



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

X-BAND SOFTWARE-DEFINED RADIO FOR CUBESAT APPLICATIONS

by

Greg Bischoff

September 2019

Thesis Advisor:

Co-Advisor:

James H. Newman

Giovanni Minelli

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 September 2019	3. REPORT TYPE AND DATES COVERED Master's thesis		
4. TITLE AND SUBTITLE X-BAND SOFTWARE-DEFINED RADIO FOR CUBESAT APPLICATIONS			5. FUNDING NUMBERS	
6. AUTHOR(S) Greg Bischoff				
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) DoD Space			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) Software-defined radios (SDRs) are used in many communications applications and offer flexibility in designing a communication network by offering rapid configuration changes after the radio is already produced. This is of interest in the small satellite industry, where small, lightweight, low-cost, and programmable radios will be needed for a wide array of upcoming missions of increasing complexity. Previous Naval Postgraduate School (NPS) research has produced a C-band SDR payload in a 1U CubeSat form factor. The X-band of the radio frequency spectrum offers more bandwidth, and opens the possibility of higher data rate transmissions. The objective for this project was to build an X-band SDR that can be used in future NPS research and flown on multiple platforms, including high altitude balloons, high altitude rockets, and CubeSats. This project built on previous C-band SDR research by converting the transmissions to X-band, expanding the over-air software protocol, implementing a new waveform, increasing the over-air data rate, and assessing the overall performance of the system.				
14. SUBJECT TERMS X-band, SDR, software-defined radio, CubeSat			15. NUMBER OF PAGES 149	
			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.

X-BAND SOFTWARE-DEFINED RADIO FOR CUBESAT APPLICATIONS

Greg Bischoff
Lieutenant Commander, United States Navy
BS, Purdue University, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SPACE SYSTEMS OPERATIONS

from the

**NAVAL POSTGRADUATE SCHOOL
September 2019**

Approved by: James H. Newman
Advisor

Giovanni Minelli
Co-Advisor

James H. Newman
Chair, Department of Space Systems Academic Group

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Software-defined radios (SDRs) are used in many communications applications and offer flexibility in designing a communication network by offering rapid configuration changes after the radio is already produced. This is of interest in the small satellite industry, where small, lightweight, low-cost, and programmable radios will be needed for a wide array of upcoming missions of increasing complexity. Previous Naval Postgraduate School (NPS) research has produced a C-band SDR payload in a 1U CubeSat form factor. The X-band of the radio frequency spectrum offers more bandwidth, and opens the possibility of higher data rate transmissions. The objective for this project was to build an X-band SDR that can be used in future NPS research and flown on multiple platforms, including high altitude balloons, high altitude rockets, and CubeSats. This project built on previous C-band SDR research by converting the transmissions to X-band, expanding the over-air software protocol, implementing a new waveform, increasing the over-air data rate, and assessing the overall performance of the system.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	CUBESAT.....	1
B.	MOBILE CUBESAT COMMAND AND CONTROL	3
C.	SOFTWARE-DEFINED RADIO	4
D.	RADIO FREQUENCY SELECTION	4
E.	X-BAND RADIO CONCEPT	6
F.	THESIS ORGANIZATION.....	6
II.	BACKGROUND	9
A.	DIGITAL COMMUNICATIONS	9
B.	FORWARD ERROR CORRECTION	12
C.	SOFTWARE-DEFINED RADIO	13
D.	LINK BUDGET	13
E.	EXISTING CUBESAT X-BAND TRANSMITTERS	15
F.	PREVIOUS NPS RESEARCH	16
III.	HARDWARE	19
A.	REQUIREMENTS.....	19
B.	HARDWARE USED.....	19
1.	Payload: X-Band Radio.....	19
2.	Bus	37
IV.	SOFTWARE.....	39
A.	REQUIREMENTS.....	39
B.	SOFTWARE DESIGN	39
1.	Payload Software	39
2.	Bus Software.....	47
V.	RESULTS	49
A.	COMPONENT TESTING	49
1.	ADF 4356	49
2.	Ettus B205mini-i.....	51
3.	Raspberry Pi.....	52
B.	PAYLOAD BENCH TESTING.....	52
1.	Payload Sub-Assembly Testing	52
2.	Payload Integration Testing.....	55
C.	FIELD TESTING.....	65

1.	Outdoor Testing	65
2.	Flight Test	66
VI.	CONCLUSIONS	85
A.	CONCLUSIONS	85
B.	FUTURE WORK	86
1.	Hardware Improvements	86
2.	Software Improvements	88
3.	Additional Testing	89
	APPENDIX A. LINK BUDGETS	91
	APPENDIX B. ADF-4356 REGISTER VALUES	93
	APPENDIX C. START_PAYLOAD.PY	95
	APPENDIX D. ADF4356.PY	97
	APPENDIX E. 6300MHZ_M4DBM.TXT	99
	APPENDIX F. XBAND_CAMERA.PY	101
	APPENDIX G. XBAND_TX.PY	103
	APPENDIX H. FIXEDLEN_PACKET_SYNCHRONIZER.PY	109
	APPENDIX F. CHUNKING.PY	111
	LIST OF REFERENCES	125
	INITIAL DISTRIBUTION LIST	129

LIST OF FIGURES

Figure 1.	Binary data modulated with (a) ASK; (b) FSK; (c) PSK. Source: [11].	10
Figure 2.	Constellation diagram of QPSK. Source: [11].	11
Figure 3.	Occupied bandwidth of a 500 kbps signal at 8250 MHz	12
Figure 4.	Block diagram of X-band SDR payload configured for 10 GHz	20
Figure 5.	X-band payload equipment shelf A, interior side	22
Figure 6.	X-band payload equipment shelf A, exterior side	22
Figure 7.	X-band payload equipment shelf B, interior side	23
Figure 8.	X-band payload equipment shelf B, exterior side with cover removed	23
Figure 9.	Ettus B200 enclosure compared with a U.S. quarter. Source: [23].	24
Figure 10.	Raspberry Pi 4 model B	25
Figure 11.	Raspberry Pi 3 model A+	26
Figure 12.	Power and data distribution panel (front)	27
Figure 13.	Power and data distribution panel (back)	27
Figure 14.	Schematic of power and data distribution panel	28
Figure 15.	Image of Analog Devices ADF-4356 evaluation board. Source: [26].	29
Figure 16.	Screen capture of ADF-4356 control software	30
Figure 17.	Frequency mixer: Mini-Circuits ZX05-153-S+	31
Figure 18.	Intermediate frequency and local oscillator amplifiers: Mini-Circuits ZX60-83LN-S+	32
Figure 19.	Final amplifier: Mini-Circuits ZX60-183A-S+	33
Figure 20.	10 GHz final bandpass filter: Pasternack PE87LF1000	33
Figure 21.	8 GHz final bandpass filter: Mini-Circuits ZVBP-8250-S+	34
Figure 22.	Local oscillator filter: Mini-Circuits VBFZ-6260-S+	34

Figure 23.	Local oscillator filter: Mini-Circuits ZVBP-5800-S+	35
Figure 24.	Helical antennas	37
Figure 25.	GNU Radio flowgraph for QPSK transmitter.....	43
Figure 26.	GNU Radio flowgraph for QPSK receiver	44
Figure 27.	Local oscillator block diagram.....	51
Figure 28.	Initial HAB layout for the X-band payload	56
Figure 29.	Assembled X-band HAB bus (view 1)	56
Figure 30.	Assembled X-band HAB bus (view 2)	56
Figure 31.	Ground station down conversion hardware	58
Figure 32.	Over air transmission packet error rates with additional attenuation	64
Figure 33.	Google Earth image of the HAB launch site near Chualar, CA. Source: [41].....	67
Figure 34.	Planned HAB flight profile. Source: [40].	69
Figure 35.	HAB flight profile without balloon release. Source: [40].....	70
Figure 36.	Google Earth image of HAB recovery site. Source: [41].	72
Figure 37.	Photograph of the HAB payload at the landing site	73
Figure 38.	Balloon harness line and actuator sleeve caught on CubeSat frame.....	74
Figure 39.	Recovered HAB payload	75
Figure 40.	Recovered HAB payload on desk	75
Figure 41.	HAB GPS data with launch and recovery sites in Google Earth. Source: [41].....	76
Figure 42.	HAB GPS data over Salinas Valley. Source: [41].	77
Figure 43.	Top down view of HAB GPS data. Source: [41].	77
Figure 44.	GPS altitude versus GPS time for HAB flight.....	78
Figure 45.	First picture captured after launch (#17).....	79

Figure 46.	Last picture prior to entering cloud layer (#19)	80
Figure 47.	Picture taken exiting cloud layer (#20)	81
Figure 48.	Final complete picture taken before power down (#180)	82

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	MC3 frequency ranges. Adapted from [7].	3
Table 2.	Examples of commercially available X-band transmitters for CubeSats	16
Table 3.	X-band SDR requirements	19
Table 4.	Components used to build the X-band payload	21
Table 5.	Selected ADF-4356 local oscillator output parameter values	29
Table 6.	RF mixer input and output parameters	31
Table 7.	Input and output values for LO and IF amplifiers	32
Table 8.	Helical antenna design parameters	37
Table 9.	Transmission data frame	40
Table 10.	Xband_tx.py input parameters and default values	41
Table 11.	Xband_rx.py input parameters and default values	42
Table 12.	Local oscillator frequency output	50
Table 13.	Local oscillator power output	50
Table 14.	Local oscillator power to mixer	50
Table 15.	Power output measurement test results for the Ettus B205mini-i	51
Table 16.	Maximum data transmission rate when running xband_tx.py	52
Table 17.	Payload power consumption test results	53
Table 18.	Payload power to antenna at 10.25 GHz	54
Table 19.	Payload power to the antenna at 8.25 GHz	54
Table 20.	Packet transfer error rate at 3.95 GHz	60
Table 21.	Time to download files at 3.95 GHz	60
Table 22.	Packet transfer error rate at 10.25GHz with direct cable connection w/ 10 dB of attenuation	61

Table 23.	Packet transfer error rates at various attenuations for 10.25 GHz (low gain)	61
Table 24.	Packet transfer error rates at various attenuations for 10.25 GHz (high gain)	62
Table 25.	Equivalent free space path loss for cable connected transmitter and receiver.....	62
Table 26.	Packet transfer error rate at 10.25 GHz over-the-air at various attenuations	63
Table 27.	Equivalent free space path loss for over-the-air lab test	64
Table 28.	Packet transfer error rate for outdoor testing	65
Table 29.	X-band CubeSat transmitter requirements and results.....	83

LIST OF ACRONYMS AND ABBREVIATIONS

AM	amplitude modulation
ASCII	American standard code for information interchange
ASK	amplitude shift keying
BER	bit error rate
C&DH	command and data handling
COTS	commercial-off-the-shelf
CRC	cyclic redundancy check
dB	decibel
DoD	Department of Defense
DSP	digital signal processing
EPS	electrical power system
FAA	Federal Aviation Administration
FEC	forward error correction
FIFO	first in first out
FM	frequency modulation
FPGA	field programmable gate array
FSK	frequency shift keying
FSPL	free space path loss
GHz	gigahertz
GPIO	general purpose input/output
GRC	GNU Radio companion
HAB	high-altitude balloon
IF	intermediate frequency
I/O	input/output
JPEG	Joint Photographic Experts Group
Kbps	kilobit per second
LO	local oscillator
Mbps	megabit per second
MC3	Mobile CubeSat Command and Control
MHz	megahertz

NTIA	National Telecommunications and Information Administration
OQPSK	offset quadrature phase shift keying
PDDP	power and data distribution panel
PM	phase modulation
PSK	phase shift keying
QPSK	quadrature phase shift keying
R&D	research and development
RF	radio frequency
SDR	software-defined radio
SNR	signal to noise ratio
RS	Reed-Solomon
SSAG	Space Systems Academic Group
SWaP	size, weight, and power
UART	universal asynchronous receiver/transmitter
UDP	user datagram protocol
UHF	ultra-high frequency
USB	universal serial bus
VSWR	voltage standing wave ratio

ACKNOWLEDGMENTS

It is with great appreciation that I would like to thank those that supported my thesis and the X-band CubeSat radio project. First, I would like to thank the entire SSAG small satellite laboratory staff. Without your help, this project would have never flowed. I also thank David for help with the RF theory, HAB flight operations, and set up of multiple experiments, and Alex for his answers to the constant stream of Linux and Raspberry Pi questions, HAB flight prep, and assistance with the soldering component. I also want to acknowledge Jim for all of the computer coding and support software; Ron for help with wiring harnesses, power systems, and the last-minute scrambling to get all of the systems working; and Dan for the numerous 3-D printed parts. My gratitude extends to Mike for the education in SDR theory, and more importantly, for guidance getting the GNU Radio code to work. I am also grateful to my thesis advisors, Dr. Newman and Dr. Minelli, for their guidance, patience, and support as I worked on this project. Finally, I would like to thank Lisa for being a friend and lending an ear while I vented my frustrations while completing this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

This project was undertaken to develop and demonstrate operation of a low-cost X-band radio for data downlink from a CubeSat. Current commercially available X-band transmitters for a CubeSat form factor are costly, at a minimum of \$25,000 each [1]. Additionally, when purchasing a prebuilt transmitter, the operator gives up some control of the radio signal characteristics and protocols to the point that the radio may essentially become a black box that requires vendor support in order to make changes or get updates. The objective of this project is to create and demonstrate operation of an X-band transmitter that will fit inside a 10 cm x 10 cm x 10 cm volume, otherwise known as one standard CubeSat unit or 1 U; have a mass less than 1.3 kg, the standard mass limit for 1U; transmit data rates of at least 1 Mbps; and cost less than commercially available transmitters. In an effort to keep the cost of the radio low, the development of the X-band transmitter will be accomplished through the use of commercial-off-the-shelf (COTS) components for hardware and open-source software for signal processing. This radio is designed for follow on use in future Naval Postgraduate School (NPS) and Mobile CubeSat Command and Control (MC3) missions.

A. CUBESAT

CubeSats are a class of small satellites that have been used by academic institutions and hobbyists for educational and experimental purposes. CubeSats have become increasingly popular for commercial applications such as Planet Lab's PlanetScope satellite imagery satellites [2] and could prove to be a valuable gap filler in the Department of Defense's (DoD) satellite inventory. As of April 2018, over 1000 CubeSats have been launched with almost 700 still in orbit [3]. CubeSats are particularly useful due to their relatively low cost and the speed at which they can be constructed and launched. Since CubeSats have a shorter anticipated lifetime than traditional large satellites, they would need less rigorous testing and specifications; this will yield savings compared to traditional satellite acquisition programs. In the face of a contested space environment, with traditional large satellites threatened by anti-satellite systems [4], the use of CubeSat systems could

provide the DoD with a relatively rapid, stopgap measure to partially replace the loss of other space-based capabilities. However, if the DoD plans to use CubeSats as a stopgap measure, then the development of CubeSat capabilities must not be neglected until the moment they are needed. The various anticipated components must be acquired and tested; they must have suppliers, vendors, or components ready to be built up when needed.

The CubeSat standard was created as a joint effort between California Polytechnic State University and Stanford University in 1999 and has subsequently served as a standard for satellite programs at all levels of education [5]. The CubeSat standard is based on 10cm x 10cm x 10cm cube shaped volume [5]. A cube of this size is designated as 1U; the typical mass of 1U ranges from 1 kg to 1.3 kg [5]. Since the development of the CubeSat standard, sizes larger than 1U have become popular. Sizes 1.5U, 2U, 3U, and 6U are among the most common [5]. This standard has also provided a platform for more than exclusively educational satellites, Planet Labs PlanetScope satellite is an example of a commercial satellite built with a 3U CubeSat form factor [2].

CubeSats are typically put into orbit as a secondary payload on other launch missions, where a large launch vehicle has spare performance margin [5]. This provides a cost effective launch solution, since the institution that puts the CubeSat into orbit does not have to pay for the full launch vehicle itself. The limitation for this is that, as a secondary payload, the CubeSats and their launcher are often placed at the base of the final stage of the launch vehicle [6], which results in the CubeSats experiencing higher vibration and acoustic loads.

Traditionally, CubeSat designs have been designed and operated by educational institutions and hobbyists who did not have a high demand for high-speed data downlinks. For the CubeSat platform to be useful to the DoD, improving over-the-air data rates enables their use in missions of ever-increasing complexity. As commercial organizations enter the CubeSat sector, there will be research and development (R&D) into more capable systems and components. However, the DoD should not wait for this R&D: it should instead look to procure or develop those capabilities that will suit its needs. This project attempts to provide the capability for that higher data rate in CubeSat form factor.

The DoD is pursuing CubeSats for future use in cases where a rapid, short-term, spaces based capability is needed but the DoD has high data-rate requirements and, in order to use CubeSats with high data-rate capabilities, a ground station network must exist that has similar data-rate capabilities. This thesis is a means of developing a low-cost downlink for testing new ground station data rate capabilities from platforms such as balloons, rockets, and CubeSats before high-value missions demand the increase in performance from spacecraft and their ground stations.

B. MOBILE CUBESAT COMMAND AND CONTROL

The NPS Space Systems Academic Group (SSAG) operates the MC3 network for the DoD. MC3 is a DoD-led ground station network built with “common-use infrastructure [to support] communications and mission operations of small satellites for a wide range of U.S. government organizations, contractors, universities, and foreign partners” [7]. The network is a collection of ground stations at participating DoD academic institutions, and provides access to those organization’s satellites via Internet connection [7]. The MC3 network is the ground network that will be used to control and communicate with future NPS SSAG satellites. Currently, the MC3 network operates in the UHF and S-band frequencies and plans to expand into the X-band. Table 1 shows the frequency ranges that MC3 operates in.

Table 1. MC3 frequency ranges. Adapted from [7].

Band	Frequencies	Designator
UHF uplink	449.75-450.25 MHz	12K5F1D 43K0F1D
UHF downlink	902-928 MHz	115KG1D
S-band uplink	2025-2110 MHz	2M00G2D 2M45G1D
S-band downlink	2200-2290 MHz	1M60G1D 2M00G2D 2M45G1D
X-band uplink	7190-7250 MHz	(future)
X-band downlink	8025-8400 MHz	(future)

C. SOFTWARE-DEFINED RADIO

Software-defined radios (SDR) are a type of radio transceiver that has its parameters of operation set through software instead of being defined by its hardware. Using an SDR allows modification of the physical layer properties of a radio to be modified through software [8]. This allows the parameter of the radio signal to be modified with a software update instead of a hardware change. With an SDR, a space vehicle can have many more applications/missions with the same hardware, and those missions can be changed with a software update. Hardware changes in orbit are impractical for space vehicles, and while it is still possible to make hardware changes on orbit, it is unreasonable in all but the most extreme cases (e.g., Hubble). Also, such changes will be exceedingly expensive. Hardware can be upgraded by continuously flying new upgraded CubeSats, but the turnaround time is still on the order of many months or years. A software update to change a spacecraft's radio settings while in orbit may be precarious for multiple reasons. There is a chance of misconfiguring the radio, which can lead to degradation, downtime or even loss of the mission. Software updates should therefore be handled with caution and lots of testing will be required in order to reduce risk. The use of an SDR will also bring tradeoffs that must be considered. With an SDR, some controlling computer must be included, which will add cost, weight, power consumption, and complexity.

Using an SDR gives the satellite operator the flexibility to easily change radio parameters in testing. The possibility of changing parameters in orbit is desirable for this CubeSat radio project. This will allow future for studies and development iterations to improve the operation of this radio without the purchase and integration of new hardware.

D. RADIO FREQUENCY SELECTION

The need for higher data rates necessitates the use of a higher carrier frequency. Bandwidth is directly related to data rate, therefore, in order to achieve higher data rates more bandwidth must be used; larger bandwidths require higher carrier frequencies. These relationships between data rate, bandwidth and carrier frequency are discussed in Chapter II. The higher frequency of X-band over S-Band and UHF provides more bandwidth, which

in turn allows higher data rates and a less crowded spectrum. It is for these reasons that the X-band is chosen for the radio's operation.

The use of radio frequency, both in space and on earth, is coordinated and controlled by various organizations. Internationally, the ITU allocates radio bands along with frequency assignments for orbital positions in geosynchronous orbit to prevent interference of radios in different countries [9]. Internal to the United States, the use of radio frequency is regulated by the Department of Commerce and rules for the use of frequency is published by the National Telecommunications and Information Administration (NTIA) in the *Manual of Regulations and Procedures for Federal Radio Frequency Management* [10]. Satellites must take into account the rules for radio transmissions for all countries they would operate over. The study for this thesis will take place on a high altitude balloon (HAB) and will remain within the continental U.S. Therefore, it will only need to comply with NTIA regulations.

Initially this study focused on the lower end of the X-band (8 GHz – 12 GHz) of the radio frequency spectrum. The 8025–8400 MHz bands were selected by the project sponsor. These three bands are allocated for space-to-earth transmissions on both the international tables and U.S. tables [10]. These frequencies are allocated on a primary basis for non-federal use. In the United States, fixed-satellite and mobile-satellite use is limited to military systems [10]. The NTIA manual further specifies that in the 8025–8215 MHz band,

Although the military space radio communication systems, which include earth stations near the proposed meteorological-satellite installations will precede the meteorological-satellite installations, engineering adjustments to either the military or the meteorological-satellite systems or both will be made as mutually required to assure compatible operations of the systems concerned. [10]

Internationally geostationary satellites using the 8025–8400 MHz band must limit their power flux-density based on angles of arrival to the ground. Since this study's planned satellite (low earth orbit) and proposed HAB test flight are not geosynchronous (GEO) satellites this provision did not apply.

At the time of the flight test of this project, clearance for the use of RF spectrum in the 8025–8400 MHz range had not been granted; therefore, the amateur band of the X-band (10,000 – 10,500 MHz) was used instead.

E. X-BAND RADIO CONCEPT

CubeSats have a very limited budget for size, weight, and power (SWaP). Historically, X-band transmitters have been too power hungry for use on such a small platform. However, advances in hardware and software have allowed the creation of CubeSat form factor X-band radios. While some commercially developed X-band transmitters are available, these were too high in cost to be of use in this study. Therefore, an X-band transmitter was created from commercially available parts at a dramatically lower cost than the price of a ready-made transmitter.

The available SDRs considered for this study have an upper limit of 6 GHz and do not reach into the X-band. Alternative X-band SDRs were significantly more expensive; therefore, a lower cost SDR that does not operate in the X-band was selected. In this case, the Ettus B-205mini-i is used. To make use of a lower frequency SDR, a mixer and local oscillator must be used to step up the SDR output signal to the desired transmission frequency. This setup required that additional RF hardware be used to get the desired output transmission. Particularly, an RF mixer, a frequency synthesizer, and amplifiers would be needed. The SDR is controlled by a single circuit-board computer, in this case a Raspberry Pi 4. This computer will not only control the SDR, but also initialize the frequency synthesizer, interface with the HAB bus and capture, store, and process data for transmission.

F. THESIS ORGANIZATION

This thesis will be organized into six chapters plus additional appendices. Chapter II covers the background material for the design of this project. Chapter III is a description of the hardware components used in this project, along with details on how it is assembled. Chapter IV describes the software used and programming that was created for this project. Chapter V is a description of testing conducted and the results of those tests. This includes

developmental tests, component testing, assembly testing, and a HAB flight test. Chapter VI contains the conclusions of this project and future work to be completed.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

Previous CubeSats have primarily transmit in the S- and C-bands of the radio spectrum. NPS/MC3 currently operate three CubeSats in orbit which use the S-band and UHF frequencies for their data links. This thesis will continue previous SSAG work to create and demonstrate operation of a low-cost radio for data downlink from a CubeSat that will operate in the X-band and provide at least a 1 Mbps data transfer rate.

A. DIGITAL COMMUNICATIONS

Digital communication is the process of transmitting information from a source to a destination after the message has been sampled and encoded as a series of discrete symbols [11]. For the use of computers these symbols are often 0 and 1, i.e., binary. For transmission in the electromagnetic spectrum, these binary symbols need to be converted to an RF signal, called modulation [11]. Reversing the process the receiver will demodulate the signal in order to return it to the original binary data.

There are multiple ways to modulate data on an RF signal. The major categories are frequency modulation (FM), amplitude modulation (AM), and phase modulation (PM) [11]. In each of these cases, the parameter that is modulated can be shifted or changed to a new value. This shift is used to indicate a 1 or a 0 of the data that is being transmitted. These modulation techniques are called shift keying. Figure 1 shows an example of modulating binary data for amplitude shift keying (ASK), frequency shift keying (FSK), and phase shift keying (PSK). Each technique has its own advantages and disadvantages, all of which requires tradeoffs in system design. PSK techniques have become the standard for modern satellite communications due to its advantages in both bit energy (E_b) and spectral efficiency (bps/Hz) [12].

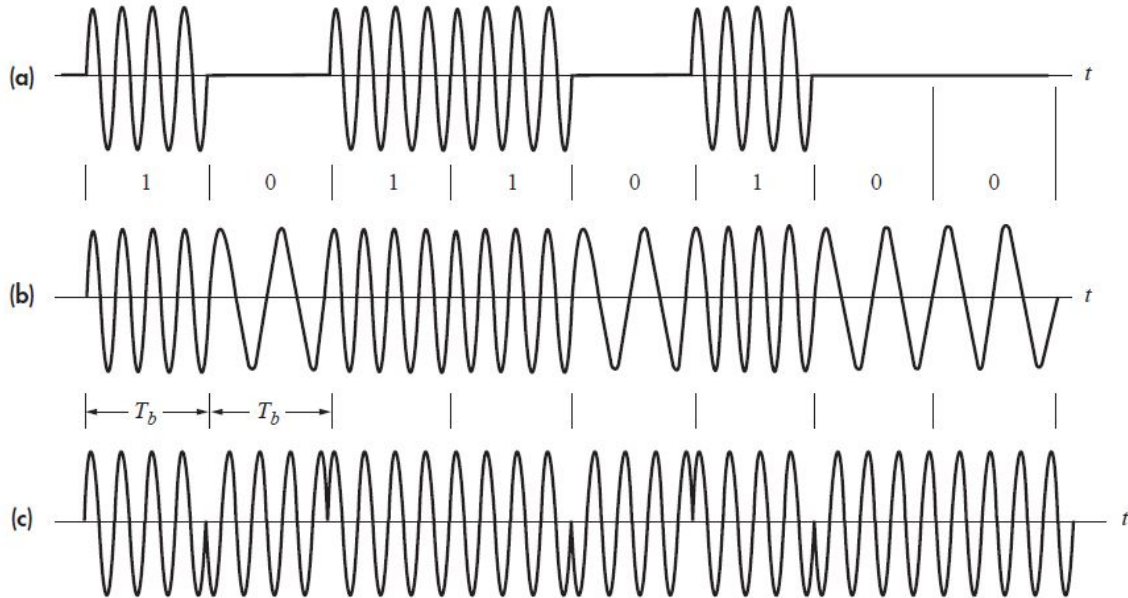


Figure 1. Binary data modulated with (a) ASK; (b) FSK; (c) PSK.
Source: [11].

The change in the variation parameter can have multiple levels, allowing more than one symbol to be encoded at a time. The phase of a signal can vary by up to 2π . When modulating information the selected levels are spaced as widely as possible so that the receiver has the best chance of distinguishing the difference in the signal. If two symbols are used in modulation, then the modulation levels will be separated by π radians; if four symbols are encoded then the modulation levels are separated by $\pi/2$ radians. For example, quadrature phase shift keying uses four different phase shift values to indicate four digital values (00, 01, 10, 11). This is depicted in Figure 2 [11].

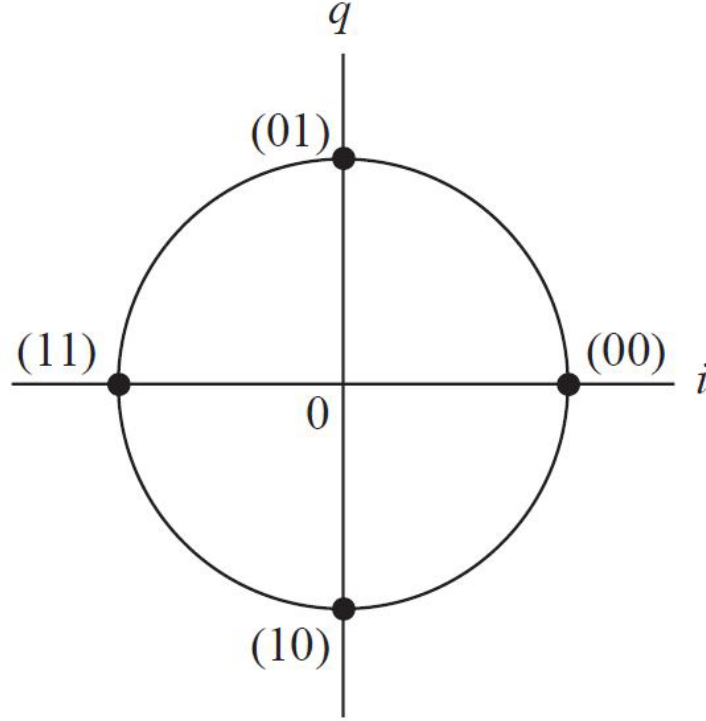


Figure 2. Constellation diagram of QPSK. Source: [11].

The phenomenon of reciprocal spreading is that as the pulse duration (τ) decreases the frequency width increases [11]. This is because the fast variations in time of the transmitted signal require higher frequency components [11]. In order to achieve higher data transmission rates, the data bits or symbols must be smaller and have shorter intervals between them, thus yielding a larger frequency width. Therefore, when modulating data on an RF signal the bandwidth used will be at least the data rate as shown in Equation (1) [11]. This can be seen in the signal measured for a 500 kbps signal modulated at 8250 MHz in Figure 3.

$$B \geq \frac{1}{\tau} \quad (1)$$

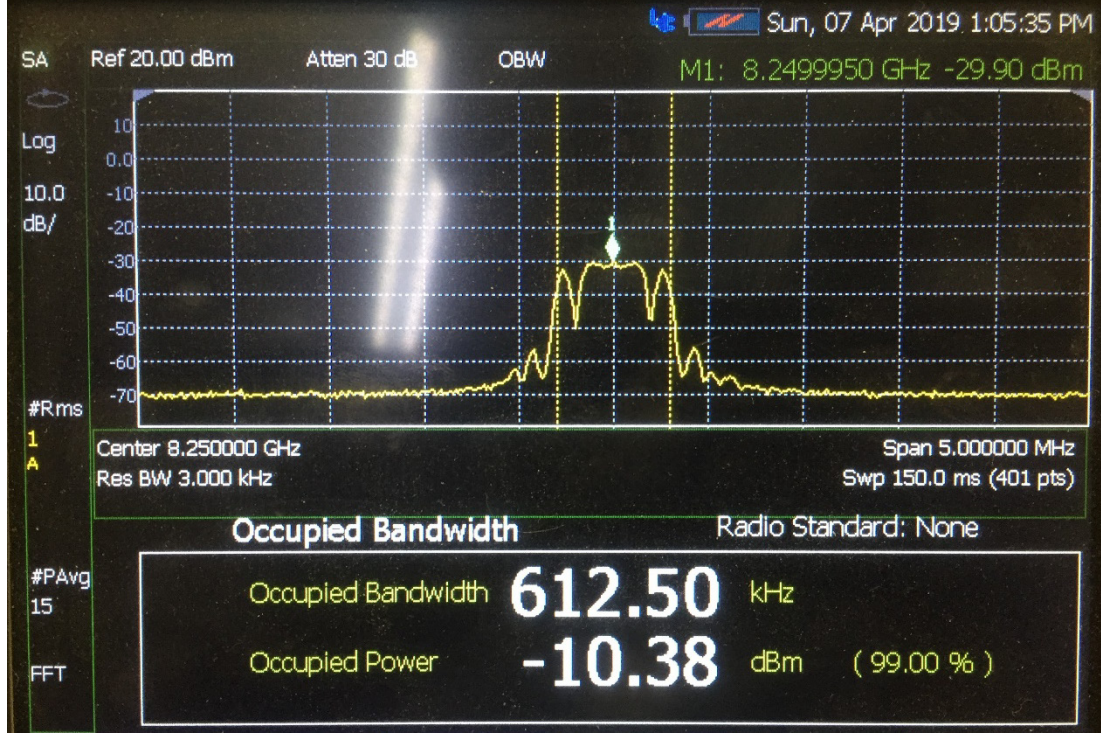


Figure 3. Occupied bandwidth of a 500 kbps signal at 8250 MHz

Desired bandwidth must be considered when selecting a carrier or center frequency. RF components, including antennas, amplifiers, and bandpass filters, require the bandwidth to be relatively small compared to the carrier frequency in order to produce a low distortion signal [11]. A rule of thumb for the ratio of bandwidth (B) and carrier frequency is given by Equation (2) [11]. This sets a practical maximum bandwidth and therefore data transmission rate for a given carrier frequency.

$$0.01 < \frac{B}{f_c} < 0.1 \quad (2)$$

B. FORWARD ERROR CORRECTION

Forward error correction (FEC) is a process by which redundant bits are added to a digital message in anticipation of errors occurring during transmission. These redundant bits are used by the receiver to detect and, if possible, correct bit errors in the received signal. There are two major categories of FEC, convolutional and block codes [13]. FEC has an associated code rate (ρ) which is the ratio of the number of bits containing

information (n) to the total bits sent, information bits (n) plus redundant bits (r). The formula for code rate is shown in Equation (3) [12]. As the code rate gets smaller more errors can be caught and corrected, but this comes with the tradeoff of slower data transfer. The selected code rate must be chosen with the expected SNR and desired data rate in mind.

$$\rho = n/(n+r) \quad (3)$$

C. SOFTWARE-DEFINED RADIO

Tuttlebee [14] describes an SDR as a radio where “the digitization is performed at some stage downstream from the antenna.” Typically, this is after filtering, amplification, and down conversion of the received RF [14]. The Wireless Innovation Forum defines an SDR as a radio where part or all of the physical radio layer functions are performed by software [15]. According to Wyglinski and Pu [8], “until the 1980s, almost all wireless communication systems were based on relatively static, hard-wired platforms using technologies such as application-specific integrated circuits (ASICs).” With advancements in technologies of processors and FPGAs in the recent decades, many of the functions performed by hardware in old radios can now be performed by software [15]. This combined with the reduction in price of those technologies has led to a growth of SDRs in multiple sectors including military, research, commercial, and amateur systems [15]. Of particular interest in this project is the setting and modification of the radio’s frequency, bandwidth, modulation, and data rate.

D. LINK BUDGET

When a signal is transmitted from a sender to a receiver, there must be a minimum amount of power received for the signal to be recoverable and decipherable. Transmitted power, distance, antennas, amplifiers, and environmental effects are all factors in determining the power received. Along with the transmitted signal, there is noise that the receiver will pick up. This noise is a combination of background noise, noise from system components, and other signals. The ratio of the signal strength to the noise level is called signal-to-noise ratio (SNR), also represented by S/N ; it is a key metric that is used to

determine if a signal can be received. Additionally, data rate, modulation type, and error correction also factor into the determination of whether a signal can be decoded [16]. In digital communications, the ratio of energy per bit (E_b) and noise density (N_0) is often used instead of S/N. This ratio, $\frac{E_b}{N_0}$, is often used to predict the bit error rate (BER). All of these factors are typically expressed in decibels (dB) and are either added or subtracted depending on whether they increase or decrease the signal received. A basic link equation is given by Equation (4), where P_r is the received power, P_t is the transmitted power, G_t is the gain of the transmit antenna, G_r is the gain of the receive antenna, and L is the total losses including transmitter, receiver, free space dispersion, and atmospheric losses [12].

$$P_{Rx} = P_{Tx} + G_{Tx} + G_{Rx} - L_{Tx+Rx} \quad (4)$$

The gain of an antenna depends on its shape, size, and the frequency of the signal it is sending. Two antenna types were used for this research. The first type was a purchased rectangular horn antenna; the second was a custom designed helical antenna. Further descriptions of these two antennas are given in Chapter III. The gain for the antennas that were used in this project are given by Equation (5) (helical antenna) and specified in the product data sheet (rectangular horn antenna) [12].

$$G_{helical} = 10.3 + 10 \log(C^2 L / \lambda^3) \quad (5)$$

Free space path loss is calculated using Equation (6), where S is the distance in km and f is the frequency in GHz [16]. Loss from pointing error is calculated using Equation (7) where θ_{Tx} is the transmit beam width and e_{Tx} is the pointing error.

$$L_s = 20 \log_{10} S + 20 \log_{10} f + 92.45 \quad (6)$$

$$-12 \left(\frac{\theta_{Tx}}{e_{Tx}} \right)^2 \quad (7)$$

The link budget takes into account system and background noise using Equation (8), where -228.6 is the Boltzmann constant, T_s is the system temperature and B_N is the bandwidth of the signal [12].

$$N = -228.6 + T_s + B_N \quad (8)$$

Wertz shows how to calculate the ratio of power received to noise call carrier to noise ratio ($\frac{C}{N_0}$) [12]. The data rate (R_b) is subtracted from carrier to noise ratio to give the link's $\frac{E_b}{N_0}$, shown in Equation (9). Given a modulation and a coding scheme, there is a relationship between $\frac{E_b}{N_0}$ and BER, which can be looked up in various tables. This value is the required $\frac{E_b}{N_0}$. The difference between the required $\frac{E_b}{N_0}$ and the calculated $\frac{E_b}{N_0}$ is called the link margin (M_L), Equation (10) [12]. For this project, the desired link margin is +3 dB.

$$\frac{E_b}{N_0} = \frac{C}{N_0} - R_b \quad (9)$$

$$M_L = \left(\frac{E_b}{N_0} \right)_{\text{Predicted}} - \left(\frac{E_b}{N_0} \right)_{\text{Required}} \quad (10)$$

E. EXISTING CUBESAT X-BAND TRANSMITTERS

There are benefits and drawbacks to using a commercially available radio. The benefits are that the transmitters are a finished product, tested, space rated, ready to drop into a CubeSat, and come with engineering support. However, the downside is their higher cost, and operators have less ability to customize or choose modulation and coding schemes.

There are commercially available X-band transmitters in a CubeSat form factor available; two examples were investigated when starting this project. Syrlinks and EnduroSat both sell CubeSat form factor X-band radios. Syrlinks has both an X-band transmitter and an X/S-band transceiver [17] [18]. EnduroSat also sells both an X-band transmitter and a separate S-band receiver [1] [19]. Table 2 shows basic details for these radios.

Table 2. Examples of commercially available X-band transmitters for CubeSats

Module	Manufacturer	Frequency (MHz)	Data Rate (Mbps)	Power (W)	Cost (\$)
EWC27-FM	Syrlinks	8025-8400	3-50	1-2	64,800
EWC27-FM	Syrlinks	8025-8400	100	1-2	89,900
X-band Transmitter	EnduroSat	8025-8400	50	2	25,500
Swift-XTS	Tethers Unlimited	7000-8500	100	1-7	N/A

F. PREVIOUS NPS RESEARCH

Multiple projects of the Space Systems Academic Group’s (SSAG) Small Satellite Lab have preceded this one. This research has included small satellite development, communications systems development, and ground station research. Previous SSAG theses have developed CubeSat payloads and flown them on HAB flights. Two specific projects that have served as a foundation for this research are “Critical Vulnerabilities in the Space Domain: Using Nanosatellites as an Alternative to Traditional Satellite Architecture” by Philip Swintek, and “Software-Defined Radio Payload Design for CubeSat and X-band Communications” by Bianca Lovdahl.

Swintek proposed and developed a VHF communications relay CubeSat called Software Assisted VHF Information Overhead Relay-CubeSat (SAVIOR-Cube) [20]. In 2018, SAVIOR-Cube demonstrated transmission for digital voice over a VHF radio link from a CubeSat payload on a HAB flight [20]. This demonstrated the feasibility of using CubeSat payloads for communications applications.

In 2018, Com-Cube was developed by Lovdahl in order to demonstrate the use of an SDR in a CubeSat payload form factor to transmit digital data [21]. Com-Cube was flown in 2018 on a HAB [21]. Com-Cube would have operated in the X-band, but a delay in getting frequency use approval prevented the use of the 8025–8400 MHz band. Subsequently Com-Cube was operated in the C-band [21]. The Com-Cube HAB flight demonstrated the use of a CubeSat payload to transmit data (pictures from an on board camera) from a CubeSat to a ground station [21]. This thesis was started as a continuation

of Com-Cube and, in addition to using an X-band frequency, new digital signal processing would be developed to produce higher data rates and introduce forward error correction.

THIS PAGE INTENTIONALLY LEFT BLANK

III. HARDWARE

The hardware for this project has been chosen to provide a cost effective, commercial-off-the-shelf X-band SDR solution that would fit inside a 1U form factor.

A. REQUIREMENTS

The requirements for the X-band SDR payload are listed in Table 3. The primary objective of the project was to achieve a 1 Mbps transmission. The SWaP requirements are based on the 2U HAB bus that has 1U of space for payload, can supply 3 A of power at 5 V to the payload, and is limited to a total weight (bus plus payload) of 4 lb (1.81 kg) by FAA regulations [22].

Table 3. X-band SDR requirements

Requirement	Threshold	Objective
Data Rate	300 kbps	1 Mbps
Size	1U	0.5U
Mass	1 kg	0.5 kg
Power	15W	10W

B. HARDWARE USED

This project has two major assemblies. First is the HAB bus and the second is the X-band radio payload. The HAB bus provides the housing and structure, power, and command and data handling processes. The payload performs the functions of transmitting data in the X-band and taking pictures to be transmitted.

1. Payload: X-Band Radio

The X-band radio consists of an Ettus B-205 mini-i SDR controlled by a Raspberry Pi 3 A+. Since the B-205 is limited to a maximum frequency of 6 GHz, additional RF components are required to step up the transmitted frequency to X-band. These were a local oscillator (LO), a frequency mixer, and amplifiers. The local oscillator in this case was the ADF-4356 frequency synthesizer. Figure 4 is a schematic of the X-band radio payload. The

Raspberry Pi, Ettus B-205mini-i, and the ADF-4356 had been identified as the preferred hardware components by the previous NPS SSAG Com-Cube project [21]. The additional elements (amplifiers, filters, and mixer) were selected during this study. Table 4 lists the components used in this project. The total cost of payload components listed in Table 4 was \$2375.33. Not included in the cost estimate are the 6 RF cables used. These cables were surplus equipment and did not have a cost estimate. Future versions will have to consider the cost of cabling or use additional connectors such as the PE9418 used in this project.

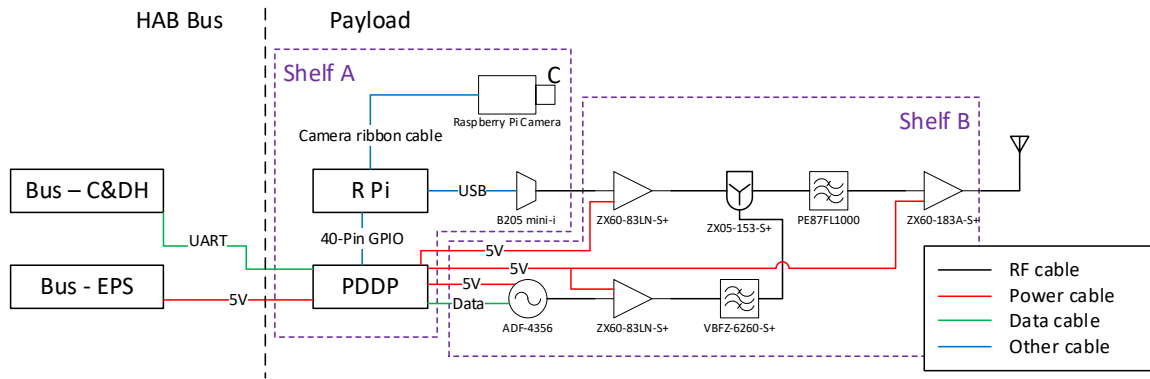


Figure 4. Block diagram of X-band SDR payload configured for 10 GHz

Two hardware configurations were assembled and tested in the laboratory. These two were based on the final transmission frequency of either 8250 MHz or 10,250 MHz. At the point it became necessary to switch the transmission frequency from 8250 MHz to 10,250 MHz, a change in the LO frequency also had to be made. This required a change in the filters on the LO and final RF. This was a swap from the ZVBP-8250-S+ and ZVBP-5800-S+ to the PE87FL1000 and the VBFZ-6260-S+.

Table 4. Components used to build the X-band payload

Component	Manufacturer	Quantity	Cost (\$)
Raspberry Pi 3 B+	Raspberry Pi	1	35.00
B205mini-i	Ettus Research	1	942.00
ADF4356-EVAL	Analog Devices	1	300.00
ZX60-183A-S+	Mini Circuits	1	169.95
ZX60-83LN-S+	Mini Circuits	2	154.95
ZX05-153-S+	Mini Circuits	1	48.95
VBFZ-6260-S+	Mini Circuits	1	39.95
PE87FL1000	Pasternack	1	481.63
PE9418	Pasternack	1	47.95
Perma-Proto HAT	Adafruit	1	4.95
Total			2,380.28

The components for this payload were organized on two shelves. Due to the size of the ADF-4356 development board (105 mm x 75 mm) being larger than the available internal cross-section of the HAB (95 mm x 95 mm), the two shelves of the payload had to be mounted longitudinally, as opposed to the traditional transverse mounting for CubeSat components. The length of the ADF-4356 board also means that the X-band radio payload would occupy greater than 1U of volume.

Equipment shelf A holds the SDR and Raspberry Pi on opposite sides of the shelf, shown in Figure 5 and Figure 6. Shelf B holds the ADF-4356 development board on one side with all of the other RF components on the other side, shown in Figure 7 and Figure 8.

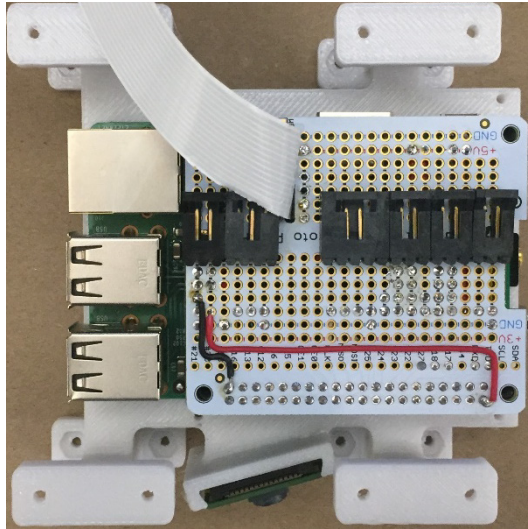


Figure 5. X-band payload equipment shelf A, interior side



Figure 6. X-band payload equipment shelf A, exterior side

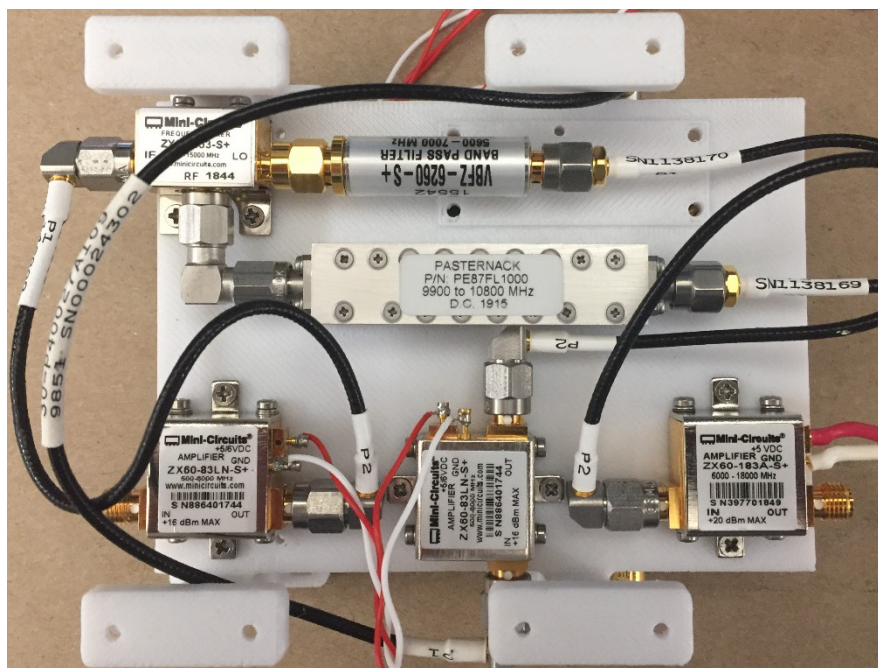


Figure 7. X-band payload equipment shelf B, interior side

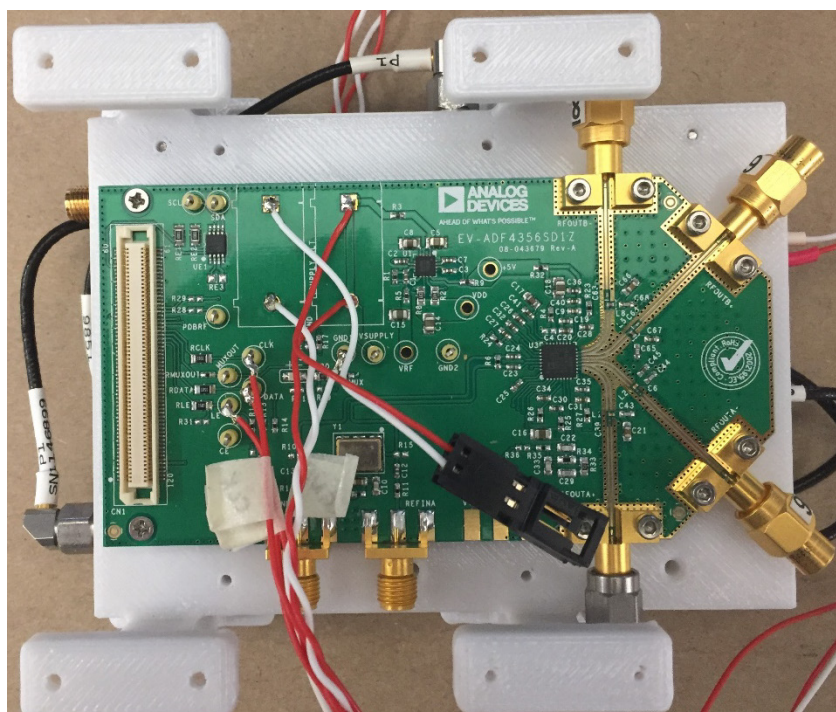


Figure 8. X-band payload equipment shelf B, exterior side with cover removed

a. Ettus B-205 Mini-i

The Ettus B-205 mini-i is a credit card-sized, duplex transceiver with a frequency range of 70 MHz - 6 GHz, capable of 56 MHz of bandwidth [23]. A USB-3 interface provides power and data from the Raspberry Pi. The B-205 has software support for Python, C/C++, GNU Radio, and UHD hardware drivers [23]. Figure 9 is a picture of the Ettus B200mini enclosure, the B205mini enclosure is the same size.

The B-205mini is used to modulate data at a designated intermediate frequency, which will then be up converted to the desired final RF for transmission. The intermediate frequency is controlled by an input to the GNU Radio software. For this project, two different IF were used: an IF of 2.45 GHz for a final transmission frequency of 8.25 GHz and an IF of 4.45 GHz for a final transmission frequency of 10.25 GHz. In this project, the B-205mini is used only as a transmitter and the receive functionality is not used.



Figure 9. Ettus B200 enclosure compared with a U.S. quarter. Source [23].

b. Raspberry Pi 3 A+ / Raspberry Pi 4 B

The Raspberry Pi is a low-cost, single-board computer developed by the Raspberry Pi Foundation. Initially the Raspberry Pi 3 was used in the design and development of this project. Two form factors of the Raspberry Pi 3 were tested: the A and the B. Both form factors have the same performance but have different input/output (I/O) connections. The A form factor has one USB 2 connector; the B form factor has four USB 2 connectors and

one ethernet connector. The Raspberry Pi 4 was released midway through this project. It introduces a faster processor and replaces two USB 2 connectors with two USB 3 connectors providing a significant boost of data transfer to USB devices. [24] Since the Raspberry Pi 3 B+ and Raspberry Pi 4 B have the same form factor, both were tested. The Raspberry Pi 3 was selected for the flight hardware. Figure 10 is a picture of a Raspberry Pi 4 and Figure 11 is a picture of a Raspberry Pi 3 A+.

The Raspberry Pi is used as the controller for the X-band payload. It runs all payload software, communicating with the HAB bus C&DH via UART interface, capturing and storing digital images with the piCamera, packetizing data with the chunking.py software (see appendix X), and performing digital signal processing with the GNU Radio program. Additionally, the Raspberry Pi interfaces with the SDR via USB connection, providing power to the SDR and controlling the SDR. The Raspberry Pi sends initialization commands to the ADF-4356 and sets the parameters of that board in order to get the desired LO signal.

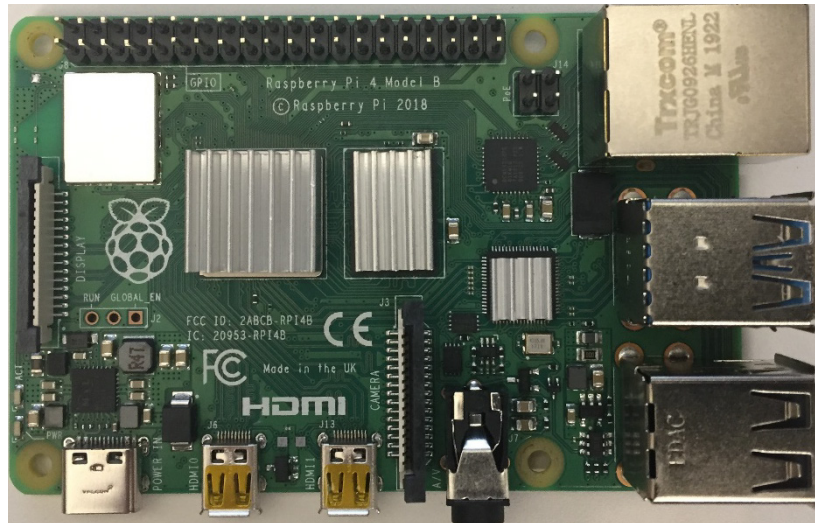


Figure 10. Raspberry Pi 4 model B

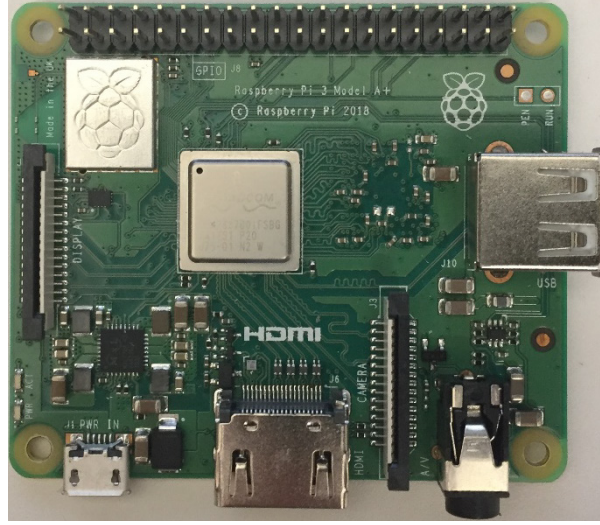


Figure 11. Raspberry Pi 3 model A+

c. Power and Data Distribution Panel

A custom power and data distribution panel (PDDP) was built for this project. Figure 12 and Figure 13 are pictures of the PDDP. The PDDP serves multiple purposes. It provides a single power connection to connect to the HAB bus; it distributes power to the various components in the X-band radio; it provides data connections to the HAB bus; and it provides a data connection to necessary X-band radio components. The PDDP was built using the Adafruit Perma-Proto HAT for Pi Mini Kit. This board provided a 40-pin GPIO connection point to the Raspberry Pi GPIO header. This board houses five connectors: three 2-pin connectors for power, one 3-pin connector for UART data (BUS), and one 4-pin connector for data to the ADF-4356. Additionally, for power conditioning the PDDP contains one each of 10 μf , 0.1 μf , and 0.01 μf capacitors on the power line to the ADF-4356 LO. An additional power and ground line were run on the top of the PDDP directly to the corresponding GPIO pins in order to provide a lower resistance and therefore lower voltage drop on the power supplied to the Raspberry Pi. Figure 12 is a picture of the top of the PDDP, Figure 13 is a picture of the back of the PDDP, and Figure 14 is a schematic of the PDDP. The Pi HAT serves as a convenient location to connect the various power and data wires needed in the X-band radio. The power conditioning was performed to prevent power fluctuations from the local oscillator.

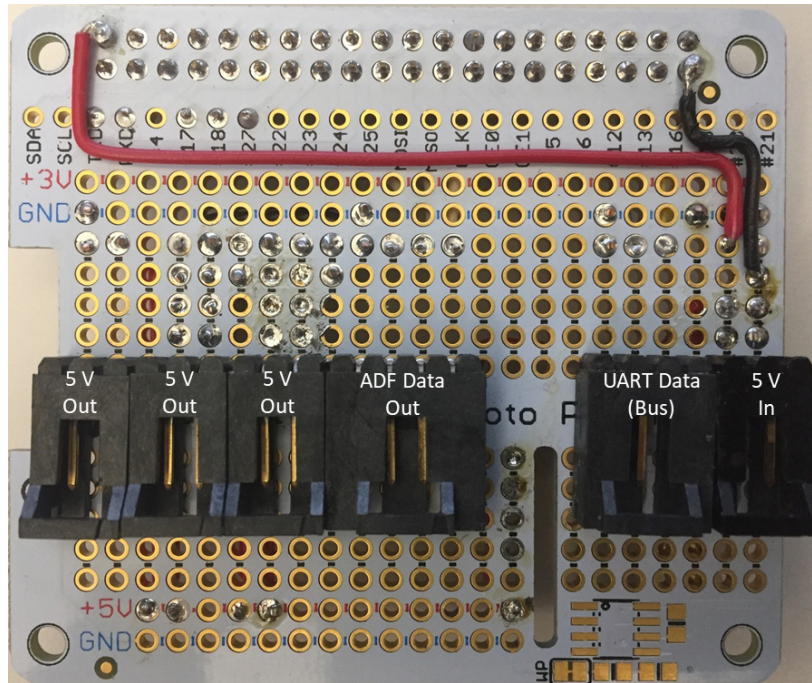


Figure 12. Power and data distribution panel (front)

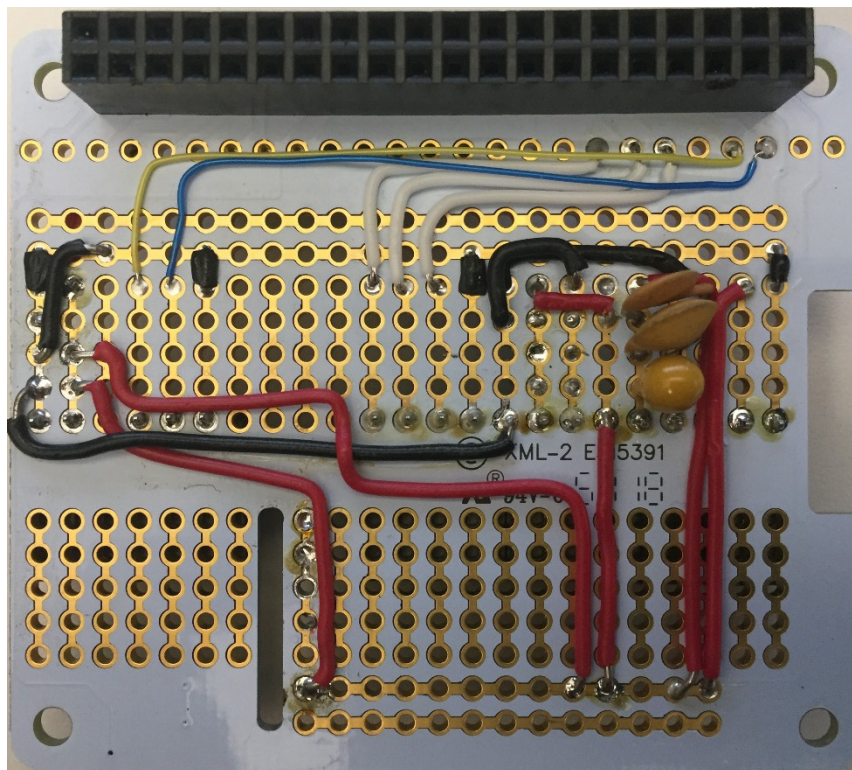


Figure 13. Power and data distribution panel (back)

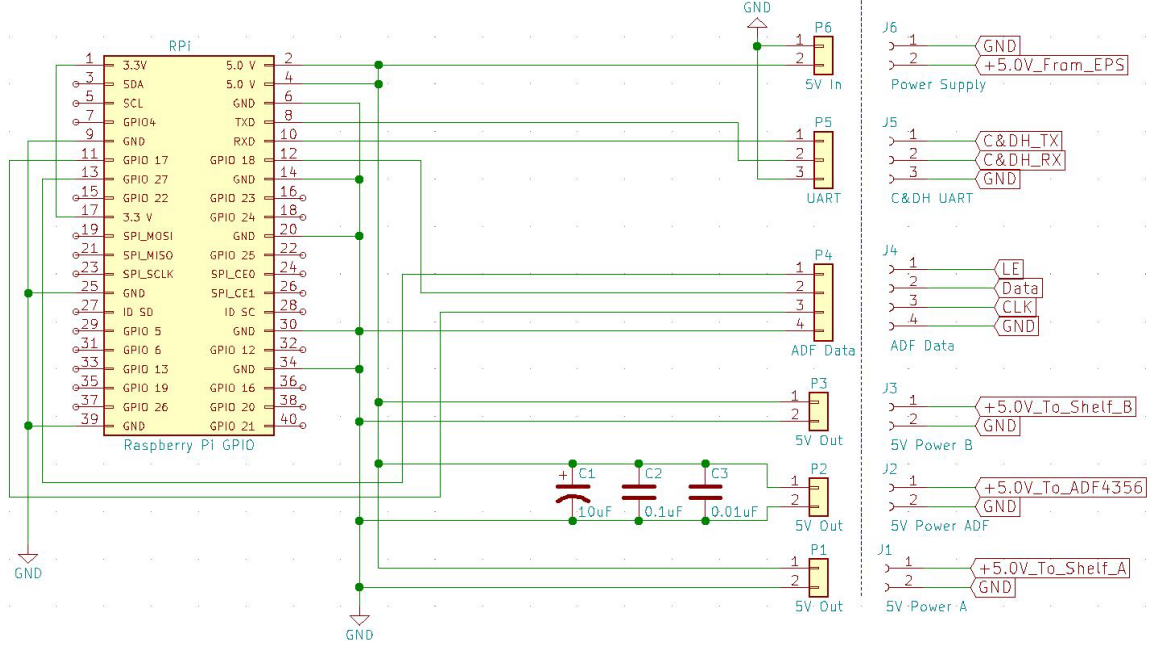


Figure 14. Schematic of power and data distribution panel

d. Analog Devices ADF-4356 Evaluation Board

The Analog Devices ADF-4356 frequency synthesizer evaluation board is used as a local oscillator (LO) to provide an RF signal to mix with the signal from the SDR. Figure 15 is a picture of the ADF-4356 evaluation board. The ADF-4356 can be controlled by a SDP-S controller board, which is in turn controlled by a windows program via USB connector. Alternatively, the ADF-4356 can be controlled by sending data, clock, and LE signals direct to the board. The procedures to directly set the register values are contained in the ADF-4356 data sheet [25].

Register values for the ADF-4356 were obtained using the Windows program provided by Analog Devices shown in Figure 16 [26]. In addition to controlling the SDP-S board, this program also calculates and display the required register values for the ADF-4356. The register values for the desired ADF-4356 settings were placed into a text file. The ADF4356.py program read this text file and transmitted the register values to the ADF-4356. The register values are listed in Appendix A. The parameters of the output signal from the ADF4356 are in Table 5.

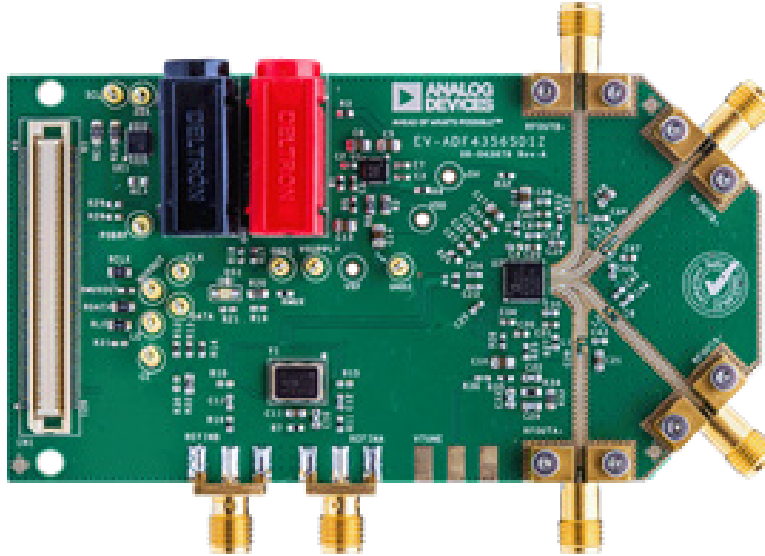


Figure 15. Image of Analog Devices ADF-4356 evaluation board.
Source: [26].

Table 5. Selected ADF-4356 local oscillator output parameter values

Parameter	Value
VCOout (frequency)	5800 MHz
RF Power out (channel A)	-4 dBm

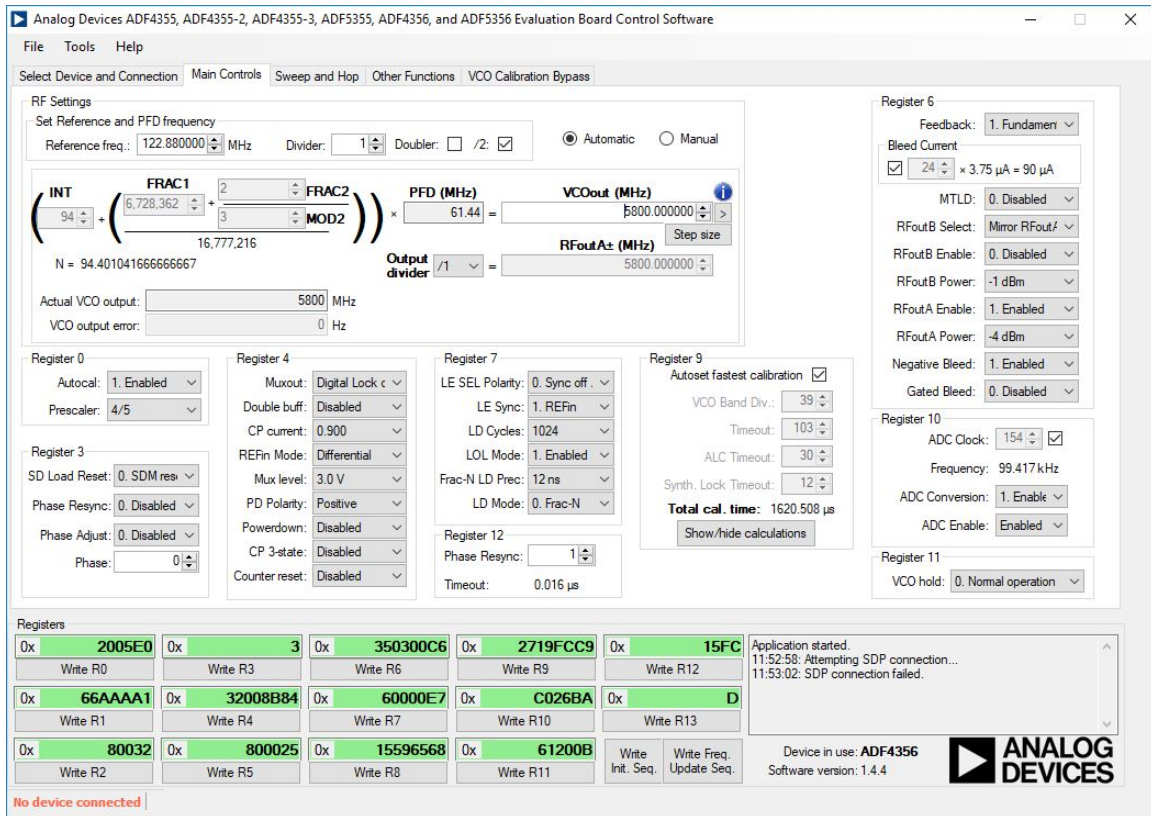


Figure 16. Screen capture of ADF-4356 control software

e. ZX05-153-S+ Mixer

The ZX05-153-S+ is an enclosed RF mixer with SMA connectors. It is rated for a 7 dBm LO input at frequencies ranging from 3400 MHz to 15,000 MHz. The specified IF input frequency ranges from DC to 4000 MHz. The final RF is rated for 3400 MHz to 15,000 MHz. Table 6 shows the LO and IF inputs for the desired RF output. Figure 17 is a picture of the frequency mixer used.

The frequency mixer is used as an up-converter and combines the LO input from the ADF-4356 and the IF signal from the B205mini to create an RF signal at the desired transmission frequency.

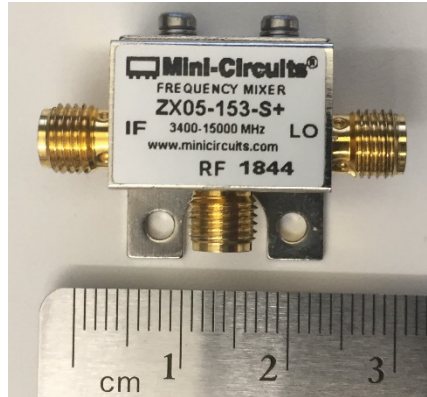


Figure 17. Frequency mixer: Mini-Circuits ZX05-153-S+

Table 6. RF mixer input and output parameters

Inputs				Output	
IF freq	IF power	LO freq	LO power	RF freq	RF power
2450 MHz	-3.1 dBm	5800 MHz	7.15 dBm	8250 MHz	-11.2 dBm
3950 MHz	-3.5 dBm	6300 MHz	8.75 dBm	10,250 MHz	-17.3 dBm

f. ZX60-83LN-S+ Amplifier

The ZX60-83 is an enclosed wide band amplifier with SMA connectors designed for use in the 0.5–8 GHz frequency range. It provides a typical gain of 20.8 dB at 4 GHz with an output power of 17.6 dBm at 1 dB compression. Its noise figure at 4 GHz is 1.4 dB. This project uses two of these amplifiers, one each to boost the LO and IF signal strengths to approximately 7 dBm prior to reaching the mixer. Figure 18 is a picture of the ZX60-83 amplifier. Table 7 shows the input and output power of the two ZX-60-83 amplifiers along with the LO and IF input settings.



Figure 18. Intermediate frequency and local oscillator amplifiers:
Mini-Circuits ZX60-83LN-S+

Table 7. Input and output values for LO and IF amplifiers

Input Device	Device setting	Input power	Output power
LO, 6300 MHz	-4 dBm	-6.26 dBm	13.2 dBm
IF, 3950 MHz	50 dB	-25.7 dBm	-3.5 dBm

g. ZX60-183A-S+ Amplifier

The ZX60-183 is a two-stage amplifier designed for use in the 6–18 GHz frequency range. It provides a typical gain of 28.6 dB at 10 GHz with an output power of 18.8 dBm at 1 dB compression [27]. Its noise figure at 10 GHz is 5.1 dB. In a satellite application, this amplifier would be a low noise amplifier used to feed a high power amplifier prior to transmission. In this project, the ZX60-183 amplifier is used as the final amplifier. Figure 19 is a picture of the ZX60-183 amplifier.

The input to this amplifier is the filtered signal from the mixer of the up-converted, modulated data at the final transmission RF. The output of the amplifier goes to the transmission antenna.

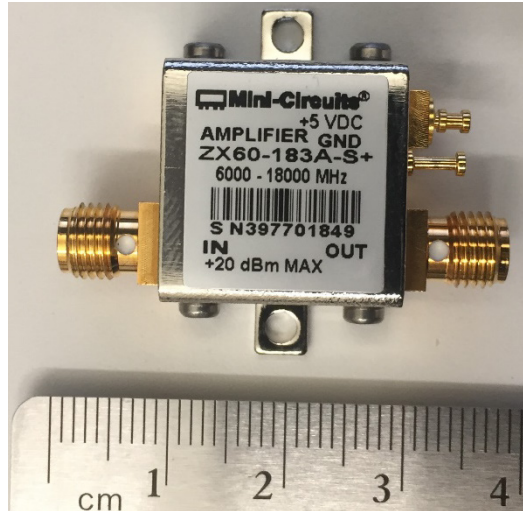


Figure 19. Final amplifier: Mini-Circuits ZX60-183A-S+

h. PE87FL1000 Filter

The PE87FL1000 is an eight-section bandpass filter designed with a passband from 9,900 MHz to 10,800 MHz, with a maximum loss of 1 dB in the pass band. [28] This filter is used to eliminate all but the desired transmission frequency from the mixer prior to amplification and transmission. This filtering is required since the process of mixing two RF signals generates a sum and difference and the two original signals and in this case only the sum of the LO and IF is desired. Figure 20 is a picture of the PE87FL1000 filter.



Figure 20. 10 GHz final bandpass filter: Pasternack PE87LF1000

i. ZVBP-8250-S+ Filter

The ZVBP-8250-S+ was the final filter after the RF up-conversion by the RF mixer in the 8250 MHz configuration. The pass band of the ZVBP-8250 (8025-8475 MHz) does not pass the desired 10,250 MHz signal [29]. Therefore, it was replaced by the PE87FL1000 in the 10,250 MHz configuration. Figure 21 is a picture of the ZVBP-8250 filter.



Figure 21. 8 GHz final bandpass filter: Mini-Circuits ZVBP-8250-S+

j. VBFZ-6260-S+ Filter

The VBFZ-6260 is a bandpass filter by mini circuits with a designed pass band of 5600–7000 MHz with an insertion loss of 1.2 dB at 6260 [30]. This filter has an input of an amplified 6300 MHz signal from the local oscillator. It is used to filter any harmonics or spurious signals from the local oscillator so that they do not reach the mixer and produce undesirable output. The output from the VBFZ-6260 is to the LO port of the RF mixer. Figure 22 is a picture of the VBFZ-6260 filter.



Figure 22. Local oscillator filter: Mini-Circuits VBFZ-6260-S+

k. ZVBP-5800-S+ Filter

The ZVBP-5800 was the initial filter used between the LO amplifier and the RF mixer. When the transmission frequency was raised to 10,250 MHz, the LO frequency also needed to be raised, in this case to 6300 MHz. The passband of the ZVBP-5800 (5725-5875 MHz) does not pass the new LO frequency [31]. Therefore, it was replaced by the VBFZ-6260 in the 10,250 MHz configuration. Figure 23 is a picture of the ZVBP-5800 filter.



Figure 23. Local oscillator filter: Mini-Circuits ZVBP-5800-S+

l. Raspberry Pi Camera

The Raspberry Pi camera is used as a sensor to record data. This camera is attached to the Raspberry Pi 3 A+ with a ribbon cable to the camera port. Detailed information can be found in [32].

m. Antennas

Multiple antenna types for this project were considered. This includes dipole, helical, and patch antennas. Antenna selection was based on transmission frequency, polarization, antenna size, antenna gain, and antenna beam width. The frequency was determined, as discussed in Chapters I and II. The future MC3 X-band ground segment antenna has a right hand circular polarization (RHCP); the polarization of the antenna for this project was chosen to match the ground station's polarization. Antenna size is limited to what can fit onto a CubeSat bus. Antenna gain and beam width have an inverse

relationship and, since it is expected that the payload will swing side to side on both its ascent and descent a wider beam width is preferred.

Patch antennas are particularly attractive for this study, as they have a small form factor, and the ones investigated for this project typically had large beam widths with an acceptable gain. Patch antennas can also be made with RHCP. The X-band patch antenna from Endurosat was initially identified as suitable for use on both the HAB flight and follow on space flights. Other manufacturers, such as Sage Millimeter [33], also make patch antennas in the 8025–8400 MHz frequency range; however, all of these were linearly polarized and a non-recurring engineering fee was required to get a RHCP antenna. Furthermore, the change in the operating frequency made both of these antennas unsuitable.

Multiple helical antennas were designed based on the online helical antenna calculator from Coppens [34]. The values from the calculator were used to design a 3-D printed core which had wire wrapped in printed grooves corresponding to the number of turns and coil spacing. The polycarbonate core was then attached to a machined aluminum ground plane and a SMA coaxial connector was added to the back. After construction, the antenna was attached to a Keysight N9918A spectrum analyzer and the frequency with the lowest voltage standing wave ratio (VSWR) was measured. The ideal VSWR is 1, which indicates that all of the energy input to the antenna is radiated as an RF signal and no energy is reflected back. This frequency with the lowest VSWR is called the tuned frequency for the antenna. The beam width of the antenna was calculated using Equation (11) [12]

$$\theta = \frac{52}{\sqrt{C^2 L / \lambda^3}} \quad (11)$$

When analyzed, the initial design for a 10.25 GHz antenna did not have a tuned frequency at the desired point. Instead, the tuned frequency with the lowest VSWR was higher than designed at 11.46 GHz. This ratio of 10.25/11.46 was used to select a new design frequency of 9.168 GHz. This second antenna was created using the same process; its tuned frequency was 10.44 GHz, which fell inside the desired operating band of 10.0 GHz – 10.5 GHz and was deemed acceptable for use in this project. A third antenna was

created with a design frequency at 8.25 GHz for the possibility that use of the 8.025-8.4 GHz frequency band was approved. The antennas created for this project are shown in Figure 24. Table 8 lists the design parameters and the resulting tuned frequencies.

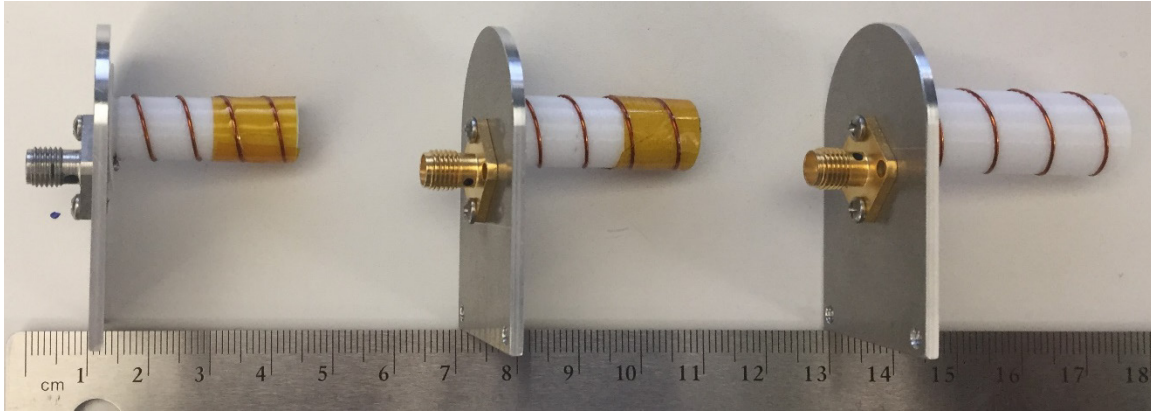


Figure 24. Helical antennas

Table 8. Helical antenna design parameters

Design Frequency (GHz)	Wavelength (mm)	Diameter (mm)	Gain (dB)	Winding Step (mm)	Separation of adapter section (mm)	Beam width Calculated (deg)	Final Tuned Frequency (GHz)	VSWR
10.25	29.2	9.9	9.39	6.7	0.2	43.3	11.46	1.031
9.168	32.7	11	9.39	7.5	0.3	42.3	10.44	N/A
8.25	36.3	12.3	9.39	8.3	0.3	45.4	8.946	1.011

2. Bus

The HAB bus has been used in previous small sat lab classes and theses. The bus consists of a 3-D printed structure that houses bus components and the payload. The bus contains EPS and C&DH boards that control the parachute and balloon release, and provide telemetry and payload commands via a MHz radio link at 915 MHz [35]. Detailed information can be found in the HAB interface control document (ICD).

The EPS supplies power to the payload via a single connector at 5V and is rated for 3 A. During testing, the X-band payload used approximately 2.25 A at 5 V; however, when attempting to pull this current through the EPS the linear regulator for the payload power supply quickly overheated. An alternate power supply board was found to supply the payload with 5V power. This was the LTC 3119, which uses a 2129A switch power supply to step down the input 8V power to 5V. This board was able to supply the required power without overheating.

During testing it was found that when the Raspberry Pi and SDR were running at high data rates (≥ 1 Mbps), the current draw through the PDDP would cause a voltage drop between 0.1 V and 0.2 V. Any time that the voltage dropped below 5.0 V with the radio running, the Raspberry Pi would display an under-voltage warning and GNU Radio would show underrun errors. These errors would also cause transmission errors with corresponding dropped packets at the receiver. To compensate for this the payload power supply was modified to provide 5.25 V, so with the resulting voltage drop upon running the radio the voltage to the Raspberry Pi remained at or above 5.0 V.

IV. SOFTWARE

A. REQUIREMENTS

The stated project requirements (low cost, SWaP limits, 1 Mbps data-rate, QPSK modulation, incorporation of FEC) along with a desire for simplicity constrained the choice of software and software tools used in this project. The requirement for low-cost hardware drove the selection of a Raspberry Pi computer as a system controller that would limit the software tools that could be used for digital signal processing (DSP). The author's limited programming ability biased the software selection to graphical tools for DSP instead of directly writing code.

B. SOFTWARE DESIGN

There were two separate computers in this project running separate software. The payload software was developed for use in this project and the HAB bus software is common software for all SSAG HAB bus flights.

1. Payload Software

Software for this project either was written in Python or generated using GNU Radio. Digital signal processing software was created using the GNU Radio Companion (GRC) graphical interface where flowgraphs created in GRC defined the DSP functions to be performed by the radio. Additional software to perform other functions were written in Python. These other functions include initialization of the ADF-4356 evaluation board, capturing images from the camera (sensor data), processing and packetizing the sensor data, and sending packetized data to the radio.

GNU Radio is an open source software development package that provides the digital signal processing for SDR [36]. GRC is a graphical user interface for GNU Radio that allows functional blocks to be connected graphically where each block represents a single DSP function. GRC then generates a python program that will perform the digital signal processing that is desired. Each block in GRC represents a piece of software code that can be written in either Python or C++. In addition to the blocks provided in GRC,

users can develop and write their own blocks in either Python or C++. All blocks except one used in this payload flowgraph were default blocks provided with GRC. The one additional block was the FSL: Synchronizer Tagger (RX) custom block taken from a CC1101 transmitter and receiver [37]. This block was then modified to correct errors that occurred when used in this project.

a. GNU Radio Flowgraph

Two GRC flowgraphs were created for this project. These flowgraphs were for the transmitter and receiver functions; the one for the transmitter would be run on the Raspberry Pi and the one for the receiver would be run on a Linux laptop.

The transmitter is designed to read data in binary from a first in first out (FIFO) file, separate the data into transmission frames, and apply a 1/2 rate convolutional FEC, doubling the frame size. Next, a ramp up header, a synchronization header, access code, and a trailing buffer are added to the data. The data stream is then modulated with a QPSK modulator and the signal strength is multiplied by $\sqrt{2}/2$ in order to avoid saturating the transmitted. The transmission frame is depicted in Table 9. The GRC transmitter flowgraph, shown in Figure 25, depicts the signal processing performed for transmission.

Table 9. Transmission data frame

Ramp up sequence	Synchronization sequence	Access Code	Data Packet	Buffer
8 bits	24 bits	32 bits	1024 bits	32 bits

The output of the GRC flowgraph is a Python program, named `xband_tx.py`, which performs the digital signal processing for the transmitter. The code for `xband_tx.py` is contained in Appendix G. When this transmitter program runs for the first time after power up, a reboot, or a reset of the USB bus, the process of initializing the SDR can take more than a minute. If the other processes, particularly the camera, ran during this time, there would be a backlog of data by the time the GRC transmitter program finished starting up. In order to prevent this from happening two lines of code are manually inserted into the

transmitter program. These two lines create a file, called “grc.start,” after the SDR is initialized. This grc.start file is used by the startup program as an indication to continue launching the remaining programs.

The transmitter program is designed with multiple parameters that can be passed to it when it is called. These parameters are SDR output frequency, sample rate, input file name, data packet size, samples per transmitted symbol, and transmitter gain. These parameters, their short and long titles used in calling the program, and the default values are listed in Table 10.

Table 10. Xband_tx.py input parameters and default values

Parameter	Short title	Long title	Default value
SDR frequency	-f	--freq	2.45 _E 9
Sample rate	-s	--samp-rate	1.00 _E 6
Sample per symbol	-p	--samples-per-symbol	2.0
Excess Bandwidth	-e	--eb	0.350
Data packet size	-l	--frame-len	1024
Input file name	-n	--fifoname	/home/pi/xband_tx/file2
Transmitter gain	-g	--tx-gain	50.0

The receiver is designed to receive, synchronize, and demodulate data then pass it via UDP port to other programs. This is accomplished by the flowgraph shown in Figure 26. First, an automatic gain control is applied to adjust the signal level to 1. Then the Correlation Estimator block looks for the access code applied by the transmitter. When the access code is found Correlation Estimator block applies a tag with estimated errors in frequency and phase. The Polyphase Clock Sync and Costas Loop block use that estimate and correct for errors in frequency and phase. The Constellation Demodulator block demodulates the signal from RF baseband to binary data using the same constellation as the transmitter. Since QPSK has two bits of data in each symbol, the Unpack K Bits block is used to separate the data into individual bits and the bits are remapped with the Map block. The Correlate Access Code – Tag block searches for the access code. When a series of bits matches the given access code within a given threshold, in this case 3, then the Correlate Access Code – Tag block adds tag indicating the beginning of a data packet. The

FSL: Synchronizer Tagger (RX) block searches for the access code tag. When the tag is found the FSL: Synchronizer Tagger (RX) block passes one packet's worth of bits, in this case 2048 bits (the data packet size is 1024 bits doubled due to the FEC). Next, the FEC Extended Tagged Decoder block decodes the data and passes 1024-bit packets to the UDP port.

The receiver program is also designed with parameters that can be passed to it when it is called. These parameters are SDR output frequency, sample rate, samples per transmitted symbol, and receiver gain. These parameters, their short and long titles used in calling the program, and the default values are listed in Table 11.

Table 11. Xband_rx.py input parameters and default values

Parameter	Short title	Long title	Default value
SDR frequency	-f	--freq	4.00 _E 9
Sample rate	-s	--samp-rate	1.00 _E 6
Sample per symbol	-p	--sps	2.0
Excess Bandwidth	-e	--eb	0.350
Receiver gain	-g	--rx-gain	55.0

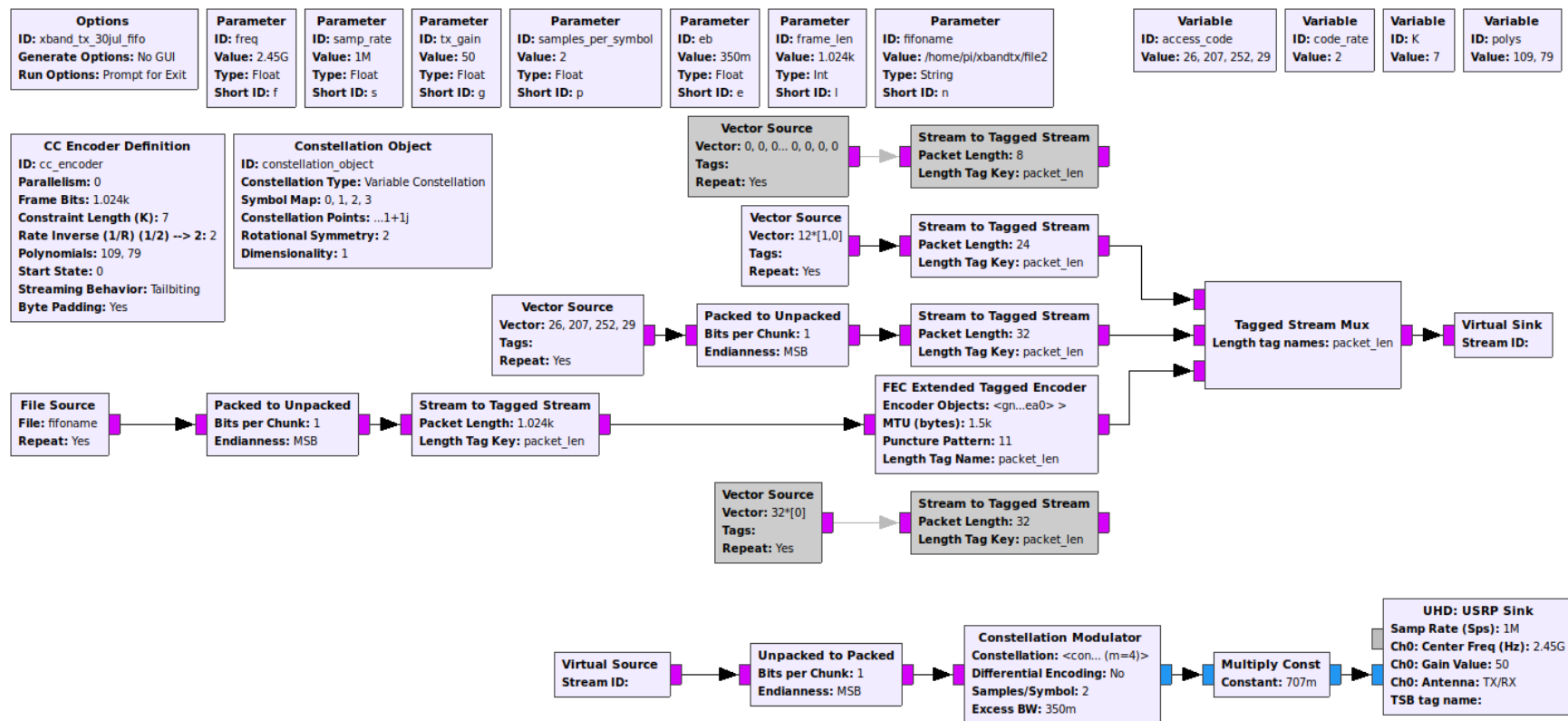


Figure 25. GNU Radio flowgraph for QPSK transmitter

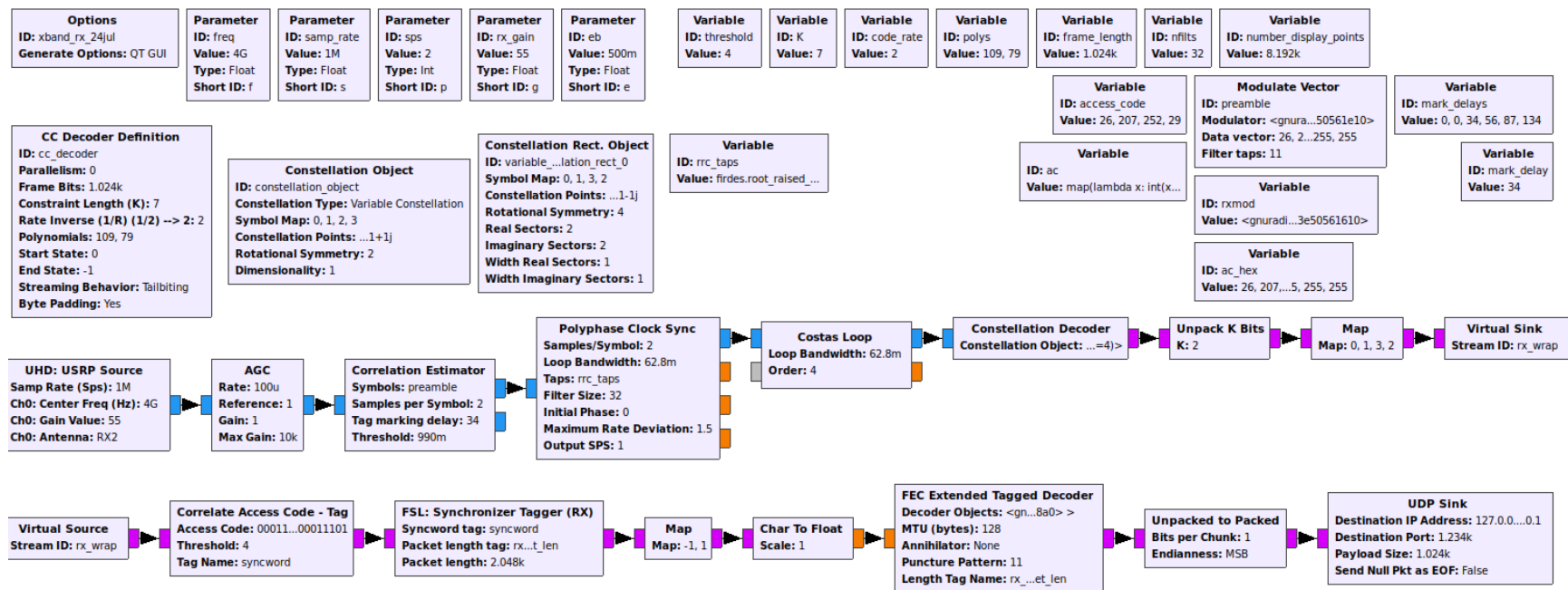


Figure 26. GNU Radio flowgraph for QPSK receiver

b. Standards Applied

The Consultative Committee for Space Data Systems (CCSDS) publishes recommended communications and data standards for spacecraft. Their stated objective is to “enhance governmental and commercial interoperability and cross-support, while also reducing risk, development time and project costs” [38]. Among their many standards and recommendations there are specified access codes to use for different forward error correction schemes. The CCSDS recommended access code for a convolutional encoder FEC is listed in the CCSDS Recommended Standard for TM Synchronization and Channel Coding [39].

c. Startup Program: start_payload.py

The startup program `start_payload.py` launches when the payload Raspberry Pi is powered on and subsequently starts all of the other programs required for the payload to operate. The startup program first launches the ADF initialization program (`adf4356.py`), and then launches the GNU Radio transmitter program (`xband_tx.py`), the camera program (`xband_camera.py`), and the chunking program (`chunking.py`). After the transmitter program is started, `start_payload.py` will continually check for the file `grc.start` prior to continuing to load the remaining two programs. The programs `xband_tx.py`, `xband_camera.py` and `chunking.py` are periodically checked and if not running, they will be restarted. The code for `start_payload.py` is in Appendix C.

d. ADF 4356 Initialization Program: adf4356.py

`Adf4356.py` initializes the ADF4356 frequency synthesizer. This is done by sending binary data to the board in the procedure described in the ADF 4356 datasheet [25, p. 5]. `Adf4356.py` is called with one argument, a filename for the register values to be sent to the ADF 4356 evaluation board. The register value definitions, the method to calculate the register values, and the order to send the register values to the evaluation board are also contained in the ADF 4356 datasheet [25]. The register value file is stored as American standard code for information interchange (ASCII) text data in two comma-delimited columns. The first column is the two-digit register number, and the second column is the

hexadecimal value sent to the register. Appendix D is the code for adf4356.py and Appendix E is the register file used to initialize the ADF 4356 for a frequency of 6300 MHz with a power output of -4 dBm.

e. Camera Program: xband_camera.py

The camera program will periodically take pictures and save them in a directory for subsequent transmission. Initially, xband_camera.py looks for a directory called “photos,” and if the directory does not exist, it is created. Next, xband_camera.py checks if there are pre-existing photos, and if there are, it finds the last file number and starts numbering new photographs with the next number. Photos are taken every 45 seconds and stored in the photo directory named “photo_XXXX.jpg,” where XXXX is the four digit file number starting with 0000 and counting up. The camera is set to take 640x480 pixel photographs and save them in Joint Photographic Experts Group (JPEG) format with an approximate size of 715–720 kB. With a 1 Mbps transmission speed and a 1/2 rate FEC, each photo will take approximately 12 seconds to transmit. The longer wait period between photographs will allow each photograph to be sent twice and have time remaining to send any missing packets requested by the ground station. The code for xband_camera.py is in Appendix F.

f. Packetizing Program: chunking.py

The packetizing program, chunking.py, was adapted from the one used in the COM-CUBE program [21] by SSAG Software Engineer Jim Horning. This program is used for both transmit and receive processing. On the transmit side chunking.py checks for files to transmit, reads those files, divides those files into data packets, assigns file numbers and packet numbers to those packets, calculates a cyclic redundancy check (CRC), and finally adds a header that contains file and packet numbers and CRC. This data packet with its header is then written to a FIFO file for the GRC program to read for transmission. The code for chunking.py is in Appendix G.

The transmitter will send each new file, in full, three times and then move on to the next file in the photos directory unless either a file complete message or a missing packet request message is received. When a missing packet request is received, the transmitter will no longer send the complete file but, instead, will switch modes to send only the

requested packets. The packets on the requested list are sent in succession repeatedly until a file complete message is received. Once a file complete message is received, that file will no longer be transmitted and the transmitter will move on to the next file. If all files in the photo directory have already been sent and received, then the chunking program will wait until another photo is taken and saved to the photo directory.

When called in receive mode, chunking.py receives data packets from the GRC program via UDP port and then checks the CRC for each packet. If the CRC passes, then the packet is assembled with other packets that have passed their CRC checks in order to recreate the transmitted file. If the CRC fails then the packet is discarded; a single incorrect bit in a packet will generate an incorrect CRC.

The receive mode tracks which packets have been received in each transmitted file. When the number of missing packets is less than 100, then a request is sent to the transmitter with the missing packets listed. Once a complete file has been received, then feedback is sent that the file is complete.

2. Bus Software

The bus software is unmodified from the standard SSAG HAB bus software load. A brief description of the actions performed by bus software can be found in the HAB ICD [35].

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS

Three phases of testing were conducted for this project. First, individual components were tested and their performance was measured. Second, payload sub-assemblies, the assembled payload, and assembled CubeSat were tested in the laboratory. Third, the fully assembled CubeSat was tested in the field, initially in a NPS courtyard and then on a high altitude balloon flight.

A. COMPONENT TESTING

Individual components were tested to assess their performance when incorporated into the final payload design.

1. ADF 4356

The ADF-4356 was connected to a N9915A FieldFox spectrum analyzer while controlled by the SDP with the ADF-4356 software on a windows 10 computer. The ADF was connected to the spectrum analyzer with an 18” SEMFLEX cable with a loss of 0.50 dB. Various frequencies were input to the software and the output frequency was measured. Additionally, four power output levels were tested for selected frequencies. The results of this test are presented in Table 12 and Table 13.

Table 12. Local oscillator frequency output

Frequency Setting (MHz)	Frequency Measured (MHz)	Frequency Difference (MHz)
4000	3999.9425	- 0.0575
4250	4249.9375	- 0.0625
4500	4499.9350	- 0.0625
4750	4749.9300	- 0.0700
5000	4999.9275	- 0.0725
5250	5249.9250	- 0.0750
5500	5499.9200	- 0.0800
5750	5749.9175	- 0.0825
6000	5999.9125	- 0.0875
6250	6249.9100	- 0.0900
6500	6499.9050	- 0.0950
6750	6749.9025	- 0.0975
5800.0850	5800.0000	-0.0850
6300.0925	6300.0000	-0.0925

Based on the results in Table 13, the ADF-4356 could not supply the desired power level of 7 dBm into the mixer IF port. This required that an amplifier be added to boost the signal to the desired level. Table 14 shows the resulting power out of the amplifier at the various ADF-4356 power settings.

Table 13. Local oscillator power output

Power setting:	-4 dBm	-1 dBm	+2 dBm	+5 dBm
5800 MHz	-6.92	-4.23	-1.69	-1.24
6300 MHz	-7.27	-4.97	-2.61	-2.26

Table 14. Local oscillator power to mixer

Power setting:	-4 dBm	-1 dBm	+2 dBm	+5 dBm
5800 MHz	7.15	9.61	12.00	12.41
6300 MHz	8.75	11.23	13.46	13.84

Both include ADF4356, applicable filter, amplifier, and two Gore 6" cables.

Initial testing of the ADF-4356 showed that spurious transmissions occur at the first harmonic in the transmitted signal. In order to prevent this additional signal from being mixed with the IF, a filter was added. The filter was added between the ADF-4356 and the amplifier; however, there were still spurious signals after the amplifier, therefore the filter was placed after the amplifier prior to the mixer. In this configuration, the spurious signals were eliminated. Figure 27 shows the block diagram for the LO chain.

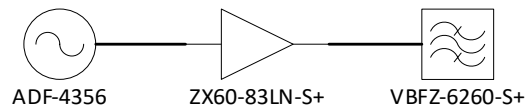


Figure 27. Local oscillator block diagram

2. Ettus B205mini-i

The power output of the B205mini was measured for a constant source and a modulated signal. The constant source GNU Radio flowgraph had a source with a magnitude of 0.707 connected directly to a USRP sink with a frequency set to 3.95 GHz. The modulated signal test used the final xband_tx.py flowgraph set to 3.95 GHz with a sample rate of 1e6. The power output for the modulated signal was measured un-amplified and amplified. (See Table 15.) The power output of the amplified signal with a gain of 65 dB would exceed the ADC limits of the N9915A and was not measured.

Table 15. Power output measurement test results for the Ettus B205mini-i

Gain setting	Peak power at 3.95 GHz (dBm)	Occupied power for 1 Mbps signal at 3.95 GHz (dBm)	Occupied power for 1 Mbps amplified signal at 3.95 GHz (dBm)
5	-70.0	In the noise	-49.1
15	-60.0	-59.6	-40.4
25	-50.5	-50.9	-30.5
35	-40.5	-40.5	-20.3
45	-30.6	-30.7	-10.4
55	-20.6	-20.6	-0.5
65	-10.5	-10.7	N/A

3. Raspberry Pi

Three versions of the Raspberry Pi were tested with the X-band payload. The versions tested were the Raspberry Pi Model 3 A+, Raspberry Pi Model 3 B+, and the Raspberry Pi Model 4 B. These tests were conducted with the Raspberry Pi connected to an Ettus B205mini-i, powered from the USB port on the Raspberry Pi, running the final transmitter GNU Radio Python program. All three versions were able to run the required software and were evaluated for the maximum data rate, which could be sustained before producing underrun errors. These results are shown in Table 16.

Table 16. Maximum data transmission rate when running xband_tx.py

Raspberry Pi version	Maximum Data Rate (bps)
Raspberry Pi Model 3 A+	1.0E6
Raspberry Pi Model 3 B+	1.0E6
Raspberry Pi Model 4 B	3.2E6

B. PAYLOAD BENCH TESTING

The X-band payload was tested as a completed sub-assembly and when integrated into the HAB bus. During sub-assembly testing the power consumed by the payload and power out to the antenna were measured. During integration testing, transmission error rates and data transfer rates were measured at a receiver radio. Integration testing also revealed thermal management issues with the HAB EPS and Raspberry Pi 4.

1. Payload Sub-Assembly Testing

Power consumption and power output tests were conducted on the payload; additionally, voltage and thermal conditions were observed.

Power usage tests were conducted on the assembled payload with both the Raspberry Pi 4 Model B and the Raspberry Pi 3 Model B+. Both models were tested at idle, and when running the transmitting program at 1 Mbps; the Raspberry Pi 3 was not tested at 2.75 Mbps since it was not capable of data transmission at that rate. The payload was connected to an Agilent E3632A DC power supply set to 5.15 V with a 3A limit. A

lead off the power supply was connected to the power cable that would plug into the HAB EPS board. The current supplied by the power supply, voltage at the power supply, voltage at the EPS connector, and voltage at the GPIO pins were recorded. The results of the power usage test are listed in Table 17. Any time that the power at the GPIO pins dropped below 5.0 V the GNU Radio program would give underrun errors. Based on the voltage drop from the EPS connector to the GPIO pins, it was determined that a voltage of approximately 5.25 V would be needed at the EPS connector.

Table 17. Payload power consumption test results

Payload	Power supply current (A)	Power supply voltage (V)	Voltage at EPS connector (V)	Voltage at GPIO pins (V)	Notes
RPi 4 – idle	1.32	5.15	5.03	4.97	-
RPi 4 – 1 Mbps	1.94	5.15	4.98	4.89	Few initial underruns
RPi 4 – 2.75 Mbps	1.92	5.15	4.98	4.89	Constant underruns
RPi 3 – idle	0.975	5.15	5.06	5.02	-
RPi 3 – 1 Mbps	1.60	5.15	5.01	4.94	Few initial underruns

After component testing, the Raspberry Pi 4 was chosen to be the computer controller for the X-band payload. However, when assembled into the payload with the PDDP attached, the Raspberry Pi 4 would reach its upper limit on processor temperature after about 3 minutes of running at a 3 Mbps data rate. Upon reaching the temperature limit, the Raspberry Pi would throttle performance and display a temperature warning. Throttling performance would be accompanied by an underrun error on the GNU Radio transmitter program and a bad data packet at the receiver. The supplied heat sinks were attached to the Raspberry Pi; this allowed about five minutes of operation before reaching the temperature limit. A reduction in data transfer rate to 1 Mbps or less was required to prevent thermal throttling. For this reason, the Raspberry Pi 3 was chosen for the final configuration.

The power output to the antenna of the assembled payload was measured in both the 8.25 and 10.25 configurations. The power output was measured for the 10.25 GHz configuration using a Keysight N9918A and for the 8.25 GHz configuration using the Keysight N9915A. Two different LO power settings were used, +2 dBm for the 8.25 GHz test and -2 dBm for the 10.25 GHz test. The gain of the B205mini was varied to produce different output power levels. The early settings for the ADF-4356 were at the +2 dBm level but were later change to -4 dBm after additional tests showed the -4 dBm setting along with an LNA produced the desired 7 dBm into the mixer. The higher LO power into the mixer did not produce a higher power out to the antenna, though both power output levels tracked linearly with B205mini gain settings until output reached ~16 dBm. Table 18 shows the power out results for the 10.25 GHz transmission and Table 19 shows the power out results for the 8.25 GHz transmission.

Table 18. Payload power to antenna at 10.25 GHz

B205mini gain setting (dB)	ADF-4356 power setting (dBm)	Channel Occupied Power (dBm)	Channel Occupied bandwidth (kHz)
5	-4	-33.4	625
20	-4	-18.9	600
35	-4	-3.4	600
45	-4	6.5	600
50	-4	11.4	600
55	-4	14.9	600

Table 19. Payload power to the antenna at 8.25 GHz

B205mini gain setting (dB)	ADF-4356 power setting (dBm)	Channel Occupied Power (dBm)	Channel Occupied bandwidth (kHz)
5	+2	-36.3	625
20	+2	-25.3	625
35	+2	-10.2	625
45	+2	-0.3	625
50	+2	4.9	625
55	+2	10.0	625
60	+2	14.1	637.5
65	+2	16.3	925

2. Payload Integration Testing

The second phase of testing occurred during the integration of the payload and the bus. These tests include fit checks, power up checks, communications checks, transmission error rate tests, data download speed tests, and an extended run test with the payload operating in its flight mode.

The initial plan used a 2U CubeSat bus. This 2U configuration had room for the payload, C&DH and EPS boards, batteries, wiring harness and power switches, and X-band antenna, but it would require external mounting of the MHX telemetry antenna and Spot tracker. Similar configurations had been used in past HAB flights [21] and was deemed acceptable for this design. Figure 28 depicts the initial planned layout for the X-band payload and HAB components in a 2U frame. Initial fit checks showed that this layout and design would work; however, the addition of the LTC 3119 power supply board and the switch from an X-band patch antenna to an X-band helical antenna required more internal space than was available. Therefore, the design was changed to a 3U frame. Figure 29 and Figure 30 are photographs of the integrated X-band payload and HAB bus in a 3U frame.

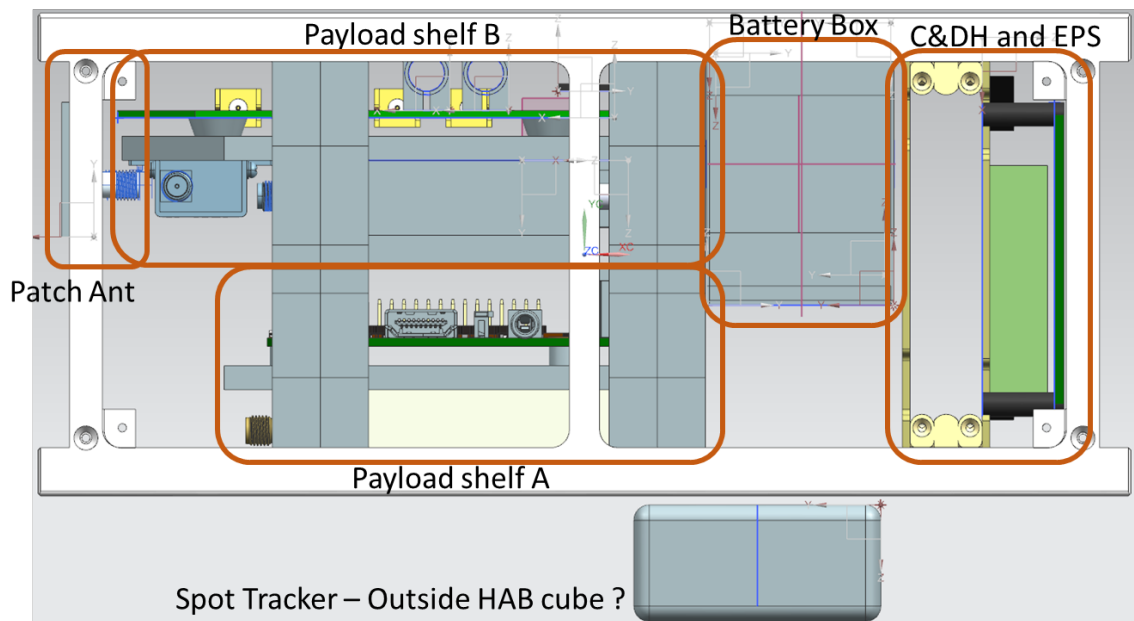


Figure 28. Initial HAB layout for the X-band payload

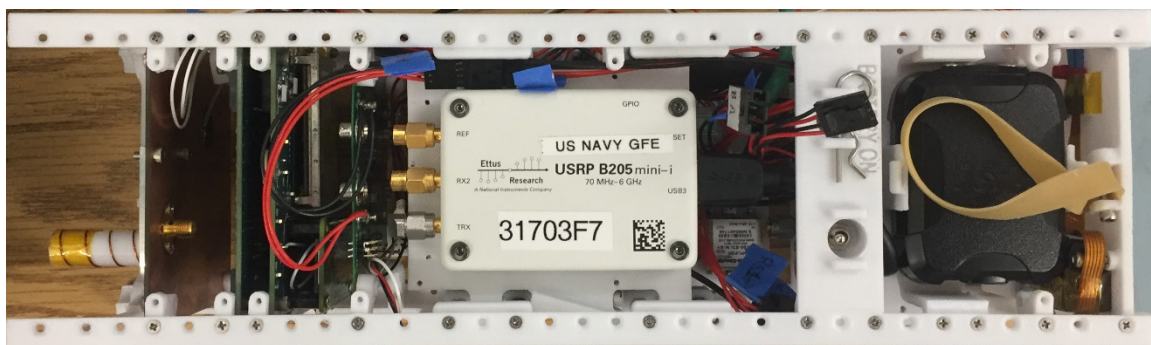


Figure 29. Assembled X-band HAB bus (view 1)

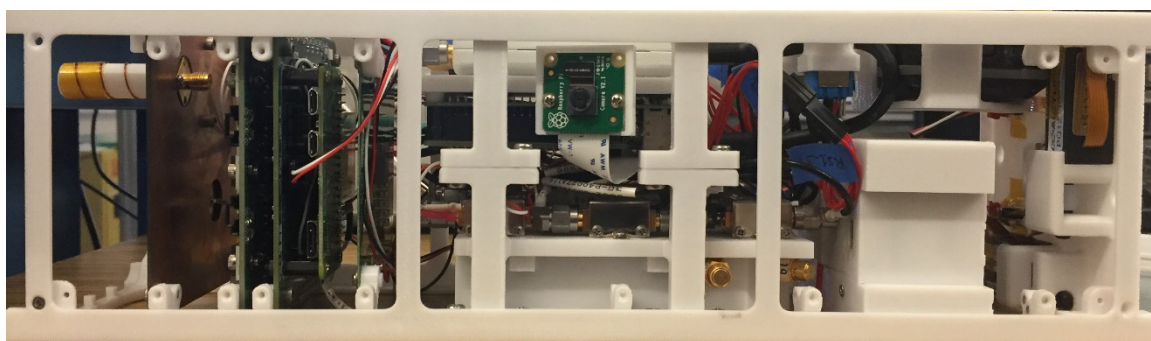


Figure 30. Assembled X-band HAB bus (view 2)

During power-up checks it was determined that the EPS could supply sufficient power out to the X-band payload when a Raspberry Pi 3 was used as the controller while running at a data rate of 1 Mbps; however, this would eventually lead to the linear regulator supplying power to the payload to overheat and shut down. This was solved by adding the LTC 3119 evaluation board as a separate power regulator for the payload. This was able to supply sufficient power to the payload without overheating, but required more space in the HAB bus frame.

Once the bus and payload were integrated and the receiver ground station was set up with a MHX telemetry radio, communications checks were conducted. The integrated payload and bus was set up in an RF isolated screen room in the SSAG laboratory to ensure only the signals transmitted from the antennas were being received. Two RF ports on the screen room were used, one for the 915 MHz telemetry signal and one for the 10.25 GHz X-band downlink signal. The 915 MHz line was connected to a vertical whip antenna and the 10.25 GHz line was connected to a Hewlett Packard 8495B variable attenuator, then to a Pasternack PE9858 10 dB gain horn antenna. The ground station had identical antennas to receive both signals. The ground station down converted the 10.25 GHz signal in a similar manner to the up conversion done by the payload. Figure 31 shows the amplifiers, mixer, and filter used for down conversion at the ground station. A Keysight N5173B is used as a ground station local oscillator; it was set to -10 dBm amplitude and 6.25 MHz frequency. The output of the down conversion was a 4 GHz signal sent to an Ettus B205mini-i connected to a Dell latitude E6540 laptop running Ubuntu operating system. A good telemetry link was established on 915 MHz between the ground station and the bus; command, data, and system status messages were passed. Additionally, a good X-band signal was received and decoded.

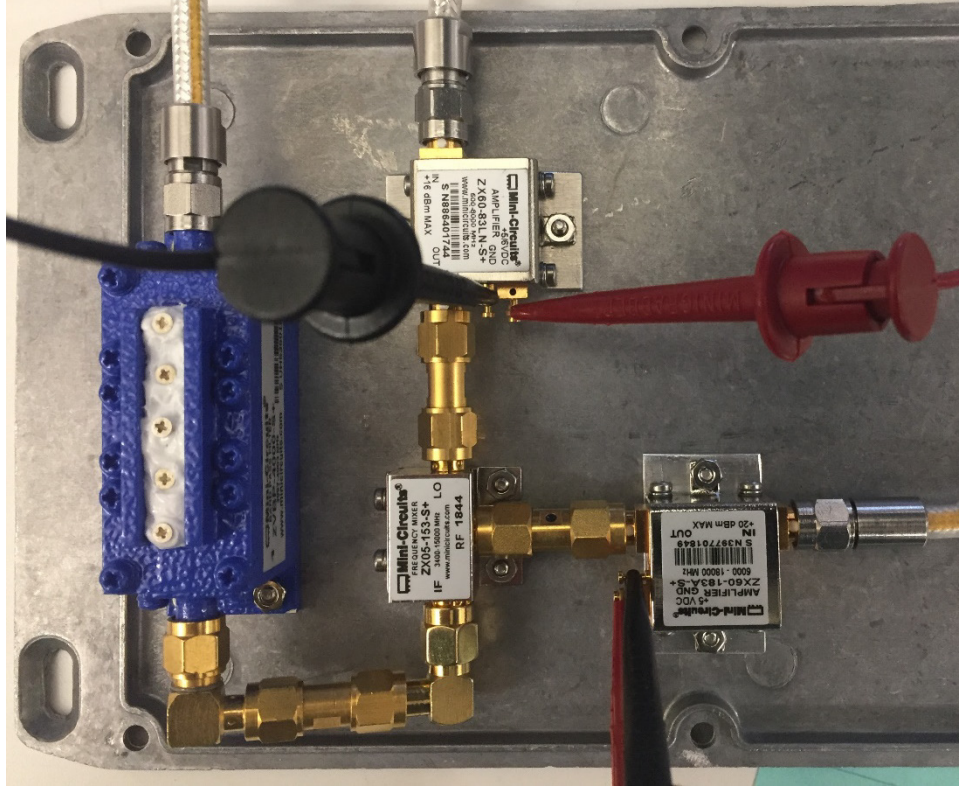


Figure 31. Ground station down conversion hardware

The next tests conducted determined the packet error rate, used to estimate bit error rate, and a download speed test. Three different connection configurations were used during these tests. First, the transmitter and receiver B205minis were directly connected using an RF cable at a transmission frequency of 3.95 GHz. Second, the full X-band payload was connected to the ground station down conversion equipment via RF cable in place of transmit and receive antennas using a frequency of 3.95 GHz. A 10.25 GHz signal was sent over the RF cable using the full payload flight hardware and ground station hardware, with the exception of the transmit and receive antennas. The third configuration was to place the integrated payload and bus into the screen room, connect through the room's RF ports to antennas, and to attenuators in the case of the X-band signal. This third configuration allowed for an over-the-air transmission test with the ability to simulate larger distances by adding additional attenuation.

Packet error rate tests were conducted using the test mode of the chunking.py program. In this mode, the transmitter sends a series of packets, where each packet consists

entirely of the packet number repeated to create a full 1024-bit packet. The receiver mode reads each packet. If every number in the packet matches, then the packet is counted as good. If a single number in the packet is different from the rest, then the packet is not counted. The receiver starts its count with the first packet received, and counts the number of good packets. The total packet count is the difference in the number of the first packet received and the latest packet received. These values are displayed along with a percentage of correct packets calculated from total and correct packets.

Packet error rate and download speed were conducted with a direct RF cable connection at 3.95 GHz. The test was conducted with a low power gain setting for both the transmitter and receiver B205mini. The transmitter was set to a gain of 20 dB and the receiver was set to a gain of 25 dB; additionally, 10 dB of attenuation was added inline. The transmission rate for both tests was 1 Mbps. Table 20 shows the results of the packet error rate test for a direct connection at 3.95 GHz. The average packet error rate was 5.53×10^{-4} . The bit error rate cannot be directly computed, since a bad packet can have anywhere between 1 and 1024 bad bits. For this test, it is assumed that a bad packet has one bad bit. The average extrapolated bit error rate was 5.40×10^{-7} .

The time to download a file was conducted with the transmitter and receiver in their normal modes. JPEG format picture files taken by the payload camera were transmitted; the average file size was 491 kB. The time to download a file was manually timed. A stopwatch was started when the transmitter program first started a file and then stopped when a “file complete” message was received by the transmitter. The results of the file download speed test are listed in Table 21. The average time to download a file was 22.19 s, or an average download speed of 196 kbps.

Table 20. Packet transfer error rate at 3.95 GHz

Data Packets Sent	Data Packets Received	Dropped Packets	Packet Error Rate	Extrapolated Bit Error Rate
20414	20410	4	1.96E-04	1.91E-07
20376	20362	14	6.87E-04	6.71E-07
20285	20276	9	4.44E-04	4.33E-07
21691	21682	9	4.15E-04	4.05E-07
20315	20312	3	1.48E-04	1.44E-07
20938	20929	9	4.30E-04	4.20E-07
32647	32637	10	3.06E-04	2.99E-07
20365	20349	16	7.86E-04	7.67E-07
20253	20223	30	1.48E-03	1.45E-06
20296	20283	13	6.41E-04	6.26E-07
Average:			5.53E-04	5.40E-07

Transmitter gain: 20 dB; Receiver gain: 25 dB; in line attenuation: 10 dB; transmission rate: 1 Mbps; transmitter and receiver connected via cable.

Table 21. Time to download files at 3.95 GHz

File size (kB)	Number of packets	Time to download (s)	Effective data rate (kbps)
491.0	4335	22.73	190.7
491.8	4342	21.53	201.7
491.3	4337	20.49	211.7
490.9	4334	23.46	184.7
490.0	4326	23.64	183.0
490.1	4327	19.84	218.1
490.9	4334	24.71	175.4
490.9	4334	22.00	197.0
490.8	4333	20.01	216.5
490.7	4332	23.49	184.4
Average:			196.3

Transmitter gain: 20 dB; Receiver gain: 25 dB; in line attenuation: 10 dB; transmission rate: 1 Mbps; transmitter and receiver connected via cable.

The packet error rate tests were repeated at 10.25 GHz. For this test, the flight configuration up-conversion hardware was included in the transmitter. The ground station used the hardware configuration that was used for the flight test. First, five trials of the test mode of chunking.py were run with a direct RF cable connection that included 10 dB on attenuation. Additionally, both the receiver and transmitter had their gain reduced from the

flight setting. The transmitter was set for 25 dB of gain and the receiver was set for 20 dB of gain. The results for the five trials are listed in Table 22; the average packet error rate was 1.15×10^{-4} , which gives an extrapolated bit error rate of 1.12×10^{-7} .

Table 22. Packet transfer error rate at 10.25GHz with direct cable connection w/ 10 dB of attenuation

Data Sent	Packets Received	Dropped Packets	Packet Error Rate	Extrapolated Bit Error Rate
20282	20280	2	9.86E-05	9.63E-08
20282	20280	2	9.86E-05	9.63E-08
32956	32953	3	9.10E-05	8.89E-08
20262	20258	4	1.97E-04	1.93E-07
22542	22540	2	8.87E-05	8.66E-08
Average:			1.15E-04	1.12E-07

Transmitter gain: 20 dB. Receiver gain: 25 dB. In line attenuation: 10 dB, transmission rate: 1 Mbps; transmitter and receiver connected via cable.

The same hardware configuration was tested with varying amount of attenuation in the line. The variable attenuation simulates different losses between the receiver and transmitter. These losses include point error and free space path loss (FSPL). This test was conducted with the low power mode used in the previous test and the flight gain settings with additional attenuation. Table 23 lists the results for the low gain setting tests, and Table 24 lists the results for the flight gain setting test.

Table 23. Packet transfer error rates at various attenuations for 10.25 GHz (low gain)

Attenuation (dB)	Data Packets Sent	Data Packets Received	Dropped Packets	Packet Error Rate	Extrapolated Bit Error Rate
20	20284	20284	0	0.0	0.0
30	20299	20298	1	4.93E-05	4.81E-08
40	20253	20249	4	1.98E-04	1.93E-07
50	20207	20200	7	3.46E-04	3.38E-07
60	20203	4476	15727	7.78E-01	7.60E-04
70	N/A	0	N/A	1.0	1.0

Transmitter gain: 20 dB. Receiver gain: 25 dB. Data transmission rate: 1 Mbps; transmitter and receiver connected via cable.

Table 24. Packet transfer error rates at various attenuations for 10.25 GHz
(high gain)

Attenuation (dB)	Data Packets Sent	Data Packets Received	Dropped Packets	Packet Error Rate	Extrapolated Bit Error Rate
55	20369	20366	3	1.47E-04	1.44E-07
65	20224	20219	5	2.47E-04	2.41E-07
75	20477	20476	1	4.88E-05	4.77E-08
85	20277	20275	2	9.86E-05	9.63E-08
95	20250	20246	4	1.98E-04	1.93E-07
105	N/A	0	N/A	1.0	1.0

Transmitter gain: 50 dB. Receiver gain: 50 dB. Data transmission rate: 1 Mbps; transmitter and receiver connected via cable.

Table 25. Equivalent free space path loss for cable connected transmitter
and receiver

Attenuation	Equivalent free space path loss (m)
20	0.23
30	0.74
40	2.3
50	7.4
55	13
60	23
65	41
70	74
75	130
85	410
95	1,300
105	4,100

Assumes a 10 dB gain antenna would be used by both transmitter and receiver.

The last lab test conducted was with an over-the-air transmission from the transmitter and receiver. The transmitter was placed in the RF screen room. Both the transmitter and receiver were connected to a PE9858 10 dB gain horn antenna; the antennas were separated by 4.1 m. Various amounts of attenuation were added to the transmitter side prior to the antenna. Table 26 lists the resulting packet error and extrapolated bit error rate

for the amount of attenuation added. Table 27 lists the equivalent FSPL in km for the amounts of attenuation added to the transmission. Figure 32 shows the packet error rate vs. the amount of attenuation. The relatively flat curve of the packet error rate to attenuation and therefore FSPL is unexpected. Bit error rate and therefore packet error rate should increase as attenuation increases. This trend is not seen in this data. Additionally, there is an unexpected better packet error rate at an attenuation of 145 dB with no explanation. Due to these to circumstances, this data should be carefully considered and should be retested.

While the test results presented in Table 26 and Figure 32 were the most surprising, all packet error tests had a relatively constant packet error rate as attenuation was increased until an attenuation threshold was reached. At this attenuation threshold, packet error rate climbed sharply until reaching a point where no packets were transferred error free. Table 23 and Table 24 both show the trend of a drop off at increasing attenuation.

Table 26. Packet transfer error rate at 10.25 GHz over-the-air at various attenuations

Attenuation (dB)	Data Packets Sent	Data Packets Received	Dropped Packets	Packet Error Rate	Extrapolated Bit Error Rate
50	20372	19952	420	2.06E-02	2.01E-05
60	20307	20148	159	7.83E-03	7.65E-06
70	20367	19997	370	1.82E-02	1.77E-05
80	20793	20227	566	2.72E-02	2.66E-05
90	20338	19778	560	2.75E-02	2.69E-05
99	20111	19466	645	3.21E-02	3.13E-05
108	20161	19415	746	3.70E-02	3.61E-05
117	20768	19881	887	4.27E-02	4.17E-05
121	20686	20320	366	1.77E-02	1.73E-05
125	20695	20146	549	2.65E-02	2.59E-05
135	20305	19731	574	2.83E-02	2.76E-05
145	20259	20247	12	5.92E-04	5.78E-07
155	21862	21522	340	1.56E-02	1.52E-05

Transmitter gain: 50 dB. Receiver gain: 50 dB. Data rate: 1 Mbps

Table 27. Equivalent free space path loss for over-the-air lab test

Attenuation (dB)	Simulated range (km)
50	1.3
60	4.1
70	13.
80	41
90	130
99	360
108	1,000
117	2,900
121	4,600
125	7,300
135	23,000
145	73,000
155	230,000

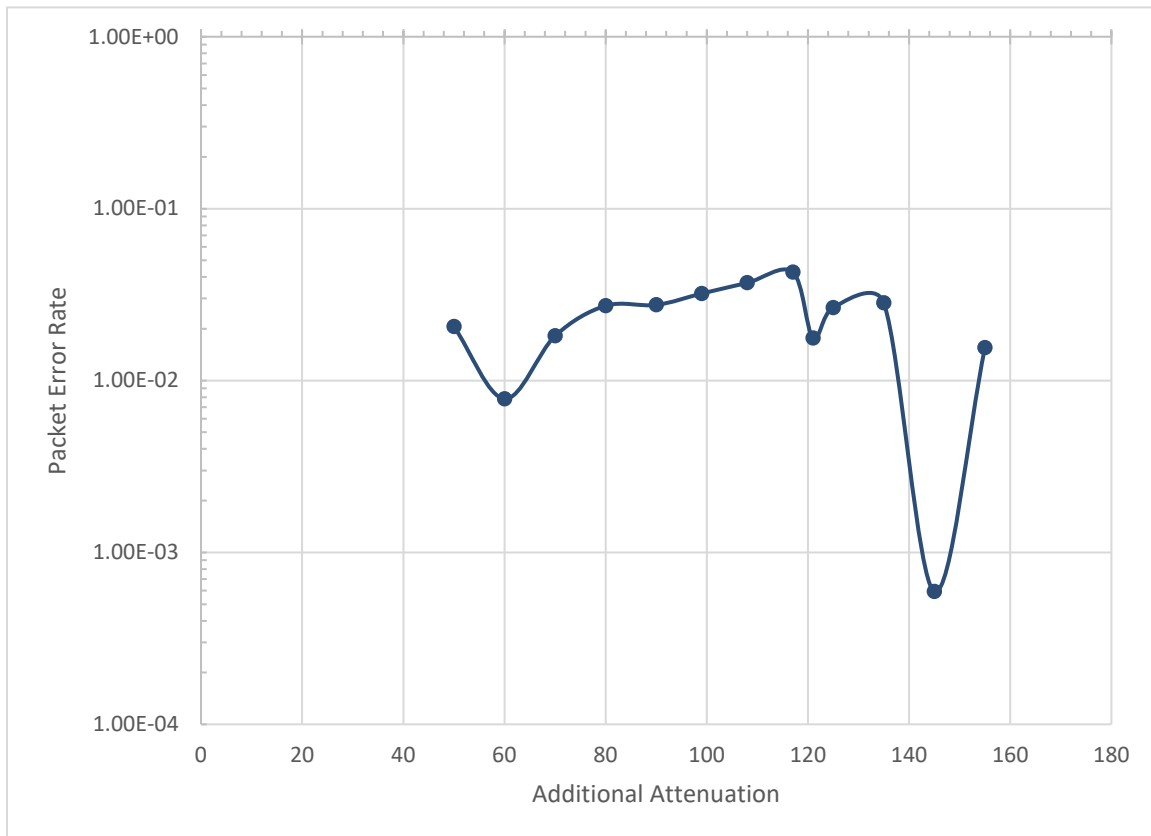


Figure 32. Over air transmission packet error rates with additional attenuation

C. FIELD TESTING

Two field tests were conducted. In the first, the ground station and assembled HAB were set up in the NPS courtyard. In the second, the X-band payload and bus were flown on a high altitude balloon flight.

1. Outdoor Testing

The outdoor ground testing in the NPS courtyard used the full HAB payload and the full ground station. The HAB was set up approximately 26 m (85 ft) from the ground station with a pointing offset of approximately 10 degrees. The chunking program was run in its test mode and five cycles of approximately 20,000 packets were received on each trial. A sixth trial was attempted; however, the battery used for the ground test died during the middle of the trial. The results are listed in Table 28.

Table 28. Packet transfer error rate for outdoor testing

Data Packets Sent	Data Packets Received	Dropped Packets	Packet Error Rate	Extrapolated Bit Error Rate
20303	20299	4	1.97E-04	1.92E-07
27350	27348	2	7.31E-05	7.14E-08
22611	22605	6	2.65E-04	2.59E-07
20302	20302	0	0	0
20445	20442	3	1.47E-04	1.43E-07
11905	11905	0	0	0
Average:			1.14E-04	1.11E-07

Transmitter gain: 50 dB; receiver gain: 20 dB; 26 m separation; 10 degree pointing error

Startup of the HAB payload was verified by powering off the bus and X-band payload, then turning on both power switches. Startup of all components was verified by visual inspection of lights on the C&DH, EPS, Raspberry Pi, and B205mini. Automatic start of transmission was verified by starting the ground station and observing the receipt of signal, data packets, and finally, successful transfer of files. All of this occurred without manually commanding the HAB.

2. Flight Test

One flight test was planned for the X-band transmitter payload on a high-altitude balloon. The flight's intent was to test the real-world capability of the X-band transmitter designed and built for this project.

a. Flight Test Concept of Operations

The flight test of the X-band transmitter payload was planned for a 30-minute flight. The launch location would be chosen based on weather and winds in order to allow the HAB payload to land in the Salinas valley. The balloon burst calculator and flight path predictor at habhub.org [40] would be used for flight predictions. Multiple launch locations and release altitudes were evaluated with the habhub tools. A launch from the Chualar RC Modeler's Flight Club was chosen with a balloon release altitude of 5000 m. Figure 33 is a Google Earth [41] image of the launch site. With a 5000 m balloon release and parachute deployment, the approximate flight time would be 30 minutes and the HAB payload would land in the Salinas valley. If prior to reaching an altitude of 5000 m the X-band data transmission signal was lost at the ground station, then the balloon release command would be sent.



Figure 33. Google Earth image of the HAB launch site near Chualar, CA.
Source: [41].

On the day of the flight, a ground station would be set up at the launch point with an X-band receiver, a balloon filling station, and a COSMOS telemetry station. COSMOS is a telemetry, and command and control open-source software package developed by Ball Aerospace. Additionally, one chase car with a COSMOS telemetry station would start at

the launch site, and after launch, drive to the predicted landing site for HAB payload recovery. After ground station setup, a functional test would be conducted on all systems. Once all functional tests were completed, the balloon would be filled and the HAB payload would be attached. After a final review of the Go/No-Go criteria the HAB would be launched.

During the HAB flight, the X-band payload would capture and store images in a designated directory on its SD card. In the event of a camera failure or malfunction, eight files were preloaded into the directory where the captured pictures were to be stored by the camera. These eight files were ASCII text files renamed to follow the naming convention used by `xband_camera.py`. The payload would transmit stored files to the ground station in numerical order. The bus of the HAB payload would provide GPS data to the COSMOS stations and respond to commands (such as balloon release and parachute release) sent by the COSMOS operators. Once the flight was complete and the HAB payload was on the ground the chase team would recover the payload.

The HAB flight was planned for 29 Aug with a release at 0900. The expected ascent rate was 4.1 m/s, with a balloon release at an altitude of 5000 m. Following balloon release the parachute release would be commanded. The expected landing point was in the vicinity of N 36.61, W 121.50. If the balloon did not release, then the expected burst location and subsequent landing point were both in the vicinity of N 36.59, W 121.36. Figure 34 shows the planned flight path and Figure 35 shows the expected flight path if the balloon did not release.

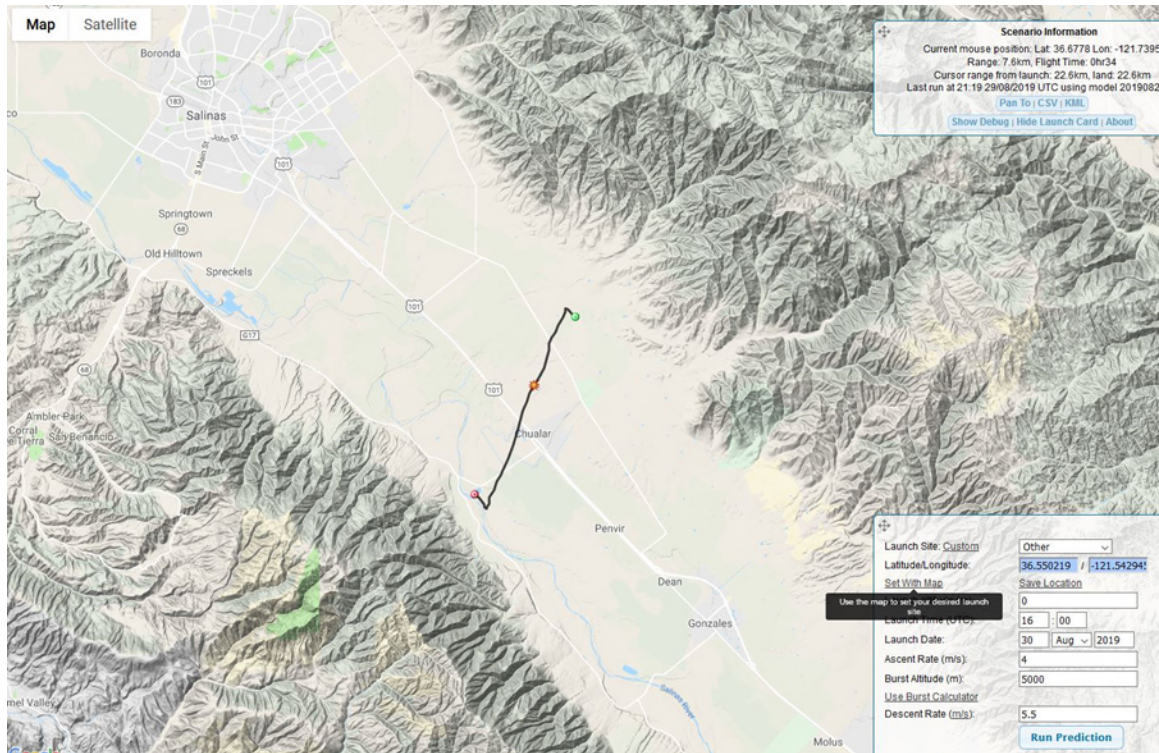


Figure 34. Planned HAB flight profile. Source: [40].

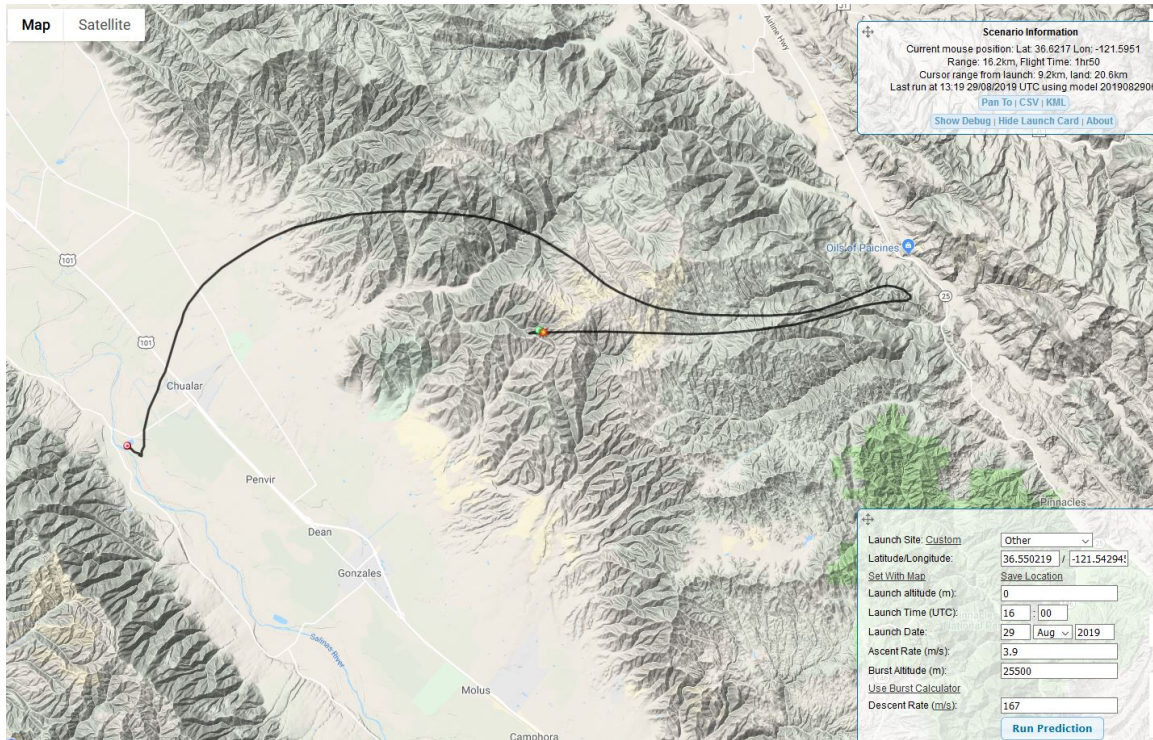


Figure 35. HAB flight profile without balloon release. Source: [40].

b. Flight Test Results

On the morning of 29 August, final weather checks, HAB flight prediction checks, and HAB payload systems checks were made, and a pre-flight brief was conducted prior to departing NPS. After arrival at the Chualar RC modelers club site, the ground station and chase car COSMOS were setup. The HAB payload bus and X-band payload were powered up at 0828. All systems for the HAB payload bus and X-band payload, chase car COSMOS system, and the ground station were turned on and tested. All systems checked out nominal. Next, a Howyee 600g balloon was filled with one tank, approximately 110 m³, of helium. Next, the HAB payload was attached to the balloon using #18, braided, nylon mason line and 50 lb. test, spectra fiber, microfilament, braided fishing line as a break-away line. After confirming all go criteria, the HAB was released at 0842.

Good telemetry and data download were established and the download of files was observed. The ground station X-band receiver antenna was manually pointed at the balloon. On its ascent, the balloon passed through a cloud layer at flight time 00:03:00 (just prior to

the fourth picture taken in flight, 20th picture taken after payload power up). At this point in time, manual pointing of the antenna became unreliable and the X-band downlink signal was lost. An update to the antenna pointing was done by guessing based on reported GPS location, but down link was not reestablished. Three files were successfully downloaded, totaling 3.21 MB or 25.6 Mb, prior to the loss of the downlink signal. All three files were of the preexisting files saved prior to flight and no photographs taken during flight were downloaded.

After a sustained loss of downlink signal, the decision was made to release the balloon prior to the HAB reaching an altitude of 5000 m, since data reception had ended and no further testing could be done. A check of the GPS by the COSMOS operator showed that the HAB was approaching U.S. Highway 101 and the decision was made to release the balloon after crossing the highway to avoid the potential of dropping the payload onto the highway. Once confirmation was received that the HAB had crossed U.S. Highway 101, balloon release was commanded and parachute release were commanded.

After initially thinking that the HAB had landed and sending the chase car to a reported GPS point, it was realized that the HAB was still ascending. This indicated that the balloon had not released from the HAB payload. The release command was sent multiple times and indications of balloon release were returned via HAB bus telemetry; however, the HAB kept rising. A check of the reported altitudes and times for those reports indicated that the HAB was rising at a rate of 2 m/s, not the expected 4 m/s. Good telemetry was received until 10:46:42 PDT, at which point, the battery voltage dropped below 5 V and the HAB bus and payload started experiencing intermittent power cycles from resets caused by the low voltage. The last good transmission from the HAB reported an altitude of 23,114 m at a time of 10:59:51 PDT. The last logged GPS fix prior to shut down was at N 36.6279, W 121.0251, at an altitude of 20,760.5 m, at a time of 10:57:14 PDT. No further indications were received via telemetry. At approximately 1140 PDT, Spot tracker indications were received indicating that the HAB payload was on the ground at a location of N 36.6346, W 121.2683. This location was found and the property owners contacted to allow the recovery of the HAB payload on the next day. Figure 36 show a Google Earth image of the location where the HAB payload was recovered.

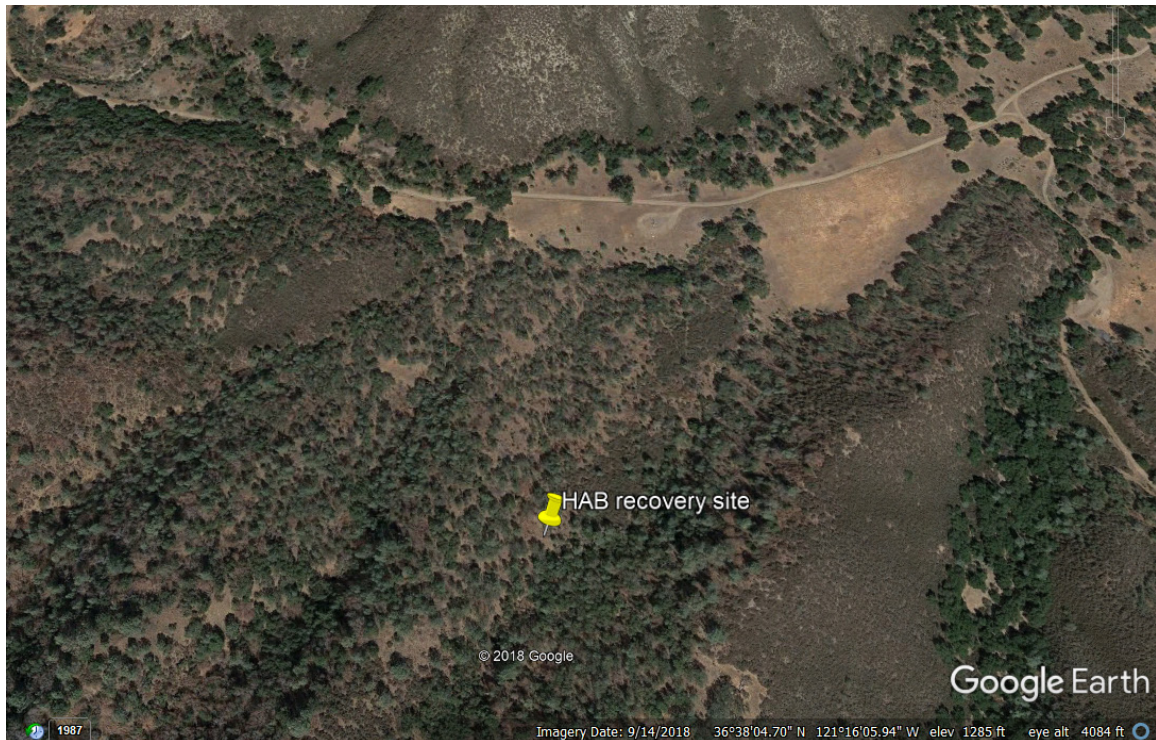


Figure 36. Google Earth image of HAB recovery site. Source: [41].

On 30 August, the remains of the HAB payload were recovered. Photographic evidence of the landing point suggests that the HAB payload experienced an unplanned, rapid disassembly upon landing. Figure 37 shows the debris field of the HAB payload at the landing site, a 120 cm hiking stick is provided for scale reference. The main parachute is on the right side of the photograph in Figure 37. Additionally, Figure 37 shows the balloon line wrapped up with the parachute lines. This wrapping of lines prevented the parachute from fully inflating, leading to a rapid descent after balloon burst. Inspection of the debris indicates that the balloon release mechanism actuated but the balloon line remained caught on the plastic sleeve around the actuator post. This sleeve was too big to fit through the hole in the top of the structure that the balloon line ran through. This caused the balloon line to remain caught on the CubeSat structure and prevented the release of the balloon from the payload. Figure 38 shows the balloon line caught on the actuator sleeve at recovery. It is expected that the parachute deployed on command but that on descent the balloon lines tangled the main parachute and prevented the attainment of a full parachute canopy. The backup parachute release was controlled by the barometric altimeter in the

Jolly Logic parachute release mechanism; the highest setting available was 1000 ft. The HAB payload crashed in to the hills with an elevation of 1285 ft, prior to the deployment of the backup parachute. Figure 39 and Figure 40 show the condition of the recovered HAB payload.



Figure 37. Photograph of the HAB payload at the landing site



Figure 38. Balloon harness line and actuator sleeve caught on CubeSat frame

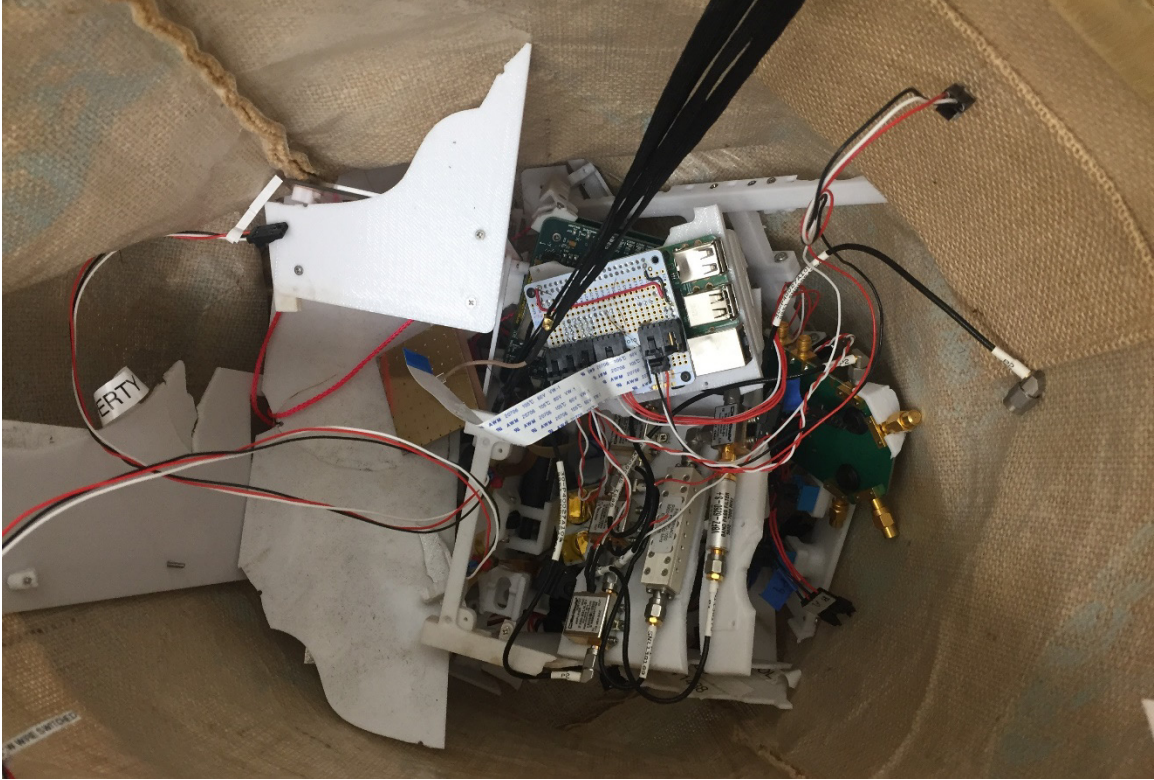


Figure 39. Recovered HAB payload

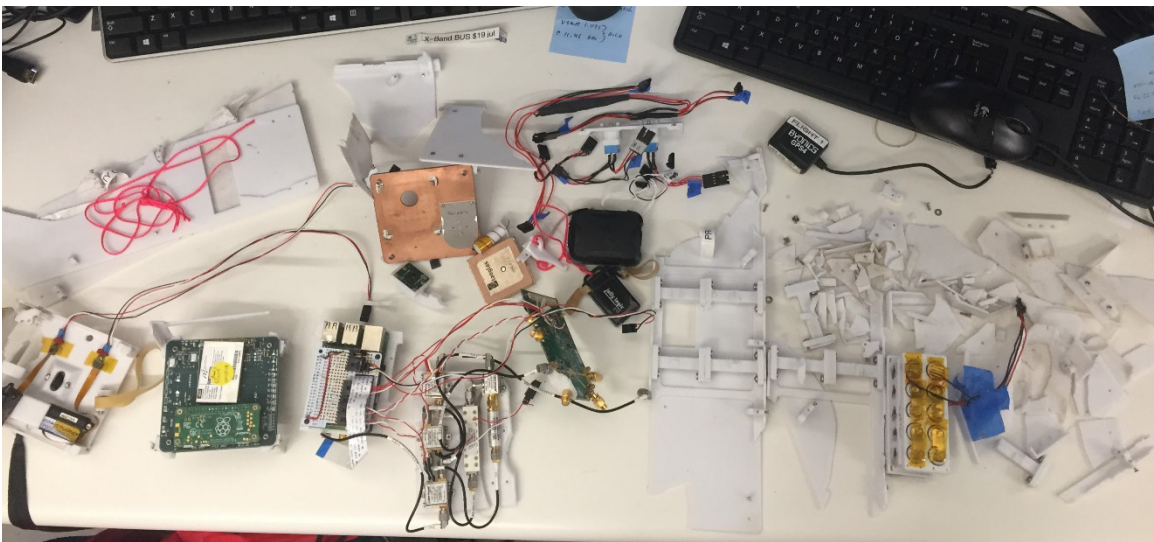


Figure 40. Recovered HAB payload on desk

The SD cards from both the HAB bus and the X-band payload were recovered and data was downloaded for analysis. Data on the bus contained both C&DH logs and GPS data logs up to bus shutdown. GPS data was used to construct a flight path for the HAB. The GPS data saved to the HAB bus computer had multiple points that had bad information. Figure 41 shows all of the GPS data imported into Google Earth. It can be seen early in the flight that there were multiple inconsistencies while over the Salinas Valley. Figure 42 shows a closer look at the inconstant GPS data over the Salinas Valley. However, looking at Figure 42, it can be seen that the upper points in the data could be connected to make a smooth path. Figure 43 shows a top down view of the GPS data points.

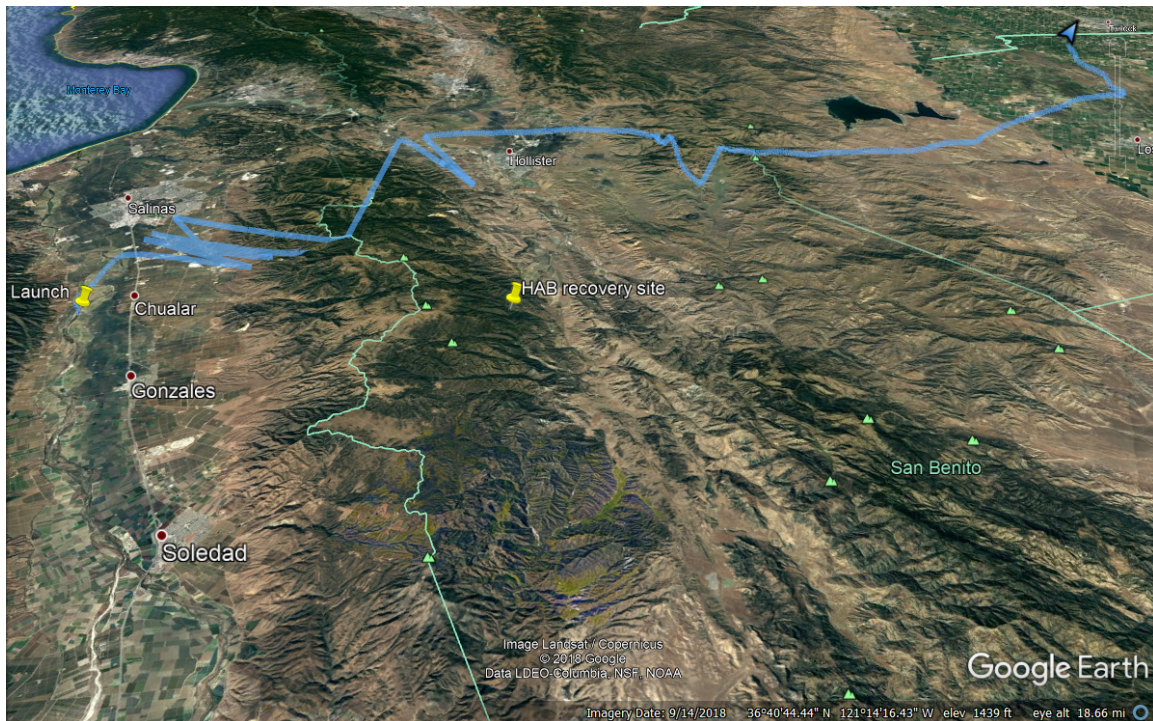


Figure 41. HAB GPS data with launch and recovery sites in Google Earth.
Source: [41].

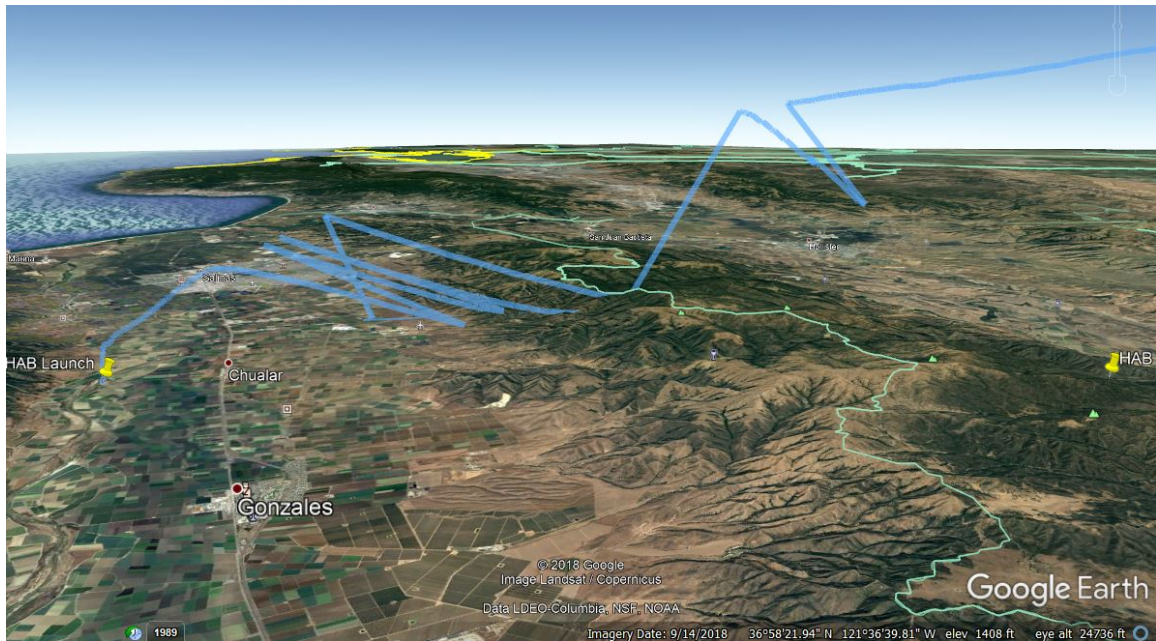


Figure 42. HAB GPS data over Salinas Valley. Source: [41].

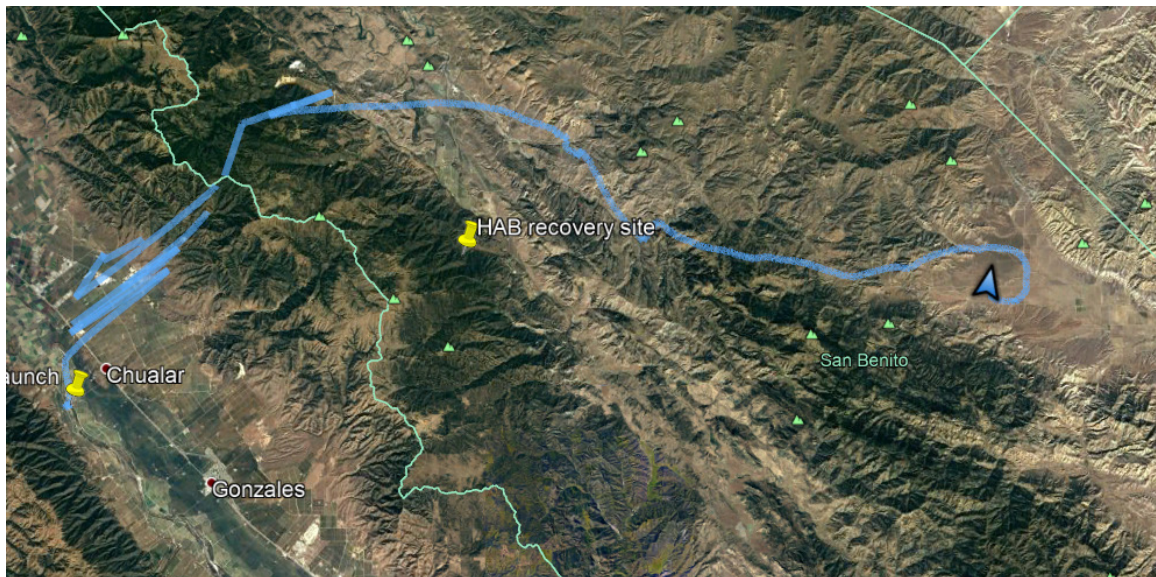


Figure 43. Top down view of HAB GPS data. Source: [41].

Figure 44 show the GPS altitude recorded by the HAB bus plotted against the GPS time for those GPS reports. Based on the altitudes in Figure 44 and path in Figure 42, the GPS data between 08:55-09:31 PDT and 09:54-10:00 PDT is assumed to be bad. No

conclusions have been made to what caused the bad GPS data. The times of bad GPS information correspond to the times that repeated balloon release and parachute release actuation commands were sent to the HAB bus.

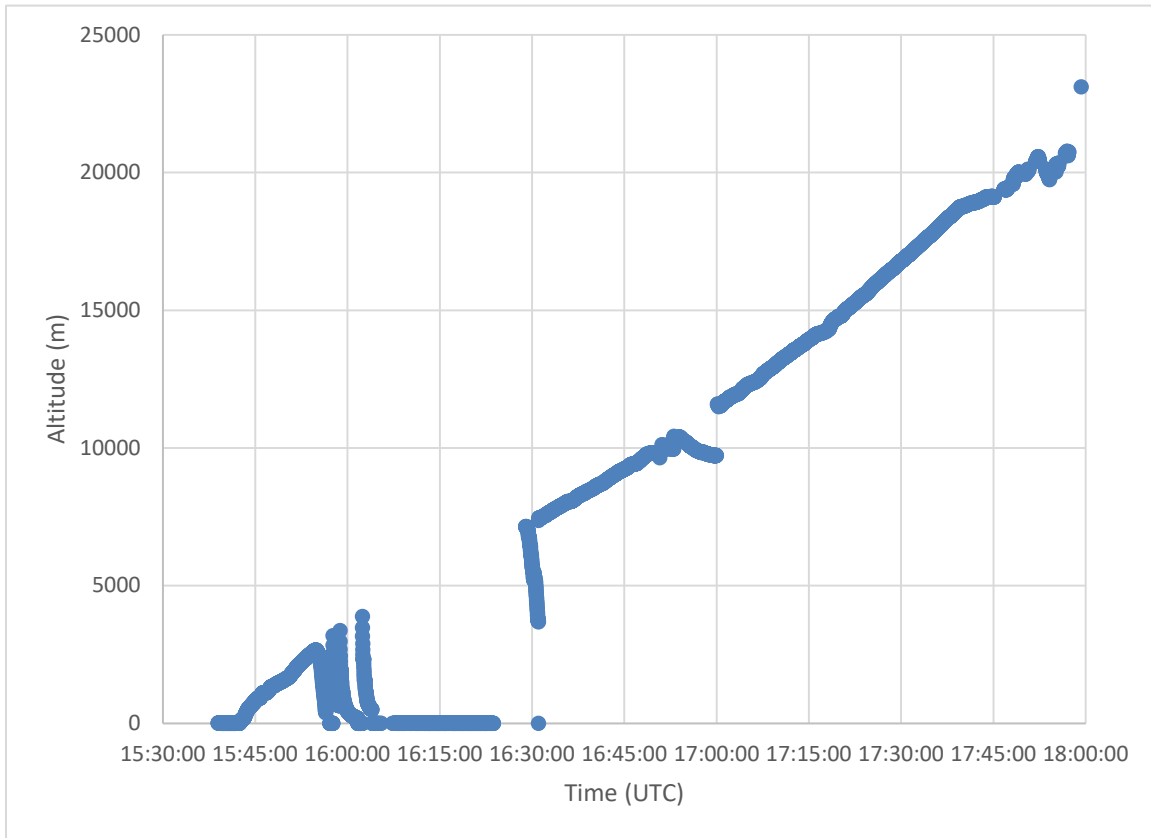


Figure 44. GPS altitude versus GPS time for HAB flight

The X-band payload had captured 181 pictures during the test from power-up until the battery voltage dropped below the level necessary to power all of the equipment. Sixteen of those pictures were taken prior to launch with one saving incorrectly; 165 pictures were taken after launch with the last one being an incomplete file. Figure 45 is the first picture taken after the HAB launch. Figure 46 and Figure 47, respectively, are sequential pictured taken prior to and after exiting the cloud layer. Figure 48 is the final complete picture taken prior to loss of power.



Figure 45. First picture captured after launch (#17)



Figure 46. Last picture prior to entering cloud layer (#19)



Figure 47. Picture taken exiting cloud layer (#20)

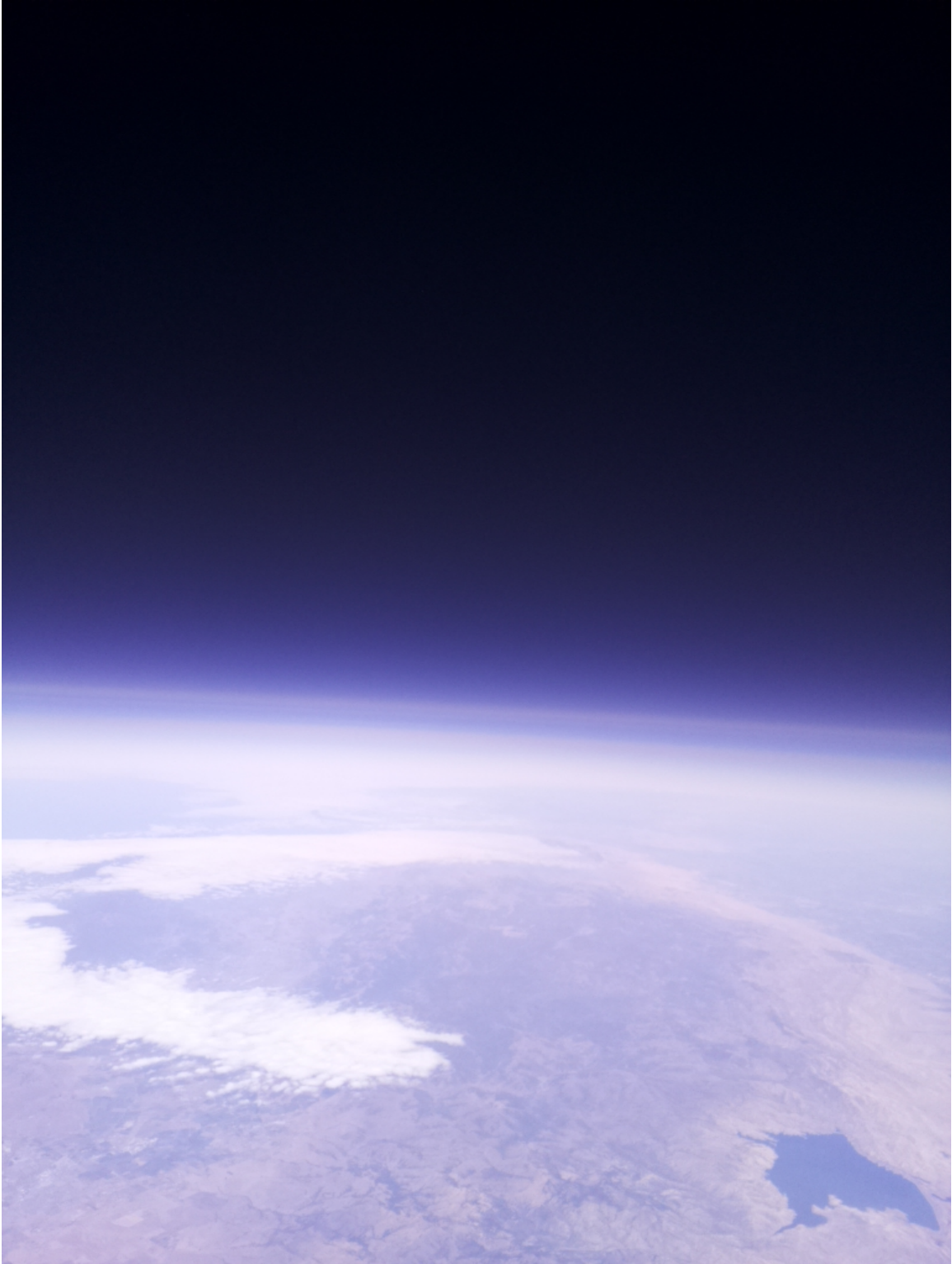


Figure 48. Final complete picture taken before power down (#180)

This project set out to meet the goals listed in Table 3. The results are compared to those objectives in Table 29. All but one of the thresholds were met; unfortunately, the volume constraint of 1U could not be met with the hardware used. The volume required was constrained by unmodified COTS components. Two components required more space than available in a 1U constraint. The frequency synthesizer card was longer than 100 mm and exceeded the 100 mm constraint. Additionally, the Ettus B205mini would also exceed 100 mm in length. While the B205mini itself is less than 100 mm in length, when the RF cable connector was attached to one end and the USB cable was connected to the other end, the combined length exceeded the 100 mm offered inside a 1U volume.

Table 29. X-band CubeSat transmitter requirements and results

Requirement	Threshold	Objective	Result	Note
Data Rate	300 kbps	1 Mbps	1 Mbps	Met Objective
Size	1U	0.5U	1.1 U	Did not meet Threshold
Mass	1 kg	0.5 kg	0.65 kg	Met Threshold
Power	15 W	15 W	11.25 W	Met Threshold
Data transfer in flight	1 file	All photos taken during flight	3 files	Met Threshold

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

A. CONCLUSIONS

This project set out to design, produce, and demonstrate a low-cost, high data rate, X-band transmitter for use in a CubeSat platform. This thesis demonstrated that an X-band transmitter for CubeSats could be produced for a cost an order of magnitude less than those that are commercially available. High data rate transmission was demonstrated in the laboratory, during ground testing, and during flight testing.

Laboratory testing demonstrated that an attenuated signal designed to match that of a CubeSat in orbit can be received and decoded. Laboratory testing also showed that there are still issues in either the receiver GNU Radio program or the chunking.py program. Text data was transmitted at a significantly lower error rate than JPEG files. When sending ASCII text using any of the three tested methods, either with GNU Radio directly reading a text file, through the chunking.py program test mode, or through the chunking.py normal transmit and receive mode using plain text files that had been renamed with .jpg extensions, data was transferred with very low error rates. However, when JPEG files were sent, select data packets would repeatedly fail to download correctly, requiring those data packets to be resent repeatedly. These data packets failed CRC checks the same way with each transmission indicating that there is a problem in the transmission programming that mishandles certain data. This caused significantly longer download times for JPEG files compared to ASCII text files.

The over-the-air in laboratory testing results are significantly better than expected and showed no correlation between attenuation and BER. These results are presented in Table 27 and Figure 32. The change in attenuation had no impact on error rate which was expected and the low error rate at high levels of attenuation (and therefore, large FSPL distances) is significantly better than expected for this project. It is recommended that this result be reexamined and the FSPL test be conducted again.

Flight testing was limited by two major items. First, the beam pattern of the transmission antenna, and second, the tracking ability of the ground station antenna. The

transmission antenna had a calculated beam width of 42 degrees; this is a relatively narrow beam width for an application that does not have a capability to point the antenna. An omnidirectional antenna or hemispherical antenna would have been a better choice for a HAB application. This would allow for farther down-range travel before losing signal and allow for better maintenance of signal when the HAB payload bounced around in the winds during its flight. The second limitation was the manual pointing of the ground station antenna. Once the HAB ascended through the cloud layer, visual tracking was no longer possible. Best guesses were made for antenna pointing, but the signal was quickly lost. HAB flight should be made on clear days that permit visual tracking of the HAB, or a tracking system should be included for more precise pointing of the ground antenna.

The Ettus B205mini with a Raspberry Pi controller combined with the RF hardware used in this project make an effective low-cost transmitter capable of functioning across the full range of the X-band. The digital signal processing done in GNU Radio proved to be effective at sending and receiving data at the desired data rates.

B. FUTURE WORK

Multiple areas for future work were noted during this project. Areas for improvement were noted for both hardware and software.

1. Hardware Improvements

Multiple potential hardware improvements were noted during this project. This includes updates for the HAB bus and recommendations for a follow on X-band CubeSat transmitter design.

a. Remote Shutdown of the Payload

A method to remotely shut down the payload needs to be maintained. When the payload was powered through a separate power supply (the LTC 3119), the payload power on and off circuit of the C&DH board was circumvented. When the ground station lost track of the payload and could no longer maintain a downlink, the payload should have been shut off to reduce power consumption. This additional power consumption exceeded the capacity of the installed battery for the duration of the flight, and the bus and X-band

payload lost power and shut down prior to reaching the burst altitude of the balloon. The additional data from the HAB GPS would have been useful in finding the payload on the ground and in recording its flight profile.

A circuit was built that would have used the payload power switch from the C&DH to turn the power to the X-band payload on and off; however, there was not sufficient time to integrate the switch into the payload.

b. Faster Onboard Computer

The use of the Raspberry Pi 3 to control the Ettus B205mini caused a data rate limit of 1 Mbps. Work in the lab showed that using a faster computer would allow higher data rates. The Raspberry Pi 4 was demonstrated to deliver three times the data rate, up to 3.1 Mbps. Integration of the Raspberry Pi 4 will require a thermal management solution, due to overheating issues when it was put into the payload.

c. Heat Dissipation

Without a thermal management solution in place, the processor temperature of the Raspberry Pi 3 approached its limit during transmission operations, the Raspberry Pi 4 quickly reached its upper temperature limits, and throttled its performance to compensate. In order to use the Raspberry Pi 3 in a space environment or to use the Raspberry Pi 4 in any environment, a method of transferring heat away from the computer processors will have to be found.

d. Integrate RF Components onto a Single Custom PCB

The RF components used to up-convert the output of the SDR into the X-band took up over half the volume and mass of the X-band transmitter. The use of a custom designed printed circuit board (PCB) could be used with all of the up-conversion components installed on it. This would avoid the need to use connectorized filters, amplifiers, mixers, and frequency synthesizers which add mass and volume. It is expected that a single PCB in a form factor to fit in a CubeSat would contain all the required components.

e. High Pass Filter Vice Band Pass Filter

A band pass filter was used after the RF mixer prior to the final amplifier. The use of a band pass filter necessitated that the filter be changed when the final transmission frequency was changed from 8.25 GHz to 10.25 GHz. This hardware change could have been avoided with the use of a high pass filter with a cut-off frequency at around 8 GHz.

f. Metal Housing

A metal housing for the RF components should be considered prior to use in a space application. The use of a custom designed housing will allow the Ettus B205 to be used with the custom housing, replacing its own housing. Additionally, the computer controller and the PCB with the RF up conversion components could also be held in the same housing. The use of one housing for all of the components will save volume and mass, while providing better protection in terms of mechanical stability, RF shielding, environmental radiation, and thermal sinking for the components in a space environment.

g. BUS Improvements

The limitation of the EPS board to supply 2A at 5V to the payload drove the use of an alternate power supply for the X-band transmitter payload. Redesigning the EPS board to use higher capacity linear regulators to include a heat dissipation solution to prevent the linear regulators from overheating, or to use a switching regulator like the one on the LTC 3119 power supply, which would allow the EPS to supply more power to a payload while maintaining the use of its features, such as controlling power on/off to the payload and measuring current to the payload.

2. Software Improvements

Areas identified for future software work involve desired DSP components that were not available in GNU Radio, and therefore, could not be implemented

a. Custom GNU Radio Block

The one custom block used in GNU Radio is suspected of being the cause of the need to send certain block repeatedly. This block should either be replaced with a standard

GNU Radio block or its programming should be examined to see if there is something systemic in its operation that would cause specific blocks to fail repeatedly.

b. Incorporate RS Coding

The only FEC used in this project was a convolutional encoder and decoder. Traditionally, spacecraft have used a Reed-Solomon (RS) FEC in conjunction with convolutional FEC. The RS FEC was not used because there was no standard block in GNU Radio that applied RS FEC byte stream data. There is a RS FEC intended for digital video broadcast but it uses a custom data type not compatible with the byte stream used in this project's GNU Radio flowgraphs. Future projects could write custom RS FEC encoder / decoder blocks or switch to a DSP tool that includes RS FEC, such as Simulink.

c. OQPSK Modulation

Offset quadrature phase shift keying (OQPSK) was the desired modulation scheme for this project. The OQPSK modulation scheme is not available as a standard block in GNU Radio. Future projects could write a custom OQPSK modulation block or switch to a DSP tool that includes OQPSK modulation, such as Simulink.

d. Simulink

Simulink is a software tool that is available with MATLAB and capable of DSP. Simulink cannot be run on a Raspberry Pi, and, therefore, cannot be run directly on the transmitter. However, Simulink offers the option to compile the code of a flowgraph for specific hardware, which could then be run on a less capable processor such as a Raspberry Pi. Simulink also offers RS FEC and OQPSK modulation, which were two desired components for this project that could not be implemented due to the limitation in GNU Radio.

3. Additional Testing

Environmental testing was not conducted on this project, since the HAB does not experience the same vibrational, acceleration, or acoustic loads that occur on a space launch, nor does the planned flight altitude reach extreme temperatures. Future work for a

CubeSat payload that would be launched into space should include vibration and thermal vacuum chamber testing.

APPENDIX A. LINK BUDGETS

	X-band HAB Payload	X-band HAB Payload to approximate LEO	Estimated CubeSat in LEO
Frequency	10.25 GHz	10.25 GHz	8.25 GHz
Tx Power	11.4 dBm (14mW)	11.4 dBm (14mW)	33.0 dBm (2W)
Tx Line Loss	-1 dB	-1 dB	-1 dB
Tx Beam width	42.3 degrees	42.3 degrees	74 degrees
Peak Tx antenna gain	9.0 dB	9.0 dB	6 dB
Estimated pointing error	35 degrees	35 degrees	10 degrees
Tx pointing loss	-8.2 dB	-8.2 dB	-0.22 dB
Tx antenna gain	0.78 dB	0.78 dB	5.8 dB
EIRP	11.2 dBm	11.2 dBm	37.8 dBm
Propagation path length	35 km	0.46 km	2000 km
Free space path loss	-143.5 dBm	-105.9 dBm	-176.8 dBm
Peak rx antenna gain	10 dB	10 dB	41.1 dBi
Rx beam width	42 degrees	42 degrees	1.1 degrees
Estimated pointing error	20 degrees	20 degrees	0.1 degree
Rx pointing loss	-2.7 dB	-2.7 dB	-.11 dB
Rx antenna gain	7.3 dBi	7.3 dBi	41.0 dBi
System noise temperature	135 K	135 K	135 K
Data rate	1.00E6 bps	1.00E6 bps	1.00E6 bps
C/N0	69.6 dB-Hz	108.7 dB-Hz	108.7 dB-Hz
Eb/N0	11.1 dB	48.7 dB	48.7 dB
BER	1E-5	1E-5	1E-5
Req Eb/N0 for BER	4.4 dB	4.4 dB	4.4 dB
Implementation loss	-2 dB	-2 dB	-2 dB
Link Margin	4.6 dB	42.3 dB	42.3 dB

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. ADF-4356 REGISTER VALUES

The register values used to initialize the ADF-4356 for flight to provide an output frequency of 6300 MHz at -4 dBm.

Register Number	Hex Value
0	200660
1	8A62AA1
2	80032
3	3
4	32008B84
5	800025
6	350300C6
7	60000E7
8	15596568
9	2719FCC9
10	C026BA
11	61200B
12	15FC
13	D

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. START_PAYLOAD.PY

```
from __future__ import print_function
import os
import subprocess
import shlex
import time
import traceback

# In the final build of the XBAND GRC transmitter Python (auto build)
program:
#
#         after          :          self.digital_constellation_modulator_0          =
digital.generic_mod(.....)
#
# insert:
# fp = open('grc.start', 'w')
# fp.close()

#systemd notes:
# /etc/systemd/system/HAB.services is used to make this program start up
after reboot
# [Unit]
# description=HAB
#
# [Service]
# WorkingDirectory=/home/pi/HAB
# ExecStart=/usr/bin/python start_HAB.py
#
# [Install]
# WantedBy=multi-user.target
# sudo systemctl enable HAB.service
# sudo systemctl start HAB.service

def main():
    #run once
    # sudo python adf4356.py 6300mhz_m4dbm.txt
    os.system("/usr/bin/python adf4356.py 6300mhz_m4dbm.txt")

    inputfile = "grc.fifo"
    samp_rate = 1000000
    freq = 3950000000
    eb = 0.5

    #run always
    # python xband_camera.py
    # sudo python xband_tx_30_jul_fifo.py -n inputfile -s samp_rate -f
freq -e eb
```

```

if os.path.exists('grc.start'):
    os.remove('grc.start')
    os.system('/bin/sync')

pnames = ["/usr/bin/python xband_tx.py -n %s -s %u -f %u -e
%1.1f"%(inputfile, samp_rate, freq, eb),
          "/usr/bin/python xband_camera.py",
          "/usr/bin/python chunking.py --tx"]

processes = []
for pname in pnames:
    processes.append(subprocess.Popen(shlex.split(pname),
stdin=subprocess.PIPE))
    if 'xband_tx.py' in pname:
        while True:
            if os.path.exists('grc.start'):
                break
            time.sleep(1)

# when starting the process wait a bit longer for it to start
time.sleep(3)
start = time.time()
while True:
    for process in processes:
        if process.poll() != None:
            process = subprocess.Popen(shlex.split(pname),
stdin=subprocess.PIPE)
            time.sleep(1)

#####
#####
#####
#####
#####

if __name__ == "__main__":
    time.sleep(3)
    while (True):
        try:
            main()
        except:
            s = traceback.format_exc()
            fp = open("start_payload_traceback.txt," "a")
            fp.write(s+"\n")
            fp.close()

```

APPENDIX D. ADF4356.PY

```
#####
#
#   adf4356.py
#
#   Support for the GPIO based loading of registers of the ADF4356
#
#   Revision History:
#   =====
#   Date Who  What
#   -----+-----+-----
#   2019-04-26   Jah      Creation
#
#####

from __future__ import print_function
import os
import RPi.GPIO as GPIO
import sys

class ADF4356(object):
    CLK_PIN = 14
    DATA_PIN = 15
    LE_PIN = 16

    def __init__(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(CLK_PIN, GPIO.OUT)
        GPIO.setup(DATA_PIN, GPIO.OUT)
        GPIO.setup(LE_PIN, GPIO.OUT)
        GPIO.output(CLK_PIN, False)
        GPIO.output(LE_PIN, True)

    def load_reg(self, reg_value):
        GPIO.output(LE_PIN, False)
        i = 31
        while i >= 0:
            data = reg_value & (1<<i)
            if data > 0:
                GPIO.output(DATA_PIN, True)
            else:
                GPIO.output(DATA_PIN, False)
            GPIO.output(CLK_PIN, True)
            GPIO.output(CLK_PIN, False)
            i -= 1
        GPIO.output(LE_PIN, True)
```

```

def load_regs_from_file(self, fname):
    fp = open(fname, 'r')
    lines = fp.read().split('\n')
    fp.close()
    for line in lines:
        if '#' in line:
            i = line.find('#')
            line = line[:i]
        s = line.split(',')
        if len(s) == 2:
            reg = s[0].strip()
            value = s[1].strip()
            try:
                if '0x' in value[0:2]:
                    value = int(value, 16)
                else:
                    value = int(value)
            except ValueError:
                print('Error: %s cannot be converted to a
number' % s[1])
                print('reg=%s, value=%u'%(reg, value))
                self.load_reg(value)

if __name__ == "__main__":
    adf = ADF4356()
    try:
        fname = sys.argv[1]
    except:
        print('%s <filename>' % sys.argv[0])
    else:
        if os.path.isfile(fname):
            adf.load_regs_from_file(fname)
        else:
            print('Error: %s does not exist' % fname)

```

APPENDIX E. 6300MHZ_M4DBM.TXT

13, 0xD	# xyz reg
12, 0x15FC	# xyz reg
11, 0x61200B	# xyz reg
10, 0xC026BA	# xyz reg
09, 0x2719FCC9	# xyz reg
08, 0x15596568	# xyz reg
07, 0x60000E7	# xyz reg
06, 0x350300C6	# xyz reg
05, 0x800025	# xyz reg
04, 0x32008B84	# xyz reg
03, 0x3	# xyz reg
02, 0x9BC2712	# xyz reg
01, 0x8A4B211	# xyz reg
00, 0x200660	# xyz reg
13, 0xD	# xyz reg
10, 0xC026BA	# xyz reg
02, 0x9BC2712	# xyz reg
01, 0x8A4B211	# xyz reg
00, 0x200660	# xyz reg

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. XBAND_CAMERA.PY

```
from picamera import PiCamera
from time import sleep
import glob
import os

def get_last_file_number(dir):
    files = glob.glob(os.path.join(dir, 'photo*.jpg'))
    files.sort()
    file_numbers = []
    for file in files:
        n = int(file.split('_')[1].split('.')[0])
        file_numbers.append(n)
    return file_numbers [-1]

if __name__ == "__main__":
    if not os.path.exists('photos'):
        os.mkdir('photos')
        i = 0
    else:
        i = get_last_file_number('photos') + 1
    print('i = %u'%i)

    camera = PiCamera()
    #camera.start_preview()

    while True:
        camera.capture('/home/pi/xbandtx/photos/photo_%04u.jpg' % i)
        sleep(45)

    #camera.stop_preview()
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G. XBAND_TX.PY

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
#####
# GNU Radio Python Flow Graph
# Title: Xband Tx 26Aug Fifo
# Generated: Mon Aug 26 17:15:44 2019
#####

from gnuradio import blocks
from gnuradio import digital
from gnuradio import eng_notation
from gnuradio import fec
from gnuradio import gr
from gnuradio import uhd
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from optparse import OptionParser
import pmt
import time

class xband_tx_26aug_fifo(gr.top_block):

    def __init__(self, eb=.35, fifoname='/home/pi/xbandtx/grc.fifo',
frame_len=1024, freq=3950000000, samp_rate=1000000,
samples_per_symbol=2, tx_gain=50):
        gr.top_block.__init__(self, "Xband Tx 26Aug Fifo")

        #####
        # Parameters
        #####
        self.eb = eb
        self.fifoname = fifoname
        self.frame_len = frame_len
        self.freq = freq
        self.samp_rate = samp_rate
        self.samples_per_symbol = samples_per_symbol
        self.tx_gain = tx_gain

        #####
        # Variables
        #####
        self.polys = polys = [109,79]
        self.code_rate = code_rate = 2
        self.K = K = 7

        self.constellation_object = constellation_object =
digital.constellation_calcdist((digital.psk_4()[0]),
(digital.psk_4()[1]), 2, 1).base())
```

```

    self.cc_encoder = cc_encoder = fec.cc_encoder_make(frame_len, K,
code_rate, (polys), 0, fec.CC_TAILBITING, True)

    self.access_code = access_code = [0x1a, 0xcf, 0xfc, 0x1d]

#####
# Blocks
#####
self.uhd_usrp_sink_0 = uhd.usrp_sink(
    ", ".join("'", " "),
    uhd.stream_args(
        cpu_format="fc32",
        channels=range(1),
    ),
)
self.uhd_usrp_sink_0.set_samp_rate(samp_rate)
self.uhd_usrp_sink_0.set_center_freq(freq, 0)
self.uhd_usrp_sink_0.set_gain(tx_gain, 0)
self.uhd_usrp_sink_0.set_antenna('TX/RX', 0)
self.fec_extended_tagged_encoder_0 =
fec.extended_tagged_encoder(encoder_obj_list=cc_encoder, puncpat='11',
lentagname='packet_len', mtu=1500)
self.digital_constellation_modulator_0 = digital.generic_mod(
    constellation=constellation_object,
    differential=False,
    samples_per_symbol=samples_per_symbol,
    pre_diff_code=True,
    excess_bw=eb,
    verbose=False,
    log=False,
)
fp = open('grc.start', 'w')
fp.close()
self.blocks_unpacked_to_packed_xx_0_0 =
blocks.unpacked_to_packed_bb(1, gr.GR_MSB_FIRST)
self.blocks_packed_to_unpacked_xx_1 = blocks.packed_to_unpacked_bb(1,
gr.GR_MSB_FIRST)
self.blocks_packed_to_unpacked_xx_0 = blocks.packed_to_unpacked_bb(1,
gr.GR_MSB_FIRST)
self.blocks_multiply_const_vxx_0 = blocks.multiply_const_vcc((.707, ))
self.blocks_file_source_0 = blocks.file_source(gr.sizeof_char*1,
fifoname, True)
self.blocks_file_source_0.set_begin_tag(pmt.PMT_NIL)
self.Tagged_Stream_MUX_0 = blocks.tagged_stream_mux(gr.sizeof_char*1,
'packet_len', 0)
self.Sync_Vector = blocks.vector_source_b(access_code, True, 1, [])
self.Sync_Stream = blocks.stream_to_tagged_stream(gr.sizeof_char, 1,
len(access_code)*8, "packet_len")
self.Rampup_Vector_Stream =
blocks.stream_to_tagged_stream(gr.sizeof_char, 1, 8, "packet_len")
self.Rampup_Vector = blocks.vector_source_b(8*[0], True, 1, [])
self.Preamble_Vector = blocks.vector_source_b(12*[1,0], True, 1, [])
self.Preamble_Stream_0_0 =
blocks.stream_to_tagged_stream(gr.sizeof_char, 1, frame_len,
"packet_len")

```

```

    self.Preamble_Steam = blocks.stream_to_tagged_stream(gr.sizeof_char, 1,
24, "packet_len")
    self.Buffer_Vector = blocks.vector_source_b(32*[0], True, 1, [])
    self.Buffer_Stream = blocks.stream_to_tagged_stream(gr.sizeof_char, 1,
32, "packet_len")

```

```

#####
# Connections
#####
self.connect((self.Buffer_Stream, 0), (self.Tagged_Stream_MUX_0, 4))
self.connect((self.Buffer_Vector, 0), (self.Buffer_Stream, 0))
self.connect((self.Preamble_Steam, 0), (self.Tagged_Stream_MUX_0, 1))
self.connect((self.Preamble_Steam_0_0, 0),
(self.fec_extended_tagged_encoder_0, 0))
self.connect((self.Preamble_Vector, 0), (self.Preamble_Steam, 0))
self.connect((self.Rampup_Vector, 0), (self.Rampup_Vector_Stream, 0))
self.connect((self.Rampup_Vector_Stream, 0),
(self.Tagged_Stream_MUX_0, 0))
self.connect((self.Sync_Steam, 0), (self.Tagged_Stream_MUX_0, 2))
self.connect((self.Sync_Vector, 0),
(self.blocks_packed_to_unpacked_xx_1, 0))
self.connect((self.Tagged_Stream_MUX_0, 0),
(self.blocks_unpacked_to_packed_xx_0_0, 0))
self.connect((self.blocks_file_source_0, 0),
(self.blocks_packed_to_unpacked_xx_0, 0))
self.connect((self.blocks_multiply_const_vxx_0, 0),
(self.uhd_usrp_sink_0, 0))
self.connect((self.blocks_packed_to_unpacked_xx_0, 0),
(self.Preamble_Steam_0_0, 0))
self.connect((self.blocks_packed_to_unpacked_xx_1, 0),
(self.Sync_Steam, 0))
self.connect((self.blocks_unpacked_to_packed_xx_0_0, 0),
(self.digital_constellation_modulator_0, 0))
self.connect((self.digital_constellation_modulator_0, 0),
(self.blocks_multiply_const_vxx_0, 0))
self.connect((self.fec_extended_tagged_encoder_0, 0),
(self.Tagged_Stream_MUX_0, 3))

```

```

def get_eb(self):
    return self.eb

```

```

def set_eb(self, eb):
    self.eb = eb

```

```

def get_fifoname(self):
    return self.fifoname

```

```

def set_fifoname(self, fifoname):
    self.fifoname = fifoname
    self.blocks_file_source_0.open(self.fifoname, True)

```

```

def get_frame_len(self):
    return self.frame_len

```

```

def set_frame_len(self, frame_len):
    self.frame_len = frame_len
    self.Preamble_Steam_0_0.set_packet_len(self.frame_len)
    self.Preamble_Steam_0_0.set_packet_len_pmt(self.frame_len)

def get_freq(self):
    return self.freq

def set_freq(self, freq):
    self.freq = freq
    self.uhd_usrp_sink_0.set_center_freq(self.freq, 0)

def get_samp_rate(self):
    return self.samp_rate

def set_samp_rate(self, samp_rate):
    self.samp_rate = samp_rate
    self.uhd_usrp_sink_0.set_samp_rate(self.samp_rate)

def get_samples_per_symbol(self):
    return self.samples_per_symbol

def set_samples_per_symbol(self, samples_per_symbol):
    self.samples_per_symbol = samples_per_symbol

def get_tx_gain(self):
    return self.tx_gain

def set_tx_gain(self, tx_gain):
    self.tx_gain = tx_gain
    self.uhd_usrp_sink_0.set_gain(self.tx_gain, 0)

def get_polys(self):
    return self.polys

def set_polys(self, polys):
    self.polys = polys

def get_code_rate(self):
    return self.code_rate

def set_code_rate(self, code_rate):
    self.code_rate = code_rate

def get_K(self):
    return self.K

def set_K(self, K):
    self.K = K

def get_constellation_object(self):
    return self.constellation_object

```

```

def set_constellation_object(self, constellation_object):
    self.constellation_object = constellation_object

def get_cc_encoder(self):
    return self.cc_encoder

def set_cc_encoder(self, cc_encoder):
    self.cc_encoder = cc_encoder

def get_access_code(self):
    return self.access_code

def set_access_code(self, access_code):
    self.access_code = access_code
    self.Sync_Vector.set_data(self.access_code, [])
    self.Sync_Steam.set_packet_len(len(self.access_code)*8)
    self.Sync_Steam.set_packet_len_pmt(len(self.access_code)*8)

def argument_parser():
    parser = OptionParser(usage="%prog: [options]", option_class=eng_option)
    parser.add_option(
        "-e",          "--eb",          dest="eb",          type="eng_float",
        default=eng_notation.num_to_str(.35),
        help="Set eb [default=%default]")
    parser.add_option(
        "-n",          "--fifoname",    dest="fifoname",    type="string",
        default='/home/pi/xbandtx/grc.fifo',
        help="Set fifoname [default=%default]")
    parser.add_option(
        "-l",          "--frame-len",   dest="frame_len",   type="intx",        default=1024,
        help="Set frame_len [default=%default]")
    parser.add_option(
        "-f",          "--freq",        dest="freq",        type="eng_float",
        default=eng_notation.num_to_str(3950000000),
        help="Set freq [default=%default]")
    parser.add_option(
        "-s",          "--samp-rate",    dest="samp_rate",   type="eng_float",
        default=eng_notation.num_to_str(1000000),
        help="Set samp_rate [default=%default]")
    parser.add_option(
        "-p",          "--samples-per-symbol", dest="samples_per_symbol",
        type="eng_float", default=eng_notation.num_to_str(2),
        help="Set samples_per_symbol [default=%default]")
    parser.add_option(
        "-g",          "--tx-gain",      dest="tx_gain",      type="eng_float",
        default=eng_notation.num_to_str(50),
        help="Set tx_gain [default=%default]")
    return parser

def main(top_block_cls=xband_tx_26aug_fifo, options=None):
    if options is None:
        options, _ = argument_parser().parse_args()

```

```
tb      =      top_block_cls(eb=options.eb,      fifoname=options.fifoname,
frame_len=options.frame_len,      freq=options.freq,
samp_rate=options.samp_rate,
samples_per_symbol=options.samples_per_symbol, tx_gain=options.tx_gain)
tb.start()
tb.wait()

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

APPENDIX H. FIXEDLEN_PACKET_SYNCHRONIZER.PY

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# Copyright 2016 <+YOU OR YOUR COMPANY+>.
#
# This is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# This software is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this software; see the file COPYING. If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

import numpy
from gnuradio import gr
import pmt

import collections

class fixedlen_packet_synchronizer(gr.basic_block):
    """
    docstring for block fixedlen_packet_synchronizer
    """
    def __init__(self, synctag, plentag, plen, streamtype):
        gr.basic_block.__init__(self,
            name="fixedlen_packet_synchronizer",
            in_sig=[streamtype],
            out_sig=[streamtype])
        self.syncword_tag = pmt.string_to_symbol(synctag)
        self.packetlen_tag = pmt.string_to_symbol(plentag)
        self.packet_len = plen
    def forecast(self, noutput_items, ninput_items_required):
        num_frames = int(noutput_items/self.packet_len)
        ninput_items_required[0] = (96 + self.packet_len)*num_frames
    def general_work(self, input_items, output_items):
        input = input_items[0]
        output = output_items[0]
        if len(output_items[0]) < self.packet_len: return 0
        tags = self.get_tags_in_range(0, self.nitems_read(0),
self.nitems_read(0) + len(input), self.syncword_tag)
```

```

tag = tags[0]
n = tag.offset - self.nitems_read(0)
output_items[0][:self.packet_len] = input[n:n + self.packet_len]
consume_len = len(input[:n + self.packet_len])
self.add_item_tag(0, self.packet_len, self.packetlen_tag,
pmt.from_long(self.packet_len))
self.consume(0, consume_len)
return self.packet_len

```

APPENDIX F. CHUNKING.PY

```
#####
#####
#
#   chunking.py
#
#   Revision History:
#   =====
#   Date   Who   What
#   -----+-----+-----
-----
#   2019-10-10       Jah           Creation:   Adopted   from   program
written for Lovdahl
#   2019-08-27       Jah           Modified (RX) to co-exist w/ Cosmos
with
#                                   cosmos_serial_server.py
#
#####
#####

from __future__ import print_function
import binascii
import glob
import optparse
import os
import pickle
import Queue
import random
import select
import serial
import socket
import struct
import sys
import time
import threading
import traceback

import hexdump
import RADIO

FILE_REPEATS = 3

DELAY_BETWEEN_PACKETS = 0
PACKET_LEN = 128
GRC_TO_PORT = 1234
# packet structure (128 bytes):
#   2       file number                2
#   2       file number (repeat)        4
#   4       file size                   8
#   2       block number                10
```

```

# 4 CRC 14
# 114 block data 128
BLOCK_DATA_SIZE = 114
BLOCK_DATA_OFFSET = PACKET_LEN - BLOCK_DATA_SIZE

FIFO_PATH = 'grc.fifo'
MAX_QUEUE_SIZE = 2500

# MHX files complete feedback
MHX_FEEDBACK_SIZE_BYTES = 80
MHX_FEEDBACK_MSG = 'PLFILES '
MHX_FILES_FEEDBACK_MSG = 'PLFILES '
MHX_TCP_PORT = 6061 # for co-exisiting with Cosmos using
cosmos_serial_server.py
NEWLINE = b'\x0A'
USE_WITH_COSMOS = False

# MHX file blocks missing feedback
MHX_BLOCKS_FEEDBACK_MSG = 'PLBLOCKS '
RX_FEEDBACK_MIN_THRESHOLD_DEFAULT = 100 # min. number of missing
blocks where this feedback begins

payload_port = '/dev/ttyS0'
radio_baudrate = 9600
ground_port = '/dev/ttyUSB0'

def make_files(count):
    if not os.path.exists('photos'):
        os.mkdir('photos')

    files = glob.glob('photos/photo*.jpg')
    if len(files) == 0:
        last = 0
    else:
        files.sort()
        files.reverse()
        last = int(files[0].split('_')[1].split('.')[0])
    for i in range(last, last+count):
        fp = open(os.path.join('photos', 'photo_%04u.jpg'%i), 'wb')
        fp.write(os.urandom(1024*1024))
        fp.close()

def get_file_numbers(dir):
    files = glob.glob(os.path.join(dir, 'photo*.jpg'))
    files.sort()
    file_numbers = []
    for file in files:
        n = int(file.split('_')[1].split('.')[0])
        file_numbers.append(n)
    return file_numbers

```

```

class SENDER(object):
    def __init__(self, comm=None):
        self.queue = Queue.Queue()
        self.thread = threading.Thread(target=self.send_to_GRC_thread, args=(self.queue,))
        self.thread.daemon = True
        self.thread.start()

        self.files = {}

        if os.path.exists('files_recieved.txt'):
            fp = open('files_recieved.txt', 'r')
            data = fp.read()
            fp.close()
            print('Files already downloaded:')
            for line in data.split('\n'):
                if len(line) < 1:
                    continue
                fn = int(line.strip())
                self.files[fn] = {'size':0, 'N':0,
'remainder':0, 'complete': True, 'missing_blocks_to_send':[]}
                print(fn)

            if comm == None:
                self.radio = RADIO.RADIO(payload_port, radio_baudrate,
eol='\x0D')
            else:
                self.radio = RADIO.RADIO(comm, radio_baudrate,
eol='\x0D')
                self.eol = '\x0D'

        def send_to_GRC_thread(self, queue):
            while True:
                fifo_fp = open(FIFO_PATH, 'wb')
                while True:
                    try:
                        packet = self.queue.get(timeout=0.01)
                    except Queue.Empty:
                        pass
                    else:
                        try:
                            fifo_fp.write(packet)
                        except IOError:
                            print('FIFO pipe broken, retrying.')
                            break

        def send_files(self):
            while True:
                file_numbers = get_file_numbers('photos')
                for fn in file_numbers:
                    if fn in self.files:

```

```

        if self.files[fn]['complete']:
            continue
        for i in range(0, FILE_REPEATS):
            self.send_file(fn)

    def get_file_data(self, fn):
        if fn not in self.files:
            try:
                size = os.path.getsize(os.path.join('photos',
'photo_%04u.jpg'%fn))
            except os.error:
                print('Error: cannot get size of
%s'%os.path.join('photos', 'photo_%04u.jpg'%fn))
                # cause it to skip future tries
                self.files[fn] = {'size':size, 'N':N,
'remainder':remainder, 'complete': False, 'missing_blocks_to_send':[]}
                return None, None, None, None

                N = int(size/BLOCK_DATA_SIZE)
                remainder = size - N*BLOCK_DATA_SIZE
                if remainder > 0:
                    N += 1
                self.files[fn] = {'size':size, 'N':N,
'remainder':remainder, 'complete': False, 'missing_blocks_to_send':[]}
            else:
                size = self.files[fn]['size']
                N = self.files[fn]['N']
                remainder = self.files[fn]['remainder']

        try:
            fp = open(os.path.join('photos', 'photo_%04u.jpg'%fn),
'rb')

            data = fp.read()
            fp.close()
        except:
            print('Error: could not open %s'%fname)
            return None, None, None, None

        blocks = {}
        for n in range(0, N):
            if n < N-1:
                blocks[n] =
data[n*BLOCK_DATA_SIZE:n*BLOCK_DATA_SIZE+BLOCK_DATA_SIZE]
            else:
                block_data = data[n*BLOCK_DATA_SIZE:]
                while len(block_data) < BLOCK_DATA_SIZE:
                    block_data += struct.pack('<B',
random.randint(0, 255))
                blocks[n] = block_data

        return size, N, remainder, blocks

    def send_file(self, fn):

```

```

os.system('/opt/vc/bin/vcgenclm measure_temp >/tmp/temp')
fp = open('/tmp/temp', 'r')
data = fp.read().strip()
fp.close()
payload = '11 %10u %s' %(int(time.time()), data)
self.radio.write(payload+self.eol, send_eol=True)

size, N, remainder, blocks = self.get_file_data(fn)
if size == None:
    return

print('fn=%u, N=%d'%(fn, N), end='')
if self.files[fn]['complete']:
    print(', complete')
else:
    print()

n = 0
last_missing_block_index = 0
last_feedback_check = time.time()
b = 0
missing_loops = 0
mhx_data = ''
while True:
    if self.files[fn]['missing_blocks_to_send'] == []:
        b += 1
        bn = b
        if b >= len(blocks):
            # only once through if in full file tx mode
            break
        data = blocks[b][:]
    else:
        # will stay in this mode until the file is
        # completely downloaded
        if self.files[fn]['complete']:
            break
        try:
            bn
            =
self.files[fn]['missing_blocks_to_send'][last_missing_block_index]
        except IndexError:
            last_missing_block_index = 0
            continue
        else:
            last_missing_block_index += 1
            if last_missing_block_index >=
len(self.files[fn]['missing_blocks_to_send']):
                last_missing_block_index = 0
                data = blocks[bn][:]

        try:
            crc = binascii.crc32(data)&0xFFFFFFFF
        except KeyError:
            print('fn=%u, bn=%u KeyError'%(fn, bn))
            continue
        packet = struct.pack('<HHIHI', fn, fn, size, bn, crc)

```

```

        packet += data

    while self.queue.qsize() > MAX_QUEUE_SIZE:
        time.sleep(0.05)
    self.queue.put(packet)

#         print('fn=%u, n=%d, N=%d, queue = %d'%(fn, b, N,
self.queue.qsize()))

    if time.time() - last_feedback_check >= 0.5:
        last_feedback_check = time.time()
        data = None
        data = self.radio.get_line()

        if data != None:
            hexdump.hexdump(data)
            if MHX_FILES_FEEDBACK_MSG in data:
                numbers =
data[len(MHX_FILES_FEEDBACK_MSG):].split(',')
                for num in numbers:
                    if len(num) == 0:
                        break
                    fs = int(num, 16)
                    if fs not in self.files:
                        self.files[fs] =
{'size':size, 'N':N, 'remainder':remainder, 'complete': True}
                    else:
                        if
self.files[fs]['complete']:
                        print('File %u is
complete'%fs)

                        self.files[fs]['complete'] = True
                        break
                    elif MHX_BLOCKS_FEEDBACK_MSG in data:
                        numbers =
data[len(MHX_BLOCKS_FEEDBACK_MSG):].split(',')
                        fn = int(numbers[0], 16)
                        missing_blocks = numbers[1:]
                        if fn in self.files:
                            for bn in missing_blocks:
                                if len(bn) == 0:
                                    break
                                bn_num = int(bn, 16)
                                if bn_num not in
self.files[fn]['missing_blocks_to_send']:
                                self.files[fn]['missing_blocks_to_send'].append(bn_num)
                                size, N, remainder, blocks =
self.get_file_data(fn)

                        if self.files[fn]['complete']:
                            fp = open('files_recieved.txt', 'w')
                            for fn in self.files:

```

```

        fp.write('%u\n'%fn)
    fp.close()

def test_tx(self):
    count = 0
    while True:
        packet = ''
        for i in range(0, 32):
            packet += struct.pack('<I', count)
        while self.queue.qsize() > MAX_QUEUE_SIZE:
            time.sleep(0.05)
        self.queue.put(packet)
        count += 1
        if count % 100 == 0:
            print('%6u packets sent'%count)

def extract_block(self, fn, block):
    print(fn, block)
    size, N, remainder, blocks = self.get_file_data(fn)
    print(size, N, remainder)
    print(len(blocks))
    print('%08X'%binascii.crc32(blocks[block]))
    hexdump.hexdump(blocks[block])

class RECEIVER(object):
    def __init__(self, comm=None,
rx_feedback=RX_FEEDBACK_MIN_THRESHOLD_DEFAULT):
        self.queue = Queue.Queue()
        self.files = {}
        self.rx_feedback = rx_feedback

        self.rx_sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
        self.rx_sock.bind(('', 1234))

        self.feedback_index = None

# 2019-08-27:
# changed to co-exist w/ Cosmos
    if not USE_WITH_COSMOS:
        if comm == None:
            self.radio = RADIO.RADIO(ground_port,
radio_baudrate, eol='\x0D')
        else:
            self.radio = RADIO.RADIO(comm, radio_baudrate,
eol='\x0D')
    else:
        try:
            self.feedback_sock =
socket.socket(socket.AF_INET, socket.SOCK_STREAM)

```

```

        self.feedback_sock.connect(('127.0.0.1',
MHX_TCP_PORT))
    except:
        print('Error:          cannot          connect          to
cosmos_serial_server')
        sys.exit()

    self.last_feedback_check = time.time()

    if not os.path.exists('download'):
        os.mkdir('download')

    def receive_files(self):
        file_numbers = get_file_numbers('download')
        for fn in file_numbers:
            self.files[fn] = {'size':0, 'N':0, 'remainder':0,
'received':set(), 'complete': True, 'data':{}}
            while True:
                self.receive_file()

    def send_feedback_files_received(self):
        # send feedback about completely received files.

        # format for feedback to HAB payload is:
        # 'PLFILES '          8 bytes
        # XXX                  3 bytes (ASCII-encoded HEX: value from 0-
FFF or 0-4095)
        # payload can be MHX_FEEDBACK_SIZE_BYTES in size
        if len(self.files) == 0:
            return

        if self.feedback_index == None:
            self.feedback_index = min(self.files.keys())

        if time.time() - self.last_feedback_check < 2:
            return
        self.last_feedback_check = time.time()

        payload = MHX_FILES_FEEDBACK_MSG
        start_i = self.feedback_index
        # print('feedback_index = %u (%u - %u)'%(self.feedback_index,
min(self.files.keys()), max(self.files.keys())))

        while True:
            if self.feedback_index in self.files:
                if self.files[self.feedback_index]['complete']:
                    payload += '%03X,%'%self.feedback_index
                if len(payload) >= MHX_FEEDBACK_SIZE_BYTES:
                    break
                self.feedback_index += 1
            if self.feedback_index > max(self.files.keys()):
                self.feedback_index = min(self.files.keys())
            if self.feedback_index == start_i:

```

```

        break
    if len(payload) > len(MHX_FILES_FEEDBACK_MSG):
        if not USE_WITH_COSMOS:
            self.radio.write_print(payload, send_eol=True)
        else:
            try:

self.feedback_sock.sendall(payload+NEWLINE)
            except socket.error:
                self.feedback_sock.close()
                time.sleep(1)
                self.feedback_sock
socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.feedback_sock.connect(('127.0.0.1',
MHX_TCP_PORT))

def send_feedback_blocks_received(self, fn, received):
    # send feedback about missing blocks of a partially received
files.

    # format for feedback to HAB payload is:
    # 'PLBLOCKS '      9 bytes
    # FN                4 bytes (ASCII-encoded file number)
    # XXXX              4 bytes (ASCII-encoded HEX: value
from 0-FFFF or 0-65535)
    # payload can be MHX_FEEDBACK_SIZE_BYTES in size

    if len(self.files) == 0:
        return

    if self.feedback_index == None:
        self.feedback_index = min(self.files.keys())

    if time.time() - self.last_feedback_check < 2:
        return
    self.last_feedback_check = time.time()

    payload = MHX_BLOCKS_FEEDBACK_MSG + '%04X,%fn
    print('fn=%u: len(rx)=%u'%(fn, len(received)))
    for b in range(0, self.files[fn]['N']):
        if b not in received:
            payload += '%04X,%b
            if len(payload) > MHX_FEEDBACK_SIZE_BYTES:
                break
    if not USE_WITH_COSMOS:
        self.radio.write_print(payload, send_eol=True)
    else:
        try:
            self.feedback_sock.sendall(payload+NEWLINE)
        except socket.error:
            self.feedback_sock.close()
            time.sleep(1)

```

```

        self.feedback_sock =
socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.feedback_sock.connect(('127.0.0.1',
MHX_TCP_PORT))

    def receive_file(self):
        stat_timer = 0
        while True:
            block_feedback = None
            (rd, wr, err) = select.select([self.rx_sock], [], [],
0.001)

            if self.rx_sock in rd:
                success = True
                packet = self.rx_sock.recv(128)
                if packet:
                    (fn, fn2, size, n, crc) =
struct.unpack('<HHIHI', packet[0:BLOCK_DATA_OFFSET])
                    gen_crc =
binascii.crc32(packet[BLOCK_DATA_OFFSET:])&0xFFFFFFFF
                    if gen_crc != crc:
                        print('CRC failed: fn=%u (%u) b=%u
%08X != %08X' %(fn, fn2, n, gen_crc, crc))
                        hexdump.hexdump(packet)
                        success = False
                    if fn != fn2:
                        print('fn=%u != fn2=%u b=%u %08X !=
%08X' %(fn, fn2, n, gen_crc, crc))
                        hexdump.hexdump(packet)
                        success = False

                if success:
                    if fn in self.files:
                        if self.files[fn]['complete']:
                            break

                    if fn not in self.files:
                        N = int(size/BLOCK_DATA_SIZE)
                        remainder = size -
N*BLOCK_DATA_SIZE

                        if remainder > 0:
                            N += 1
                            self.files[fn] = {'size':size,
'N':N, 'remainder':remainder, 'received':set(), 'complete': False,
'data':{}}

                            print('fn=%u, size=%u, n=%u,
N=%u' %(fn, size, n, N))

                        else:
                            N = self.files[fn]['N']
                            remainder =

self.files[fn]['remainder']

                    data = packet[BLOCK_DATA_OFFSET:]
                    if n == self.files[fn]['N']-1:

```

```

data[:remainder]
self.files[fn]['data'][n] =
else:
self.files[fn]['data'][n] =
data[:]:
if n not in
self.files[fn]['received']:
self.files[fn]['received'].add(n)
if len(self.files[fn]['received'])
== self.files[fn]['N']:
fp =
open(os.path.join('download', 'photo_%04u.jpg'%fn), 'wb')
for n in
sorted(self.files[fn]['data']):
fp.write(self.files[fn]['data'][n])
fp.close()
print('fn=%u complete.'%fn)
self.files[fn]['complete'] =
True
block_feedback = None
break
if time.time() - stat_timer >= 1:
print('fn=%u: %u/%u (%1.1f
%%)'%(fn, len(self.files[fn]['received']), N,
100.0*float(len(self.files[fn]['received']))/float(N)))
stat_timer = time.time()
if not self.files[fn]['complete']:
if self.files[fn]['N'] -
len(self.files[fn]['received']) < self.rx_feedback:
block_feedback = fn
self.send_feedback_blocks_received(fn, self.files[fn]['received'])
else:
self.send_feedback_files_received()
else:
self.send_feedback_files_received()
else: # ELSE of (if success:)
if block_feedback != None:
if not
self.files[block_feedback]['complete']:
if
self.files[block_feedback]['N'] -
len(self.files[block_feedback]['received']) < self.rx_feedback:
self.send_feedback_blocks_received(block_feedback,
self.files[block_feedback]['received'])

```

```

else:

self.send_feedback_files_received()
else:

self.send_feedback_files_received()

self.send_feedback_files_received()

def test_rx(self):
    counter = 0
    first_count = None
    while True:
        (rd, wr, err) = select.select([self.rx_sock], [], [],
0.001)
        if self.rx_sock in rd:
            data = self.rx_sock.recv(128)
            if data:
                counts = struct.unpack('<32I', data)
                good = True
                if True: #try:
                    count0 = counts[0]
                    for i in range(1, 32):
                        if counts[i] != count0:
                            good = False
                            break
                else: #except:
                    good = False
                if good:
                    counter += 1
                    if first_count == None:
                        first_count = count0
                    total = count0 - first_count + 1
                    print('RX (%u) Count = %6u, Total
Count = %6u, Percentage = %1.2f %%%'(count0, counter, total,
100.0*float(counter)/float(total)))

if __name__ == "__main__":
    parser = optparse.OptionParser()
    parser.usage = "usage: %prog [options]."
    parser.add_option("--comm", help="Communication port: <portname>")
    parser.add_option("--fname", help="File to transmit: <source>")
    parser.add_option("--make", help="Make random photo files:
<number>")
    parser.add_option("--rx", help="Receive data",
action="store_true")
    parser.add_option("--tx", help="Transmit data.",
action="store_true")
    parser.add_option("--test", help="Test: <tx> or <rx>")
    parser.add_option("--rxfb", help="Receive feedback: <min.
threshold>")
    parser.add_option("--block", help="fn,block")

```

```

(options, args) = parser.parse_args()

if options.block:
    (fn, block) = options.block.split(',')
    sender = SENDER(comm=None)
    sender.extract_block(fn=int(fn), block=int(block))

if options.comm:
    comm = options.comm
else:
    comm = None

if options.make:
    make_files(int(options.make))

elif options.rx:
    if options.rxfb:
        rxfb = int(options.rxfb)
    else:
        rxfb = RX_FEEDBACK_MIN_THRESHOLD_DEFAULT
    receiver = RECEIVER(comm=comm, rx_feedback=rxfb)
    receiver.receive_files()
    sys.exit()

elif options.tx:
    sender = SENDER(comm=comm)
    if options.fname:
        sender.send_file(options.fname)
    else:
        sender.send_files()

elif options.test:
    if options.test.lower() == 'tx':
        sender = SENDER(comm=comm)
        sender.test_tx()
    elif options.test.lower() == 'rx':
        receiver = RECEIVER(comm=comm)
        receiver.test_rx()
    else:
        print('ERROR: test mode is either tx or rx')

```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [1] EnduroSat, “X-band transmitter CubeSat communication module.” Accessed 20 Nov 2018. [Online]. Available: <https://www.endurosat.com/cubesat-store/cubesat-communication-modules/x-band-transmitter/>
- [2] Planet Labs, “Planet imagery product specifications.” Accessed Jun 2019, [Online]. Available: <https://assets.planet.com/docs/combined-imagery-product-spec-final-may-2019.pdf>
- [3] M. Swartwout, “CubeSat database,” Saint Louis University, 15 Aug 2019. [Online]. Available: <https://sites.google.com/a/slu.edu/swartwout/home/cubesat-database>
- [4] Defense Intelligence Agency, “Challenges to security in space,” Jan 2019. [Online]. Available: https://www.dia.mil/Portals/27/Documents/News/Military%20Power%20Publications/Space_Threat_V14_020119_sm.pdf
- [5] National Aeronautics and Space Administration, *CubeSat 101*, Oct 2017. [Online]. Available: https://www.nasa.gov/sites/default/files/atoms/files/nasa_csli_cubesat_101_508.pdf
- [6] United Launch Alliance, “CubeSat Express: The future of rideshare launch,” April 2017. [Online]. Available: [https://www.ulalaunch.com/docs/default-source/rideshare/cubesat-express-the-future-of-rideshare-launch-\(cubesat-developer’s-workshop-may-2017\).pdf?sfvrsn=a021057e_2](https://www.ulalaunch.com/docs/default-source/rideshare/cubesat-express-the-future-of-rideshare-launch-(cubesat-developer’s-workshop-may-2017).pdf?sfvrsn=a021057e_2)
- [7] G. Minelli, L. Magallanes, N. Weitz, D. Rigmaiden, J. Horning and J. Newman, “The Mobile CubeSat Command and Control (MC3) ground station network: An overview and look ahead,” presented at the 33rd Annual AIAA/USU Conference on Small Satellites, Logan, UT, USA, 7 Aug 2019.
- [8] A. M. Wyglinski and D. Pu, *Digital Communication Systems Engineering with Software-Defined Radio*. Boston: Artech House, 2013.
- [9] International Telecommunications Union, “Mission statement.” Accessed 21 Jun 2019. [Online]. Available: <https://www.itu.int/en/ITU-R/information/Pages/mission-statement.aspx>

- [10] *Manual of Regulations and Procedures for Federal Radio Frequency Management*, 47 CFR Part 300, National Telecommunications and Information Administration, Washington, DC, 2017 [Online]. Available: <https://www.ntia.doc.gov/page/2011/manual-regulations-and-procedures-federal-radio-frequency-management-redbook>
- [11] A. B. Carlson and P. B. Crilly, *Communication Systems*, Boston: McGraw-Hill, 2010.
- [12] J. R. Wertz, D. F. Everett and J. J. Puschell, *Space Mission Engineering: The New SMAD*, Hawthorne, CA: Microcosm Press, 2011.
- [13] G. E. Corazza, *Digital Satellite Communications*, New York: Springer, 2007.
- [14] W. Tuttlebee, *Software Defined Radio: Enabling Technologies*, Chichester, England: John Wiley & Sons, 2002.
- [15] A. Tato, "Software Defined Radio: A brief introduction," *Proceedings*, vol. 2, no. 18, 2018. [Online]. <https://doi.org/10.3390/proceedings2181196>
- [16] G. D. Gordon and W. L. Morgan, *Principles of Communications Satellites*, New York: John Wiley & Sons, 1993.
- [17] Syrlinks, "X-band transmitter for nanosatellites," [Online]. Available: <https://www.syrlinks.com/en/space/nano-satellite/x-band-transmitter-ewc27>
- [18] Syrlinks, "X/S transceiver for nanosatellites," [Online]. Available: <https://www.syrlinks.com/en/space/nano-satellite/x-s-transceiver-ewc27-srx>
- [19] EnduroSat, "S-band receiver - CubeSat communications," [Online]. Available: <https://www.endurosat.com/cubesat-store/cubesat-communication-modules/s-band-receiver/>.
- [20] P. Swintek, "Critical vulnerabilities in the space domain: Using nanosatellites as an alternative to traditional satellite architectures," M.S. thesis, Space Sys. Academic Group, NPS, Monterey, CA, 2018. [Online]. Available: <http://hdl.handle.net/10945/59600>
- [21] B. L. Lovdahl, "Software-defined Radio payload for CubeSat and X-band communications," M.S. thesis, Space Sys. Academic Group, NPS, Monterey, A, 2018. [Online]. Available: <http://hdl.handle.net/10945/61218>
- [22] Code of Federal Regulations Title 14, Chapter I, Subchapter F, Part 101.1. 2019
- [23] Ettus Research, "USRP B205mini-i," Accessed 6 Nov 2018. [Online]. Available: <https://www.ettus.com/all-products/usrp-b205mini-i/>

- [24] Raspberry Pi Foundation, “Raspberry Pi 4 model B specifications,” Accessed 02 Aug 2019. [Online]. Available: <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/>
- [25] Analog Devices, “ADF4356 data sheet,” Jun 2017. [Online]. Available: <https://www.analog.com/media/en/technical-documentation/data-sheets/ADF4356.pdf>
- [26] Analog Devices, “EVAL-ADF4356 evaluation board,” Accessed 7 Feb 2019. [Online]. Available: <https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-adf4356.html#eb-relatedsoftware>
- [27] Mini Circuits, “Wideband microwave amplifier ZX60-183A+,” Accessed 27 Nov 2018. [Online]. <https://www.minicircuits.com/pdfs/ZX60-183A-S+.pdf>
- [28] Pasternack, “PE87FL1000,” Accessed 5 Jun 2019. [Online]. Available: <https://www.pasternack.com/images/ProductPDF/PE87FL1000.pdf>
- [29] Mini-Circuits, “ZVBP-8250+,” Accessed 1 Mar 2019. [Online]. Available: <https://www.minicircuits.com/pdfs/ZVBP-8250+.pdf>
- [30] Mini-Circuits, “VBFZ-6260+,” Accessed 2 Aug 2019. [Online]. Available: <https://www.minicircuits.com/pdfs/VBFZ-6260+.pdf>
- [31] Mini-Circuits, “ZVBP-5800+,” Accessed 2 Aug 2019. [Online]. Available: <https://www.minicircuits.com/pdfs/ZVBP-5800+.pdf>
- [32] Raspberry Pi foundation, “Camera module—Raspberry Pi documentation,” Accessed 02 Aug 2019. [Online]. Available: <https://www.raspberrypi.org/documentation/hardware/camera/>
- [33] Sage Millimeter, “SAM-8428520865-SF-L1 X-band microstrip patch antenna,” Accessed 1 Jul 2019. [Online]. Available: <https://www.sagemillimeter.com/content/datasheets/SAM-8428520865-SF-L1.pdf>
- [34] J. Coppens, “Antennas - Helical - Javascript on-line calculator.” Accessed 15 July 2019. [Online]. Available: <http://jcoppens.com/ant/helix/calc.en.php>
- [35] Naval Postgraduate School Space Systems Academic Group, “HAB ICD.” Monterey, CA, 6 Nov 2018. Unpublished.
- [36] The GNU Radio Foundation, Inc, “About GNU Radio.” Accessed 01 Aug 2019. [Online]. Available: <https://www.gnuradio.org/about/>
- [37] J. Fran, “CC1101 decoder with FEC implemented for GNU Radio,” 23 Oct 2017. [Online]. Available: https://github.com/juan0fran/gr-cc_sdr

- [38] Consultative Committee for Space Data Systems, “CCSDS.org - The Consultative Committee for Space Data Systems (CCSDS).” Accessed 01 Aug 2019. [Online]. Available: <https://public.ccsds.org/default.aspx>
- [39] *Recommended Standard for TM Synchronization and Channel Coding*, CCSDS 131.0-B-3. Washington, DC, Sep 2017. [Online]. Available: <https://public.ccsds.org/Pubs/131x0b3e1.pdf>
- [40] UKHAS, “habhub.” Accessed: 29-Aug-2019. [Online]. Available: <http://habhub.org/>
- [41] Google LLC, Mountain View, CA. 2019. Google Earth Pro, ver 7.3.2.5776 (64-bit). [Online]. Available: <https://www.google.com/earth/versions/>

INITIAL DISTRIBUTION LIST

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