications. Source: [27].....	43
Table 6.	Overall RPV cost	51
Table 7.	Cost of AV Bay components	51
Table 8.	3D printing time and cost of PC AV Bay	52
Table 9.	3D printing time and cost of tough PLA AV Bay	53
Table 10.	RPi connections to other components.....	59
Table 11.	Connections for the six-pin connector	59
Table 12.	ADXL connections to the shield.....	60
Table 13.	RPV transmitter shield connections.....	60
Table 14.	Ground station receiver shield connections	61

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AGL	above ground level
APKWS	advanced precision kill weapon system
ASD	acceleration spectrum density
AV	avionics
COTS	commercial off the shelf
CPU	central processing unit
CRSA	crushable rear shock absorber
CSV	common separated values
DOF	degrees of freedom
DR	descent rate
DUT	device under test
E-match	electronic matches
F2M2	forward firing miniature munition
FTA	fault tree analysis
G	gravity
GEVS	general environmental verification standard
GNC	guidance, navigation, and control
GPIO	general purpose input/output
GPS	global positioning system
HAT	hardware attached on top
I2C	inter-integrated circuit
ICP	integrated circuit piezoelectric
IDE	integrated development environment
IMU	inertial measurement unit
INS	inertial navigation system
LAN	local area network
LiPo	lithium-ion polymer
LoRa	long range
Mil-Spec	military specific
NAR	national association of rocketry

NPS	Naval Postgraduate School
OEM	original equipment manufacturer
PC	polycarbonate
PF	perfect flite
PLA	polylactic acid
PSI	pounds per square inch
RCAF	Royal Canadian Air Force
RPi	raspberry-pi
RPV	rocket-power vehicle
RTV	room-temperature-vulcanizing
SD	secure digital
SPI	serial peripheral interface
TD	touchdown
UAV	unmanned aerial vehicle
WRC	wireless remote control

ACKNOWLEDGMENTS

I would like to thank the NPS rocket lab staff, specifically my advisor, Dr. Chris Brophy, my co-advisor, Dr. Carson Vogt, and my friend, Alexis Thoeny for guiding me through what was literally rocket science. Special shout out to LCDR Kyle Decker, USN, for being such a supportive teammate throughout this project because without you I would not be known as “shorty.” I would also like to thank my cohort, better known as the 7 Amechos, for all the help throughout the various classes.

I would like to thank my friends for all the unbelievable memories and the countless laughs that have accompanied them. I would also like to thank my sister, brother-in law, niece, and nephew for allowing me the joys of a family without the hard work. I would like to thank my parents, who both immigrated to the United States from the Philippines, for their unwavering support and unconditional love, I share this accomplishment with them.

My dad, Artel Adamos, completed three years of apprenticeship at Naval Base Ship Repair Facility (SRF), Subic Bay, Philippines, where following he enlisted in the U.S. Navy through the Philippine Enlistment Program (PEP) and served 20 honorable years from 29NOV82 until he retired on 02DEC02 as a Machinery Repairman (MR). I am honored to follow in his footsteps as an engineer in the U.S. Navy.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

Guided missiles for military applications are very expensive as they are meant to target, counteract, or defend high value assets. In contrast, unmanned aerial vehicles (UAVs) and other threats are becoming cheaper and easier to weaponize for offensive purposes. Guided missiles, like the Tomahawk, Javelin, and Hellfire have estimated costs of \$1.5 million, \$147,000, and \$115,000 respectively. A low-cost version is the Advanced Precision Kill Weapon System (APKWS II) and costs \$28,000, while the Forward Firing Miniature Munition (F2M2), costs \$5,000 and is intended to defend against some individual UAVs at close range [1]. In contrast, “a moderate mini or micro-UAV can cost a few thousand U.S. dollars” [2]. Due to their relative low costs, multiple or a swarm of UAV’s can easily be produced and deployed by an adversary.

Various countries have swarm UAV programs, such as the “French Icarus project, the Russian Lightning, the Spanish RAPAZ, the U.K.’s Blue Bear swarm and the UAE/South African N-Raven as well as the U.S. Marine Corps with kamikaze drone swarms“ [3]. Real world events utilizing drones to attack have occurred. “Between 1994 and 2018, more than 14 planned or attempted terrorist attacks took place using aerial drones ... to conduct many different types of operations, including intelligence collection, explosive delivery, and chemical weapon delivery” [4]. For example, in “January 2018, Russia’s Khmeimim airbase in western Syria spotted 13 incoming drones at low level. The Russians shot down seven drones and jammed the remaining six. It was the first recorded instance of a mass-drone attack by non-state actors in a combat operation” [5]. UAV swarms are very effective, as only a few are needed to inflict the desired damage as the number of drones in a swarm increases and the ability to detect swarms of UAVs at a distance also improves.

The necessity to develop a flight vehicle to rapidly deploy multiple munitions to counteract and defeat these potential threats with a low cost and symmetric response rises. Rocket-powered and other flight vehicles are developed and constructed for various

applications by hobbyists, academic groups, and businesses. These efforts range from hobbyist-level activities occurring throughout the country within groups such as National Association of Rocketry (NAR) and Tripoli Rocketry Association, to colleges and universities, to small businesses like Firefly Aerospace.

The Naval Postgraduate School (NPS) has been developing a rocket-powered vehicle (RPV) over the past five years, but largely as an academic exercise. Various improvements to the system are occurring to result in a more reliable, configurable, and responsive system. In 2016, ENS Fletcher Rydalch, USN, developed an RPV, as seen in Figure 1. His areas of focus included aerodynamics servo motor control, structural design for the nose cone, fins, and separation, inertial navigation system (INS)/ global positioning system (GPS) recording, guidance, navigation, and control (GNC) development, and parachute deployment. The 2016 RPV utilized an Arduino Uno processor, Adafruit 10 degrees of freedom (DOF) inertial measurement unit (IMU), and Raven altimeter for electronics.

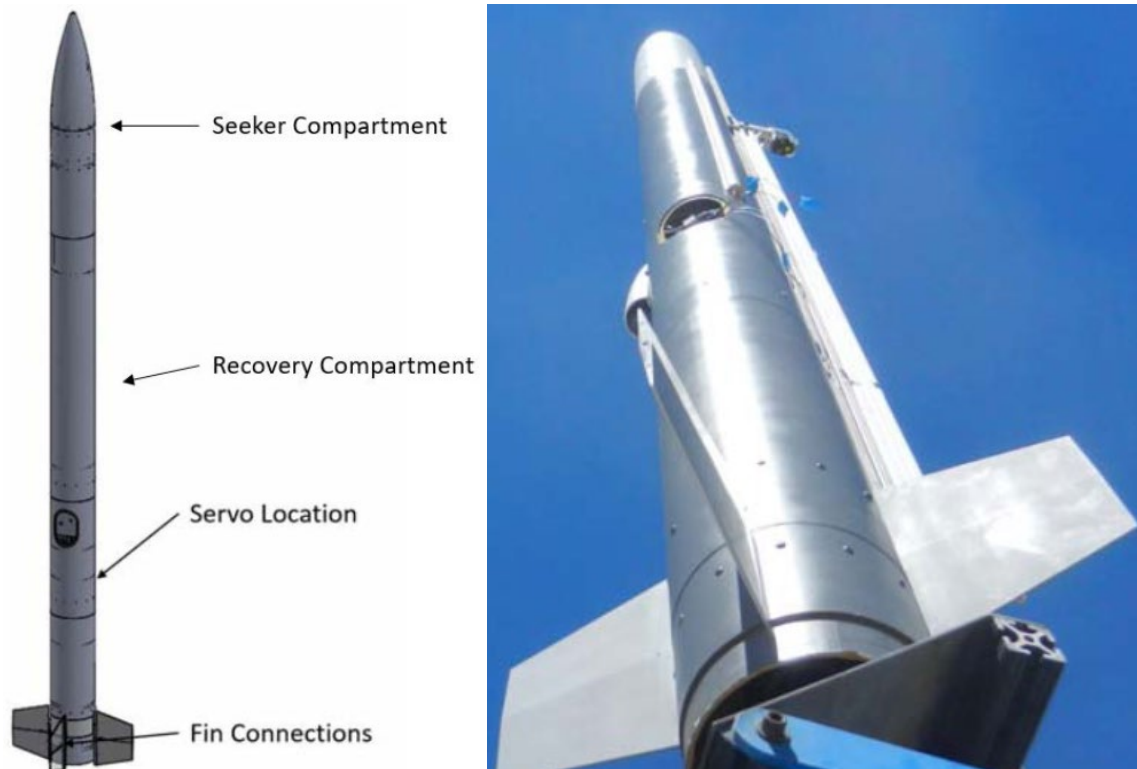


Figure 1. 2016 NPS RPV concept and design. Source: [6].

In 2017, CAPT Kai Grohe, Royal Canadian Air Force (RCAF) further developed the RPV, as seen in Figure 2, and moved the control section forward from the aft end. His areas of focus included aerodynamics, fin design and actuation, and separation. For electronics, the RPV utilized an Arduino Due processor, BNO055 IMU, Perfect Flite (PF) altimeter, and Raspberry Pi 3 processor for data logging.

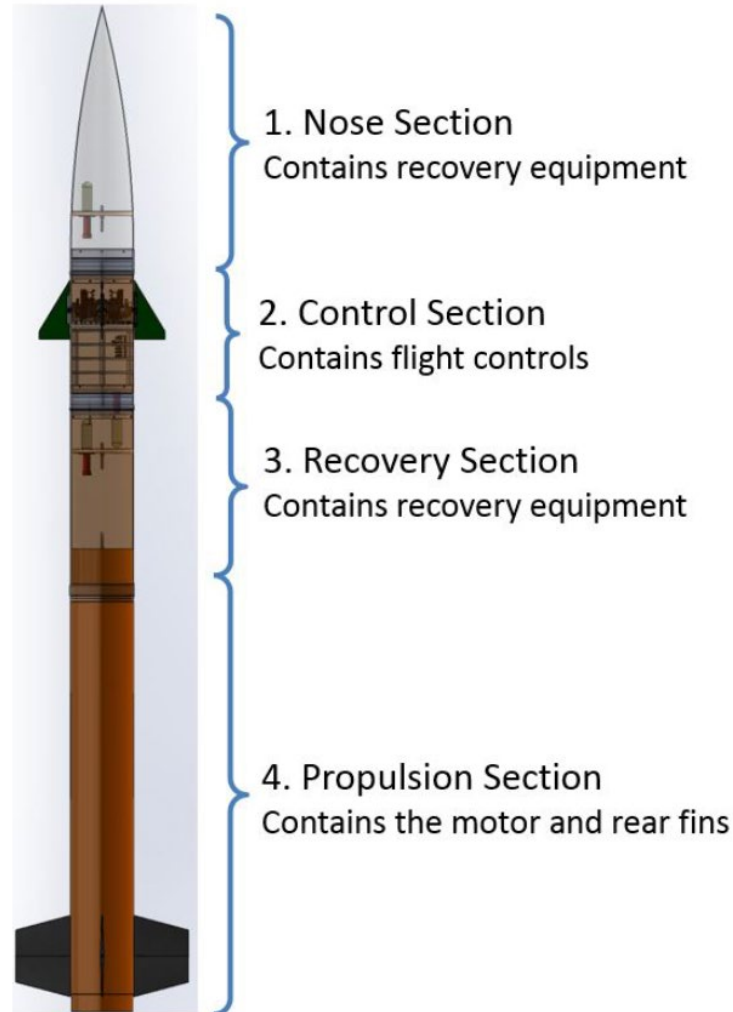


Figure 2. 2017 NPS RPV concept. Source: [7].

In 2018, CAPT Keith Lobo, RCAF, utilized the RPV developed by CAPT Grohe and focused on submunition deployment, as seen in Figure 3, which involved the deployment of a single submunition unit from the nose cone of the vehicle.

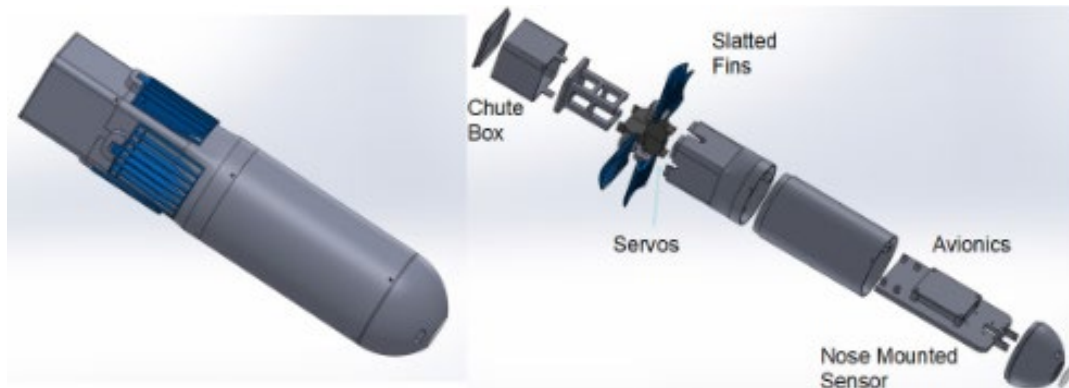


Figure 3. Bomblet SolidWorks model. Source: [8].

In 2019, LT Matthew Busta, USN, continued to develop the RPV, as seen in Figure 4, and his areas of focus included GNC, structures, aerodynamics, and aft fin actuation. In the same year, LT Robert Thyberg, USN, focused on multi-munition deployment, seen in Figure 5, and target analysis for his RPV, seen in Figure 6. The RPV utilized a Raven 4 altimeter, Missile Works wireless remote control (WRC) and MPRLS port pressure sensor for electronics.

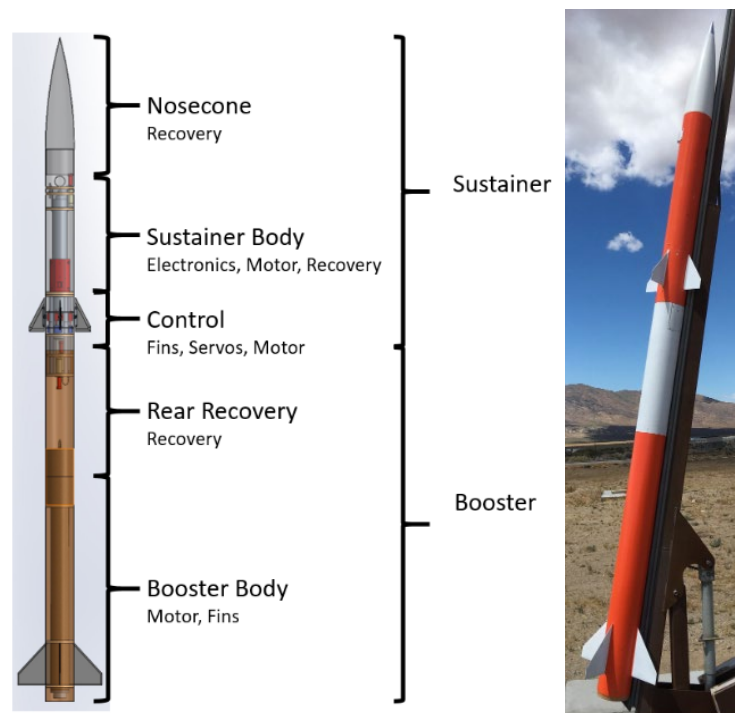


Figure 4. 2019 NPS RPV concept and design. Source: [7].

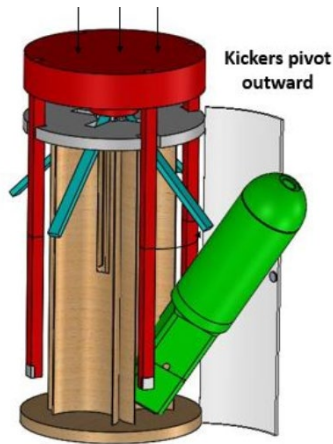


Figure 5. Multi-munition concept. Source: [9].

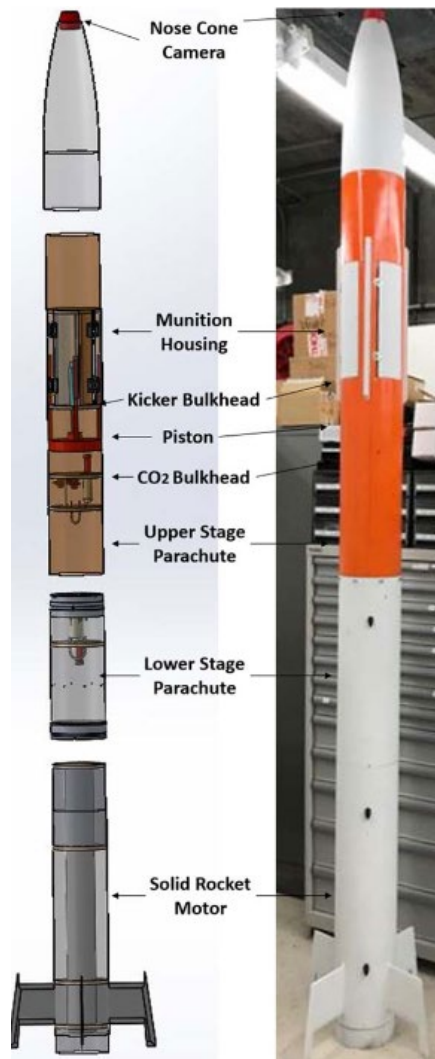
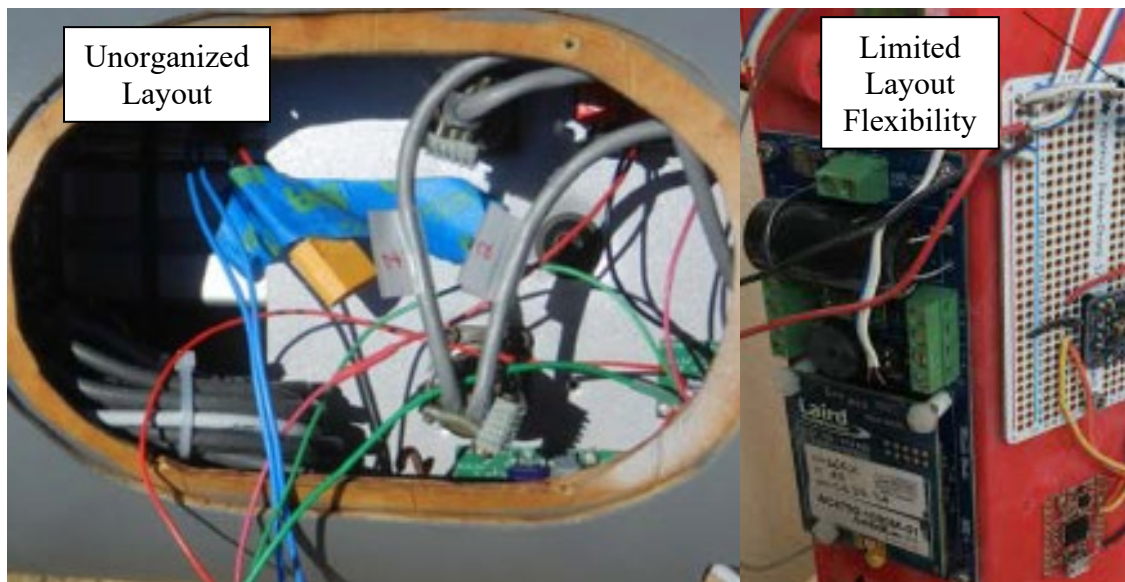


Figure 6. 2019 NPS RPV concept and design for munitions. Source: [9].

B. MOTIVATION

Due to various students working on the general vehicle layout over years without sufficient overlap, a proper turnover of knowledge has been lacking and has consistently resulted in a steep learning curve for the next group. This has resulted in a lack of standardization and a low maturity level. Previous vehicles have been piecemealed ad hoc with haphazard wiring, inconsistent assembly approaches, and short development timelines, which have led to a loss of connection, power, data, and sensor fusion during flight-testing, as seen in Figure 7.

Five previous RPV theses have focused on the nose cone, control surface actuation, separation, guidance, navigation, and control (GNC), parachute deployment, payload, and propulsion, but there has yet to be a direct focus on the integration and design process for a universal Avionics Bay, referred to as the “AV” Bay as well as a telemetry system, which is vital for obtaining real-time data during a flight test. The lack of standardization of past AV Bays has led to a lack of modularity, flexibility, and reliability.



The left image is the parachute bay in the aft section and the right image is the mounting for the electronics in the forward section.

Figure 7. 2016 and 2019 NPS RPV electronics wiring and mounting.
Sources: [5], [7].

Past attempts at general AV Bay designs are shown in Figure 8, where the electronics are not easily accessible, the wiring is unorganized, loose or weak, memory cards are unsecured, and the connections are not standard. Due to the high dynamic loads of the shock cord deployment and main parachute deployment process, the associated loads through the coupler portion of the RPV, which typically has the flight electronics often resulted in some form of damage and was often a high failure point, as seen in Figure 8. The coupler unit was recently replaced with an aluminum unit.

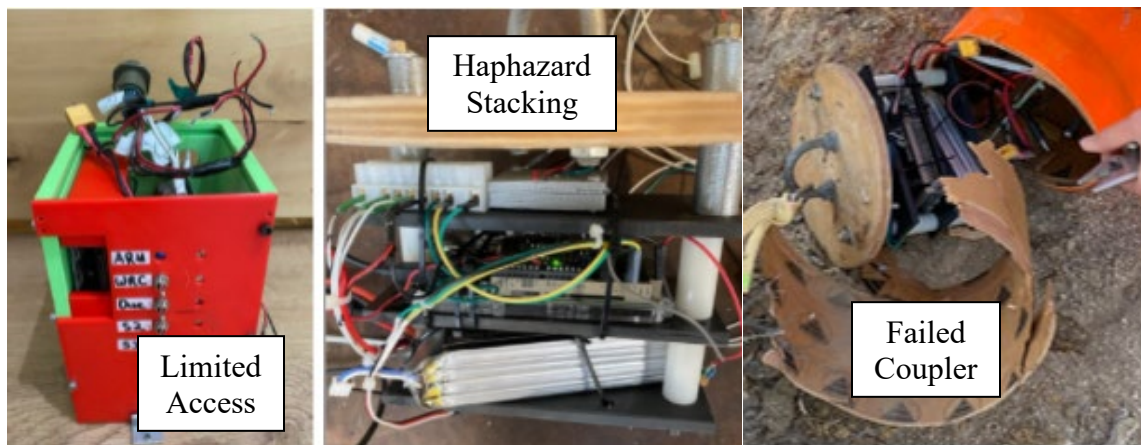


Figure 8. Previous NPS AV Bay designs

Although various ground tests have been executed prior to launch to ensure the RPV is qualified for flight, such as testing out electronics and CO₂ charges, there has not been a test to ensure connections between components are able to sustain the forces associated with the launch, flight, and recovery process. A structural vibration spectrum of the RPV is necessary to pre-certify components prior to launch. Therefore, a data logging system was developed with a high enough sampling rate to detect the appropriate vibration spectrum for the system, both during flight and simulated flight loads on a shaker table.

A successful launch requires many things to function as expected, therefore cascading damage becomes a major issue. Proper power delivery, wiring, calibration, and software are essential for data collection and fin control. Fin control is essential to ensure the vehicle can reach a desired apogee location for separation. Proper separation is required

to allow for deployment of parachutes and a slow descent. A slow descent is necessary to prevent damage to the airframe and electronic components and recover components necessary for data collection and post-flight analysis. Therefore, an in-flight health monitoring system becomes essential for documenting and determining undesirable events that arise during flight, especially when data becomes unrecoverable.

Further development of the AV Bay design and integration will allow for a higher level of confidence for success as the standard architecture establishes a benchmark design that can incorporate traceable design changes to be made from lessons learned. A standard AV Bay design can be improved and developed to be modular to allow for new hardware requirements and future improvements, flexible for electronic substitutions, and reliable to ensure proper connections for electronic components to sustain tactical launch and flight vibrations. The research at NPS will also benefit further development of similar systems at other military institutions, such as the Naval and Air Force Academy as well as West Point. Survivability of the vehicle is critical for reusability and advancement.

1. Survivability and Fault Tree Analysis

Ball [10] defines “the ability of an aircraft to avoid and withstand the man-made hostile environment as the survivability of the aircraft.” For this scenario, the hostile environment will be the harsh conditions of the desert and high-speed flight, as the ability to avoid and withstand high temperatures and a high-speed landing is vital to the RPV’s survivability. Additionally, Ball [10] defines the Fault Tree Analysis (FTA) as “a top-down approach that starts with an undesired outcome or event, then determines what event or combination of events can cause the undesired event.” Critical components and systems were identified when the FTA’s were conducted. For the NPS RPV, FTA’s were made for the most common/major issues. Three FTA’s, seen in Figure 9, Figure 10, and Figure 11, were developed to outline the separation, data collection failures and/or issues, and AV Bay structural integrity.

a. Separation

Separation is a primary failure mode and requires redundancy and reliability from the hardware. Figure 9 demonstrates the impact of incomplete or unexpected separation of the RPV, which results in cascading damage as the high-speed impacts lead to structure failure, damaged electronics, and unrecoverable data. For example, if the AV Bay structure fails due to a high-speed impact, the electronics can overheat from the damage, and the SD cards can be lost or inaccessible for post-flight data analysis. Issues in separation led to damaged data hardware, which made the data inaccessible.

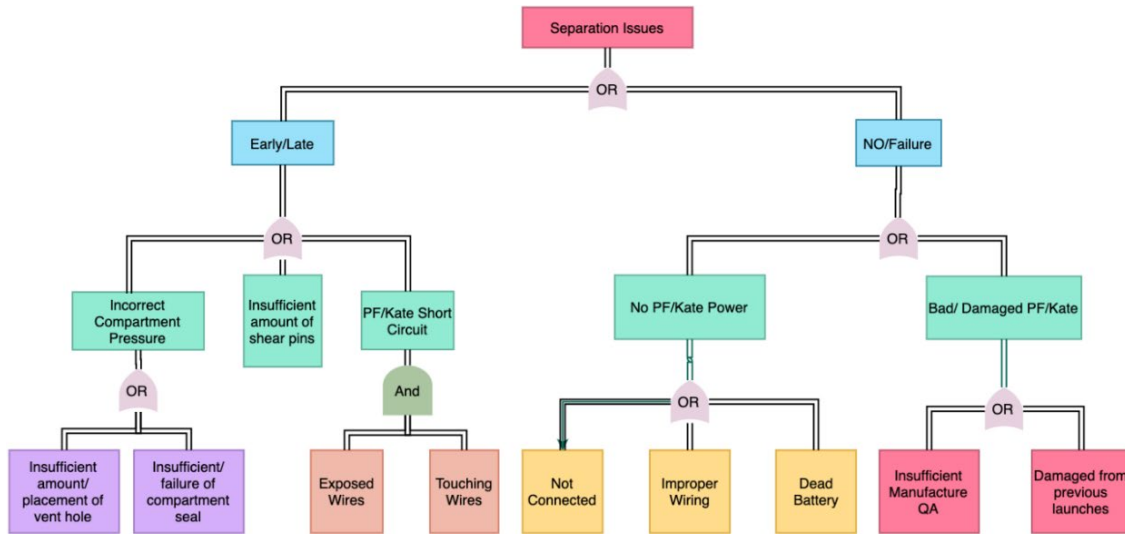


Figure 9. Separation FTA. Adapted from [11].

b. Processor/altimeter data

A loss or degradation of flight data, as seen in Figure 10, directly impacts the ability to properly conduct post-flight analysis. Data preservation is vital to collect data to determine the performance or sources of failure for a given flight during post-flight analysis. For example, an update to software can cause programming issues on the processor, where the data is not saved or stored to the SD card, and the root cause of failures cannot be derived for an unsuccessful flight. Separation issues have led to the loss of the AV Bay structure, which led to damage hardware and lost data.

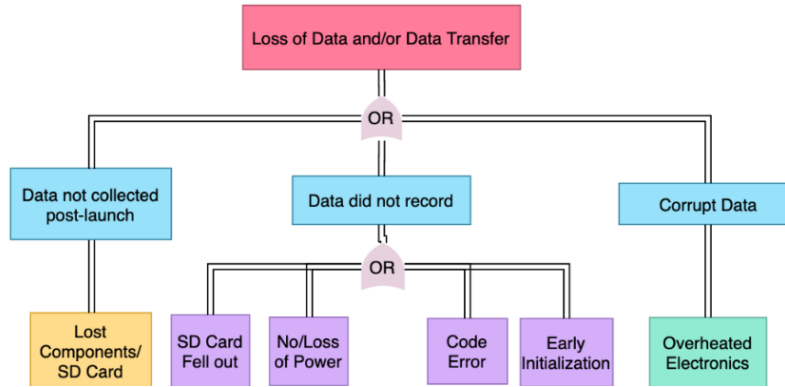


Figure 10. Data FTA. Adapted from [11].

c. *AV Bay structural integrity*

A successful booster landing is integral to preserving the structural integrity of the AV Bay. Figure 11 demonstrates cascading damage as structural deformation between points of separation will cause separation issues, main and drogue deployment failures, high-speed impact, and the loss of the AV Bay structure as well as other structures throughout the entire vehicle.

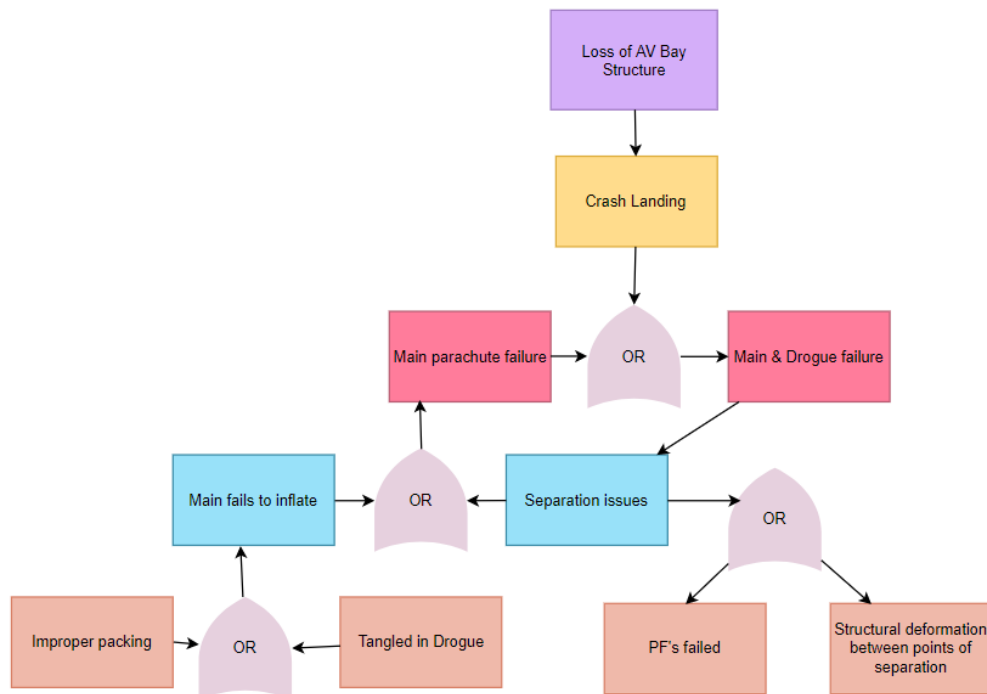


Figure 11. AV Bay structural integrity FTA. Adapted from [11].

C. OBJECTIVES

This effort emphasizes the use of commercial off the shelf (COTS) resources to ensure low costs, modularity to ensure standardization, adaptable to various mission requirements, reliable to ensure proper connections, and flight proven to sustain the necessary stresses of a launch.

The thesis has three objectives to improve the integration of the electronics, capabilities, and reliability of the RPV.

- Design a baseline avionics bay (AV Bay) with a standardized form factor, modularity, and connection harness that is configurable for multiple flight requirements.
- Incorporate a data collection system to record and transmit vital flight parameters during flight and record higher bandwidth information for post-flight analysis.
- Establish a qualification protocol to evaluate the connectivity and functionality of the AV Bay components and complete assembly under simulated loads before a flight.

THIS PAGE INTENTIONALLY LEFT BLANK

II. DESIGN PROCESS

A. NPS RPV OVERVIEW

The NPS RPV, as seen in Figure 12 and Figure 13, is a single stage rocket that is typically flown to altitudes of 1,200 to 3,000 meters with various off-the-shelf composite rocket motors. The body is made of a fiberglass nose cone, phenolic coupler, and fiberglass reinforced phenolic airframe. The system's length is nominally about 4 meters long and has an initial weight of 45 to 55 kilograms on the launch pad depending on the rocket motor used and system configuration. The RPV was reused and launched five times from November 2020 through October 2021, with each launch producing various lessons learned. The vehicle was designed to separate into two parts; the upper stage contains the nose cone and sled, and the lower stage contains the interstage coupler and booster.

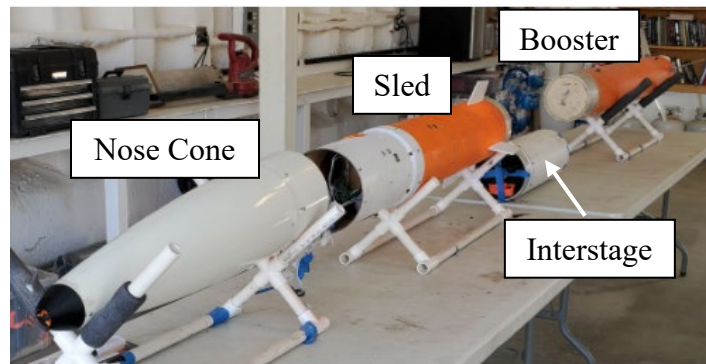


Figure 12. NPS RPV prior to launch

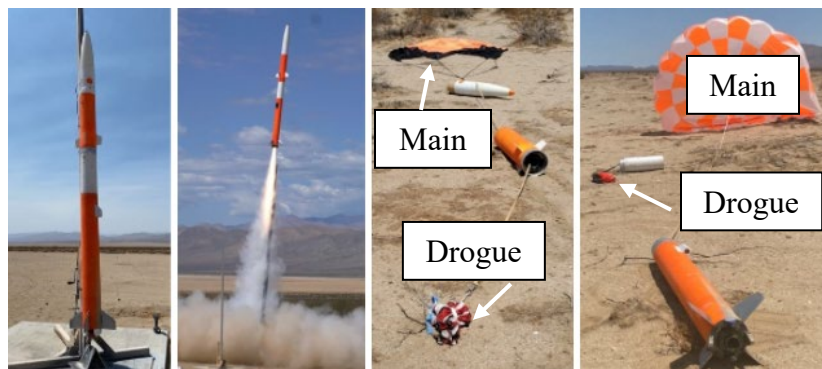


Figure 13. NPS RPV on launch pad, at launch, and recovery

a. Interstage

The interstage for the launch system, as seen in Figure 14, is the upper portion of the lower stage of the RPV and contains the recovery electronics, parachute, and an avionics bay. Figure 15 contains multiple views of the interstage components. Due to this vehicle having an aluminum coupler that held the AV Bay in the booster section, the health monitoring system was placed in the interstage to avoid interference for the telemetry system that would be caused by the aluminum section of tubing. The drogue was located in between the sled and the interstage, while the main parachute was between the interstage and the booster.



Figure 14. RPV interstage

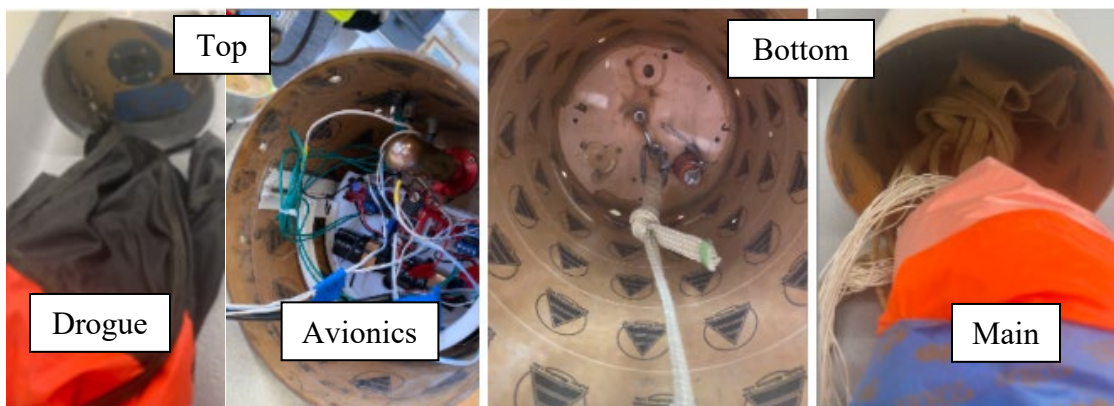


Figure 15. RPV interstage top and bottom views

(1) Health Monitoring System

The health monitoring system transmitted selected data channels to the ground (telemetry) through an Arduino Uno transmitter in the RPV down to the Arduino Uno receiver on the ground, known as the ground station. Pressure data was transmitted and demonstrated a successful operational test of the telemetry system. Although only two parameters were demonstrated, the system may be configured to transmit multiple and various parameters tailored to the needs of the mission, such as velocity, altitude, acceleration, pressure, temperature, voltage, and current. A labeled view of the critical components is shown in Figure 16 and Figure 17 and the software for both the flight vehicle and ground station is included in Appendix D. This system can be relocated to the booster section if required.

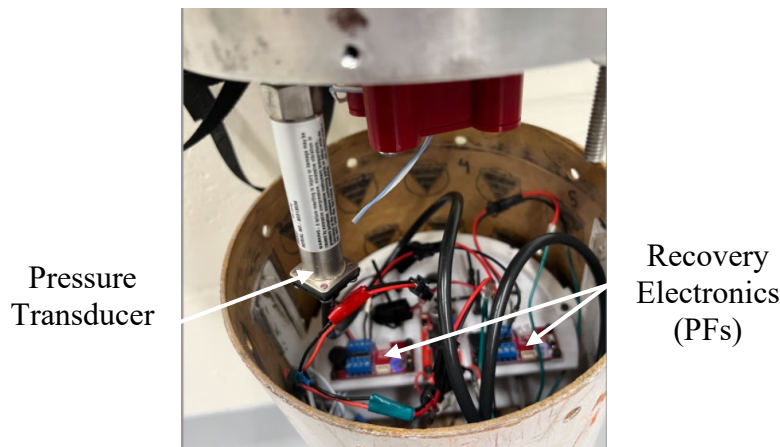


Figure 16. Health monitoring system location in the interstage

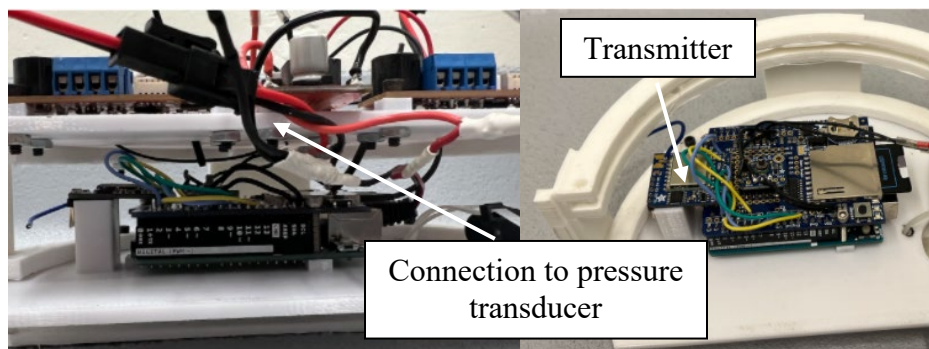


Figure 17. Telemetry and separation electronics

b. Booster

The booster, as seen in Figure 18, was the primary component of the lower stage of the RPV. In addition to containing the solid rocket motor, it contained an aluminum coupler, avionics bay, motor, servos, fins, and crushable rear shock absorbers (CRSAs) for landing loads.



Figure 18. RPV booster

(1) Avionics Bay

The Avionics (AV) Bay is an essential component to any guided system as it contains inertial sensors, an onboard processor, and power to provide guidance and control to the vehicle through control of the fins. The AV Bay, as seen in Figure 19, contains the following electronic components:

- Lithium-ion polymer (LiPo) battery: Provides power to the servos and fins
- Instrumentation battery: A portable power bank that supplies 5 volts to the RPi
- Relay module: Cuts power to the servos and fins after a specified time
- BNO055 orientation sensor: The inertial orientation sensor vital to GNC
- Arduino Due: A microprocessor used for data logging
- ADXL345 accelerometer: Determines system accelerations for data logging

- Raspberry Pi (RPi) 4B processor: A central processing unit (CPU) for GNC software which controls the fins

The AV Bay support structure was 3D printed out of either polycarbonate (PC) or tough polylactic acid (PLA) and held in place with two bolts going through the 3D printed rail and bay, as seen in Figure 20. It has the ability to incorporate other hardware and electronic components, such as those flown in the interstage due to capacity constraints from previous design decisions. The components that could be relocated to the AV Bay:

- Separation Electronics and Hardware
 - Two perfect flite (PF) altimeters, which were pre-programmed for two-event deployment
 - Four electronic matches (E-matches), which were ignited by the PFs
 - Two CO₂ cartridges, which generated pressure to separate components
 - One drogue, which was deployed after reaching apogee
 - One main parachute, which deployed after reaching a pre-set altitude during descent
- Health Monitoring System
 - Omega pressure transducer (vacuum to 1 ATM), which monitored the pressure of the separation bay between the sled and the interstage
 - DC-DC boost converter, which converted 9V to 15V for the pressure transducer
 - Adafruit pressure sensor, which monitored the compartment pressure of the interstage

- Arduino Uno Rev3 SMD with Adafruit LoRa, which was the RPV transmitter

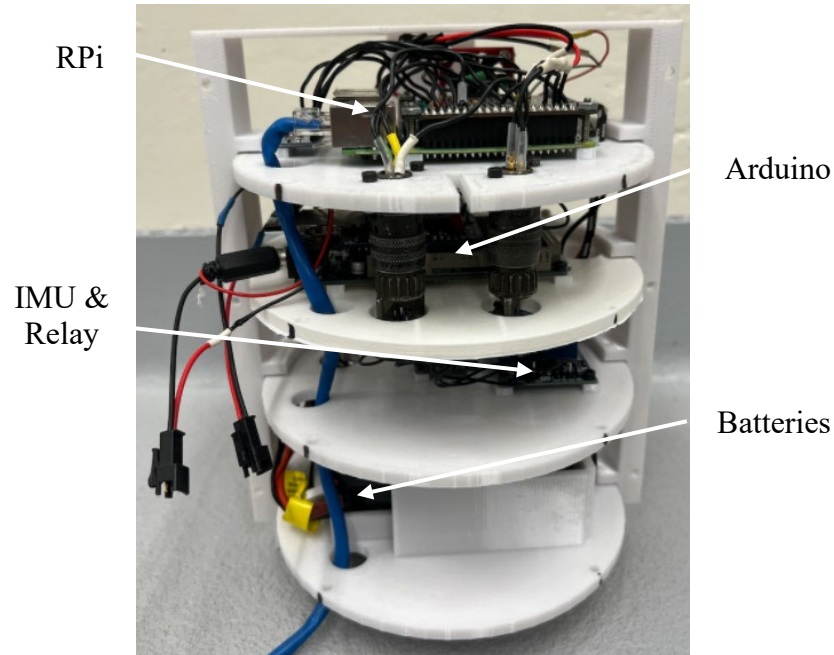


Figure 19. AV Bay components

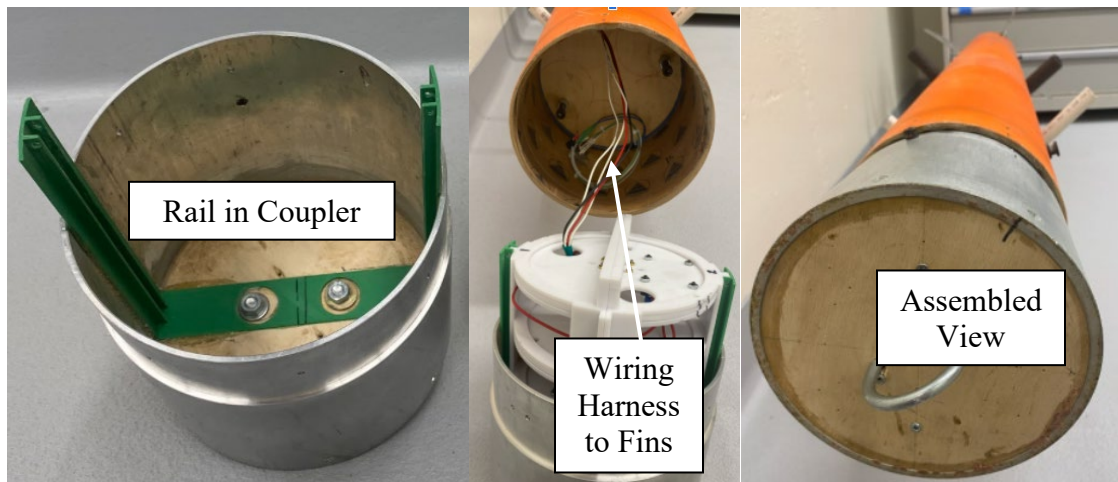


Figure 20. AV Bay location in the booster

B. AV BAY

1. Approach

The design of the AV Bay, sensor modularity, and hardware placement was inspired by the CubeSat form factor, as seen in Figure 21, but utilized a circular form vice a square one in order to maximize the usable volume for the vehicle cross section. “A CubeSat is a class of satellites that adopt a standard size and form factor, which unit is defined as ‘U’. A 1U CubeSat is a 10 cm cube with a mass of up to 2 kg” [12].

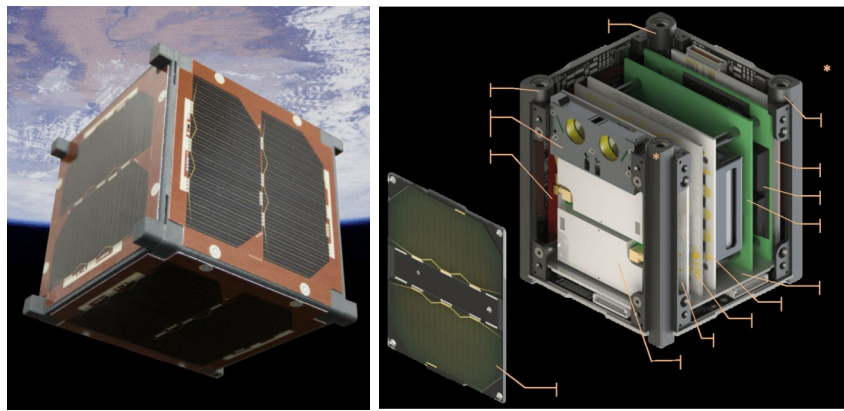
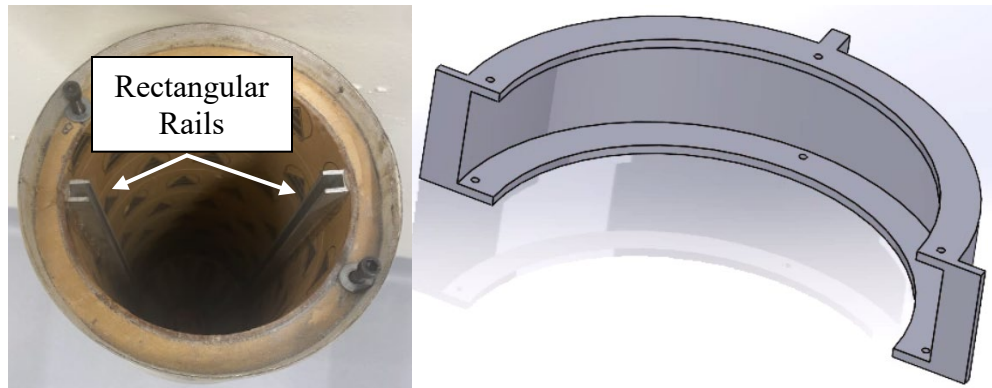


Figure 21. CubeSat examples. Sources: [12], [13].

The AV Bay alignment and securement was initially designed to utilize the aluminum guide rail concept utilized in the nose cone, as seen in the left image of Figure 22. Ensuring the proper engagement of the AV Bay into the two aluminum railings on each side proved to be difficult. It was decided that the railing, as seen in Figure 23, would be custom designed and 3D printed for proper alignment and would have a circular outer dimension to fit the vehicle's contour in contrast to the direct attachment of the rectangular aluminum railing.

The first initial concept for the AV housing, as seen in the right image of Figure 22, allowed the electronics platters to be easily accessible and the number of levels to be customized. The first design was not ideal as the upper support for half of the platter was limited and would cause more vibrations as the material flexed, similar to a cantilever beam.



The left image has the rectangular guide rails in the nose cone.

Figure 22. Initial rail and AV Bay concept

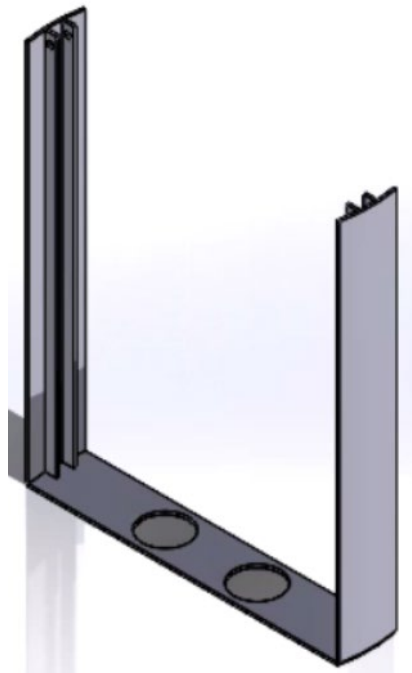


Figure 23. 3D printed rail

The connectors, as seen in Figure 24, are commonly utilized for CubeSats and were initially going to be used for most of the required connections as they are flight proven to be reliable and able to sustain flight vibrations. The connector has “a four fingered clip (that) claps the round pin tightly, ensuring reliable connection even under severe conditions. Additionally, the hexagonal slotted floating jackscrews ensure vibration and shock requirements are met, and give maximum strain relief” [14].



Figure 24. Harwin connectors. Sources: [14], [15].

The orientation of connection, as seen in Figure 25, was originally misunderstood. This required a custom board to interface with as it contains both data and power pins. These connectors were not used as acquiring a custom board would require a long lead time and increased cost but could still be utilized in future design modifications.

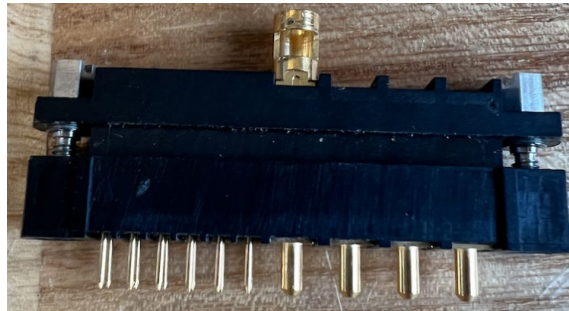


Figure 25. Connection of Harwin connectors

2. Avionics Bay Version 1

The AV Bay housing consisted of two shells and platters, as seen in Figure 26, that were 3D printed with tough PLA using Ultimaker 3 extended 3D printer. Due to the failure of the single wall phenolic coupler that housed the AV Bay on the 21NOV20 launch, the coupler, as seen in Figure 20, was replaced with a machined aluminum coupler. Although the aluminum coupler proved to be much stronger than the phenolic one, one major disadvantage of this is the inability to place a GPS system or a transmitter/health monitoring system in the AV Bay due to electromagnetic shielding.

The housing and 15.24 cm (6") diameter instrument platters had a 3.175 mm (1/8") thickness. A bolt was placed near the RPi SD card to ensure the SD card did not fall out midflight, which was experienced on the 21NOV20 launch. Version 1 utilized a RPi for fin control which replaced the previous Arduino and implemented a relay module to cut power to the fins prior to landing to prevent servo/fin damage. The housing was designed to fit into the rail without any tolerance, which was an issue as the rail was epoxied to the aluminum coupler and any slight misalignment or variations in the layer of epoxy prevented the housing from sliding into the rail. Future dimensional tolerances between different 3D parts were taken into account when designing future components.

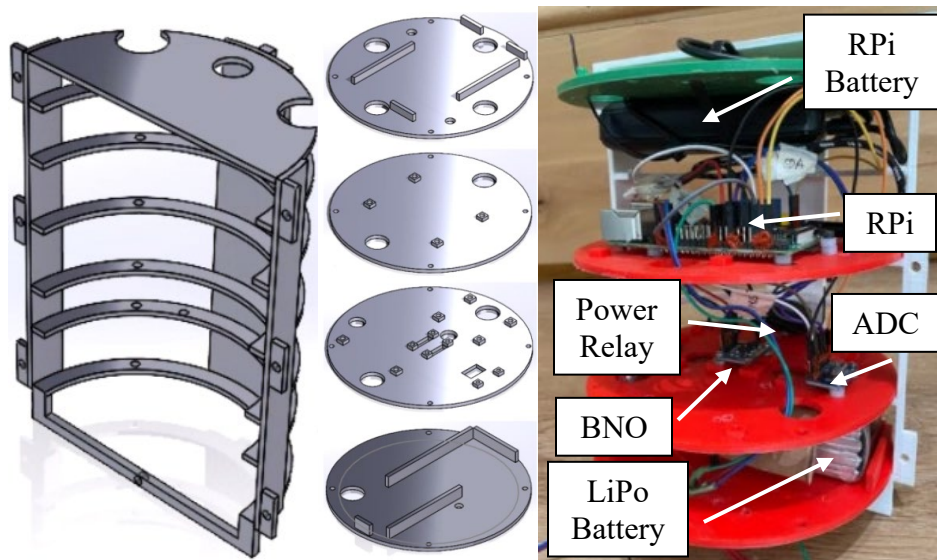


Figure 26. AV Bay flown on 17APR21 launch

Since there was not enough time prior to launch to order and receive replacement connectors before the next launch date, breadboard jumper wires were utilized and reinforced with room-temperature-vulcanizing (RTV) silicone and electrical tape. The RPi connections to servos had to be soldered as the connection required multiple breadboard wires, which were not felt to be reliable, to reach the connector at the bottom. The design was modular and flexible as the various platters could be changed out for various mission requirements. The shell and removable platters concepts allowed for easily accessible electronics.

3. Avionics Bay Version 2

Version 2, as seen in Figure 27, was going to be launched during the hottest temperatures of the year, 41.7°C (107°F), and since tough PLA is known to deform under high temperatures, it was decided that it would be printed in polycarbonate on a Stratasys Fortus 450mc 3D printer. Version 1 required the platters to be bolted on the shell in order to connect the electronics between platters. A shelf concept was implemented in version 2 to improve layout flexibility and allow the platters to be placed on the shell without bolts using a clamped approach. The inertial measurement unit (IMU), BNO055, was placed lower in the stack to try to achieve more stability and be less susceptible to flex in the platter.

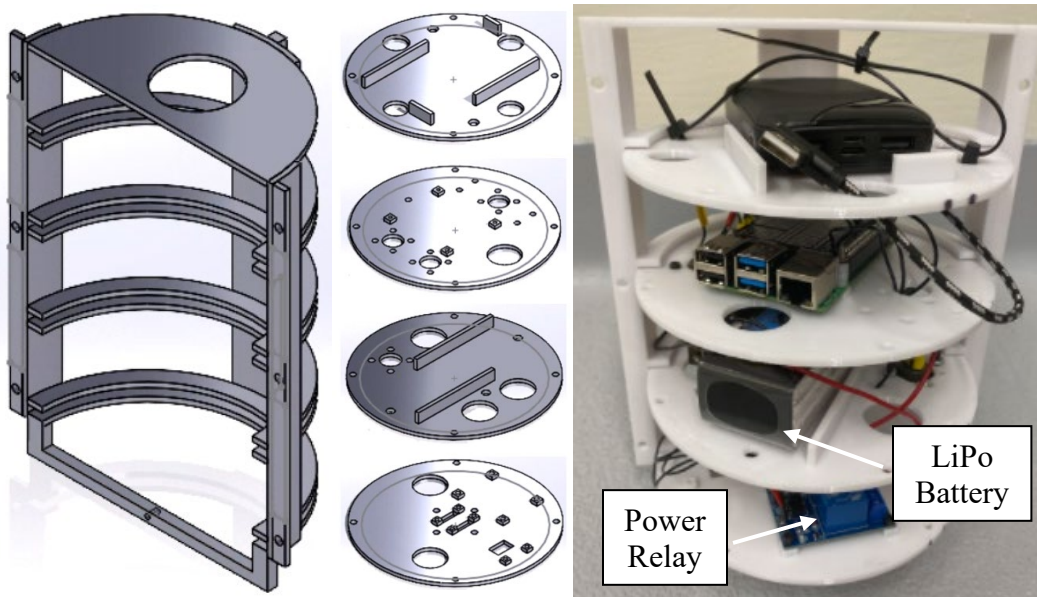


Figure 27. AV Bay flown on 17JUL21 and 21AUG21 launch

A RPi general purpose input/output (GPIO) expansion board with mechanical terminals, as seen in Figure 28, was initially going to be used for connections, but the board was suspected to be faulty as that RPi would short when connecting the board. Since there was not enough time prior to the next launch date to order and receive a replacement board, wires were soldered to a prototyping board hardware attached on top (HAT), as seen in Figure 28.

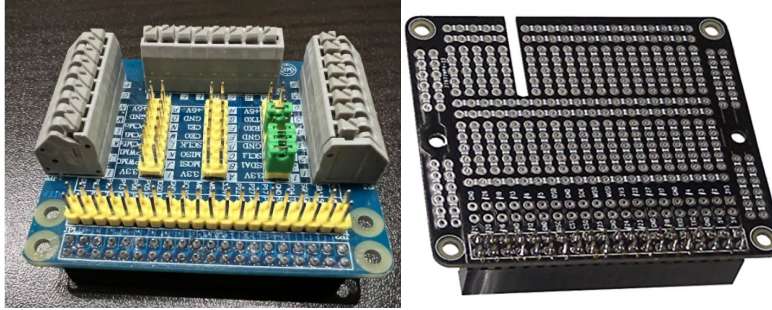


Figure 28. RPi GPIO expansion board and RPi HAT. Source: [16].

Since the RPi has 40 pins, a 37-pin military specific (mil-spec) connector was going to be utilized for connections. This connector was not utilized as it was large, heavy, and did not allow the various components to be easily changed as initially thought. Instead, three six-pin mil-spec connectors, as seen in Figure 29, were used to connect to the BNO, relay, and servos to the RPi HAT. Three connectors improved reliability as each component had their own designated connector. The box mount for the female connector allowed for reliability as it was securely mounted to the platter in the same orientation for the various connectors.



Part numbers are MS3112E10-6S and MS3116F10-6P.

Figure 29. Female and male six-pin mil-spec connector. Sources: [17], [18].

On the 17JUL21 launch, the RPi had difficulty connecting to the configuration PC through Wi-Fi, which prepared the system for launch, possibly due to the high component temperatures. This step is necessary to calibrate the AV Bay on the launch pad for proper orientation. In order to ensure proper connectivity and improve reliability, a local area network (LAN) connection, as seen in Figure 30, was mounted to the outside of the RPV for the 21AUG21 launch.

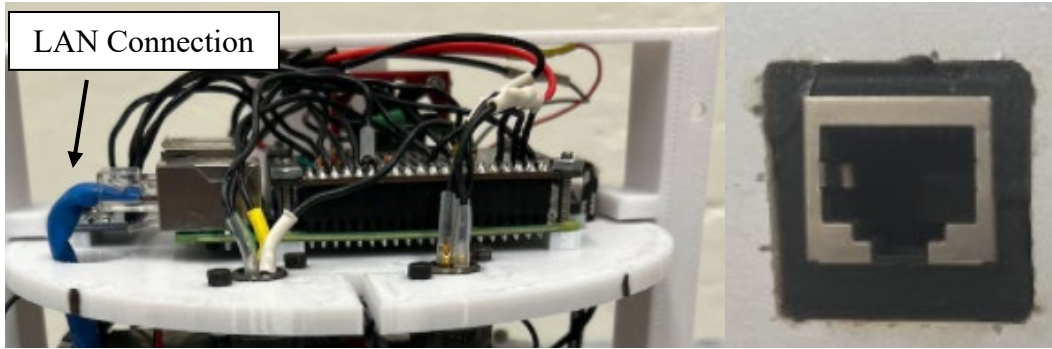


Figure 30. RPi LAN connection

The 21AUG21 flight experienced problems with the drogue/main parachute deployment process. Post flight video footage, as seen in Figure 31, revealed that the main parachute shock cord was not long enough and when the main deployed it got tangled with the drogue parachute and was unable to fully inflate.

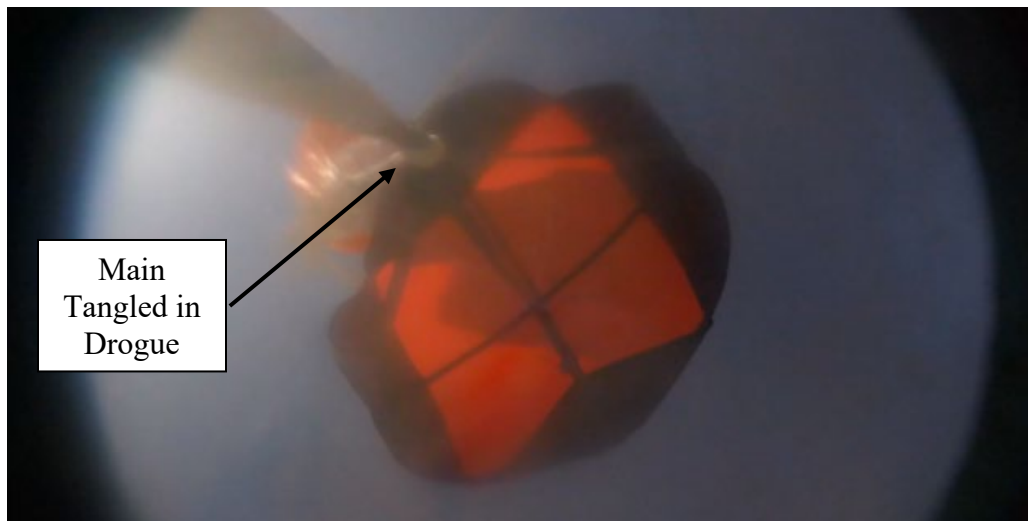


Figure 31. Still shot of forward GoPro footage 21AUG21 launch

The result of the incomplete inflation of the parachute was the booster impacted and experienced greater than 16G's, which left the AV Bay in pieces, as seen in Figure 32 and Figure 33. Reconstructing the platters revealed localized areas of weaknesses that are susceptible to failure. The mil-spec connectors proved to be reliable, as seen in the middle photo of Figure 32, as the connection remained despite the impact.

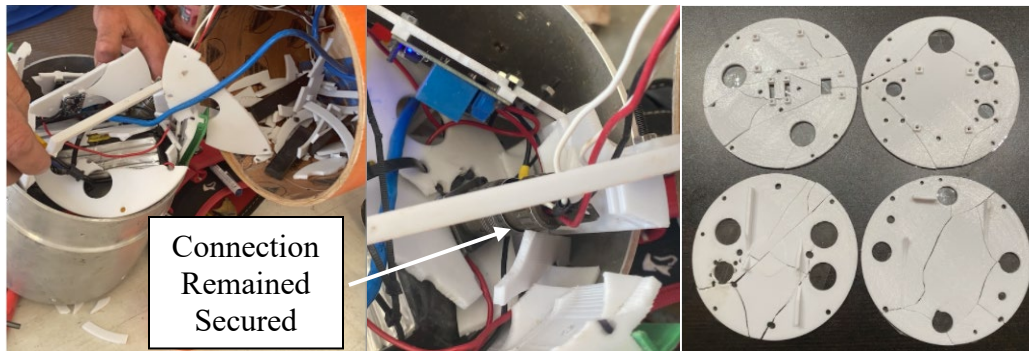


Figure 32. AV Bay pieces after impact and reconstructed platters

Although the electronics appeared intact, as seen in Figure 33, those electronics were not used for the next flight. The connectors were flexible and reused as the connections were easily unsoldered and soldered for the next flight.

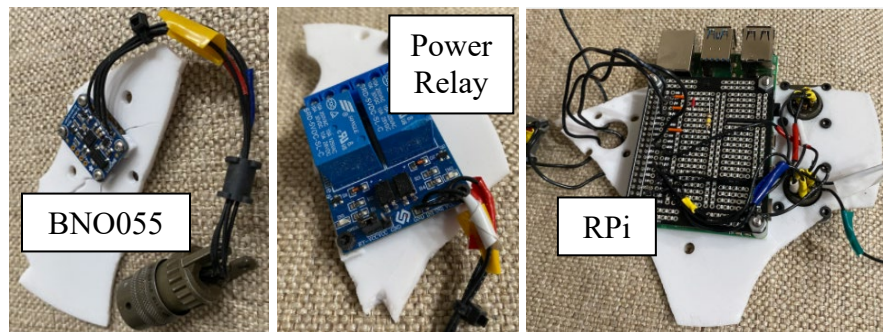


Figure 33. Electronic components after impact

4. Avionics Bay Version 3

Due to the brittle nature of the polycarbonate and the failure experienced on the previous launch, the thickness of the platter and shell shelves thickness was increased to 5 mm (0.2”), as seen in Figure 34. The bay was once again printed with PC. The reconstructed platters from the hard landing, as seen in Figure 32, demonstrated weak areas and all unnecessary holes were removed for this version. Holes were placed to allow connections, such as the LAN, servo, and RPi connections, to be accessed through a seamless vertical path through the different platters to ensure reliability.

An analog to digital converter (ADC) had to be added and proved the bay to be modular as it was seamlessly integrated. Since the last launch had a hard impact, the LiPo and RPi batteries were placed at the bottom to attempt to minimize damage in the event of a hard impact. The LiPo and RPi batteries were placed on one platter, seen in Figure 36, to allow room for the addition of the Arduino data logger.

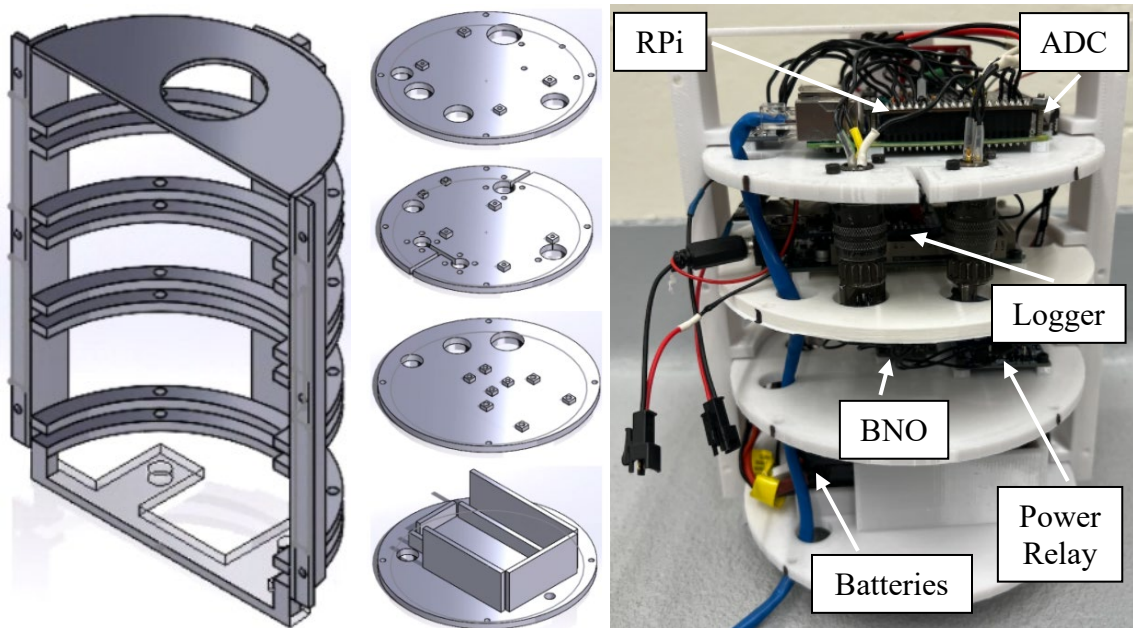


Figure 34. AV Bay flown on 22OCT21 launch

In version 2, the BNO and relay connectors were placed on the RPi platter, while the servo connector was placed on LiPo battery platter, which did not allow the platters to be completely separated. In this version, all female connectors were mounted to the RPi platter, as seen in Figure 35, to improve flexibility and ensure each platter was able to separate from another. Additionally, cutouts were put in place to improve modularity, as seen in Figure 37, to allow the female connectors to be removed from the platter.

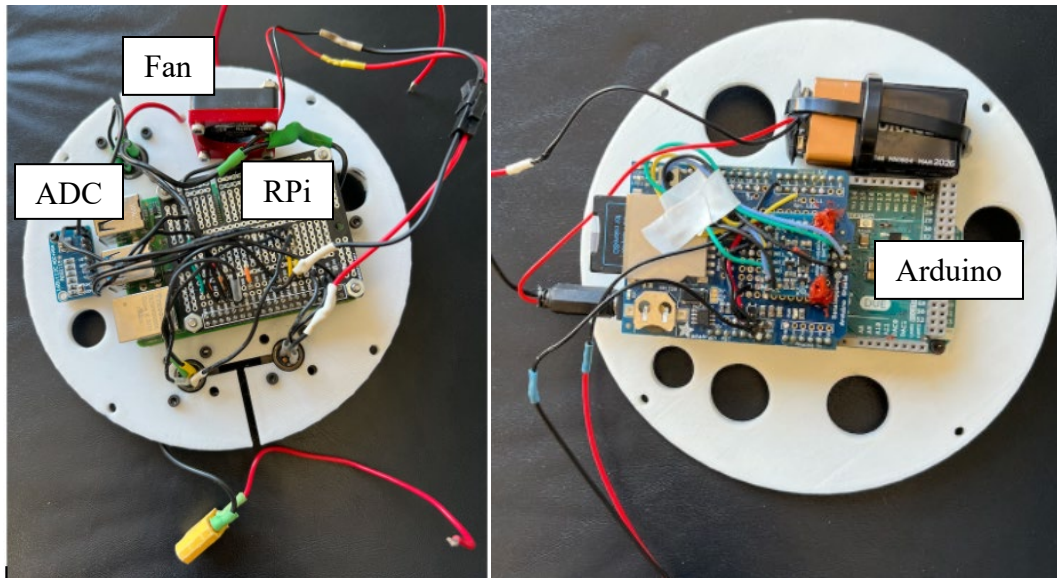


Figure 35. Top view of RPi and Arduino platters with electronics mounted

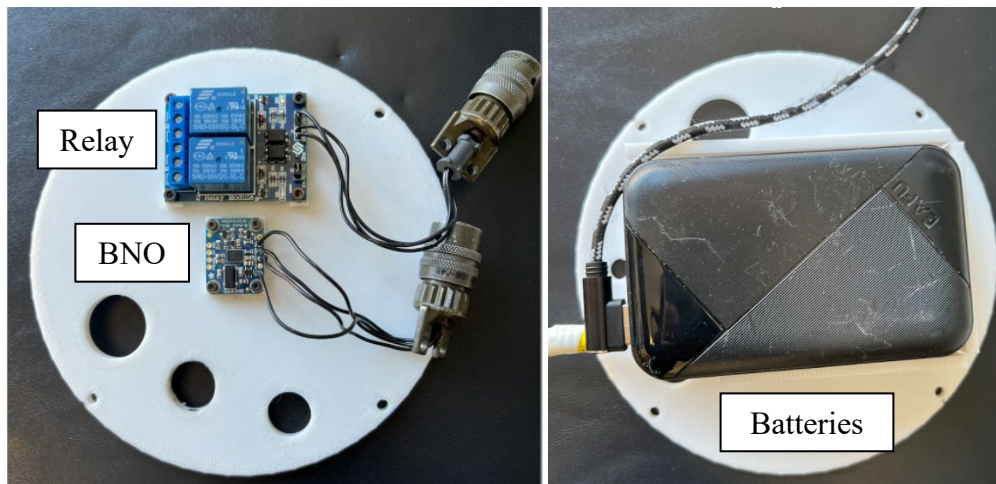


Figure 36. Top view of the two bottom platters with electronics mounted

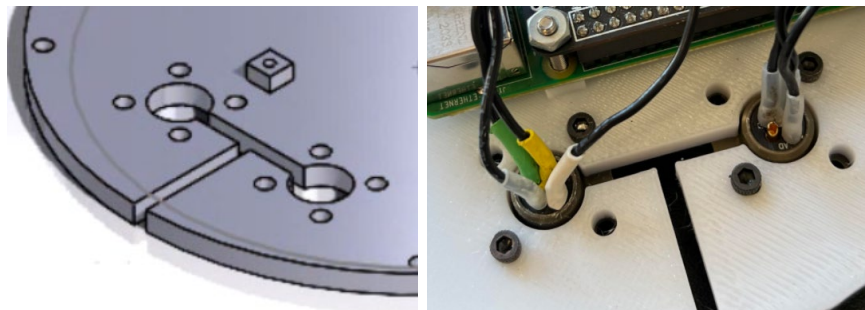


Figure 37. Wire cutouts

Due to the possible harsh desert temperatures as experienced during the 21JUL21 launch, the thermal management of the electronics was critical to ensure proper operation. This version was improved by the implementations of heat sinks and a fan to the RPi, as seen in Figure 38.

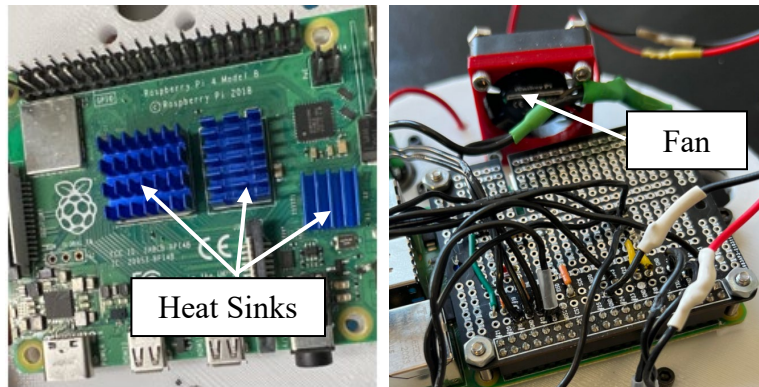


Figure 38. RPi heat sink and fan

C. TELEMETRY/ HEALTH MONITORING SYSTEM

During flight testing, unexpected events can and have occurred for the RPV, which have led to substantial damage. The use of a health monitoring system capable of transmission to the ground can help explain anomalies during flight. The initial integration of the Reyax Rylr896 Long Range (LoRa), shown in Figure 39, was attempted with an Arduino Due, but due to the 3.3V logic, which differs from the Arduino's 5V signal as well as a lack of community support for the product, the module was not implemented.



Figure 39. Reyax Rylr898 LoRa

An Adafruit RFM95W LoRa Radio Transceiver was used with an Arduino Uno, as seen in Figure 40, for both the ground station and the RPV transmitter. The Arduino Uno has 14 digital I/O pins and six analog inputs, which allowed for multiple and various data values to be monitored and selectively transmitted. The ground station was therefore able to receive the pressures values from the interstage bay and the separation bay between the booster and the interstage. Ground tests proved that the system was operational at a range of at least 500 meters with line of sight.

Although the health monitoring software and data logger displayed both pressures in the local onboard serial monitor, the ground station serial monitor only received one value intermittently. It was suspected that the buffer caused data to transmit intermittently. The code was then altered to include a D for digital or A for analog to make post processing easier, but the error still existed. The final solution to the issue was to print both sensor values on one line of output.

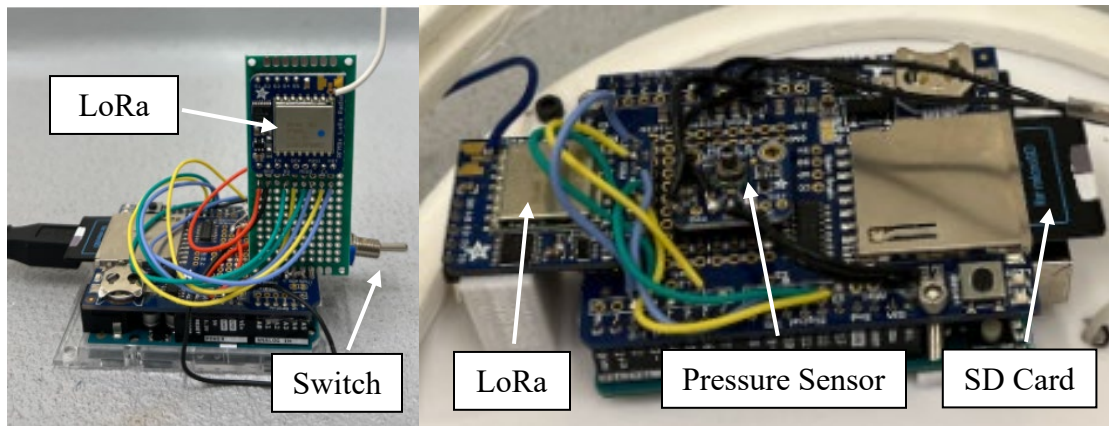


Figure 40. Ground station receiver and RPV transmitter

D. DATA LOGGER

In order to certify and test the AV Bay prior to flight, a vibration spectrum can be constructed from flight data collected from the data logger. The initial desired sampling rate for the data logger was determined by examining a similar solid rocket's power spectrum as no previous data sets were available for the NPS RPV.

“The Nike type M88, solid-propellant rocket and a boilerplate payload has a nominal thrust of 213,000 N and a total weight of the vehicle of 705.2 kg or over 1500 lbs” [19]. The M3400 motor experiences a 3983N nominal thrust level, which is 53 times less and the RPV weighed around 45 kilograms, which is 15 times less. Schoenster [19] shows the Nike “peaks in the power spectrum at frequencies 600 Hz, 1000 Hz, 1300 Hz, and 1500 Hz can be seen in the data for both time periods.”

Additionally, “the bandwidth, or maximum frequency is a function of the sampling rate as the sampling rate must be greater than two times the bandwidth of any sensor” [20]. Vibrations were seen in the 600 to 1500 Hz range, which means a minimum sampling frequency of 3000 Hz is desired. The Nike is the closest comparable vehicle with documented frequencies. Therefore, a sampling rate of 3000 Hz was chosen to be the initial sampling rate for the NPS RPV data logger.

Originally the BNO055 sensor was going to be used for the data logger, but after examining the data sheets, the sensor cannot achieve an output rate above 100 Hz due to a hardware based Kalman filter. Although 1000 Hz can be achieved by reading the raw data and setting it to a ‘non-fusion’ mode, seen in data sheet [21] tables 3–5 and 3–8, a higher sampling rate of 3000 Hz would be preferred.

An ADXL345 was used on a data logging shield attached to an Arduino Uno for version one. The ADXL345 Adafruit library [22] defaults to Inter-Integrated Circuit (I2C) communications, which is a much slower serial protocol, but can be easily used to integrate multiple devices. The SD card requires Serial Peripheral Interface (SPI) communications protocol, which supports multiple peripherals, but requires a more in-depth procedure. When the ADXL345 was in SPI mode, it could only read the accelerometer data or log data, but could not do both simultaneously. The I2C was used for version one, but a sampling rate of only 600 Hz was achieved utilizing this method.

The low latency data logger ADXL345 code, which was adapted from the Adafruit SdFat fork library [23], was used to measure acceleration in units of gravity, G. The base G variable was assigned the sensor's output value that was equivalent to the force of gravity, while the system was stationary on a workbench. The Arduino was programmed to trigger when it experienced 4.9 m/s^2 above a static value of 9.8 m/s^2 .

For version two, the Arduino Due processor was used, as seen in Figure 41, and the library was changed from default (SD.h) to the Adafruit Fork library (SdFat.h). Changing libraries allowed the system to operate all on the SPI protocol, which allowed it to quickly alternate between writing and reading data.

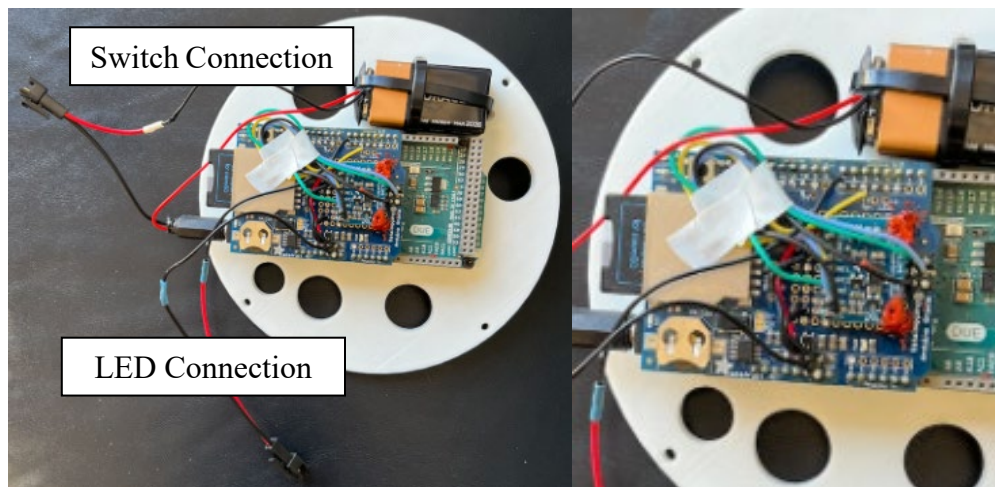


Figure 41. Data logger

Additionally, helper functions allowed the system to write binary (bin) data to a file, which was converted post-logging to a common separated values (CSV) file with a separate code. There was an issue with the creation of a temporary file that did not allow the data to be opened initially, but when manually deleted, there was access to the file. When the software was loaded, the Arduino integrated development environment (IDE) serial monitor gave options to list the files, select the desired file, and convert the desired file to a CSV file. The serial monitor must be set to NL & CR and the proper baud rate. The default baud rate setting is 9600, but can be changed and verified in the Serial.begin function.

The Arduino was triggered when it experienced 1.5 times base G and logged at a 3200 Hz sampling rate for 60 seconds. Empirical data was collected and verified on the ground. The system was attached to an LED light that would blink slowly to indicate that it was powered on, but not logging and would blink fast if there was an error. It would show a solid light when it was logging and then a fast blink to indicate it was still powered on, but had completed logging.

THIS PAGE INTENTIONALLY LEFT BLANK

III. RESULTS AND DISCUSSION

A. AV BAY

The five launches each produced valuable results to learn from, which allowed for improvements to the AV Bay with each subsequent version. Table 1 shows a test log for the five launches to include motor type, apogee in meters above ground level (AGL), maximum temperature, and issues for the AV Bay. As shown in the table, deployment and inflation of the main parachute was critical for the AV Bay structural survival, which was dependent on proper separation between stages and components.

Table 1. Flight test log

AV Bay Version	Launch Date	Motor type	Apogee (m AGL)	Max Temp (K)	Issues	Survived?
N/A	21NOV21	M3400	~1400	300	SD card not secured	Yes
1	17APR21	M3400	~1200	299	N/A	Yes
2	17JUL21	N2900	~1400	315	N/A	Yes
	21AUG21	M3400	~1600	310	Main did not fully inflate, as seen in Figure 31	No
3	22OCT21	N3301	~1500	300	Separation failure. Main & drogue failed to deploy	No

Although the booster does not have a system to determine impact or touchdown (TD) velocity, the nose cone has one and can be utilized as an approximate condition for the booster for similar recovery conditions. The Kate 3.0 transmitter, as seen in Figure 42, was a GPS Tracking System mounted in the nose cone that provided real time in-flight data such as altitude, velocity, distance, bearing, descent rate, ground speed, direction, drift rate, motor burn, flight time, and landing speed, which was seen and heard on the receiver.

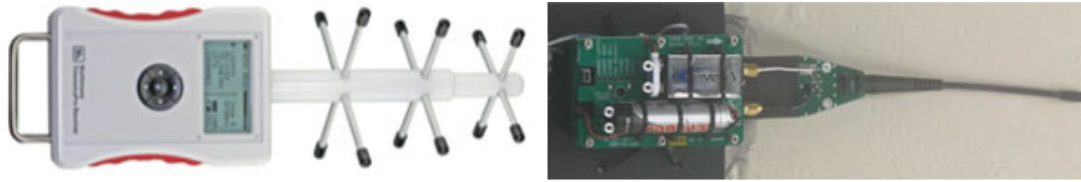


Figure 42. Kate receiver (left) & transmitter (right). Source: [24].

Table 2 shows the descent rate (DR) on drogue only, main parachute, and TD from the Kate altimeter and tracking system, along with issues that occurred during the four launches. Once again, proper separation was needed to deploy the parachutes and inflation was required to descend at a slow rate. The 21AUG21 launch had no issues and showed expected decent rates.

Table 2. Kate flight data

Launch Date	Motor Type	Apogee (m AGL)	Drogue DR (m/s)	Main DR (m/s)	TD DR (m/s)	Issues
21NOV21	M3400	~1400	4.27	3.96	4.57	None
17APR21	M3400	~1200	5.49	4.27	5.49	Early separation between nose cone & sled, nose cone unrecoverable
17JUL21	N2900	~1400	33.53	37.49	38.40	Upper stage drogue failed to inflate, as seen in Figure 13
21AUG21	M3400	~ 1600	13.72	3.66	3.35	None
22OCT21	N3301	~1500	71.32	52.43	47.55	Separation failure, Main & Drogue failed to deploy

Due to cascading damage on the vehicle during the 22OCT21 launch, there were issues of separation and the main and drogue parachutes did not deploy correctly. The Kate recovery system reported the nose cone impacted at 48 m/s (106 mph). Although the impact velocity of the booster is unknown, it is assumed the booster impacted at similar speeds of the nose cone. Figure 43 shows the effects of the impact. Although the batteries caught on fire and burnt the electronics, as seen in Figure 44 and Figure 45, two of the three connections remained intact.



Figure 43. Recovery of the booster with a zoomed in view

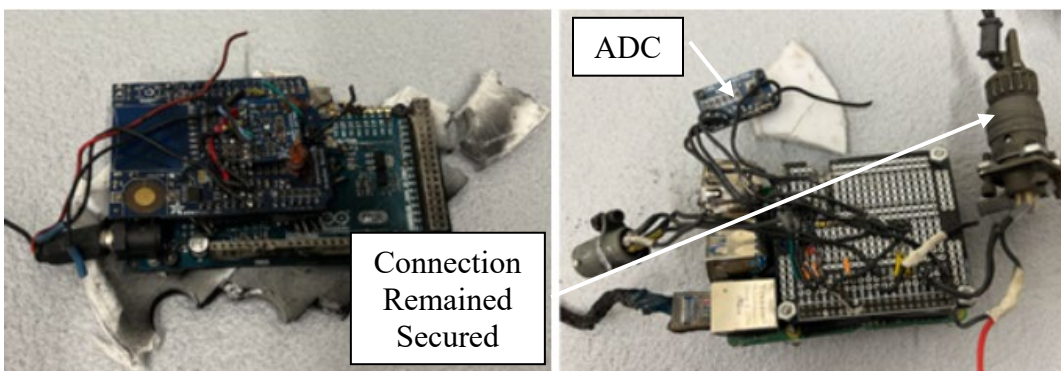


Figure 44. Arduino and RPi with ADC after impact

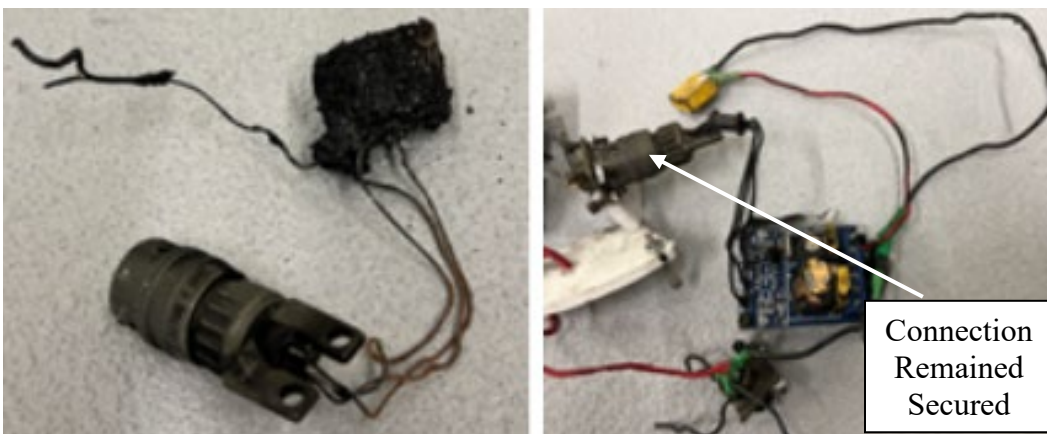


Figure 45. BNO055 and relay after impact

Examination of damages sustained, as seen in Figure 46, from impact allowed the inference of causes for unexpected flight performance. Figure 47 demonstrates damages post flight and compares the components pre-flight. Although the vehicle did experience multiple failures and sustained damage, the telemetry system implemented on this flight was able to provide confirmation of the deployment charges.

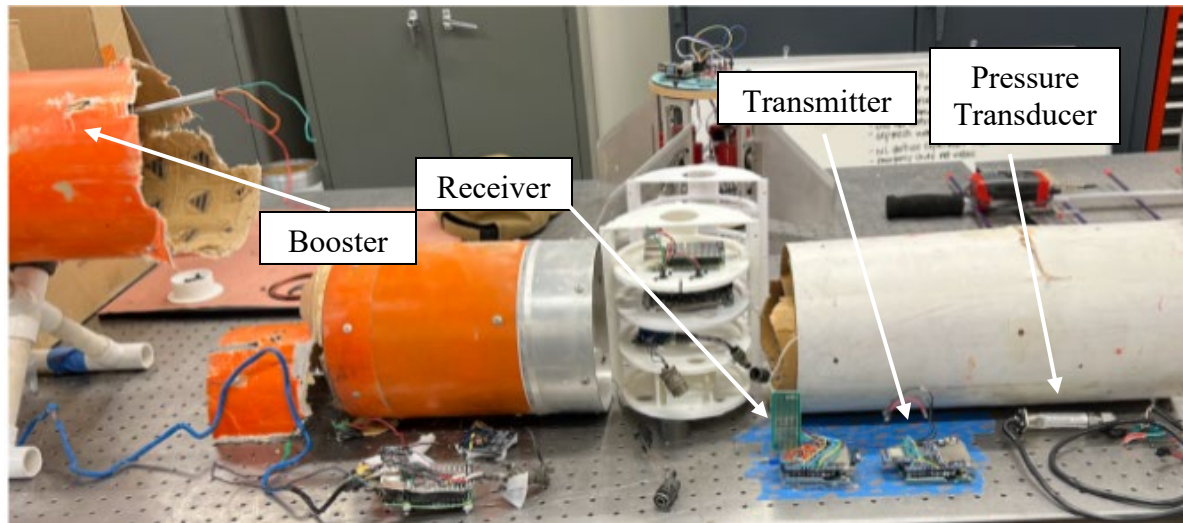


Figure 46. Damages sustained by the booster and interstage

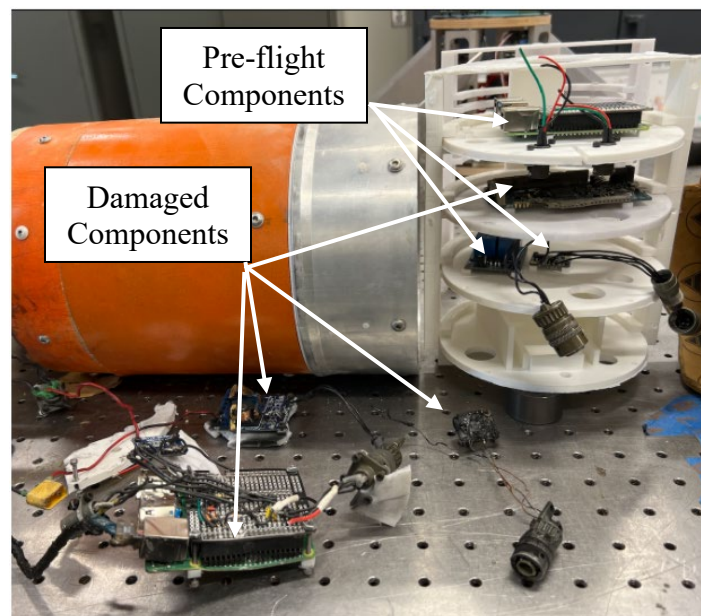


Figure 47. AV bay components pre-flight and post impact

B. DATA LOGGER

The data logger was tested during two launches. During the 21AUG21 launch, version one included a key switch, which passed power to the Arduino and illuminated a light, which was visible through vent holes. Due to the microprocessor design of the Arduino, a software program would immediately begin to run once the unit was powered. The logger was programmed to be triggered when it experienced 1.5G absolute acceleration, which was proven to be too low. Post flight analysis showed that the data collect was unusable since it was discovered that while trying to arm the PF's while the RPV was vertical on the launch pad, the batteries for the PFs were not connected. The RPV had to be lowered to fix the issue and it was concluded that the logger was accidentally triggered during the process.

A LED indication light, as seen in Figure 48, was then installed through the fuselage and became essential to indicate the status of the logging system, which was included in version two. Although the SD card from the 22OCT21 launch was inaccessible due to damage, post-flight analysis visually demonstrated the logger through the LED light as it was blinking on the launch pad and turned solid after launch.

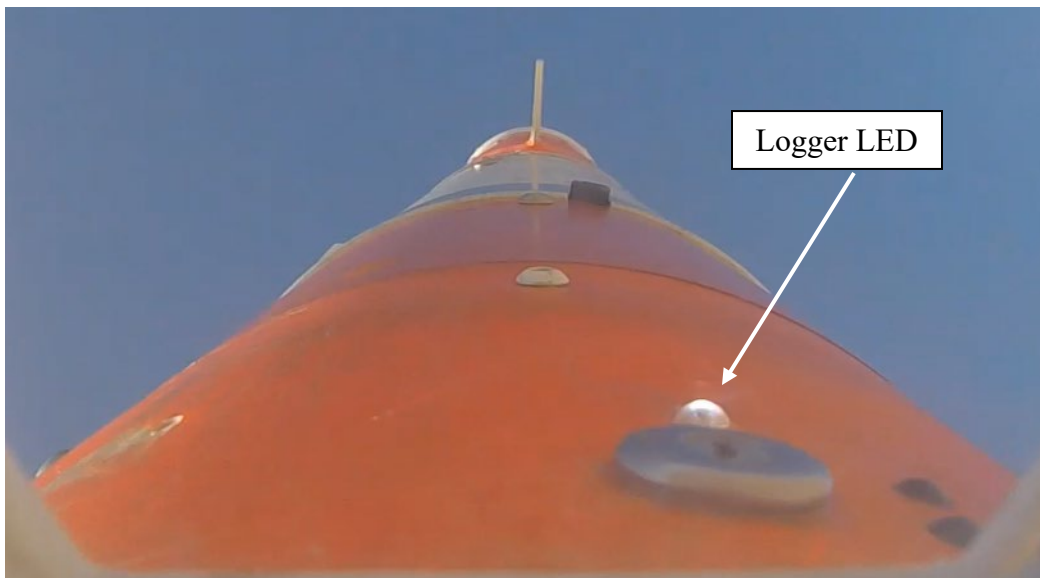


Figure 48. Still shot of forward GoPro footage 22OCT21 launch

C. TELEMETRY/ HEALTH MONITORING SYSTEM

The telemetry system was demonstrated during the last launch on 22OCT21 and indicated the RPV failed to properly separate and sustained damage both during flight and the hard landing, as seen in Figure 49 and Figure 50. It was originally suspected that the PFs failed to ignite the e-match charges, but it was seen in the telemetry data that there was an increase in pressure in the separation bay to prove this theory incorrect. The interstage attempted to separate from the sled and it was suspected that due to mechanical distortion because of aerodynamic loads on the vehicle, the pieces were unable to do so.

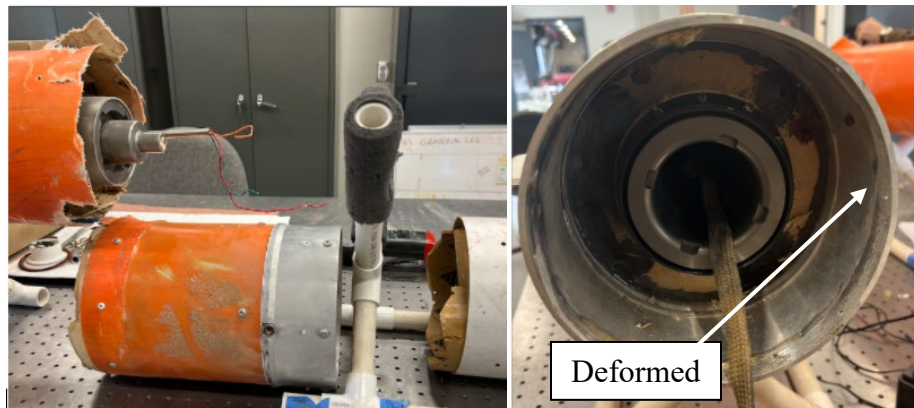


Figure 49. Damages between booster and interstage (left) and sled coupler (right)

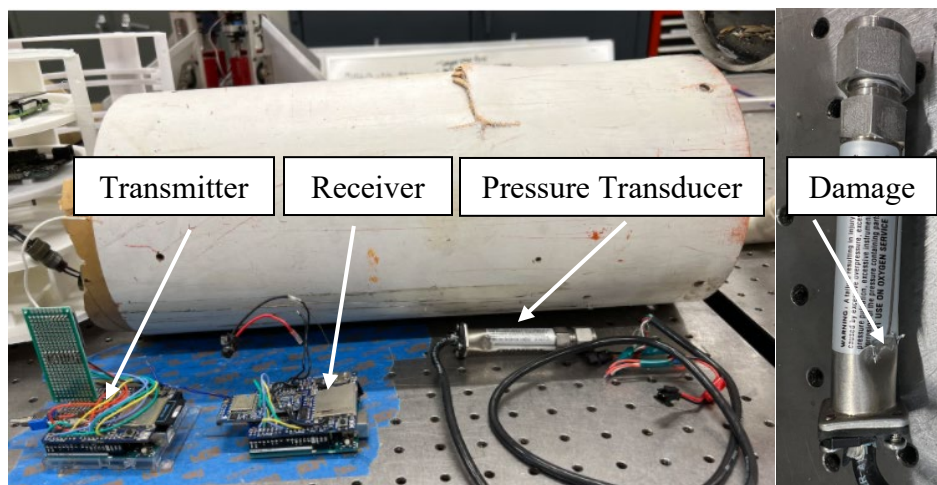


Figure 50. Telemetry components post impact and damaged pressure transducer

Table 3 shows comparable launch event times between the telemetry system in the interstage and Kate in the nose cone for the 22OCT21 launch.

Table 3. LoRa radio vs. Kate

Event	LoRa time (PDT)	Kate time (PDT)
Liftoff	13:04:21.164	13:04:22.0
Apogee	13:04:33.176	13:04:32.4
TD	13:04:50.644	13:04:52.0

The data logging system sensed the pressure of the interstage bay as well as the pressure of the bay in between the interstage and the sled, where separation occurred. This data was transmitted from the RPV down to the ground station at a baud rate of 115200. The LoRa radio initialized at 12:51:444 and frequency was set to 915.00 with initial values of 940 hPa (13.63 psi) in the interstage bay. Assuming the pressure transducer in the interstage was accurate, the digital number reported by the pressure sensor in the bay between the sled and the interstage can be converted into Pascals or pounds per square inch (psi).

The Arduino Uno has a 10-bit analog to digital converter, which allowed pressure voltage to be read into a digital value. Using this relationship, the pressure in the separation bay was converted to psi and the initial value was determined to be 923.90 hPa (13.4 psia), which is close to the 13.63 as previously determined. At apogee, as shown in Table 4, the interstage bay experienced 932 hPa (13.52 psi) in the interstage bay and 1049.38 hPa (15.22 psi) in the separation bay.

Table 4. Pressure data at separation

Time	Bay Pressure (hPa)	Separation Pressure (hPa)
13:04:32.680	840	993
13:04:32.819	840	989
13:04:33.176	932	1047
13:04:33.321	865	932
13:04:33.431	844	916
13:04:33.571	842	920

At apogee, the system experienced a pressure difference of 115 hPa, which resulted in an axial separation force of approximately 306 N (~69 lbs-f) as the airframe had a diameter of 18.415 cm (7.25"). This pressure was not able to exceed the pressure necessary for separation as six shear pins were placed and each shear pin required approximately 112 N (25 lbs-f) of axial shear force. This was likely due to excessive vent holes intended to reduce the pressure in the system during ascent.

D. SHAKER TABLE

In order pre-certify systems, hardware, and connections prior to a launch, a protocol must be developed. An air-cooled Data Physics 1216VH shaker table, as seen in Figure 51, was utilized to conduct a random vibrations test. Since there was no payload user's guide for the NPS RPV, the NASA General Environmental Verification Standard (GEVS), GSFC-STD-7000B [25], acceleration spectrum density (ASD), seen in Table 5, was used to perform a closed-loop vibration control of the shaker.



Figure 51. NPS shaker table and data sheet shaker table. Source: [26].

Table 5. Random vibrations test specifications. Source:[27].

For All Axes 10.0 g _{rms} , 1 minute	
Frequency (Hz)	ASD (g ² /Hz)
20	0.0130
50	0.0800
80	0.0800
100	0.0800
150	0.0800
500	0.0800
700	0.0800
800	0.0800
2000	0.0130

An integrated circuit piezoelectric (ICP) accelerometer, as seen in Figure 52, was mounted to the same platter as the data logger. Functional testing of the data logger was conducted prior to mounting. The AV Bay housing with the data logger, as seen in Figure 53, weighed 579 grams and was mounted directly to the table as the handle included a mounting fixture. The battery was placed on the static part of the shaker table due to the lack of documentation of a G-limit or a UN 38.3 certification/tests for the battery as well as the test exceeding 5G's.

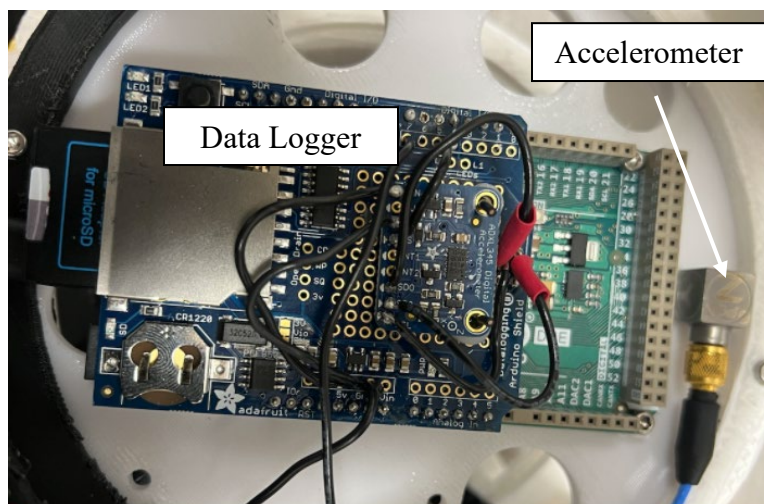


Figure 52. Accelerometer mounted to a platter

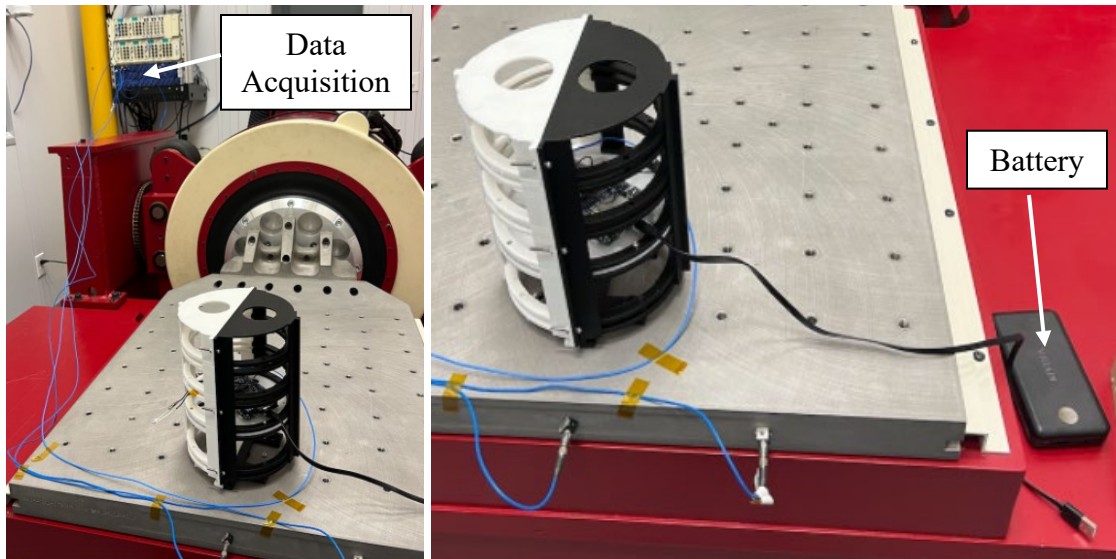
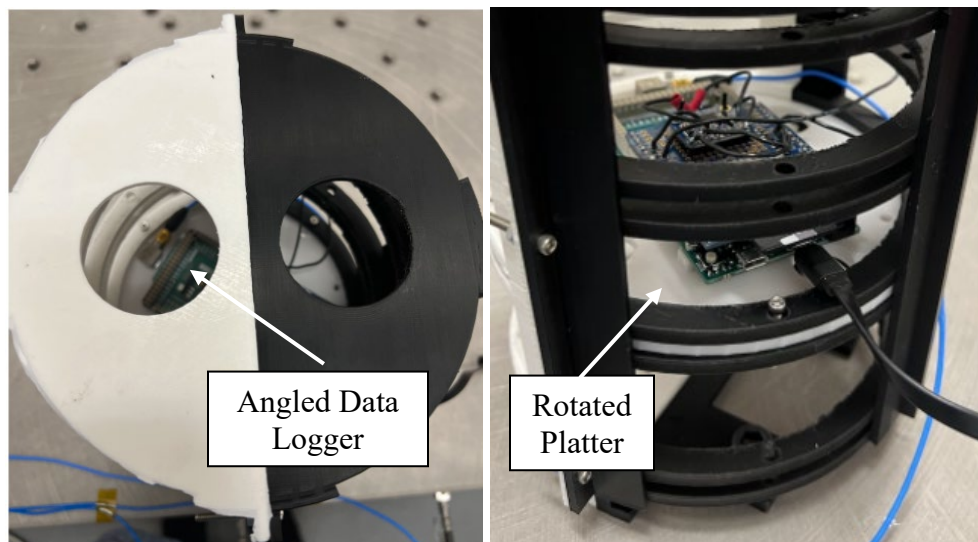


Figure 53. AV Bay mounted to shaker table for horizontal testing

The data logger and ICP accelerometer, as seen in Figure 54, were placed at a 45 degree angle as the platter had to be rotated to allow access to the SD card and power cable. Due to this placement, the horizontal testing for the x and y axis would be similar. It was decided that the second horizontal testing would not be conducted and the tests would be concluded with the vertical testing, as seen in Figure 55.



The accelerometer part number is 353B18.

Figure 54. ICP accelerometer mounted to AV Bay

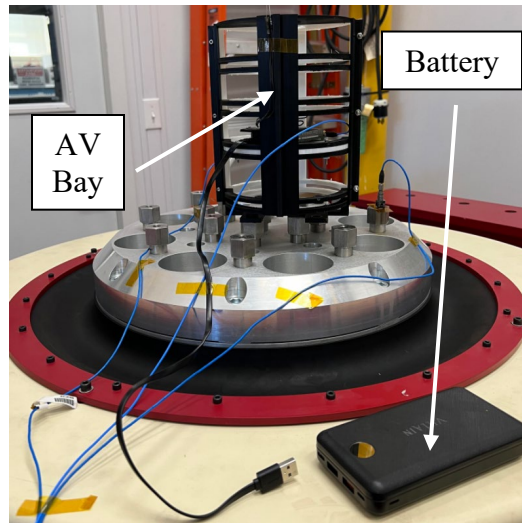


Figure 55. AV Bay mounted to the shaker table for vertical testing

The test flow, as seen in Figure 56, utilized the test equipment seen in Figure 57. The test, as seen in Figure 58, takes approximately an hour or less to run per axis, but extra time is used for instrumentation and setup (adjusting the table from horizontal to vertical testing takes approximately half an hour).

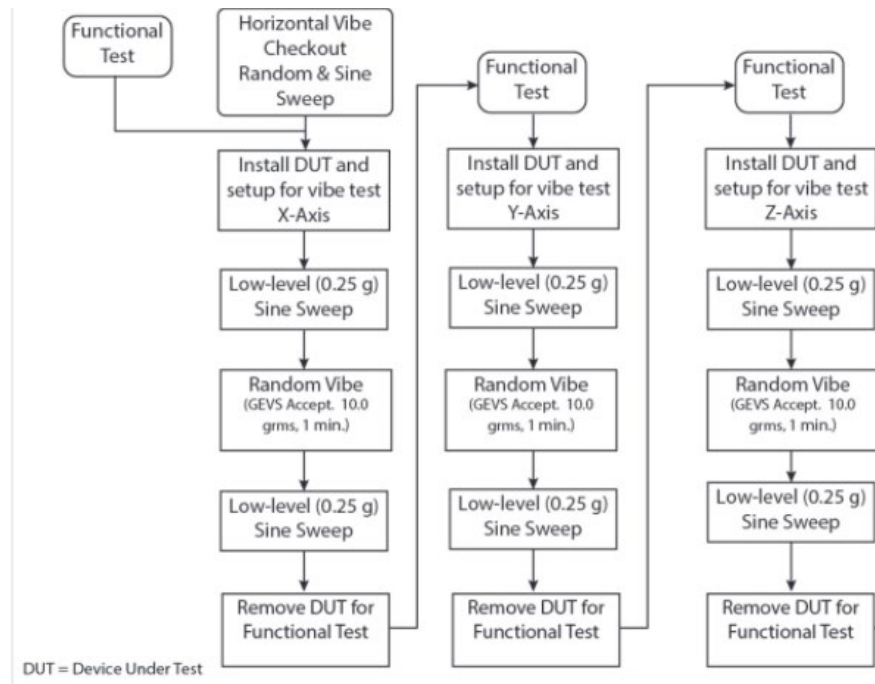


Figure 56. Shaker table test flow. Source: [27].

Name	Description
Data Physics 1216VH	Electrodynamic Shaker / Slip Table Monobase
DSA10-80	Power Amplifier
	Vibration Control Computer system with m+p software
VibRunner®	Data Acquisition Unit and Signal Source
PCB Piezoelectronics Single Axis Accelerometers	Used for control accelerometers
PCB Piezoelectronics Tri-axial Accelerometers	Used for vibration measurements
Shaker Adapter Plate	
Microdot to BNC Cables	

Figure 57. Shaker table test equipment. Source: [27].

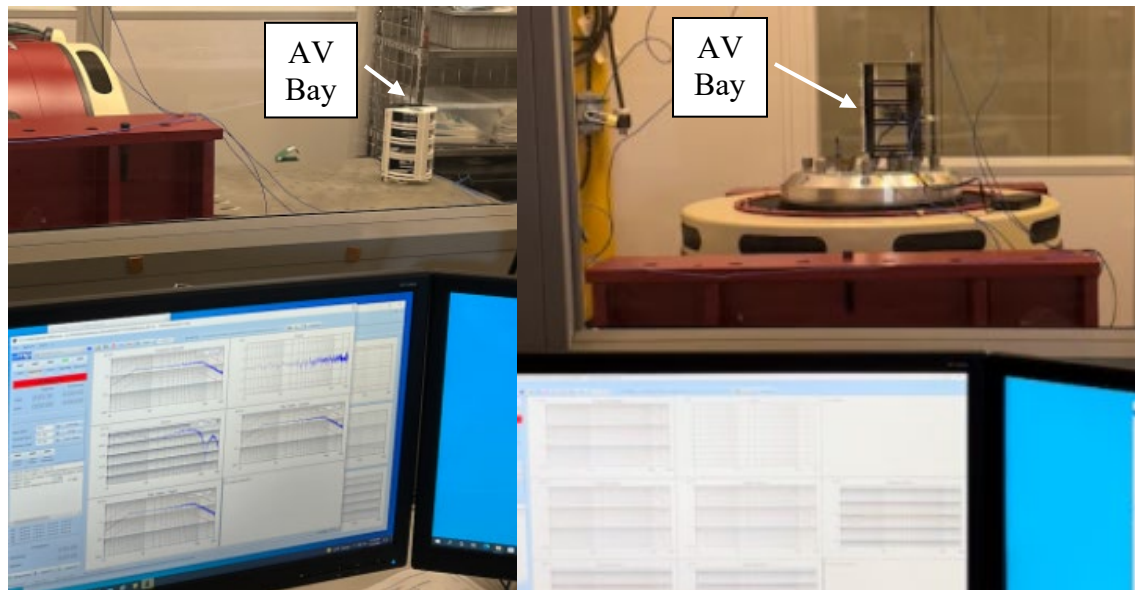


Figure 58. Horizontal and vertical shaker table testing of the AV Bay

The data logger's functional test was satisfactory prior to testing, but failed to log during testing. The response of the AV Bay to the horizontal test, as shown in Figure 59, compared the three axes of the accelerometer mounted to the platter to the input accelerometer mounted to the shaker table.

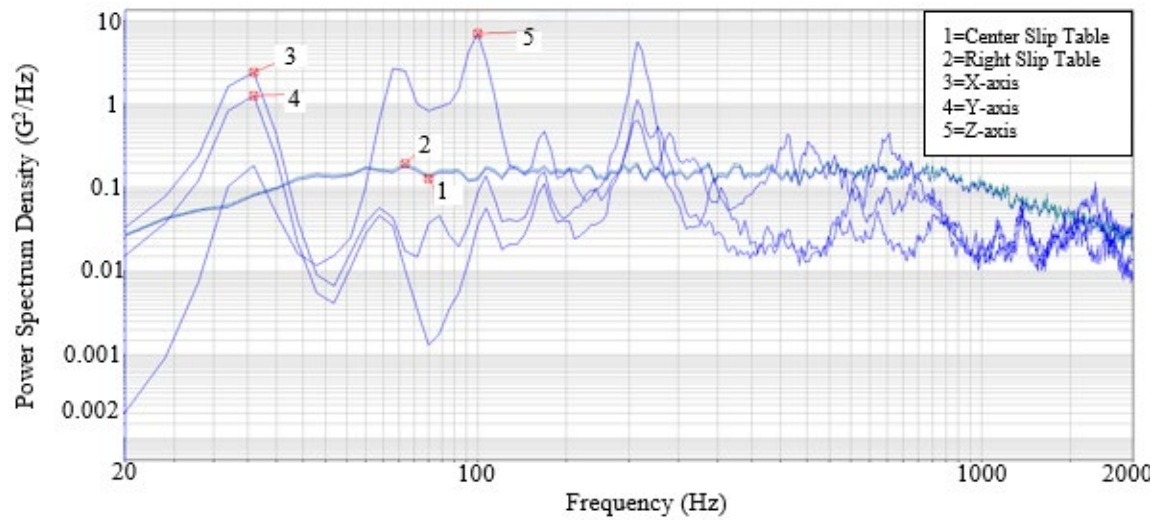


Figure 59. Horizontal vibrations test

The sine sweep characterizes the structure and was performed once prior to testing and once after as a test for structural changes. Figure 60, Figure 61, and Figure 62 compare the responses of the AV Bay for the initial sine sweep (blue) with the final sweep (green) for the x, y, and z-axes, respectively. The initial peaks of the graphs show the initial adjustment or settlement of the components of the device under test (DUT), such as the nuts and bolts. According to the test conductor, any difference of less than 10% in the x and y between the initial and final values are acceptable.

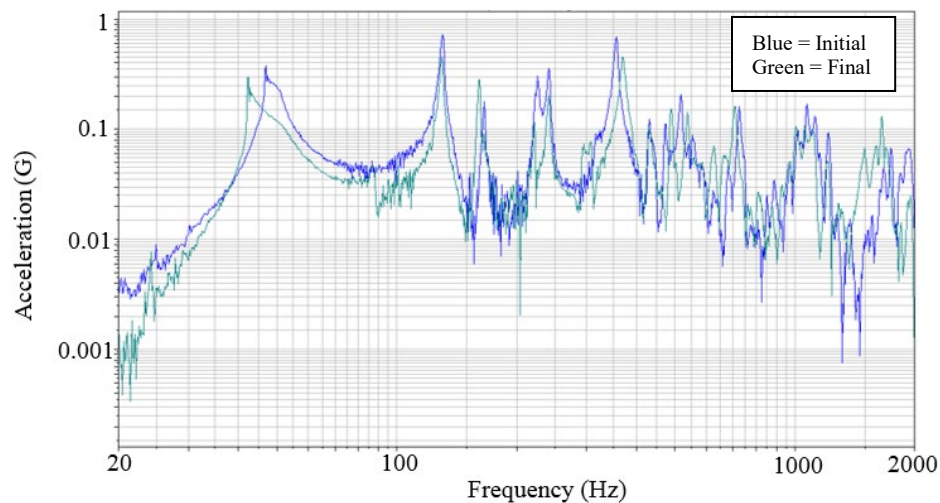


Figure 60. X-axis sine sweep test

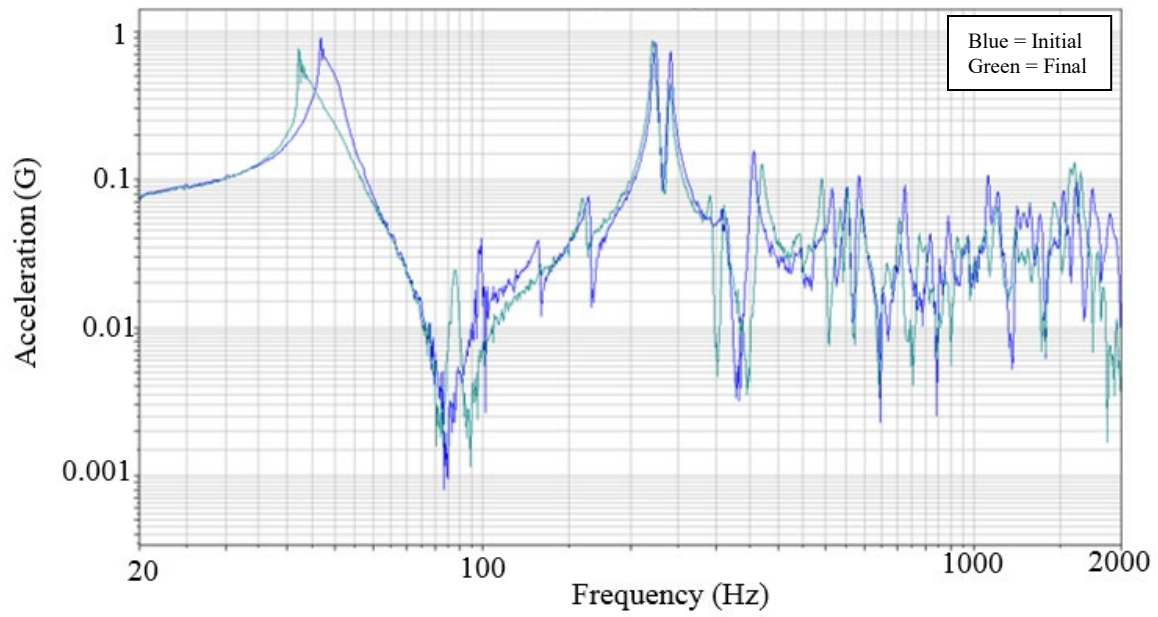


Figure 61. Y-axis sine sweep test

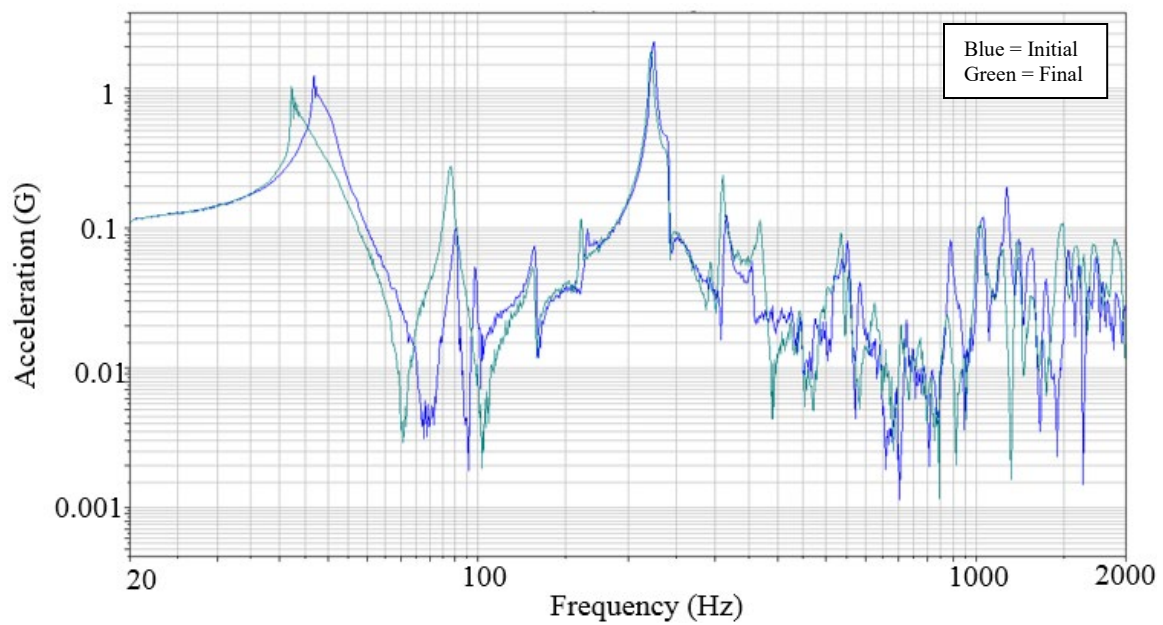


Figure 62. Z-axis sine sweep test

The vertical test ran and was stopped after 13 seconds, which is far below the specified time duration of 150 seconds, due to structural failure, as seen in Figure 63. Only data for the initial sine sweep, as seen in Figure 64, was collected, where the center slip table and the right slip table are represented in blue, while the z-axis is represented in green. The failure does not accurately simulate failures that could occur during flight as the DUT mounting is not indicative of the AV Bay mounting and complete securement in the RPV. It does expose areas of concern that could be addressed in the future.

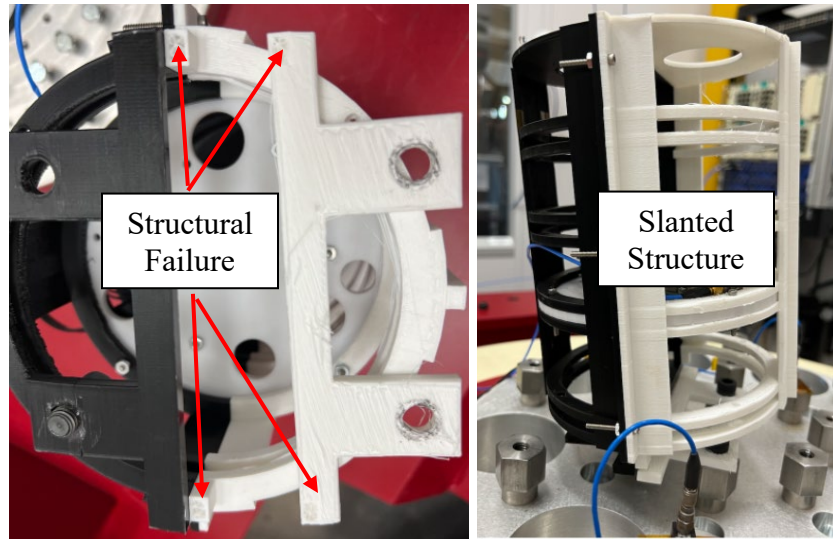


Figure 63. Structural failure for vertical testing

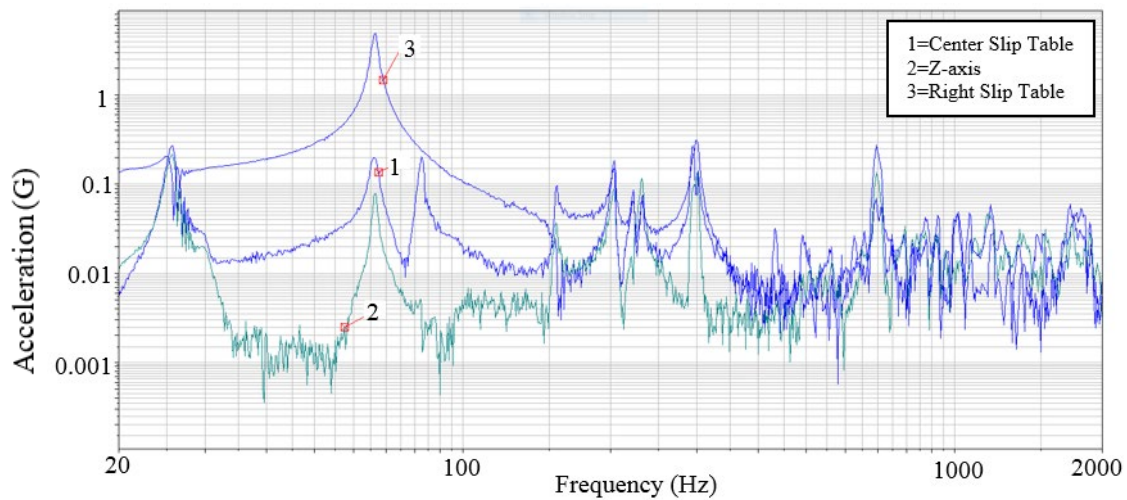


Figure 64. Initial vertical sine sweep

For future testing, it is recommended that the AV bay be mounted and secured in the coupler to better represent the inflight configuration and mounted to the shaker table with an adapter plate. The adapter plate will allow for a smooth transition between testing of the three axes. Additionally, it is recommended a weighted object be put in place for the battery to ensure proper weight is tested. Since a method to certify systems, hardware, and connections pre-flight was established, future testing should reveal issues, such as cold solder joints and poor harness connections, which should ensure reliability during flight testing.

IV. COST ANALYSIS

Table 6 breaks down the cost of the NPS RPV in five categories with more details found in Appendix B, which breaks down each component cost and link for purchases. Table 7 examines the cost for the AV Bay. Since COTS resources were used, the RPV resulted in a low-cost option as compared to other missile systems previously discussed.

Table 6. Overall RPV cost

Rocket Body & Fins	\$1,244.49
Motor & Hardware	\$1,291.91
Chutes & Accessories & CO2 Kit	\$1,680.84
Electronics (AV Bay, nose cone, separation, etc.)	\$3,699.37
Motor Cage	\$6,500.00
Grand Total	\$14,416.61

Table 7. Cost of AV Bay components

Component	Unit Cost	Quantity	Total Cost
LiPo battery	\$102.99	1	\$102.99
RPi battery	\$29.99	1	\$29.99
BNO055 IMU	\$34.95	1	\$34.95
Arduino Due	\$34.25	1	\$34.25
Arduino data logging shield	\$13.95	1	\$13.95
ADXL345 accelerometer	\$17.50	1	\$17.50
SD card	\$12.45	2	\$24.90
Raspberry Pi 4B/8Gb	\$75.00	1	\$75.00
RPi HAT	\$4.95	1	\$4.95
6-pin connector (female)	\$38.58	3	\$115.74
6-pin connector (male)	\$51.07	3	\$153.21
Grand Total			\$607.43

Additionally, Table 8 shows the print time and cost for PC with 0.254 mm layer thickness and 100% infill on the Stratasys Fortus 450 MC, which was determined by

software, as seen in Figure 65. The total print time was 35.03 hours (1.46 days) and with labor costs at \$125, the total cost for the bay was \$289.77.

Table 8. 3D printing time and cost of PC AV Bay

Component	Hrs	Mins	Support (in ³)	Cost (\$)	Material (in ³)	Cost
Battery	2	47	0.282	0.61	8.412	\$18.93
Arduino	1	32	0.282	0.61	5.747	\$12.93
RPi	1	42	0.278	0.60	5.849	\$13.16
BNO	1	39	0.282	0.61	5.981	\$16.46
Shell 1	13	41	9.210	19.82	14.273	\$32.11
Shell 2	13	41	9.210	19.82	14.273	\$32.11
Total						\$164.77

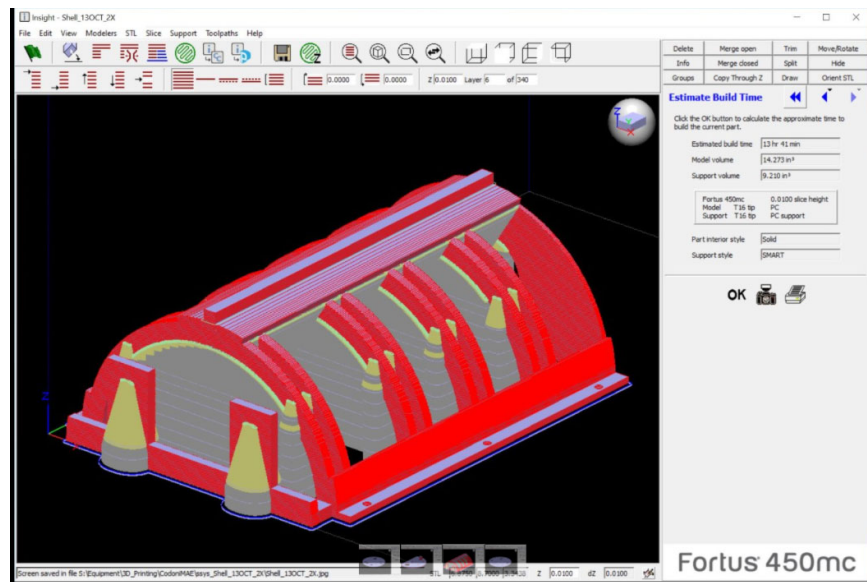


Figure 65. PC print orientation

In comparison, Table 9 shows the print time and cost for tough PLA with the following settings: 0.2 mm layer height, 50% infill, and support material set to everywhere, as shown in Figure 66. The total print time was 113.07 hours or 4.71 days with the total cost of \$85.91. Although tough PLA was three times cheaper, it took more than twice the time as PC. Both are viable options, but one may be more advantageous based on mission requirements.

Table 9. 3D printing time and cost of tough PLA AV Bay

Component	Hrs	Mins	Support (g)	Material(g)	Cost
Battery	12	34	0.57	125	\$8.36
Arduino	8	3	0.50	78	\$5.20
RPi	8	3	0.38	80	\$5.35
BNO	8	23	0.42	81	\$5.42
Shell X2	56	32	0.53	463	\$30.87
Total					\$85.91

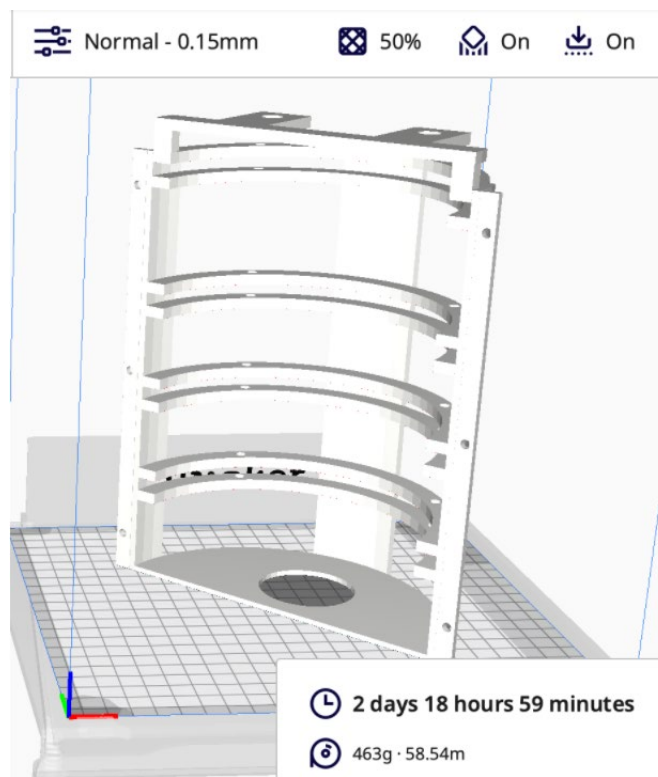


Figure 66. Tough PLA shell print orientation

THIS PAGE INTENTIONALLY LEFT BLANK

V. SUMMARY AND CONCLUSION

As rocket components become increasingly available commercially, the feasibility of a low-cost rocket-powered vehicle (RPV) with capabilities tailored to the current and future needs of the Navy or DOD increases. Based on the cost analysis, the NPS RPV is a viable low-cost option for the U.S. Navy as it can be tailored for unique applications or to provide a symmetric response to inexpensive potential threats. The NPS RPV Avionics (AV) Bay has been proven to be a critical component to the mission as it provides power, guidance, and control to the vehicle.

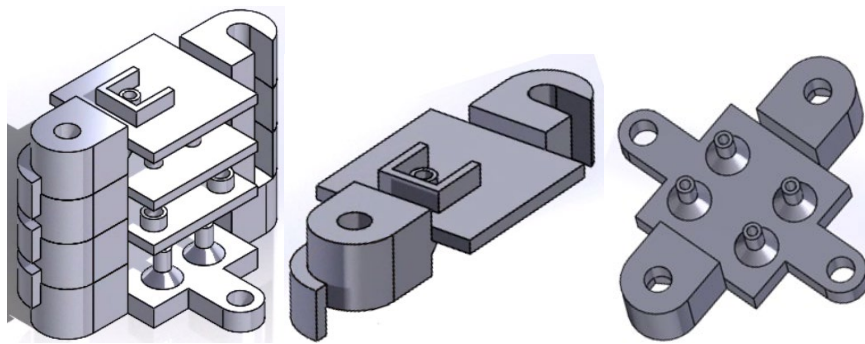
The modularity of the new AV Bay design can incorporate various hardware and software tailored to the changing needs and mission requirements due to the flexibility built into the platter design. The design is flexible since platters can be custom designed to electronics and can be easily inserted and removed as needed. Proven to be reliable under the expected flight loads, the connections have not only sustained launch and flight vibrations, some components and connections have sustained disastrous impacts. Improvements to future AV Bays include: a support structure that does not require as much support material to cut down print time without compromising structural integrity, a design to allow for different spacing between platters, and a lighter connection harness.

The initial vehicle health monitoring system was designed, built, and coupled with a data logging system with a high sampling rate that transmitted selected flight parameters during flight. In-flight transmission of selected data logger variables was demonstrated and assisted greatly in preserving valuable data for determining issues that have led to flight failures during post flight analysis. A data logger developed for collecting flight-test data can continue to be utilized in future flights to construct a representative structural vibration spectrum of a delivery vehicle over tactical flight profiles, such that future AV components be certified pre-flight utilizing a shaker table.

Although the process to ensure connectivity and functionality of the fully loaded AV Bay under simulated loads was not completed pre-flight, a protocol was established and demonstrated on the AV Bay structure, which will be utilized for future flights.

VI. FUTURE WORK RECOMMENDATIONS

A stackable concept, as seen in Figure 67, is recommended in the future as it allows the stresses of a launch to be transferred vertically to the bolts that would connect the various pieces. This concept is also better for 3D printing as the various pieces do not require the support material required by the shell.



Mounting stacks designed by Alexis Thoeny at the NPS Rocket Lab.

Figure 67. Stackable bay SolidWorks concept

As far as connections, the Omnetics connectors are also used for CubeSats, but are more costly than the Harwin connectors that were previously examined. Another concept is the battery matrix, as seen in Figure 68, as this solution incorporates the plug-in play concept that was originally desired, but may cost more than desired for a low-cost RPV.



Figure 68. CubeSat battery matrix. Source: [28].

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. CONNECTIONS

A. RPI

Table 10. RPi connections to other components

Component	On Component	On RPi
Relay	VOC	5V
	IN1	F4
	GND	GND
BNO055	SDA	F17
	SCL	F27
	Vin	3V3
	GND	GND
ADC	VOO	5V
	SDA	F17
	SCL	F27
	GND	GND
	A3	GND
Fan	Vin	5V
	GND	GND

Table 11. Connections for the six-pin connector

Connection Number	Component		Connection	
1	LiPo Battery		Positive	
2	LiPo Battery	RPi	Negative	GND
3	Servos		1	
4			3	
5	ADC		A2	

B. DATA LOGGER

Table 12. ADXL connections to the shield

Component	On Component	Shield
ADXL345	CS	D5
	SDO	MISO
	SDA	MOSI
	SCL	SCK
	VIN	5V
	GND	GND
LED	Positive	D7
	Negative	GND

C. TELEMETRY

Table 13. RPV transmitter shield connections

Component	On Component	Shield
Omega Pressure Transducer	Data	A0
	GND	GND
MPRLS Ported Pressure	SDA	SDA
	SCL	SCL
	Vin	5V
	GND	GND
LORA	RST	D2
	CS	D6
	MOSI	D11
	MISO	D12
	SCK	D13
	GND	D3

Table 14. Ground station receiver shield connections

Component	On Component	Shield
LORA	RST	D2
	CS	D6
	MOSI	D11
	MISO	D12
	SCK	D13
	GND	GND
Switch	Middle	5V
	GND	GND
	Left/Read	D7
LED	Indicator Pin	D9
	Ground	Left Switch

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. RPV COMPONENT COSTS

Component	Cost	Qty	Total Cost	Source
Kate 3.0 Transmitter + Pyro Opt 25	\$1,275.00	1	\$1,275.00	Multitronix
Kate Receiver	\$175.00	1	\$175.00	Multitronix
Kate Turn On Coil	\$95.00	1	\$95.00	Multitronix
Micro SD Card Ultra 128GB	\$19.95	1	\$19.95	PI Shop
Raspberry Pi 4B/8Gb	\$75.00	2	\$150.00	PI Shop
RPi Adafruit Perma-Proto HAT	\$4.95	2	\$9.90	Adafruit
Raspberry Pi Sense HAT	\$35.45	1	\$35.45	PI Shop
Raspberry Pi HQ Camera	\$50.00	1	\$50.00	PI Shop
RPi Camera Cable, 1m	\$3.95	1	\$3.95	PI Shop
RPi Camera Cable Joiner	\$3.25	1	\$3.25	PI Shop
RPi Camera Cable Cover	\$31.00	1	\$31.00	Wire Care
Portable Battery	\$29.99	3	\$89.97	Villain Electronics
MicroSDXC Card 64GB	\$12.45	2	\$24.90	PI Shop
RPi USB to Micro, 1.5 ft	\$3.99	2	\$7.98	Newegg
LiPo Battery	\$102.99	2	\$205.98	Thunder Power
Bosch BNO055	\$34.95	2	\$69.90	Digikey
4k Record & Camera System	\$89.00	1	\$89.00	FPV
HD Camera	\$58.99	1	\$58.99	FPV
Arduino Uno Atmega328	\$20.90	1	\$20.90	Digikey
MicroSD for Arduino	\$5.20	1	\$5.20	Digikey
4 Channel Relay Module - 5V	\$5.95	2	\$11.90	PI Shop
6 pin connector (Female)	\$38.58	10	\$385.80	Newegg
6 pin connector (Male)	\$51.07	10	\$510.70	Newegg
Perfect Flites	\$54.95	4	\$219.80	Merlin Missiles
Tough PLA	\$49.95	3	\$149.85	Indicate Tech
CO2 Kit 23–45gram	\$162.00	2	\$324.00	Fruity Chutes
Nosecone, 7.5"	\$139.95	1	\$139.95	Public Missiles
Coupler 7.3"x48"L	\$82.95	1	\$82.95	Public Missiles

Airframe, 7.5"x48"L Fiberglass	\$229.95	4	\$919.80	Public Missiles
Fins Sheet 0.1875"x12"	\$101.79	1	\$101.79	Public Missiles
Drogue, 60",CD 2.2	\$258.42	2	\$516.84	Rocketman
Parachute, 120",CD 2.2	\$315.00	2	\$630.00	Rocketman
Chute Flame Shield, 12"	\$20.50	4	\$82.00	Rocketman
Main Shock Cord 30ft. 2520lbs	\$48.00	2	\$96.00	Rocketman
Parachute Swivel, 15000 lbs	\$3.00	4	\$12.00	Rocketman
Parachute Quick Link, 1/4"	\$5.00	4	\$20.00	Rocketman
Motor, M3400	\$688.22	1	\$688.22	Dehate
Motor Hardware (Retainer, Spacer, Casing)	\$603.69	1	\$603.69	Dehate
Motor Cage Bracket	\$100.00	4	\$400.00	Custom
Motor Cage Mount Disk	\$200.00	1	\$200.00	Custom
Motor Cage Plate	\$100.00	1	\$100.00	Custom
Motor Cage Mount Brackets	\$25.00	4	\$100.00	Custom
Motor Cage Servos	\$250.00	4	\$1,000.00	Custom
Motor Cage Gear Box	\$750.00	4	\$3,000.00	Custom
Motor Cage Bearings	\$25.00	4	\$100.00	Custom
Motor Cage Fins	\$150.00	4	\$600.00	Custom
Motor Cage Shaft	\$50.00	4	\$200.00	Custom
MC Titanium Mount Bracket	\$200.00	4	\$800.00	Custom
	Grand Total		\$14,416.61	

APPENDIX C. ARDUINO CODE

A. GROUND STATION UNO

```
"#include <SD.h>
#include <Wire.h>
#include <RH_RF95.h>

// LoRa
#define RFM95_CS 6
#define RFM95_RST 2
#define RFM95_INT 3
#define RF95_FREQ 915.0

// LED
#define INDICATOR_PIN 9

const int toggle_read_pin = 7;

/*****SD Card*****/
const int chip_select = 10;
// File data_file;

/*****LoRa*****/
RH_RF95 rf95(RFM95_CS, RFM95_INT);

void setup(){
  Serial.begin(115200);
  pinMode(toggle_read_pin, INPUT);
  pinMode(INDICATOR_PIN, OUTPUT);

  // Setup SD card
  // if(!SD.begin(chip_select)){
  //   Serial.println("Unable to find SD card");
  //   while(1){}
  // }
  // else{
  //   Serial.println("FOUND SD CARD");
  // }
  // // Create a new file
  // char filename [] = "LOGGER00.CSV";
  // for (uint8_t i = 0; i < 100; i++) {
  //   filename [6] = i/10 + '0';
  //   filename [7] = i%10 + '0';
  //   if (! SD.exists(filename)) {
  //     // only open a new file if it doesn't exist
  //     data_file = SD.open(filename, FILE_WRITE);
  //     break; // leave the loop!
  //   }
  // }
  // Set the LoRa
  pinMode(RFM95_RST, OUTPUT);
  digitalWrite(RFM95_RST, HIGH);
```

```

// manual reset
digitalWrite(RFM95_RST, LOW);
delay(10);
digitalWrite(RFM95_RST, HIGH);
delay(10);

if(!rf95.init()) {
    Serial.println("LoRa radio init failed");
}
else{
    Serial.println("LoRa radio initialized...");
}

// Defaults after init are 434.0MHz, modulation GFSK_Rb250Fd250,
+13dbM
if (!rf95.setFrequency(RF95_FREQ)) {
    Serial.println("setFrequency failed");
}
Serial.print("Frequency set to: "); Serial.println(RF95_FREQ);
}

void loop(){
    if(digitalRead(toggle_read_pin)){
        digitalWrite(INDICATOR_PIN, HIGH);

        char radiopacket [11];
        snprintf(radiopacket,8,"%c," 'r');
        rf95.send((uint8_t *)radiopacket, 11);
        rf95.waitPacketSent();

        uint8_t buf [RH_RF95_MAX_MESSAGE_LEN];
        uint8_t len = sizeof(buf);
        // Wait for record flag
        if(rf95.waitAvailableTimeout(1000)){
            if(rf95.recv(buf, &len)){
                Serial.println((char*)buf);
                // data_file.println((char *)buf);
            }
        }

    }
    else{
        digitalWrite(INDICATOR_PIN, LOW);
    }
}
}” Adapted from [29], [30], and [23].

```

B. ROCKET TRANSMITTER

```

#include <SPI.h>
#include <Wire.h>
#include <RH_RF95.h>
#include "Adafruit_MPRLS.h"

// LoRa

```

```

#define RFM95_CS 6
#define RFM95_RST 2
#define RFM95_INT 3
#define RF95_FREQ 915.0

// Indicator
#define LED_PIN 7

// MPRLS
#define RESET_PIN -1
#define EOC_PIN -1

/*****LoRa*****/
RH_RF95 rf95(RFM95_CS, RFM95_INT);

/*****MPRLS*****/
Adafruit_MPRLS mpr = Adafruit_MPRLS(RESET_PIN, EOC_PIN);

/**Analog Pressure Sensor**/
int analogPressurePin = A0;

bool digital_press = true;

void setup()
{
  // Begin Serial
  Serial.begin(115200);

  // Set the indicator LED
  pinMode(LED_PIN, OUTPUT);

  // Set the LoRa
  pinMode(RFM95_RST, OUTPUT);
  digitalWrite(RFM95_RST, HIGH);

  // manual reset
  digitalWrite(RFM95_RST, LOW);
  delay(10);
  digitalWrite(RFM95_RST, HIGH);
  delay(10);

  if(!rf95.init()) {
    Serial.println("LoRa radio init failed");
  }
  else{
    Serial.println("LoRa radio initialized...");
  }

  // Defaults after init are 434.0MHz, modulation GFSK_Rb250Fd250,
+13dbM
  if (!rf95.setFrequency(RF95_FREQ)) {
    Serial.println("setFrequency failed");
  }
  Serial.print("Frequency set to: "); Serial.println(RF95_FREQ);

```

```

// Set up MPRLS
if(!mpr.begin()){
  Serial.println("Failed to start MPRLS sensor...");
}
else{
  Serial.println("Started MPRLS sensor...");
}
}

void loop()
{
  // Read pressure from each device
  float pressure_hPa = mpr.readPressure();
  int analog_pressure = analogRead(analogPressurePin);

  // Begin transmitting
  char radiopacket [16];

  snprintf(radiopacket, 13, "%d,%d," (int)pressure_hPa,
(int)analog_pressure);

  Serial.println(radiopacket);
  rf95.send((uint8_t *)radiopacket, 16);
  rf95.waitPacketSent();
}" Adapted from [29], [30], [23], and [31].

```

C. DATA LOGGER

```


/**
 * This program logs data to a binary file.
 * Samples are logged at regular intervals. The maximum logging rate
 * depends on the quality of your SD card and the time required to
 * read sensor data. This example has been tested at 500 Hz with
 * good SD card on an Uno. 4000 HZ is possible on a Due.
 *
 * Data is written to the file using a SD multiple block write command.
 */
#include <SPI.h>
#include "SdFat.h"
#include "FreeStack.h"
#include "UserTypes.h"

#ifdef __AVR_ATmega328P__
#include "MinimumSerial.h"
MinimumSerial MinSerial;
#define Serial MinSerial
#endif // __AVR_ATmega328P__

//=====
// Start of configuration constants.
//=====


```

```

// Abort run on an overrun.
#define ABORT_ON_OVERRUN 1

#define INDICATOR_LED 7

// Variables for shaker table test
// Number of Gs to trigger
int g_trigger = 1;
// CHECK THIS make sure the resting g value is known
int base_g = 112;
// Are we at a high g?
bool high_g = false;
// Length of time to record in milliseconds
long int record_time = 60000;
// Where we hold data
data_t accel_data;

// Blink when we're in main loop
long int blink_time = 500; // millis
long int start_blink = 0;

//-----
//Interval between data records in microseconds.
const uint32_t LOG_INTERVAL_USEC = 200;

//-----
// Set USE_SHARED_SPI non-zero for use of an SPI sensor.
// May not work for some cards.
#ifdef USE_SHARED_SPI
#define USE_SHARED_SPI 0
#endif // USE_SHARED_SPI

//-----
// Pin definitions.
//
// SD chip select pin.
const uint8_t SD_CS_PIN = 10;

//-----
// File definitions.
//
// Maximum file size in blocks.
// The program creates a contiguous file with FILE_BLOCK_COUNT 512 byte
// blocks.
// This file is flash erased using special SD commands. The file will
// be
// truncated if logging is stopped early.
const uint32_t FILE_BLOCK_COUNT = 256000;
// log file base name if not defined in UserTypes.h
#ifdef FILE_BASE_NAME
#define FILE_BASE_NAME "data"

```

```

#endif // FILE_BASE_NAME

//-----
// Buffer definitions.
#ifndef RAMEND
// Assume ARM. Use total of ten 512 byte buffers.
const uint8_t BUFFER_BLOCK_COUNT = 10;
//
#elif RAMEND < 0X8FF
#error Too little SRAM
//
#elif RAMEND < 0X10FF
// Use total of two 512 byte buffers.
const uint8_t BUFFER_BLOCK_COUNT = 2;
//
#elif RAMEND < 0X20FF
// Use total of four 512 byte buffers.
const uint8_t BUFFER_BLOCK_COUNT = 4;
//
#else // RAMEND
// Use total of 12 512 byte buffers.
const uint8_t BUFFER_BLOCK_COUNT = 12;
#endif // RAMEND
//=====
// End of configuration constants.
//=====

// Temporary log file. Will be deleted if a reset or power failure
occurs.
#define TMP_FILE_NAME FILE_BASE_NAME "##.bin"

// Size of file base name.
const uint8_t BASE_NAME_SIZE = sizeof(FILE_BASE_NAME) - 1;
const uint8_t FILE_NAME_DIM = BASE_NAME_SIZE + 7;
char binName [FILE_NAME_DIM] = FILE_BASE_NAME "00.bin";

SdFat sd;

SdBaseFile binFile;

// Number of data records in a block.
const uint16_t DATA_DIM = (512 - 4)/sizeof(data_t);

//Compute fill so block size is 512 bytes. FILL_DIM may be zero.
const uint16_t FILL_DIM = 512 - 4 - DATA_DIM*sizeof(data_t);

struct block_t {
    uint16_t count;
    uint16_t overrun;
    data_t data [DATA_DIM];
    uint8_t fill [FILL_DIM];
};

```

```

//=====
=====
// Error messages stored in flash.
#define error(msg) {sd.errorPrint(&Serial, F(msg));fatalBlink();}
//-----
-----
void fatalBlink() {
    while (true) {
        SysCall::yield();
        digitalWrite(INDICATOR_LED, HIGH);
        delay(200);
        digitalWrite(INDICATOR_LED, LOW);
        delay(200);
    }
}

//-----
-----
void deleteTempFile(){
    // Delete old tmp file.
    if (sd.exists(TMP_FILE_NAME)) {
        Serial.println(F("Deleting tmp file " TMP_FILE_NAME));
        if (!sd.remove(TMP_FILE_NAME)) {
            error("Can't remove tmp file");
        }
    }
}

//-----
-----
void createBinFile() {
    // max number of blocks to erase per erase call
    const uint32_t ERASE_SIZE = 262144L;
    uint32_t bgnBlock, endBlock;

    // Delete old tmp file.
    if (sd.exists(TMP_FILE_NAME)) {
        Serial.println(F("Deleting tmp file " TMP_FILE_NAME));
        if (!sd.remove(TMP_FILE_NAME)) {
            error("Can't remove tmp file");
        }
    }
    // Create new file.
    Serial.println(F("\nCreating new file"));
    binFile.close();
    if (!binFile.createContiguous(TMP_FILE_NAME, 512 * FILE_BLOCK_COUNT))
    {
        error("createContiguous failed");
    }
    // Get the address of the file on the SD.
    if (!binFile.contiguousRange(&bgnBlock, &endBlock)) {
        error("contiguousRange failed");
    }
    // Flash erase all data in the file.

```

```

Serial.println(F("Erasing all data in file"));
uint32_t bgnErase = bgnBlock;
uint32_t endErase;
while (bgnErase < endBlock) {
    endErase = bgnErase + ERASE_SIZE;
    if (endErase > endBlock) {
        endErase = endBlock;
    }
    if (!sd.card()->erase(bgnErase, endErase)) {
        error("erase failed");
    }
    bgnErase = endErase + 1;
}
}

//-----
void renameBinFile() {
    while (sd.exists(binName)) {
        if (binName [BASE_NAME_SIZE + 1] != '9') {
            binName [BASE_NAME_SIZE + 1]++;
        } else {
            binName [BASE_NAME_SIZE + 1] = '0';
            if (binName [BASE_NAME_SIZE] == '9') {
                error("Can't create file name");
            }
            binName [BASE_NAME_SIZE]++;
        }
    }
    if (!binFile.rename(binName)) {
        error("Can't rename file");
    }
    Serial.print(F("File renamed: "));
    Serial.println(binName);
    Serial.print(F("File size: "));
    Serial.print(binFile.fileSize()/512);
    Serial.println(F(" blocks"));
}

//-----
void recordBinFile() {
    const uint8_t QUEUE_DIM = BUFFER_BLOCK_COUNT + 1;
    // Index of last queue location.
    const uint8_t QUEUE_LAST = QUEUE_DIM - 1;

    // Allocate extra buffer space.
    block_t block [BUFFER_BLOCK_COUNT - 1];

    block_t* curBlock = 0;

    block_t* emptyStack [BUFFER_BLOCK_COUNT];
    uint8_t emptyTop;
    uint8_t minTop;

```

```

block_t* fullQueue [QUEUE_DIM];
uint8_t fullHead = 0;
uint8_t fullTail = 0;

// Use SdFat's internal buffer.
emptyStack [0] = (block_t*)sd.vol()->cacheClear();
if (emptyStack [0] == 0) {
    error("cacheClear failed");
}
// Put rest of buffers on the empty stack.
for (int i = 1; i < BUFFER_BLOCK_COUNT; i++) {
    emptyStack [i] = &block [i - 1];
}
emptyTop = BUFFER_BLOCK_COUNT;
minTop = BUFFER_BLOCK_COUNT;

// Start a multiple block write.
if (!sd.card()->writeStart(binFile.firstBlock())) {
    error("writeStart failed");
}
Serial.print(F("FreeStack: "));
Serial.println(FreeStack());

bool closeFile = false;
uint32_t bn = 0;
uint32_t maxLatency = 0;
uint32_t overrun = 0;
uint32_t overrunTotal = 0;
uint32_t logTime = micros();

long int start_record = millis();
while(1) {
    // Time for next data record.
    logTime += LOG_INTERVAL_USEC;
    if ((millis() - start_record) > record_time) {
        closeFile = true;
    }
    if (closeFile) {
        if (curBlock != 0) {
            // Put buffer in full queue.
            fullQueue [fullHead] = curBlock;
            fullHead = fullHead < QUEUE_LAST ? fullHead + 1 : 0;
            curBlock = 0;
        }
    }
    else {
        if (curBlock == 0 && emptyTop != 0) {
            curBlock = emptyStack [--emptyTop];
            if (emptyTop < minTop) {
                minTop = emptyTop;
            }
            curBlock->count = 0;
            curBlock->overrun = overrun;
            overrun = 0;
        }
    }
}

```

```

    if ((int32_t)(logTime - micros()) < 0) {
        error("Rate too fast");
    }
    int32_t delta;
    do {
        delta = micros() - logTime;
    } while (delta < 0);
    if (curBlock == 0) {
        overrun++;
        overrunTotal++;

        digitalWrite(INDICATOR_LED, HIGH);

#ifdef ABORT_ON_OVERRUN
        Serial.println(F("Overrun abort"));
        break;
#endif // ABORT_ON_OVERRUN
    } else {
#ifdef USE_SHARED_SPI
        sd.card()->spiStop();
#endif // USE_SHARED_SPI
        acquireData(&curBlock->data [curBlock->count++]);
#ifdef USE_SHARED_SPI
        sd.card()->spiStart();
#endif // USE_SHARED_SPI
        if (curBlock->count == DATA_DIM) {
            fullQueue [fullHead] = curBlock;
            fullHead = fullHead < QUEUE_LAST ? fullHead + 1 : 0;
            curBlock = 0;
        }
    }
}
if (fullHead == fullTail) {
    // Exit loop if done.
    if (closeFile) {
        break;
    }
} else if (!sd.card()->isBusy()) {
    // Get address of block to write.
    block_t* pBlock = fullQueue [fullTail];
    fullTail = fullTail < QUEUE_LAST ? fullTail + 1 : 0;
    // Write block to SD.
    uint32_t usec = micros();
    if (!sd.card()->writeData((uint8_t*)pBlock)) {
        error("write data failed");
    }
    usec = micros() - usec;
    if (usec > maxLatency) {
        maxLatency = usec;
    }
    // Move block to empty queue.
    emptyStack [emptyTop++] = pBlock;
    bn++;
    if (bn == FILE_BLOCK_COUNT) {
        // File full so stop

```

```

        break;
    }
}

if (!sd.card()->writeStop()) {
    error("writeStop failed");
}
Serial.print(F("Min Free buffers: "));
Serial.println(minTop);
Serial.print(F("Max block write usec: "));
Serial.println(maxLatency);
Serial.print(F("Overruns: "));
Serial.println(overrunTotal);
// Truncate file if recording stopped early.
if (bn != FILE_BLOCK_COUNT) {
    Serial.println(F("Truncating file"));
    if (!binFile.truncate(512L * bn)) {
        error("Can't truncate file");
    }
}
}

//-----
void setup(void) {
    Serial.begin(9600);

    FreeStack();
    Serial.print(F("Records/block: "));

    if (sizeof(block_t) != 512) {
        error("Invalid block size");
    }
    // Allow userSetup access to SPI bus.
    pinMode(SD_CS_PIN, OUTPUT);
    digitalWrite(SD_CS_PIN, HIGH);

    // Tell us if we're recording or not
    pinMode(INDICATOR_LED, OUTPUT);
    digitalWrite(INDICATOR_LED, LOW);

    // Setup sensors.
    userSetup();

    // Initialize at the highest speed supported by the board that is
    // not over 50 MHz. Try a lower speed if SPI errors occur.
    if (!sd.begin(SD_CS_PIN, SD_SCK_MHZ(40))) {
        sd.initErrorPrint(&Serial);
        fatalBlink();
    }

    start_blink = millis();
}

```

```

//-----
-----
void loop(void) {
#if WDT_YIELD_TIME_MICROS
    fatalBlink();
#endif
    // Create the binary file to store our data
    createBinFile();
    while(1){
        // Let the world know we're in the loop
        if((millis() - start_blink) > blink_time){
            if(digitalRead(INDICATOR_LED)){
                digitalWrite(INDICATOR_LED, LOW);
            }
            else{
                digitalWrite(INDICATOR_LED, HIGH);
            }
            start_blink = millis();
        }

        // Get measurements
        delayMicroseconds(250);
        acquireData(&accel_data);
        if(accel_data.accel [2] > (base_g + base_g * g_trigger)){
            high_g = true;
        }

        // Wait for high G
        if(high_g){
            // Begin writing to SD card for n seconds
            digitalWrite(INDICATOR_LED, HIGH);
            recordBinFile();
            digitalWrite(INDICATOR_LED, LOW);
            // Attempt to rename the file to avoid overlap
            renameBinFile();
            deleteTempFile();
            while(1){
                digitalWrite(INDICATOR_LED, HIGH);
                delay(100);
                digitalWrite(INDICATOR_LED, LOW);
                delay(100);
            }
        }
    }
}
}” Adapted from [29], [23], and [22].

```

D. BIN TO CSV CONVERSION

```

"/**
 * This program logs data to a binary file. Functions are included
 * to convert the binary file to a csv text file.
 *
 * Samples are logged at regular intervals. The maximum logging rate
 * depends on the quality of your SD card and the time required to
 * read sensor data. This example has been tested at 500 Hz with

```

```

* good SD card on an Uno. 4000 HZ is possible on a Due.
*
* If your SD card has a long write latency, it may be necessary to use
* slower sample rates. Using a Mega Arduino helps overcome latency
* problems since 12 512 byte buffers will be used.
*
* Data is written to the file using a SD multiple block write command.
*/
#include <SPI.h>
#include "SdFat.h"
#include "FreeStack.h"
#include "UserTypes.h"

#ifdef __AVR_ATmega328P__
#include "MinimumSerial.h"
MinimumSerial MinSerial;
#define Serial MinSerial
#endif // __AVR_ATmega328P__
//=====
// Start of configuration constants.
//=====
// Abort run on an overrun. Data before the overrun will be saved.
#define ABORT_ON_OVERRUN 1
//-----
//Interval between data records in microseconds.
const uint32_t LOG_INTERVAL_USEC = 2000;
//-----
// Set USE_SHARED_SPI non-zero for use of an SPI sensor.
// May not work for some cards.
#ifndef USE_SHARED_SPI
#define USE_SHARED_SPI 0
#endif // USE_SHARED_SPI
//-----
// Pin definitions.
//
// SD chip select pin.
const uint8_t SD_CS_PIN = SS;
//
// Digital pin to indicate an error, set to -1 if not used.
// The led blinks for fatal errors. The led goes on solid for
// overrun errors and logging continues unless ABORT_ON_OVERRUN
// is non-zero.
#ifdef ERROR_LED_PIN
#undef ERROR_LED_PIN
#endif // ERROR_LED_PIN
const int8_t ERROR_LED_PIN = -1;
//-----
// File definitions.
//

```

```

// Maximum file size in blocks.
// The program creates a contiguous file with FILE_BLOCK_COUNT 512 byte
blocks.
// This file is flash erased using special SD commands. The file will
be
// truncated if logging is stopped early.
const uint32_t FILE_BLOCK_COUNT = 256000;
//
// log file base name if not defined in UserTypes.h
#ifndef FILE_BASE_NAME
#define FILE_BASE_NAME "data"
#endif // FILE_BASE_NAME
//-----
//
// Buffer definitions.
//
// The logger will use SdFat's buffer plus BUFFER_BLOCK_COUNT-1
additional
// buffers.
//
#ifndef RAMEND
// Assume ARM. Use total of ten 512 byte buffers.
const uint8_t BUFFER_BLOCK_COUNT = 10;
//
#elif RAMEND < 0X8FF
#error Too little SRAM
//
#elif RAMEND < 0X10FF
// Use total of two 512 byte buffers.
const uint8_t BUFFER_BLOCK_COUNT = 2;
//
#elif RAMEND < 0X20FF
// Use total of four 512 byte buffers.
const uint8_t BUFFER_BLOCK_COUNT = 4;
//
#else // RAMEND
// Use total of 12 512 byte buffers.
const uint8_t BUFFER_BLOCK_COUNT = 12;
#endif // RAMEND
//=====
//
// End of configuration constants.
//=====
//
// Temporary log file. Will be deleted if a reset or power failure
occurs.
#define TMP_FILE_NAME FILE_BASE_NAME "##.bin"

// Size of file base name.
const uint8_t BASE_NAME_SIZE = sizeof(FILE_BASE_NAME) - 1;
const uint8_t FILE_NAME_DIM = BASE_NAME_SIZE + 7;
char binName [FILE_NAME_DIM] = FILE_BASE_NAME "00.bin";

SdFat sd;

```

```

SdBaseFile binFile;

// Number of data records in a block.
const uint16_t DATA_DIM = (512 - 4)/sizeof(data_t);

//Compute fill so block size is 512 bytes. FILL_DIM may be zero.
const uint16_t FILL_DIM = 512 - 4 - DATA_DIM*sizeof(data_t);

struct block_t {
    uint16_t count;
    uint16_t overrun;
    data_t data [DATA_DIM];
    uint8_t fill [FILL_DIM];
};
//=====
// Error messages stored in flash.
#define error(msg) {sd.errorPrint(&Serial, F(msg));fatalBlink();}
//-----
//
void fatalBlink() {
    while (true) {
        SysCall::yield();
        if (ERROR_LED_PIN >= 0) {
            digitalWrite(ERROR_LED_PIN, HIGH);
            delay(200);
            digitalWrite(ERROR_LED_PIN, LOW);
            delay(200);
        }
    }
}
//-----
// read data file and check for overruns
void checkOverrun() {
    bool headerPrinted = false;
    block_t block;
    uint32_t bn = 0;

    if (!binFile.isOpen()) {
        Serial.println();
        Serial.println(F("No current binary file"));
        return;
    }
    binFile.rewind();
    Serial.println();
    Serial.print(F("FreeStack: "));
    Serial.println(FreeStack());
    Serial.println(F("Checking overrun errors - type any character to
stop"));
    while (binFile.read(&block, 512) == 512) {
        if (block.count == 0) {
            break;
        }
    }
}

```

```

    if (block.overrun) {
        if (!headerPrinted) {
            Serial.println();
            Serial.println(F("Overruns:"));
            Serial.println(F("fileBlockNumber,sdBlockNumber,overrunCount"));
            headerPrinted = true;
        }
        Serial.print(bn);
        Serial.print(',');
        Serial.print(binFile.firstBlock() + bn);
        Serial.print(',');
        Serial.println(block.overrun);
    }
    bn++;
}
if (!headerPrinted) {
    Serial.println(F("No errors found"));
} else {
    Serial.println(F("Done"));
}
}
//-----
// Convert binary file to csv file.
void binaryToCsv() {
    uint8_t lastPct = 0;
    block_t block;
    uint32_t t0 = millis();
    uint32_t syncCluster = 0;
    SdFile csvFile;
    char csvName [FILE_NAME_DIM];

    if (!binFile.isOpen()) {
        Serial.println();
        Serial.println(F("No current binary file"));
        return;
    }
    Serial.println();
    Serial.print(F("FreeStack: "));
    Serial.println(FreeStack());

    // Create a new csvFile.
    strcpy(csvName, binName);
    strcpy(&csvName [BASE_NAME_SIZE + 3], "csv");

    if (!csvFile.open(csvName, O_WRONLY | O_CREAT | O_TRUNC)) {
        error("open csvFile failed");
    }
    binFile.rewind();
    Serial.print(F("Writing: "));
    Serial.print(csvName);
    Serial.println(F(" - type any character to stop"));
    printHeader(&csvFile);
    uint32_t tPct = millis();
    while (!Serial.available() && binFile.read(&block, 512) == 512) {

```

```

uint16_t i;
if (block.count == 0 || block.count > DATA_DIM) {
    break;
}
if (block.overrun) {
    csvFile.print(F("OVERRUN,"));
    csvFile.println(block.overrun);
}
for (i = 0; i < block.count; i++) {
    printData(&csvFile, &block.data [i]);
}
if (csvFile.curCluster() != syncCluster) {
    csvFile.sync();
    syncCluster = csvFile.curCluster();
}
if ((millis() - tPct) > 1000) {
    uint8_t pct = binFile.curPosition() / (binFile.fileSize() / 100);
    if (pct != lastPct) {
        tPct = millis();
        lastPct = pct;
        Serial.print(pct, DEC);
        Serial.println('%');
    }
}
if (Serial.available()) {
    break;
}
}
csvFile.close();
Serial.print(F("Done: "));
Serial.print(0.001 * (millis() - t0));
Serial.println(F(" Seconds"));
}
//-----
-----
void createBinFile() {
    // max number of blocks to erase per erase call
    const uint32_t ERASE_SIZE = 262144L;
    uint32_t bgnBlock, endBlock;

    // Delete old tmp file.
    if (sd.exists(TMP_FILE_NAME)) {
        Serial.println(F("Deleting tmp file " TMP_FILE_NAME));
        if (!sd.remove(TMP_FILE_NAME)) {
            error("Can't remove tmp file");
        }
    }
    // Create new file.
    Serial.println(F("\nCreating new file"));
    binFile.close();
    if (!binFile.createContiguous(TMP_FILE_NAME, 512 * FILE_BLOCK_COUNT))
    {
        error("createContiguous failed");
    }
    // Get the address of the file on the SD.

```

```

    if (!binFile.contiguousRange(&bgnBlock, &endBlock)) {
        error("contiguousRange failed");
    }
    // Flash erase all data in the file.
    Serial.println(F("Erasing all data"));
    uint32_t bgnErase = bgnBlock;
    uint32_t endErase;
    while (bgnErase < endBlock) {
        endErase = bgnErase + ERASE_SIZE;
        if (endErase > endBlock) {
            endErase = endBlock;
        }
        if (!sd.card()->erase(bgnErase, endErase)) {
            error("erase failed");
        }
        bgnErase = endErase + 1;
    }
}
//-----
// dump data file to Serial
void dumpData() {
    block_t block;
    if (!binFile.isOpen()) {
        Serial.println();
        Serial.println(F("No current binary file"));
        return;
    }
    binFile.rewind();
    Serial.println();
    Serial.println(F("Type any character to stop"));
    delay(1000);
    printHeader(&Serial);
    while (!Serial.available() && binFile.read(&block , 512) == 512) {
        if (block.count == 0) {
            break;
        }
        if (block.overrun) {
            Serial.print(F("OVERRUN,"));
            Serial.println(block.overrun);
        }
        for (uint16_t i = 0; i < block.count; i++) {
            printData(&Serial, &block.data [i]);
        }
    }
    Serial.println(F("Done"));
}
//-----
// log data
void logData() {
    createBinFile();
    recordBinFile();
    renameBinFile();
}

```

```

//-----
void openBinFile() {
    char name [FILE_NAME_DIM];
    strcpy(name, binName);
    Serial.println(F("\nEnter two digit version"));
    Serial.write(name, BASE_NAME_SIZE);
    for (int i = 0; i < 2; i++) {
        while (!Serial.available()) {
            SysCall::yield();
        }
        char c = Serial.read();
        Serial.write(c);
        if (c < '0' || c > '9') {
            Serial.println(F("\nInvalid digit"));
            return;
        }
        name [BASE_NAME_SIZE + i] = c;
    }
    Serial.println(&name [BASE_NAME_SIZE+2]);
    if (!sd.exists(name)) {
        Serial.println(F("File does not exist"));
        return;
    }
    binFile.close();
    strcpy(binName, name);
    if (!binFile.open(binName, O_RDONLY)) {
        Serial.println(F("open failed"));
        return;
    }
    Serial.println(F("File opened"));
}
//-----

void recordBinFile() {
    const uint8_t QUEUE_DIM = BUFFER_BLOCK_COUNT + 1;
    // Index of last queue location.
    const uint8_t QUEUE_LAST = QUEUE_DIM - 1;

    // Allocate extra buffer space.
    block_t block [BUFFER_BLOCK_COUNT - 1];

    block_t* curBlock = 0;

    block_t* emptyStack [BUFFER_BLOCK_COUNT];
    uint8_t emptyTop;
    uint8_t minTop;

    block_t* fullQueue [QUEUE_DIM];
    uint8_t fullHead = 0;
    uint8_t fullTail = 0;

    // Use SdFat's internal buffer.
    emptyStack [0] = (block_t*)sd.vol()->cacheClear();
    if (emptyStack [0] == 0) {

```

```

    error("cacheClear failed");
}
// Put rest of buffers on the empty stack.
for (int i = 1; i < BUFFER_BLOCK_COUNT; i++) {
    emptyStack [i] = &block [i - 1];
}
emptyTop = BUFFER_BLOCK_COUNT;
minTop = BUFFER_BLOCK_COUNT;

// Start a multiple block write.
if (!sd.card()->writeStart(binFile.firstBlock())) {
    error("writeStart failed");
}
Serial.print(F("FreeStack: "));
Serial.println(FreeStack());
Serial.println(F("Logging - type any character to stop"));
bool closeFile = false;
uint32_t bn = 0;
uint32_t maxLatency = 0;
uint32_t overrun = 0;
uint32_t overrunTotal = 0;
uint32_t logTime = micros();
while(1) {
    // Time for next data record.
    logTime += LOG_INTERVAL_USEC;
    if (Serial.available()) {
        closeFile = true;
    }
    if (closeFile) {
        if (curBlock != 0) {
            // Put buffer in full queue.
            fullQueue [fullHead] = curBlock;
            fullHead = fullHead < QUEUE_LAST ? fullHead + 1 : 0;
            curBlock = 0;
        }
    } else {
        if (curBlock == 0 && emptyTop != 0) {
            curBlock = emptyStack [--emptyTop];
            if (emptyTop < minTop) {
                minTop = emptyTop;
            }
            curBlock->count = 0;
            curBlock->overrun = overrun;
            overrun = 0;
        }
        if ((int32_t)(logTime - micros()) < 0) {
            error("Rate too fast");
        }
        int32_t delta;
        do {
            delta = micros() - logTime;
        } while (delta < 0);
        if (curBlock == 0) {
            overrun++;
            overrunTotal++;
        }
    }
}

```

```

        if (ERROR_LED_PIN >= 0) {
            digitalWrite(ERROR_LED_PIN, HIGH);
        }
    #if ABORT_ON_OVERRUN
        Serial.println(F("Overrun abort"));
        break;
    #endif // ABORT_ON_OVERRUN
    } else {
    #if USE_SHARED_SPI
        sd.card()->spiStop();
    #endif // USE_SHARED_SPI
        acquireData(&curBlock->data [curBlock->count++]);
    #if USE_SHARED_SPI
        sd.card()->spiStart();
    #endif // USE_SHARED_SPI
        if (curBlock->count == DATA_DIM) {
            fullQueue [fullHead] = curBlock;
            fullHead = fullHead < QUEUE_LAST ? fullHead + 1 : 0;
            curBlock = 0;
        }
    }
}

if (fullHead == fullTail) {
    // Exit loop if done.
    if (closeFile) {
        break;
    }
} else if (!sd.card()->isBusy()) {
    // Get address of block to write.
    block_t* pBlock = fullQueue [fullTail];
    fullTail = fullTail < QUEUE_LAST ? fullTail + 1 : 0;
    // Write block to SD.
    uint32_t usec = micros();
    if (!sd.card()->writeData((uint8_t*)pBlock)) {
        error("write data failed");
    }
    usec = micros() - usec;
    if (usec > maxLatency) {
        maxLatency = usec;
    }
    // Move block to empty queue.
    emptyStack [emptyTop++] = pBlock;
    bn++;
    if (bn == FILE_BLOCK_COUNT) {
        // File full so stop
        break;
    }
}

if (!sd.card()->writeStop()) {
    error("writeStop failed");
}

Serial.print(F("Min Free buffers: "));
Serial.println(minTop);
Serial.print(F("Max block write usec: "));

```

```

Serial.println(maxLatency);
Serial.print(F("Overruns: "));
Serial.println(overrunTotal);
// Truncate file if recording stopped early.
if (bn != FILE_BLOCK_COUNT) {
    Serial.println(F("Truncating file"));
    if (!binFile.truncate(512L * bn)) {
        error("Can't truncate file");
    }
}
}
}
//-----
-----
void recoverTmpFile() {
    uint16_t count;
    if (!binFile.open(TMP_FILE_NAME, O_RDWR)) {
        return;
    }
    if (binFile.read(&count, 2) != 2 || count != DATA_DIM) {
        error("Please delete existing " TMP_FILE_NAME);
    }
    Serial.println(F("\nRecovering data in tmp file " TMP_FILE_NAME));
    uint32_t bgnBlock = 0;
    uint32_t endBlock = binFile.fileSize()/512 - 1;
    // find last used block.
    while (bgnBlock < endBlock) {
        uint32_t midBlock = (bgnBlock + endBlock + 1)/2;
        binFile.seekSet(512*midBlock);
        if (binFile.read(&count, 2) != 2) error("read");
        if (count == 0 || count > DATA_DIM) {
            endBlock = midBlock - 1;
        } else {
            bgnBlock = midBlock;
        }
    }
    // truncate after last used block.
    if (!binFile.truncate(512*(bgnBlock + 1))) {
        error("Truncate " TMP_FILE_NAME " failed");
    }
    renameBinFile();
}
//-----
-----
void renameBinFile() {
    while (sd.exists(binName)) {
        if (binName [BASE_NAME_SIZE + 1] != '9') {
            binName [BASE_NAME_SIZE + 1]++;
        } else {
            binName [BASE_NAME_SIZE + 1] = '0';
            if (binName [BASE_NAME_SIZE] == '9') {
                error("Can't create file name");
            }
            binName [BASE_NAME_SIZE]++;
        }
    }
}

```

```

    if (!binFile.rename(binName)) {
        error("Can't rename file");
    }
    Serial.print(F("File renamed: "));
    Serial.println(binName);
    Serial.print(F("File size: "));
    Serial.print(binFile.fileSize()/512);
    Serial.println(F(" blocks"));
}
//-----
void testSensor() {
    const uint32_t interval = 200000;
    int32_t diff;
    data_t data;
    Serial.println(F("\nTesting - type any character to stop\n"));
    // Wait for Serial Idle.
    delay(1000);
    printHeader(&Serial);
    uint32_t m = micros();
    while (!Serial.available()) {
        m += interval;
        do {
            diff = m - micros();
        } while (diff > 0);
        acquireData(&data);
        printData(&Serial, &data);
    }
}
//-----
void setup(void) {
    if (ERROR_LED_PIN >= 0) {
        pinMode(ERROR_LED_PIN, OUTPUT);
    }
    Serial.begin(9600);

    // Wait for USB Serial
    while (!Serial) {
        SysCall::yield();
    }
    Serial.print(F("\nFreeStack: "));
    Serial.println(FreeStack());
    Serial.print(F("Records/block: "));
    Serial.println(DATA_DIM);
    if (sizeof(block_t) != 512) {
        error("Invalid block size");
    }
    // Allow userSetup access to SPI bus.
    pinMode(SD_CS_PIN, OUTPUT);
    digitalWrite(SD_CS_PIN, HIGH);

    // Setup sensors.
    userSetup();
}

```

```

// Initialize at the highest speed supported by the board that is
// not over 50 MHz. Try a lower speed if SPI errors occur.
if (!sd.begin(SD_CS_PIN, SD_SCK_MHZ(50))) {
    sd.initErrorPrint(&Serial);
    fatalBlink();
}
// recover existing tmp file.
if (sd.exists(TMP_FILE_NAME)) {
    Serial.println(F("\nType 'Y' to recover existing tmp file "
TMP_FILE_NAME));
    while (!Serial.available()) {
        SysCall::yield();
    }
    if (Serial.read() == 'Y') {
        recoverTmpFile();
    } else {
        error("`Y' not typed, please manually delete " TMP_FILE_NAME);
    }
}
}
//-----
void loop(void) {
    // Read any Serial data.
    do {
        delay(10);
    } while (Serial.available() && Serial.read() >= 0);
    Serial.println();
    Serial.println(F("type:"));
    Serial.println(F("b - open existing bin file"));
    Serial.println(F("c - convert file to csv"));
    Serial.println(F("d - dump data to Serial"));
    Serial.println(F("e - overrun error details"));
    Serial.println(F("l - list files"));
    Serial.println(F("r - record data"));
    Serial.println(F("t - test without logging"));
    while (!Serial.available()) {
        SysCall::yield();
    }
    #if WDT_YIELD_TIME_MICROS
        Serial.println(F("LowLatencyLogger can not run with watchdog timer"));
        SysCall::halt();
    #endif

    char c = tolower(Serial.read());

    // Discard extra Serial data.
    do {
        delay(10);
    } while (Serial.available() && Serial.read() >= 0);

    if (ERROR_LED_PIN >= 0) {
        digitalWrite(ERROR_LED_PIN, LOW);
    }
    if (c == 'b') {

```

```

    openBinFile();
} else if (c == 'c') {
    binaryToCsv();
} else if (c == 'd') {
    dumpData();
} else if (c == 'e') {
    checkOverrun();
} else if (c == 'l') {
    Serial.println(F("\nls:"));
    sd.ls(&Serial, LS_SIZE);
} else if (c == 'r') {
    logData();
} else if (c == 't') {
    testSensor();
} else {
    Serial.println(F("Invalid entry"));
}
}" Adapted from [29] and [23].

```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [1] Rockets Galore. *The Economist*, Sep 29, 2012.
- [2] Cevik, P., Kocaman, I., Akgul, A., and Akca, B. “The Small and Silent Force Multiplier: A Swarm UAV—Electronic Attack.” *Journal of Intelligent & Robotic Systems*, Vol. 70, 2013, pp. 595–608. <https://doi.org/10.1007/s10846-012-9698-1>.
- [3] Hambling, D. “What Are Drone Swarms and Why Does Every Military Suddenly Want One?” *Forbes*, 2021.
- [4] Pledger, T. “The Role of Drones in Future Terrorist Attacks.” *Association of the United States Army*.
- [5] Joshi, S. “‘Drone Swarms’ Are Coming, and They Are the Future of Wars in the Air.” *The Print*, 2021.
- [6] Rydalch, F. *Missile Demonstrator for Counter UAV Applications*. Naval Postgraduate School, 2016.
- [7] Busta, M. *Design and Development of an Adaptable Flight Demonstration Vehicle and Modular Guidance and Control*. Naval Postgraduate School, 2019.
- [8] Lobo, K. *Submunition Design for a Low-Cost Small UAS Counter-Swarm Missile*. Naval Postgraduate School, 2018.
- [9] Thyberg, R. *Design and Testing of a Multi-Unit Payload Delivery and Tracking System for Guided Munitions to Combat UAV Swarm Threats*. Naval Postgraduate School, 2019.
- [10] Ball, R. *The Fundamentals of Aircraft Combat Survivability: Analysis and Design*. American Institute of Aeronautics and Astronautics, Inc., Reston, Virginia, 2003.
- [11] Electrical Fault Tree. *Visual Paradigm*. <https://online.visual-paradigm.com/app/diagrams/#diagram:proj=0&type=FaultTreeAnalysis&gallery=/repository/3f11f402-5763-4376-a7a4-9aaef279ce3.xml&name=Electrical%20Fault%20Tree%20Analysis>.
- [12] Samarrai, F. “UVA ‘Cubesat’ Deployed; Students Seek Contact from Ground Station.” *UVA Today*, 2019.
- [13] Kramer, A. A CAD Model of the UWE-4 CubeSat.

- [14] M80-4C11205F1-04-325-00-000. *Digi-Key*.
<https://www.digikey.com/en/products/detail/harwin-inc/M80-4C11205F1-04-325-00-000/2264251?s=N4IgTCBcDaILIA4AMBaALAYQIxbEgrAGJYpJooDMY%2BpqS9IAugL5A>.
- [15] M80-5T11205M1-04-331-00-000. *Digi-Key*.
<https://www.digikey.com/en/products/detail/harwin-inc/M80-5T11205M1-04-331-00-000/2264252?s=N4IgTCBcDaILIA4AMBaArAFQIxbEtcWKSALCgMzlFKo1IgC6AvkA>.
- [16] Raspberry Pi 3 Prototyping Board Proto Breadboard HAT by MakerSpot Soldered with Female Pin Connector PCB for Raspberry Pi 2 Pi 3 A+ B+ (1 Pack). *Amazon*. <https://www.amazon.com/Raspberry-Prototyping-Breadboard-MakerSpot-Connector/dp/B01MY8TYSD>.
- [17] MS3112E10-6S. *Newark*. <https://www.newark.com/amphenol-industrial/ms3112e10-6s/circular-connector-receptacle/dp/85C4066?ost=71-300210-06s>.
- [18] MS3116F10-6P. *Newark*. <https://www.newark.com/amphenol-industrial/ms3116f10-6p/circular-connector-plug-size-10/dp/85C4087>.
- [19] Schoenster, J. Measurements and Analysis of Solid-Propellant-Rocket Vibrations Obtained During a Captive Flight.
<https://ntrs.nasa.gov/api/citations/19710028063/downloads/19710028063.pdf>.
- [20] Pauly, W. FFT Relationship Between Sampling Rate, Bandwidth, Resolution, Etc:
<http://www.vibrationworld.com/AppNotes/Sampling.html>.
- [21] Townsend, K. Adafruit BNO055 Absolute Orientation Sensor. *Adafruit*.
<https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor>.
- [22] SparkFun Electronics. 1.0.0. *SparkFun ADXL345 Arduino Library*.
- [23] Greiman, B. 1.2.4. *SdFat - Adafruit Fork*. Adafruit.
- [24] TelemetryPro Receiver. *Multitronix*. <https://www.multitronix.com/receiver.html>.
- [25] General Environmental Verification Standard (GEVS) for GSFC Flight Programs and Projects. <https://standards.nasa.gov/standard/gsfsc/gsfsc-std-7000>.
- [26] Air-Cooled Shakers. http://www.dataphysics.com/downloads/literature/LE-1216M_LE-1216_LE-1224-1.pdf.
- [27] Sakota, D. Vibration and TVAC Procedure for Firefly Orbital-1 CubeSat.

- [28] EXA TITAN-1 350Whr High Energy Density Battery Matrix. *CubeSat Shop*.
<https://www.cubesatshop.com/product/titan-1-high-energy-density-battery-matrix/>.
- [29] Adafruit. 1.9.3. *BusIO*. Adafruit.
- [30] Adafruit. *RadioHead-Master*. Adafruit.
- [31] Adafruit. 1.2.0. *MPRLS*. Adafruit.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California