

Montana Tech Library

Digital Commons @ Montana Tech

Graduate Theses & Non-Theses

Student Scholarship

Summer 2020

MODELING AND PROTOTYPING A MODULAR, LOW-COST COLLISION AVOIDANCE SYSTEM FOR UAVS

Tyler Holliday

Follow this and additional works at: https://digitalcommons.mtech.edu/grad_rsch



Part of the [Electrical and Computer Engineering Commons](#)

MODELING AND PROTOTYPING A MODULAR, LOW-COST COLLISION AVOIDANCE SYSTEM FOR UAVS

by
Tyler Holliday

A thesis submitted in partial fulfillment of the
requirements for the degree of

Masters of Science Electrical Engineering

Montana Technological University

2020



Abstract

Many challenges arise when attempting to use unmanned aerial vehicles (UAVs) in indoor environments, such as the lack of a GPS signal for use in navigation and the smaller margin of error in movements. Typically, those challenges are addressed by using a collision avoidance system. However, most commercially available collision avoidance systems are expensive, limited in suppliers, and are restricted to use on a specific platform. Additionally, some of the collision avoidance systems choose to forego obstacle detection in one or more directions, usually the upward direction. This work proposes that it is possible to develop a custom, low-cost collision avoidance system with modular capabilities, allowing it to be adapted to any UAV platform. The feasibility of the proposed system was determined by creating a single-direction prototype that was implemented on a small quadcopter and tested by flying the quadcopter towards a wall at slow speeds. To develop the system's control algorithm a model of a quadcopter was built. Two different control algorithms were developed and tested via simulation with the model, and the better performing algorithm was implemented in the prototype. The feasibility of the proposed collision avoidance system is promising with the prototype able to prevent the quadcopter from colliding with a wall. However, further refinement in the methodology and techniques used to develop the system is needed to improve performance and reliability of the system, especially as obstacle detection is added in other directions of motion.

Keywords: unmanned aerial vehicles, collision avoidance systems, feedback control system design, modeling of UAVs, obstacle detection

Dedication

This work is dedicated to my parents, Cindy and Mike Holliday, and my best friends, Benjamin Rathman and Cody Barnett. All of you have been critical to my success, believing in and supporting me when I struggled to myself. I could not have done any of this without all of you.

Acknowledgements

I would like to thank my advisor, Dr. Bryce Hill. His guidance and expertise were critical to my undertaking of this project. He was also extraordinarily helpful at keeping my morale high and providing constant insight. I would also like to thank Dr. Kevin Negus, who alongside Dr. Hill, convinced me to continue my education and pursue my interests more deeply.

I would like to thank my remaining committee members, Dr. Josh Wold and Dr. Curtis Link. I am grateful for Dr. Wold's guidance during the development of the model and controller. Dr. Link's observations and critiques were much appreciated when I was determining how to present and defend my work. I also extend the same gratitude to the entire Electrical Engineering department. All were instrumental in supporting and encouraging me during my tenure at Montana Tech.

A special thanks goes to Mary MacLaughlin of the Mining Engineering department and the Montana Space Grant Consortium for providing funding for my research.

Finally, I extend my gratitude to Charles Linney. His piloting skills were crucial for collecting the data needed for modeling the Canary. Also, his vast knowledge of UAVs was a valuable resource when troubleshooting.

Table of Contents

ABSTRACT	II
DEDICATION	III
ACKNOWLEDGEMENTS	IV
LIST OF TABLES	VIII
LIST OF FIGURES.....	IX
LIST OF EQUATIONS	XIII
GLOSSARY OF ACRONYMS & TERMS	XVI
1. INTRODUCTION	1
1.1. <i>Background</i>	1
1.2. <i>Problem Statement</i>	2
1.3. <i>Overview of Obstacle Detection</i>	3
1.4. <i>Thesis Organization</i>	8
2. RELATED WORKS	10
2.1. <i>Commercially Available Systems</i>	10
2.2. <i>Published Literature</i>	16
3. TECHNICAL REVIEW	19
3.1. <i>Quadcopter Basics</i>	19
3.2. <i>SBUS Communication Protocol</i>	21
3.3. <i>Kalman Filter</i>	28
3.4. <i>Root Locus Technique</i>	30
4. MODELING	33
4.1. <i>Methodology</i>	34
4.2. <i>Data Processing</i>	36

4.3.	<i>Estimating the Canary's Position</i>	39
4.4.	<i>Step Response Results</i>	41
4.5.	<i>Deriving the Model Equation</i>	45
5.	CONTROLLER DESIGN	53
5.1.	<i>Overview</i>	53
5.2.	<i>Design Methodology</i>	53
5.3.	<i>Simulation Results</i>	57
5.4.	<i>Noise Resiliency</i>	61
5.5.	<i>Choosing a Controller</i>	64
6.	HARDWARE	68
6.1.	<i>Overview</i>	68
6.2.	<i>Testing Platforms</i>	70
6.3.	<i>RC Receiver & Transmitter</i>	74
6.4.	<i>Sensor Board</i>	75
6.5.	<i>MITM</i>	93
6.6.	<i>Final Prototype Design/Layout</i>	110
7.	PROTOTYPE TESTING	113
7.1.	<i>Methodology</i>	113
7.2.	<i>Results</i>	114
8.	CONCLUSIONS	119
9.	FUTURE WORK	123
	REFERENCES	125
	APPENDIX A: STANDARD COMMUNICATION PROTOCOLS	134
	APPENDIX B: PCB LAYOUTS	136
	APPENDIX C: TECHNICAL DRAWINGS	138

APPENDIX D: MATLAB SCRIPTS	142
APPENDIX E: SENSOR BOARD CODE	162
APPENDIX F: MITM CODE – RASPBERRY PI VERSION.....	176
APPENDIX G: MITM CODE – GPS & ACCELEROMETER VERSION	192
APPENDIX H: MITM CODE – MSP430G2553 & ULTRASONIC VERSION.....	205

List of Tables

Table I: Mavic 2 Pro sensor specifications	13
Table II: SBUS decoding values.....	25
Table III: SBUS encoding values	26
Table IV: SBUS channel naming scheme.....	28
Table V: Canary step input magnitudes.....	35
Table VI: Conversion factors per flight for latitude & longitude to meters	38
Table VII: Estimates of R	48
Table VIII: Ultrasonic linear regression values	79
Table IX: Sensor Board I2C commands	84
Table X: LCAS Sensor Board components	89
Table XI: Sensor Board MSP430G2553 pin mapping	93
Table XII: Sensor Board identifiers. I2C addresses, and locations	101
Table XIII: LCAS MITM Docking Board components	107

List of Figures

Figure 1: DJI Mavic 2 Pro rear stereovision cameras [5]	4
Figure 2: How the HC-SR04 ultrasonic range finding sensor measures distances [6]	5
Figure 3: tinyLiDAR single-point TOF LiDAR [15]	7
Figure 4: Velodyne Puck multi-point rotating LiDAR [13]	7
Figure 5: DJI Mavic Air 2.....	12
Figure 6: Visualization of the Mavic Air 2’s backward obstacle detection [20]	12
Figure 7: DJI Mavic 2 Pro’s collision avoidance system [21].....	14
Figure 8: Skydio 2 [24]	16
Figure 9: Rotating LiDAR-based collision avoidance system used in [29].....	17
Figure 10: Low-cost, multi-sensor system used in [9].....	18
Figure 11: Quadcopter motor “X” configuration, showing reaction torques [34]	19
Figure 12: Quadcopter motion [35]	20
Figure 13: SBUS frame structure.....	22
Figure 14: SBUS signal inverter	23
Figure 15: Traditional feedback control loop	30
Figure 16: Root locus and transient response of the open-loop system.....	31
Figure 17: Root locus and transient response of the closed-loop system	32
Figure 18: One-dimensional free-body diagram of a UAV flying towards a wall	33
Figure 19: Canary testing area, Leonard Field [44]	35
Figure 20: Flight 1 position estimate	42
Figure 21: Flight 2 position estimate	42
Figure 22: Flight 1 position versus time	43

Figure 23: Flight 2 position versus time	44
Figure 24: Flight 1 step responses	45
Figure 25: Flight 2 step responses	45
Figure 26: Results from the \mathbf{R} estimation using Flight 1 parameters	47
Figure 27: Results from the \mathbf{R} estimation using Flight 2 parameters	48
Figure 28: Model simulation with $\mathbf{R} = \mathbf{1.2849}$ using Flight 1 parameters	49
Figure 29: Model simulation with $\mathbf{R} = \mathbf{1.2849}$ using Flight 2 parameters	49
Figure 30: Model simulation with $\mathbf{R} = \mathbf{1.6270}$ using Flight 1 parameters	50
Figure 31: Model simulation with $\mathbf{R} = \mathbf{1.6270}$ using Flight 2 parameters	51
Figure 32: Phase I controller block diagram.....	54
Figure 33: Open-loop system root locus.....	55
Figure 34: Phase II closed-loop system root locus	56
Figure 35: Phase II closed-loop system step response.....	56
Figure 36: Scenario 1 simulation results.....	58
Figure 37: Scenario 2 simulation results.....	59
Figure 38: Scenario 3 simulation results.....	60
Figure 39: Scenario 4 simulation results.....	61
Figure 40: Case 1 simulation results for Phase I controller	62
Figure 41: Case 1 simulation results for Phase II controller.....	62
Figure 42: Case 2 simulation results for Phase I controller	63
Figure 43: Case 2 simulation results for Phase II controller.....	64
Figure 44: Case 1 simulation results for Phase II controller with a SMA	65
Figure 45: Case 2 simulation results for Phase II controller with a SMA	66

Figure 46: LCAS block diagram.....	69
Figure 47: MITM block diagram	69
Figure 48: Servo-based rover testing platform	70
Figure 49: Canary quadcopter platform.....	72
Figure 50: Matek F722-SE flight controller [48].....	73
Figure 51: Spedix ES30-HV electronic speed controller.....	74
Figure 52: FrSky X8R receiver.....	74
Figure 53: FrSky Taranis Q X7 transmitter	75
Figure 54: HC-SR04 ultrasonic range finding sensor [6]	77
Figure 55: Ultrasonic linear regression.....	80
Figure 56: tinyLiDAR TOF Range Finder Sensor [15]	81
Figure 57: LCAS Sensor Board's MSP430G2553	82
Figure 58: Flowchart of Sensor Board distance value processing.....	87
Figure 59: LCAS Sensor Board	89
Figure 60: Sensor Board power circuit schematic	90
Figure 61: Bidirectional voltage shifter schematic [56]	91
Figure 62: LCAS Sensor Board schematic	92
Figure 63: Location of the MITM in the layout of a quadcopter control system	93
Figure 64: MSP430G2553 version of the MITM on the SBUS-to-PWM Rover	96
Figure 65: Raspberry Pi 3 Model B [59]	99
Figure 66: Raspberry Pi GPIO pinout [60]	99
Figure 67: 3DR GPS module & its connection to the MITM.....	105
Figure 68: BMA280 accelerometer & interfacing board.....	105

Figure 69: LCAS MITM Docking Board	107
Figure 70: MITM Docking Board I2C bus schematic	108
Figure 71: MITM Docking Board SBUS communication bus schematic	108
Figure 72: LCAS MITM Docking Board schematic	109
Figure 73: LCAS-Canary mounting plate.....	110
Figure 74: LCAS Sensor Board mounting bracket.....	111
Figure 75: LCAS prototype layout on the Canary	112
Figure 76: LCAS prototype testing area	114
Figure 77: Canary position during LCAS testing flight.....	115
Figure 78: Canary input during LCAS testing flight	115
Figure 79: Prototype LCAS Test 1 results	116
Figure 80: Prototype LCAS Test 2 results	116
Figure 81: Prototype LCAS Test 3 results	117
Figure 82: Prototype LCAS Test 4 results	118
Figure 83: Prototype LCAS Test 3.5 results.....	120
Figure 84: General structure of a UART signal [62]	134
Figure 85: I2C bus adapted from [63].....	135
Figure 86: Overview of I2C protocol structure [63].....	135

List of Equations

Equation

(1)	5
(2)	5
(3)	25
(4)	27
(5)	28
(6)	28
(7)	29
(8)	29
(9)	29
(10)	29
(11)	30
(12)	31
(13)	33
(14)	33
(15)	34
(16)	34
(17)	34
(18)	38
(19)	38
(20)	38
(21)	39

(22)	40
(23)	40
(24)	40
(25)	40
(26)	40
(27)	40
(28)	40
(29)	41
(30)	41
(31)	41
(32)	46
(33)	46
(34)	46
(35)	46
(36)	46
(37)	46
(38)	46
(39)	46
(40)	47
(41)	47
(42)	51
(43)	51
(44)	52

(45)	52
(46)	52
(47)	54
(48)	56
(49)	57
(50)	57
(51)	66
(52)	67
(53)	71
(54)	71
(55)	71
(56)	71
(57)	72
(58)	77
(59)	78
(60)	78
(61)	79
(62)	88
(63)	88
(64)	103
(65)	104

Glossary of Acronyms & Terms

Term	Definition
Canary	260-millimeter quadcopter used for testing the LCAS
GPS	Global Positioning System
I2C	Inter-Integrated Circuit serial communication protocol
LCAS	Low-cost collision avoidance system
LiDAR	Light detection and ranging
MITM	Monkey-in-the-middle
PCB	Printed circuit board
RC	Radio control
RX	Received data
SBUS	Proprietary serial communication protocol developed by Futaba Corp.
SMA	Simple moving average
tinyLiDAR	Single-point LiDAR module
TOF	Time-of-flight
TX	Transmitted data
UART	Universal Asynchronous Receiver/Transmitter serial communication protocol
UAV	Unmanned aerial vehicle
Ultrasonic	Short for ultrasonic range finding sensor

1. Introduction

1.1. Background

Over the course of the last decade, interest in unmanned aerial vehicles (UAVs) has grown beyond the hobbyist level. Large-scale factories and warehouses have begun to use UAVs to monitor and track inventories [1]; while mining operations have seen the potential of using UAVs for surveying areas that are hazardous to employees [2] [3]. All of these operations share a common application: flying UAVs in indoor environments.

There are many challenges associated with flying UAVs in an indoor environment, such as the lack of access to the Global Positioning System (GPS) for aid in navigation and the strict boundaries of the environment itself. The obvious method to overcome these challenges is to have an experienced pilot, who has hours of practice flying in restrictive conditions and a steady hand on the controls. However, even the most skilled pilot would be limited by his or her field of vision and reaction time, when navigating indoor environments. This is where collision avoidance systems come in. Collision avoidance systems look to aid, or even supplant, the pilot in control of the UAV. Using an array of range finding sensors, the system can identify obstacles and modify the control algorithm of the UAV to avoid said obstacles.

There are commercial collision avoidance systems available, but the systems are expensive and limited in suppliers. Another downside to a commercial system is that typically the system is designed to work only on a specific platform, and cannot be easily transferred to another. Finally, most commercial collision avoidance systems lack full collision avoidance, choosing to forego obstacle detection in one or multiple directions. The most notable direction foregone is the upward direction since most UAVs are not intended to fly in upward restrictive environments.

1.2. Problem Statement

This work proposes that it is possible to design a custom, low-cost collision avoidance system (LCAS) with the modularity needed to be transferred between UAV platforms without significant modification to the new platform.

To lower the cost, while ensuring obstacle detection in every direction, the LCAS made use of hobbyist-grade range finding sensors alongside lower power microcontrollers and processors. The modularity of the LCAS is defined by two separate major components: a centralized processor and sensor modules. The centralized processor, referred to as the Monkey-in-the-Middle (MITM), captures and decodes the control signal coming into the UAV from a radio control (RC) receiver and modifies it before re-encoding and passing the signal onto the UAV's flight controller. The modifications are made by a custom feedback controller using distance measurements provided by sensor modules. The LCAS's sensor modules, referred to as the Sensor Boards, utilize embedded microcontrollers to control an array of range finding sensors and determine the smallest distance measurement from the array. One Sensor Board operates independently from another, allowing each direction of motion to have its own board.

In addition to the prototyping of the LCAS hardware, a one-dimensional model of a UAV was developed to aide in the design of the MITM's custom feedback controller for the forward direction. The model was derived by curve-fitting the response of the Canary quadcopter to a series of increasing step inputs. The Canary is a 260-mm quadrotor platform used for developing and testing the prototype LCAS. Using the computational software MATLAB, simulations were conducted that tested the model's response to the same step inputs used previously. The model was then validated by comparing the model's responses to those of the Canary quadcopter.

The overall goal of this work was to provide a proof of concept for and determine the feasibility of the LCAS for use on UAVs in indoor environments. To complete this work in a

reasonable amount of time, the proof of concept was done for only a single direction of motion: the forward direction. As a result, the methodology created can be used for developing future models and controllers for the remaining directions of motion.

1.3. Overview of Obstacle Detection

Within the context of use in the LCAS, obstacle detection is defined as the process of measuring the linear distance from the UAV platform to an obstacle. When maneuvering in a three-dimensional environment obstacle detection is needed for each direction about the center of a UAV. This means that a total of six directions must have obstacle detection. The following sections describe the obstacle detection methods considered for use in the LCAS.

1.3.1. Stereovision

Stereovision is an increasingly popular method used on commercial UAVs, such as on the DJI Mavic 2 Pro and Skydio 2. This method uses one or more cameras to capture images of the environment around a UAV. Through image processing, certain information about obstacles can be extracted, notably the relative angle of the UAV to the obstacle. However, it can be computationally difficult to estimate the location, size, and distance to an obstacle; that is not even taking into consideration the quality of the image, such as lighting and any restriction on the cameras' field-of-view [4].

Figure 1 shows the usage of stereovision by the DJI Mavic 2 Pro for backward direction obstacle detection.



Figure 1: DJI Mavic 2 Pro rear stereovision cameras [5]

While recent advancements in the field of image processing have made stereovision more consistent and reliable, the implementation is complex and can be difficult in low-cost systems like the LCAS.

1.3.2. Ultrasonic Sensing

A classic method for obstacle detection is sonar via ultrasonic range finding sensors (ultrasonic for short). The use of ultrasonic sound waves in navigation has been around for decades and can be found in many transportation systems [4]. Ultrasonics emit bursts of high frequencies that are bounced back as an echo when encountering an obstacle. The time between the burst emission and detection of an echo can be used to calculate a distance to the obstacle, since the speed of sound is known [6] [7]. Figure 2 depicts the ultrasonic sensing method used by the HC-SR04 ultrasonic.

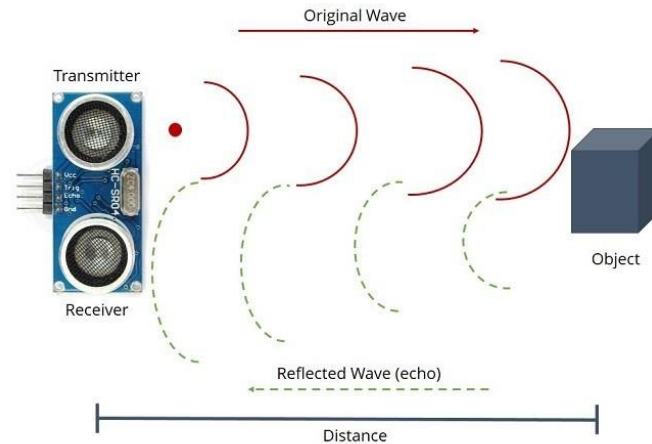


Figure 2: How the HC-SR04 ultrasonic range finding sensor measures distances [6]

Ultrasonics are considered very reliable, not being affected by dust or atmospheric conditions. Also, ultrasonics have rather large sensing ranges and areas, such as the 4-m range offered by the HC-SR04 ultrasonic [4] [8]. However, ultrasonics are not without faults. Porous materials tend to absorb sound waves thus reducing the effectiveness and measurement accuracy of the sensor when encountering such obstacles [9]. Finally, compared to other methods of obstacle detection, ultrasonic sensing is slow due to the operating speed being limited to the speed of sound. For example, consider there to be an object at 2 m from an ultrasonic. The operating time required to measure that distance can be found using the following equation, adapted from [8]:

$$t = \frac{d}{v_{sound}} \quad (1)$$

Substituting 2 m for d and 343 m/s for v_{sound} , the operating time is

$$t = \frac{(2 \text{ m})}{343 \text{ m/s}} = 5.83 \text{ ms} \quad (2)$$

An operating time of 5.83 ms seems reasonable, but that is only for one sensor. A collision avoidance system like the LCAS will need an ultrasonic for each of a UAV's six directions of

motion. Additionally, each of the six ultrasonics would have to be operated sequentially to prevent the sensors from interfering with each other, as low-cost ultrasonics like the HC-SR04 do not have the means to differentiate between its own echo or that of another ultrasonic.

Extrapolating the operating time from Equation (2) to account for an additional five ultrasonics, the total operating time would be approximately 35 ms. Considering the speeds at which a UAV can fly, a 35-ms operating time is too slow for providing distance measurements to the LCAS. Either a different obstacle detection method is needed or a different type of ranging sensor is needed to provide distance measurements in the interim between ultrasonic distance measurements.

1.3.3. TOF LiDAR Systems

Another common method for obstacle detection is the use of light detection and ranging, or LiDAR. A LiDAR operates on the concept of light reflection. When triggered, a LiDAR emits an infrared laser light pulse and measures the amount of time it takes for the reflected pulse to be detected [7] [10]. Different types of LiDAR are defined by how the time measurements are handled. In regards to the LCAS, the type of LiDAR under consideration is time-of-flight (TOF), which derives a distance value from the time measurement using the same methodology as ultrasonic sensing.

Unlike ultrasonics, a LiDAR is quick to make distance measurements since it operates at the speed of light. However, a LiDAR is susceptible to interference from dust and atmospheric conditions. Also, the infrared pulses used by a LiDAR can be adversely affected by objects with a sheen or a property that alters how the infrared pulse is reflected [9] [11] [12].

It might be assumed that a LiDAR system is an expensive option for use in the LCAS, given most professional applications use a rotating platform and high-end optics. For example,

the Velodyne Puck comes in at around \$9000 and features 16 measurement channels on a rotating platform that provide up to 300,000 points per second [13] [14]. However, there are lower-cost versions of LiDAR systems available. Instead of using a single multi-point, rotating LiDAR like the Puck, multiple single-point, static TOF LiDAR sensors can be used to measure distances to obstacles. Single-point TOF LiDAR sensors are significantly cheaper than the more popular multi-point LiDAR sensors but the lower monetary cost comes at the cost of ranging distance and measurement accuracy [15]. The tinyLiDAR is a single-point TOF LiDAR that comes in at a price around \$25 [16], significantly more cost effective than the \$9000 Puck [14]. Furthermore, the tinyLiDAR features a built-in microcontroller that handles the complex interface of the infrared TOF flight sensor. The tinyLiDAR and Velodyne Puck are shown in Figures 3 and 4.

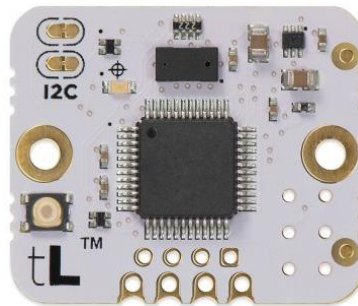


Figure 3: tinyLiDAR single-point TOF LiDAR [15]



Figure 4: Velodyne Puck multi-point rotating LiDAR [13]

For use in the LCAS, the decrease in ranging distance and measurement accuracy associated with single-point TOF LiDARs is negligible due to the tight operating conditions of the LCAS and the availability of additional distance-measuring sensors.

1.3.4. Complementary Sensors

Ultimately, the obstacle detection method chosen for the LCAS was to combine ultrasonic sensing with a TOF LiDAR system. This multi-sensor system allowed for the faults of each individual method to be addressed by the other, thus the complementary nature. Ultrasonics suffer from inaccuracies when bouncing signals off of porous materials but are not affected by the sheen of materials' surfaces. Also, in general, ultrasonics are slow in operation being limited to the speed of sound for operation. LiDARs do not struggle to measure distances to porous materials but are prone to inaccuracies introduced by unpredictable light reflections off of materials with a sheen. Furthermore, LiDARs operate significantly faster than ultrasonics because by using infrared light LiDARs operate at or near the speed of light.

There were two main reasons behind the choice to combine ultrasonic sensing with a TOF LiDAR. The first reason was the availability of low-cost sensors for each method. The HC-SR04 ultrasonic and tinyLiDAR used in the LCAS came in at around \$4 [17] and \$25 [16], respectively. The second reason was the precedented use of the two methods in complementary sensor systems, such as in [9] and [18].

1.4. Thesis Organization

Section 2, **Related Works**, takes a look at three, commercially-available collision avoidance systems. Next, a pair of published papers are reviewed for how the collision avoidance systems proposed in each performed. Section 3, **Technical Review**, provides an overview of technologies and methods used when developing the LCAS that are mentioned throughout this

document. Section 4, **Modeling**, is an in-depth discussion into the development and validation of the Canary quadcopter model. This section also provides the reasoning behind the decision to focus on a single direction of motion. Section 5, **Controller Design**, covers the development of the LCAS's forward direction feedback controller. Section 6, **Hardware**, takes a highly-detailed look at the hardware used in the LCAS. Furthermore, the section details the operational concepts behind the system's programming algorithm. Section 7, **Prototype Testing**, covers the testing procedure and results for the prototype LCAS. Section 8, **Conclusions**, discusses the results from all aspects of this work and draws conclusions on the feasibility of the LCAS. Finally, Section 9, **Future Work**, provides suggestions on how to improve the LCAS.

Multiple appendices are included with this thesis. **Appendix A** provides an overview of the UART and I2C serial communication protocols. **Appendix B** contains the printed circuit board layouts for the LCAS's Sensor Board and MITM Docking Board. Technical drawings for the 3D printed components of the Canary are found in **Appendix C**. The MATLAB scripts used throughout this work are presented in **Appendix D**. The code behind the LCAS Sensor Boards is given in **Appendix E**. **Appendices F, G, and H** contain the code for the three different versions of the MITM.

2. Related Works

As the use of UAVs has expanded outside of the hobbyist space and into the commercial space, there has been a significant increase in the research and development of collision avoidance systems. Most of the research has focused on the development of systems for autonomous flight, such as on the DJI Mavic 2 Pro and the Skydio 2. These systems tend to be platform specific and expensive.

This section reviews three examples of commercially-available collision avoidance systems. After the commercial systems review, a look is taken at recent published literature on custom collision avoidance systems like the LCAS.

2.1. Commercially Available Systems

2.1.1. DJI Mavic Air 2

Intended for hobbyists and entry-level professionals looking for a smooth and reliable platform for aerial cinematography and video streaming, the DJI Mavic Air 2's collision avoidance system is utilized to stabilize the UAV while in flight [19]. The Mavic Air 2, released in April 2020, is capable of sensing objects in three directions: forward, backward, and downward. The forward and backward directions feature dual vision sensors capable of measuring distances between 0.35 m and 47.2 m. The forward dual vision sensors have a field of view of 71° and 56° horizontally and vertically, respectively. The backward sensor is more restricted in its field of view with 44° and 57° horizontally and vertically, respectively. For the downward direction the Mavic Air 2 combines infrared TOF sensors with another pair of vision sensors. Using the dual vision sensors alone, the UAV has a hovering range of 0.5 m to 60 m; however, by adding data from the infrared TOF sensors the UAV can increase the precision of its hovering position at the cost of lowering the range to between 0.5 m and 30 m [19] [20].

The Mavic Air 2 is only capable of full autonomy when using DJI's ActiveTrack 3.0. When in this mode the UAV is set to follow a predetermined target, such as a runner or a vehicle, and the collision avoidance system is used to detect and avoid obstacles in the tracking path. If an obstacle is detected, the Mavic Air 2 will attempt to fly around it and will hover in place if a suitable path cannot be detected [19].

The collision avoidance system is further utilized by the Mavic Air 2's Advanced Pilot Assistance System (APAS). This system complements a user piloting the UAV manually by using data from the sensors to generate a real-time map of its surroundings and determine an appropriate path to avoid any obstacles detected while in flight [19] [20].

A major drawback of the Mavic Air 2's collision avoidance lies in the type of sensors and the absence of collision avoidance in the left, right, and upward directions. By only using vision and infrared sensors, the UAV is susceptible to measurement errors when an obstacle's surface has a sheen or is reflective. Under low light conditions the vision sensors are not able to work, leaving only the downward infrared TOF sensor to provide distance measurements. Also, the Mavic Air 2's collision avoidance system is not capable of sensing small objects, such as electrical wires, and tracking moving objects, such as people [19].

At the time of writing the UAV is available, from DJI, for \$799 [20]. The Mavic Air 2 is shown in Figure 5 and an operational visualization of the UAV's backward sensors is given in Figure 6.



Figure 5: DJI Mavic Air 2

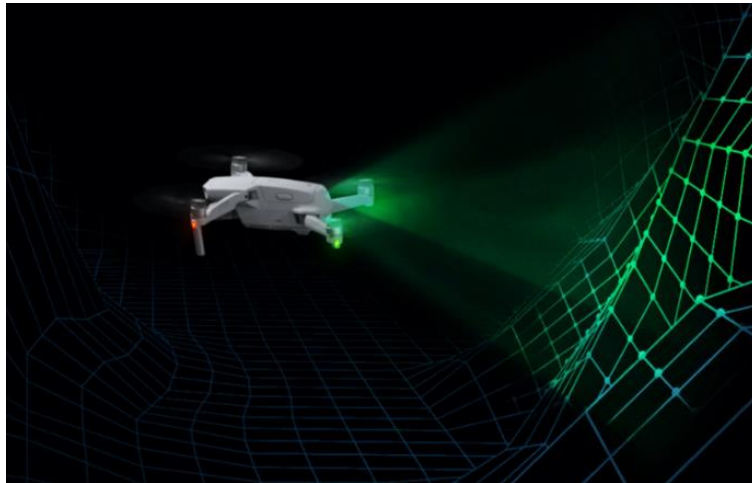


Figure 6: Visualization of the Mavic Air 2's backward obstacle detection [20]

2.1.2. DJI Mavic 2 Pro

DJI's Mavic 2 Pro was released in August 2018. [21]. The Mavic 2 Pro is DJI's first UAV capable of obstacle detection in all six directions. Dual vision sensors are used in the forward, backward, and downward directions, while single vision sensors are used in the lateral (left-right) directions. Dual, three-dimensional infrared sensors are used on both the upward and

downward directions. The measurement ranges and field of view angles for all directions are summarized in the Table I, adapted from [22].

Table I: Mavic 2 Pro sensor specifications

Direction	Sensor type(s)	Range [m]	Horizontal FOV [°]	Vertical FOV [°]
Forward	Dual vision	0.5 – 40	40	70
Backward	Dual vision	0.5 – 32	60	77
Downward	Dual vision Dual infrared	0.5 – 22	n/a	n/a
Upward	Dual infrared	0.1 – 8	n/a	n/a
Lateral	Single vision	0.5 – 10	80	65

The Mavic 2 Pro expands upon the Mavic Air 2's autonomous flight capabilities by introducing additional flight modes that utilize the increased number of sensors and sensing directions. The first mode is Waypoint Navigation. In this mode the user draws a path for the UAV to follow by marking GPS waypoints in DJI's mission planning software. During the flight, the UAV will attempt to navigate to the waypoints and utilize the collision avoidance system to detect any obstacles in the path. If obstacles are detected the UAV will scan for the most appropriate path that will avoid the obstacle [21] [22]. The second mode is an autonomous Return to Home (RTH). When switched into RTH, the Mavic 2 Pro automatically creates a flight path to return to the last known home position using GPS data. Once following the flight path, the UAV uses its collision avoidance system to scan for obstacles and adjust the path as necessary [21].

Furthermore, the Mavic 2 Pro offers stability and tracking accuracy in ActiveTrack 2.0. By utilizing its obstacle detection sensors, the UAV can track targets at high speeds and maintain a constant distance. When tracking, the UAV can detect and actively avoid obstacles in the forward and backward direction, provided the collision avoidance system can determine an

appropriate path. This methodology is the same one used in the newer Mavic Air 2's ActiveTrack 3.0.

Finally, the Mavic 2 Pro's APAS operates the same as on the Mavic Air 2. When under manual control, the APAS scans the environment looking for obstacles in the forward and backward directions. Upon detection of obstacles, the system determines the most appropriate path to avoid those obstacles and overrides manual control in order to follow the path [21] [22]. Also, the Mavic 2 Pro shares the same faults of the Mavic Air 2, being susceptible to sensor failure in low-light conditions.

At time of this writing, the UAV is available directly from DJI for \$1599 [22]. The Mavic 2 Pro is shown in Figure 7 with the UAV's collision avoidance system sensors labeled.

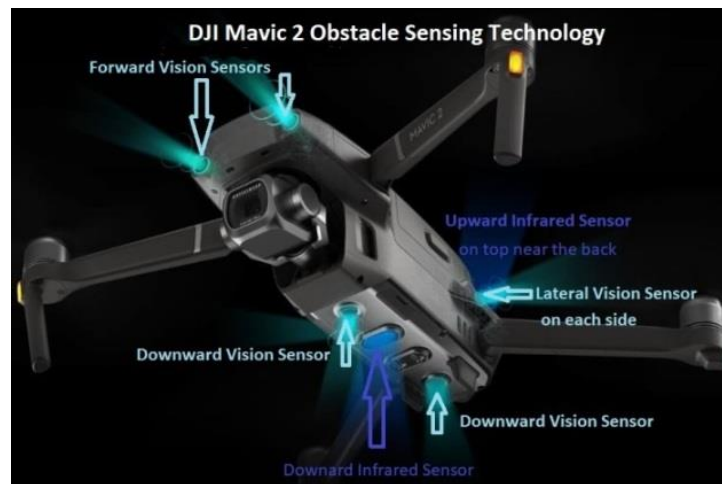


Figure 7: DJI Mavic 2 Pro's collision avoidance system [21]

2.1.3. Skydio 2

One of the main competitors to DJI's Mavic 2 Pro is the Skydio 2. The intended use of the Skydio 2, released in October 2019, is for capturing smooth, cinematic footage of actively moving targets. The UAV is known for superior camera stability and excellent autonomous navigation [23]. The Skydio 2's collision avoidance system makes use of six professional-grade,

4K cameras for building a three-dimensional map of the UAV's environment. The cameras are arranged in trinocular configurations on the top and bottom sides of the Skydio 2. Each camera has a 200° field of view and via the Skydio Autonomy Engine the UAV's navigation system can build a 360° model of its environment. The collision avoidance system uses the model to predict the changes in the environment and makes decisions on any changes to the UAV's flight path. All of this is done 500 times per second, with the cameras providing 30 frames per second. The measurement range of the Skydio 2's cameras is not specified, though, the UAV is reported to be capable of tracking and following targets at a maximum height of 8 m, which can be increased to 16 m for larger targets, such as vehicles [23] [24].

The Skydio 2's collision avoidance system is, by default, always enabled. Therefore, when the UAV is under manual control the system overrides the pilot's input when the UAV needs to avoid an obstacle. While both the DJI Mavic Air 2 and Mavic 2 Pro were limited to forward and backward obstacle detection during manual flight, the Skydio 2 is not so limited, allowing the pilot to have obstacle avoidance assistance in all six directions [24] [25].

The only drawbacks of the Skydio 2 are that the UAV's navigation cameras are not able to function properly in low-light conditions or darkness and cannot detect objects smaller than 12 mm in diameter, much like both DJI UAVs [23].

At the time of this writing the Skydio 2 is available from Skydio for \$999, with an additional \$149 for an RC controller to enable manual control [24]. The Skydio 2 is shown in Figure 8.



Figure 8: Skydio 2 [24]

2.2. Published Literature

Significant amounts of research have gone into the development of collision avoidance systems for UAVs. As mentioned previously most of the research has been for systems that work in conjunction with, or are a component of, autonomy systems, such as [26], [27], and [28]. However, there has been literature published on the concept of low-cost, collision avoidance systems providing aid to a pilot when a UAV is under manual control.

Described in [29] is a low-cost system that utilized a rotating TOF LiDAR on top of the UAV to detect obstacles. When in operation, the LiDAR scanned the environment around the UAV to produce a constantly updating 360° scan. The scan data were split into eight zones that the system's obstacle detection algorithm classified on a threat scale. If the same threat was detected three times in a row in the same zone, then the system defined the threat as an obstacle to avoid. The system then chose a zone with the lowest threat level to move into. The reactionary force used to move to the safest zone was determined based on the threat level of the obstacle and the distance of the UAV to the obstacle. While the system was able to avoid collisions, it limited the UAV to low operating speeds and small roll and pitch angles [29]. Furthermore, by

using a rotating LiDAR the system was only able to detect obstacles in the forward, backward, and lateral directions. Finally, a single sensor offered no redundancy and in-flight error checking that multiple sensors would. Figure 9 shows the system.



Figure 9: Rotating LiDAR-based collision avoidance system used in [29]

A system similar to the LCAS is described in [9]. The system attempted to improve upon simple collision avoidance systems that drive the UAV in the opposite direction of detected obstacles by applying techniques used in SLAM-based systems. A SLAM, or simultaneous localization and mapping, algorithm is a process that uses measurements from range finding sensors to construct a virtual map of an environment and determine the location of the device running the SLAM algorithm. Significant research, as detailed in [30] and [31], has gone into developing SLAM algorithms. However, SLAM algorithms can require considerable computational power sometimes using processors on the level of desktop computers, such as in [31].

By utilizing the complementary nature of ultrasonic and infrared sensors, the system in [9] had great reliability and could operate in a variety of conditions. An inertial measurement unit (IMU) and optical flow sensors were used in conjunction with the ultrasonic and infrared

sensors to improve distance estimations via sensor fusion. Sensor fusion is the process of combining data from a variety of sensors to produce a single estimate that has less uncertainty than estimates created from the individual sensors would [32]. The system then used the distance estimations and the locations of the sensors to build a model of its environment, divided into 12 sectors. The sectors were then ranked by threat level and the system chose the safest sector to move into. A PID (proportional, integral, and derivative) controller was used to control the strength of the reaction, with the proportional and derivative parts adjusted according to the distance in order to prevent overshooting and improve stability. The main goal of the system was to allow autonomous operation of a UAV, but the system could be used to create a pilot assistance system [9]. The drawbacks of the system were the requirement of replacing a flight controller with the system's main processing unit and the complexity of the collision avoidance, limiting the modularity of the system. The system is shown in Figure 10.



Figure 10: Low-cost, multi-sensor system used in [9]

3. Technical Review

This section provides explanations and background information on the technologies and methods used in the development of the LCAS and mentioned throughout this document. Key concepts include quadcopter basics, the decoding and encoding of SBUS signals, sensor fusion via a Kalman filter, and control system design via root locus.

3.1. Quadcopter Basics

The main platform of development for the custom collision avoidance system was on a UAV with a quadrotor arrangement. More commonly known as a quadcopter, this UAV platform has the advantage of combining high reliability and stability with simplistic design and agile maneuverability [33].

A quadcopter utilizes an X-configuration for the placement of four propellers on four motors. Two propellers spin clockwise, and the other two propellers spin counterclockwise. The propeller pairs are placed diametrically about the quadcopter's center, allowing the torque from each motor to balance out the torque of the corresponding motor. By balancing the motor torques the X-configuration yields greater stability and control when the quadcopter is in flight [33] [34]. Figure 11 shows the quadcopter X-configuration and motor arrangement.

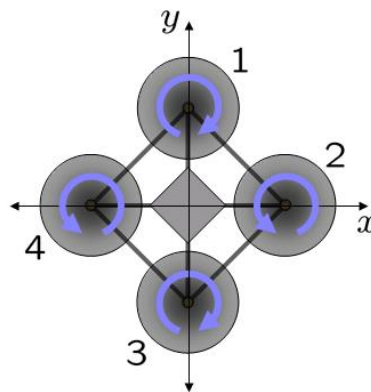


Figure 11: Quadcopter motor “X” configuration, showing reaction torques [34]

The direction of movement for a quadcopter is controlled by adjusting the speed of the motors independently. Movement of a quadcopter is defined as follows: pitch for the forward and backward directions, roll for the left and right directions, yaw for rotation about the center, and altitude for the upward and downward directions. For pitch and roll a quadcopter will increase the speed of one or two motors and decrease the speed of one or two diametrically opposing motors. For yaw a quadcopter will increase the speed of same-spinning direction propellers. Finally, for altitude a quadcopter increases or decreases the speed of all motors, equally. Figure 12 provides a pictorial depiction of the changing motor speeds needed for quadcopter movement.

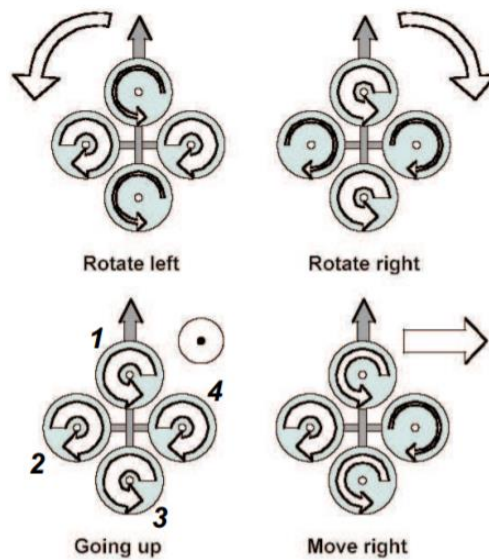


Figure 12: Quadcopter motion [35]

Control of a quadcopter is mainly handled by a flight controller. A flight controller receives user input from an RC receiver and translates the input into individual motor commands. Individual motor commands are sent to the corresponding motor's electronic speed controller (ESC), which translate the commands into voltage levels that dictate the rotation speed

of the connected motor. For a depiction of the layout of a quadcopter control system refer to Figure 63 in Section 6.5.

3.2. SBUS Communication Protocol

The SBUS protocol is a specialized serial communication protocol developed by Futaba Corporation for use in RC devices. The protocol is designed to condense 16 individual channels into a single data frame that can then be transferred over a single line.

SBUS is based on the UART communication protocol (see Appendix A) but uses inverted voltage levels and a specialized 100,000 baud rate. Additionally, SBUS bytes are structured with the most significant bit first, differing from UART's byte structure of least significant bit first. A standard SBUS transmission is comprised of 25 bytes, which all together form the SBUS frame. The first byte is the header, or start byte, which identifies the beginning of the SBUS frame. The following 22 bytes contain the data for the 16 channels in the format of one start bit, eight data bits, one even parity bit, and two stop bits. The 23rd byte contains information for two digital channels and the "frame lost" and "failsafe" flags. The final byte is the footer, or end byte, which identifies the end of the SBUS frame [36]. The SBUS frame structure is summarized in Figure 13, adapted from [37] and [38].

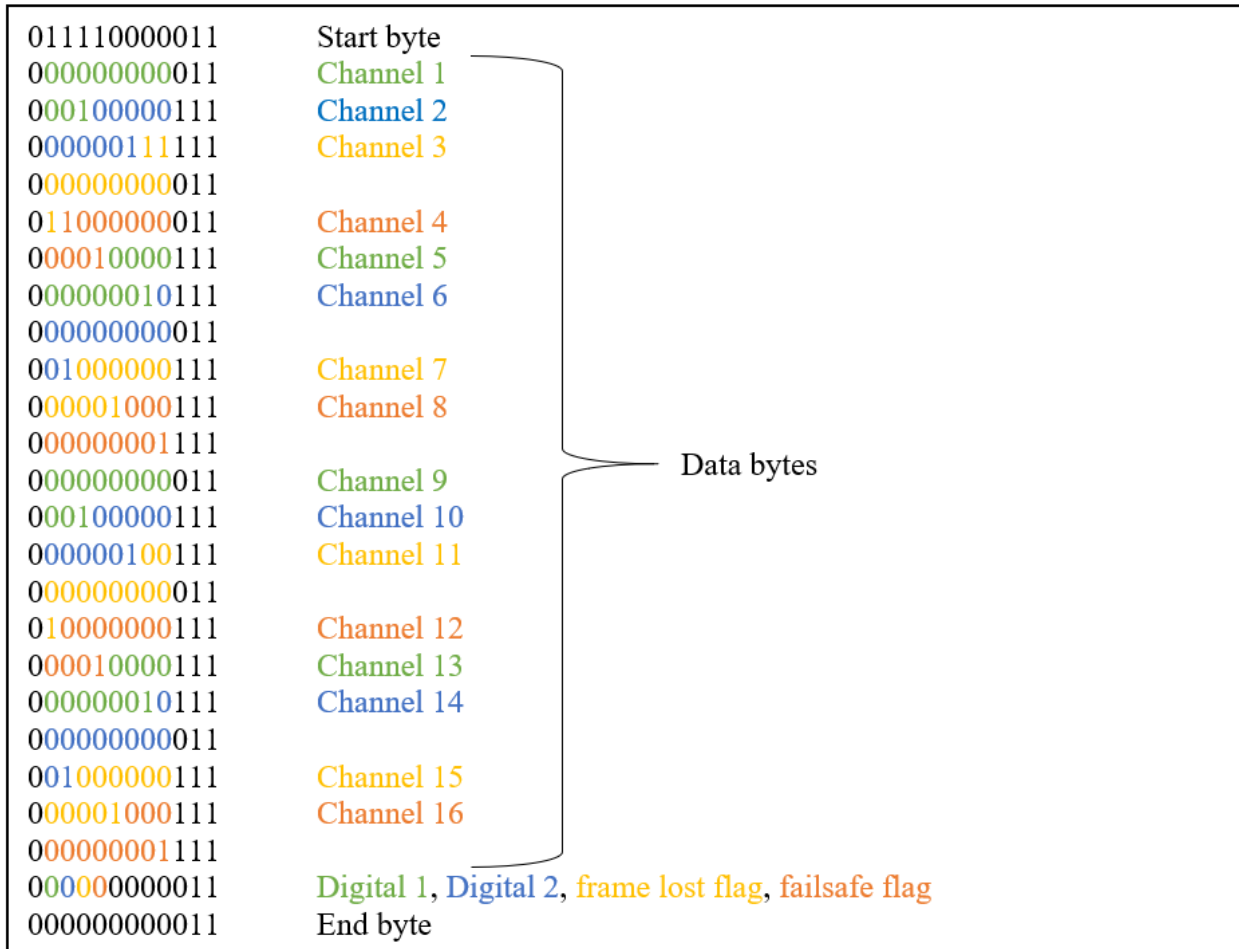


Figure 13: SBUS frame structure

3.2.1. Decoding

Since SBUS is a non-standard communication protocol, the ability to read and write in the protocol is not a feature on the processor used for the MITM. That means in order to interpret an SBUS signal a processor must be able to read and decode the frame before it can modify any of the channel values.

Before the MITM can even receive an SBUS signal the signal has to be inverted. The reason for the inversion is that SBUS is based on an inverted UART signal (see Appendix A). To invert the SBUS signal, a 2N7002 metal-oxide-semiconductor field-effect transistor (MOSFET)

was used. A hardware solution was chosen over a software solution for the SBUS inversion because of the availability of a UART bus on the processors used for the MITM. A schematic of the SBUS inverter is shown in Figure 14.

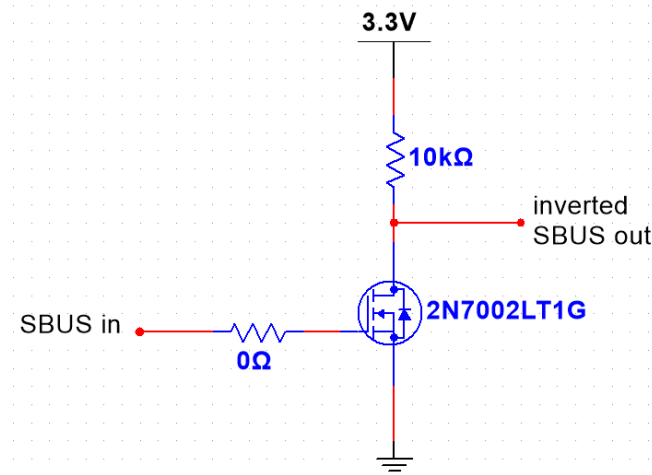


Figure 14: SBUS signal inverter

As shown in the figure the original SBUS signal is passed into the gate of the 2N7002 MOSFET and the inverted SBUS signal is output at the drain. The inversion works by the MOSFET adjusting current flow based on the original signal's voltage levels. When the input voltage is high the MOSFET will allow the current to flow to ground, resulting in the output signal being pulled low. When the input voltage is low the MOSFET will be disabled, allowing the current to flow to the output where the signal will be pulled high by the 10-kΩ pullup resistor.

With the SBUS signal inverted, the MITM can now read the signal like it would any standard UART signal using the MITM processor's built-in UART bus; however, the UART bus must be initialized to read the SBUS signal at the specialized 100,000 baud rate. With the baud rate accounted for the MITM captures 50 bytes, or two SBUS frames worth of data. Since the SBUS start and end bytes are predetermined, the MITM searches the 50 bytes for an end byte

and then searches for a start byte 24 bytes before the found end byte, thus mapping a single SBUS frame. Additionally, by capturing two frames worth of bytes the MITM is guaranteed to capture at least one frame, thus eliminating concerns over clock synchronization and clock drift.

Once an SBUS frame is mapped the MITM checks if the new frame is different from the previous frame and if so the MITM can begin parsing the 22 data bytes for the 16 individual channels. If the frame is the same as the previous frame then the MITM uses the channel values from the previous frame, skipping the decoding process for the new frame. The first step is to perform a bit endian swap on each byte because the UART bus reads the bytes as having the least significant bit first but SBUS bytes are structured with the most significant bit first. Next since the data for individual channels is comprised of 11 bits and spaced across multiple bytes, the parsing must operate at the bit level, using bitshifting and bitmasking. For example, to parse the data for Channel 1 the entire first data byte (8 bits) and 3 bits from the second data byte are needed. For simplicity the bitshifting and bitmasking values for each channel are detailed in Table II. Note that the “Byte Index” column does not apply to the channels explicitly. Rather the values refer to the bytes that contain information for the channels.

Table II: SBUS decoding values

Channel	Low bitshift	Low bitmask	Mid bitshift	Mid bitmask	High bitshift	High bitmask	Byte Index
1	5	0xE0	3	0xFF	0	0x00	0
2	2	0xFC	6	0x1F	0	0x00	1
3	7	0x80	1	0xFF	9	0x03	3
4	4	0xF0	4	0x7F	0	0x00	4
5	1	0xFE	7	0x0F	0	0x00	5
6	6	0xC0	2	0xFF	10	0x01	7
7	3	0xF8	5	0x3F	0	0x00	8
8	0	0xFF	8	0x07	0	0x00	9
9	5	0xE0	3	0xFF	0	0x00	11
10	2	0xFC	6	0x1F	0	0x00	12
11	7	0x80	1	0xFF	9	0x03	14
12	4	0xF0	4	0x7F	0	0x00	15
13	1	0xFE	7	0x0F	0	0x00	16
14	6	0xC0	2	0xFF	10	0x01	18
15	3	0xF8	5	0x3F	0	0x00	19
16	0	0xFF	8	0x07	0	0x00	20

Using the information from the table, the following equation, in the Python coding language syntax, was used to determine the channel values:

$$\begin{aligned}
channel[k] = & (byte[ndx[k]] \& highbitmask[k]) \ll highbitshift[k] + \\
& + (byte[ndx[k + 1]] \& midbitmask[k]) \ll midbitshift[k] \\
& + (byte[ndx[k + 2]] \& lowbitmask[k]) \gg lowbitshift[k]
\end{aligned} \tag{3}$$

where *channel* is a unitless integer value ranging from 172 to 1811, *byte* is a data byte from the SBUS frame, and *ndx* is the byte index from Table II.

The final step is to perform a bit endian swap on the individual channel values to change from little to big endian. This was done because the SBUS protocol encodes the channel values as little endian across the data bytes.

As a result of the decoding process, the channel values take on an integer range of 172 to 1811. Thus, the SBUS minimum, SBUS neutral, and SBUS maximum values are 172, 992, and 1811, respectively.

3.2.2. Encoding

Any data being transmitted to the flight controller must be in the SBUS format.

Therefore, the MITM must encode the individual channel values back into bytes that when output over the UART bus take the form of an SBUS frame.

The first step is to perform a bit endian swap on the individual channels, changing the bit order from big to little endian. This is the reverse of the final step of the decoding procedure.

Secondly, the start byte is added as the first byte. Next the channel values are split across the data bytes using bitshifting and bitmasking. The values used for the bit operations are summarized in

Table III.

Table III: SBUS encoding values

Data byte	First channel	First channel bitmask	First channel bitshift	Second channel	Second channel bitmask	Second channel bitshift
1	0	0x00	0	0	0xFF	3
2	0	0xE0	5	1	0x1F	6
3	1	0xFC	2	2	0x03	9
4	2	0x00	0	2	0xFF	1
5	2	0x80	7	3	0x7F	4
6	3	0xF0	4	4	0x0F	7
7	4	0xFE	1	5	0x0F	10
8	5	0x00	0	5	0xFF	2
9	5	0xC0	6	6	0x3F	5
10	6	0xF8	3	7	0x07	8
11	7	0xFF	0	8	0x00	0
12	8	0x00	0	8	0xFF	3
13	8	0xE0	5	9	0x1F	6
14	9	0xFC	2	10	0x03	9
15	10	0x00	0	10	0xFF	1
16	10	0x80	7	11	0x7F	4
17	11	0xF0	4	12	0x0F	7
18	12	0xFE	1	13	0x01	10
19	13	0x00	0	13	0xFF	2
20	13	0xC0	6	14	0x3F	5
21	14	0xF8	3	15	0x07	8
22	15	0xFF	0	16	0x00	0

Using the information from Table III, the data byte values can be found using the following equation, written in the Python coding language syntax:

$$\begin{aligned}
\text{byte}[k] = & ((\text{channel}[\text{chan1}[k]] \ll \text{chan1BS}[k]) \& \text{chan1BM}[k]) \\
& | ((\text{channel}[\text{chan2}[k]] \gg \text{chan2BS}[k]) \& \text{chan2BM}[k])
\end{aligned}
\tag{4}$$

where *channel* is a channel value, *chan1* is the first channel index, *chan2* is the second channel index, *chan1BS* is the first channel bitshift, *chan1BM* is the first channel bitmask, *chan2BS* is the second channel bitshift, and *chan2BM* is the second channel bitmask.

The second to last byte is set to zero and the final byte is set as the end byte. Finally, with all the data in the SBUS frame the frame's 25 bytes are endian swapped. The reason for doing this was because the UART module on the MITM's processor transmits the lowest bit first but the SBUS bytes are read with the highest bit first by the Canary's flight controller.

With encoding finished, the MITM outputs the SBUS frame through another signal inverter like the one in Figure 14. However, 5 V is used on the MOSFET's drain rather than 3.3 V. It would later be discovered that the shift in voltage was not needed since SBUS voltage levels are at 3.3 V rather than 5 V.

3.2.3. Channel Naming Scheme

Throughout this work the two different SBUS signals, received and transmitted, will be referred to as RX and TX, respectively. Furthermore, the individual SBUS channels were given specific names that will be referenced throughout this work. The channel names were derived from standard channel names for UAV control, such as *Thr* for throttle and *Ele* for elevator, as well as custom names used in this work, such as *LOG* for enabling data logging. For clarity and simplicity, the full naming scheming is detailed in the Table IV, with standard SBUS channel names denoted by an asterisk.

Table IV: SBUS channel naming scheme

Channel	Name	Description
1	<i>Thr</i>	Throttle; altitude (upward/downward) control*
2	<i>Ail</i>	Aileron; roll (left/right) control*
3	<i>Ele</i>	Elevator; pitch (forward/backward) control*
4	<i>Rud</i>	Rudder; yaw (rotation) control*
5	<i>ARM</i>	System arm/disarm*
6	<i>Hld</i>	Altitude hold enable
7	<i>LOG</i>	Data logging enable
8	<i>sbEN</i>	Enables communications with the Sensor Boards & feedback system
	<i>vEN</i>	Overrides <i>Ele</i> with the step magnitude from VEL; used only during modeling data collection
9	<i>ctrlEN</i>	Enables LCAS feedback controllers; <i>sbEN</i> has to be enabled first
10	<i>VEL</i>	Controls magnitude of <i>Ele</i> step; used only during modelling data collection
11	n/a	Unused
12	n/a	Unused
13	n/a	Unused
14	n/a	Unused
15	n/a	Unused
16	n/a	Unused

3.3. Kalman Filter

To aid in the development of the Canary model data fusion was used to combine GPS and accelerometer data into a position estimate that was more accurate and consistent than an estimate based on the individual sensors. A common method of data fusion, the Kalman filter is a recursive algorithm that estimates unknown states, or variables, based on an estimation of a joint probability distribution over the known states for each sampling period [39] [40]. There are two stages to the Kalman filter: prediction/extrapolation and update.

3.3.1. State-space Model

To begin with, a state-space model has to be derived for the system. The system model used is given by the following equations adapted from [41, pp. 123-124]:

$$x_{k+1} = Ax_k + Bu_k + w_k \quad (5)$$

$$y_k = Cx_k + v_k \quad (6)$$

where x is the state vector, y is the output vector, u is the input vector, A is the system matrix, B is the input matrix, C is the output matrix, and both w_k and v_k are jointly Gaussian noise vectors. Since there were no deterministic inputs used in the modeling of the Canary the Bu_k term was ignored.

3.3.1.1. Prediction Stage

The prediction stage of the Kalman filter is focused entirely on the state estimation. The state estimate from the previous sampling period is used to produce a state estimate for the current sampling period. The predicted state estimate and the predicted estimate covariance were produced using the following equations, adapted from [39] and [40],

$$\hat{x}_{k+1}^- = A\hat{x}_k \quad (7)$$

$$P_{k+1}^- = AP_kA^T + Q \quad (8)$$

where \hat{x}_{k+1}^- is the predicted state estimate, \hat{x}_k is the previous state estimate, P_{k+1}^- is the predicted estimate covariance, P_k is the previous estimate covariance, and Q is the process noise covariance matrix. The process noise covariance matrix is associated with the level of uncertainty, q , towards the system measurements.

3.3.1.1. Update Stage

The update stage of the Kalman filter uses the measurements of the current sampling period to refine the state estimate and estimate covariance matrix produced by the prediction stage. During this stage the optimal Kalman gain is calculated to aid in the refinements. The update stage equations are as follows, adapted from [39] and [40]:

$$K_{k+1} = P_{k+1}^- C^T [C P_{k+1}^- C^T + R]^{-1} \quad (9)$$

$$\hat{x}_{k+1} = \hat{x}_{k+1}^- + K_{k+1} [y_{k+1} - C \hat{x}_{k+1}^-] \quad (10)$$

$$P_{k+1} = [I - K_{k+1}C] P_{k+1}^- \quad (11)$$

where K_{k+1} is the optimal Kalman gain, \hat{x}_{k+1} is the updated state estimate, and R is the measurement noise covariance matrix.

3.4. Root Locus Technique

Root locus is a control system design technique that uses a graphical representation of a closed-loop system's poles and zeros to assess system stability and observe the effect of varying certain system parameters [41, pp. 388, 456-459]. Before delving into the specifics of the technique, consider the example traditional closed-loop, or feedback control, system depicted in Figure 15.

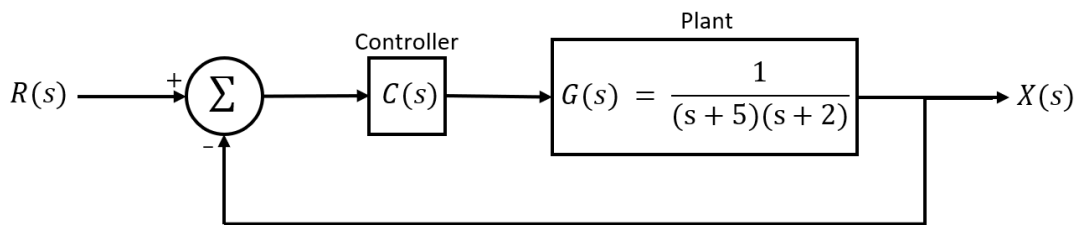


Figure 15: Traditional feedback control loop

Plotting the locations (locus) of the plant's poles and zeros in the s-plane as a function of the loop gain, the stability of the closed-loop system can be observed via the system's transient response. The system is stable when the poles and zeros are located in the left-half plane. The root locus and transient response of the plant (controller set to unity) from Figure 15 are shown in Figure 16.

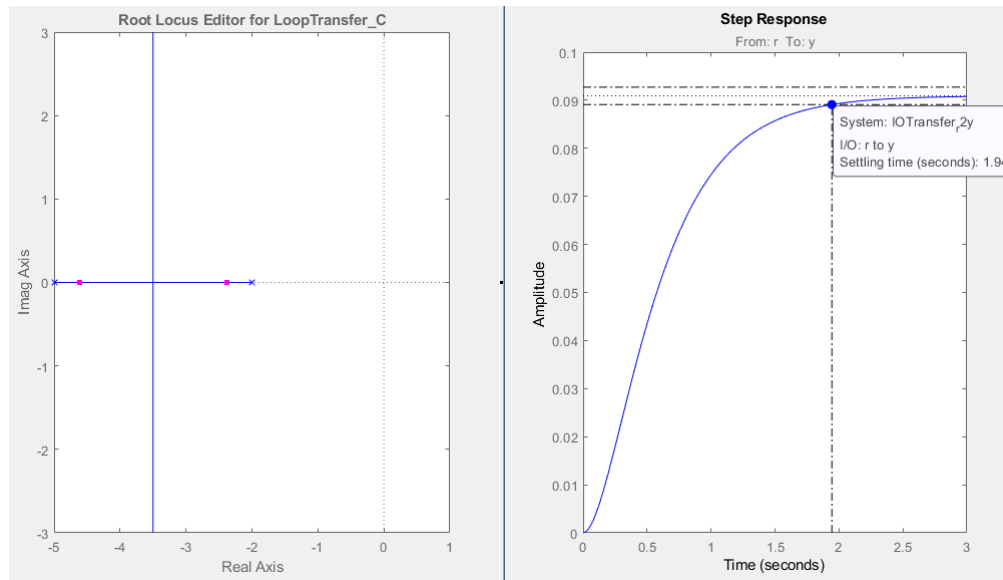


Figure 16: Root locus and transient response of the open-loop system

To improve the transient response and system stability, additional poles and zeros can be added to the controller. For example, to decrease the system's settling time to under 1.9 seconds a single pole and zero can be added, on top of increasing the gain. The controller now takes the form:

$$C(s) = \frac{1.25(s + 4)}{(s + 10)} \quad (12)$$

Implementing the controller, the root locus and transient response shown in Figure 17 is produced for the closed-loop system.

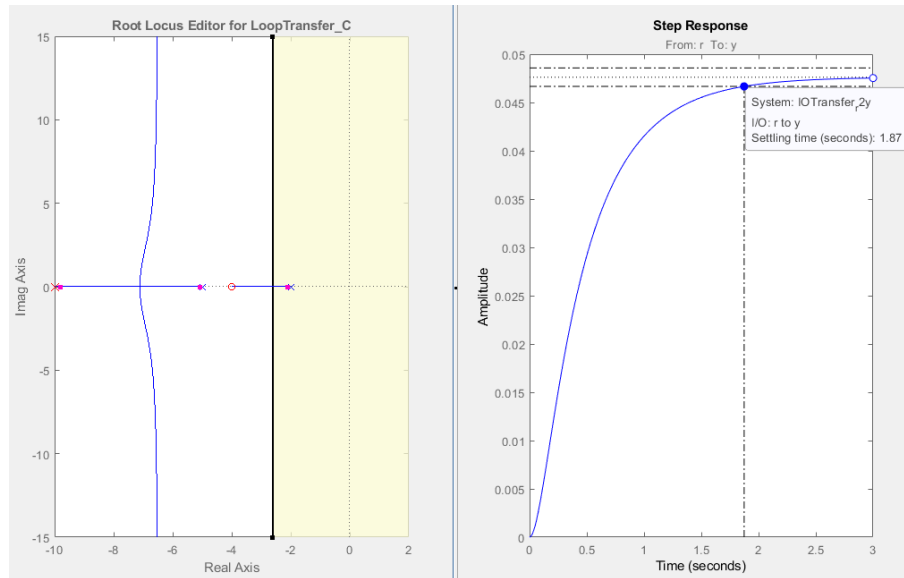


Figure 17: Root locus and transient response of the closed-loop system

By implementing the designed controller, the closed-loop system can achieve a settling time of less than 1.9 seconds. Further design requirements, such as rise time and percent overshoot, can be addressed by adjusting the controller's gain and/or by adding or removing poles and zeros to the controller.

4. Modeling

There are numerous variables to consider when attempting to model a UAV in flight. Even when just hovering in place, a UAV is experiencing and generating forces in a variety of directions. From motor torques to air resistance, building a mathematical model for a UAV is a complicated task. Entire research projects and papers are dedicated to the task, such as [42] and [43]. Therefore, the building of a full mathematical model for tuning the LCAS was out of the scope of this thesis. However, by limiting the variables and experimental parameters it was possible to derive a reasonable model of a UAV for tuning the LCAS.

Instead of considering all six directions, the derived model was only for a single direction, specifically the forward direction. By focusing only on one direction, the model became one-dimensional and thus the forces were simplified. The free-body diagram in Figure 18 provides a depiction of this concept.

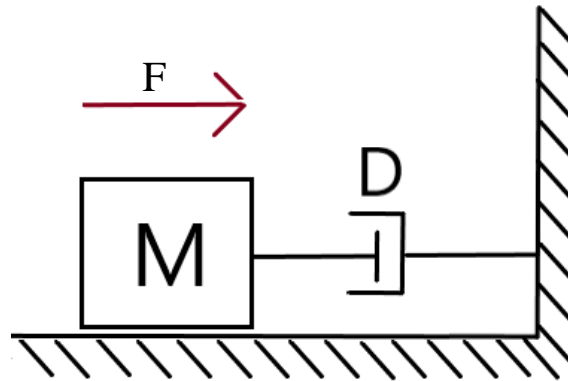


Figure 18: One-dimensional free-body diagram of a UAV flying towards a wall

Working in the frequency domain, the forces from Figure 18 can be depicted using the following equations adapted from [41, pp. 35, 64]:

$$s = \sigma + j\omega \quad (13)$$

$$F_M(s) = Ms^2X(s) \quad (14)$$

$$F_D(s) = DsX(s) \quad (15)$$

where s is the complex frequency parameter; F_M is the force exerted by the UAV; F_D is the damping force from air resistance; D is the damping coefficient; and X is displacement.

Applying Newton's second law, the sum of the forces is

$$F(s) - DsX(s) = Ms^2X(s) \quad (16)$$

Solving for $X(s)$,

$$X(s) = \frac{F(s)}{Ms^2 + Ds} \quad (17)$$

where $X(s)$ is the displacement of the UAV, in meters, and $F(s)$ is the normalized input.

Looking at Equation (17), the parameters M and D need to be derived to finish the model equation. D , the damping coefficient, is derived from the effect of air resistance of UAV while it is in flight. A UAV, like the Canary, is unlikely to reach high enough speeds, especially in indoor environments, for the air resistance to start causing a damping effect on the UAV. Therefore, D can be considered negligible. M , on the other hand, is a bit more complex. Typically, M is the mass but when it comes to a UAV the parameter is a function of the flight controller. Essentially, the flight controller handles numerous parameters of the UAV while in flight, such as motor speed and motor torque, which causes M to be dynamic. Instead of attempting to mathematically model the dynamics of M , it was decided that it would be simpler to use data-driven modeling and curve-fit the response of the Canary to a known input to derive an equation for M .

4.1. Methodology

4.1.1. Step Input

A step input was chosen to test the response of the Canary quadcopter. Over the course of two flights a series of six steps were tested, with each step a constant SBUS value applied in the

Canary's forward direction. The magnitude of each step was controlled by a dial on the RC transmitter (see Section 6.5.4.1). Table V shows the step magnitudes used for measuring the step responses of the Canary.

Table V: Canary step input magnitudes

Flight	Step number	Magnitude [SBUS value]
1	1	1044
1	2	1182
1	3	1407
2	4	1683
2	5	1811
2	6	1810

4.1.2. Testing Area

The area used for testing the step responses of the Canary was Leonard Field on the east side Montana Tech's campus. The field and the UAV's area of operation (red box) are shown below in Figure 19.



Figure 19: Canary testing area, Leonard Field [44]

4.1.3. Procedure

Before triggering each step, the Canary was set to hover at an arbitrary altitude. Once the Canary was hovering the step's magnitude was set and then the step was triggered. The response to the step was captured as position and acceleration values (see Section 6.5.4 for details on the hardware and methodology). Upon approach of the end of the testing area, the step input was ended, and the UAV was returned to its starting position. The same procedure was then repeated for each step magnitude.

4.1.4. Data Collection

The parameters measured when recording the step response of the Canary were the position and acceleration. Position of the UAV was captured as latitude, longitude, and altitude values by a GPS module approximately every 100 ms. Acceleration on the x, y, and z axes were measured by an accelerometer approximately every 20 ms and filtered using a 10-point moving average. Both parameters were stored in logs.

In addition to the position and acceleration values, the RX and TX SBUS frames were logged in order to provide time alignment of the position and acceleration data to the triggering of the steps.

4.2. Data Processing

After copying the GPS, accelerometer, and SBUS logs off the MITM, the data were imported into MATLAB for processing.

4.2.1. SBUS Data

Before working with the GPS and accelerometer, the relevant SBUS channel values had to be parsed from the SBUS log. Since the step response testing was done in the forward

direction of the Canary, the channel value that was modified by the MITM was *Ele*. Triggering of the step was handled by *vEN*. For more details on *Ele* and *vEN* refer to Table IV.

The first step was to separate the TX data from the RX data. This was done since the MITM used the RX channel values for *Ele* and *vEN* to determine the value of *Ele* to be transmitted to the flight controller. The TX data was parsed by using every even numbered row in the data log to obtain the relevant values for *Ele*, *vEN*, and sample time (*Tsbus*).

The second step was to set the zero point of the sample time vector and check for time logging errors. The zero-point had to be set since the sample time was recorded by the MITM in Unix time, or the number of seconds since midnight on January 1, 1970. By subtracting the first value of the sample time vector from every value in the vector, the vector started at zero, corresponding with the start time of the Canary's flight. With the zero-point set, the modified sample time was checked for errors. The error-checking was needed after it was observed that the MITM had arbitrarily added more than 1500 seconds to the sample times in the middle of one of the flights. When a significant change in error time was found during error checking, the difference between the last valid sample time and the first invalid sample time was subtracted from the first invalid sample time and the succeeding sample times.

The final step in processing the SBUS data was to resample. Resampling was done since the sampling rate of the SBUS data was not consistent, ranging from 19 to 23 ms thus making the data difficult to work with later. The inconsistent sampling rate was a result of the MITM checking if new RX SBUS frames were the same as previous frames. If a new frame was different from the previous frame then the MITM needed additional time to decode the frame. If the new frame was the same as the previous frame then the MITM skipped the decoding process and reused the SBUS data from the previous frame. Using MATLAB's built-in *resample()*

function, Ele and vEN were resampled with a sampling period of 0.02 seconds. A new sample time vector was generated by the function to match the resampled values of Ele and vEN .

4.2.2. GPS Data

The first step in processing the GPS data was to set the zero point and error checking. The process was the same as the one used in the processing of the SBUS data.

The second step was to convert latitude and longitude values to positional values. The conversion factors for latitude and longitude are dependent on the location of the coordinates in the world. This stems from the fact that as latitude increases the width of a second decreases, which in turn also affects longitude [45]. Taking that into consideration, the following equations were used to determine the latitude and longitude to meters conversion factors, courtesy of [45]:

$$lat_m = 111,132.92 - 559.82 \cos 2\varphi - 0.0023 \cos 6\varphi \quad (18)$$

$$long_m = 111,412.84 \cos \varphi - 93.5 \cos 3\varphi + 0.118 \cos 5\varphi \quad (19)$$

where lat_m is the conversion factor for latitudinal degrees to meters; $long_m$ is the conversion factor for longitudinal degrees to meters; and φ is the initial latitude coordinate. Both equations return the conversion factors in meters per degree. The conversion factors were calculated for each flight and the values are shown in Table VI.

Table VI: Conversion factors per flight for latitude & longitude to meters

Flight	lat_m (m/degree)	$long_m$ (m/degree)
1	111,473.52	-49,356.33
2	111,473.46	-49,362.63

Using the conversion factors, the GPS log's latitude and longitude values were converted to positional values with the initial values set to zero via the following equations:

$$y_{gps}(k) = lat_m(latitude(k) - latitude(0)) \quad (20)$$

$$x_{gps}(k) = long_m(longitude(k) - longitude(0)) \quad (21)$$

The final step was to resample the GPS data to align with the sampling rate of the SBUS data. The *resample()* function was used to resample x_{gps} and y_{gps} .

4.2.3. Accelerometer Data

Processing of the accelerometer data began with the multiplying the acceleration values by gravitational acceleration (9.81 m/s^2) in order to convert from g-force. Next the zero point was set in the sample time vector and the entire vector was error-checked using the same method as in the SBUS and GPS data processing. The final step in processing the acceleration values was to resample.

4.3. Estimating the Canary's Position

By itself, the GPS position could have produced a relatively accurate estimation of the Canary's position during the two valid step response testing fights. However, the original GPS data were limited to about 10 samples per second, while both the SBUS and accelerometer data were recorded at about 50 samples per second. Resampling of the GPS data was able to increase the number of samples, but minimally improved the accuracy of the position estimation.

Using the accelerometer data to estimate the position of the Canary would have been an exercise in futility. Accelerometers are prone to degrading efficiency over time due to compounding errors and are susceptible to hysteresis [46]. Some of the error was addressed by using a moving average filter when the MITM recorded values, but the filter could not account for all errors. Without removing errors, such as offsets, integrating to get a position estimate would have only resulted in an estimation of the total error.

In order to produce an accurate position estimate, while addressing the shortcomings of both sensors, data fusion was used. Data fusion is the process of combining relevant information

from multiple sources into a single more consistent and accurate estimate compared to using only one of the individual sources [47]. The data fusion algorithm chosen for combining the GPS and accelerometer data was the Kalman filter (Section 3.3).

The first step in using the Kalman filter was to develop a state-space model of the system. Given the known states are the positions and accelerations and the unknown states are the velocities, the following relationships were considered:

$$x_{k+1} = x_k + v_{x,k}\Delta t \quad (22)$$

$$y_{k+1} = y_k + v_{y,k}\Delta t \quad (23)$$

$$v_{x,k+1} = v_{x,k} + a_{x,k}\Delta t \quad (24)$$

$$v_{y,k+1} = v_{y,k} + a_{y,k}\Delta t \quad (25)$$

$$a_{x,k+1} = a_{x,k} + w_{x,k} \quad (26)$$

$$a_{y,k+1} = a_{y,k} + w_{y,k} \quad (27)$$

where x and y are the positions; v_x and v_y are the velocities; a_x and a_y are the accelerations; and w_x and w_y are random variables representing the system's process noise, such as disturbances.

The following sixth-order system model was derived by substituting the relationships, linearly, into Equations (5) and (6).

$$\underbrace{\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ v_{x,k+1} \\ v_{y,k+1} \\ a_{x,k+1} \\ a_{y,k+1} \end{bmatrix}}_{x_{k+1}} = \underbrace{\begin{bmatrix} 1 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 1 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_A \underbrace{\begin{bmatrix} x \\ y \\ v_x \\ v_y \\ a_x \\ a_y \end{bmatrix}}_{x_k} + w_k \quad (28)$$

$$\underbrace{\begin{bmatrix} x_k \\ y_k \\ a_{x,k} \\ a_{y,k} \end{bmatrix}}_{y_k} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_c \underbrace{\begin{bmatrix} x \\ y \\ v_x \\ v_y \\ a_x \\ a_y \end{bmatrix}}_{x_k} + v_k \quad (29)$$

With the state-space model built, the GPS and accelerometer data were passed into the Kalman filter algorithm. For the prediction stage the uncertainty, q , was optimized to be 1×10^{-5} after a few iterations of the Kalman filter. Therefore, the process noise covariance was defined as

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & q & 0 \\ 0 & 0 & 0 & 0 & 0 & q \end{bmatrix} \quad (30)$$

For the update stage the measurement noise covariance matrix, R , was derived by calculating the magnitude of the autocorrelation of the measurement noise for each of the known states (x , y , a_x , and a_y). Thus, the matrix was defined as

$$R = \begin{bmatrix} R_{xx} & 0 & 0 & 0 \\ 0 & R_{yy} & 0 & 0 \\ 0 & 0 & R_{a_x a_x} & 0 \\ 0 & 0 & 0 & R_{a_y a_y} \end{bmatrix} \quad (31)$$

4.4. Step Response Results

The GPS and Kalman filter position estimates for both flights are shown in Figures 20 and 21. Note that the x- and y-positions refer to the Canary's latitude and longitude, respectively, in meters.

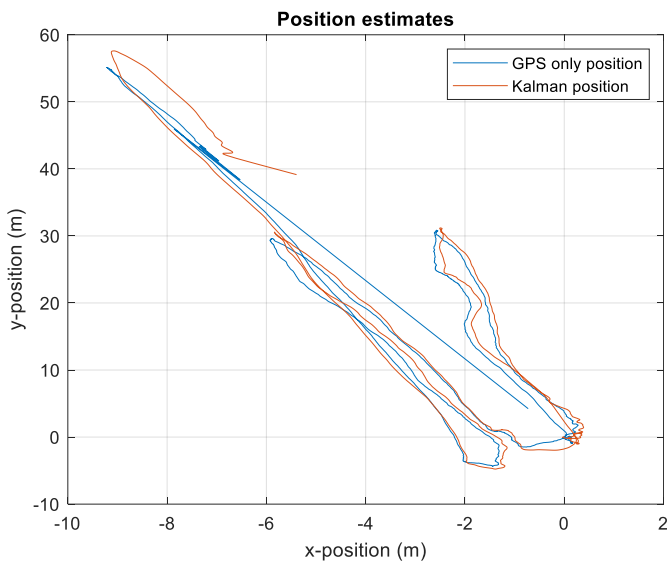


Figure 20: Flight 1 position estimate

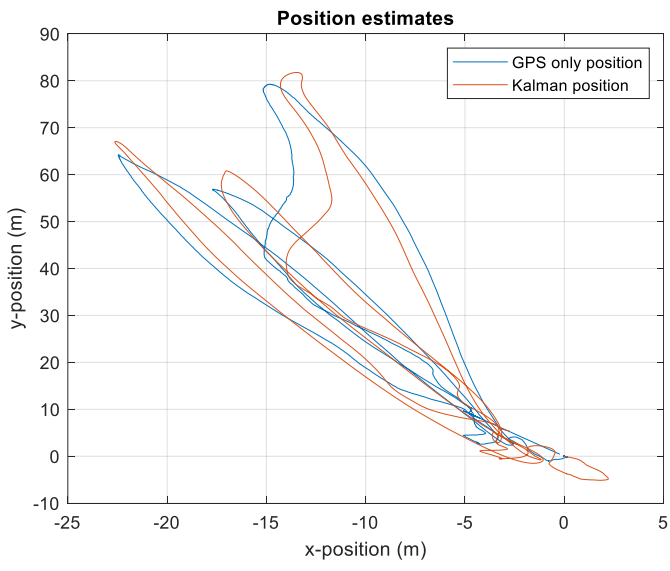


Figure 21: Flight 2 position estimate

From both figures it can be seen that the GPS position estimate was representative of the Canary's position during either flight, but exhibited sharper turns and less of the flowing motion that the Canary exhibited during the flights. The Kalman filter estimate, on the other hand,

showed smoother turns and overall appeared to better portray the flowing motion of the Canary during the flights.

To better show the individual step responses, the Kalman filter position estimate was separated into component x- and y- positions and plotted versus time. These plots are showing in Figures 22 and 23. Also, the SBUS channel *vEN* (step enable) was included to show the time duration of the steps.

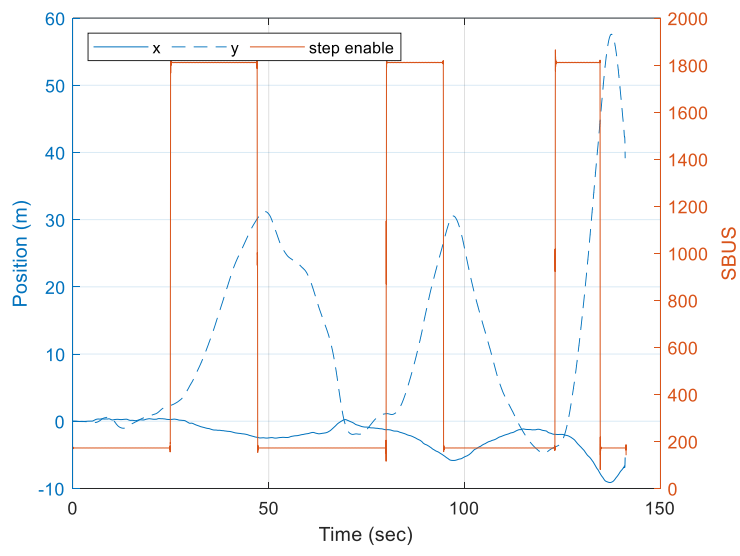


Figure 22: Flight 1 position versus time

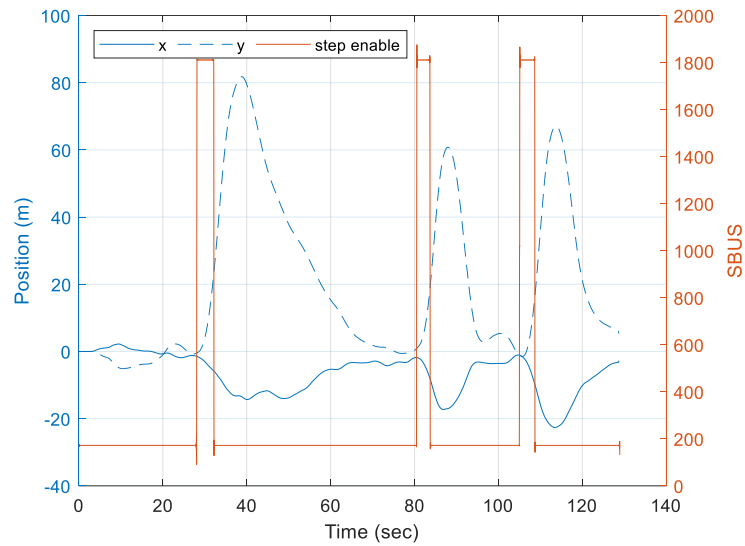


Figure 23: Flight 2 position versus time

As can be seen in the figures, both Canary flights contained three individual steps in the forward direction. The magnitudes of each step were set before the triggering of each step. To aid in the derivation of the Canary's model, the individual steps and the respective responses were parsed from the data. To simplify calculations to a single dimension, the magnitude between the two position components was calculated using the Pythagorean theorem. Finally, each step was set to start at time zero with the initial position at the origin. The parsed step responses are shown in Figure 24, for Flight 1, and Figure 25, for Flight 2. The magnitude (in SBUS values) of each step is noted above the respective plot.

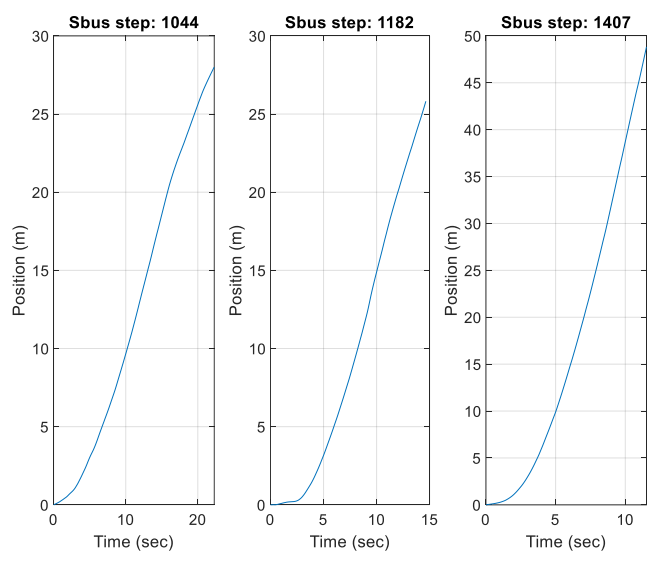


Figure 24: Flight 1 step responses

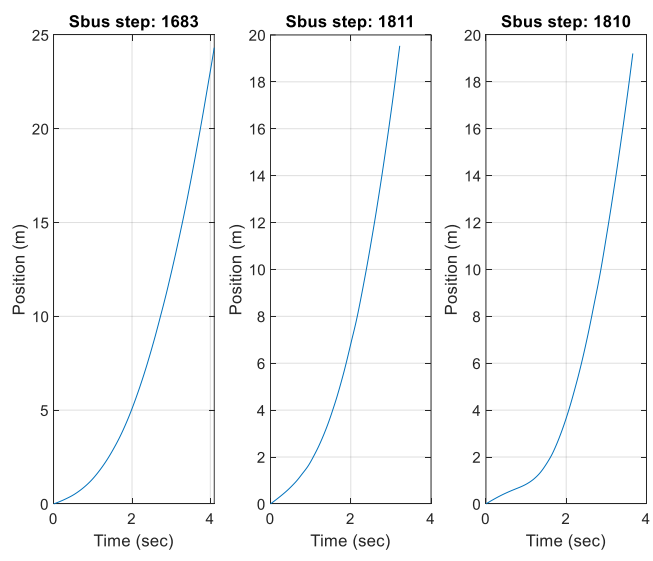


Figure 25: Flight 2 step responses

4.5. Deriving the Model Equation

4.5.1. Curve-fitting the Model Equation

In order to curve-fit the model, an estimated equation had to be used to determine what parameters need to be derived from the data. Rearranging Equation (17),

$$X(s)(Ms^2 + Ds) = F(s) \quad (32)$$

Taking the inverse Laplace transform to return to the time domain,

$$M\ddot{x}(t) + D\dot{x}(t) = f(t) \quad (33)$$

As discussed previously the effect of D is negligible. Also, $f(t)$ is constant at a given t . Ignoring D and substituting the constant A for $f(t)$,

$$M\ddot{x}(t) = A \quad (34)$$

Rearranging and taking the integral,

$$\int \ddot{x}(t) dt = \frac{1}{M} \int A dt \quad (35)$$

$$\dot{x}(t) = A \frac{1}{M} t + v_0 \quad (36)$$

where v_0 is the initial velocity. Integrating again to get position,

$$\int \dot{x}(t) dt = \int A \frac{1}{M} t + v_0 dt \quad (37)$$

$$x(t) = A \frac{1}{2M} t^2 + v_0 t + x_0 \quad (38)$$

where x_0 is the initial position.

However, the methodology used for collecting the data had the Canary at a stationary hover before triggering the steps, thus there was no initial velocity. Furthermore, when processing the data, the initial position of the Canary for each step was normalized to zero. So, zeroing the initial velocity and position,

$$x(t) = A \frac{1}{2M} t^2 \quad (39)$$

The coefficient, $\frac{1}{2M}$, can be simplified into a single quantifiable value, R , as a function of M with units of m/s^2 . Substituting in R for $\frac{1}{2M}$, the model equation is

$$x(t) = ARt^2 \quad (40)$$

Solving for R ,

$$R = \frac{x(t)}{At^2} \quad (41)$$

An estimate for R was made by substituting the position estimates from the step response testing in for $x(t)$ and the step magnitudes in for A . The model was then simulated and compared to the Canary's responses. Results from the simulation are shown in Figures 26 and 27. The estimated value of R for each step is given in Table VII.

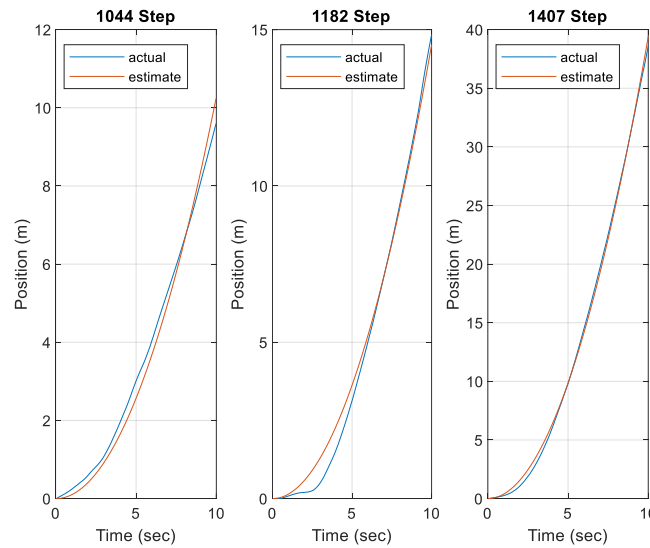


Figure 26: Results from the R estimation using Flight 1 parameters

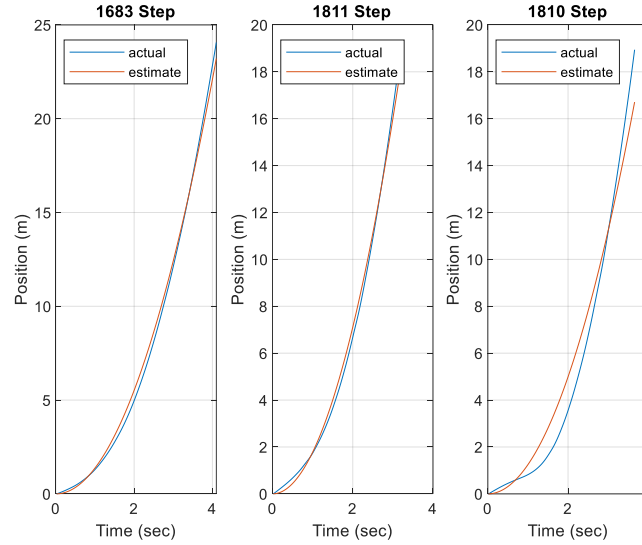


Figure 27: Results from the R estimation using Flight 2 parameters

Table VII: Estimates of R

Flight	Step Magnitude	R
1	1044	1.4152
1	1182	0.5789
1	1407	0.7622
2	1683	1.6577
2	1811	1.8080
2	1810	1.2711

Taking the average of the estimates, the value of R was found to be 1.2489. To test if the estimate of R was reasonable, it was substituted into Equation (40), simulated again with the same step magnitudes, and compared to the Canary's step responses. The simulation results are shown in Figures 28 and 29.

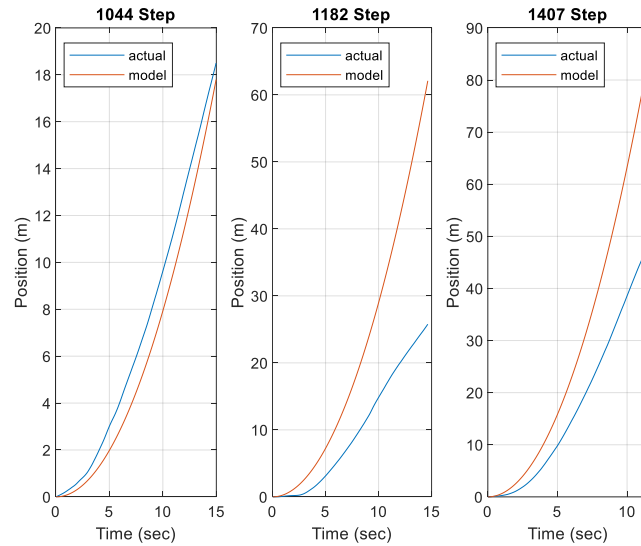


Figure 28: Model simulation with $R = 1.2849$ using Flight 1 parameters

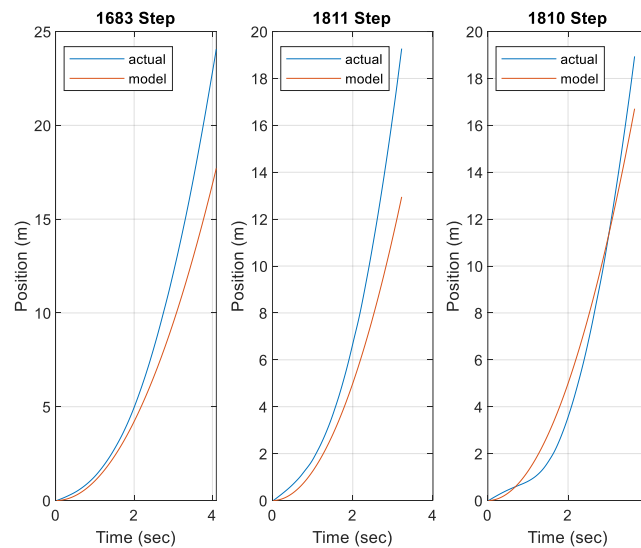


Figure 29: Model simulation with $R = 1.2849$ using Flight 2 parameters

Looking at the figures, the 1.2849 estimate for R did not provide an adequate approximation of the Canary, with the model's step responses not quite replicating the Canary's step responses. The reasons behind this conclusion can be seen in the R estimates for the Canary's 1182 and 1407 step responses. From Table VII, the R estimates for the 1182 and 1407

step responses were 0.5789 and 0.7622, respectively. Both of the estimates were significantly lower than the next largest R estimate, 1.2711 for the 1810 step. Looking at the model's response versus the Canary's response for either step it can be seen that the Canary was slower than the prediction made by the model. Furthermore, looking at the 1810 step response the Canary did not respond as quickly as it did to the 1811 step, as if the Canary experienced a disturbance when the step was trigger which caused the initial response to be slowed.

Attempting to improve the model approximation, the values of R for the 1182, 1407, and 1810 steps were left out of the average R estimate calculation. Thus, resulting in a new average R value of 1.6270. Running the validation simulation again, the results given in Figures 30 and 31 were produced. The 1182, 1407, and 1810 step simulations are included to show how the model responded to those step values.

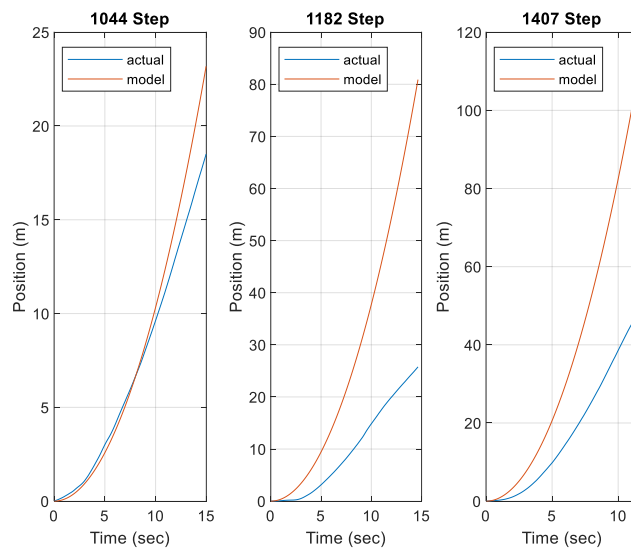


Figure 30: Model simulation with $R = 1.6270$ using Flight 1 parameters

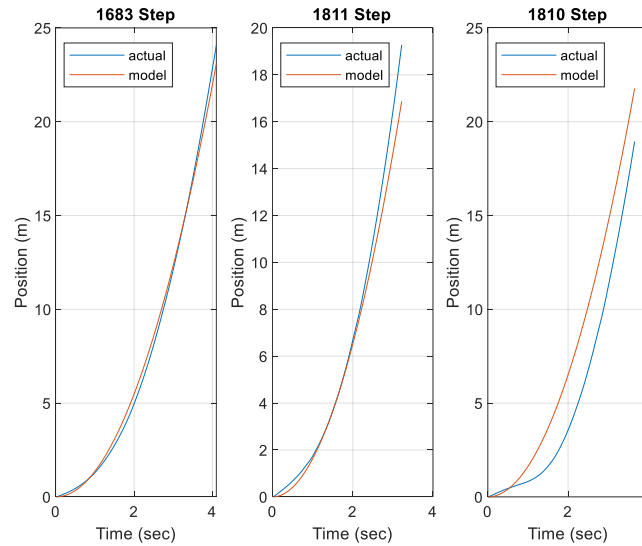


Figure 31: Model simulation with $R = 1.6270$ using Flight 2 parameters

By leaving out the R estimates from the 1182, 1407, and 1810 steps a better approximation of the Canary was produced. Under the 1044, 1683, and 1811 steps the model appeared to better match up with the Canary's responses.

The new value of R proved to be more reasonable and provided a better approximation of the Canary. Substituting the value into Equation (40),

$$x(t) = A(1.6270)t^2 \quad (42)$$

4.5.2. Discretizing the Model

So far most of the model derivation had been in the continuous time domain. However, in order to develop the LCAS's controller the model must be in discrete time, since the controller will only be able to operate on a sample-by-sample basis.

Moving Equation (42) to the Laplace domain via the Laplace transform,

$$X(s) = A \frac{2(1.6270)}{s^2} \quad (43)$$

Substituting the Laplace form of the input $F(s)$ in for A and solving for the output, $X(s)$, over the input, the system transfer function is

$$\frac{X(s)}{F(s)} = \frac{2(1.6270)}{s^2} \quad (44)$$

Using MATLAB's continuous-to-discrete time function, $c2d()$, Equation (43) was discretized with a 0.02-second sampling period, resulting in:

$$\frac{X(z)}{F(z)} = \frac{0.0006508 + 0.0006508z^{-1}}{1 - 2z^{-1} + z^{-2}} \quad (45)$$

Using the inverse Z-transform, the model's difference equation is

$$x[k] = 0.0006508f[k] + 0.0005608f[k-1] + 2x[k-1] - x[k-2] \quad (46)$$

5. Controller Design

5.1. Overview

In the process of developing the LCAS feedback controller, two separate controller designs, utilizing two different control system design techniques, were built. The Phase I controller was designed first and tuned using time domain techniques and trial-and-error until a reasonable response was attained for four simulated scenarios. While able to drive the model to a desired position, the controller was characterized by unfavorable, oscillatory behavior and slow response times (shown later on).

Seeking to improve on the Phase I design, a second design phase was conducted to produce a new controller. The Phase II controller design was made using the root locus technique to observe the effect of the controller's poles and zeros on the closed-loop system stability and transient response. Overall, the Phase II controller was able to improve upon Phase I substantially, exhibiting reduced oscillatory behavior and a faster response time across all four scenarios.

5.2. Design Methodology

5.2.1. Phase I

The Phase I controller was based on the idea of modifying the input, $f[k]$, when the position is detected to be within an activation window. For example, when the position is greater than the activation threshold, x_{act} , the controller overrides the input with a new input that drives the model to x_{des} . The controller's block diagram is shown in Figure 32.

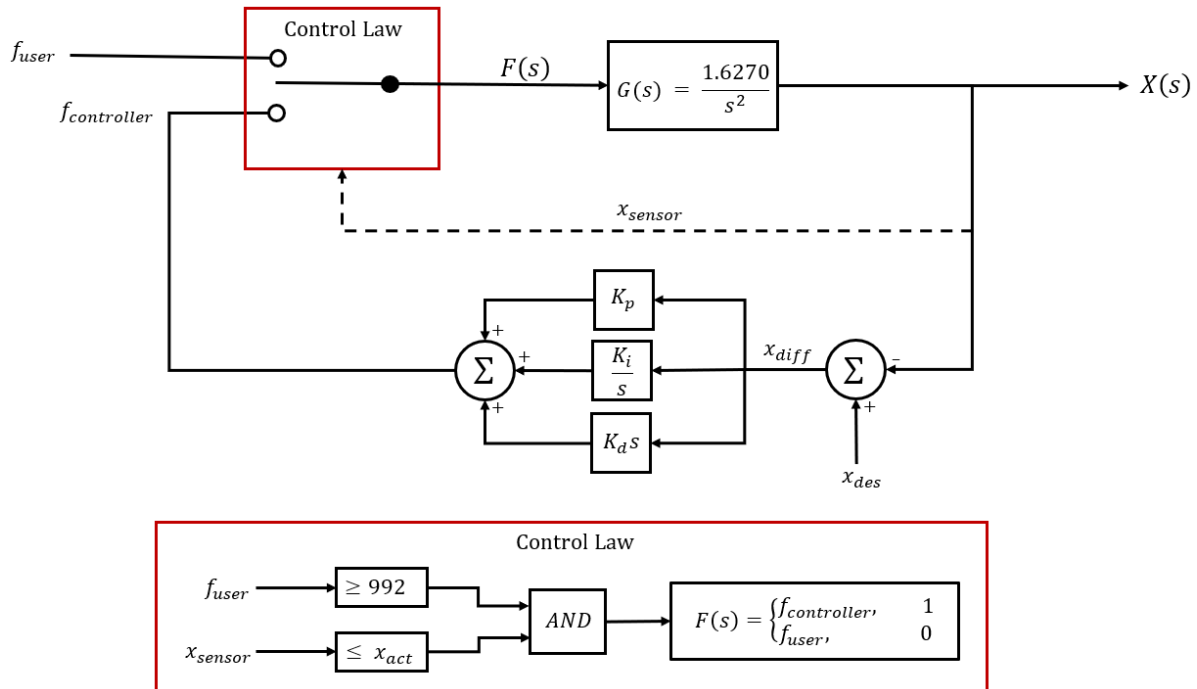


Figure 32: Phase I controller block diagram

Most of the design of the Phase I controller was done in the discrete time domain via simulation in MATLAB. Before simulating, the controller algorithm from the block diagram in Figure 32 was translated into the following difference equation:

$$f_{cont}[k] = K_p(x_{diff}[k]) + K_i(x_{diff}[k] + x_{diff}[k - 1]) + K_d(x_{diff}[k] - x_{diff}[k - 1]) \quad (47)$$

where x_{diff} is the difference between the desired position and the system's actual position. The gains, K_p , K_i , and K_d , were found by iterating the controller until it produced a reasonable response. The final values of the gains were 1.2 for K_p , 0.1 for K_i , and 5 for K_d .

5.2.2. Phase II

Unlike the design method for the Phase I controller, the Phase II controller's design took a more traditional approach in the root locus technique (detailed in Section 3.4). Most of the design was done via MATLAB's Control System Designer App.

Importing the frequency-domain model transfer function, Equation (44), into *rltool()* the following root locus for the open-loop system was produced:

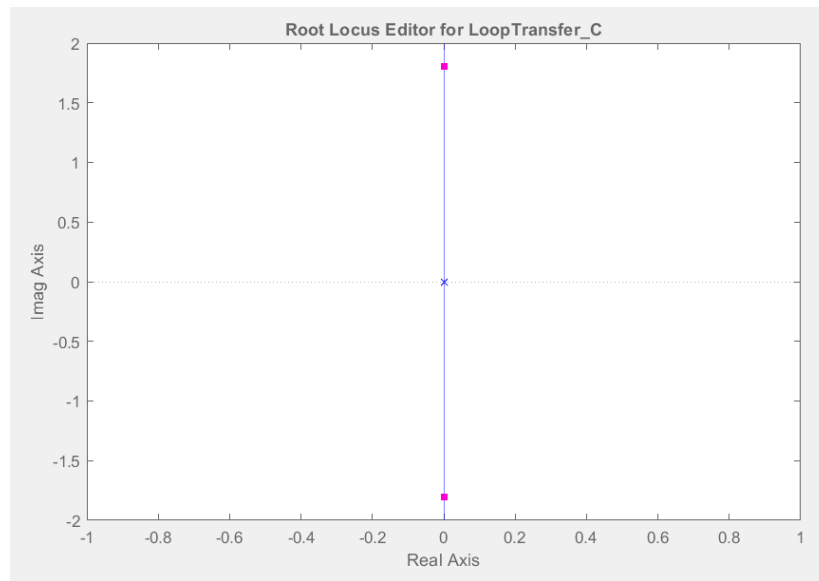


Figure 33: Open-loop system root locus

Based on the response of the Phase I controller, it was determined that the Phase II controller would need to drive the model with less overshoot and a faster settling time. Therefore, to determine the pole, zero, and gain of the controller an overshoot of 40 percent and a settling time of 10 seconds were arbitrarily chosen as the design requirements when the controller is subjected to a 1.0-magnitude step (1811-magnitude in SBUS).

By adding the controller to the root locus, the values of the pole and zero were adjusted graphically in the Control Designer App until the step response of the closed-loop system met the design requirements. The resulting closed-loop root locus and step response are shown in Figures 34 and 35, respectively.

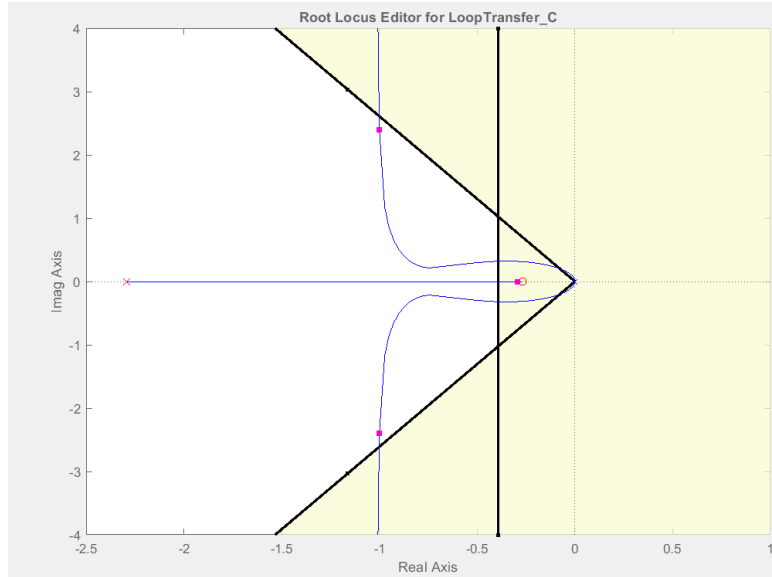


Figure 34: Phase II closed-loop system root locus

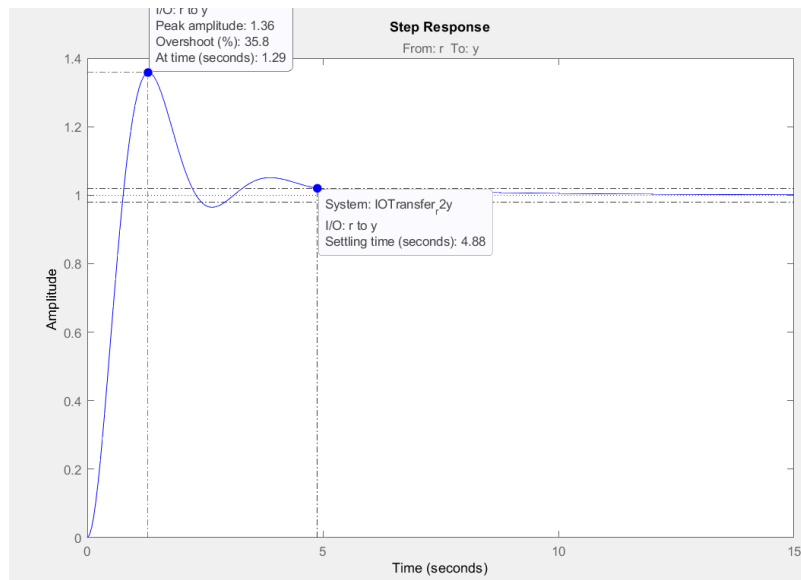


Figure 35: Phase II closed-loop system step response

With the design requirements met, the frequency domain transfer function for the Phase II controller was

$$C(s) = \frac{F_c(s)}{E(s)} = \frac{2.2435 (s + 0.2685)}{(s + 2.292)} \quad (48)$$

where F_c is the output of the controller and E is the error between the actual position and desired position. The equation was discretized using MATLAB's $c2d()$ function at a sampling period of 0.02 seconds, resulting in

$$C(z) = \frac{F_c(z)}{E(z)} = \frac{2.2435(z - 0.9948)}{(z - 0.9552)} \quad (49)$$

Using the inverse Z-transform to find the controller's difference equation,

$$f_c[k] = 0.9552f_c[k - 1] + 2.2435e[k] - 2.2318e[k - 1] \quad (50)$$

5.3. Simulation Results

Both phases of the LCAS controller design were subjected to four simulation scenarios that were intended to simulate the behavior of the Canary under control of a pilot. All simulations were conducted under ideal conditions, meaning no measurement noise was added.

For visualization purposes, consider there to be a wall at 1 m and each controller is attempting to drive the model to maintain a desired position, x_{des} , of 0.5 m. The controller activates when the model's position is within 0.25 m (x_{act}) of the desired position.

5.3.1. Scenario 1

The first scenario set the input as a constant 1200 SBUS value as if the pilot were continuing to fly towards the wall without attempting to avoid a collision. The results from Scenario 1 are shown in Figure 36.

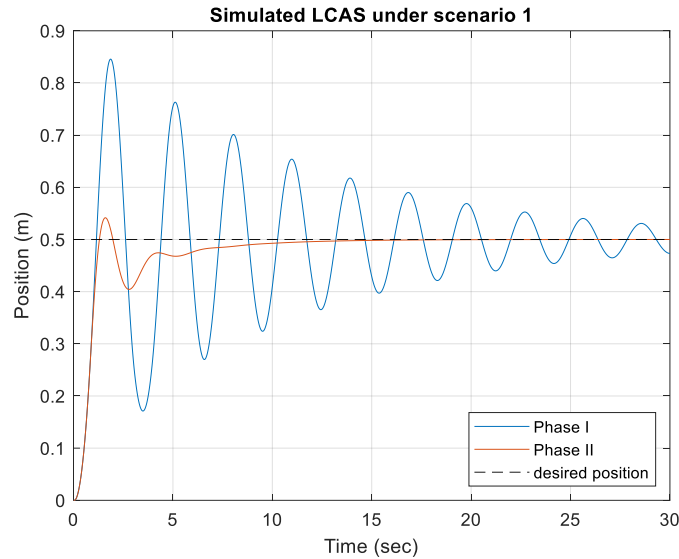


Figure 36: Scenario 1 simulation results

Under the first scenario, both controllers were able to drive the system away from the wall and towards the desired position. However, the Phase I controller exhibited oscillatory behavior causing the model to swing about the desired position and was unable to reach a steady state in the 30-second window. The Phase II controller, on the other hand, did not cause the system to oscillate and was able to achieve a steady state in just over five seconds. Also, the Phase II controller limited the system's overshoot to 8.3% compared to Phase I's 69%.

5.3.2. Scenario 2

The second scenario repeated the first; however, the pilot actively attempted to avoid a collision. Every time the system reached a position of 0.45 m the pilot momentarily dropped the input SBUS value to 785, attempting to slow the system's approach to the desired position. Furthermore, after the first reaction the pilot reduced the regular input to 1050. Figure 37 shows the results for Scenario 2.

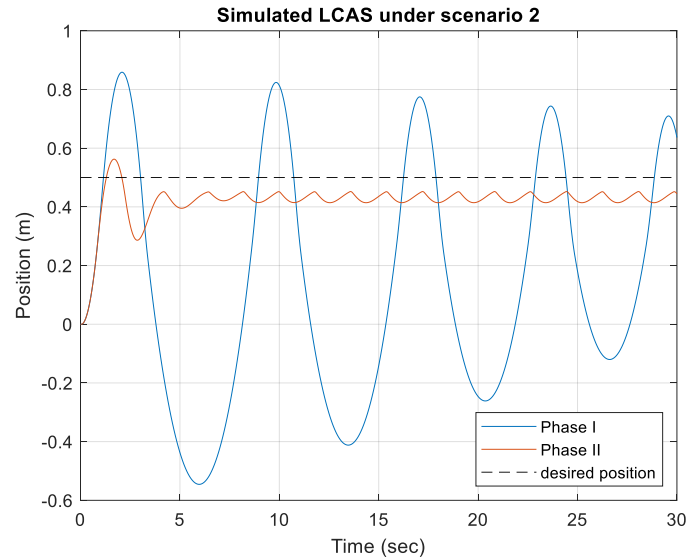


Figure 37: Scenario 2 simulation results

For Scenario 2, the Phase I controller appeared to amplify the effect of the pilot's intervention, driving the system below the zero position. This would be unacceptable behavior in real applications. Thus, the design advantage of the Phase II controller is quite evident. Instead of amplifying the pilot's actions, the controller complemented the pilot, allowing the system to settle about the 0.45-m position. The controller was less intrusive in its influence on the pilot's inputs, showcasing a significant improvement over the Phase I controller.

5.3.3. Scenario 3

Scenario 3 used the SBUS value 1000. By setting the input to a value close to neutral, the scenario simulated a slow drift towards the wall. And as akin to the first scenario, the pilot did not react to the approach of the desired position. The results are shown in Figure 38.

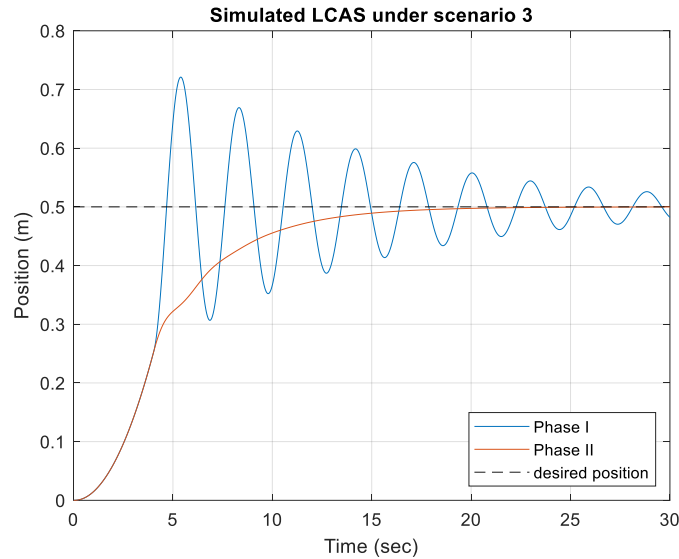


Figure 38: Scenario 3 simulation results

Another unfavorable behavior was exhibited by the Phase I controller under the third scenario. Once the system's position crossed into the activation window at 0.25 meters the Phase I controller activated and instead of attempting to slow the model the controller's output caused the system to increase in speed. This then began the same oscillatory behavior exhibited by the controller under Scenario 1.

Again, the design improvements of the Phase II controller over Phase I are clear. The Phase II controller not only drove the system to a steady state at the desired position, but also did not overshoot the desired position. Also, the system did not increase in speed at any point but rather slowed as it approached the desired position.

5.3.4. Scenario 4

Scenario 4 combined the slow drift of Scenario 3 with the pilot's intervention actions from Scenario 2. Every time the system reached a position of 0.45 m the pilot lowered the input to 825; however, the pilot would let the system return to the original input after the initial reaction instead of a lower value. The results for the scenario are shown in Figure 39.

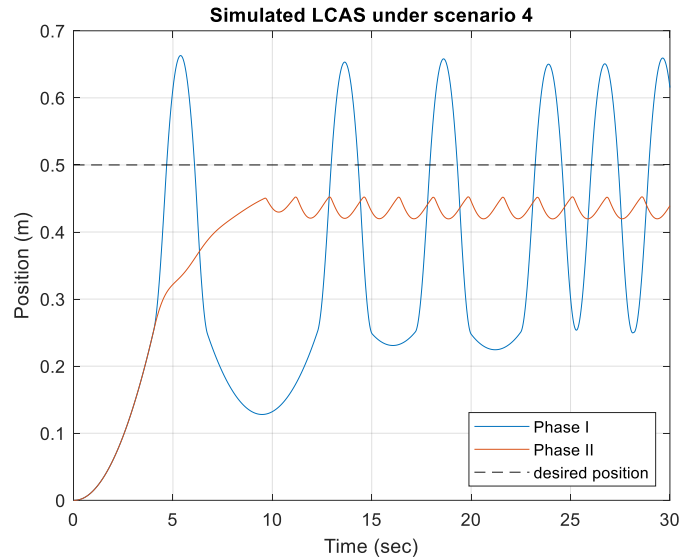


Figure 39: Scenario 4 simulation results

For the final scenario, the Phase I controller displayed peculiar behavior that was deemed entirely unacceptable. The increasing of the system's speed was exhibited once again as were the oscillations. It appears as if the controller has more control of the system than the pilot.

The Phase II controller's response is a stark difference. The controller complemented the pilot's actions by driving the system to the pilot's reaction position, and, if allowed, the controller would have approached the desired position in the same manner as the third scenario.

5.4. Noise Resiliency

The final testing stage of the LCAS feedback controller was to observe how each of the Phases handled error and/or noise in the position measurements. Each controller was subjected to two different cases. Both cases added normally distributed noise to the position values seen by the controller. The cases were differentiated by the scaling factor applied to the magnitude of the noise. The parameters of Scenario 1 were used to provide the simulation environment.

5.4.1. Case 1

The first case scaled the added noise by a value of 0.0036821. This value was found by calculating the standard deviation of a tinyLiDAR's distance measurements of an object at 150 mm over the course of 15 minutes, with a sampling period of approximately 15 ms. The Case 1 results for each controller are shown in Figures 40 and 41.

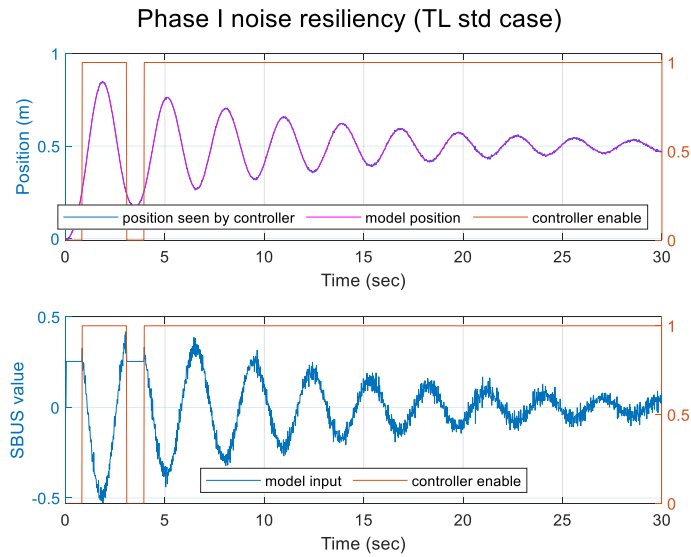


Figure 40: Case 1 simulation results for Phase I controller

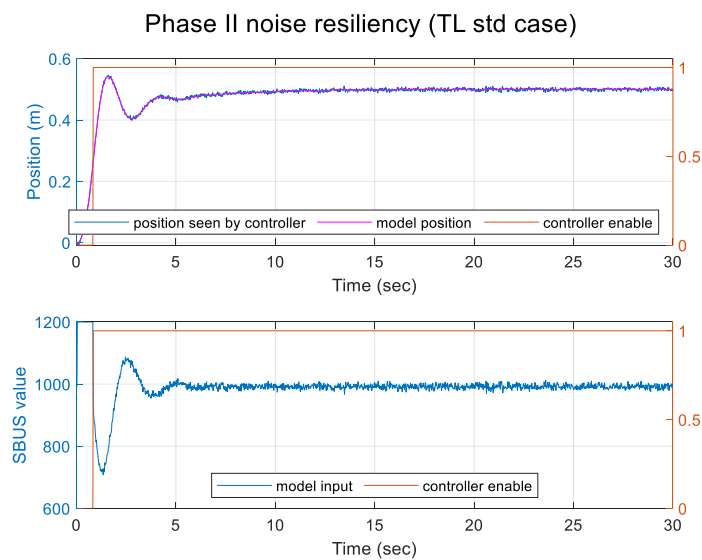


Figure 41: Case 1 simulation results for Phase II controller

Under the standard deviation of the tinyLiDAR's error, both controllers were able to drive the system to the desired position. In comparison to the position results for Scenario 1 in Figure 36, both controllers maintained the same response to the model despite the noise.

An important observation is the effect of the noise on the input to the model. The Phase I controller appeared to not suppress the measurement error, causing the model input to be noisy. The Phase II controller had a better response to the noise, but the fluctuation of the input about the SBUS neutral was a concern.

5.4.2. Case 2

The second case scaled the added noise by a factor of 0.1. This value was chosen to simulate an extreme error in position measurement. The response of each controller to Case 2 is shown in Figures 42 and 43.

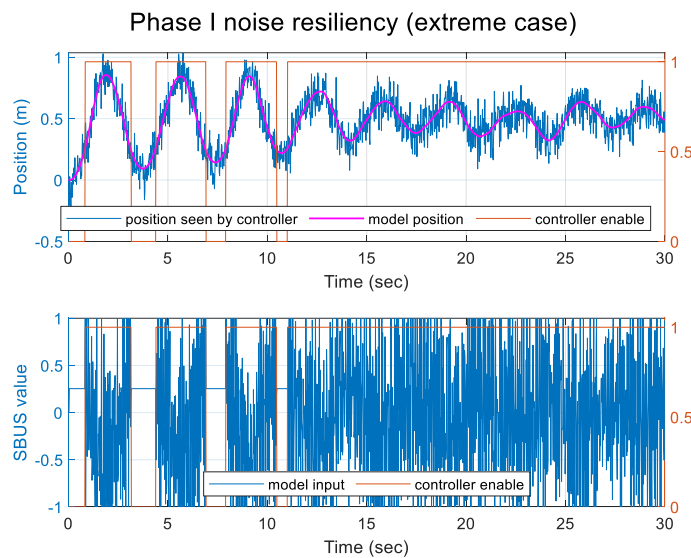


Figure 42: Case 2 simulation results for Phase I controller

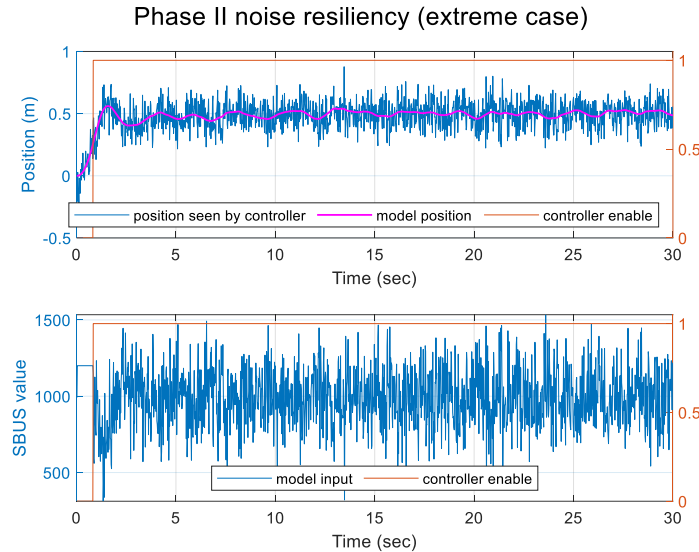


Figure 43: Case 2 simulation results for Phase II controller

Again, the controllers were able to drive the system to the desired position, but the fluctuations from Case 1 were exaggerated by the increase in noise magnitude. The Phase I controller oscillated the model input between the minimum and maximum SBUS values. The Phase II controller did not saturate the model input, but did cause the model input to fluctuate between 500 and 1500.

5.5. Choosing a Controller

Based on the results from the four scenarios, the decision between which controller to use was simple: the Phase II controller. The Phase II controller showed significant advantages over the Phase I controller. The Phase II controller did not exhibit oscillatory behavior about the desired position, and instead was able to drive the system to a steady state at the desired position. Overshooting of the desired position was dramatically reduced by the Phase II controller. Furthermore, the Phase II controller appeared to better complement the pilot. Under Scenarios 2 and 4, the Phase I controller amplified the pilot's reactions when the pilot intervened. The Phase

II controller, in comparison, did not amplify the reactions by incorporating previous model inputs to limit the amount of change that could be made to the new model input.

The ability to account for previous inputs and outputs gave the Phase II controller an advantage over the Phase I controller in the noise resiliency testing. Both controllers were able to handle Case 1, but Case 2 exposed the Phase I controller's inability to regulate its output, swinging it between the SBUS extremes. The Phase II controller handled Case 2 better by not letting its output swing so sporadically.

It is quite apparent, though, that the Phase II controller did not reduce the effect of noise on the system, as the noise was passed from the position seen by the controller to the model input. However, the noise in the controller's output had minimal effect on the model's actual position because of the relatively slow response of the model to changes in its input.

Since it is possible that a noisy input could cause the Canary's flight controller to go into failsafe, a 10-point simple moving average (SMA) filter was applied to the input of the Phase II controller. The results are shown in Figures 44 and 45.

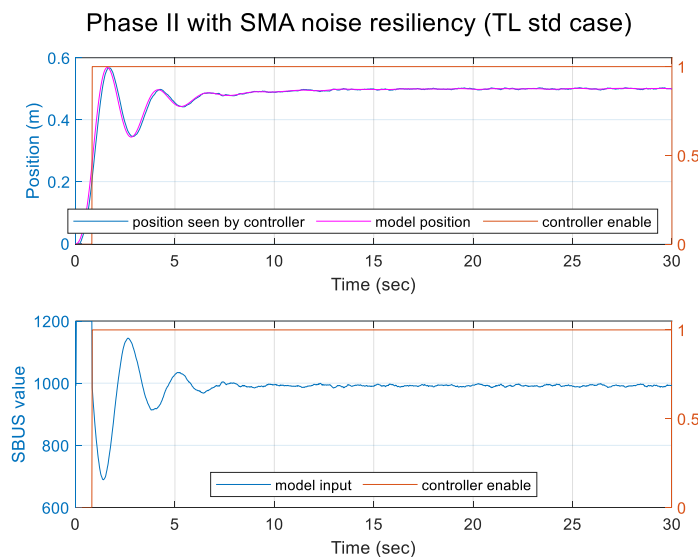


Figure 44: Case 1 simulation results for Phase II controller with a SMA

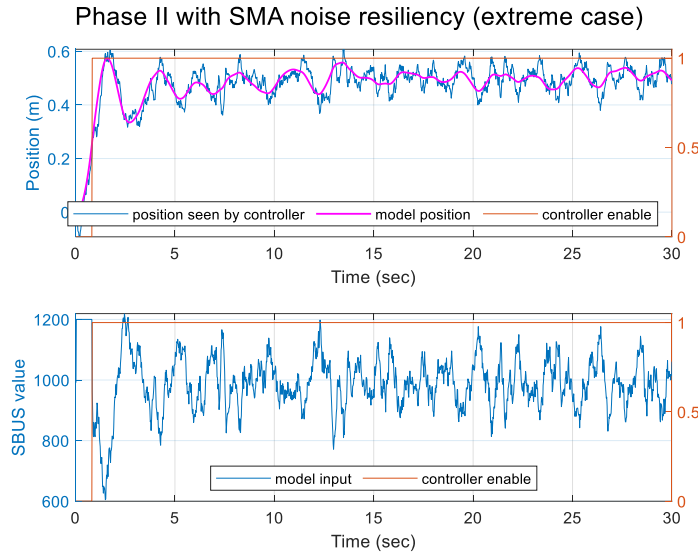


Figure 45: Case 2 simulation results for Phase II controller with a SMA

The addition of the SMA filter improved the noise resiliency of the Phase II controller noticeably. The noise of Case 1 had no effect on the system, with the response matching the ideal response of Scenario 1 (Figure 36). The improvement was more notable in Case 2. The sporadic fluctuations of the controller output never exceeded the original input of 1200 or fell too far below the SBUS neutral when at steady state.

As a note, the implementation of the Phase II equation in the LCAS prototype had an extra pole and zero at the origin. However, the additional pole and zero cancel each other out, resulting in the implemented equation matching equations in the Phase II design (Section 5.2.2). This simplification was discovered when documenting this work, which was after the LCAS prototype testing, thus why the extra pole and zero do not show up in the Phase II controller design but do show up in the LCAS prototype. The implemented equation is as follows:

$$C(s) = \frac{F_c(s)}{E(s)} = \frac{2.2435 s (s + 0.2685)}{s (s + 2.292)} \quad (51)$$

Discretizing with a sampling period of 0.02 seconds and then taking the inverse Z-transform the difference equation used for the LCAS feedback control algorithm was

$$\begin{aligned} f_c[k] = & (-0.9552f_c[k-2] + 1.9552f_c[k-1]) + (2.2317e[k-2] \\ & - 4.4753e[k-1] + 2.2435e[k]) \end{aligned} \quad (52)$$

6. Hardware

An in-depth review of the hardware used in the LCAS is provided in this section. Section 6.1, **Overview**, details the basic functions and operations of the LCAS hardware. Section 6.2, **Testing Platforms**, looks at the servo-based rover and Canary quadcopter used for the development and testing of the LCAS. Section 6.3, **RC Receiver & Transmitter**, covers the RC hardware used to control the Canary. Section 6.4, **Sensor Board**, discusses, in-depth, the design and functions of the LCAS Sensor Boards. The main controller of the LCAS is given a detailed look in Section 6.5, **MITM**. Section 6.6, **Final Prototype Design/Layout**, takes a look at the layout of the LCAS prototype when installed on the Canary.

6.1. Overview

The intended implementation of the LCAS is to be installed on a quadcopter with minimal modification to the platform and maintaining modularity of the LCAS. There are two main parts of the system: the Monkey-in-the-Middle (MITM) and the Sensor Boards. The MITM is placed in-between the RC receiver and the flight controller, where it can intercept SBUS signals and modify the signals. The Sensor Boards are branched off of the MITM and measure distances between obstacles and the quadcopter. Figure 46 is a general block diagram of the LCAS, while a more detailed block diagram of the MITM is shown in Figure 47.

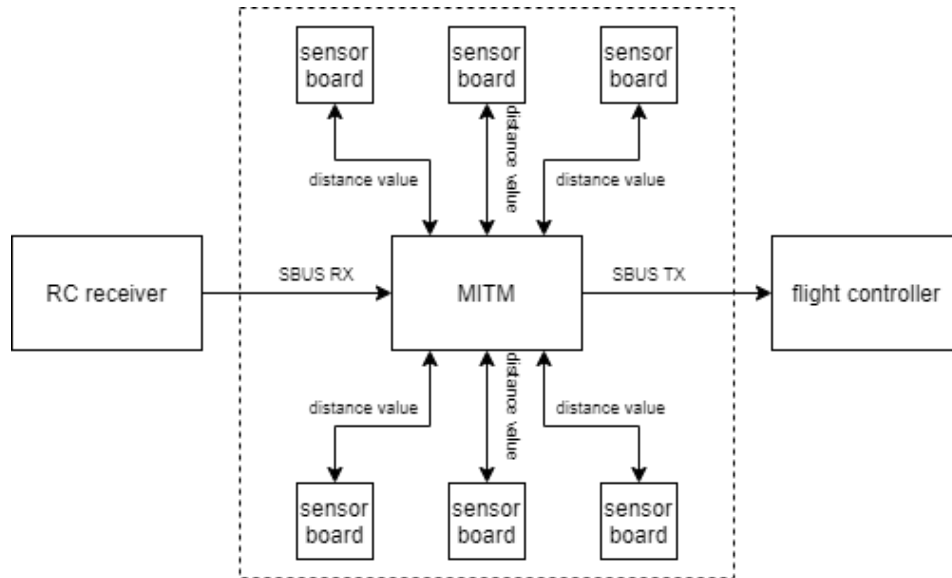


Figure 46: LCAS block diagram

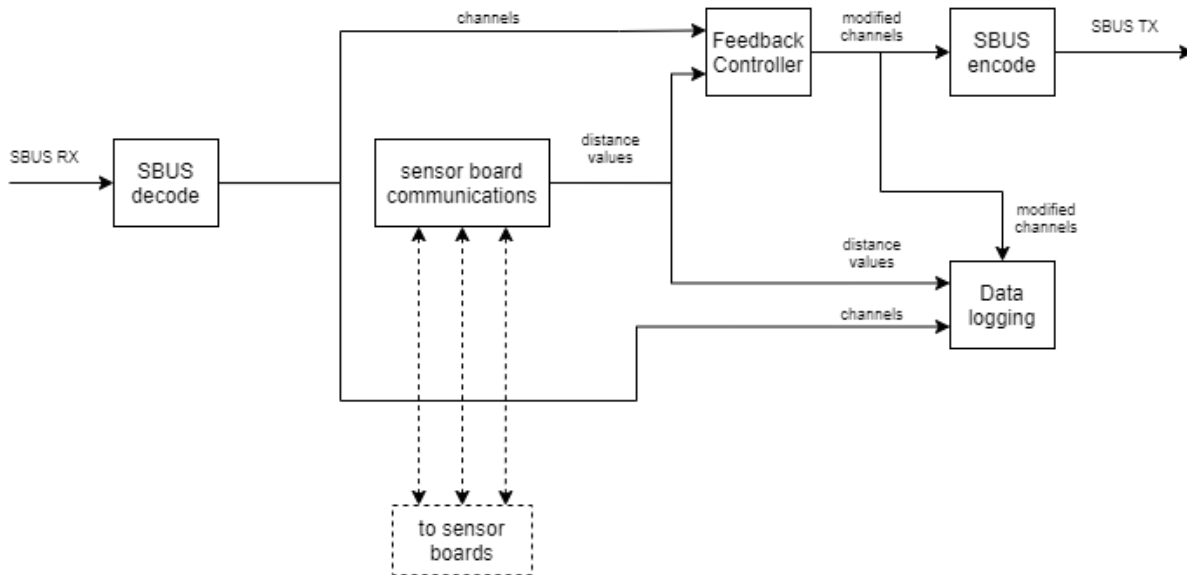


Figure 47: MITM block diagram

In the block diagram of Figure 46 SBUS signals are output by the RC receiver and immediately intercepted by the MITM. Figure 47 shows the MITM then decodes a single SBUS frame into individual channels. From there the MITM requests distance values from the Sensor Boards and uses the values in a feedback control system to determine if it needs to modify the

channels to avoid obstacles. After channel modification, the MITM encodes the channels into a new SBUS frame that is then transmitted to the flight controller. With the SBUS frame transmitted, the MITM logs the RX and TX SBUS frames and the distance values reported by the Sensor Boards.

6.2. Testing Platforms

The LCAS system used two different platforms for the testing of the individual components. The first platform was a servo-based rover platform. The second platform was a 260-mm quadcopter known as the Canary.

6.2.1. Servo-based Rover

The servo-based rover was a simple metal platform propelled by two servos at the front with a drag wheel at the rear. The rover was mostly used in the early development stages of the LCAS, serving as the testing platform for the first version of the MITM (Section 6.5.2). Since the rover did not have a dedicated flight controller, a MSP430G2553 microcontroller was used to decode SBUS signals and convert the values to PWM (pulse width modulation) signals. Figure 48 shows the rover with the SBUS-to-PWM converter.

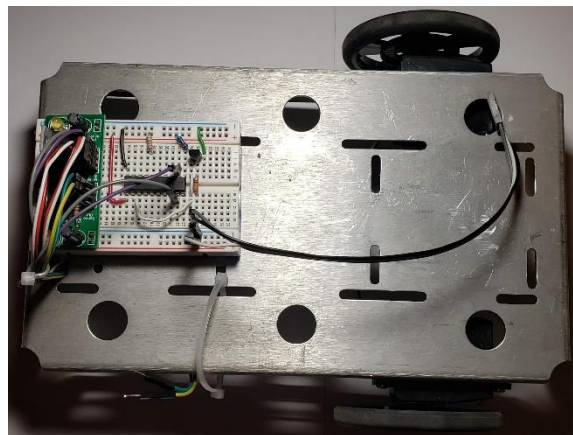


Figure 48: Servo-based rover testing platform

6.2.1.1. SBUS-to-PWM Converter

In order to provide the proper PWM signals to the servos, a microcontroller was needed to receive and decode SBUS signals. Since it was readily available, a Texas Instruments MSP430G2553 microcontroller was used. SBUS signals were passed through inverters (Figure 14) before being received by the microcontroller. Once the signals were received, the microcontroller converted the SBUS values for *Ail* and *Ele* (see Table IV) to PWM values using the following procedure and equations:

1. Change the range of the channels to be within [-800, 800]

$$\begin{aligned}x &= Ele - 800 \\y &= Ail - 800\end{aligned}\tag{53}$$

2. Find the angle, θ , between x and y

$$\begin{aligned}a &= 570 \tan^{-1}\left(\frac{x}{y}\right) + 225 \\ \theta &= \begin{cases} a - 3600, & a < 0 \\ a, & otherwise \end{cases}\end{aligned}\tag{54}$$

3. Find the resultant magnitude, R , between x and y

$$\begin{aligned}r &= \sqrt{x^2 + y^2} \\ R &= \begin{cases} 25, & r < 25 \\ r, & 25 \leq r \leq 800 \\ 800, & r > 800 \end{cases}\end{aligned}\tag{55}$$

4. Find the conversion gains

$$k_{left} = \begin{cases} 1, & \theta < 1350 \\ 0, & 1350 \leq \theta < 1800 \\ -1, & \theta \geq 1800 \end{cases}\tag{56}$$

$$k_{right} = \begin{cases} -1, & \theta < 450 \\ 0, & 450 \leq \theta < 900 \\ 1, & 900 \leq \theta < 2250 \\ 0, & 2250 \leq \theta < 2700 \\ -1, & \theta \geq 2700 \end{cases}$$

5. Find servo PWM values

$$\begin{aligned} left_{PWM} &= (R * k_{left} + 992) + 1280 \\ right_{PWM} &= -(R * k_{right} + 992) + 1280 \end{aligned} \tag{57}$$

6.2.2. Canary Quadcopter

The Canary is a 260-millimeter quadcopter that features a carbon fiber frame and four motors with 5-in, 3-blade propellers. Control of the Canary is handled by a Matek flight controller with outputs to four ESCs. The quadcopter is powered by a 11.1-V 3-cell lithium polymer battery. Figure 49 shows the base configuration of the Canary.

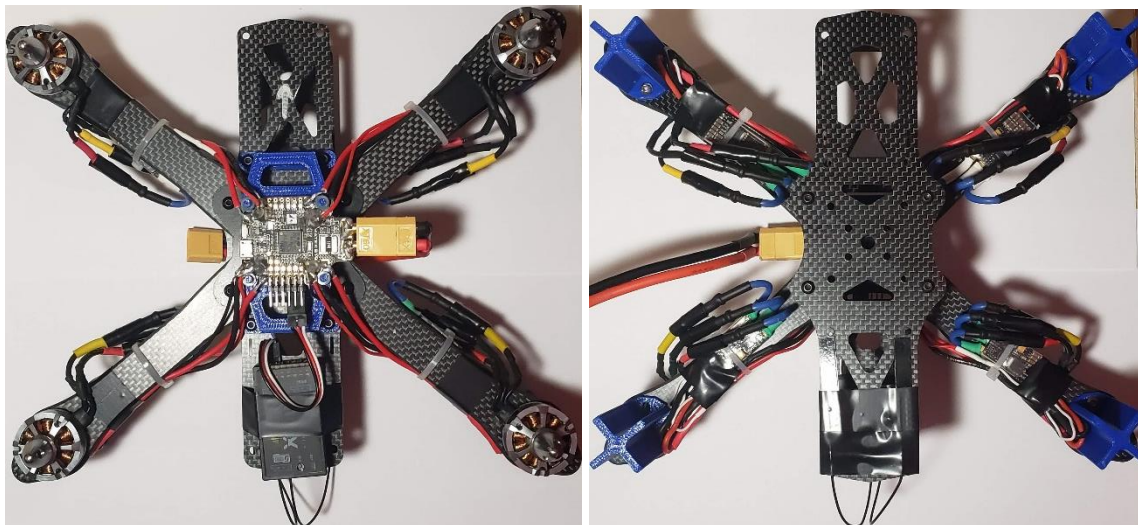


Figure 49: Canary quadcopter platform

6.2.2.1. Flight Controller

The Matek F722-SE flight controller used on the Canary features a 216-MHz STM32F722RET6 microcontroller, dual gyroscopes, a barometer, and an accelerometer. The

controller supports up to four ESCs and has built-in inverters for SBUS signals [48]. The flight controller was chosen for its easy programmability via the iNav Configurator software, blackbox logging, and automatic flight stabilization. Figure 50 shows a stock photo of the flight controller.

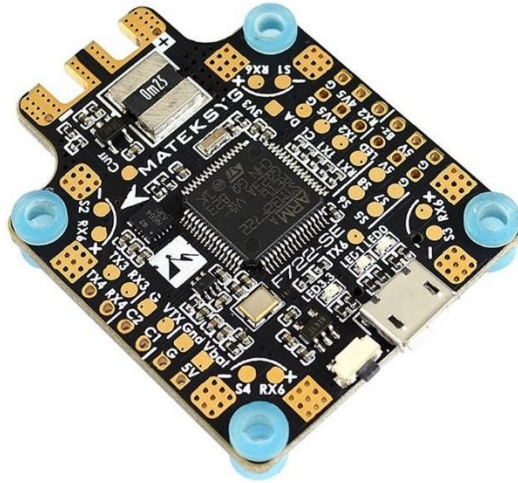


Figure 50: Matek F722-SE flight controller [48]

6.2.2.2. Motors & ESCs

The motors on the Canary are Gattt ML2205S, 2300kV DC motors. Each motor is controlled by a Spedix ES30-HV ESC. The ESCs are capable of handling 30 A of continuous current with support of up to 40 A of burst current. The integrated microcontroller provides reliability and quick response times to flight controller commands [49]. Figure 51 shows a stock photo of the ES30-HV.



Figure 51: Spedix ES30-HV electronic speed controller

6.3. RC Receiver & Transmitter

During the development of the LCAS, a FrSky X8R 2.4-GHz RC receiver was used. The X8R features 16-channel support for SBUS and telemetry feedback [50]. The X8R is shown in Figure 52.

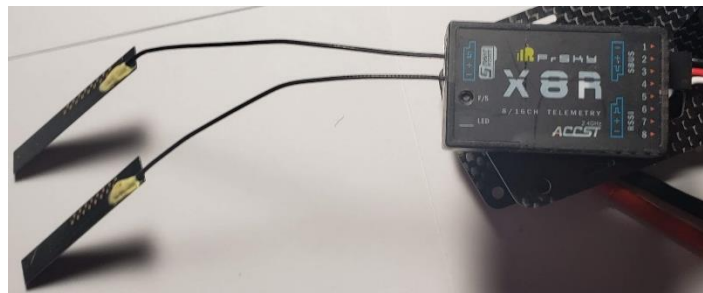
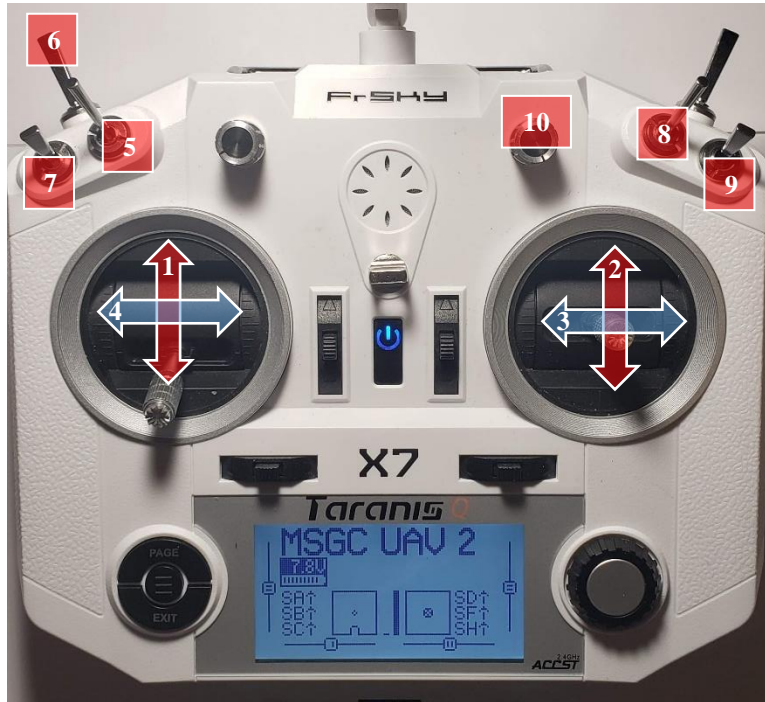


Figure 52: FrSky X8R receiver

The transmitter used was the FrSky Taranis Q X7. The Taranis Q X7 supports up to 32 channels and uses the OpenTX software for customizing control schemes [51]. The Taranis Q X7 is shown in Figure 53 with the controls used for testing labelled. For details about the channel naming scheme refer to Section 3.2.3.



- | | |
|------------------|--------------------|
| 1. <i>Thr</i> | 2. <i>Ele</i> |
| 3. <i>Ail</i> | 4. <i>Rud</i> |
| 5. <i>ARM</i> | 6. <i>Hld</i> |
| 7. <i>LOG</i> | 8. <i>sbEN/vEN</i> |
| 9. <i>ctrlEN</i> | 10. <i>VEL</i> |

Figure 53: FrSky Taranis Q X7 transmitter

6.4. Sensor Board

As the main distance measurement device of the LCAS, the Sensor Board controls the operation of a single ultrasonic sensor alongside dual tinyLiDARs. The Sensor Board utilizes a dedicated MSP430G2553 microcontroller to control the sensors and process the distance measurements. By using a dedicated microcontroller, the Sensor Board is able to operate mostly independent of the MITM, only needing commands for triggering and distance reporting. Through this independency, multiple Sensor Boards are able to be operated at the same time.

6.4.1. Operation Concept

The Sensor Board conducts several operations in order to provide the smallest distance measurement to the MITM. To begin with the Sensor Board is connected as an I2C slave device

(see Appendix A) to the MITM and is assigned a unique address. The Sensor Board will only operate if its address is called by the MITM and the proper commands are received. When a triggering command is received, the Sensor Board triggers its sensors and records the measured distances. Then the Sensor Board processes the distance measurements, checking for invalid measurements and comparing the measurements to determine the smallest distance measurement. With the smallest measurement determined, the Sensor Board stores the value until the MITM calls its address again and requests the value.

6.4.2. Sensors

The Sensor Board uses two different types of sensors in order to make measurements: the HC-SR04 ultrasonic sensor and the tinyLiDAR TOF LiDAR. Each sensor complements the other. For example, the ultrasonic struggles with measuring distances to porous materials as the sound waves are absorbed but the infrared light pulses from the tinyLiDAR are not. Furthermore, since the ultrasonic is relatively slow to provide distance measurements, the tinyLiDARs are used to provide supplemental distance measurements in the interim, allowing the LCAS to keep operating while waiting for the ultrasonic. Once the ultrasonic provides a measurement the Sensor Board can then compare that to the measurements provide by the tinyLiDARs, choosing the smallest value.

6.4.2.1. HC-SR04 Ultrasonic Sensor

The HC-SR04 ultrasonic sensor is a low-cost, range-finding sensor that utilizes sonar to determine the distance to objects. The sensor has a measuring range of 2 to 400 cm, a 30° measuring angle, and a resolution of 3 mm. The HC-SR04 package contains modules for both transmitting and receiving ultrasonic signals [6]. An example of the HC-SR04 package is shown in Figure 54.



Figure 54: HC-SR04 ultrasonic range finding sensor [6]

The Sensor Board microcontroller controls the ultrasonic by first sending a 10- μ s pulse to the trigger pin. When the ultrasonic detects the trigger pulse, the transmitter module will output eight cycles of a 40-kHz signal [8]. If an echo of the signal is detected the ultrasonic pulls the ECHO pin high and keeps it high for the same amount of time there was between emitting the trigger signal and receiving an echo. The time between the raising and lowering of the echo pin is proportional to the distance traveled by the 40-kHz signal. Therefore, the distance from the ultrasonic to an object is proportional to half of the echo time.

After capturing the echo time, the Sensor Board's microcontroller can determine the distance by using the following equation, courtesy of [8]:

$$d = \frac{1}{2} t v_{sound} \quad (58)$$

where d is the measured distance in meters, t is the length of the echo time in seconds, and v_{sound} is the speed of sound (343 m/s).

However, Equation (58) does not take into consideration the clock speed of the microcontroller and that the microcontroller's measurement is a count of clock cycles rather than

time. Taking the clock speed and clock cycle count into consideration, the echo time can instead be found using:

$$t = \frac{n_{cycles}}{f_{clock}} \quad (59)$$

where n_{cycles} is the number clock cycles in the echo time and f_{clock} is the frequency of the microcontroller's clock (16 megacycles per second). Now substituting Equation (59) into Equation (58):

$$d = \frac{1}{2} \left(\frac{n_{cycles}}{f_{clock}} \right) v_{sound} \quad (60)$$

An issue arises when using this equation on the Sensor Board's MSP430G2553 microcontroller, however. The multiple multiplication and division operations result in floating point values and register overflow errors.

Instead of using Equation (60), a linear regression was used to simplify the conversion and remove the errors caused by register overflow. The independent variable was made the actual distance, in millimeters, and the dependent variable was the number of clock cycles counted by the microcontroller. For each 50-mm increase of the actual distance, the number of clock cycles was recorded. The recorded data are shown in Table VIII.

Table VIII: Ultrasonic linear regression values

Distance [mm]	Clock Cycles
50	762
100	1319
150	1981
200	2496
250	3195
300	3646
350	4030
400	4408
450	4985
500	5836
550	6467
600	6938
650	7576
700	8221
750	8748
800	9323
850	9894
900	10,437
950	10,965
1000	11,255

The values were then imported into MATLAB and using the *regression()* function the following equation was generated:

$$d = 0.0884n_{cycles} - 16 \quad (61)$$

where d is the distance in millimeters.

Figure 55 is a graphical representation of Equation (61).

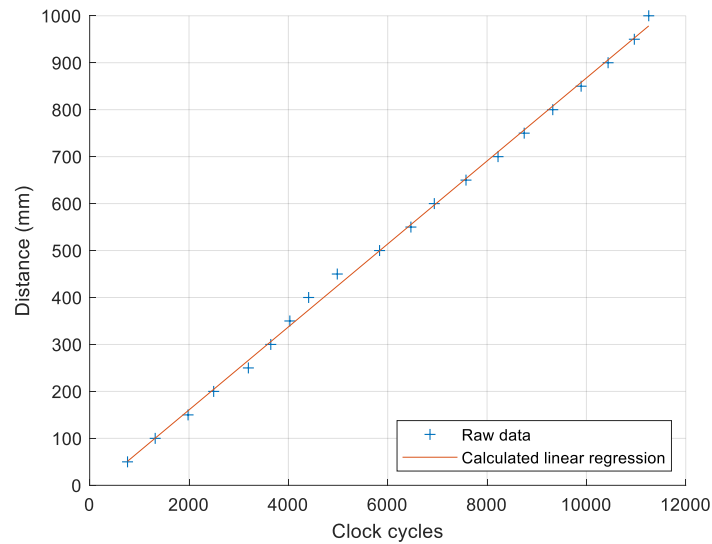


Figure 55: Ultrasonic linear regression

6.4.2.2. tinyLiDAR

The tinyLiDAR TOF Range Finder Sensor is a module that combines a single point TOF LiDAR with a dedicated microcontroller. Developed by MicroElectronicDesign, the module is designed around ST Microelectronics' VL53LOX TOF laser-ranging sensor, which is intended for high performance devices, such as smartphones. A 32-bit, ARM-based STM32L051C8 microcontroller was implemented to serve as a dedicated controller for the VL53LOX. The microcontroller handles communication and control of the VL53LOX and simplifies the operation to simple I2C commands received from an I2C master [15] [52]. Figure 56 shows the tinyLiDAR module and labels the module's main components.

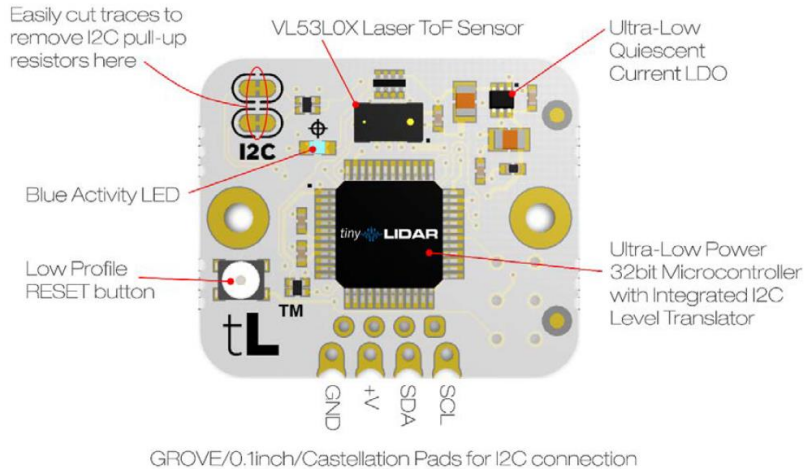


Figure 56: tinyLiDAR TOF Range Finder Sensor [15]

The tinyLiDAR's VL53L0X infrared range finder operates on the same principles as the HC-SR04 ultrasonic sensor. When triggered, the range finder emits a 940-nm infrared pulse via a Class 1 Vertical Cavity Surface-Emitting Laser [53]. Moving at the speed of light, the pulse moves through an environment until it is reflected off of an object. The reflection is then captured by the range finder. The tinyLiDAR's microcontroller interprets the time between trigger and capture as proportional to the distance the infrared pulse traveled.

The Sensor Board utilizes dual tinyLiDARs for distance measurements. In order to control the modules, the Sensor Board's microcontroller hosts an I2C bus (see Appendix A), acting as the master with the tinyLiDARs acting as the slaves. When requesting a distance measurement, the Sensor Board's microcontroller calls each tinyLiDAR's I2C address and transmits the distance capture command. The tinyLiDARs then capture individual distance values using the method described above. Next, the Sensor Board's microcontroller calls for each tinyLiDAR's distance value and compares the two values to determine the smallest distance measurement. For a more detailed operation of the tinyLiDARs refer to Section 6.4.4.3.

The tinyLiDAR's VL53LOX range finder has a range of up to 2 m [15]. Outside of this range, the tinyLiDAR will report nonsensical distance values. To account for such values the Sensor Board's microcontroller compares the reported distance values to an error threshold equivalent to the distance from the Sensor Board to the tips of the propellers on the UAV platform. If one of the tinyLiDAR's distance value is within the error threshold, that distance value is discarded and the other tinyLiDAR's distance value is chosen. If both distance values are within the error threshold, then both values are discarded and the Sensor Board's microcontroller defaults to the distance value reported by the ultrasonic.

6.4.3. Main Controller

The main controller of the LCAS Sensor Board is a Texas Instruments MSP430G2553 mixed signal microcontroller. This ultra-low-power microcontroller uses a 16-bit RISC central processing unit (CPU) that operates at a frequency of up to 16-MHz. Alongside the CPU, the microcontroller has 512 bytes of random-access memory (RAM) and 16 kilobytes of flash storage. The MSP430G2553 features two 16-bit timers, up to 24 input/output pins, a 10-bit analog-to-digital converter, and a Universal Serial Communication Interface (USCI) [54]. Figure 57 shows the MSP430G2553 package used on the LCAS Sensor Board.

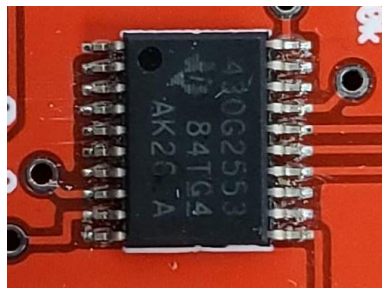


Figure 57: LCAS Sensor Board's MSP430G2553

The Sensor Board's microcontroller operates at the maximum 16-MHz frequency and uses a 20-pin thin-shrink small-outline package (TSSOP). The three sensors and communication to the MITM are controlled using a variety of combinations of the microcontroller's features. The ultrasonic is handled by voltage level control on a pair of pins and the time between trigger and echo is measured via one of the 16-bit timers. The tinyLiDARs are handled by a bitbanged I2C bus setup on another pair of pins. And finally, communication to the MITM is handled by the built-in I2C bus under the USCI.

6.4.4. Detailed Operation

The LCAS Sensor Board is responsible for recording and reporting the minimum distance to obstacles and it does so in a specific order of operation. The order of operation is required because of the difference in operating times of the tinyLiDARs and the ultrasonic. To simplify the description the Sensor Board's operations are separated into the following sections: initialization, ultrasonic control, tinyLiDAR control, distance value processing, and MITM distance reporting.

6.4.4.1. Initialization

Initialization of the Sensor Board begins the moment that the board receives power from the MITM. Most of the initialization is handled by the Sensor Board's microcontroller with the tinyLiDARs initializing themselves to firmware defaults. The microcontroller starts by setting the CPU clock and timer reference clocks to 16 MHz and then disables the automatic watchdog timer so that the Sensor Board can run its code on a loop indefinitely. Next the microcontroller initializes communication to the MITM by opening the I2C bus on the USCI. With the USCI's I2C enabled, the microcontroller sets up a separate I2C bitbanging bus for control of the tinyLiDARs. The microcontroller then proceeds onto the ultrasonic, where it sets up the needed

hardware interrupts for the trigger and echo pins and configures one of the timers for tracking the echo time. With all three sensors initialized, the microcontroller begins listening on the USCI I2C bus for commands sent by the MITM. The commands accepted by the Sensor Board are listed in Table IX.

Table IX: Sensor Board I2C commands

Command	Description
0x45	Set tinyLiDAR error threshold
0x54	Trigger only tinyLiDARs and report minimum distance
0x55	Trigger ultrasonic
0x56	Trigger tinyLiDARs, capture ultrasonic distance value, and report minimum distance

Before the Sensor Board can control the sensors, the MITM must provide an error threshold. The error threshold is a minimum distance value that the Sensor Board can see before the UAV crashes. For example, on a UAV the error threshold could be the distance from the Sensor Board to the tips of propeller blades. The main use of the error threshold is for checking tinyLiDAR values, since when the tinyLiDARs do not detect an obstacle it will report small, nonsensical distance values. If the value is smaller than the threshold, then the Sensor Board overrides the reading with a large value that the tinyLiDAR would not normally report.

6.4.4.2. Ultrasonic Control

The ultrasonic trigger sequence begins when the Sensor Board's microcontroller receives a 0x55 command. The microcontroller raises the voltage level on the TRIG pin (2.3) and holds the voltage high for approximately 10 μ s. This triggers the ultrasonic to emit a 40-kHz pulse for eight cycles. The microcontroller also sets up a hardware interrupt to detect a low-to-high voltage on the ECHO pin (2.4).

When the ultrasonic receives an echo, it raises the voltage level of its ECHO pin and holds the pin high for the same amount of time measured between emission of the trigger signal

and the echo received. The microcontroller detects the voltage change and interrupts. A timer counter is then started and counts the clock cycles of the microcontroller until a high-to-low voltage shift is detected on the ECHO pin. Finally, the count is stored in memory until the microcontroller is ready to process the value alongside tinyLiDAR values.

If there is no high-to-low voltage shift after approximately 25 ms (amount of time needed for a sound wave to travel 8 m), then the timer interrupts and flags the ultrasonic as timed out and no distance value was captured.

6.4.4.3. tinyLiDAR Control

Control of the tinyLiDARs is initiated by the Sensor Board's microcontroller receiving a 0x54 or 0x56 command. The only difference between the two commands is whether the microcontroller only processes the tinyLiDARs' outputs (0x54) or includes the ultrasonic output in the processing (0x56).

Once either command is received, the microcontroller opens bitbanged I2C communication to the first tinyLiDAR (0x10 address). The microcontroller then transmits a 0x44 command to the tinyLiDAR. Upon receiving the command, the tinyLiDAR emits an infrared light pulse and waits to detect a reflection of the light. The tinyLiDAR then converts the time between emission and reflection capture to a distance value. The tinyLiDAR transmits the distance value across two bytes, which the microcontroller combines to reproduce the value. With operation of the first tinyLiDAR complete, the microcontroller repeats the same operation with the second tinyLiDAR (0x50 address).

6.4.4.4. Distance Value Processing

The Sensor Board processes the distance values from the tinyLiDARs and the ultrasonic in two stages. The first stage (0x54 command) triggers, error-checks, and compares the

tinyLiDAR values, choosing the smallest. The second stage (0x56 command) compares the result from the first stage to the ultrasonic value and chooses the smallest. This final value is the minimum distance reported that the Sensor Board will report to the MITM.

For the first stage, the Sensor Board triggers and then error-checks tinyLiDAR distance values by comparing each value to the error threshold indicated by the MITM during initialization. If either value is below the threshold, then the value is changed to the error value 0xB BBB (a value larger than any distance value that can be captured by the tinyLiDARs). If the first tinyLiDAR's distance value is smaller than or equal to the second tinyLiDAR's distance value, then the first tinyLiDAR's distance value is chosen as the minimum distance reported from the tinyLiDARs. If the second tinyLiDAR's distance value is smaller, then that value is reported as the minimum distance.

The second stage takes the resultant distance value from the first stage and compares it to the ultrasonic distance value. If the tinyLiDAR distance value is smaller than or equal to the ultrasonic distance value, then the tinyLiDAR distance value is chosen as the minimum distance to be reported to the MITM. If the ultrasonic distance value is smaller than the tinyLiDAR distance, then the ultrasonic distance value is chosen as the minimum distance to be reported to the MITM.

The Sensor Board distance value processing is summarized in Figure 58.

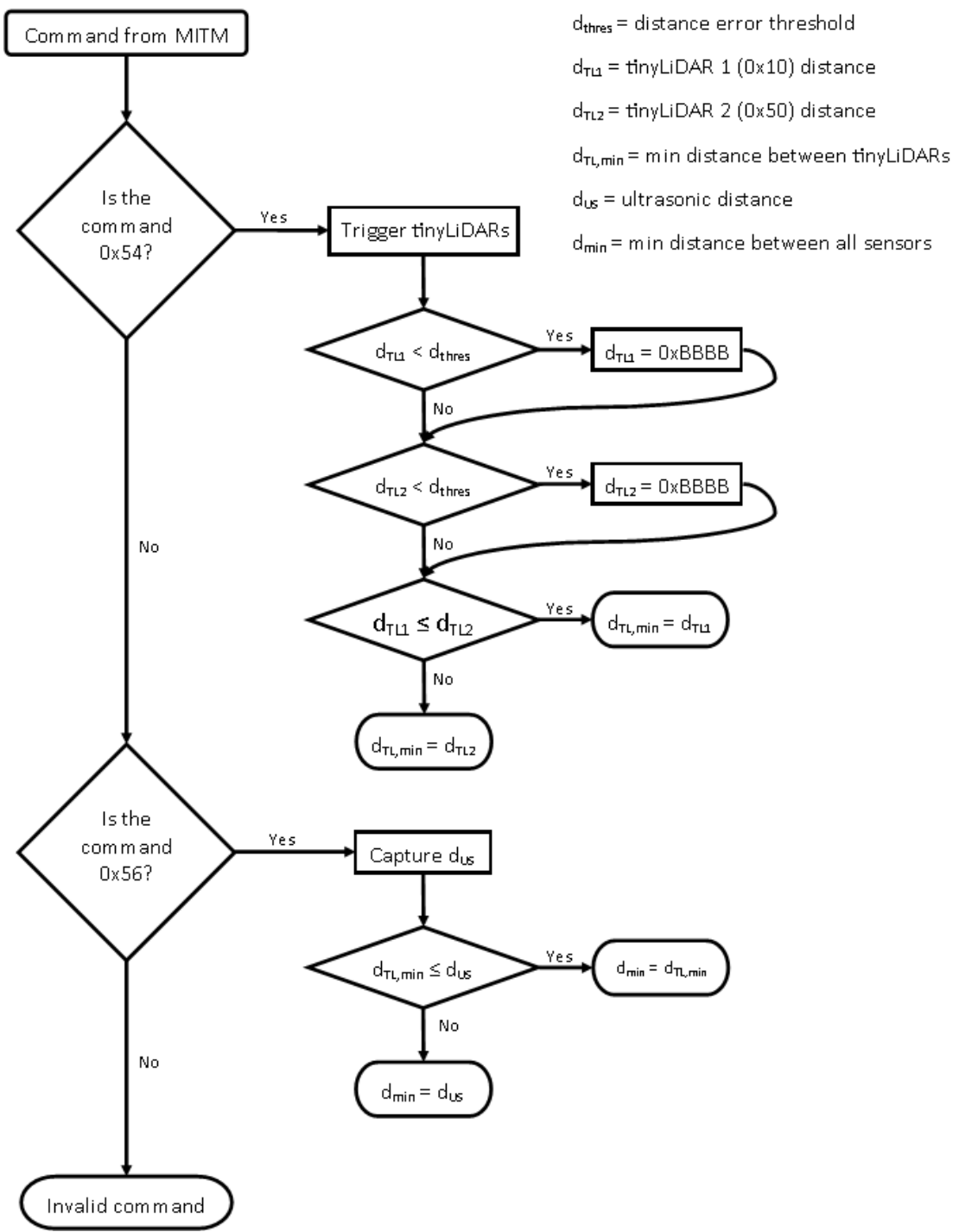


Figure 58: Flowchart of Sensor Board distance value processing

6.4.4.5. MITM Distance Reporting

The final operation of the Sensor Board is the reporting of the minimum distance between the three sensors to the MITM. When the Sensor Board receives a read request from the MITM, it sends the MITM four bytes carrying three data values.

The first byte is a status flag, that indicates whether the Sensor Board was able to capture a distance value or not. A successful capture is indicated by *P*; an unsuccessful capture is indicated by *F*.

The second and third bytes contain a 16-bit representation of the minimum distance value. The distance value is split across the bytes using the following equations:

$$byte_2 = (dist \& 0xFF00) \gg 8 \quad (62)$$

$$byte_3 = dist \& 0x00FF \quad (63)$$

The second byte is created by Equation (62) which separates out the most significant byte of the distance value. Created by Equation (63), the third byte contains the least significant byte of the distance value. For example, if the distance value was 300 (0x012C in hexadecimal), then the second byte would be 0x01 via Equation (62) and the third byte would be 0x2C via Equation (63).

The fourth and final byte contains the device identifier for which distance value was selected as the minimum distance. If the ultrasonic distance value was selected, then the fourth byte is *U*. If the first or second tinyLiDAR distance value was selected, then the fourth byte is *1* or *2*, respectively. If there was an error when capturing distance values, then the fourth byte is set to *X*.

6.4.5. PCB Design

The LCAS Sensor Board is a custom printed circuit board (PCB) that condenses all of the board's components into a small package. The board measures 60 mm by 26 mm and contains the following components listed in Table X. Figure 59 provides a photo of the Sensor Board and labels the components, according to Table X.

Table X: LCAS Sensor Board components

Label	Component	Amount
U1	MSP430G2553 microcontroller	1
U2	AP2210N-3.3TRG1 voltage regulator	1
U3	BSS138 MOSFET	2
U4	Grove 4-pin female connector	3
U5	4-pin male pinheader	1
U6	4-pin female pinheader	1
U7	10-k Ω resistor	5
U8	2.2- μ F capacitor	1
U9	1- μ F capacitor	1

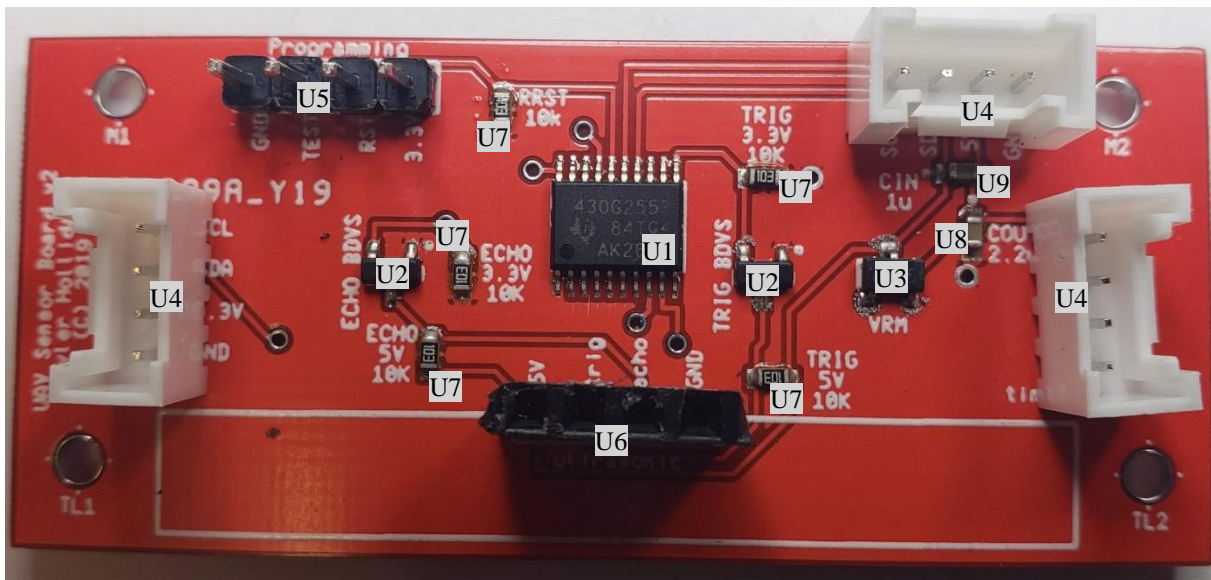


Figure 59: LCAS Sensor Board

There are three main circuits on the Sensor Board. The first is the power circuit, which uses the AP2210N-3.3TRG1 regulator. The second circuit is the bidirectional voltage shifter,

using the BSS138 MOSFET. The third circuit is the MSP430G2553 interface, including the tinyLiDAR and MITM I2C buses.

The PCB layout of the Sensor Board can be found in Appendix B.

6.4.5.1. Power Circuit

The Sensor Board power circuit uses an AP2210N-3.3TRG1 regulator to reduce the 5 V input to the 3.3 V level used by the MSP430G2553 and tinyLiDARs. Figure 60 shows a schematic of the power circuit.

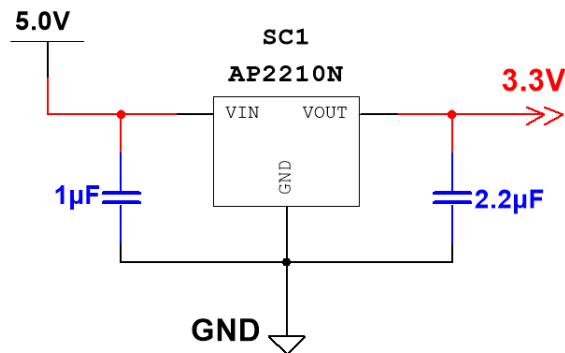


Figure 60: Sensor Board power circuit schematic

The AP2210N-3.3TRG1 regulator was chosen for its excellent output accuracy ($\pm 1\%$) and power supply ripple rejection (75 dB at 100 Hz) [55]. The regulator is capable of outputting a steady 3.3 V, a critical component of maintaining power to the Sensor Board's microcontroller and tinyLiDARs.

The 1- μ F and 2.2- μ F capacitors are used to suppress noise caused by the length of cable used to power the Sensor Board.

6.4.5.2. Bidirectional Voltage Shifters

A pair of bidirectional voltage shifters (BDVS) are needed to shift the voltage levels between the Sensor Board microcontroller and the HC-SR04 ultrasonic sensor. The ultrasonic is

controlled by 5 V levels but, as mentioned in the previous section, the MSP430G2553 microcontroller operates at 3.3 V. The BDVS serves to raise and lower the voltage levels of the TRIG and ECHO lines, respectively. A schematic of the BDVS, courtesy of [56], is shown below in Figure 61.

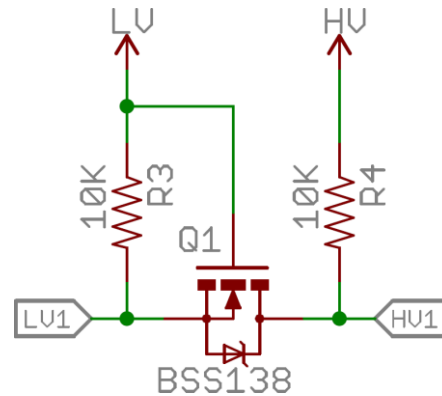


Figure 61: Bidirectional voltage shifter schematic [56]

The BSS138 is an N-channel MOSFET that isolates the two voltage levels, in this case LV is 3.3 V and HV is 5 V. The voltage shifting occurs in three different cases.

The first case is when the LV1 is not being pulled low by a device, and is thus pulled to 3.3 V via the pullup resistor. Since the BSS138's gate and source are both at 3.3 V, the MOSFET is not conducting, allowing the pullup resistor on HV1 to pull the line to 5 V [57].

The second case is when LV1 is pulled low by a device, causing the source to be low. The voltage differential between the gate and source allows the BSS138 to become conducting. Thus, HV1 is pulled low by LV1 [57].

The third case is when HV1 is pulled low by a device. The drain-substrate diode pulls LV1 low enough to cause the BSS138's gate and source to have a voltage differential. This allows the MOSFET to become conducting and thus pulls LV1 low to match HV1 [57].

6.4.5.3. MSP430G2553 Interface

The MSP430G2553 microcontroller controls the entirety of the Sensor Board's operations and thus has several connections. The microcontroller uses two separate I2C buses, one for the tinyLiDARs and another for the MITM. Control of the ultrasonic is passed through the BDVS. And finally, a programming interface is included. Figure 62 shows the full schematic of the LCAS Sensor Board, including the MSP430G2553 interface. Table XI provides a summary of the pin mapping for the microcontroller as shown in the figure.

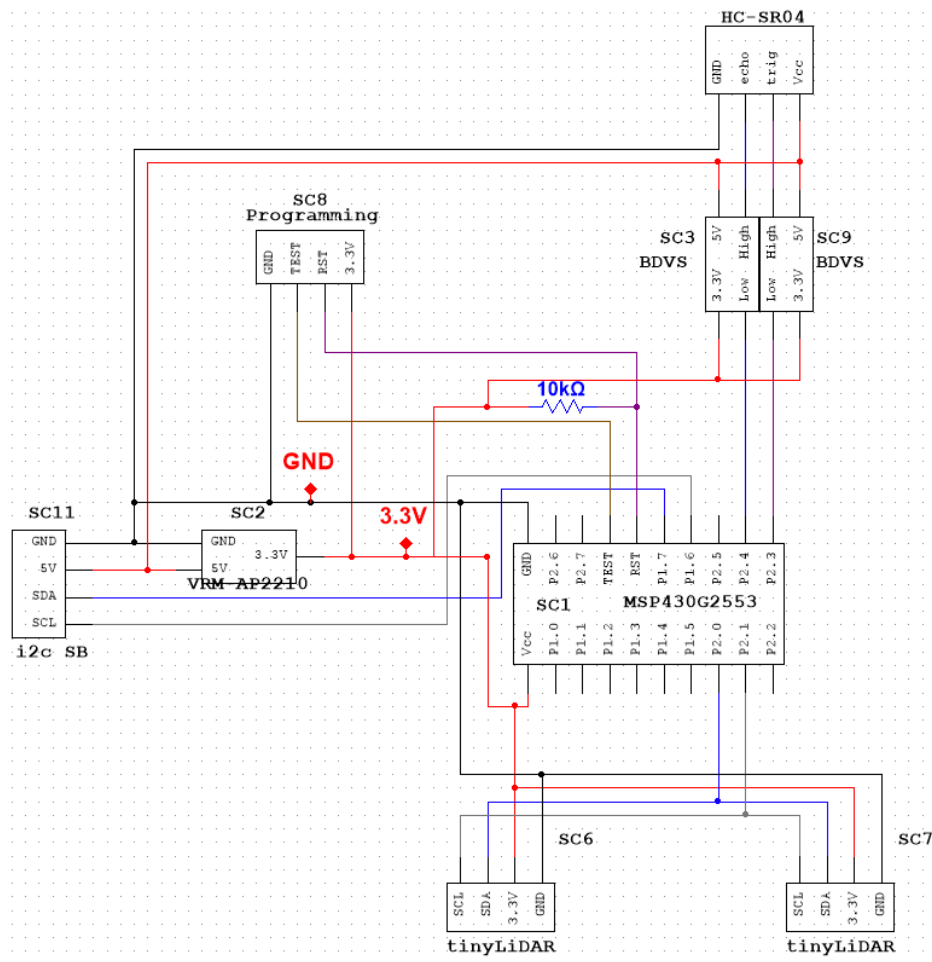


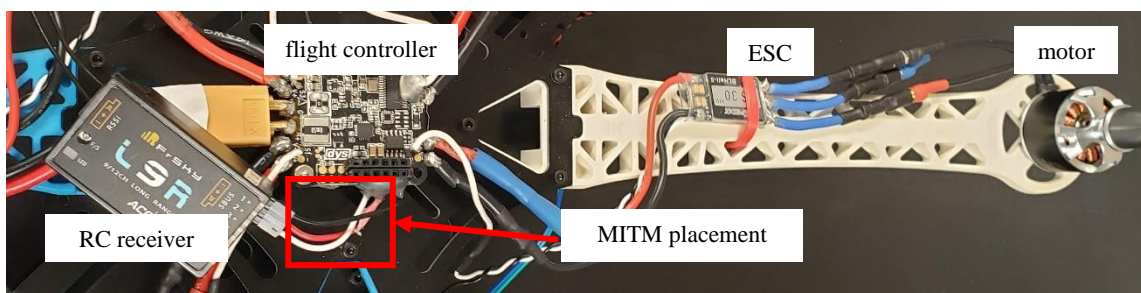
Figure 62: LCAS Sensor Board schematic

Table XI: Sensor Board MSP430G2553 pin mapping

Pin	Description
Vcc	3.3V supply
P1.0 – P1.5	unused
P1.6	MITM SCL
P1.7	MITM SDA
P2.0	tinyLiDAR SDA
P2.1	tinyLiDAR SCL
P2.2	unused
P2.3	Ultrasonic TRIG via BDVS
P2.4	Ultrasonic ECHO via BDVS
P2.5 – P2.7	unused
RST	Reset (programming)
TEST	Testing mode enable (programming)
GND	Ground

6.5. MITM

The Monkey-in-the-Middle (MITM) of the LCAS is the system's main processing unit and controls all functions of the system. Implementation of the MITM places it between a UAV's RC receiver and flight controller. In this location the MITM intercepts the SBUS signals being output by the receiver. The MITM decodes, modifies, and encodes the signals before passing the signals onto the flight controller. The advantage of this arrangement is that the flight controller is not aware of the MITM's modifications to the receiver's output. In other words, the flight controller sees the MITM's output as the output of the RC receiver, essentially rendering the MITM as an invisible middle step, hence the name. Figure 63 shows the layout of a quadcopter control system and where the MITM would be placed.

**Figure 63: Location of the MITM in the layout of a quadcopter control system**

There have been two versions of the MITM. The first version (Section 6.5.2) made use of a MSP430G2553 microcontroller for SBUS decoding and encoding, alongside control of a single HC-SR04 ultrasonic range finder. Due to challenges associated with using a low power microcontroller, such as data logging and processing power, a second version of the MITM was developed. The second and current version (Section 6.5.3) of the MITM utilizes a Raspberry Pi, which is significantly more powerful than the previous version's MSP430G2553.

6.5.1. Operational Concepts

Since the MITM is the center of processing in the LCAS, it is responsible for the following: capturing and decoding SBUS frames, control of the Sensor Boards, feedback control based on distance measurements, encoding and transmitting SBUS frames, and logging of all critical data.

For SBUS communications, the MITM intercepts the SBUS frames being passed from the RC receiver to the flight controller. Once a full SBUS frame is captured, the MITM decodes the frame into individual channels using the method described in Section 3.2.1.

Control of the Sensor Boards is accomplished by communication over an I2C bus (see Appendix A). The second version of the MITM is capable of identifying the number of Sensor Boards available and iteratively controlling each board. From each board, the MITM requests a value that corresponds to the distance from the board to a possible obstacle.

The distance values are filtered using a 10-point SMA (simple moving average) and then compared to a desired position. The error between the two and the RX SBUS channels are input into the Phase II controller algorithm, Equation (50). Using the current and previous inputs, the controller calculates new channel values that over time will drive the Canary to the desired position. If the measured position is determined to be in the controller's activation window and

the relevant SBUS channel is either greater than or lower than the SBUS neutral, depending on the direction, then the MITM will replace the relevant SBUS channel values with the new channel values. At the time of writing, the controller is only designed to operate in the forward direction, so the SBUS channel will be greater than the SBUS neutral.

Having completed modification of the SBUS channels, the MITM encodes the channels into a new frame using the method described in Section 3.2.2. The frame is then transmitted to the flight controller.

The final operation of the MITM is the logging of data used during the other operations. This includes the SBUS channel values from both the receive and transmit operations, as well as the distance values reported by the Sensor Boards.

6.5.2. MSP430G2553 Version

The first version of the MITM used a Texas Instruments MSP430G2553 microcontroller (specifications detailed in Section 6.4.3). The microcontroller was used since it was readily available and a familiar platform. Operating at approximately 16 MHz, the microcontroller was fully capable of receiving and transmitting SBUS signals. Before being replaced, control of a single ultrasonic was added to the MSP430G2553's programming algorithm.

Figure 64 shows the first version of the MITM using a MSP430G2553 for SBUS communications and control of an ultrasonic.

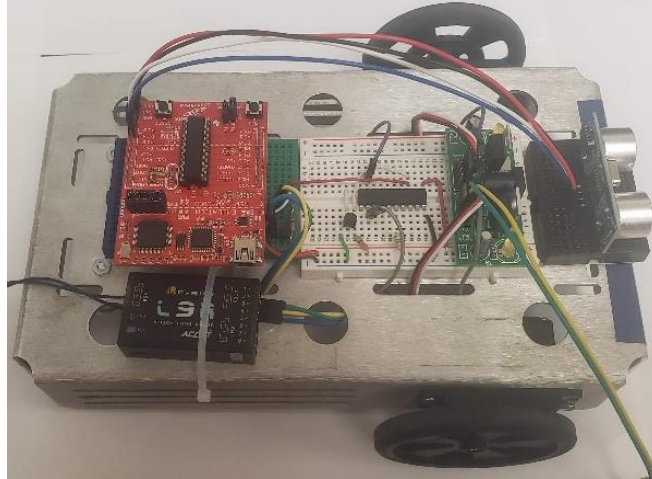


Figure 64: MSP430G2553 version of the MITM on the SBUS-to-PWM Rover

6.5.2.1. SBUS Communications

The MSP430G2553 version of the MITM served as the main platform for developing the algorithm needed for decoding and encoding SBUS signals. Since the SBUS protocol is based on UART (see Appendix A), the MSP43G2553 was able to make use of its built-in serial communication interface. However, the SBUS signal out of the RC receiver uses inverted voltage levels that cannot be interpreted by the MSP430G2553. Therefore, the signal was inverted using the inverter circuit shown in Figure 14.

Since SBUS uses a non-standard baud rate of 100 kilobits per second, a time difference constant had to be calculated so that the microcontroller could synchronize with the SBUS timing. The constant was determined by using two of the built-in timers on the MSP430G2553. One timer counted the clock cycles of the microcontroller's internal oscillator, while the other timer counted the clock cycles of an external 32-kHz crystal oscillator. A time difference between the two counts was calculated and then stored in the MSP430G2553's flash storage. Every time the SBUS communications are initialized the time difference is loaded from the flash storage and used to offset the microcontroller's clock, synchronizing with the SBUS timing.

With the time difference constant and non-standard baud rate set, the MSP430G2553 reads in a single SBUS frame. The frame is decoded into individual channels and then immediately encoded back into a new SBUS frame that is then transmitted to the servo controller.

6.5.2.2. Ultrasonic Control

Before development of the Sensor Board, another MSP430G2553 version of the MITM was developed to test how communications with the Sensor Board should be done in the second version of the MITM. An ultrasonic was used because it is the slower of the two sensors used in the final Sensor Board design. Since the ultrasonic operates in two stages, TRIG and ECHO (see Section 6.4.4.2 for detailed ultrasonic operation), it was decided that the MITM would trigger the ultrasonic on the first loop and then conduct SBUS communications. At some point during the SBUS signal processing the ultrasonic reports the time duration for the ECHO, which the microcontroller stores as a clock cycle count. On the second loop, the microcontroller converts the count to a distance value. After a new SBUS frame is received and decoded the distance value is compared to a pair of distance thresholds. If the distance value is between 250 mm and 500 mm, then the channel value related to the forward motion of the rover is scaled down, using Equation (61). If the distance value is 250 mm or less, then the same channel value is set to the SBUS neutral value, preventing the rover from moving forward.

6.5.2.3. Reasons for Replacement

As the complexity of the MITM increased, the MSP430G2553 became limited in function and had to be replaced for the second version of the MITM. The MSP430G2553 is only capable of conducting one task at a time, thus multiple tasks have to be done in series. The second version of the MITM introduced new features that were best suited for a faster processor.

The most significant feature introduced in the second version was the ability of the MITM to log all of the data relevant to its operations, such as both RX and TX SBUS frames and distance values captured from the Sensor Boards. The MSP430G2553 does not natively support data logging and implementing it would have required using even more of the microcontroller's clock cycles.

Another complication introduced by the second version of the MITM, was the need for more processing power. The second version incorporated communication with multiple Sensor Boards and the processing of the boards' data. With the Sensor Board data available, the MITM would then begin to use a feedback control algorithm for each direction that had a Sensor Board. By adding more tasks and need for processing power, it was obvious that the MSP430G2553 would not suffice as the main processing unit of the MITM.

6.5.3. Raspberry Pi Version

A Raspberry Pi 3 Model B (Pi for short) was chosen as the MSP430G2553's successor in the second version of the MITM due to the significant increase in processing power and available coding libraries. The Pi features a quad-core 1.2-GHz Broadcom BCM2837 64-bit ARM-based processor supplemented with 1 GB of RAM [58].

There are a variety of connectivity methods on the Pi, such as a 40-pin general-purpose input/output (GPIO) and built-in Wi-Fi. Figure 65 shows an example of the Pi with major components labeled. Figure 66 shows the pinout for the Pi's 40-pin GPIO.

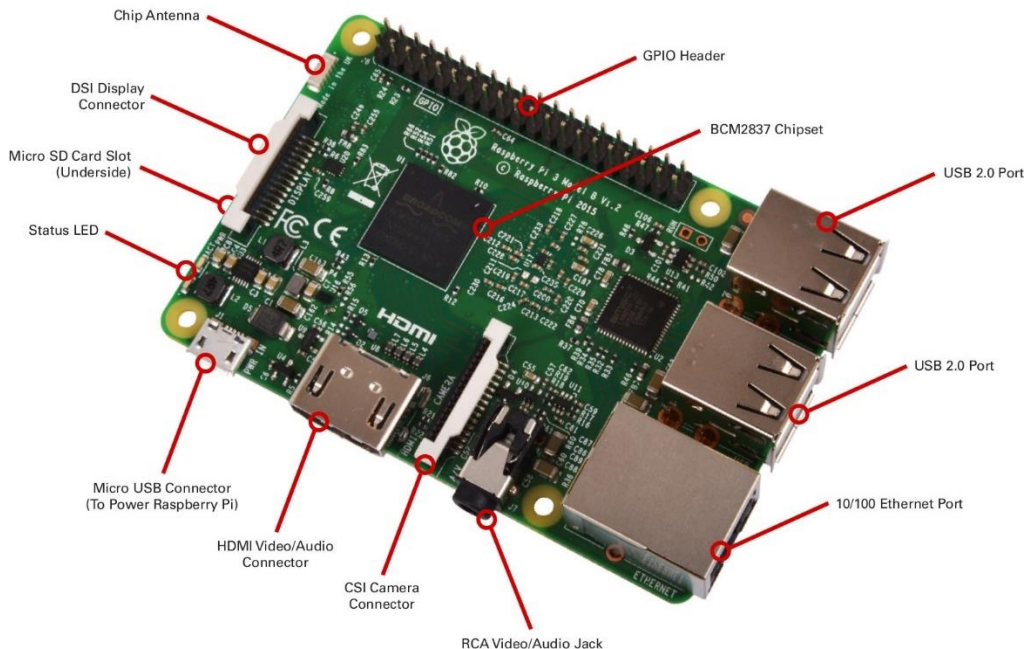


Figure 65: Raspberry Pi 3 Model B [59]

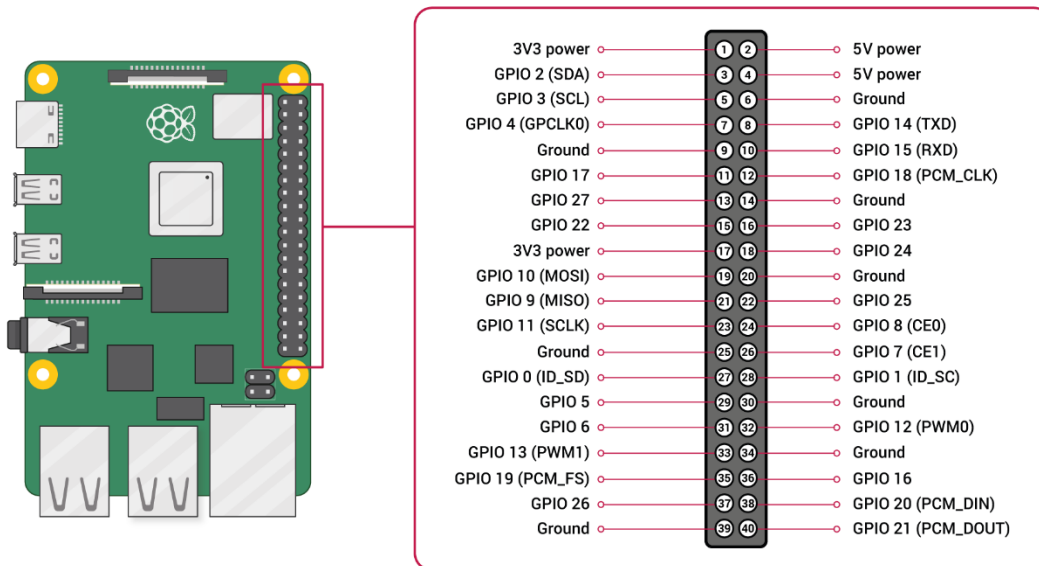


Figure 66: Raspberry Pi GPIO pinout [60]

6.5.3.1. SBUS Communications & Logging

The receiving and transmitting of SBUS signals on the Pi version of the MITM uses similar methodology as the MSP430G2553 version used (Section 6.5.2.1). Unlike the

MSP430G2553, the Pi is capable of using custom baud rates in serial communication without external calibration. Therefore, the UART bus (GPIO 14 and 15 in Figure 66) is initiated at 100k baud and set to read a data packet with a length of eight bits, even parity, and two stop bits.

Incoming SBUS signals are first passed through a voltage level inverter (Figure 14). With the SBUS signals modified to simulate UART, the MITM captures 50 packets of data (length of two SBUS frames). Since a SBUS frame's start and end bytes are known, the MITM checks for an end byte, starting with the last received byte and working backwards. Once an end byte is found, the MITM checks for a start byte that is 24 bytes before the detected end byte. Now that the start and end bytes are found, the MITM separates out the 25 bytes from the captured SBUS signals.

Decoding of the SBUS frame is done via the method described in Section 3.2.1. With the individual channels decoded, the MITM uses distance values reported by the Sensor Boards in a feedback control loop to determine if the channel values need to be altered. Once alterations, if any, are made to the channels the MITM encodes the channels back into a SBUS frame.

Transmission of the SBUS frame generated by the MITM is done by re-encoding (Section 3.2.2) and then outputting the frame over the UART bus. The output on the UART bus is passed through another voltage level inverter (Figure 14), inverting the signal and raising the voltage level to 5 V from 3.3 V.

After the generated frame is transmitted, the MITM logs the individual channel values used in control of the Canary from both the RX and TX SBUS frames in a CSV (Comma Separated Values) file.

6.5.3.2. Sensor Board Control & Logging

Besides data logging, the other main feature added in the second version of the MITM was the control of up to six Sensor Boards. As discussed in Section 6.4.4, the LCAS Sensor Boards are communicated with over an I2C bus. The MITM features a built-in I2C bus that is connected to via the serial pins in the 40-pin GPIO (see Figure 66).

Initialization of the Sensor Board communications begins with the MITM scanning the I2C bus for available devices. The Sensor Boards have predetermined I2C addresses that identify the board to the MITM when the board's address is detected. Table XII lists the identifiers, I2C addresses, and locations of the Sensor Boards.

Table XII: Sensor Board identifiers, I2C addresses, and locations

Identifier	I2C Address	Location
F	0x12	Front
B	0x24	Back
L	0x36	Left
R	0x48	Right
U	0x5A	Up
D	0x6C	Down

Since the MITM controls each Sensor Board iteratively, and for simplicity, the following explanation of how the MITM controls the Sensor Boards is limited to a single board.

To initialize the Sensor Board, the MITM sends a 0x45 command and a distance value in millimeters. The 0x45 command sets the Sensor Board's error threshold based on the distance value sent with the command.

Once initialization of the Sensor Board is completed the MITM begins SBUS communications. After an RX SBUS frame is decoded, the MITM checks the *sbEN* channel (see Table IV) to determine if the communication with the Sensor Board is enabled.

Communication with the Sensor Boards occurs in three stages that alternate on each loop of the MITM's code. The reasons for the three alternating stages are the slow measuring speed of

the ultrasonic sensors, the requirement that the MITM must be transmitting a SBUS frame every 15 to 20 ms, and the desire for the MITM to have a distance measurement for each loop of its code.

The first stage begins with the MITM sending the ultrasonic trigger command, 0x55. Next the 0x54 command is issued, which has the Sensor Board request a minimum distance measurement from only the tinyLiDARs. After the first iteration of the Sensor Board stages, the first stage outputs the same distance measurement reported during stage three.

The second stage of the Sensor Board communications occurs on the next loop of the MITM's code. First, the MITM requests the distance value reported by the tinyLiDARs under the 0x54 command in the first stage. Second, the MITM sends a 0x56 command. This command has the Sensor Board trigger its tinyLiDARs, convert the ultrasonic echo time to a distance measurement, and determine the minimum distance between all three of its sensors.

For the third stage the MITM, requests the minimum distance value found using the 0x56 command in the second stage.

The distance values reported during the second and third stages are error-checked and filtered by the MITM. If the distance values are 0xFFFF or 0BBBB, then the MITM replaces the value with the last known distance value and changes the sensor identifier to *X* or *E*, respectively. A third identifier, *R*, is used when the Sensor Board communications are in the first stage after the first iteration of communications. After error-checking, the MITM passes the distance value through a 10-point SMA filter to reduce measurement noise. Also, for each loop of its code, the MITM logs the Sensor Board distance values in a CSV file.

6.5.3.3. Feedback Control

The feedback control of the MITM uses the distance measurements reported by the Sensor Boards and the channel values from the RX SBUS frame to determine if the controller needs to adjust channel values to avoid possible collisions. When the LCAS is enabled the MITM uses the Sensor Board identifiers from the initialization of the Sensor Boards to determine which direction the feedback controller needs to be working in.

To start off, the controller checks if the user is not actively avoiding the detected obstacle and if the distance measured is within the activation window for the controller. For example, the controller will only start modifying channel values if the distance measured is within 250 mm of the minimum distance. If it is in the activation window, the controller finds the difference between the measured distance and the desired minimum distance. The value is passed into the Phase II controller, Equation (52). Next the controller output, f_c , is converted into a SBUS value using the following equation:

$$SBUS_{new} = f_c(1811 - 992) + 992 \quad (64)$$

The new SBUS value then replaces the channel value that is associated with the direction in which the LCAS is attempting to avoid an obstacle.

As of this writing the only feedback controller designed is for the forward direction. Therefore, the controller is modifying the *Ele* channel (see Table IV) when it is greater than SBUS neutral.

6.5.4. Raspberry Pi with GPS & Accelerometer Version

A GPS and Accelerometer (GPSA) version of the MITM was developed for use in modeling the Canary quadcopter. The Pi remained as the main processor of the MITM and still conducted SBUS communications in the same manner, albeit with a few modifications. The

Sensor Board communications and feedback controller were disabled, and replaced with the logging of position and acceleration data from a GPS module and an accelerometer, respectively.

6.5.4.1. Modification to SBUS Communications & Logging

The GPSA version of the MITM maintained the same procedure for receiving and transmitting SBUS signals that was used in other versions of the MITM. The only changes made were how the MITM labeled and handled certain channel values. The *sbEN* channel was renamed to *vEN* and used to trigger a step in the forward direction of motion. Another channel, *VEL*, was added that controlled the magnitude of the step. For more details on the channel names refer to Table IV.

When the *vEN* was triggered, the MITM would use the value from *VEL* to replace the *Ele* channel and set the magnitude of the step. However, *Ele* can only range from SBUS neutral to SBUS maximum values when the Canary is in forward motion. Therefore, the value of *VEL* had to be scaled to be in the same range, using the following linear regression:

$$Ele' = \frac{1}{2}VEL + 986 \quad (65)$$

where *Ele'* is the new value of *Ele*.

The procedure for logging the RX and TX SBUS frames was only modified to change the name of *sbEN* to *vEN* and to add *VEL*.

6.5.4.2. GPS Control & Logging

To record the GPS position (latitude, longitude, and altitude) of the Canary, the GPSA made use of a 3D Robotics uBlox GPS with Compass module. Using the `gps3` Python library, the MITM could communicate with the module over UART. Since the MITM's built-in UART bus was being used for SBUS communications, a USB-to-TTL adapter was used to create the GPS module's UART bus. Figure 67 shows the GPS module and its connection to the MITM.



Figure 67: 3DR GPS module & its connection to the MITM

The capturing of GPS data was done using the built-in functions of the `gps3` library. Any time the GPS module had a new data packet available it would send the packet over the UART connection to the MITM, where it would be stored in a buffer. When logging was enabled, the MITM would check the buffer for a new packet each time after transmitting an SBUS frame. If there was a new packet available, the MITM parsed the packet for latitude, longitude, and altitude data. The data were then logged in a CSV file.

6.5.4.3. Accelerometer Control & Logging

The second sensor used in the GPSA version of the MITM was a BMA280 accelerometer from Bosch Sensortec. The BMA280 is a triaxial, low-g acceleration sensor controlled over I2C [61]. This accelerometer was chosen since it was readily available. The BMA280 and its interfacing board are shown in Figure 68.

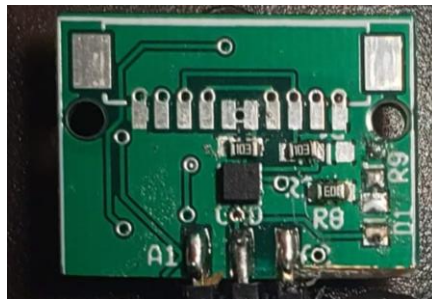


Figure 68: BMA280 accelerometer & interfacing board

Initialization of the sensor involved setting the measurement sensitivity at 2g and the measurement bandwidth at 62.5 Hz—approximately a 125-Hz sample rate. Next compensation values are calculated using the BMA280’s built-in “fast compensation” method. The method captures 16 consecutive acceleration values and uses the average to find the offset from the target value [61]; this was done for the x-, y-, and z-axes, with target values of 0g, 0g, and +1g, respectively. The calculated offsets are then stored by the sensor in its registers, which are read and logged by the MITM.

Much like the logging of GPS values, the MITM requested and logged acceleration values from the BMA280 in a CSV file after transmitting a SBUS frame. Since accelerometers are known to record noisy measurements [46], the MITM used a 10-point SMA filter on each axis to improve sample accuracy.

6.5.5. PCB Design

In order to keep the MITM modular, a PCB was designed. Known as the Docking Board, the MITM’s PCB allows multiple Sensor Boards to connect to the Pi’s I2C bus on the GPIO pins, along with containing RX and TX SBUS signal inverters. An additional, but unrealized, feature of the board is the inclusion of an inertial measurement unit (IMU).

The Docking Board measures 65 mm by 30 mm and contains the following components listed in Table XII. Figure 69 shows the Docking Board with the major components labelled, in accordance with Table XII.

Table XIII: LCAS MITM Docking Board components

Label	Component	Amount
U1	40-pin female pinheader	1
U2	BMX055 IMU	1
U3	2N7002 MOSFET	2
U4	10-k Ω resistor	2
U5	0- Ω resistor	2
U6	4.7-k Ω resistor	2
U7	3-pin pinheader	2
U8	4-pin Grove female connectors	1-10

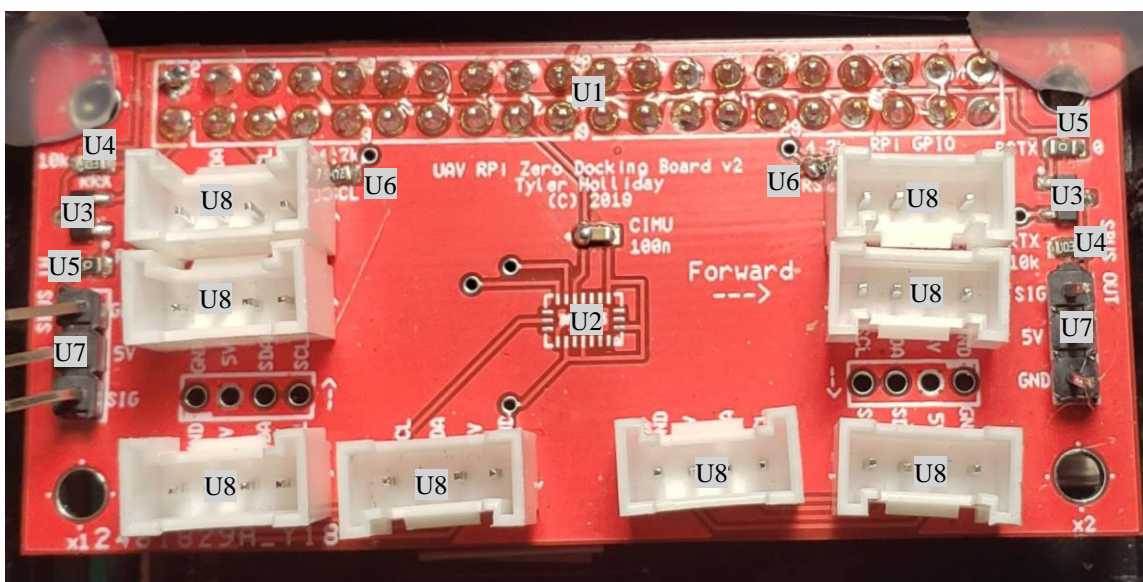


Figure 69: LCAS MITM Docking Board

The PCB layout of the MITM Docking Board can be found in Appendix B.

6.5.5.1. I2C Bus

The Docking Board's I2C bus uses a maximum of 10 Grove 4-pin female connectors to connect the Sensor Boards and other I2C devices, such as the BMA280 accelerometer, to the SDA (GPIO 2) and SCL (GPIO 3) pins shown in Figure 66. The Grove connectors were chosen for the compact size and reliable connection. To ensure that the voltage levels of the SDA and SCL signals were 3.3 V, a 4.7-k Ω pullup resistor was used on each line. Figure 70 shows the schematic for the Docking Board's I2C bus.

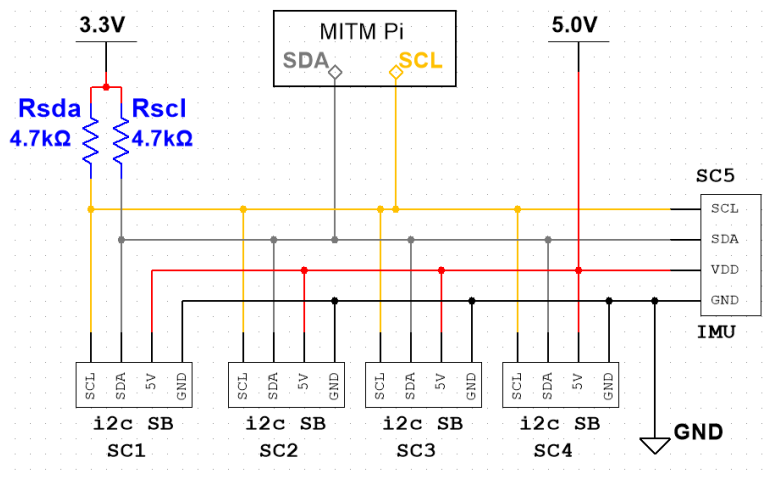


Figure 70: MITM Docking Board I2C bus schematic

6.5.5.2. SBUS Input, Output, & Inverters

The Docking Board's SBUS communication bus uses a pair of SBUS signal inverters (Figure 14) to process RX and TX SBUS signals. The respective inverters are connected to the UART pins of the Pi's GPIO (Figure 66). Figure 71 shows a schematic of the Docking Board's SBUS communication bus.

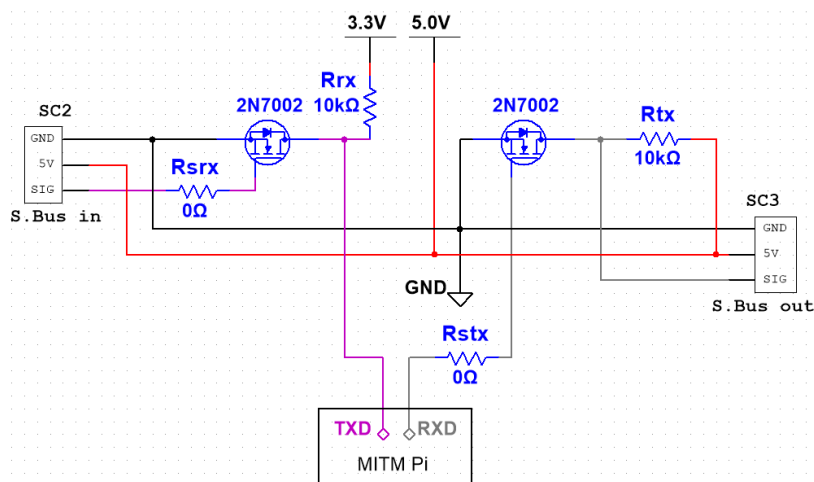


Figure 71: MITM Docking Board SBUS communication bus schematic

6.5.5.3. IMU

The MITM Docking Board's IMU is a BMX055 module from Bosch Sensortec. The module is a compact sensor that features a gyroscope, accelerometer, and magnetometer. By combining the three sensors, the BMX055 detects motion on nine axes. The BMX055 was chosen for its small size and accurate measurements. However, as of the writing of this document, the IMU is not implemented on the MITM.

6.5.5.4. Raspberry Pi Interface

The MITM Docking Board connects to the Pi by the 40-pin GPIO pins. This connects the I2C and SBUS communication buses to the respective pins on the Pi, and provides the Pi power from the flight controller or an external power source. Figure 72 shows the full schematic of the MITM Docking Board, including the Pi interface.

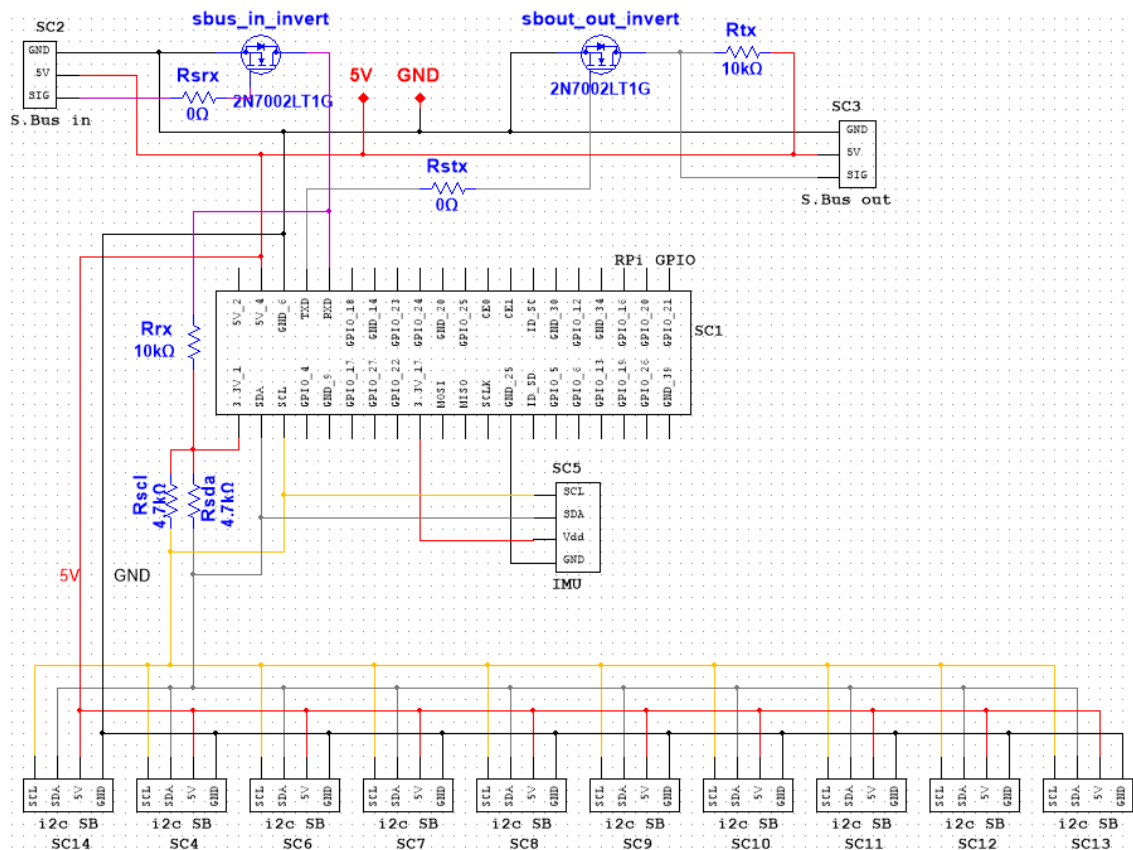


Figure 72: LCAS MITM Docking Board schematic

6.6. Final Prototype Design/Layout

Due to the small size of the Canary quadcopter platform, the layout and mounting of the LCAS was a challenge. With the top of the Canary's frame occupied by the flight controller, and the underside needed for mounting the battery, a custom mounting plate had to be designed to hold the components of the LCAS. Four 35-mm standoffs were used to raise the custom plate above the top of the Canary's frame and out of the way of the Canary's propellers. The custom plate was designed using the computer-aided design (CAD) software Autodesk Inventor. The CAD model of the LCAS-Canary mounting plate is shown below in Figure 73. For a technical drawing of the custom plate refer to Appendix C.

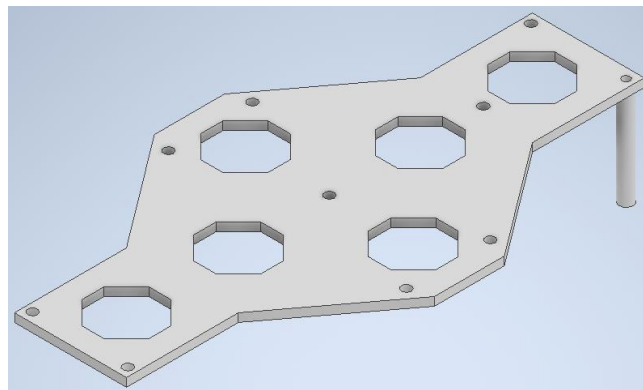


Figure 73: LCAS-Canary mounting plate

An additional mounting bracket was designed for the Sensor Boards. This bracket was designed to be removeable from the mounting plate and interchangeable between any of the Sensor Boards. Figure 74 shows the CAD model of the LCAS Sensor Board mounting bracket. For a technical drawing of the bracket refer to Appendix C.

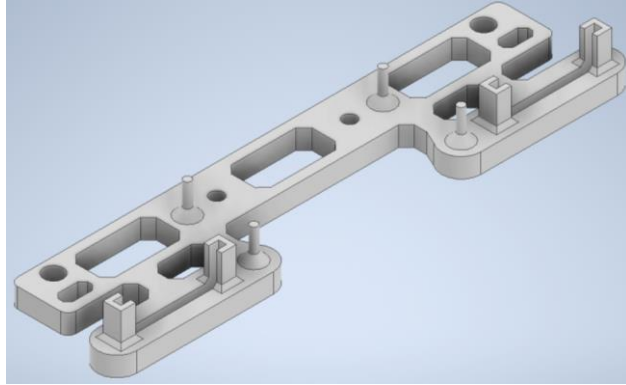
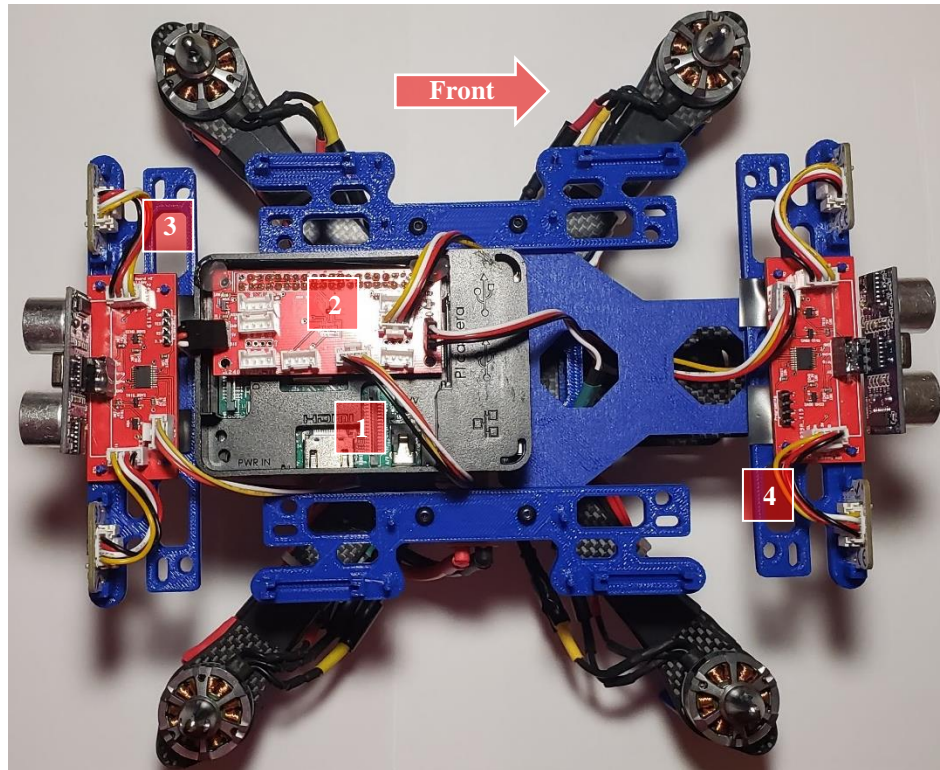


Figure 74: LCAS Sensor Board mounting bracket

The Sensor Board, with accompanying ultrasonic, was placed centrally with the tinyLiDARs flanking on either side. This arrangement ensured that all three sensors would be evenly spaced and centered on the axis that the Sensor Board would be operating on.

The final prototype layout of the LCAS on the Canary resulted in an off-centered MITM, allowing for the mounting space for other non-Sensor Board sensors, such as the GPS module and accelerometer from the GPSA version. The Sensor Boards' and respective mounting brackets were arranged in a plus shape with the front and back Sensor Boards further apart than the left and right boards. This was done to allow space for the MITM and other sensors. As of the writing of this document only four Sensor Board mounting brackets have been incorporated into the layout of the LCAS on the Canary. Figure 75 shows the final prototype layout of the LCAS on the Canary quadcopter platform.



1. Raspberry Pi 3 Model B
2. MITM Docking Board
3. Back Sensor Board
4. Front Sensor Board

Figure 75: LCAS prototype layout on the Canary

7. Prototype Testing

7.1. Methodology

Testing of the prototype LCAS proved to be a significant challenge. There were many variables that could not easily be held constant. For instance, any wind in the environment would cause disturbances in the Canary's flight that would be up to a pilot to correct for. To limit the number of variables in the testing environment a testing methodology was developed.

The first component of the methodology was to use a single Sensor Board in the LCAS. The Sensor Board was implemented to detect and measure distances to obstacles in the forward direction, matching the testing parameters used when deriving the Canary's model (Section 4) and designing the forward feedback controller (Section 5).

The second component of the methodology dealt with the forward direction SBUS input *Ele* (see Table IV). The maximum value of *Ele* was limited to only 1200 to match the input used in the Scenario 1 simulations from the feedback controller design. Furthermore, the 1200 limit was intended to simulate how a pilot would handle the UAV in an indoor environment.

The third methodology component was the setting of the desired minimum distance from obstacles to 500 mm and the controller activation distance to 1000 mm from obstacles. The desired minimum distance was carried over from the feedback controller design. The 1000-mm activation distance was chosen to provide more time for the LCAS to react.

The fourth and final component of the methodology focused on minimizing the impact of environmental variables on the Canary. Minimization of wind was accomplished by choosing a testing sight that was partially shielded from wind gusts. Since it was not possible to prevent all wind disturbances, the Canary was orientated with its forward direction orthogonal to the wind's

direction. To ensure valid distance readings from the Sensor Board the testing perimeter was limited to 4 m from the wall.

The environment used for testing the LCAS prototype was the grassy area between Main Hall and the Museum Building on Montana Tech's campus. A picture of the testing area is shown in Figure 76.



Figure 76: LCAS prototype testing area

7.2. Results

A total of five tests were conducted on the LCAS over the course of one flight using the methodology described in the previous section. The position of and the input to the Canary for the full flight are shown in Figures 77 and 78, respectively.

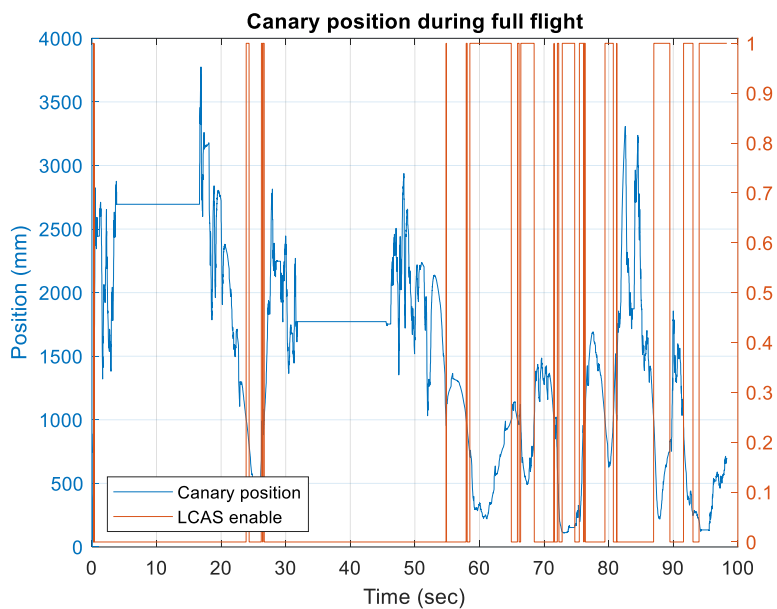


Figure 77: Canary position during LCAS testing flight

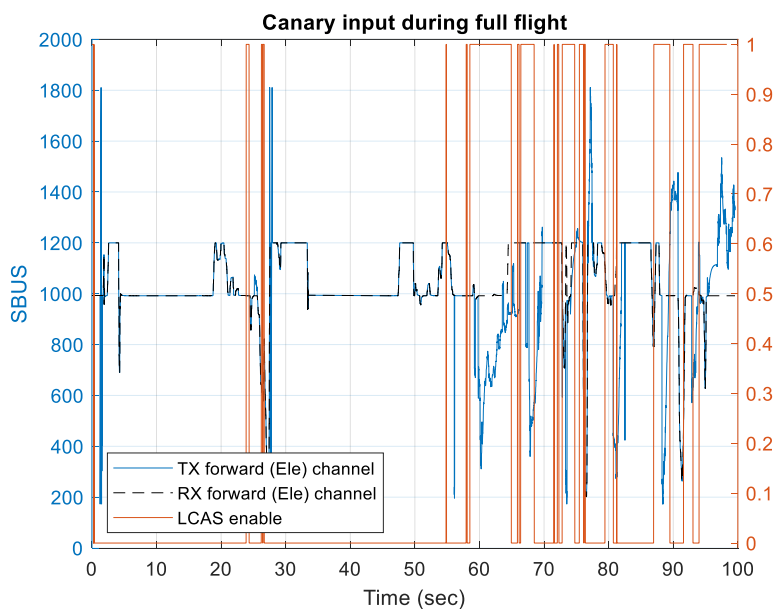


Figure 78: Canary input during LCAS testing flight

To better interpret the data the five individual test results were separated out. The first test results are shown in Figure 79.

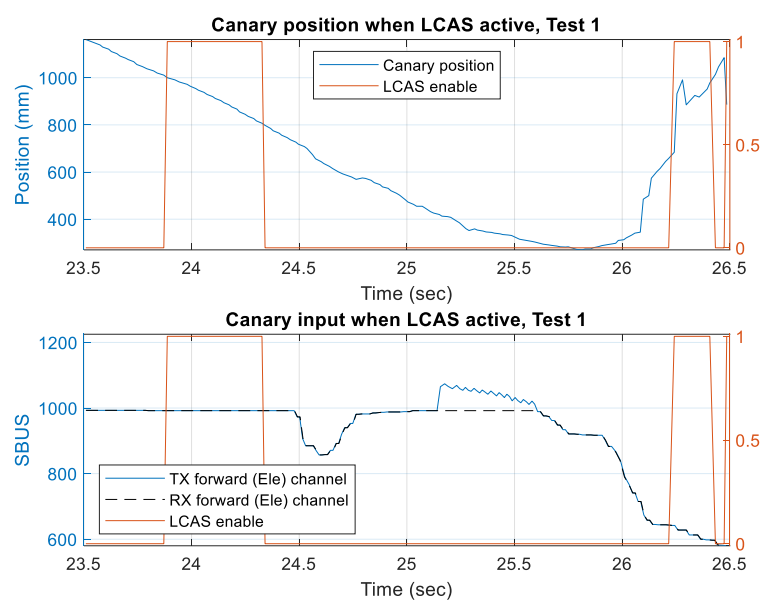


Figure 79: Prototype LCAS Test 1 results

Looking at Figure 79, the LCAS failed to respond to the activation distance and did not modify the *Ele* channel. Instead the pilot prevented the Canary from colliding with the wall.

Test 2 results are shown in Figure 80.

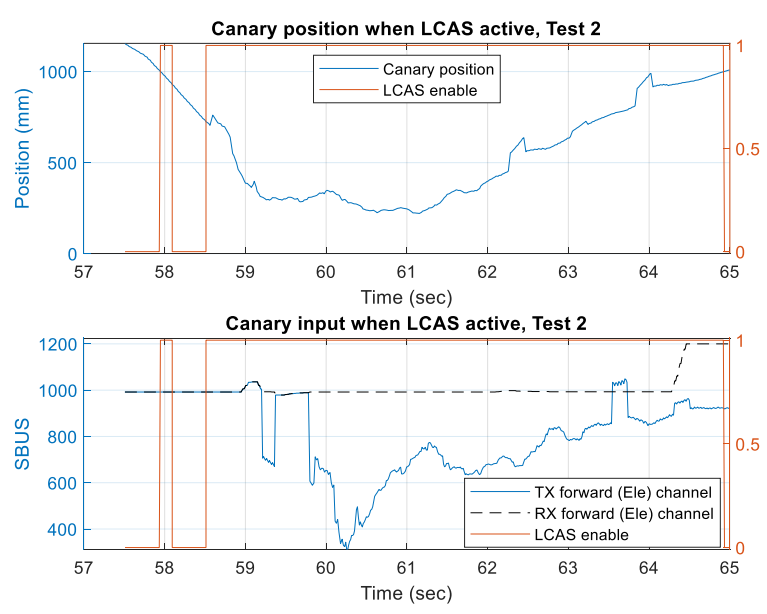


Figure 80: Prototype LCAS Test 2 results

The LCAS displayed a better response during the second test. Even though the pilot was sending the Canary the SBUS neutral value, akin to Scenario 3 from the controller design (Section 5.3.3), the LCAS was able to prevent the Canary from colliding with the wall. However, the LCAS could not get the Canary to maintain the desired minimum distance from the wall.

The LCAS continued to show promise in Test 3. These results can be seen in Figure 81.

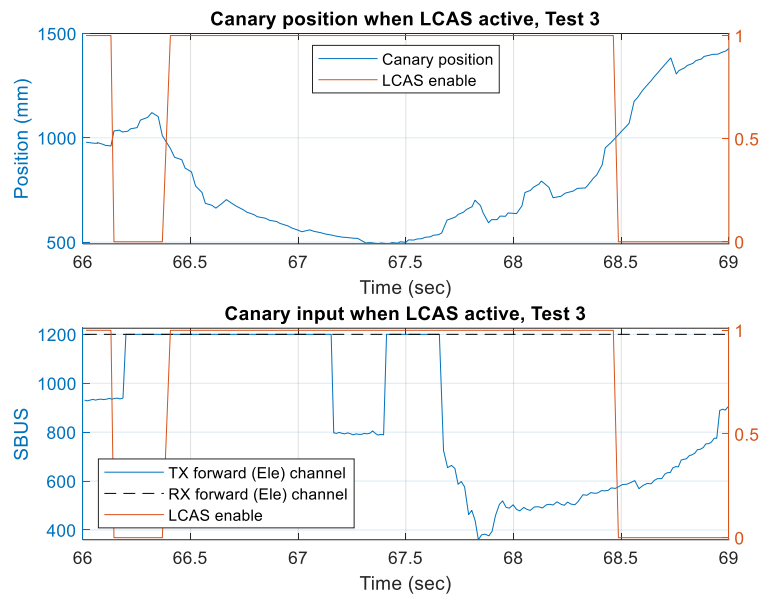


Figure 81: Prototype LCAS Test 3 results

During Test 3, the LCAS operated as intended but did have some delay. While the system did respond to the activation distance, it did not significantly alter the Canary's input until the desired minimum distance was detected. However, despite the delayed response, the LCAS was able to maintain the desired minimum distance.

The results for the fourth and final test are shown in Figure 82.

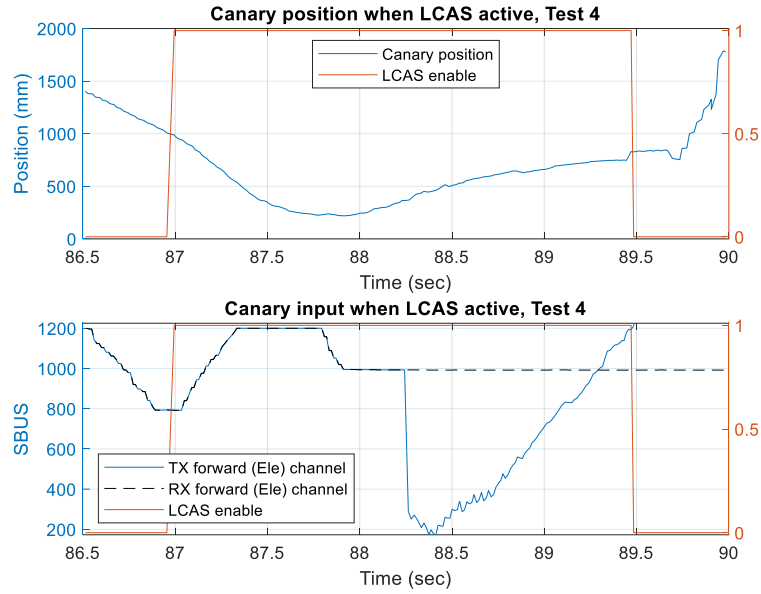


Figure 82: Prototype LCAS Test 4 results

The LCAS's response to Test 4 appeared to combine the responses of Tests 2 and 3. The system was able to prevent the Canary from colliding with the wall but struggled to maintain the desired minimum distance, akin to Test 2. The struggle to maintain the minimum desired distance can be attributed to a delay in the system's operation, a behavior seen in Test 3.

8. Conclusions

A methodology for modeling a UAV in a single direction of motion was developed and validated. The response of the Canary to a series of step inputs in the forward direction was comprised of position and acceleration measurements from a GPS module and an accelerometer, respectively. Using a position estimate aided by a Kalman filter, the Canary's step responses were curve-fitted to produce a model of Canary. This model was then validated by subjecting it to the same step inputs used in the Canary step response testing.

Once validated, the Canary model was used to design the LCAS's feedback controller. The control design was done in two phases. Phase I focused on tuning a PID controller using time domain analysis and a trial-and-error approach. Phase II took a different approach by using the root locus technique to focus on system stability. In simulations the Phase I controller was able to drive the Canary model to the desired position but an oscillatory behavior led to a concern of system stability if the controller was used in the LCAS. Noise resiliency testing proved the validity of this concern. The Phase II controller improved significantly upon the Phase I design in simulations, by reducing oscillations and achieving a steady state at the desired position. Noise resiliency proved to be a challenge for the Phase II controller but the addition of a 10-point SMA filter lessened the effect of noise on the system. Results from the Phase II controller simulations showed promise for the feasibility of the LCAS.

The overall goal of this work was to produce a proof of concept and test the feasibility of a custom, low-cost collision avoidance system that could be implemented on UAVs in indoor environments. A single-direction prototype of the LCAS was developed, tested, and showed promising results as a proof of concept. In addition, the prototype was designed with modularity in mind. The final version of the MITM used in this work was capable of communicating with up

to six Sensor Boards, with the potential for more. As for feasibility, there are two observations from the prototype testing to consider before a final assessment can be made.

The first observation is the inconsistency of the prototyped LCAS. While the LCAS performed admirably during the four reported tests, what was not shown were previous failed tests and flights. These results were not included for a variety of reasons, most notably a failure in logging (more on that later). For example, two additional tests were conducted during the flight that produced results for the five tests. During the first additional test, which occurred after Test 1, the LCAS failed to operate when it did not properly detect the activation distance. The second additional test was made between Tests 3 and 4. The LCAS's response to this test, simply known as Test 3.5, was the best representation of the system's inconsistency. The results of this test are shown in Figure 83.

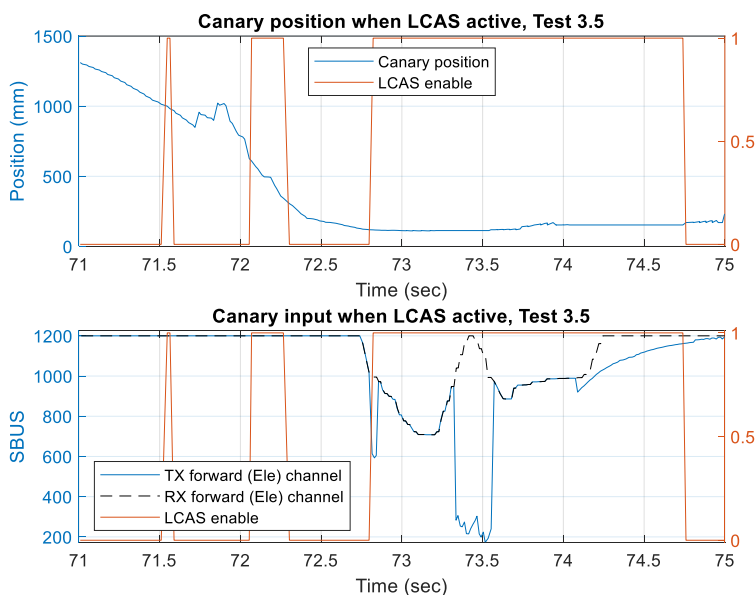


Figure 83: Prototype LCAS Test 3.5 results

Looking at the figure, it appears that the LCAS struggled to remain active once the activation distance was detected. As a result, the Canary failed to maintain the desired minimum

distance from the wall, getting as close as 200 mm. While not shown in the position measurements, the Canary collided with the wall around the 74-second mark. The Canary did not crash but rather ground its propellers against the wall. Looking at the plot of the Canary's input it is seen that the LCAS failed to even alter the input when the Canary was against the wall. The LCAS was only able to operate properly again after the pilot landed the Canary and reset for the next test.

The second observation is the glitchy-ness of the LCAS's programming algorithm. The LCAS was subject to as many as 20 flights over the course of several weeks. The first half of flights served as a means to troubleshoot and tune the LCAS's feedback controller algorithm. For the next three flights the tuned LCAS algorithm showed promise, with the system displaying the ability to prevent the Canary from colliding with the wall. However, when reviewing the logs from those flights it was observed that the Sensor Board was reporting large variations between distance measurements. The variations were attributed to the tinyLiDARs providing nonsense measurements that passed the Sensor Board's error-checking when the distance to the wall was greater than 1000 mm. The nonsensical measurements were dealt with by limiting the testing perimeter to 4 m in the testing methodology. Only the seventh and last flight provided the viable results seen in Section 7.2. The other six flights experienced one type of failure or another. Two of the flights had logging failures, while four others experienced a complete failure of the LCAS controller algorithm. The failure of the controller algorithm is the most notable, because when the controller failed the Canary either collided with the wall or the pilot lost control in the forward and backward directions.

Combining all aspects of this work, the single-direction UAV model, the Phase II controller design, and the prototype LCAS, the feasibility of developing a collision avoidance

system using low-cost range finding sensors looks promising. The Canary model proved to be adequate enough to design a viable feedback controller in the Phase II controller. When implemented in the prototype LCAS, the controller was able to prevent the Canary from colliding with a wall in the forward direction of motion during a couple of the prototype test flights. However, based on the observations from the prototype testing, further refinement is needed before the LCAS can be fully realized.

9. Future Work

A significant limitation of this work was in the methodology used to derive the Canary model. While the methodology produced an adequate and valid model, there proved to be too many variables introduced by the testing environment that inversely impacted the performance of the Canary and the quality of the captured step responses. A suggestion is to use a different testing area that offers better protection from the impact of wind and air density, such as a large indoor space. Furthermore, the number of step inputs used should be increased to provide more data points.

Another future consideration is the development of a Phase III controller. The Phase II controller performed admirably in the prototype testing but its slow response to the activation distance is a concern, especially if the Canary had been allowed to operate at higher speeds, as seen in Test 1. A suggestion for the Phase III controller is to incorporate an improved control law that accounts for the amount of change in distance from sample to sample. Alternatively, an obstacle threat zoning methodology, like that used in [29], could be used to determine the strength of the reaction needed to avoid an obstacle.

Once the modeling methodology is improved, a Phase III controller has been designed, and both validated, the next suggested step would be to start developing models and controllers for the remaining five directions of motion. At that point all six controllers would need to be tested and tuned to work with each other. Since the LCAS is intended to be implemented on a UAV operating in restrictive environments it is important to consider how the controllers will react when coming into conflict with each other. For example, if the UAV is flying in a narrow tunnel and the LCAS is sensing obstacles in both the right and left directions, then the respective

controllers have to compromise on the modification to the relevant SBUS channel so that the UAV does not collide with either wall.

Further software development is needed to improve the reliability of the LCAS programming algorithm. As discussed in the observations from the prototype testing, the LCAS suffered from inconsistencies when it came to measuring distances, determining when to activate the feedback controller, and logging data. The LCAS could greatly benefit from its algorithm being separated into multiple parallel operations. By utilizing parallel operations, the SBUS and Sensor Board communications could be separated, thus eliminating the need for different operating states when sending commands and receiving data from a Sensor Board.

As discussed throughout the development and testing of the LCAS hardware and software, the Sensor Board proved to be the most difficult component to realize. By being constrained to using low-cost, hobbyist-grade range finding sensors, the Sensor Board suffered from measurement inaccuracy and needed multiple stages of error-checking. While the measurement inaccuracy from the ultrasonic was addressed by using a linear regression, the tinyLiDARs' tendency to report nonsensical distance measurements when obstacles were not in range was not able to be properly addressed in software. A suggestion for improving measurement accuracy is to implement the IMU that was incorporated into the design of the MITM Docking Board. The data from the IMU could be used to improve upon the position estimates; a method used in [9].

References

- [1] Hardis Group, "Eyeseer Drone Inventory," 16 February 2017. [Online Video]. Available: https://www.youtube.com/watch?v=Bb7tIr_-r7w&feature=youtu.be. [Accessed 9 June 2020].
- [2] E. A. Russell, "UAV-based geotechnical modeling and mapping of an inaccessible underground site," M.S. thesis, Montana Technological Univ., Butte, MT, USA, 2018.
- [3] R. Becker, "Development of a methodology for the evaluation of UAV-based photogrammetry: Implementation at an underground mine," M.S. thesis, Montana Technological Univ., Butte, MT, USA, 2019.
- [4] H. Pham, S. A. Smolka, S. D. Stoller, D. Phan and J. Yang, "A survey of unmanned aerial vehicle collision avoidance systems," 2015. [Online]. Available: <https://arxiv.org/ftp/arxiv/papers/1508/1508.07723.pdf>. [Accessed 1 April 2020].
- [5] D. Atkinson, "DJI Mavic 2 in depth series - part 2 - aircraft safety," HELIGUY.com, 20 September 2018. [Online]. Available: <https://www.heliguy.com/blog/2018/09/20/dji-mavic-2-in-depth-series-part-2-aircraft-safety/>. [Accessed 23 July 2020].
- [6] R. Santos, "Complete guide for ultrasonic sensor HC-SR04 with Arduino," Random Nerd Tutorials, November 2013. [Online]. Available: <https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/>. [Accessed 29 January 2020].
- [7] F. Corrigan, "12 top collision avoidance drones and obstacle detection explained," DoneZon, 9 March 2020. [Online]. Available: <https://www.dronezon.com/learn-about-drones-quadcopters/top-drones-with-obstacle-detection-collision-avoidance-sensors-explained/>. [Accessed 1 April 2020].

- [8] ELEC Freaks, "Ultrasonic ranging module HC-SR04," [Online]. Available: <https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf>. [Accessed 4 October 2019].
- [9] N. Gageik, P. Benz and S. Montenegro, "Obstacle detection and collision avoidance for a UAV with complementary low-cost sensors," *IEEE Access*, vol. 3, pp. 599-609, 12 May 2015.
- [10] F. Corrigan, "12 top lidar sensors for UAVs, lidar drones and so many great uses," DroneZon, 31 December 2019. [Online]. Available: <https://www.dronezon.com/learn-about-drones-quadcopters/best-lidar-sensors-for-drones-great-uses-for-lidar-sensors/>. [Accessed 4 May 2020].
- [11] J. Pei, "11 myths about LiDAR technology," *Electronic Design*, 7 February 2019. [Online]. Available: <https://www.electronicdesign.com/markets/automotive/article/21807556/11-myths-about-lidar-technology>. [Accessed 4 May 2020].
- [12] D. Kohanbash, "LIDAR (laser scanner) fundamentals," *Robots for Roboticists*, 5 May 2014. [Online]. Available: <http://robotsforroboticists.com/lidar-fundamentals/>. [Accessed 4 May 2020].
- [13] Velodyne Lidar, "Puck," Velodyne Lidar, 2020. [Online]. Available: <https://velodynelidar.com/products/puck/>. [Accessed 1 July 2020].
- [14] air-supplyaerial, "Velodyne LiDAR Puck VLP-16," eBay, 2020. [Online]. Available: https://www.ebay.com/itm/Velodyne-LiDAR-Puck-VLP-16/123792318465?hash=item1cd298e001%3A%3A%7EhwAAOSw5i5arF3K&LH_ItemCondition=3. [Accessed 1 July 2020].

- [15] MicroElectronicDesign, "tinyLiDAR reference manual," 2018. [Online]. Available: <https://www.robotshop.com/media/files/pdf/tinylidar-tof-range-finder-sensor-datasheet.pdf>. [Accessed 4 October 2019].
- [16] MicroElectronicDesign, "tinyLiDAR ToF Range Finder Sensor," RobotShop, 2020. [Online]. Available: <https://www.robotshop.com/en/tinylidar-tof-range-finder-sensor.html>. [Accessed 1 July 2020].
- [17] "Ultrasonic distance sensor - HC-SR04," SparkFun Electronics, 2020. [Online]. Available: <https://www.sparkfun.com/products/15569>. [Accessed 23 July 2020].
- [18] J. Llorens, E. Gil, J. Llop and A. Escola, "Ultrasonic and LIDAR sensors for electronic canopy characterization in vineyards: Advances to improve pesticide application methods," *Sensors*, vol. 11, no. 2, pp. 2177-2194, 15 February 2011.
- [19] F. Corrigan, "DJI Mavic Air 2 review of features, specs and FAQs answered," DroneZon, 17 May 2020. [Online]. Available: <https://www.dronezon.com/drone-reviews/dji-mavic-air-2-review-includes-features-specs-faqs/>. [Accessed 19 May 2020].
- [20] DJI, "Mavic Air 2," April 2020. [Online]. Available: <https://www.dji.com/mavic-air-2/specs>. [Accessed 19 May 2020].
- [21] F. Corrigan, "DJI Mavic 2 Pro and zoom review includes features, specs with FAQs," DroneZon, 20 April 2020. [Online]. Available: <https://www.dronezon.com/drone-reviews/dji-mavic-2-pro-zoom-review-of-features-specifications-with-faqs/>. [Accessed 19 May 2020].
- [22] DJI, "Mavic 2," 23 August 2018. [Online]. Available: <https://www.dji.com/mavic-2>. [Accessed 19 May 2020].

- [23] F. Corrigan, "Skydio 2 drone review including features, specs, follow technology and FAQs," DroneZon, 17 April 2020. [Online]. Available: <https://www.dronezon.com/drone-reviews/skydio-2-drone-review-of-features-specs-faqs/>. [Accessed 20 May 2020].
- [24] Skydio, "Skydio 2," 1 October 2019. [Online]. Available: <https://www.skydio.com/#overview>. [Accessed 20 May 2020].
- [25] Skydio, "How does Skydio 2 work?," 2019. [Online]. Available: <https://support.skydio.com/hc/en-us/articles/360036116834-How-does-Skydio-2-work->. [Accessed 20 May 2020].
- [26] R. He, R. Wei and Q. Zhang, "UAV autonomous collision avoidance approach," *Automatika*, vol. 58, no. 2, pp. 195-204, 9 November 2017.
- [27] X. Yu and Y. Zhang, "Sense and avoid technologies with applications to unmanned aircraft systems: Review and prospects," *Progress in Aerospace*, vol. 74, no. 0376-0421, pp. 152-166, April 2015.
- [28] B. N. Chand, P. Mahalakshmi and V. P. S. Naidu, "Sense and avoid technology in unmanned aerial vehicles: A review," in *International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, Mysuru, India, 15-17 December 2017.
- [29] A. Stulgis, L. Ambroziak and M. Kondratiuk, "Obstacle detection and avoidance system for unmanned multirotors," in *23rd International Conference on Methods & Models in Automation & Robotics (MMAR)*, Miedzyzdroje, Poland, 27-30 August 2018.
- [30] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: Part I," *IEEE Robotics & Automation Magazine*, vol. 13, no. 2, pp. 99-110, June 2006.

- [31] J. Zhang, C. Hu, R. G. Chadha and S. Singh, "Maximum likelihood path planning for fast aerial maneuvers and collision avoidance," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Macau, China, 2019.
- [32] Aptiv, "What is sensor fusion?," 3 March 2020. [Online]. Available: <https://www.aptiv.com/newsroom/article/what-is-sensor-fusion#:~:text=Sensor%20fusion%20is%20the%20ability,strengths%20of%20the%20different%20sensors.> [Accessed 28 July 2020].
- [33] J. Villbrandt, "The quadrotor's coming of age," *Illumin*, vol. XII, no. II, 1 July 2010.
- [34] G. Hoffmann, "Schematic of reaction torques on each motor of a quadrotor aircraft, due to spinning rotors," 15 January 2007. [Online]. Available: https://commons.wikimedia.org/wiki/File:Quadrotor_yaw_torque.png. [Accessed 13 January 2020].
- [35] S. Bouabdallah, P. Murrier and R. Siegwart, "Design and control of an indoor micro quadrotor," in *IEEE International Conference on Robotics and Automation*, New Orleans, LA, USA, 2004.
- [36] M. Faessler, "SBUS protocol," GitHub, 20 March 2018. [Online]. Available: https://github.com/uzh-rpg/rpg_quadrotor_control/wiki/SBUS-Protocol. [Accessed 12 December 2019].
- [37] AutoQad Forum, "S-BUS protocol," [Online]. Available: <http://forum.autoquad.org/download/file.php?id=2319>. [Accessed 12 December 2019].
- [38] U. Gartmann, "Futaba S-BUS controlled by mbed," Arm Limited, 9 March 2012.

- [Online]. Available: <https://os.mbed.com/users/Digixx/notebook/futaba-s-bus-controlled-by-mbed/>. [Accessed 12 December 2019].
- [39] R. E. Kalman, "A new approach to linear filtering," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35-45, March 1960.
- [40] Y. Kim and H. Bang, "Introduction to Kalman filter and its applications," in *Kalman Filter*, London, IntechOpen, 2018, pp. 1-6.
- [41] N. S. Nise, *Control Systems Engineering*, Sixth, Ed., Delhi: John Wiley & Sons Inc., 2011.
- [42] G. M. Hoffman, H. Huang, S. L. Waslander and C. J. Tomlin, "Quadrotor helicopter flight dynamics and control: Theory and control," in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, Hilton Head, South Carolina, 2007.
- [43] W. Dong, G.-Y. Gu, X. Zhu and H. Ding, "Modeling and control of a quadrotor UAV with aerodynamic concepts," *World Academy of Science, Engineering and Technology*, vol. 7, no. 5, pp. 901-906, 2013.
- [44] Google, "Montana Tech," [Online]. Available: <https://www.google.com/maps/place/Montana+Tech/@46.0113454,-112.5556061,216m/data=!3m1!1e3!4m5!3m4!1s0x535b09d196ddb55:0x1920f7f5381536bd!8m2!3d46.0116149!4d112.5569969?hl=en>. [Accessed 6 April 2020].
- [45] whuber, "Understanding terms in length of degree formula?," Stackexchange, 25 October 2013. [Online]. Available: <https://gis.stackexchange.com/questions/75528/understanding-terms-in-length-of-degree-formula>. [Accessed 7 April 2020].
- [46] RF Wireless World, "Advantages of accelerometer | disadvantages of accelerometer,"

2012. [Online]. Available: <https://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-Accelerometer.html>. [Accessed 8 April 2020].
- [47] M. Haghighat, M. Abdel-Mottaleb, and W. Alhalabi, "Discriminant correlation analysis: Real-time," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 9, pp. 1984-1996, September 2016.
- [48] GetFPV, "Matek F722-SE AIO Flight Controller," GetFPV LLC, 2019. [Online]. Available: <https://www.getfpv.com/matek-f722-se-aio-flight-controller.html>. [Accessed 28 January 2020].
- [49] GetFPV, "Spedix ES30 HV 3-6s BLHeLi_S 30A ESC," 2019. [Online]. Available: <https://www.getfpv.com/spedix-es30-hv-3-6s-blheli-s-30a-esc.html>. [Accessed 28 January 2020].
- [50] FrSky, "X8R," 2019. [Online]. Available: <https://www.frsky-rc.com/product/x8r/>. [Accessed 31 March 2020].
- [51] FrSky, "Taranis Q X7," 2019. [Online]. Available: <https://www.frsky-rc.com/product/taranis-q-x7-2/>. [Accessed 31 March 2020].
- [52] D. Bhatia, "tinyLiDAR: The maker-friendly laser sensor," Indiegogo, 2017. [Online]. Available: <https://www.indiegogo.com/projects/tinylidar-the-maker-friendly-laser-sensor#/>. [Accessed 5 February 2020].
- [53] STMicroelectronics, "VL53LOX," April 2018. [Online]. Available: <https://www.st.com/resource/en/datasheet/vl53l0x.pdf>. [Accessed 4 February 2020].
- [54] Texas Instruments, "MSP430G2x53, MSP430G2x13 mixed signal microcontroller datasheet," May 2013. [Online]. Available: <http://www.ti.com/lit/ds/symlink/>

- msp430g2553.pdf. [Accessed 4 February 2020].
- [55] Diodes Incorporated, "AP2210," June 2016. [Online]. Available: <https://www.diodes.com/assets/Datasheets/AP2210.pdf>. [Accessed 26 February 2020].
- [56] SparkFun Electronics, "Bi-directional logic level converter hookup guide," [Online]. Available: <https://learn.sparkfun.com/tutorials/bi-directional-logic-level-converter-hookup-guide/all>. [Accessed 26 February 2020].
- [57] Phillips Semiconductors, "Bi-directional level shifter for I2C-bus and other systems," 4 August 1997. [Online]. Available: <http://cdn.sparkfun.com/tutorialimages/BD-LogicLevelConverter/an97055.pdf>. [Accessed 26 February 2020].
- [58] Raspberry Pi Foundation, "Raspberry Pi 3 Model B," 2016. [Online]. Available: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>. [Accessed 23 March 2020].
- [59] J.-L. Aufrance, "Raspberry Pi 3 board is powered by Broadcom BCM2837 Cortex A53 processor, sells for \$35," CNX Software, 29 February 2016. [Online]. Available: <https://www.cnx-software.com/2016/02/29/raspberry-pi-3-board-is-powered-by-broadcom-bcm2827-cortex-a53-processor-sells-for-35/>. [Accessed 19 June 2020].
- [60] Raspberry Pi Foundation, "GPIO," [Online]. Available: <https://www.raspberrypi.org/documentation/usage/gpio/>. [Accessed 23 March 2020].
- [61] Bosch Sensortec, "BMA280," August 2019. [Online]. Available: <https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bma280-ds000.pdf>. [Accessed 30 March 2020].

- [62] SparkFun Electronics, "Serial Communication," SparkFun Electronics, 18 December 2012. [Online]. Available: <https://learn.sparkfun.com/tutorials/serial-communication/all>. [Accessed 11 December 2019].
- [63] SparkFun Electronics, "I2C," SparkFun Electronics, 8 July 2013. [Online]. Available: <https://learn.sparkfun.com/tutorials/i2c>. [Accessed 11 December 2019].

Appendix A: Standard Communication Protocols

UART

Universal Asynchronous Receiver/Transmitter (UART) is a direct connection between two devices that implements a serial connection without the need for a clock signal. Instead UART uses two data lines: transmit (TX) and receive (RX).

A UART signal is generated by raising and lowering voltage levels on a device's TX line. A receiving device will interpret the changing voltage levels in binary, with high voltage being a one and a low voltage being a zero. Figure 84 shows the general structure of a UART signal.

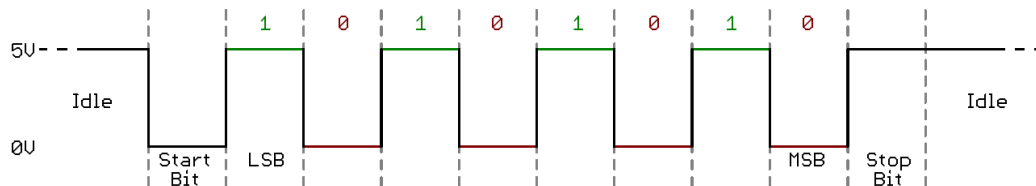


Figure 84: General structure of a UART signal [62]

I2C

The Inter-integrated Circuit (I2C) is a serial communication protocol intended to allow multiple slave devices to connect to a single master device. All devices share the same serial clock (SCL) and serial data (SDA) lines. A basic I2C connection showing the SCL and SDA lines is shown in Figure 85.

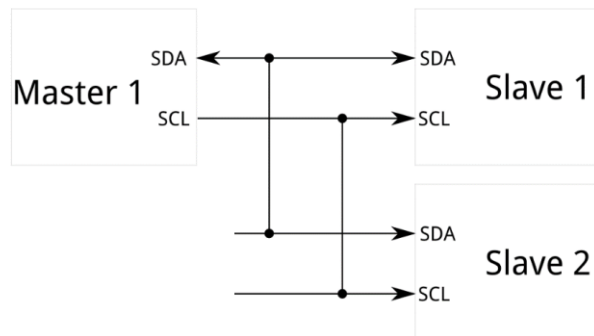


Figure 85: I2C bus adapted from [63]

In order to communicate with a specific slave, the master will call the slave's unique address. Once the slave acknowledges that its address has been called, the master will begin either writing or reading data from the slave. The general structure of an I2C is shown in Figure 86.

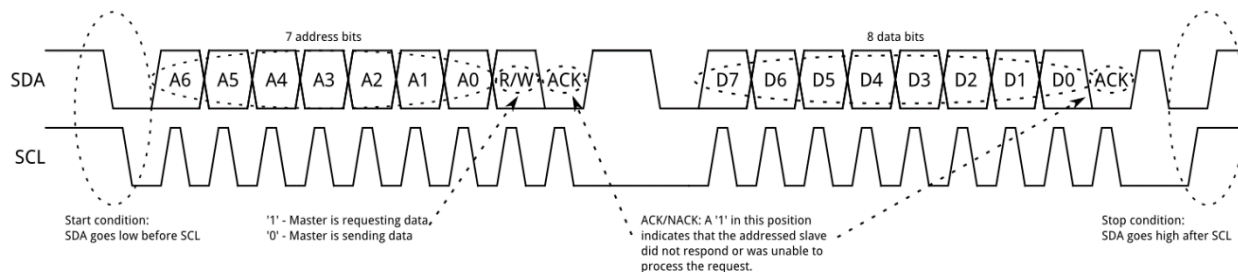
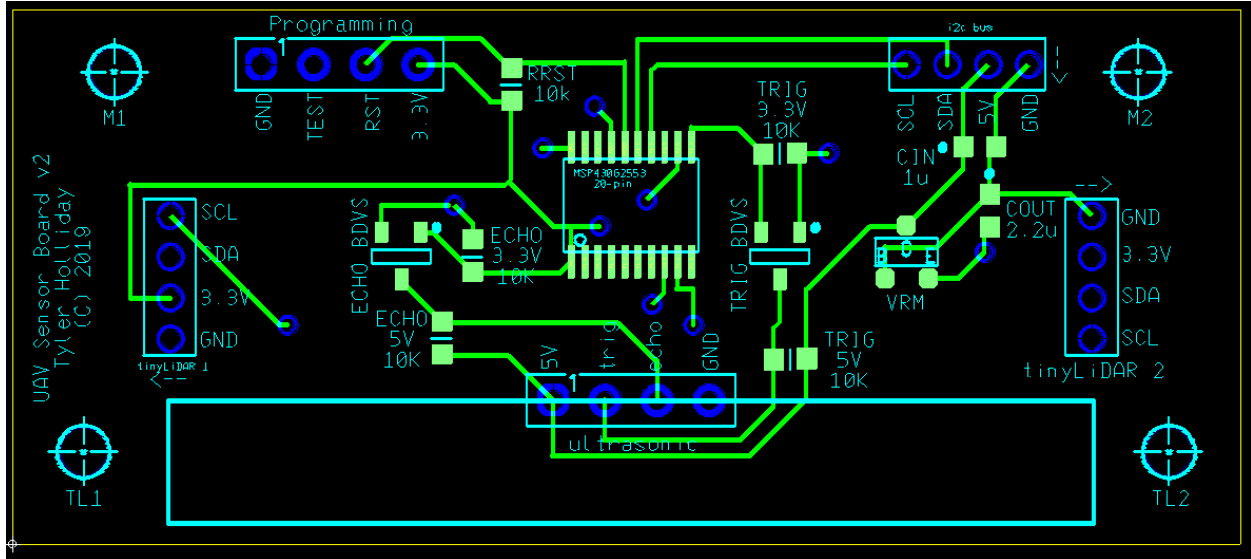


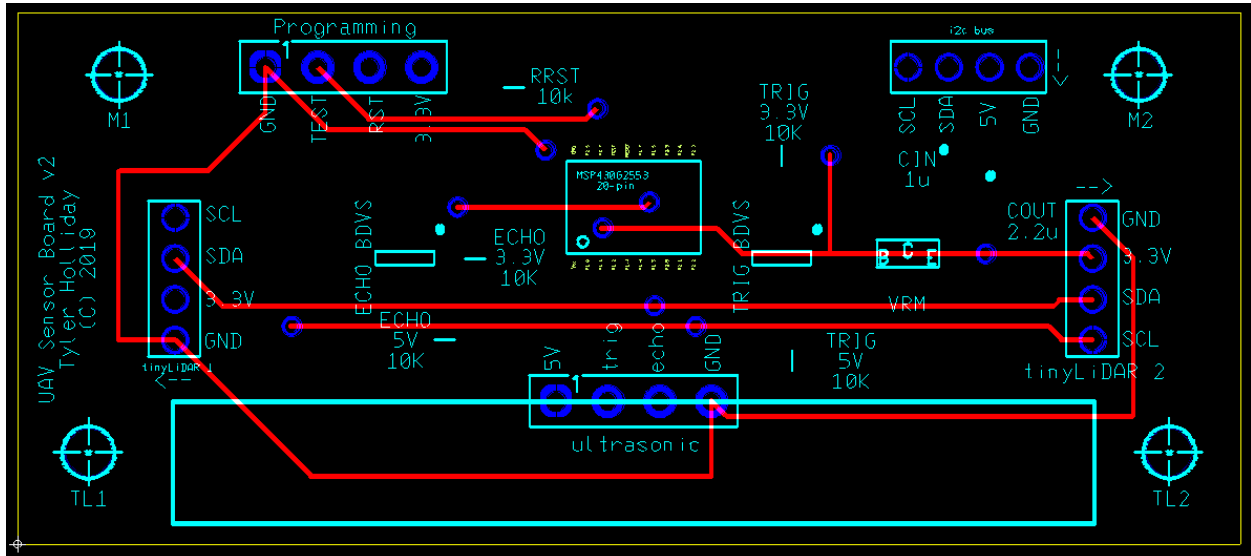
Figure 86: Overview of I2C protocol structure [63]

Appendix B: PCB Layouts

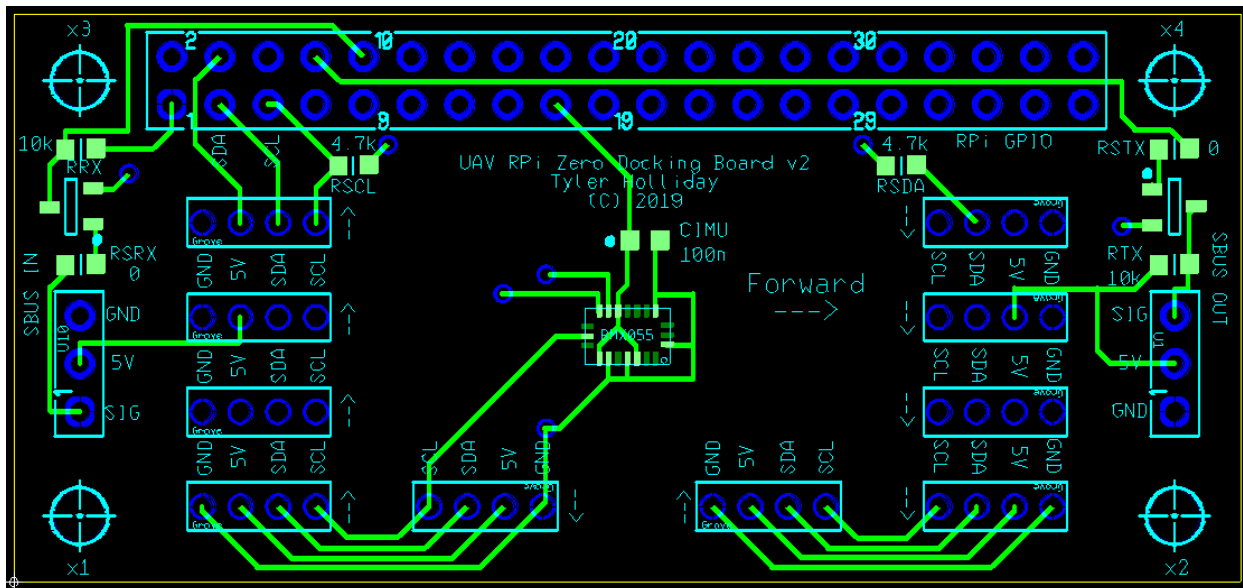
Sensor Board top layer



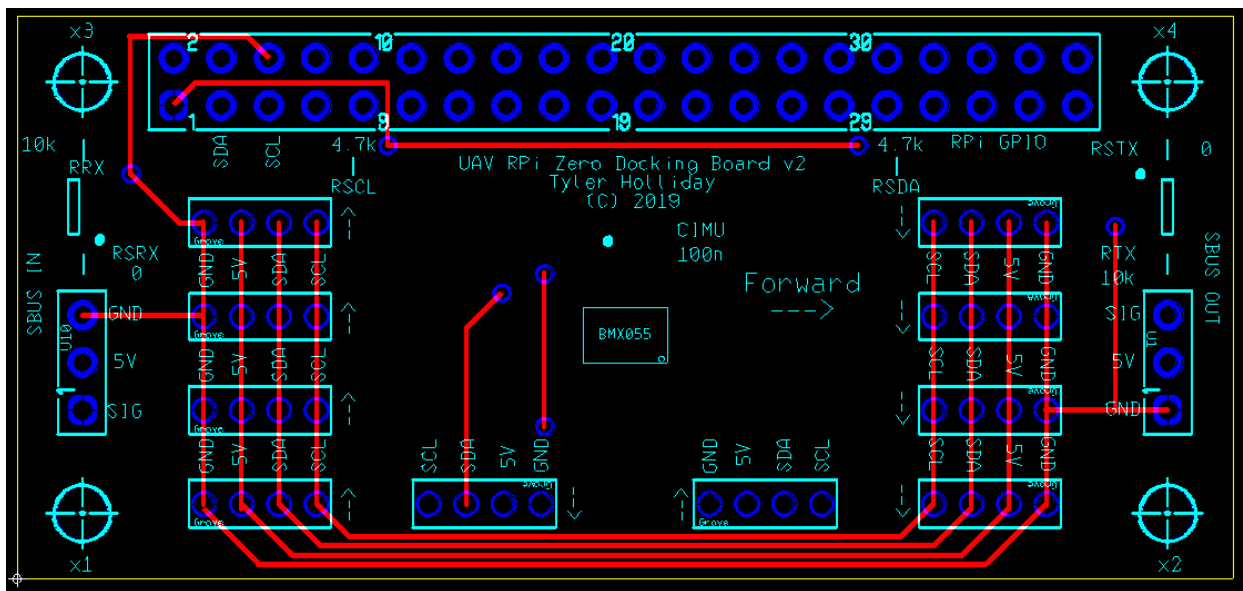
Sensor Board bottom layer



MITM Docking Board top layer

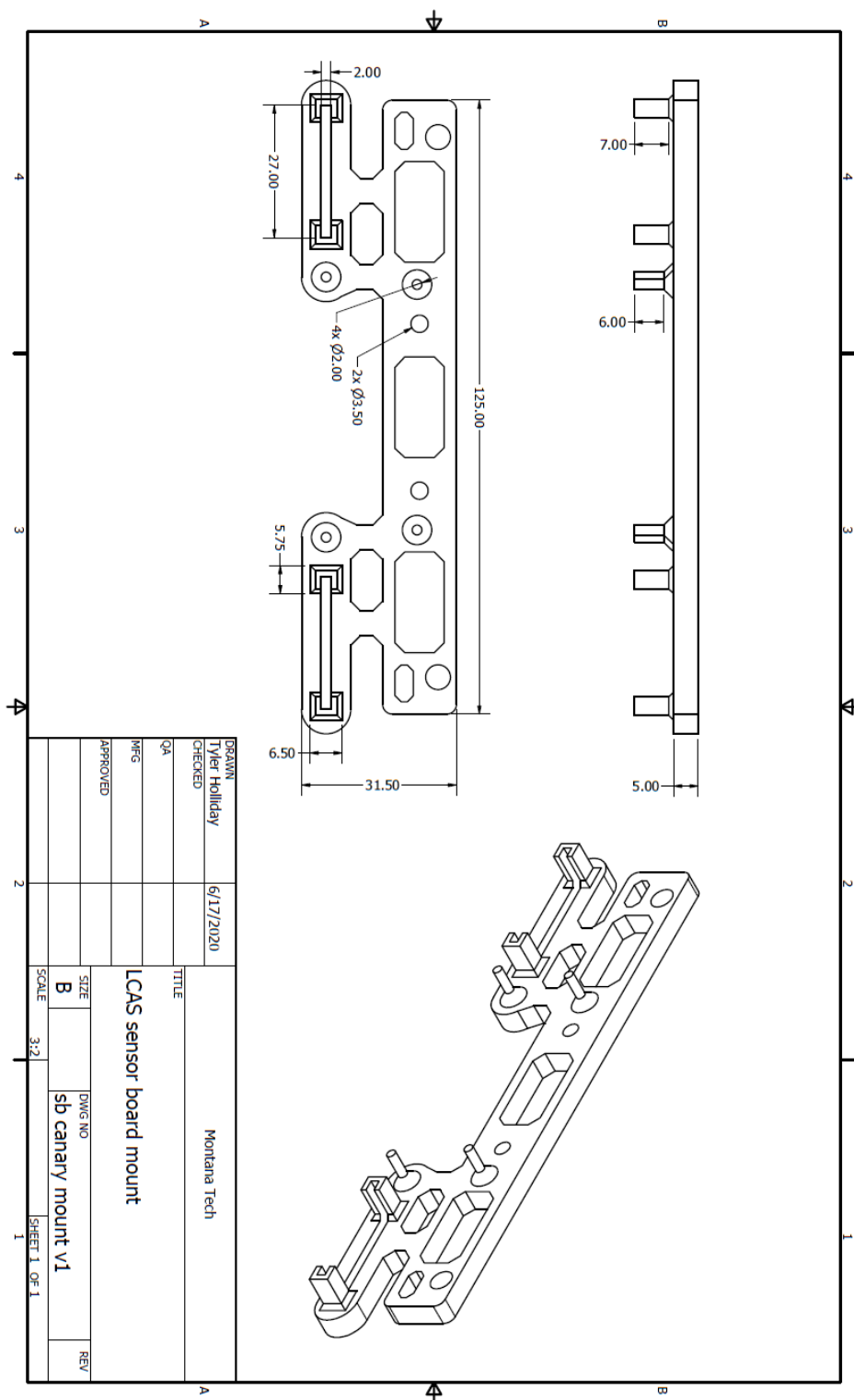


MITM Docking Board bottom layer

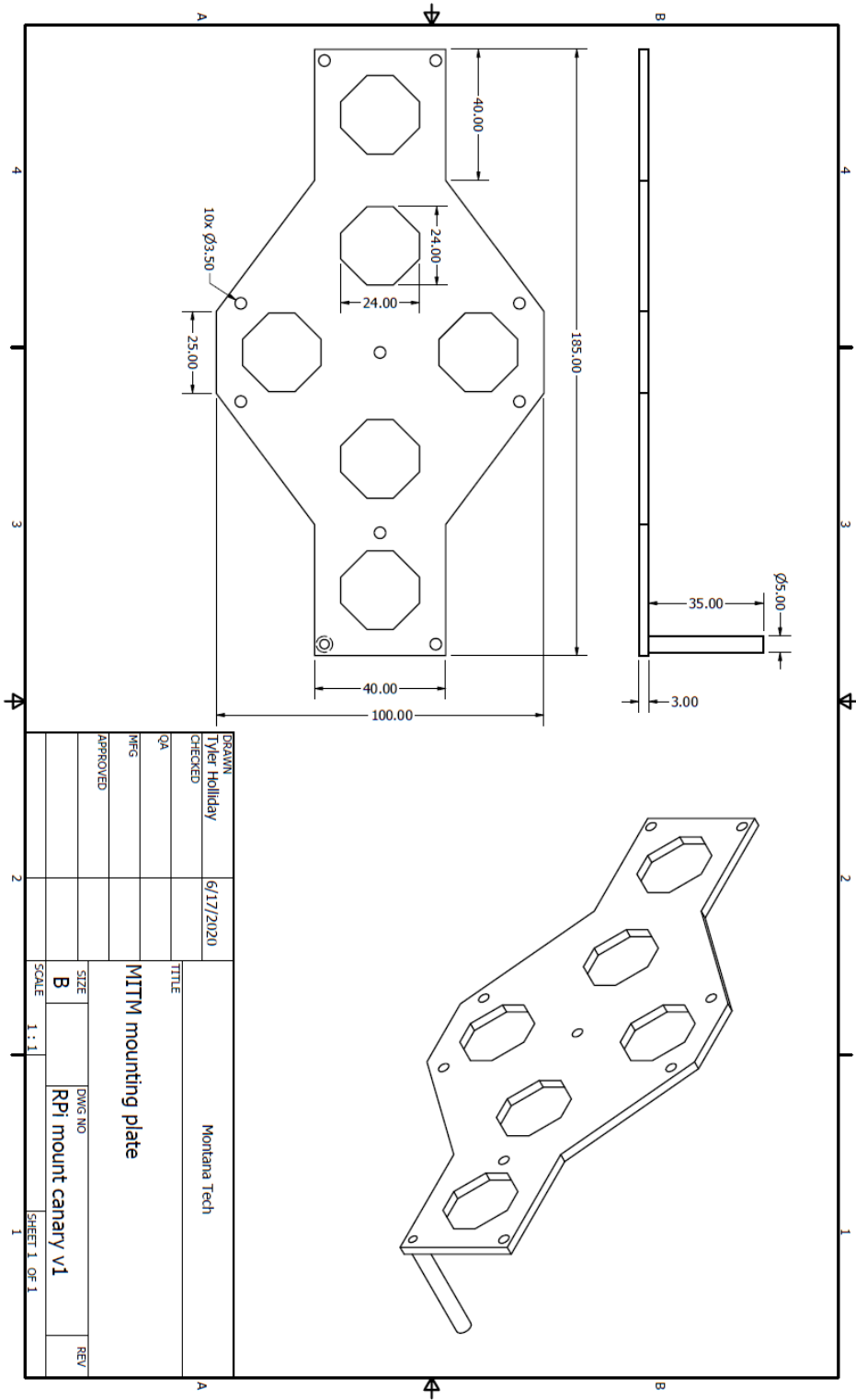


Appendix C: Technical Drawings

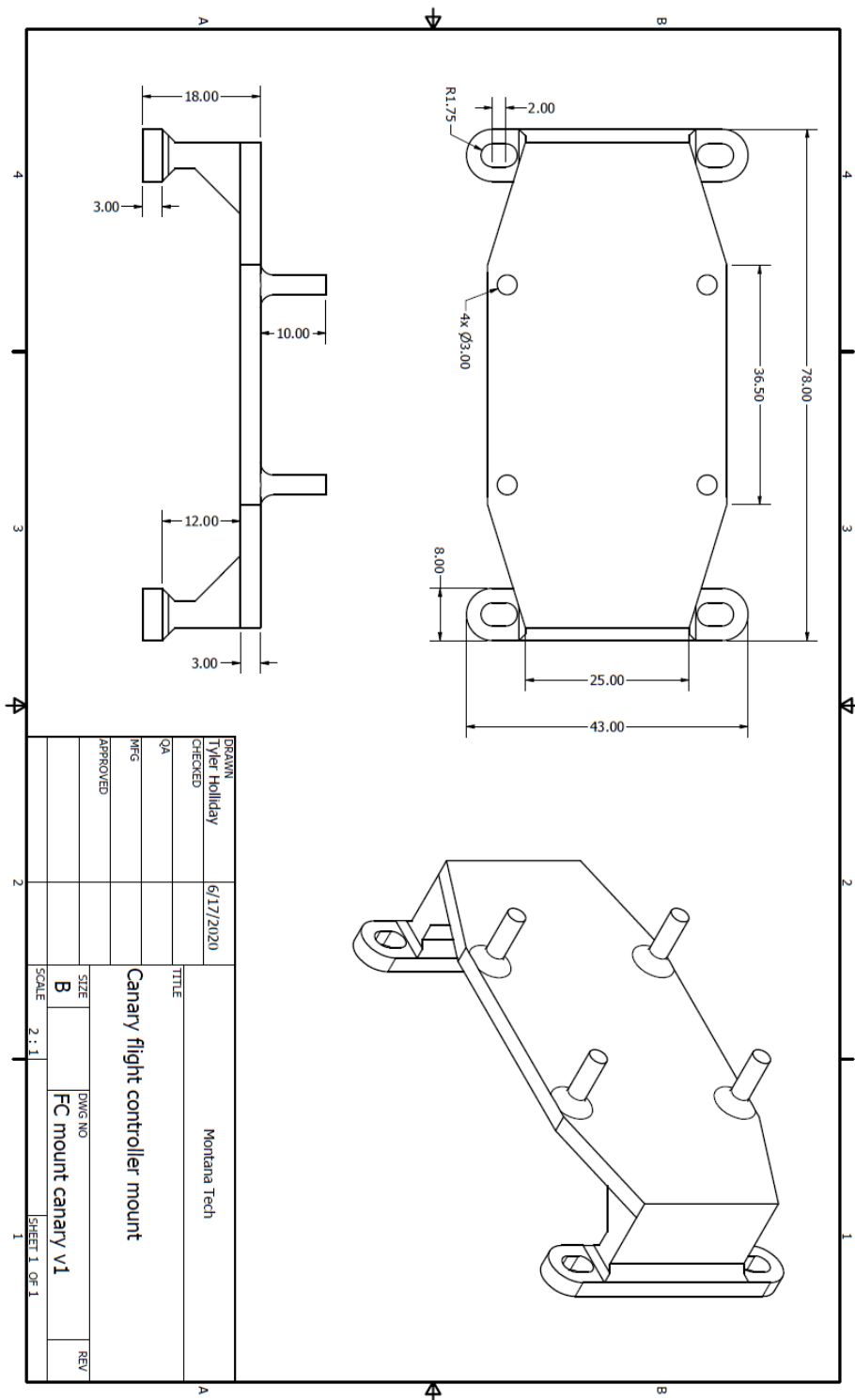
LCAS Sensor Board mounting brackets



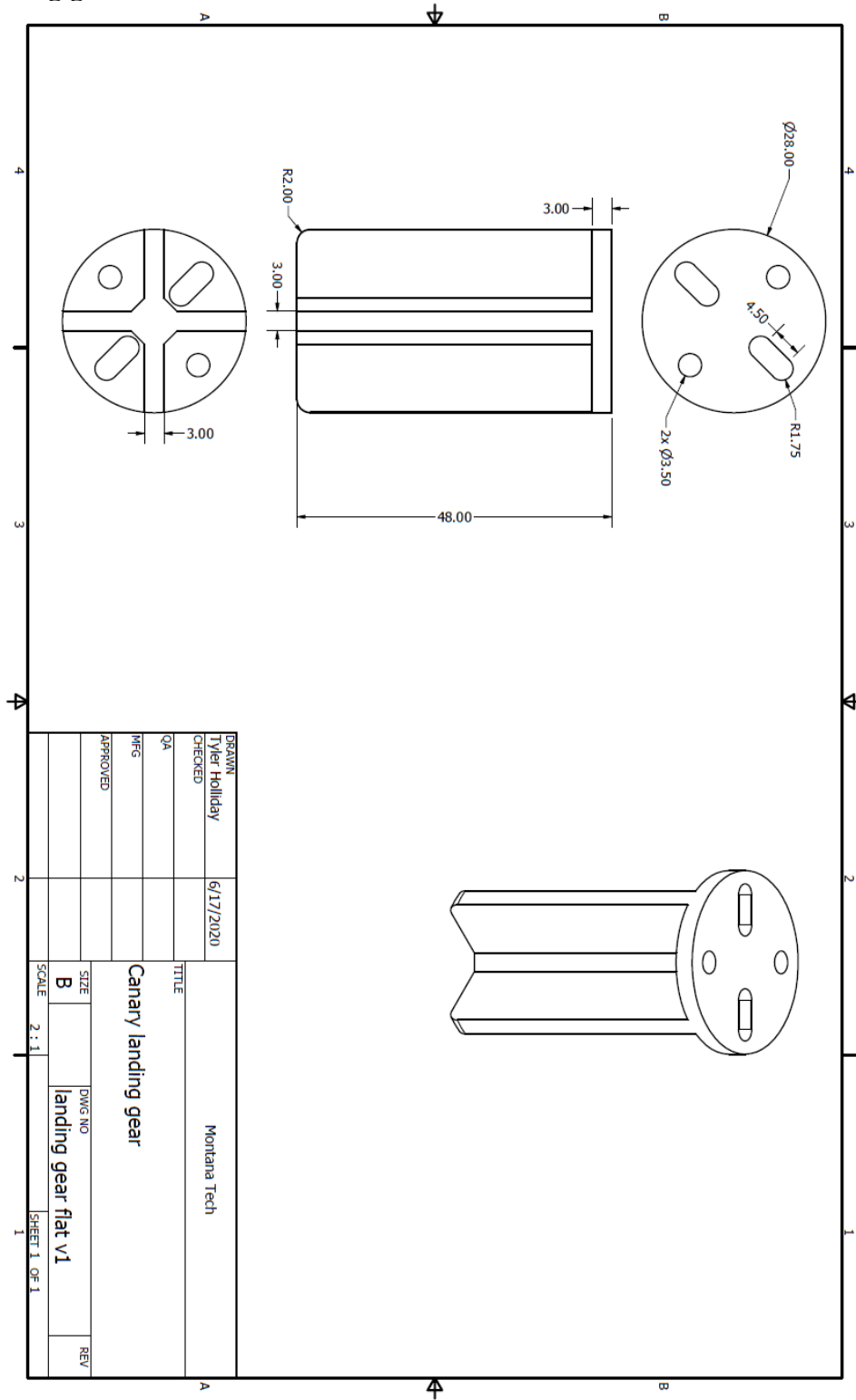
LCAS MITM mounting plate



Canary flight controller mount



Canary landing gear



Appendix D: MATLAB Scripts

Ultrasonic linear regression

```
close all;

y = [50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950 1000];
x = [762 1319 1981 2496 3195 3646 4030 4408 4985 5836 6467 6938 7576 8221 8748 9323
9894 10437 10965 11255];

[r,m,b] = regression(x,y);

dist = m*x + b;

figure
scatter(x,y, '+')
hold on
plot(x,dist)
ylabel('Distance (mm)'); xlabel('Clock cycles');
grid
legend('Raw data','Calculated linear regression','location','southeast')
```

GPS & accelerometer data processing

```
% data_manipulation_v2.m

% Created: Feb 28, 2020
% Modified: Mar 9, 2020
% Author: THolliday

% This script uses logged Sbus (tx), GPS, and Accel data to aid in
% developing the CCAS model.

% NOTE: import data and remove NaN values before running script

close all; format long;

%% Clear Variables
clearvars -except Tsbus vEN Ail Ele ... % imported sbus values
        Tgps lat long alt ... % imported gps values
        Taccel Xaccel Yaccel Zaccel % imported accel values

%% Parameters
saveFlag = 0;
filename = '2020_7_3_F3.mat';
rsFlag = 1; % resampling and Kalman enable
eThres = 1500; % sec, sample error threshold
Ts = 0.02; % sec, sampling period
Fs = 1/Ts; % Hz, sampling period

%% Sbus data
% separate tx values
vEN_tx = vEN(2:2:end); % step enable
Tsbus_tx = Tsbus(2:2:end); % sbus tx time
Ele_tx = Ele(2:2:end); % elevator
Ail_tx = Ail(2:2:end); % aileron
t_sbus_or = Tsbus_tx - Tsbus_tx(1); % set time zero

% check for sample time errors
ndx = find(t_sbus_or >= eThres); % find time values greater than
error threshold
if isempty(ndx) ~= 1
```

```

    err = t_sbus_or(ndx(1)) - t_sbus_or(ndx(1)-1); % calculate time error
    t_sbus_or(ndx) = t_sbus_or(ndx) - err; % remove time error
end

% resampling?
if rsFlag == 1
    [Ele_tx,t_sbus] = resample(Ele_tx,t_sbus_or,Fs);
    [Ail_tx,t_sbus] = resample(Ail_tx,t_sbus_or,Fs);
    [vEN_tx,t_sbus] = resample(vEN_tx,t_sbus_or,Fs);
else
    t_sbus = t_sbus_or;
end

%% GPS data
% adjust time data
t_gps_or = Tgps-Tgps(1); % set time zero

% check for sample time errors
ndx = find(t_gps_or>=eThres); % find time values greater than
error threshold
if isempty(ndx) ~= 1
    err = t_gps_or(ndx(1)) - t_gps_or(ndx(1)-1); % calculate time error
    t_gps_or(ndx) = t_gps_or(ndx) - err; % remove time error
end

% convert to meters
lat_m = 111132.92 - 559.82*cos(2*lat(1)) + 1.175*cos(4*lat(1)) - 0.0023*cos(6*lat(1));
% m/degree
long_m = 111412.84*cos(lat(1)) - 93.5*cos(3*lat(1)) + 0.118*cos(5*lat(1));
% m/degree
y_gps = (lat-lat(1))*lat_m; % meters, first value set to origin
x_gps = (long-long(1))*long_m*-1; % meters, first value set to origin, invert for
proper direction

% resampling?
if rsFlag == 1
    [x_gps,t_gps] = resample(x_gps,t_gps_or,Fs);
    [y_gps,t_gps] = resample(y_gps,t_gps_or,Fs);
else
    t_gps = t_gps_or;
end

%% Accel data
% convert from g's to m/s^2
g = 9.80746; % m/s^2, gravity in Butte, MT
x_accel = Xaccel*g; % x-acceleration
y_accel = Yaccel*g; % y-acceleration
z_accel = (Zaccel*g)-g; % z-acceleration
t_accel_or = Taccel-Taccel(1); % set time zero

% check for sample time errors
ndx = find(t_accel_or>=eThres); % find time values greater
than error threshold
if isempty(ndx) ~= 1
    err = t_accel_or(ndx(1)) - t_accel_or(ndx(1)-1); % calculate time error
    t_accel_or(ndx) = t_accel_or(ndx) - err; % remove time error
end

% resampling?
if rsFlag == 1
    [x_accel,t_accel] = resample(x_accel,t_accel_or,Fs);
    [y_accel,t_accel] = resample(y_accel,t_accel_or,Fs);
    [z_accel,t_accel] = resample(z_accel,t_accel_or,Fs);

```



```

else
    t_accel = t_accel_or;
end

%% Kalman Filter
if rsFlag == 1
    % shorten data
    q = 1e-5; % uncertainty for process covariance
    cutoff = length(t_gps); % length of resampled gps data
    x_accel_Kal = x_accel(1:cutoff); % shorten x accel data
    y_accel_Kal = y_accel(1:cutoff); % shorten y accel data

    % run Kalman filter
    [x_Kal,y_Kal] = kalman(Ts,x_gps,y_gps,x_accel_Kal,y_accel_Kal,q);

    % plotting
    figure
    plot(x_gps,y_gps,x_Kal,y_Kal)
    xlabel('x-position (m)'); ylabel('y-position (m)');
    title('Position estimates')
    legend('GPS only position','Kalman position','location','northwest')
    grid; axis equal;

    figure
    plot(t_gps,x_Kal,t_gps,y_Kal,t_gps,x_gps,t_gps,y_gps)
    xlabel('Time (sec)'); ylabel('Position (m)');
    % title('Position estimates vs. time');
    legend('Kalman x','Kalman y','GPS x','GPS y','location','northwest'); grid;

    figure
    yyaxis left
    plot(t_gps,x_Kal,t_gps,y_Kal)
    ylabel('Position (m)');

    yyaxis right
    % plot(t_sbus,Ele_tx,t_sbus,vEN_tx);
    plot(t_sbus,vEN_tx);
    ylabel('SBUS');

    xlabel('Time (sec)');
    % title('Kalman position & SBUS vs. Time');
    grid;
    % legend('Kalman x','Kalman y','forward/back (Ele)','enable
(vEN)','location','northwest');
    legend('x','y','step enable','location','northwest','orientation','horizontal');
end

%% Separate out step responses
% find time position of steps
step_ctrl = find(vEN_tx >= 400);

% separate steps
cc = 1;
sct = 0;
for rr = 2:length(step_ctrl)
    t_step(rr-1-sct,cc) = t_gps(step_ctrl(rr-1)); % step time
    x_Kal_step(rr-1-sct,cc) = x_Kal(step_ctrl(rr-1)); % x-position during step
    y_Kal_step(rr-1-sct,cc) = y_Kal(step_ctrl(rr-1)); % y-position during step
    ele_step(rr-1-sct,cc) = floor(Ele_tx(step_ctrl(rr-1))); % sbus forward during step
    if step_ctrl(rr)-step_ctrl(rr-1) ~= 1 % check for new step
        interval
            cc = cc + 1; % increase column count
            sct = rr - 1; % reset row count
        end
    end
end

```

```

end

% normalize
[numR,numC] = size(t_step);
t_step_end = zeros(1,numC);
for kk = 1:numC
    t_step(:,kk) = t_step(:,kk) - t_step(1,kk); % time zero point
    x_Kal_step(:,kk) = x_Kal_step(:,kk) - x_Kal_step(1,kk); % x-position zero point
    y_Kal_step(:,kk) = y_Kal_step(:,kk) - y_Kal_step(1,kk); % y-position zero point
    r_Kal_step(:,kk) = sqrt(x_Kal_step(:,kk).^2 + y_Kal_step(:,kk).^2);
    temp = find(t_step(:,kk)<0); % find end of step
    if isempty(temp) ~= 1
        t_step_end(kk) = temp(1)-1; % step end point
    else
        [t_step_end(kk),b] = size(t_step(:,kk)); % first step end point
    end
end
end

% plotting
pndx = 100+numC*10; % setup subplot index for 1 row, numC columns
figure
for kk = 1:numC
    subplot (pndx+kk)
    % plot(t_step((1:t_step_end(kk)),kk),x_Kal_step((1:t_step_end(kk)),kk),...
    % t_step((1:t_step_end(kk)),kk),y_Kal_step((1:t_step_end(kk)),kk))
    plot(t_step((1:t_step_end(kk)),kk),r_Kal_step((1:t_step_end(kk)),kk))
    grid
    xlabel('Time (sec)'); ylabel('Position (m)');
    title(['Sbus step: ',num2str(ele_step(42,kk))])
end

%% Export Results
if saveFlag
    sbus_step_val = ele_step(42,:);
    save(filename,'t_step','x_Kal_step','y_Kal_step','t_step_end','sbus_step_val',...
        't_sbus','Ele_tx','vEN_tx','x_Kal','y_Kal');
End

```

Kalman filter

```

% kalman.m

% Created: Mar 3, 2020
% Modified: Mar 11, 2020
% Author: THolliday

% This script runs a Kalman filter to create position estimates given GPS &
% accel data.

function [x_Kal,y_Kal] = kalman(Ts,gpsx,gpsy,ax,ay,q)

%% Parameters
fs = 1/Ts; % sampling frequency
N = length(gpsx); % number of data points

%% Build Model
% initialize matrices for Kalman
xhat = zeros(6,N); % estimated state vector
yk = [gpsx,gpsy,ax,ay]'; % output vector
Pk = zeros(6,6);

```

```

x_Kal = zeros(1,N);           % Kalman x-postion
y_Kal = zeros(1,N);           % Kalman y-postion

% build state matrix A
Ak = eye(6);
for ii = 1:4
    Ak(ii,ii+2) = Ts;
end

% build output matrix
Ck = [1,0,0,0,0,0;0,1,0,0,0,0;0,0,0,0,1,0;0,0,0,0,0,1];

% build process noise covariance matrix Q
Q = zeros(6,6);
% q = 1e-8;           % set uncertainty
Q(5,5) = q;           % fill in acceleration uncertainty
Q(6,6) = q;

% build measurement noise covariance matrix R
R = zeros(4,4);
L = 8;                 % number of segments
M = floor(N/L);        % length of window
for ii = 1:length(R)
    S = pwelch(yk(ii,:),M,[],[],fs,'twosided'); % find PSD of each data set
    S_trim = S(50:end-50); % only use "noise" content
    Rac = ifft(S_trim); % inv FFT to find autocorrelation
    R(ii,ii) = abs(Rac(1)); % store magnitude of R(0)
end

%% Run Kalman filter
for k = 2:N
    % Extrapolation
    xhatm = Ak*xhat(:,k-1);
    Pm = Ak*Pk*Ak' + Q;

    % Update
    K = (Pm*Ck')/(Ck*Pm*Ck'+R);
    Pk = (eye(6)-K*Ck)*Pm;
    xhat(:,k) = xhatm+K*(yk(:,k)-Ck*xhatm);

    % separate position data
    x_Kal(k) = xhat(1,k);
    y_Kal(k) = xhat(2,k);
end
end

```

Canary model derivation

```

% lcas_model_derivation_v2.m

% Created: Mar 10, 2020
% Modified: Mar 26, 2020
% Author: THolliday

% This script uses step responses found via GPS, sbus, and accel data to
% derive an equivalent model equation.

clear; close all; format long;
addpath('data');

%% Load Variables
filename = '2020_7_3_F3';

```

```

load(filename);

%% Select Step
time_end = 10;      % sec, response cutoff
sbus_max = 1811;   % maximum value of sbus
sbus_mid = 992;    % mid value of sbus

%% Normalize Input
F = (sbus_step_val-sbus_mid)./(sbus_max-sbus_mid); % input scaled to 0-1

%%
pndx = 100+length(F)*10; % setup subplot index
for ii = 1:length(F)
    %% Build Parameters
    ndx = (t_step(:,ii)<=time_end)&(t_step(:,ii)>0);
    t = t_step(ndx,ii);
    x_step = x_Kal_step(1:length(t),ii); % adjust x-pos vector
    y_step = y_Kal_step(1:length(t),ii); % adjust y-pos vector
    x = sqrt(x_step.^2 + y_step.^2); % find resultant position for accuracy

    %% Find Model
    % no initial values are zero
    A_pva = F(ii)*[ones(size(t)),t,t.^2];
    R_pva = A_pva\x;
    xhat_pva = A_pva*R_pva;

    % initial position is zero
    A_va = F(ii)*[t,t.^2];
    R_va = A_va\x;
    xhat_va = A_va*R_va;

    % initial position & velocity are zero
    A_a = F(ii)*((t.^2));
    R_a = A_a\x;
    xhat_a = A_a*R_a;

    % plotting
    subplot(pndx+ii)
    plot(t,x,t,xhat_a)
    xlabel('Time (sec)'); ylabel('Position (m)');
    title([num2str(sbus_step_val(ii)), ' Step',]); grid;
    legend('actual', 'estimate', 'location', 'northwest')
end

```

Canary model testing & validation

```

% lcas_model_testing_v4.m

% Created: Apr 20, 2020
% Author: THolliday

% This script simulates the LCAS model using captured sbus data and
% compares the results to the actual position found via GPS & accel data.

clear; close all; format long;
addpath('data');

%% Load Variables
filename = '2020_7_3_F1';
% filename = '2020_7_3_F3';
load(filename);

```

```

%% Parameters
R = 1.6270;
% R = 1.2489;
step_ndx = 1;           % step selector
Ts = 0.02;             % sec, sampling period
time_end = 15;        % sec, response cutoff
sbus_max = 1811;      % maximum value of sbus
sbus_mid = 992;       % mid value of sbus

%% Normalize Input
F = (sbus_step_val-sbus_mid)./(sbus_max-sbus_mid); % input scaled to 0-1

%% Simulate Model
pndx = 100+length(F)*10; % setup subplot index
for ii = 1:length(F)
    % build parameters
    ndx = (t_step(:,ii)<=time_end)&(t_step(:,ii)>0);
    t = t_step(ndx,ii);
    x_step = x_Kal_step(1:length(t),ii); % adjust x-pos vector
    y_step = y_Kal_step(1:length(t),ii); % adjust y-pos vector
    x = sqrt(x_step.^2 + y_step.^2); % find resultant position for accuracy

    % simulate model
    xsim = F(ii)*(R*t.^2); % estimate position via system model

    % plotting
    subplot(pndx+ii)
    plot(t,x,t,xsim)
    xlabel('Time (sec)'); ylabel('Position (m)');
    title([num2str(sbus_step_val(ii)), ' Step',]); grid;
    legend('actual','model','location','northwest')
end

```

tinyLiDAR standard deviation

```

% sensor_noise_testing_v1.m

% Created: May 14, 2020
% Author: THolliday

% This script takes a look at the type and distribution of the measurement
% noise in the LCAS sensor board. This info will then be used to tune the
% Phase II controller.

clear; close all;

%% Import Data
load('sb_noise_data.mat');

%% Parameters
Ts = 0.02; % sec, sampling period
Fs = 1/Ts; % Hz, sampling frequency

%% Adjust time vectors
ttl = ttl - ttl(1);
tus = tus - tus(1);

%% Calculate raw std
tl_std = std(tldist);
us_std = std(usdist);

```

```

%% Histograms
% separate noise
% tln = tldist - mean(tldist);
% usn = usdist - mean(usdist);
tln = tldist - 150;
usn = usdist - 150;

figure
histogram(tln)
xlabel('mm off 150'); ylabel('Number of samples');
title('TL noise distribution'); grid;
figure
histogram(usn)
title('US noise distribution'); grid;
xlabel('mm off 150'); ylabel('Number of samples');

%% Resample
[tldistr,ttlr] = resample(tldist,ttl,Fs);
[usdistr,tusr] = resample(usdist,tus,Fs);

%% TL PSD
L = 8; % number of segments
M = floor(length(tldistr)/L); % length of window
[Stl,ftl] = pwelch(tldistr,M,[],[],Fs,'twosided'); % find PSD

figure
plot(ftl,Stl)
xlabel('Frequency (Hz)'); ylabel('Magnitude');
title('PSD of TL distance data'); grid;

%% US PSD
L = 8; % number of segments
M = floor(length(usdistr)/L); % length of window
[Sus,fus] = pwelch(usdistr,M,[],[],Fs,'twosided'); % find PSD

figure
plot(fus,Sus)
xlabel('Frequency (Hz)'); ylabel('Magnitude');
title('PSD of US distance data'); grid;

```

Phase I controller design

```

% lcas_controller_design_v5.m

% Created: Mar 26, 2020
% Modified: July 30, 2020
% Author: THolliday

% This script simulates, in discrete time, the LCAS model with the
% controller designed in the time domain (Phase 1).

% Note:
% The "desired position" is an arbitrary distance away from a theoretical
% wall at 1 meter.

% Scenarios:
% 1) user/stick input is high constant, as if user keeps flying towards
% the desired position
% 2) same values as scenario 1, but the user intervenes to avoid desired
% position and uses a lower input after first reaction
% 3) user/stick input is low constant, as if the drone is drifting
% towards the desired position on its own accord
% 4) same as scenario 3, but the user intervenes to avoid desired

```

```

% position

clear; close all; format long;

%% Parameters
runs = 4; % number of scenarios to run
x_act = 0.25; % m, distance from desired to activate controller
x_des = 0.5; % m, desired position away from obstacle
R = 1.6270; % model coefficient
Ts = 0.02; % sec, sampling period
rng(256); % set rng seed
% nmag = 0; % magnitude of added noise (ideal case)
nmag = 0.0036821; % magnitude of added noise (std of TL measurements)
% nmag = 0.1; % magnitude of added noise (extreme case)

%% Initialize
t = (0:Ts:30)'; % sec, time vector
N = length(t); % number of samples
x = zeros(N,runs); % linear position vector
f = zeros(N,runs); % linear input vector
x_diff = zeros(N,runs); % linear difference vector
cact = zeros(N,runs); % controller enable vector
x_ns = zeros(N,runs); % measurement noise vector

%% Gains
Kp = 1.2; % proportional gain
Ki = 0.1; % integral gain
Kd = 5; % derivative gain

%% Build System TF
s = tf('s'); % create TF variable
sys_c = 2*R/s^2; % build continuous time model
sys_d = c2d(sys_c,Ts); % discretize

% extract coefficients of the TF
[num,den] = tfdata(sys_d);
num = num{1}'; % change to vector
den = den{1}'; % change to vector
n = length(num); % number of coefficients

%% Simulate Controller
for cc = 1:runs % run scenarios
    reactflag = 0;
    for kk = 3:N % iterate through samples

        % ----- user input ----- %
        fusr = scenario_input(cc,x(kk-1,cc),reactflag);

        % ----- control law ----- %
        if (x(kk-1,cc)<(x_des-x_act)) || (fusr<=0)
            % not in activation window or user is correcting
            f(kk,cc) = fusr;
            if fusr <= 0
                reactflag = 1;
            end
        else
            % in activation window and controller has control
            cact(kk,cc) = 1;
            f(kk,cc) = Kp*x_diff(kk-1,cc) + Ki*(x_diff(kk-1,cc)+x_diff(kk-2,cc))...
                + Kd*(x_diff(kk-1,cc)-x_diff(kk-2,cc));
        end

        % ----- saturation check ----- %
        if f(kk,cc) > 1
            f(kk,cc) = 1;
        end
    end
end

```

```

elseif f(kk,cc) < -1
    f(kk,cc) = -1;
end

% ---- estimate response ---- %
x(kk,cc) = sum(-flipud(den(2:end)).*x(kk-n+1:kk-1,cc)) ...
    + sum(flipud(num).*f(kk-n+1:kk,cc));

% ---- find distance error ---- %
x_ns(kk,cc) = x(kk,cc) + nmag*randn(1,1);
x_diff(kk,cc) = x_des - x_ns(kk,cc);

end

% Plotting
figure
subplot 311
plot(t,x(:,cc));
ylabel('Position (m)'); grid;
title('Actual model position')
subplot 312
plot(t,x_ns(:,cc));
ylabel('Position (m)'); grid;
title('Model position as seen by controller');
subplot 313
plot(t,f(:,cc));
ylabel('normalized SBUS'); xlabel('Time (sec)'); grid;
title('Input to the model')
sgtitle(['CCAS forward controller under scenario ',num2str(cc)]);
end

```

Root locus for Phase II controller design

```

% root_locus_design.m

% Created: Apr 2, 2020
% Modified: July 30, 2020
% Author: THolliday

% This script builds the CCAS model equation and inputs it into the Root
% Locus Design Tool, so that the PID controller can be tuned.

clear; close all; format long;

%% Build Plant TF
R = 1.6270;           % plant coefficient
s = tf('s');        % create TF variable
sys_c = 2*R/s^2;     % build continuous time model

%% Build Controller TF
zc = [0,-0.2685];    % zeros
pc = [0,-2.292];     % poles
kc = 2.2435;         % gain
Cs = zpk(zc,pc,kc);  % build continuous time TF

%% Root Locus
rltool(sys_c)        % plant-only (used for designing controller)
% rltool(sys_c,Cs)   % plant + controller

```


Phase II controller design

```

% lcas_pid_controller_sim_v1.m

% Created: Apr 2, 2020
% Modified: July 31, 2020
% Author: THolliday

% This script simulates, in discrete time, the LCAS model and PID
% controller designed via Root Locus (root_locus_design.m).

% Note:
% The "desired position" is an arbitrary distance away from a theoretical
% wall at 1 meter.

% Scenarios:
% 1) user/stick input is a constant 1200, as if user keeps flying
% towards the desired position
% 2) same as scenario 1, but the user reacts to the approach of
% the desired position and uses a lower input after first reaction
% 3) user/stick input is a constant 1000, as if the drone is drifting
% towards the desired position on its own accord
% 4) same as scenario 3, but the user reacts to the approach of
% the desired position

clear; close all; format long;

%% Simulation Parameters
runs = 4; % number of scenarios to run
SMAflag = 1;
x_act = 0.25; % m, activation threshold
x_des = 0.5; % m, desired position
R = 1.6270; % model coefficient
numRead = 10; % SMA window
Ts = 0.02; % sec, sampling period
rng(256); % set rng seed
% nmag = 0; % magnitude of added noise (ideal case)
% nmag = 0.0036821; % magnitude of added noise (std of TL measurements)
nmag = 0.1; % magnitude of added noise (extreme case)

%% Initialize
t = (0:Ts:30)'; % sec, time vector
N = length(t); % number of samples
x = zeros(N,runs); % linear position vector
f = zeros(N,runs); % input vector
fc = zeros(N,runs); % controller output vector
e = zeros(N,runs); % error vector
cact = zeros(N,runs); % controller enable vector
x_ns = zeros(N,runs); % measurement noise vector
runSum = zeros(1,runs); % running sum of SMA
readBuff = zeros(numRead,runs); % buffer for SMA
fcrunSum = zeros(1,runs); % running sum of SMA
fcBuff = zeros(numRead,runs); % buffer for SMA
des = ones(length(t),1)*0.5; % desired position vector

%% Build System TF
s = tf('s'); % create TF variable
Gs = 2*R/s^2; % build continuous time model
Gz = c2d(Gs,Ts); % discretize

% extract coefficients of the TF
[num,den] = tfdata(Gz);
Gnum = num{1}'; % change to vector
Gden = den{1}'; % change to vector

```

```

Gn = length(Gnum); % number of coefficients

%% Build Controller TF
zc = -0.2685; % zeros
pc = -2.292; % poles
% zc = [0,-0.2685]; % zeros
% pc = [0,-2.292]; % poles
% kc = 1.1225; % gain
kc = 2.2435; % gain
Cs = zpk(zc,pc,kc); % build continuous time TF
Cz = c2d(Cs,Ts); % discretize

% extract coefficients
[num,den] = tfdata(Cz);
Cnum = num{1}'; % change to vector
Cden = den{1}'; % change to vector
Cn = length(Cnum); % number of coefficients

%% Simulate
for cc = 1:runs
    reactflag = 0;
    readNdx = 1;
    fcreadNdx = 1;
    for kk = 3:N % iterate through samples
        % ----- user input ----- %
        fusr = scenario_input(cc,x(kk-1,cc),reactflag);

        % ----- control law ----- %
        if (x(kk-1,cc)<(x_des-x_act)) || (fusr<=0)
            % not in activation window or user
            f(kk,cc) = fusr;
            if fusr <= 0 % first reaction?
                reactflag = 1;
            end
        else
            % in activation window and controller has control
            f(kk,cc) = fc(kk-1,cc);
            cact(kk,cc) = 1;
        end

        % ----- saturation check ----- %
        if f(kk,cc) > 1
            f(kk,cc) = 1;
        elseif f(kk,cc) < -1
            f(kk,cc) = -1;
        end

        % ----- estimate response ----- %
        x(kk,cc) = sum(-flipud(Gden(2:end)).*x(kk-Gn+1:kk-1,cc)) ...
            + sum(flipud(Gnum).*f(kk-Gn+1:kk,cc));

        % ----- add measurement noise ----- %
        x_ns(kk,cc) = x(kk,cc) + nmag*randn(1,1);

        % ----- moving average ----- %
        if SMAflag==1
            temp = x_ns(kk,cc);
            temp = temp/numRead;
            runSum(cc) = runSum(cc) + temp;
            temp = readBuff(readNdx,cc);
            temp = temp/numRead;
            runSum(cc) = runSum(cc) - temp;
            readBuff(readNdx,cc) = x_ns(kk,cc);
            x_ns(kk,cc) = runSum(cc);
        end
    end
end

```

```

        if readNdx ~= numRead
            readNdx = readNdx + 1;
        else
            readNdx = 1;
        end
    end

    % ---- find distance error ---- %
    e(kk,cc) = x_des - x_ns(kk,cc);

    % ---- run controller ---- %
    fc(kk,cc) = sum(-flipud(Cden(2:end)).*fc(kk-Cn+1:kk-1,cc))...
        + sum(flipud(Cnum).*e(kk-Cn+1:kk,cc));

end

% convert f back to SBUS
f(:,cc) = f(:,cc)*(1811-992)+992;

% Plotting
figure
subplot 311
plot(t,x(:,cc),t,des);
ylabel('Position (m)'); grid;
title('Actual model position')
subplot 312
plot(t,x_ns(:,cc),t,des);
ylabel('Position (m)'); grid;
title('Model position as seen by controller');
subplot 313
plot(t,f(:,cc))
ylabel('normalized SBUS'); xlabel('Time (sec)'); grid;
title('Input to the model')
sgtitle(['LCAS forward controller under scenario ',num2str(cc)]);
end

%% Save data
if nmag == 0
    x_p2_id = x;
    cact_p2_id = cact;
    f_p2_id = f;
    x_ns_p2_id = x_ns;
    if SMAflag==0
        save('P2_results_ideal.mat','x_p2_id','cact_p2_id','f_p2_id','x_ns_p2_id');
    else
        save('P2_sma_results_ideal.mat','x_p2_id','cact_p2_id','f_p2_id','x_ns_p2_id');
    end
end

elseif nmag==0.0036821
    x_p2_t1 = x;
    cact_p2_t1 = cact;
    f_p2_t1 = f;
    x_ns_p2_t1 = x_ns;
    if SMAflag==0
        save('P2_results_t1std.mat','x_p2_t1','cact_p2_t1','f_p2_t1','x_ns_p2_t1');
    else
        save('P2_sma_results_t1std.mat','x_p2_t1','cact_p2_t1','f_p2_t1','x_ns_p2_t1');
    end
end

elseif nmag==0.1
    x_p2_ext = x;
    cact_p2_ext = cact;
    f_p2_ext = f;
end

```

```

    x_ns_p2_ext = x_ns;
    if SMAflag==0

save('P2_results_extreme.mat','x_p2_ext','cact_p2_ext','f_p2_ext','x_ns_p2_ext');
    else

save('P2_sma_results_extreme.mat','x_p2_ext','cact_p2_ext','f_p2_ext','x_ns_p2_ext');
    end

end

```

Scenario input generation function (used in both Phase I & II testing)

```

function fnorm = scenario_input(scenario,x,reactflag)
% scenario_input.m

% Created: Mar 26, 2020
% Modified: May 14, 2020
% Author: THolliday

% Function to generate the input for the LCAS controller simulation of
% flying towards a wall at 1 meter, starting from 0 meter.

% Inputs:
% scenario    scenario number
% x           distance measure from model
% reactflag   denotes if user has reacted once (only used for Phase 1)

% Outputs:
% fnorm       normalized SBUS value used as input to model

%% Parameters
sbus_mid = 992;
sbus_max = 1811;

%% Scenarios
if scenario==1
    % user/stick input is a constant 1200, as if user keeps flying towards
    % the desired position
    fsbus = 1200;
%     fsbus = sbus_max;
elseif scenario==2
    % same as scenario 1, but the user reacts to the approach of the
    % desired position and uses a lower input after first reaction
    if reactflag == 1 && x<0.45
        fsbus = 1050;
    elseif x >= 0.45
        fsbus = 785;
    else
        fsbus = 1200;
    end
end

elseif scenario==3
    % user/stick input is a constant 1000, as if the drone is drifting
    % towards the desired position on its own accord
    fsbus = 1000;

elseif scenario==4
    % same as scenario 3, but the user reacts to the approach of the
    % desired position and then uses a neutral input after first reaction
    if reactflag == 1 && x<0.45

```

```

        fsbus = 1000;
    elseif x >= 0.45
        fsbus = 825;
    else
        fsbus = 1000;
    end
end

end

%% Normalize SBUS
fnorm = (fsbus-sbus_mid)/(sbus_max-sbus_mid); % sets to within [-1,1] range
end

```

Comparing Phase I & Phase II

```

% ctrl_result_plotting.m

% Created: May 28, 2020
% Modified: July 31, 2020
% Author: THolliday

% Script for plotting CCAS results

clear; close all;

%% Import Results
load('ctrl_case_results_v4.mat')

%% Variables
[rr,cc] = size(x_p1_id);
Ts = 0.02; % sec, sampling period
t = (0:Ts:30)'; % sec, time vector
des = ones(length(t),1)*0.5; % desired position vector

%% Ideal case plotting
for ii = 1:cc
    figure
    plot(t,x_p1_id(:,ii),t,x_p2_id(:,ii),t,des,'k--');
    xlabel('Time (sec)'); ylabel('Position (m)'); grid;
    title(['Simulated LCAS under scenario ',num2str(ii)]);
    legend('Phase I','Phase II','desired position','location','southeast')
end

%% Noise resiliency - TL std case
% Phase I
figure
subplot 211
yyaxis left
hold on
plot(t,x_ns_p1_tl(:,1))
plot(t,x_p1_tl(:,1),'m-')
hold off
ylabel('Position (m)')
yyaxis right
plot(t,cact_p1_tl(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('position seen by controller','model position','controller enable',...
    'location','south','orientation','horizontal')

subplot 212
yyaxis left
plot(t,f_p1_tl(:,1))
ylabel('SBUS value')

```

```

yyaxis right
plot(t,cact_p1_tl(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('model input','controller
enable','location','south','orientation','horizontal')
sgtitle('Phase I noise resiliency (TL std case)')

% Phase II
figure
subplot 211
yyaxis left
hold on
plot(t,x_ns_p2_tl(:,1))
plot(t,x_p2_tl(:,1),'m-')
hold off
ylabel('Position (m)')
yyaxis right
plot(t,cact_p2_tl(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('position seen by controller','model position','controller enable',...
'location','south','orientation','horizontal')

subplot 212
yyaxis left
plot(t,f_p2_tl(:,1))
ylabel('SBUS value')
yyaxis right
plot(t,cact_p2_tl(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('model input','controller
enable','location','south','orientation','horizontal')
sgtitle('Phase II noise resiliency (TL std case)')

%% Noise resiliency - extreme case
% Phase I
figure
subplot 211
yyaxis left
hold on
plot(t,x_ns_p1_ext(:,1))
plot(t,x_p1_ext(:,1),'m-','linewidth',1)
hold off
ylabel('Position (m)')
yyaxis right
plot(t,cact_p1_ext(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('position seen by controller','model position','controller enable',...
'location','south','orientation','horizontal')

subplot 212
yyaxis left
plot(t,f_p1_ext(:,1))
ylabel('SBUS value')
yyaxis right
plot(t,cact_p1_ext(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('model input','controller
enable','location','south','orientation','horizontal')
sgtitle('Phase I noise resiliency (extreme case)')

% Phase II
figure
subplot 211
yyaxis left

```

```

hold on
plot(t,x_ns_p2_ext(:,1))
plot(t,x_p2_ext(:,1),'m-','linewidth',1)
hold off
ylabel('Position (m)')
yyaxis right
plot(t,cact_p2_ext(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('position seen by controller','model position','controller enable',...
       'location','south','orientation','horizontal')

subplot 212
yyaxis left
plot(t,f_p2_ext(:,1))
ylabel('SBUS value')
yyaxis right
plot(t,cact_p2_ext(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('model input','controller
enable','location','south','orientation','horizontal')
sgtitle('Phase II noise resiliency (extreme case)')

```

Phase II SMA results plotting

```

% P2_SMA_results_plotting.m

% Created: June 5, 2020
% Modified: July 31, 2020
% Author: THolliday

% Script for plotting LCAS results

clear; close all;

%% Import Results
load('P2_SMA_results_v2.mat')

%% Variables
[rr,cc] = size(x_p2_id);
Ts = 0.02; % sec, sampling period
t = (0:Ts:30)'; % sec, time vector

%% Case 1 - TL std
% Phase II
figure
subplot 211
yyaxis left
hold on
plot(t,x_ns_p2_tl(:,1))
plot(t,x_p2_tl(:,1),'m-')
hold off
ylabel('Position (m)')
yyaxis right
plot(t,cact_p2_tl(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('position seen by controller','model position','controller enable',...
       'location','south','orientation','horizontal')

subplot 212
yyaxis left
plot(t,f_p2_tl(:,1))
ylabel('SBUS value')

```

```

yyaxis right
plot(t,cact_p2_tl(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('model input','controller
enable','location','south','orientation','horizontal')
sgtitle('Phase II with SMA noise resiliency (TL std case)')

%% Case 2 - extreme
% Phase II
figure
subplot 211
yyaxis left
hold on
plot(t,x_ns_p2_ext(:,1))
plot(t,x_p2_ext(:,1),'m-','linewidth',1)
hold off
ylabel('Position (m)')
yyaxis right
plot(t,cact_p2_ext(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('position seen by controller','model position','controller enable',...
'location','south','orientation','horizontal')

subplot 212
yyaxis left
plot(t,f_p2_ext(:,1))
ylabel('SBUS value')
yyaxis right
plot(t,cact_p2_ext(:,1)); ylim([0 1.05])
xlabel('Time (sec)'); grid;
legend('model input','controller
enable','location','south','orientation','horizontal')
sgtitle('Phase II with SMA noise resiliency (extreme case)')

```

Prototype LCAS testing data processing

```

% canary_lcas_testing_06152020.m

% Created: June 16, 2020
% Author: THolliday

% This script plots the results of the Canary testing with the LCAS.

clear; close all; format long;

%% Import data
% filename = 'canary_lcas_06152020_T2_F1.mat'; manflag = 'F1';
% filename = 'canary_lcas_06152020_T2_F2.mat'; manflag = 'F2'; % barely goes below
1000 mm
% filename = 'canary_lcas_06152020_T2_F3.mat'; manflag = 'F3'; % logging failure
% filename = 'canary_lcas_06152020_T2_F4.mat'; manflag = 'F4'; % logging failure
% filename = 'canary_lcas_06152020_T2_F5.mat'; manflag = 'F5'; % controller
failure?
% filename = 'canary_lcas_06152020_T2_F6.mat'; manflag = 'F6'; % ctrl failure
filename = 'canary_lcas_06152020_T2_F7.mat'; manflag = 'F7';
load(filename);

%% SBUS data
Tsbus_tx = Tsbus(2:2:end); % sbus tx time
Tsbus_rx = Tsbus(1:2:end); % sbus rx time
Ele_tx = Ele(2:2:end); % elevator tx
Ele_rx = Ele(1:2:end); % elevator rx

```



```

ctrl_tx = Ctrl(2:2:end);           % ctrl enable
arm_tx = ARM(2:2:end);           % system arming
tsbus_tx = Tsbustx-Tsbustx(1);    % set zero point for sbus time
tsbus_rx = Tsbusrx-Tsbusrx(1);    % set zero point for sbus time

%% SB data
tsb = Tsb-Tsb(1);                % set zero point for sensor board time

%% Plot full flight
figure
yyaxis left
plot(tsb,Dist)
ylabel('Position (mm)');
yyaxis right
plot(tsb,ctrlEN)
ylim([-0.01 1.01])
xlabel('Time (sec)'); grid;
title('Canary position during full flight');
legend('Canary position','LCAS enable','location','southwest');

figure
yyaxis left
plot(tsbus_tx,Ele_tx,tsbus_rx,Ele_rx,'k--')
ylabel('SBUS')
yyaxis right
plot(tsb,ctrlEN)
ylim([-0.01 1.01])
xlabel('Time (sec)'); grid
title('Canary input during full flight')
legend('TX forward (Ele) channel','RX forward (Ele) channel','LCAS
enable','location','southwest')

%% Plot sections where LCAS was active
if strcmp(manflag,'F1')          % F1
    tndx = [62,68];

elseif strcmp(manflag,'F5')      % F5
    tndx = [12,15];

elseif strcmp(manflag,'F7')      % F7
    tndx = [23.5,26.5;57.5,65;66,69;86.5,90;71,75];
end

[rr,cc] = size(tndx);
for ii = 1:rr
    sbndx = tsb>=tndx(ii,1) & tsb<=tndx(ii,2);
    tsb_lcas = tsb(sbndx);
    ctrlEN_lcas = ctrlEN(sbndx);
    Dist_lcas = Dist(sbndx);
    sbusndx_tx = tsbus_tx>=tndx(ii,1) & tsbus_tx<=tndx(ii,2);
    tsbus_tx_lcas = tsbus_tx(sbusndx_tx);
    Ele_tx_lcas = Ele_tx(sbusndx_tx);
    sbusndx_rx = tsbus_rx>=tndx(ii,1) & tsbus_rx<=tndx(ii,2);
    tsbus_rx_lcas = tsbus_rx(sbusndx_rx);
    Ele_rx_lcas = Ele_rx(sbusndx_rx);

    figure
    subplot 211
    yyaxis left
    plot(tsb_lcas,Dist_lcas)
    ylabel('Position (mm)')
    yyaxis right
    plot(tsb_lcas,ctrlEN_lcas)
    ylim([-0.01 1.01]);
    xlabel('Time (sec)'); grid;

```

```
title(['Canary position when LCAS active, Test 3.5'])% ',num2str(ii)];
legend('Canary position','LCAS enable','location','north');

subplot 212
yyaxis left
plot(tsbus_tx_lcas, Ele_tx_lcas, tsbus_rx_lcas, Ele_rx_lcas, 'k--')
ylabel('SBUS');ylim([min(Ele_tx_lcas) 1225])
yyaxis right
plot(tsb_lcas, ctrlEN_lcas)
ylim([-0.01 1.01]);
xlabel('Time (sec)'); grid
title(['Canary input when LCAS active, Test 3.5'])% ',num2str(ii)])
legend('TX forward (Ele) channel','RX forward (Ele) channel','LCAS
enable','location','southwest')

end
```

Appendix E: Sensor Board Code

Main script

```

/* UAV i2c sensor board control v5
 *
 * main()
 *
 * Created on: Feb 14, 2020
 * Modified:   May 13, 2020
 * Author:    THolliday
 *
 * controls two tinyLiDARs and one ultrasonic based on
 * commands received from an i2c master
 *
 * this board also communicates as an i2c master to the
 * tinyLiDAR slaves
 *
 * Commands
 *
 * sensor board      tinyLiDAR      description
 * -----
 * 0x45, 'E'                set error threshold
 * 0x54, 'T'      0x##, 0x44, 'D'    trigger only tinyLiDARs
 * 0x55, 'U'                trigger ultrasonic
 * 0x56, 'V'      0x##, 0x44, 'D'    capture ultrasonic distance, trigger tinyLiDARs,
 *                                     find min distance between all three sensors
 *
 * Fault codes
 *
 * device            variable            code            description
 * -----
 * ultrasonic      distUS                0xFFFF          global failure or timeout on echo
 *                                     0BBBBB          distance value below error threshold
 * tinyLiDAR         distIR[1]                0xFFFF          i2c communication failure
 *                                     0xEEEE          invalid command or global failure
 *                                     0BBBBB          distance value below error threshold
 * sensor board      dist[1]                0xFFFF          all three sensors failed
 *                                     0xEEEE          invalid command
 */

// Inclusions
#include <msp430.h>
#include "i2c_slave_handler.h"
#include "tinyLiDAR_handler.h"
#include "ultrasonic_handler.h"

// Defines
#define slavAdd 0x12 // front, 7-bit i2c address (8-bit = 0x24)
// #define slavAdd 0x24 // back, 7-bit i2c address (8-bit = 0x48)
// #define slavAdd 0x36 // left, 7-bit i2c address (8-bit = 0x6C)
// #define slavAdd 0x48 // right, 7-bit i2c address (8-bit = 0x90)
// #define slavAdd 0x5A // up, 7-bit i2c address (8-bit = 0xB4)
// #define slavAdd 0x6C // down, 7-bit i2c address (8-bit = 0xD8)

// start main()
int main(void)
{

```

```

// MSP430 Initialization
WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
BCSCTL1 = CALBC1_16MHZ;     // Set DCO
DCOCTL = CALDCO_16MHZ;

// i2c slave initialization
i2c_slave_init(slavAdd);

// tinyLiDAR initialization
tinyLiDAR_init();

// ultrasonic initialization
ultrasonic_init();

// Variables
unsigned int dist[2], distIR[2], distUS, distbyte1, distbyte2, devID;
const int t1_us_offset = 20; // mm, forward distance between ultrasonic and tinyLiDARs
int error_threshold; // mm, threshold for checking for valid distance values

while (1){
    if(UCB0STAT&UCSTPIFG){    // check for i2c RX stop flag
        if (i2cRXflag>0){

            /* run sensors based on mode command */
            switch (i2cRXData[0]){
                case 'E':
                    /* set error threshold */
                    error_threshold = i2cRXData[1];
                    break;

                case 'T':
                    /* trigger only the tinyLiDARs */
                    trigger_tinyLiDAR(distIR,0,error_threshold); // outputs minimum distance
between tinyLiDARs
                    dist[1] = distIR[1];
                    dist[0] = 1;
                    devID = distIR[0];
                    break;

                case 'U':
                    /* trigger ultrasonic */
                    trigger_ultrasonic();           // trigger ultrasonic
                    break;

                case 'V':
                    /* capture ultrasonic distance */
                    ultrasonic_dist_capture(&distUS); // capture ultrasonic distance
                    distUS += t1_us_offset;

                    /* trigger tinyLiDARs */
//                    trigger_tinyLiDAR(distIR,0,error_threshold); // trigger tinyLiDARs,
output min distance

                    /* select smallest distance */
smallest?
                    if ((distIR[1]<=distUS)||((distUS<error_threshold)){ // IR distance
                        dist[1] = distIR[1];
                        dist[0] = 1;
                        devID = distIR[0];
                    }else{ // US distance smallest?
                        dist[1] = distUS;

```

```

        dist[0] = 1;
        devID = 'U';
    }
    break;

default:
    /* invalid operator/command */
    dist[0] = 0;
    dist[1] = 0xEEEE;
    break;
}

/* prepare i2c TX */
if ((i2cRXData[0]=='V')||(i2cRXData[0]=='T')){
    if ((dist[0]==0)||(dist[1]==0xFFFF)){
        i2cTXData[0] = 'F';           // 'fail', failed to capture

distance
        i2cTXData[1] = 0xFF;           // return dummy error values
        i2cTXData[2] = 0xFF;
        i2cTXData[3] = 'X';
    }else{
transfer
        distbyte1 = dist[1] & 0xFF00;   // separate bytes for i2c 8-bit

        distbyte1 >>= 8;
        distbyte2 = dist[1] & 0x00FF;

        i2cTXData[0] = 'P';           // 'pass', captured distance
        i2cTXData[1] = distbyte1;     // return first byte
        i2cTXData[2] = distbyte2;     // return second byte
        i2cTXData[3] = devID;        // min dist device
    }

    /* reset RX flag */
    i2cRXflag = 0;

    /* clear values */
    dist[0] = 0;
    dist[1] = 0;
    distbyte1 = 0;
    distbyte2 = 0;
    devID = 0;
}
}
}
} // end while
} // end main()

```

tinyLiDAR handler

```

/*
 * tinyLiDAR_handler.c
 *
 * Created on: Apr 15, 2019
 * Modified: Oct 11, 2019
 * Author: tholliday
 */

// Inclusions
#include <msp430.h>
#include <math.h>
#include "i2c_handler.h"

// Defines
#define duinoAdd1 0x20 // 8-bit address (7-bit = 0x10)
#define duinoAdd2 0xA0 // 8-bit address (7-bit = 0x50)

// Global Variables
char tinyLidar_1w[2], tinyLidar_2w[2], tinyLidar_1r[3], tinyLidar_2r[3];

void tinyLiDAR_init(void){
    // initializes bitbang communication with 2 tinyLiDARs on an i2c bus

    /* i2c initialization */
    i2c_bb_init();

    /* initialize i2c vectors */
    // first slave
    tinyLidar_1w[0] = duinoAdd1; // tinyLiDAR slave address, write mode
    tinyLidar_1w[1] = 0x44; // tx data, distance capture command
    tinyLidar_1r[0] = duinoAdd1+1; // tinyLiDAR slave address, read mode
    tinyLidar_1r[1] = 0x00; // rx data, first distance byte
    tinyLidar_1r[2] = 0x00; // rx data, second distance byte

    // second slave
    tinyLidar_2w[0] = duinoAdd2; // tinyLiDAR slave address, write mode
    tinyLidar_2w[1] = 0x44; // tx data, distance capture command
    tinyLidar_2r[0] = duinoAdd2+1; // tinyLiDAR slave address, read mode
    tinyLidar_2r[1] = 0x00; // rx data, first distance byte
    tinyLidar_2r[2] = 0x00; // rx data, second distance byte

} // end tinyLiDAR_init()

void trigger_tinyLiDAR(unsigned int *output, int lidarNdx, int threshold){
    /* triggers both tinyLiDARs over the i2c bus and outputs a distance value
     * based on value in lidarNdx
     *
     * lidarNdx    command
     * -----
     * 0           compare distances from both tinyLiDARs and output smallest
     * 1           output distance from tinyLiDAR 1
     * 2           output distance from tinyLiDAR 2
     */
}

```

```

// Variables
volatile unsigned int i;
unsigned int distOut1, distOut2;

/* reset i2c rx vectors */
for (i=1;i<3;i++){
    tinyLidar_1r[i] = 0x00;
    tinyLidar_2r[i] = 0x00;
}

/* trigger first LiDAR */
i2c_bb_rxtx(tinyLidar_1w,2,0);           // write mode
__delay_cycles(1000);                   // delay to allow IR capture

if (i2c_bb_rxtx(tinyLidar_1r,1,2)==1){  // successful read, ACK; read mode
    // distance capture
    distOut1 = tinyLidar_1r[1];
    distOut1 = (distOut1<<8) | tinyLidar_1r[2]; // combine returned bytes to make dist
value
    if (distOut1<threshold)
        distOut1 = 0xB BBB; // error/timeout check
}else{
    distOut1 = 0xFFFF; // unsuccessful read, no ACK
}

/* trigger second LiDAR */
i2c_bb_rxtx(tinyLidar_2w,2,0);           // write mode
__delay_cycles(1000);                   // delay to allow IR capture

if (i2c_bb_rxtx(tinyLidar_2r,1,2)==1){  // successful read, ACK; read mode
    // record distance
    distOut2 = tinyLidar_2r[1];
    distOut2 = (distOut2<<8) | tinyLidar_2r[2]; // combine returned bytes to make dist
value
    if (distOut2<threshold)
        distOut2 = 0xB BBB; // error/timeout check
}else{
    distOut2 = 0xFFFF; // unsuccessful read, no ACK
}

/* output distance based on lidarNdx */
switch (lidarNdx){
case 0: // compare distances from both tinyLiDARs and output smallest
    if (distOut1<=distOut2){
        output[0] = 1; // LiDAR indicator
        output[1] = distOut1; // first LiDAR distance smallest
    }else{
        output[0] = 2; // LiDAR indicator
        output[1] = distOut2; // second LiDAR distance smallest
    }
    break;

case 1: // output distance from tinyLiDAR 1
    output[0] = 1; // LiDAR indicator
    output[1] = distOut1; // output tinyLiDAR 1 distance
    break;

case 2: // output distance from tinyLiDAR 2
    output[0] = 2; // LiDAR indicator
    output[1] = distOut2; // output tinyLiDAR 2 distance

```

```

        break;

    default: // invalid Ndx or global failure
        output[0] = 0; // LiDAR indicator
        output[1] = 0xEEEE; // output error distance
        break;
    }
} // end trigger_tinyLiDARs

/*
 * tiny_LiDAR_handler.h
 *
 * Created on: Apr 18, 2019
 * Modified: Oct 11, 2019
 * Author: tholliday
 */

#ifndef TINYLIDAR_HANDLER_H_
#define TINYLIDAR_HANDLER_H_

void tinyLiDAR_init(void);
void trigger_tinyLiDAR(unsigned int *, int, int);

#endif /* TINYLIDAR_HANDLER_H_ */

```

Ultrasonic handler

```

/*
 * ultrasonic_handler.c
 *
 * Created on: Feb 19, 2019
 * Modified: Oct 16, 2019
 * Author: tholliday
 */

// Inclusions
#include <msp430.h>
#include <math.h>

// Defines
#define TRIG BIT3
#define ECHO BIT4
#define TRIG_DIR P2DIR
#define TRIG_OUT P2OUT
#define ECHO_DIR P2DIR
#define ECHO_IE P2IE
#define ECHO_IES P2IES
#define ECHO_IFG P2IFG

// Global variables
unsigned int Distclicks;
int UPCOUNTSTATE;

```



```

/* UPCOUNTSTATE VALUES
 *   0 - system ready/not running
 *   1 - trigger signal sent, waiting for trigger timing
 *   2 - end trigger timing, initialize echo receiving, waiting for echo
 *   3 - echo received and time value acquired
 *   4 - (or greater) timeout has occurred.
 */

void ultrasonic_init(void){

    // Initialize TRIG
    TRIG_DIR |= TRIG;           // Set pin 2.3 as a trigger for the ultrasonic sensor
    TRIG_OUT &=~ TRIG;         // Initialize 2.3 as low for the trigger (trigger is high)

    // Initialize ECHO
    ECHO_IES &=~ ECHO;         // set echo hardware interrupt to lo/high edge
    ECHO_IE |= ECHO;          // set pin 2.4 as echo hardware interrupt

    // Enable timer and hardware interrupts
    _BIS_SR(GIE);              // Enable interrupts for the Port Triggering
    TA1CTL = (TASSEL_2 + ID_3 + MC_2); // configure interrupt timer
    TA1CCR0 = 42000;

    // Initialize ultrasonic state
    UPCOUNTSTATE = 0;
}

// end ultrasonic_init()

void trigger_ultrasonic(void){

    /* state: system ready */
    if (UPCOUNTSTATE==0){
        UPCOUNTSTATE = 1;      // set state to "signal sent"
        TRIG_OUT |= TRIG;     // Trigger the output to start the signal
        ECHO_IES &=~ ECHO;    // set lo/hi edge on echo interrupt
        delay_cycles(160);    // approximately 10us wait
        TRIG_OUT &=~ TRIG;    // End the trigger sequence
    }

    if (UPCOUNTSTATE>3){
        TA1CCTL0 &=~ CCIE;    // disable timer interrupt
    }
}

// end trigger_ultrasonic()

void ultrasonic_dist_capture(unsigned int *dist){

    /* state: echo received & recorded */
    if (UPCOUNTSTATE==3){
        // TA1CCTL0 &=~ CCIE; // disable timer interrupt
        UPCOUNTSTATE = 0;    // reset state
        dist[0] = 0.0884*Distclicks - 16; // convert and output the value as a distance
        // dist[0] = Distclicks;
    }
}

```

```

/* state: timeout or global failure */
if (UPCOUNTSTATE>=4){
    TA1CCTL0 &=~ CCIE;           // disable timer interrupt
    UPCOUNTSTATE = 0;           // reset state
    dist[0] = 0xFFFF;          // output error value
}

} // end ultrasonic_dist_capture()

// ECHO hardware interrupt
#pragma vector=PORT2_VECTOR
__interrupt void Port_2(void)
{
    // when the echo is captured the value is outputted in Distclicks
    if (UPCOUNTSTATE<4){
        if (ECHO_IES & ECHO){    // End of echo time
            ECHO_IES &=~ ECHO;   // Set lo/hi edge trigger
            Distclicks = TA1R;    // store value in Distclicks
            UPCOUNTSTATE = 3;    // set state to "time value acquired"
            TA1CCTL0 &=~ CCIE;   // disable timer interrupt
        }else{                  // Beginning of echo time
            ECHO_IES |= ECHO;    // set hi/lo edge trigger
            TA1R = 0;           // clear distance register
            UPCOUNTSTATE = 2;    // set state to "waiting"
            TA1CCTL0 = CCIE;    // enable timer interrupt
        }
    }
    ECHO_IFG &=~ ECHO;          // reset echo interrupt flag
} // end echo received interrupt

// ECHO rx timeout interrupt
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMERA1_ISR (void)
{
    // timer that ends echo receiving if it is taking too long
    if (UPCOUNTSTATE==2){     // Timeout has occurred
        UPCOUNTSTATE = 4;      // set state to "timeout"
    }else{
        UPCOUNTSTATE++;
    }
} // end timer interrupt

/*
 * ultrasonic_header.h
 *
 * Created on: Feb 19, 2019
 * Modified: Sept 11, 2019
 * Author: tholliday
 */

#ifndef ULTRASONIC_HANDLER_H_
#define ULTRASONIC_HANDLER_H_

```

```
extern unsigned int UPCOUNTSTATE;

void ultrasonic_init(void);
void trigger_ultrasonic(void);
void ultrasonic_dist_capture(unsigned int *);

#endif /* ULTRASONIC_HANDLER_H_ */
```

I2C bitbang handler for tinyLiDARs

```
/*
 * i2c_handler.c
 *
 * Created on: Feb 23, 2019
 * Modified: April 25, 2019
 * Author: tholliday
 *
 *https://www.embeddedrelated.com/showcode/334.php
 */

// Inclusions
#include <msp430.h>

// Defines
#define SDA BIT0
#define SCL BIT1
#define SDA_DIR P2DIR
#define SDA_OUT P2OUT
#define SCL_DIR P2DIR
#define SCL_OUT P2OUT
#define SDA_IN P2IN
#define i2cDelay 500

void i2c_bb_init(void){
    // i2c pin initialization

    SDA_DIR |= SDA;    // set data output
    SCL_DIR |= SCL;    // set clk output
    SDA_OUT |= SDA;    // data high
    SCL_OUT |= SCL;    // clk high
}

// end i2c_bb_init()

void i2c_bb_start(void){
    // i2c start: DATA low when CLK high

    SCL_OUT |= SCL;    // clk high
    SDA_DIR |= SDA;    // set data output
    SDA_OUT &=~ SDA;    // data low
    __delay_cycles(i2cDelay);
    SCL_OUT &=~ SCL;    // clk low
```

```

    __delay_cycles(2);
} // end i2c_bb_start()

void i2c_bb_stop(void){
    // i2c stop: CLK high when DATA low

    __delay_cycles(i2cDelay);
    SCL_OUT &=~ SCL;           // clk low
    SDA_DIR |= SDA;           // set data output
    SDA_OUT &=~ SDA;         // data low
    __delay_cycles(i2cDelay);
    SCL_OUT |= SCL;          // clk high
    __delay_cycles(i2cDelay);
    SDA_OUT |= SDA;          // data high
} // end i2c_bb_stop()

int i2c_bb_rxtx(char *i2cData, int numTX, int numRX){
    /* master read and write to slave
    *
    * *i2cData = pointer to i2c buffer data
    * numTX    = number of 8-bit writes
    * numRX    = number of 8-bit reads
    *
    * returns 0 if unsuccessful (failed ACK)
    * returns 1 if successful
    */

    // Variables
    volatile unsigned int i,k,temp;

    /* Start condition */
    i2c_bb_start();

    /* start i2c TX */
    for (i=0;i<numTX;i++){
        temp = i2cData[i];           // store i-th buffer value

        /* start 8-bit tx */
        for (k=0;k<8;k++){
            __delay_cycles(i2cDelay);
            if ((temp & 0x80)==0x80){
                SDA_OUT |= SDA;       // data high
            }else{
                SDA_OUT &=~ SDA;      // data low
            }
            SCL_OUT |= SCL;           // clk high
            __delay_cycles(i2cDelay);
            temp <<= 1;
            SCL_OUT &=~ SCL;          // clk low
        }
        /* end 8-bit tx */

        /* start tx acknowledge check */
        SDA_DIR &=~ SDA;             // set data input
        __delay_cycles(i2cDelay);
        SCL_OUT |= SCL;             // clk high
    }
}

```

```

    __delay_cycles(i2cDelay);
    if (SDA_IN & SDA){           // ACK missed
        SDA_DIR |= SDA;        // set data output
        SDA_OUT &=~ SDA;       // data low
        SCL_OUT &=~ SCL;       // clk low
        __delay_cycles(i2cDelay);
        SCL_OUT |= SCL;        // clk high
        __delay_cycles(i2cDelay);
        SDA_OUT |= SDA;        // data high
        return 0;
    }
    SCL_OUT &=~ SCL;           // clk low
    SDA_DIR |= SDA;           // set data output
    __delay_cycles(i2cDelay);
    SDA_OUT &=~ SDA;          // data low
    /* end tx acknowledge check */
}
/* end i2c TX */

SDA_DIR &=~ SDA;             // set data input
SDA_OUT &=~ SDA;            // data low
if (numRX==--1)
    numRX = 100;

/* start i2c RX */
for (i=numTX;i<(numTX+numRX);i++){
    temp = 0x00;             // reset temp value
    SDA_DIR &=~ SDA;        // set data input

    /* begin 8-bit rx */
    for (k=0;k<8;k++){
        temp <<= 1;         // bitshift temp by 1
        __delay_cycles(i2cDelay);
        SCL_OUT |= SCL;    // clk high
        __delay_cycles(i2cDelay);
        if ((SDA_IN & SDA)==SDA){
            temp |= 0x01;   // set temp to hex 1
        }else{
            temp &=~ 0x01;
        }
        SCL_OUT &=~ SCL;   // clk low
        SDA_OUT |= SDA;    // data high
    }
    /* end 8-bit rx */

    /* start rx acknowledge check */
    SDA_DIR |= SDA;        // set data output
    if (i==(numTX+numRX-1)){
        SDA_OUT |= SDA;    // master send NACK
    }else{
        SDA_OUT &=~ SDA;   // master send ACK
    }
    __delay_cycles(i2cDelay);
}
/* end rx acknowledge check */

if (i==numTX){
    if (numRX==100)
        numRX = temp;
}

i2cData[i] = temp;

```

```

    __delay_cycles(i2cDelay);
    SCL_OUT |= SCL;           // clk high
    __delay_cycles(i2cDelay*3);
    SCL_OUT &=~ SCL;         // clk low
    __delay_cycles(i2cDelay);
    SDA_OUT &=~ SDA;         // data low
    SDA_DIR &=~ SDA;         // set data output
}
/* end i2c RX */

/* Stop condition */
i2c_bb_stop();
return 1;

} // end i2c_bb_rxtx()/*
 * i2c_handler.h
 *
 * Created on: March 4, 2019
 * Author: tholliday
 */

#ifdef I2C_HANDLER_H_
#define I2C_HANDLER_H_

void i2c_bb_init(void);
void i2c_bb_start(void);
void i2c_bb_stop(void);
int i2c_bb_rxtx(char *,int,int);

#endif /* I2C_HANDLER_H_ */

```

I2C slave handler

```

/*
 * i2c_slave_handler.c
 *
 * Created on: April 17, 2019
 * Author: tholliday
 */

// Inclusions
#include <msp430.h>

// Defines
#define i2c_max      12
#define SEL          P1SEL
#define SEL2         P1SEL2
#define PINS         (BIT6|BIT7)

// Global Variables
unsigned char i2cTXData[i2c_max], i2cRXData[i2c_max];
unsigned int txDataPtr = 0, rxDataPtr = 0;
unsigned int i2cRXflag = 0;
unsigned int i2cmodeflag = 0;
// 0   slave <-- master mode (TX)
// 1   slave --> master mode (RX)

```

```

void i2c_slave_init(int slavAdd){
    // initializes i2c slave using USCI_B0 timers

    // initialize pins
    SEL |= PINS;           // Assign I2C pins to USCI_B0
    SEL2 |= PINS;         // Assign I2C pins to USCI_B0

    UCB0CTL1 |= UCSWRST;   // Enable SW reset
    UCB0CTL0 = (UCMODE_3|UCSYNC); // I2C Slave, synchronous mode
    UCB0I2COA = slavAdd;  // set slave address
    UCB0CTL1 &=~ UCSWRST; // Clear SW reset, resume operation
    UCB0I2CIE |= UCSTTIE; // Enable STT interrupt
    IE2 |= (UCB0TXIE|UCB0RXIE); // enable TX interrupt

    i2cmodeflag = 0;      // set to i2c TX slave mode
}

// end i2c_slave_init()

// i2c TX interrupt
#pragma vector = USCIAB0TX_VECTOR
__interrupt void USCIAB0TX_ISR(void)
{
    unsigned int i;

    if (i2cmodeflag==1){ // i2c TX slave mode?
        UCB0TXBUF = i2cTXData[txDataPtr]; // place TX data in buffer
        txDataPtr++; // increment TX pointer
    }else{ // i2c RX slave mode?
        i2cRXData[rxDataPtr] = UCB0RXBUF; // capture RX data
        rxDataPtr++; // increment RX pointer
        i2cRXflag++; // increment RX flag
    }
}

// end i2c TX interrupt

// i2c RX interrupt
#pragma vector = USCIAB0RX_VECTOR
__interrupt void USCIAB0RX_ISR(void)
{
    /* Slave --> master mode (TX) */
    if (IFG2&UCB0TXIFG){ // check TX flags for beginning of i2c RX
slave mode
        i2cmodeflag = 1; // set to i2c RX slave mode
        if (UCB0STAT&UCSTTIFG){
            UCB0STAT &=~ (UCSTPIFG|UCSTTIFG); // clear i2c interrupt flags
            txDataPtr = 0; // reset TX pointer
        }
    }

    /* Slave <-- master mode (RX) */
    if (IFG2&UCB0RXIFG){ // check RX flags for beginning of i2c TX
slave mode
        i2cmodeflag = 0; // set to i2c TX slave mode
        if (UCB0STAT&UCSTTIFG){
            UCB0STAT &=~ (UCSTPIFG|UCSTTIFG); // clear i2c interrupt flags
            rxDataPtr = 0; // reset RX pointer
        }
    }
}

```

```
    }  
  }  
} // end i2c RX interrupt  
  
/*  
 * i2c_slave_handler.h  
 *  
 * Created on: Apr 17, 2019  
 * Author: tholliday  
 */  
  
#ifndef I2C_SLAVE_HANDLER_H_  
#define I2C_SLAVE_HANDLER_H_  
  
extern unsigned char i2cTXData[24], i2cRXData[24];  
extern unsigned int i2cRXflag;  
  
void i2c_slave_init(int);  
  
#endif /* I2C_SLAVE_HANDLER_H_ */
```


Appendix F: MITM Code – Raspberry Pi Version

Main script

```

# rpi_mitm_v5.py

# Created: June 10, 2020
# Modified: June 15, 2020
# Author: THolliday

# This script runs the prototype MITM, including:
#  sbus comms
#  sensor board control
#  feedback control on forward direction (Ele)
#  data logging

## RPi Setup
#!/usr/bin/python3

## Imports
import serial
import time
import csv
import pigpio

## Handler functions
from sbusRX_handler import *    # sbus RX
__all__ = ["SBUS_RX"]

from sbusTX_handler import *    # sbus TX
__all__ = ["SBUS_TX"]

from sbusDL_handler import *    # sbus data logging
__all__ = ["SBUS_DL"]

from multi_sb_handler_v2 import * # multi sensor board comms
__all__ = ["multi_sb_ctrl"]

from fb_ctrl_handler_v2 import * # feedback control
__all__ = ["mitm_fb_ctrl"]

## main loop
if __name__ == '__main__':

    # Constants
    sbusMIN = 172    # min sbus value
    sbusMID = 992    # neutral sbus value
    sbusMAX = 1811   # max sbus value

    # flags
    logFlag = 0      # flag for start of logging
    armFlag = 0      # flag for system arming
    sbFlag = 0       # flag for sb connected
    distflag = 0     # flag for dist capture
    fbFlag = 0       # flag for start of feedback control

    # states
    sbState = 0      # sensor board states:

```

```

# 0, trigger ultrasonic & tinyLiDARs
# 1, request min dist from tinyLiDARs, & compare dist from all sensors
# 2, request min dist between all sensors

# sb error threshold (mm)
eThres = 100;

# capture start of logging session
log_timestr = time.strftime('%d%m%Y-%H%M%S')

# initialize handlers
sbusRX = SBUS_RX('/dev/ttyAMA0')
sbusTX = SBUS_TX('/dev/ttyAMA0')
sbusDL = SBUS_DL(log_timestr)
sboard = multi_sb_ctrl(eThres,log_timestr)

# set activation and desired distances
desDist = [500,500]
actDist = [1000,1000]

# ask for activation and desired distances
#desDist = [0 for i in range(sboard.sb_count)]
#for kk in range(sboard.sb_count):
#    usrMes = 'Minimum desired distance (mm) for ' + sboard.sb_IDs[kk] + ' sensor board: '
#    desDist[kk] = int(input(usrMes))

# initialize feedback controller if sb connected
if not sboard.sb_count == 0:
    fbCtrl = mitm_fb_ctrl(sboard.sb_count)
    distBuff = [0 for ii in range(sboard.sb_count)]
    sbFlag = 1 # set flag

while True:
    # simulate time between sbus frames (~15 ms)
    time.sleep(0.008)

    # reset TX buffer
    TXbuff = [0 for ii in range(len(sbusRX.sbusChannels))]

    # sbus RX
    sbusRX.sbus_read()

    # sbus frame received?
    if sbusRX.isReady:
        rxTime = time.time() # save rx timestamp

    # system armed?
    if not sbusRX.sbusChannels[4]==sbusMAX:
        if armFlag == 0: # notify disarmed state
            print('System disarmed')
            armFlag = 1 # set flag
        else:
            if armFlag == 1: # notify armed state
                print('System armed')
                armFlag = 0 # reset flag

    # cap Ele value for fb ctrl testing
    if sbusRX.sbusChannels[2]>1200:
        sbusRX.sbusChannels[2] = 1200

    # read min dist from sensor boards?

```

```

if sbusRX.sbusChannels[7]==sbusMAX and sbFlag==1:
    # trigger ultrasonics & tinyLiDARs
    if sbState==0:
        sboard.sb_ultrasonic_trig()    # trigger ultrasonics
        sboard.sb_tinyLiDAR_trig()    # trigger tinyLiDARs & find min dist
        sbState += 1                  # update state
        if distflag==1:
            for ii in range(len(sboard.SBdist)):
                sboard.devID[ii] = 'R'
                sboard.SBdist[ii] = sboard.distBuff[ii]
                stext = 'S0'          # update state for log

    # request dist value from Tls & call for comparison between all sensors
    elif sbState==1:
        sboard.sb_dist_capture()      # request TL dist value
        sboard.sb_full_trig()        # call for min dist between all sensors
        sbState += 1                  # update state
        distflag = 1                  # set flag
        stext = 'S1'                  # update state for log

    # request min dist between all sensors
    elif sbState==2:
        sboard.sb_dist_capture()      # request dist value
        sbState = 0                  # reset flag
        stext = 'S2'                  # update state for log

else:
    sbState = 0                      # reset state
    distflag = 0                      # reset flag

    # run feedback control?
    if sbusRX.sbusChannels[7]==sbusMAX and sbusRX.sbusChannels[9]==sbusMAX and
distflag==1:
        TXbuff =
fbCtrl.fb_ctrl(actDist,desDist,sbusRX.sbusChannels,sboard.sb_IDs,sboard.SBdist)
        if fbFlag == 0:                # notify controller enable
            print('Controller enabled')
            fbFlag = 1                # set flag
        else:
            for ii in range(len(sbusRX.sbusChannels)): # fill TX buffer
                TXbuff[ii] = sbusRX.sbusChannels[ii]
            if fbFlag == 1:            # notify controller disable
                print('Controller disabled')
                for ii in range(3):    # reset controller buffers
                    fbCtrl.front_err_buff[ii] = 0
                    fbCtrl.front_fc_buff[ii] = 0
                fbFlag = 0            # reset flag

    # sbus TX
    for ii in range(len(TXbuff)):
        sbusTX.sbusChannels[ii] = TXbuff[ii]    # fill TX sbus channels
    sbusTX.sbus_write()
    txTime = time.time()                        # save tx timestamp

    # log data
    if sbusRX.sbusChannels[6]==sbusMAX:
        sbusDL.sbus_data_log(sbusRX.sbusChannels,sbusTX.sbusChannels,rxTime,txTime)
        if distflag == 1:                # log sensor board values
            sboard.sb_logging(stext,fbCtrl.ctrlEnable)
        if logFlag == 0:                # notify logging start
            print('Logging started')

```

```

        logFlag = 1                # set flag
else:
    if logFlag == 1:              # notify logging end
        print('Logging ended')
    logFlag = 0                  # reset flag

```

LCAS feedback controller handler

```
"""
```

```
fb_ctrl_handler_v2.py
```

```
Created:   Apr 3, 2020
```

```
Modified:  June 2, 2020
```

```
Author:    THolliday
```

```
Script for running feedback control on the MITM. Currently
limited to only the forward (Ele) direction.
```

```
"""
```

```

class mitm_fb_ctrl():
    def __init__(self,numSB):
        # initialization

        # Class Variables
        self.front_err_buff = [0,0,0]
        self.front_fc_buff = [0,0,0]
        self.ctrlEnable = 0

    def fb_ctrl(self,actDist,desDist,channelsIn,sbIDs,sbDist):
        # feedback controller

        # Constants
        numSB = len(sbIDs) # number of sensor boards
        sbusMIN = 172
        sbusMID = 992
        sbusMAX = 1811

        # Variables
        modChannels = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
        for ii in range(len(channelsIn)): # set initial output to input
            modChannels[ii] = channelsIn[ii]

        # run controller
        for kk in range(numSB):

            ## Front (Ele 992-1811) ##
            if sbIDs[kk]=='F':
                # controller coefficients
                a = [1,-1.9552,0.9552] # denominator
                b = [2.2435,-4.4752,2.2317] # numerator

                # calculate error to des dist & convert to meters
                self.front_err_buff[0] = (sbDist[kk]-desDist[kk])/1000

                # controller

```

```

        self.front_fc_buff[0] = (-a[2]*self.front_fc_buff[2] -
a[1]*self.front_fc_buff[1]) + \
        (b[2]*self.front_err_buff[2] + b[1]*self.front_err_buff[1] +
b[0]*self.front_err_buff[0])

        # saturation check
        if self.front_fc_buff[0] > 1:
            self.front_fc_buff[0] = 1
        elif self.front_fc_buff[0] < -1:
            self.front_fc_buff[0] = -1

        # check if in activation window
        if (sbDist[kk]<=actDist[kk]) and (channelsIn[2]>=sbusMID):
            # modify Ele sbus value
            modChannels[2] = int(self.front_fc_buff[0]*(sbusMAX-sbusMID) + sbusMID)
# scale back to sbus from normalized
            self.ctrlEnable = 1
        else:
            self.ctrlEnable = 0

        # store error values for next iteration
        self.front_err_buff[2] = self.front_err_buff[1]
        self.front_err_buff[1] = self.front_err_buff[0]
        self.front_err_buff[0] = 0

        # store controller outputs for next iteration
        self.front_fc_buff[2] = self.front_fc_buff[1]
        self.front_fc_buff[1] = self.front_fc_buff[0]
        self.front_fc_buff[0] = 0

    ## Back (Ele 172-992) ##
    if sbIDs[kk]=='B':
        pass

    ## Left (Ail 172-992) ##
    if sbIDs[kk]=='L':
        pass

    ## Right (Ail 992-1811) ##
    if sbIDs[kk]=='R':
        pass

    ## Up (Thr 992-1811) ##
    if sbIDs[kk]=='U':
        pass

    ## Down (Thr 172-992) ##
    if sbIDs[kk]=='D':
        pass

# return modified channels
return modChannels

```

Sensor board communications handler

```

"""
multi_sb_handler_v2.py

Created:    Feb 25, 2020
Modified:   June 15, 2020
Author:    THolliday

Script for controlling and logging multiple
sensor boards
"""

## RPi Setup
#!/usr/bin/env python3

## Imports
import time
import pigpio
import csv

class multi_sb_ctrl():
    def __init__(self,err,timestr):
        # sensor board comms initialization

        ## Constants
        BUS = 1
        errThres = err
        max_numSB = 10

        ## Variables
        i2c_count = 0
        self.sb_count = 0

        ## scan for sensor boards
        i2cScan = i2c_scan()
        sb_addresses = i2cScan.i2c_scanner(BUS,max_numSB) # find sb addresses
        for jj in range(len(sb_addresses)): # check for non-zero addresses
            if not sb_addresses[jj] == 0:
                i2c_count += 1

        self.sb_handles = [0 for i in range(i2c_count)] # initialize handle list
        self.sb_IDs = [0 for i in range(i2c_count)] # initialize ID list

        ## Initialize sensor board i2c
        self.i2c = pigpio.pi()

        ## initialize sensor boards
        SBcommand = 0x45 # error threshold set command
        for kk in range(i2c_count):
            try:
                # attempt to setup board at given address
                self.sb_handles[kk] = self.i2c.i2c_open(BUS,sb_addresses[kk]) # open
                self.i2c.i2c_write_device(self.sb_handles[kk],[SBcommand,errThres]) # send
            except:
                # identify board
                self.sb_count += 1

        comms with sb
        threshold

```

```

        if sb_addresses[kk] == 0x12: # front SB
            self.sb_IDs[kk] = 'F'
        if sb_addresses[kk] == 0x24: # back SB
            self.sb_IDs[kk] = 'B'
        if sb_addresses[kk] == 0x36: # left SB
            self.sb_IDs[kk] = 'L'
        if sb_addresses[kk] == 0x48: # right SB
            self.sb_IDs[kk] = 'R'
        if sb_addresses[kk] == 0x5A: # up SB
            self.sb_IDs[kk] = 'U'
        if sb_addresses[kk] == 0x6C: # down SB
            self.sb_IDs[kk] = 'D'

    except:
        print('Non-sensor board or no response: ',hex(sb_addresses[kk]))

    ## inform boards identified
    print('Identified sensor boards:',self.sb_IDs)

    ## Class Variables
    self.distBuff = [0 for i in range(self.sb_count)] # initialize dist buffer
    self.numRead = 10 # number of readings in SMA window
    self.readNdx = 0 # readings index
    self.runSum = [0 for i in range(self.sb_count)] # running sum for SMA
    self.distReadings = [[0 for i in range(self.sb_count)] for j in range(self.numRead)]
# SMA readings buffer

    ## Initialize log
    self.sb_timestr = timestr
    # create/open log csv file (with time in file name)
    with open('/home/pi/data_logs/SB_log_' + str(self.sb_timestr) + '.csv','w') as
csvfile:
        # initialize csv
        sblog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)
        # list available sensor boards
        sblog.writerow(['Sensor_boards:',self.sb_IDs])
        # write column titles
        sblog.writerow(['State','Board','Status','Device','Dist','Ctrl','Tsb'])

def sb_ultrasonic_trig(self):
    # sends ultrasonic trigger command

    SBcommand = 0x55 # 'U',85; trigger ultrasonic command
    for kk in range(self.sb_count):
        self.i2c.i2c_write_byte(self.sb_handles[kk],SBcommand) # write to SB

def sb_tinyLiDAR_trig(self):
    # sends only tinyLiDAR trigger command and requests measured dist

    SBcommand = 0x54 # 'T',84; trigger only tinyLiDARs
    for kk in range(self.sb_count):
        self.i2c.i2c_write_byte(self.sb_handles[kk],SBcommand) # write to SB

def sb_full_trig(self):
    # sends command to trigger tinyLiDARs and compare dists measured by all three sensors

    SBcommand = 0x56 # 'V'; trigger TLs and capture ultrasonic dist

```

```

for kk in range(self.sb_count):
    self.i2c.i2c_write_byte(self.sb_handles[kk],SBcommand)    # write to SB

def sb_dist_capture(self):
    # requests min dist from each sensor board

    RX_BYTES = 4
    buffFlag = 0
    self.SBdist = [0 for i in range(self.sb_count)]           # reset dist list
    self.devID = [0 for i in range(self.sb_count)]           # reset ID list
    self.sb_sample_time = [0 for i in range(self.sb_count)]   # reset sample time list

    for kk in range(self.sb_count):
        (count,data) = self.i2c.i2c_read_device(self.sb_handles[kk],RX_BYTES)    # read min
dist
        minDist = list(data)                                                    # convert
tuple
        self.SBdist[kk] = (minDist[1]<<8) | minDist[2]                          # combine
dist bytes
        self.devID[kk] = minDist[3]                                             # save
sensor ID
        self.sb_sample_time[kk] = time.time()                                   # save
sample time

        # error check distance values or if in first state
        if self.SBdist[kk]==0xffff:
            self.devID[kk] = 'X'                                                # measurement failure
            self.SBdist[kk] = self.distBuff[kk]                                # replace error with previous value

        elif self.SBdist[kk]==0xbbbb:
            self.devID[kk] = 'E'                                                # error in tinyLiDAR measurement
            self.SBdist[kk] = self.distBuff[kk]                                # replace error with previous value

        #elif (self.SBdist[kk]<1000)and(self.distBuff[kk]-self.SBdist[kk])>=1000:
        #    self.devID[kk] = 'Z'                                                # significant drop in measurements
        #    self.SBdist[kk] = self.distBuff[kk]                                # replace error with previous value

        else:
            buffFlag = 1    # set flag to update buffer after SMA

        # filter using 10-point SMA
        temp = self.SBdist[kk]                                                  # bring in new sample
        temp /= self.numRead                                                    # divide by length of window
        self.runSum[kk] += temp                                                 # add new sample to running sum
        temp = self.distReadings[self.readNdx][kk]                             # find older sample that is out of
window
        temp /= self.numRead                                                    # divide by length of window
        self.runSum[kk] -= temp                                                 # subtract old sample from running sum
        self.distReadings[self.readNdx][kk] = self.SBdist[kk]                # store new sample in SMA
buffer
        self.SBdist[kk] = round(self.runSum[kk])                               # replace with filtered value

        # SMA index check
        if not (self.readNdx >= self.numRead - 1):
            self.readNdx += 1    # increase readings count
        else:
            # reached end of window
            self.readNdx = 0    # reset count

        # update buffer
        if buffFlag==1:

```



```

        self.i2c.i2c_write_byte(h,0xFA)
        i2cDevs[count] = device      # save detected device address
        count += 1                  # update count

    except:                          # no device detected at address
        pass

    self.i2c.i2c_close(h)           # close i2c comms at address

self.i2c.stop                       # end i2c comms

return i2cDevs

```

SBUS RX handler

```

# sbusRX_handler.py

# Created on:   Nov 4, 2019
# Modified:    July 31, 2020
# Author:      THolliday

# derived from:
#   sbusPythonDriver by Donald Simonet on Framagit
#   https://framagit.org/dsimonet/sbusPythonDriver

#   sbus_ultrasonic_v5 by THolliday
#       sbus_handler.c by Bhill & THolliday

# This script captures sbus frames and decodes them into individual channels.
# An sbus frame is made up of 25 bytes, with 16 channels spread out over
# 22 of the bytes.

## Imports
import serial
import time

class SBUS_RX():
    def __init__(self, _uart_port = '/dev/ttyAMA0'):
        # sbus comms initialization

        # RPi uart initialization
        self.ser = serial.Serial(
            port = _uart_port,          # indicate UART port
            baudrate = 100000,         # sbus runs at 100k baud
            parity = serial.PARITY_EVEN, # set even parity
            stopbits = serial.STOPBITS_TWO, # set two stopbits
            bytesize = serial.EIGHTBITS, # set byte size to 8 bits
            timeout = 0                 # disable UART timeout
        )

        # set sbus constants
        self.START_BYTE = 0x0F        # startbyte
        self.END_BYTE = 0x00          # endbyte
        self.SBUS_FRAME_LEN = 25     # full frame length is 25 bytes
        self.SBUS_CHAN_LEN = 16      # 16 individual channels
        self.SBUS_BITS = 11          # number of bits per channel

```

```

# variable initialization
self.isReady = True # RX flag
self.sbusFrameIn = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # captured
sbus RX
frame
self.sbusFrame = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # endian-
swapped sbus RX frame
self.sbusChanIn = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # RX channels
self.sbusChannels = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # RX endian-
swapped channels

def chan_endian_swap(self):
# swaps the bit order of the sbus channel values

for i in range(0,(len(self.sbusChanIn))):
temp = 0
bmask = 0x8000
for j in range(0,16):
if j<8:
temp |= (self.sbusChanIn[i]&int(bmask))>>(15-(j*2)) # first byte
else:
temp |= (self.sbusChanIn[i]&int(bmask))<<((j*2)-15) # second byte
bmask /= 2 # adjust mask
temp >>= (16-self.SBUS_BITS) # account for number of
bits per packet
self.sbusChannels[i] = temp # store channel value

def frame_endian_swap(self):
# swaps the bit order of the full sbus frame

for i in range(0,len(self.sbusFrameIn)):
temp = 0
bmask = 0x80
for j in range(0,8):
if j<4:
temp |= ((self.sbusFrameIn[i])&int(bmask))>>(7-(j*2)) # first nibble
else:
temp |= ((self.sbusFrameIn[i])&int(bmask))<<((j*2)-7) # second nibble
bmask /= 2 # adjust mask
self.sbusFrame[i] = temp # store frame values

def sbus_read(self):
# reads sbus values over the serial port on the RPi

if self.ser.inWaiting() >= self.SBUS_FRAME_LEN*2: # load at least two full frames
self.isReady = False # processing frames
rxBuff = self.ser.read(self.ser.inWaiting()) # capture sbus bytes
self.numBytes = len(rxBuff) # save number of received bytes
# parse full frame
for rr in range(0,self.SBUS_FRAME_LEN): # step through bytes
# look for end byte, working backwards
if rxBuff[len(rxBuff)-1-rr] == self.END_BYTE:
# based on end byte, find start byte
if rxBuff[len(rxBuff)-rr-self.SBUS_FRAME_LEN] == self.START_BYTE:
# frame is fully mapped and parity checked due to 8E2 format
# only need to remap the frame if it is different than the last
newFrame = rxBuff[len(rxBuff)-rr-self.SBUS_FRAME_LEN:len(rxBuff)-1-rr]

```

```

        if not self.sbusFrameIn == newFrame: # check if new frame is diff
than previous frame (save CPU cycles)
            self.sbusFrameIn = newFrame # store new frame
            self.sbus_decode() # decode new frame

# sbus frame succesfully captured and checked for changes
self.isReady = True # RX frame is ready
break

def sbus_decode(self):
    # decodes an sbus frame into 16 individual channels

    # Conversion parameters
    # chan number [1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16]
    byte_num = [0, 1, 3, 4, 5, 7, 8, 9, 11,
12, 14, 15, 16, 18, 19, 20]
    lowBS = [5, 2, 7, 4, 1, 6, 3, 0, 5,
2, 7, 4, 1, 6, 3, 0]
    lowbitmask = [0xE0, 0xFC, 0x80, 0xF0, 0xFE, 0xC0, 0xF8, 0xFF, 0xE0,
0xFC, 0x80, 0xF0, 0xFE, 0xC0, 0xF8, 0xFF]
    midBS = [3, 6, 1, 4, 7, 2, 5, 8, 3,
6, 1, 4, 7, 2, 5, 8]
    midbitmask = [0xFF, 0x1F, 0xFF, 0x7F, 0x0F, 0xFF, 0x3F, 0x07, 0xFF,
0x1F, 0xFF, 0x7F, 0x0F, 0xFF, 0x3F, 0x07]
    highBS = [0, 0, 9, 0, 0, 10, 0, 0, 0,
0, 9, 0, 0, 10, 0, 0]
    highbitmask = [0x00, 0x00, 0x03, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
0x00, 0x03, 0x00, 0x00, 0x01, 0x00, 0x00]

    # swap bit order over entire frame (UART reads as LSB, but SBUS is MSB)
self.frame_endian_swap()

    # decode data bytes
for i in range(0, self.SBUS_CHAN_LEN):
    self.sbusChanIn[i] = ((self.sbusFrame[byte_num[i]]&highbitmask[i]<<highBS[i]) \
+ ((self.sbusFrame[byte_num[i]+1]&midbitmask[i]<<midBS[i]) \
+ ((self.sbusFrame[byte_num[i]+2]&lowbitmask[i]>>lowBS[i]))

    # swap channel bit order (SBUS has channels encoded as LSB)
self.chan_endian_swap()

```

SBUS TX handler

```

# sbusTX_handler.py

# Created on: Oct 29, 2019
# Modified: July 31, 2020
# Author: THolliday

# derived from:
# sbusPythonDriver by Donald Simonet on Framagit
# https://framagit.org/dsimonet/sbusPythonDriver

# sbus_ultrasonic_v5 by THolliday
# sbus_handler.c by Bhill & THolliday

# This script takes 16 individual channels and encodes them into an sbus frame.

```

```

# An sbus frame is made up of 25 bytes, with the channels spread out over
# the bytes.

## Imports
import serial
import time

class SBUS_TX():
    def __init__(self, _uart_port = '/dev/ttyAMA0'):
        # sbus comms initialization

        # RPi uart initialization
        self.ser = serial.Serial(
            port = _uart_port,                # indicate UART port
            baudrate = 100000,                # sbus runs at 100k baud
            parity = serial.PARITY_EVEN,      # set even parity
            stopbits = serial.STOPBITS_TWO,   # set to two stopbits
            bytesize = serial.EIGHTBITS,     # set byte size to 8 bits
            timeout = 0                       # disable UART timeout
        )

        # set sbus constants
        self.START_BYTE = 0xF0                # first byte
        self.END_BYTE = 0x00                 # last byte
        self.sbusNbits = 11                  # bits per channel
        self.SBUS_FRAME_LEN = 25             # bytes per frame

        # variable initialization
        self.numBytes = 0                    # number of transmitted bytes
        self.sbusChannels = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # RX channels
        self.sbusChanBitSwap = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # RX endian-
swapped channels
        self.sbusBytes = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # TX frame
        self.sbusFrame = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # TX endian-
swapped frame

    def chan_endian_swap(self):
        # swaps the bit order of the sbus channels

        for i in range(0,(len(self.sbusChannels))):
            temp = 0
            bmask = 0x8000
            for j in range(0,16):
                if (j<8):
                    temp |= (self.sbusChannels[i]&int(bmask))>>(15-(j*2)) # first byte
                else:
                    temp |= (self.sbusChannels[i]&int(bmask))<<((j*2)-15) # second byte
                    bmask /= 2 # adjust mask
            temp >>= (16-self.sbusNbits) # shift to account for
number of bits per channel
            self.sbusChanBitSwap[i] = temp # store channel

    def frame_endian_swap(self):
        # swaps the bit order of the entire frame

        for i in range(0,self.SBUS_FRAME_LEN):
            temp = 0
            bmask = 0x80

```

```

    for j in range(0,8):
        if (j<4):
            temp |= (self.sbusBytes[i]&int(bmask))>>(7-(j*2)) # first nibble
        else:
            temp |= (self.sbusBytes[i]&int(bmask))<<((j*2)-7) # second nibble
            bmask /= 2 # adjust mask
        self.sbusFrame[i] = temp # store byte

def sbus_encode(self):
    # encode 16 channels into single sbus frame

    # Conversion parameters
    # Byte number [1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19 20 21
22]
    chan1num = [0, 0, 1, 2, 2, 3, 4, 5, 5,
6, 7, 8, 8, 9, 10, 10, 11, 12, 13, 13, 14,
15]
    chan2num = [0, 1, 2, 2, 3, 4, 5, 5, 6,
7, 8, 8, 9, 10, 10, 11, 12, 13, 13, 14, 15,
16]
    chan1mask = [0x00, 0xE0, 0xFC, 0x00, 0x80, 0xF0, 0xFE, 0x00, 0xC0,
0xF8, 0xFF, 0x00, 0xE0, 0xFC, 0x00, 0x80, 0xF0, 0xFE, 0x00, 0xC0, 0xF8,
0xFF]
    chan2mask = [0xff, 0x1F, 0x03, 0xFF, 0x7F, 0x0F, 0x01, 0xFF, 0x3F,
0x07, 0x00, 0xFF, 0x1F, 0x03, 0xFF, 0x7F, 0x0F, 0x01, 0xFF, 0x3F, 0x07,
0x00]
    chan1BS = [0, 5, 2, 0, 7, 4, 1, 0, 6,
3, 0, 0, 5, 2, 0, 7, 4, 1, 0, 6, 3,
0]
    chan2BS = [3, 6, 9, 1, 4, 7, 10, 2, 5,
8, 0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8,
0]

    # swap bit order of channels to match SBUS format (reverse of the last step of decode)
    self.chan_endian_swap()

    # add startbyte
    self.sbusBytes[0] = self.START_BYTE

    # add in channels (note that the channels are spread across multiple bytes)
    for rr in range(0,22):
        tempFrame = 0
        tempFrame = (self.sbusChanBitSwap[chan1num[rr]]<<chan1BS[rr]) & chan1mask[rr] #
add in first channel data
        tempFrame |= (self.sbusChanBitSwap[chan2num[rr]]>>chan2BS[rr]) & chan2mask[rr] #
add in second channel data
        self.sbusBytes[rr+1] = tempFrame # store byte

    # add flags & digital
    self.sbusBytes[23] = 0

    # add endbyte
    self.sbusBytes[24] = self.END_BYTE

    # swap bit order of entire SBUS frame (SBUS is read as MSB but UART transmits as LSB)
    self.frame_endian_swap()

def sbus_write(self):

```

```

# transmit the full sbus frame over UART

# encode sbus frame
self.sbus_encode()

# transmit the frame
self.numBytes = self.ser.write(self.sbusFrame)

```

SBUS data logging handler

```

# sbusDL_handler.py

# Created on:   Jan 8, 2020
# Modified:    May 28, 2020
# Author:      THolliday

# This script holds dedicated functions for logging sbus data
# on a Raspberry Pi.

## Imports
import time
import csv

class SBUS_DL():
    def __init__(self, timestr):
        # sbus data logging initialization

        # create/open log csv file (with time in file name)
        self.sbus_timestr = timestr
        with open('/home/pi/data_logs/sbus_log_'+str(self.sbus_timestr)+'.csv','w') as
csvfile:
            # initialize csv
            sbuslog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)

            # add column titles

#sbuslog.writerow(['RX/TX','Ch0','Ch1','Ch2','Ch3','Ch4','Ch6','Ch7','Ch9','Tsbustime'])
sbuslog.writerow(['RX/TX','Thr','Ail','Ele','Rud','ARM','LOG','sbEN','Ctrl','Tsbustime'])

    def sbus_data_log(self,rx,tx,rxTime,txTime):
        # logs sbus channel values of RX/TX

        with open('/home/pi/data_logs/sbus_log_'+str(self.sbus_timestr)+'.csv','a') as
csvfile:
            # initialize csv
            sbuslog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)

            # log RX channels

sbuslog.writerow(['RX',str(rx[0]),str(rx[1]),str(rx[2]),str(rx[3]),str(rx[4]),str(rx[6]),
str(rx[7]),str(rx[9]),str(rxTime)])

            # log TX channels

```

```
sbuslog.writerow(['TX',str(tx[0]),str(tx[1]),str(tx[2]),str(tx[3]),str(tx[4]),str(tx[6]),  
str(tx[7]),str(tx[9]),str(txTime)])
```


Appendix G: MITM Code – GPS & Accelerometer Version

Main script

```
# sbus_gps_accel_v3.py

# Created on:   Jan 30, 2020
# Modified:    Mar 6, 2020
# Author:      THolliday

# This script runs and captures data for sbus, gps,
# accel logging

## RPi Setup
#!/usr/bin/python3

## Imports
import serial
import time
import csv
import pigpio
from gps3 import agps3
import os

## Handler functions
from sbusRX_handler import *    # sbus RX
__all__ = ["SBUS_RX"]

from sbusTX_handler import *    # sbus TX
__all__ = ["SBUS_TX"]

from sbusDL_handler import *    # sbus data logging
__all__ = ["SBUS_DL"]

from gps_handler import *       # GPS logging
__all__ = ["GPS"]

from accel_handler_v2 import *  # accel logging
__all__ = ["ACCEL"]

## start main()
if __name__ == '__main__':

    # Constants
    sbusMIN = 172
    sbusNEU = 992
    sbusMAX = 1811

    # Flags
    stepFlag = 0           # flag for start of step
    logFlag = 0           # flag for start of logging
    armFlag = 0           # flag for system arming

    # capture start of logging session
    log_timestr = time.strftime('%d%m%Y-%H%M%S')

    # initialize sbus
    sbusRX = SBUS_RX('/dev/ttyAMA0')
    sbusTX = SBUS_TX('/dev/ttyAMA0')
```

```

sbusDL = SBUS_DL(log_timestr)

# initialize gps & accel
gps = GPS(log_timestr)
accel = ACCEL(log_timestr)

while True:
    # simulate time between sbus frames (~15 ms)
    time.sleep(0.01)

    # sbus RX
    sbusRX.sbus_read()

    # check for new RX frame
    if sbusRX.isReady:
        sbusTX.sbusChannels = sbusRX.sbusChannels
        rxTime = time.time()

    # system armed?
    if not sbusTX.sbusChannels[4]==sbusMAX: # set to neutral values
        sbusTX.sbusChannels[1] = sbusNEU # aileron
        sbusTX.sbusChannels[2] = sbusNEU # elevator
        if armFlag == 0: # notify disarmed state
            print('System disarmed')
            armFlag = 1 # set flag
    else:
        if armFlag == 1: # notify armed state
            print('System armed')
            armFlag = 0 # reset flag

    # set forward (Ele) level
    if sbusTX.sbusChannels[7]==sbusMAX:
        sbusTX.sbusChannels[2] = int(0.5*sbusTX.sbusChannels[8] + 906) # convert
level to sbus value on Ele
    if stepFlag == 0: # notify
start of step
        print('Elevator step started with value: ',str(sbusTX.sbusChannels[2]))
        stepFlag = 1 # set flag
    else:
        if stepFlag == 1: # notify
end of step
        print('Elevator step ended')
        stepFlag = 0 # reset
flag

    # sbus TX
    sbusTX.sbus_write()
    txTime = time.time()

    # log data
    if sbusTX.sbusChannels[6]==sbusMAX:
        gps.gps_logging()
        accel.accel_logging()
        sbusDL.sbus_data_log(sbusRX.sbusChannels,sbusTX.sbusChannels,rxTime,txTime)
        if logFlag == 0: # notify logging start
            print('Logging started')
            logFlag = 1 # set flag
    else:
        if logFlag == 1: # notify logging end
            print('Logging ended')

```

```
logFlag = 0 # reset flag
```

GPS handler

```
# gps_handler.py

# Created on:   Jan 30, 2020
# Modified:    Mar 2, 2020
# Author:      THolliday

# This script controls a GPS via gps3

#!/usr/bin/python3

## Imports
from gps3 import agps3
import time
import csv
import os

class GPS():
    def __init__(self,timestr):

        ## Setup GPS
        #print('Setting up GPS...')
        #os.system('sudo sh ./startup/gps_startup.sh')
        #print('GPS setup')

        ## initialize gps
        self.gps_socket = agps3.GPSDSocket() # set socket to default port
        self.data_stream = agps3.DataStream() # set up data stream

        ## create/open log csv file (with time in file name)
        self.gps_timestr = timestr
        with open('/home/pi/data_logs/gps_log_'+str(self.gps_timestr)+'.csv','w') as csvfile:
            # initialize csv
            gpslog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)
            # write column titles
            gpslog.writerow(['lat','long','alt','Tgps'])

        ## start gps comms
        self.gps_socket.connect() # connect to gps module on default port
        self.gps_socket.watch() # begin data monitoring

    def gps_logging(self):
        # check for new data and if so log the data

        for new_data in self.gps_socket:
            if new_data:
                self.data_stream.unpack(new_data) # separate gps data
                timelog = time.time() # capture new sample time

                # log new gps data
                with open('/home/pi/data_logs/gps_log_'+str(self.gps_timestr)+'.csv','a') as
csvfile:
                    # initialize csv
```

```

        gpslog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)
        # write lat, long, alt, and timestamp data

gpslog.writerow([str(self.data_stream.lat),str(self.data_stream.lon),str(self.data_stream.alt)
,str(timestamp)])

        break

```

Accelerometer handler

```

# accel_handler_v2.py

# Created on:   Feb 12, 2020
# Modified:    Feb 25, 2020
# Author:      THolliday

# This handler allows a RPi to capture and log
# data from a BMA280 accelerometer using the PIGPIO
# library.

#!/usr/bin/python3

## Imports
import time
import pigpio
import csv

class ACCEL():
    def __init__(self,timestr):
        ## Registers
        self.ACCD_X_LSB = 0x02 # LSB of x-accel measurement register
        PMU_RANGE = 0x0F      # measurement sensitivity register
        PMU_BW = 0x10         # measurement bandwidth register
        SOFTRESET = 0x14      # reset register
        OFC_SETTING = 0x37    # compensation targets register
        OFC_CTRL = 0x36       # compensation control register
        OFC_OFF_X = 0x38      # x-axis offset register
        OFC_OFF_Y = 0x39      # y-axis offset register
        OFC_OFF_Z = 0x3A      # z-axis offset register

        ## Commands
        RANGE_2G = 0x03      # measurement sensitivity
        BW_62_5Hz = 0x0B     # measurement bandwidth, 125-Hz sample rate

        ## Slave address
        bma280_address = 0x19

        ## Constants
        BUS = 1               # i2c buss
        self.RX_BYTES = 6    # number of bytes to receive
        self.numRead = 10    # number of readings

        ## Variables
        offsets = [0,0,0]    # axes offsets for fast compensation
        fcRes = 7.8125       # mg/LSB, based on min trigger amount
        self.errCount = 0    # read error count
        self.accelBuff = [0,0,0] # accel values [x,y,z]
        self.readNdx = 0    # samples/readings count

```

```

self.accel_accum = [0,0,0,0] # running sum for moving average
self.adxlReadings = [[0 for i in range(3)] for j in range(self.numRead)] #
initialize moving average vector

## Initialize log
self.accel_timestr = timestr
# create/open log csv file (with time in file name)
with open('/home/pi/data_logs/accel_log_' + str(self.accel_timestr) + '.csv','w') as
csvfile:
    # initialize csv
    accellog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)
    # write column titles
    accellog.writerow(['Xaccel','Yaccel','Zaccel','Taccel'])

## Initialize i2c
self.i2c = pigpio.pi()
self.bma280_handle = self.i2c.i2c_open(BUS,bma280_address) # open i2c comms with
BMA280
time.sleep(0.005)

## Setup BMA280
try:
    self.i2c.i2c_write_byte_data(self.bma280_handle,SOFTRESET,0xB6) # reset sensor
except:
    print('Accel comms failed. Retrying...')
    time.sleep(0.005)
    self.i2c.i2c_write_byte_data(self.bma280_handle,SOFTRESET,0xB6) # reset sensor

self.i2c.i2c_write_byte_data(self.bma280_handle,PMU_RANGE,RANGE_2G) # measurement
sensitivity
time.sleep(0.005)
self.i2c.i2c_write_byte_data(self.bma280_handle,PMU_BW,BW_62_5Hz) # bandwidth
time.sleep(0.005)

## Run fast compensation
#if input("Find axes compensations for accelerometer? (y/n) ") == 'y':
# indicate beginning of process to user
print('Hold drone level and motionless for compensation')
time.sleep(2)
print('Beginning fast compensation...')

# set compensation targets (0,0,+1) g
self.i2c.i2c_write_byte_data(self.bma280_handle,OFC_SETTING,(0x20|0x01))

# x-axis compensation
self.i2c.i2c_write_byte_data(self.bma280_handle,OFC_CTRL,0x20)
while not (0x10 & self.i2c.i2c_read_byte_data(self.bma280_handle,OFC_CTRL)):
    pass

# y-axis compensation
self.i2c.i2c_write_byte_data(self.bma280_handle,OFC_CTRL,0x40)
while not (0x10 & self.i2c.i2c_read_byte_data(self.bma280_handle,OFC_CTRL)):
    pass

# z-axis compensation
self.i2c.i2c_write_byte_data(self.bma280_handle,OFC_CTRL,0x60)
while not (0x10 & self.i2c.i2c_read_byte_data(self.bma280_handle,OFC_CTRL)):
    pass

# save x-offset

```

```

        (count,data) = self.i2c.i2c_read_i2c_block_data(self.bma280_handle,OFC_OFF_X,2)
        temp = list(data)
        offsets[0] =
int.from_bytes([temp[0],temp[1]],byteorder='little',signed=True)*(fcRes/256)/1000

        # save y-offset
        (count,data) = self.i2c.i2c_read_i2c_block_data(self.bma280_handle,OFC_OFF_Y,2)
        temp = list(data)
        offsets[1] =
int.from_bytes([temp[0],temp[1]],byteorder='little',signed=True)*(fcRes/256)/1000

        # save z-offset
        (count,data) = self.i2c.i2c_read_i2c_block_data(self.bma280_handle,OFC_OFF_Z,2)
        temp = list(data)
        offsets[2] =
int.from_bytes([temp[0],temp[1]],byteorder='little',signed=True)*(fcRes/256)/1000

        # write offsets to log
        with open('/home/pi/data_logs/accel_log_' + str(self.accel_timestr) + '.csv','a') as
csvfile:
            # initialize csv
            accellog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)
            # write accel and timestamp data

accellog.writerow([str(offsets[0]),str(offsets[1]),str(offsets[2]),'mg_offsets'])

        # denote completion of compensation
        print('Compensation finished & offsets recorded. Ready to fly!')

def accel_logging(self):
    # capture and convert accel values, and run a moving average filter over 10 samples

    # check accel values
    try:
        (count,data) =
self.i2c.i2c_read_i2c_block_data(self.bma280_handle,self.ACCD_X_LSB,self.RX_BYTES)
    except:
        count = 0

    if count == self.RX_BYTES:
        # convert tuple to list
        tempVal = list(data)

        # fill buffer [x,y,z]
        self.accelBuff[0] =
int.from_bytes([tempVal[0],tempVal[1]],byteorder='little',signed=True)
        self.accelBuff[1] =
int.from_bytes([tempVal[2],tempVal[3]],byteorder='little',signed=True)
        self.accelBuff[2] =
int.from_bytes([tempVal[4],tempVal[5]],byteorder='little',signed=True)

        # convert to g's
        self.accelBuff[0] = (self.accelBuff[0]/16384)
        self.accelBuff[1] = (self.accelBuff[1]/16384)
        self.accelBuff[2] = (self.accelBuff[2]/16384)

        # moving average filter
        for k in range(3):

```

```

        temp = self.accelBuff[k]
        temp /= self.numRead
        self.accel_accum[k+1] += temp
        temp = self.adxlReadings[self.readNdx][k]
        temp /= self.numRead
        self.accel_accum[k+1] -= temp
        self.adxlReadings[self.readNdx][k] = self.accelBuff[k]
        self.accelBuff[k] = self.accel_accum[k+1]

    # index check
    if not (self.readNdx >= self.numRead - 1):
        self.readNdx += 1      # increase readings count
    else:
        self.readNdx = 0      # reset count

    # log accel data
    timelog = time.time()
    with open('/home/pi/data_logs/accel_log_' + str(self.accel_timestr) + '.csv','a')
as csvfile:
        # initialize csv
        accellog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)
        # write accel and timestamp data

accellog.writerow([str(self.accelBuff[0]),str(self.accelBuff[1]),str(self.accelBuff[2]),str(ti
melog)])
    else:
        self.errCount += 1
        print('Total read errors: ',str(self.errCount))

```

SBUS RX handler (older version than previous)

```

# sbusRX_handler.py

# Created on:   Nov 4, 2019
# Modified:    Jan 20, 2020
# Author:      THolliday

# derived from:
#   sbusPythonDriver by Donald Simonet on Framagit
#   https://framagit.org/dsimonet/sbusPythonDriver

#   sbus_ultrasonic_v5 by THolliday
#       sbus_handler.c by Bhill & THolliday

# This script captures sbus frames and decodes them into individual channels.
# An sbus frame is made up of 25 packets, with the channels spread out over
# packets.

## Imports
import serial
import time

class SBUS_RX():
    def __init__(self, _uart_port = '/dev/ttyAMA0'):
        # sbus comms initialization

```

```

# RPi uart initialization
self.ser = serial.Serial(
    port = _uart_port,           # indicate UART port
    baudrate = 100000,          # sbus runs at 100k baud
    parity = serial.PARITY_EVEN, # set even parity
    stopbits = serial.STOPBITS_TWO, # set two stopbits
    bytesize = serial.EIGHTBITS, # set byte size to 8 bits
    timeout = 0                 # disable UART timeout
)

# set sbus constants
self.START_PACKET = 0x0F
self.END_PACKET = 0x00
self.SBUS_FRAME_LEN = 25      # full frame length is 25 packets
self.SBUS_CHAN_LEN = 16      # 16 individual channels
self.SBUS_BITS = 11          # number of bits in sbus packet

# variable initialization
self.isReady = True          # RX flag
self.lastFrameTime = 0       # new frame time
self.sbusFrameIn = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # captured
sbus RX frame
self.sbusFrame = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # endian-
swapped sbus RX frame
self.sbusChanIn = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # RX channels
self.sbusChannels = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # RX endian-
swapped channels

def chan_endian_swap(self):
    # swaps the bit order of the sbus channels

    for i in range(0,(len(self.sbusChanIn))):
        temp = 0
        bmask = 0x8000
        for j in range(0,16):
            if j<8:
                temp |= (self.sbusChanIn[i]&int(bmask))>>(15-(j*2)) # first byte
            else:
                temp |= (self.sbusChanIn[i]&int(bmask))<<((j*2)-15) # second byte
                bmask /= 2 # adjust mask
            temp >>= (16-self.SBUS_BITS) # account for number of
bits per packet
        self.sbusChannels[i] = temp # store channel value

def frame_endian_swap(self):
    # swaps the byte order of the sbus frame

    for i in range(0,len(self.sbusFrameIn)):
        temp = 0
        bmask = 0x80
        for j in range(0,8):
            if j<4:
                temp |= ((self.sbusFrameIn[i]&int(bmask))>>(7-(j*2))) # first nibble
            else:
                temp |= ((self.sbusFrameIn[i]&int(bmask))<<((j*2)-7)) # second nibble
                bmask /= 2 # adjust mask
            self.sbusFrame[i] = temp # store frame values

```



```

def sbus_read(self):
    # reads sbus values over the serial port on the RPi

    if self.ser.inWaiting() >= self.SBUS_FRAME_LEN*2: # load at least two full frames
        self.isReady = False # processing frames
        rxBuff = self.ser.read(self.ser.inWaiting()) # capture sbus packets
        self.numPackets = len(rxBuff) # save number of received packets

        # parse full frame
        for rr in range(0,self.SBUS_FRAME_LEN): # step through packets
            # look for end packet, working backwards
            #print(rxBuff)
            if rxBuff[len(rxBuff)-1-rr] == self.END_PACKET:
                # based on end packet, find start packet
                if rxBuff[len(rxBuff)-rr-self.SBUS_FRAME_LEN] == self.START_PACKET:
                    # frame is fully mapped and parity checked due to 8E2 format
                    # only need to remap the frame if it is different than the last
                    newFrame = rxBuff[len(rxBuff)-rr-self.SBUS_FRAME_LEN:len(rxBuff)-1-rr]
                    if not self.sbusFrameIn == newFrame: # check if new frame is diff
                        # than previous frame (save CPU cycles)
                            self.sbusFrameIn = newFrame # store new frame
                            self.sbus_decode() # decode new frame

                    # sbus frame succesfully captured and checked for changes
                    #self.lastFrameTime = time.time() # timestamp of frame capture
                    self.isReady = True # RX frame is ready
                    break

def sbus_decode(self):
    # decodes an sbus frame into 16 individual channels

    # Conversion parameters
    # chan number [1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16]
    packet_num = [0, 1, 3, 4, 5, 7, 8, 9, 11,
12, 14, 15, 16, 18, 19, 20]
    lowBS = [5, 2, 7, 4, 1, 6, 3, 0, 5,
2, 7, 4, 1, 6, 3, 0]
    lowbitmask = [0xE0, 0xFC, 0x80, 0xF0, 0xFE, 0xC0, 0xF8, 0xFF, 0xE0,
0xFC, 0x80, 0xF0, 0xFE, 0xC0, 0xF8, 0xFF]
    midBS = [3, 6, 1, 4, 7, 2, 5, 8, 3,
6, 1, 4, 7, 2, 5, 8]
    midbitmask = [0xFF, 0x1F, 0xFF, 0x7F, 0x0F, 0xFF, 0x3F, 0x07, 0xFF,
0x1F, 0xFF, 0x7F, 0x0F, 0xFF, 0x3F, 0x07]
    highBS = [0, 0, 9, 0, 0, 10, 0, 0, 0,
0, 9, 0, 10, 0, 0]
    highbitmask = [0x00, 0x00, 0x03, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
0x00, 0x03, 0x00, 0x00, 0x01, 0x00, 0x00]

    # swap packet (frame bytes) order
    self.frame_endian_swap()

    # decode frames
    for i in range(0,self.SBUS_CHAN_LEN):
        self.sbusChanIn[i] = ((self.sbusFrame[packet_num[i]]&highbitmask[i])<<highBS[i]) \
            + ((self.sbusFrame[packet_num[i]+1]&midbitmask[i])<<midBS[i]) \
            + ((self.sbusFrame[packet_num[i]+2]&lowbitmask[i])>>lowBS[i])

    # swap channel bit order
    self.chan_endian_swap()

```

SBUS TX handler (older version than previous)

```

# sbusTX_handler.py

# Created on:   Oct 29, 2019
# Modified:    Jan 20, 2020
# Author:      THolliday

# derived from:
#   sbusPythonDriver by Donald Simonet on Framagit
#   https://framagit.org/dsimonet/sbusPythonDriver

#   sbus_ultrasonic_v5 by THolliday
#   sbus_handler.c by Bhill & THolliday

# This script takes 16 individual channels and encodes them into an sbus frame.
# An sbus frame is made up of 25 packets, with the channels spread out over
# packets.

## Imports
import serial
import time

class SBUS_TX():
    def __init__(self, _uart_port = '/dev/ttyAMA0'):
        # sbus comms initialization

        # RPi uart initialization
        self.ser = serial.Serial(
            port = _uart_port,           # indicate UART port
            baudrate = 100000,          # sbus runs at 100k baud
            parity = serial.PARITY_EVEN, # set even parity
            stopbits = serial.STOPBITS_TWO, # set to two stopbits
            bytesize = serial.EIGHTBITS, # set byte size to 8 bits
            timeout = 0                  # disable UART timeout
        )

        # set sbus constants
        self.START_PACKET = 0xF0        # first packet
        self.END_PACKET = 0x00          # last packet
        self.sbusNbits = 11             # bits per packet
        self.SBUS_FRAME_LEN = 25        # packets per frame

        # variable initialization
        self.lastFrameTime = 0          # frame time
        self.numPackets = 0             # number of transmitted packets
        self.sbusChannels = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # RX channels
        self.sbusChanBitSwap = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # RX endian-
swapped channels
        self.sbusPackets = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # TX packets
        self.sbusFrame = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] # TX endian-
swapped frame

    def chan_endian_swap(self):
        # swaps the bit order of the sbus channels

        for i in range(0,(len(self.sbusChannels)-1)):
            temp = 0

```

```

        bmask = 0x8000
        for j in range(0,16):
            if (j<8):
                temp |= (self.sbusChannels[i]&int(bmask))>>(15-(j*2))    # first byte
            else:
                temp |= (self.sbusChannels[i]&int(bmask))<<((j*2)-15)    # second byte
                bmask /= 2                                                # adjust mask
            temp >>= (16-self.sbusNbits)                                    # shift to account for
number of bits per packet
        self.sbusChanBitSwap[i] = temp                                    # store channel

```

```

def frame_endian_swap(self):
    # swaps the byte order of the sbus frame

```

```

    for i in range(0,self.SBUS_FRAME_LEN):
        temp = 0
        bmask = 0x80
        for j in range(0,8):
            if (j<4):
                temp |= (self.sbusPackets[i]&int(bmask))>>(7-(j*2))    # first nibble
            else:
                temp |= (self.sbusPackets[i]&int(bmask))<<((j*2)-7)    # second nibble
                bmask /= 2                                                # adjust mask
        self.sbusFrame[i] = temp                                          # store frame

```

```

def sbus_encode(self):
    # encode 16 channels into single sbus frame

```

```

    # Conversion parameters
    # Frame number  [1      2      3      4      5      6      7      8      9
10     11     12     13     14     15     16     17     18     19     20     21
22]
    chan1num =    [0,    0,    1,    2,    2,    3,    4,    5,    5,
6,    7,    8,    8,    9,    10,   10,   11,   12,   13,   13,   14,
15]
    chan2num =    [0,    1,    2,    2,    3,    4,    5,    5,    6,
7,    8,    8,    9,    10,   10,   11,   12,   13,   13,   14,   15,
16]
    chan1mask =   [0x00, 0xE0, 0xFC, 0x00, 0x80, 0xF0, 0xFE, 0x00, 0xC0,
0xF8, 0xFF, 0x00, 0xE0, 0xFC, 0x00, 0x80, 0xF0, 0xFE, 0x00, 0xC0, 0xF8,
0xFF]
    chan2mask =   [0xff, 0x1F, 0x03, 0xFF, 0x7F, 0x0F, 0x01, 0xFF, 0x3F,
0x07, 0x00, 0xFF, 0x1F, 0x03, 0xFF, 0x7F, 0x0F, 0x01, 0xFF, 0x3F, 0x07,
0x00]
    chan1BS =     [0,    5,    2,    0,    7,    4,    1,    0,    6,
3,    0,    0,    5,    2,    0,    7,    4,    1,    0,    6,    3,
0]
    chan2BS =     [3,    6,    9,    1,    4,    7,    10,   2,    5,
8,    0,    3,    6,    9,    1,    4,    7,    10,   2,    5,    8,
0]

```

```

    # swap bit order of channels
    self.chan_endian_swap()

```

```

    # add startbyte
    self.sbusPackets[0] = self.START_PACKET

```

```

    # add in channels (note that the channels are spread across multiple frames)
    for rr in range(0,22):

```

```

        tempFrame = 0
        tempFrame = (self.sbusChanBitSwap[chan1num[rr]]<<chan1BS[rr]) & chan1mask[rr]    #
add in first channel data
        tempFrame |= (self.sbusChanBitSwap[chan2num[rr]]>>chan2BS[rr]) & chan2mask[rr]    #
add in second channel data
        self.sbusPackets[rr+1] = tempFrame      # store frame

    # add flags & digital
    self.sbusPackets[23] = 0

    # add endbyte
    self.sbusPackets[24] = self.END_PACKET

    # swap byte order of frames
    self.frame_endian_swap()

def sbus_write(self):
    # transmit the full sbus frame over UART

    # encode sbus frame
    self.sbus_encode()

    # transmit the frame
    self.numPackets = self.ser.write(self.sbusFrame)
    #self.lastFrameTime = time.time()

```

SBUS data logging handler (older version than previous)

```

# sbusDL_handler.py

# Created on:   Jan 8, 2020
# Modified:    Mar 2, 2020
# Author:      THolliday

# This script holds dedicated functions for logging sbus data
# on a Raspberry Pi Zero.

## Imports
import time
import csv

class SBUS_DL():
    def __init__(self, timestr):
        # sbus data logging initialization

        # create/open log csv file (with time in file name)
        self.sbus_timestr = timestr
        with open('/home/pi/data_logs/sbus_log_'+str(self.sbus_timestr)+'.csv','w') as
csvfile:
            # initialize csv
            sbuslog = csv.writer(csvfile,delimiter = ' ',quotechar = '|',quoting =
csv.QUOTE_MINIMAL)

            # add column titles

#sbuslog.writerow(['RX/TX','Ch0','Ch1','Ch2','Ch3','Ch4','Ch6','Ch7','Ch8','Tsbus'])

```

```

sbuslog.writerow(['RX/TX', 'Thr', 'Ail', 'Ele', 'Rud', 'ARM', 'LOG', 'vEN', 'VEL', 'Tsbust'])

def sbus_data_log(self, rx, tx, rxTime, txTime):
    # logs sbus channel values of RX/TX

    with open('/home/pi/data_logs/sbus_log_'+str(self.sbus_timestr)+'.csv', 'a') as
csvfile:
    # initialize csv
    sbuslog = csv.writer(csvfile, delimiter = ' ', quotechar = '|', quoting =
csv.QUOTE_MINIMAL)

    # log RX channels

sbuslog.writerow(['RX', str(rx[0]), str(rx[1]), str(rx[2]), str(rx[3]), str(rx[4]), str(rx[6]),
str(rx[7]), str(rx[8]), str(rxTime)])

    # log TX channels

sbuslog.writerow(['TX', str(tx[0]), str(tx[1]), str(tx[2]), str(tx[3]), str(tx[4]), str(tx[6]),
str(tx[7]), str(tx[8]), str(txTime)])

```

Appendix H: MITM Code – MSP430G2553 & Ultrasonic Version

Main script

```

/* main.c
 *
 * Created on: Oct 10, 2019
 * Authors: B Hill, T Holliday
 *
 * SBUS read/write with single ultrasonic
 * for ENGR 5940 presentation
 */

// Inclusions
#include <msp430.h>
#include <math.h>
#include "sbus_handler.h"
#include "ultrasonic_handler.h"

// Defines
void rw_flash(char * , int, int , int );

// Start main.c
int main(void){
    // MSP430 initialization
    WDTCTL = WDTPW | WDTHOLD;          // Stop watchdog timer

    // Variables
    unsigned int k;
    unsigned int channel_vals[] =
{992,992,992,992,992,992,992,992,992,992,992,992,992,992,992,992,992,992,992,992,992}; // initialize all
channels to neutral values
    int channel_ret[16], calflash;
    char sbus_buffer[25], rx_sbus_buffer[25], calchar[2];
    unsigned char us_dist_flag = 0, usflag = 0;
    unsigned int us_dist = 0;

    // Initialize ultrasonic
    init_us_trigger();

    // Initialize clock & sbus
    rw_flash(calchar, 0,0,2);          // read time value from the flash memory to sync with sbus
protocol rx
    calflash = calchar[0];
    calflash <<= 8;
    calflash += calchar[1];
    sbus_init(calflash);
    sbus_frame_maker(channel_vals, sbus_buffer,0); // create sbus frame

    for (k=0;k<25;k++){
        tx_data_str[k] = 0x3C;          // fill sbus transmit frame
    }

    byte_endian_swap(tx_data_str,25); // perform 8-bit endian swap
    tx_data_str[0] = 0xAA;             // set values of the first two frames
    tx_data_str[1] = 0xAA;

    while(1){
//         if (usflag==0){              // ready to trigger ultrasonic?

```

```

//      trigger_us();           // trigger ultrasonic
//      usflag = 1;           // set flag
//      }else if (usflag==1){  // echo recieved?
//      trigger_distance(&us_dist); // capture distance
//      us_dist_flag = 1;     // set dist flag
//      usflag = 0;         // reset flag
//      }

    if ((eos_flag==1)&&rx_flag>18){           // sbus packet is ready
        byte_endian_swap(rx_data_str,25);   // perform 8-bit endian swap
        for (k=0;k<25;k++){
            rx_sbus_buffer[k] = rx_data_str[k]; // store the current rx value in
the sbus buffer
        }
        eos_flag=0;                         // reset flag
        sbus_frame_reader(channel_ret,rx_sbus_buffer); // read and separate channels from
rx sbus
        for (k=0;k<16;k++){
            channel_vals[k] = channel_ret[k]; // plot out the channel values
returned from the frame reader
        }

        // Ultrasonic collision detection
        if ((channel_vals[6]!=172)&&(us_dist_flag==1)&&(channel_vals[2]>992)){ // //
ultrasonic enabled while in motion?

            if ((us_dist<500)&&(us_dist>250)){ // slow forward speed
                channel_vals[2] = channel_vals[2]*2/3;
            }else if (us_dist<=250){ // stop forward motion
                channel_vals[2] = 992;
            }

            // check that new channel value is within sbus bounds
            if ((channel_vals[2]<172)||((channel_vals[2]>1811))){
                channel_vals[2] = 992; // out of bounds, stop
motion
            }
            us_dist_flag = 0; // reset ultrasonic values
//      us_dist = 0;
        }
        // end ultrasonic collision check

        // Check for system enable
        if (channel_vals[4] != 1811){ // send neutral values if system is not armed
            channel_vals[2] = 992; // elevator (forward/backward)
            channel_vals[1] = 992; // aileron (left/right)
        }

        sbus_frame_maker(channel_vals, sbus_buffer,0); // create the sbus frame with the
new/same channel values
        for (k=0;k<25;k++){
            tx_data_str[k] = sbus_buffer[k]; // set the tx value to the value
in the sbus buffer
        }
        byte_endian_swap(tx_data_str,25); // perform 8-bit endian swap
        sbus_write_fast_string(0,25);
        rx_flag = 0; // reset flags
    }
    else if (eos_flag==2){ // error in receiving sbus packet

```

```

        sbus_write_fast_string(0,25);           // send out zeros over sbus
        eos_flag = 0;                          // reset flags
        rx_flag = 0;
    }
}
} // end main()

```

```

void rw_flash(char * data_vec, int mem_mode, int stnum, int lennum){
/* looks at the timer oscillator on the board
 * and calculates a time difference constant to
 * better synchronize the clocks for reading,
 * writing, and sending sbus protocol
 */

/* mem_mode = indicates reading (0) or writing (non 0)
 * stnum =
 * lennum = maximum length of the value being read from the
 *          flash memory
 */

// Variables
char *Flash_ptr;           // pointer to value in flash memory
volatile char temp, k;

Flash_ptr = (char *) (0x1040+stnum);

if (mem_mode){ //write to the flash memory
    FCTL1 = FWKEY + ERASE;           // Set Erase bit
    FCTL3 = FWKEY;                  // Clear Lock bit
    *Flash_ptr = 0;                  // Dummy write to erase Flash segment

    FCTL1 = FWKEY + WRT;            // Set WRT bit for write operation
    for (k=0;k<lennum;k++)
        *Flash_ptr++=data_vec[k];
    FCTL1 = FWKEY;                  // Clear WRT bit
    FCTL3 = FWKEY + LOCK;           // Set LOCK bit
}
else if (mem_mode==0){ // Read from flash memory
    for (k=0;k<lennum;k++)
        data_vec[k]=*Flash_ptr++;
}
}

```

Ultrasonic handler (older version than previous)

```

/*
 * ultrasonic_handler.c
 *
 * Created on: Feb 19, 2019
 * Author: tholliday
 */

// Inclusions
#include <msp430.h>

// Defines

```



```

#define TRIG0    BIT3
#define ECH00    BIT4
#define TRIGDIR  P2DIR
#define TRIGOUT  P2OUT
#define ECHODIR  P2DIR
#define ECHOSEL  P2SEL
#define ECHOIE   P2IE
#define ECHOIES  P2IES
#define ECHOIFG  P2IFG

// Global variables
unsigned int Distclicks;
int UPCOUNTSTATE;
/* UPCOUNTSTATE VALUES
 *    0 - system ready/not running
 *    1 - trigger signal sent, waiting for trigger timing
 *    2 - end trigger timing, initialize echo receiving, waiting for echo
 *    3 - echo received and time value acquired
 *    4 - (or greater) timeout has occurred.
 */

void init_us_trigger(void){
    // Initialize ultrasonic pins
    TRIGDIR |= TRIG0;           // Set pin 2.3 as a trigger for the ultrasonic sensor
    TRIGOUT &=~ TRIG0;         // Initialize 2.3 as low for the trigger (trigger is
high)
    ECHOIES &=~ ECH00;         // set echo hardware interrupt to lo/high edge
    ECHOIE  |= ECH00;         // set pin 2.4 as echo hardware interrupt

    // Enable timer and interrupt
    _BIS_SR(GIE);              // Enable interrupts for the Port Triggering
    TA1CTL = (TASSEL_2 + ID_3 + MC_2); // configure interrupt timer
    TA1CCR0 = 42000;

    // Initialize ultrasonic state
    UPCOUNTSTATE = 0;
} // end init_us_trigger

void trigger_us(void){
    if (UPCOUNTSTATE==0){
        UPCOUNTSTATE = 1;      // set state to "signal sent"
        TRIGOUT |= TRIG0;     // Trigger the output to start the signal
        ECHOIES &=~ ECH00;    // set lo/hi edge on echo interrupt
        __delay_cycles(160);   // approximately 10us wait
        TRIGOUT &=~ TRIG0;    // End the trigger sequence
    }
    if (UPCOUNTSTATE>3){
        TA1CCTL0 &=~ CCIE;    // disable timer interrupt
    }
} // end trigger_us

void trigger_distance(unsigned int *dist_out){

    if ((UPCOUNTSTATE==3)){ // state: echo received & recorded
        UPCOUNTSTATE = 0;    // reset state
        dist_out[0] = (Distclicks/30e5)*343000; // convert and output the value as a
distance
    }
}

```

```

    if (UPCOUNTSTATE>=4){                // state: timeout
        UPCOUNTSTATE = 0;                 // reset state
        TA1CCTL0 &=~ CCIE;               // disable timer interrupt
        dist_out[0] = 0xFFFF;
    }
} // end trigger_distance

#pragma vector=PORT2_VECTOR
__interrupt void Port_2(void)
{ // when the echo is captured the value is outputed in Distclicks
  if (UPCOUNTSTATE<4){
    if (ECHOIES & ECH00){                // End of echo time
        ECHOIES &=~ ECH00;               // Set lo/hi edge trigger
        Distclicks = TA1R;               // store value in Distclicks
        UPCOUNTSTATE = 3;                // set state to "time value acquired"
        TA1CCTL0 &=~ CCIE;               // disable timer interrupt
    }else{                                // Beginning of echo time
        ECHOIES |= ECH00;                 // set hi/lo edge trigger
        TA1R = 0;                          // clear distance register
        UPCOUNTSTATE = 2;                  // set state to "waiting"
        TA1CCTL0 = CCIE;                  // enable timer interrupt
    }
  }
  ECHOIFG &=~ ECH00;                     // reset echo interrupt flag
} // end echo received interrupt

#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMERA1_ISR (void)
{ // timer that ends echo receiving if it is taking too long
  if (UPCOUNTSTATE==2){                 // Timeout has occurred
      UPCOUNTSTATE = 4;                   // set state to "timeout"
  }else{
      UPCOUNTSTATE++;
  }
} // end timer interrupt

/*
 * ultrasonic_header.h
 *
 * Created on: Feb 19, 2019
 * Author: tholliday
 */

#ifndef ULTRASONIC_HANDLER_H_
#define ULTRASONIC_HANDLER_H_

void init_us_trigger(void);
void trigger_us(void);
void trigger_distance(unsigned int *);

#endif /* ULTRASONIC_HANDLER_H_ */

```

SBUS handler

```

/*
 * sbus_handler.c
 *
 * Created on: Oct. 1, 2018; modified: Jan. 10, 2019
 * Authors: BHill, THolliday
 */

// Inclusions
#include "msp430.h"

// Defines
#define sbus_max 99

// Global Variables
unsigned int tout_counter = 0;
unsigned char rx_last = 0xff,tx_data_str[sbus_max], rx_data_str[sbus_max], rx_flag = 0,
dec_str[6], eos_flag = 0;
int tx_ptr,e_tx_ptr;

/* Defined Functions
 * endian_swap()
 * byte_endian_swap()
 * sbus_frame_maker()
 * sbus_frame_reader()
 * sbus_init()
 * sbus_write_fast_string()
 * TX & RX sbus timers
 */

void endian_swap(int * valsin,int nbitsswap, int numvals){
    /* takes the inputed 16-bit string and switches
    * the value from little-endian to big-endian
    * format
    */

    // Variables
    int i,n;
    unsigned int tempval;
    unsigned int bmask;

    for (i=0;i<numvals;i++){
        tempval = 0;
        bmask = 0x8000;
        for(n=0;n<16;n++){
            if (n<8){
                tempval |= ((valsin[i]&bmask)>>(15-(n*2)));
            }
            else{
                tempval |= ((valsin[i]&bmask)<<((n*2)-15));
            }
            bmask /= 2;
        }
        tempval >>= (16-nbitsswap);
        valsin[i] = tempval;
    }
} // end endian_swap

```

```

void byte_endian_swap(char * valsin, int numvals){
    /* takes the inputed byte (8-bits) and switches
    * the value from little-endian to big-endian
    * format
    */

    // Variables
    int i,n;
    char tempval;
    char bmask;

    for (i=0;i<numvals;i++){
        tempval = 0;
        bmask = 0x80;
        for(n=0;n<8;n++){
            if (n<4){
                tempval |= ((valsin[i]&bmask)>>(7-(n*2)));
            }
            else{
                tempval |= ((valsin[i]&bmask)<<((n*2)-7));
            }
            bmask /= 2;
        }
        valsin[i] = tempval;
    }
} // end byte_endian_swap

void sbus_frame_maker(unsigned int * channel_in, char * sbus_frame_out, char sbus_last){
    /* Outlines and creates the 22 frames in sbus protocol
    * as well as setups the placement of the individual
    * channels in the frame
    */

    // Variables
    int i;
    unsigned int tempchar;
    // Frame number
    //
    //      {1      2      3      4
    // 5      6      7      8      9      10
    // 11     12     13     14     15     16
    // 17     18     19     20     21     22}
    char channel1_num[] = {0,      0,      1,      2,      2,      2,
    3,      4,      5,      5,      6,      7,
    8,      8,      9,      10,     10,     11,
    12,     13,     13,     14,     15};
    char channel2_num[] = {0,      1,      2,      2,      2,      3,
    4,      5,      5,      6,      7,      8,
    8,      9,      10,     10,     11,     12,
    13,     13,     14,     15,     16};
    char chan1mask[] = {0x00, 0xE0, 0xFC, 0x00, 0x80, 0xF0, 0xFE, 0x00, 0xC0,
    0xF8, 0xFF, 0x00, 0xE0, 0xFC, 0x00, 0x80, 0xF0, 0xFE, 0x00, 0xC0, 0xF8,
    0xFF};
    char chan2mask[] = {0xff, 0x1F, 0x03, 0xFF, 0x7F, 0x0F, 0x01, 0xFF, 0x3F,
    0x07, 0x00, 0xFF, 0x1F, 0x03, 0xFF, 0x7F, 0x0F, 0x01, 0xFF, 0x3F, 0x07,
    0x00};
    char chan1bitshift[] = {0,      5,      2,      0,      0,      7,
    4,      1,      0,      6,      3,      0,
    0,      5,      2,      0,      7,      4,
    1,      0,      6,      3,      0};

```

```

char chan2bitshift[]   = {3,           6,           9,           1,           4,
    7,           10,          2,           5,           8,           0,
    3,           6,           9,           1,           4,           7,
    10,          2,           5,           8,           0};

endian_swap(channel_in,11,16);    // perform endian swap on the channel values
sbus_frame_out[0] = 0xF0;        // create the sbus output frame
for (i=0;i<22;i++){
    tempchar = 0;
    tempchar = (channel_in[channel1_num[i]]<<chan1bitshift[i])&chan1mask[i];
    tempchar |= (channel_in[channel2_num[i]]>>chan2bitshift[i])&chan2mask[i];
    sbus_frame_out[i+1] = tempchar;
}
sbus_frame_out[23] = sbus_last;   // set the last values of the sbus frame
sbus_frame_out[24] = 0;
endian_swap(channel_in,11,16);    // perform endian swap on entire sbus frame
} // end sbus_frame_maker

int sbus_frame_reader(unsigned int * channel_out, char * sbus_frame_in){
/* reads in sbus frame and maps out the channels
 * for possible modification based on distance
 * readings
 */

// Variables
int i;
// channel number      {1,      2,      3,      4,      5,
    6,      7,      8,      9,      10,      11,      12,      13,      14,      15,      16}
char byte_num[16]     = {0,      1,      3,      4,      5,
    7,      8,      9,      11,      12,      14,
    15,      16,      18,      19,      20};
char lowbitshift[16]  = {5,      2,      7,      4,      1,
    6,      3,      0,      5,      2,      7,
    4,      1,      6,      3,      0};
char lowbitmask[16]   = {0xE0, 0xFC, 0x80, 0xF0, 0xFE, 0xC0, 0xF8, 0xFF, 0xE0,
    0xFC, 0x80, 0xF0, 0xFE, 0xC0, 0xF8, 0xFF};
char midbitshift[16]  = {3,      6,      1,      4,      7,
    2,      5,      8,      3,      6,      1,
    4,      7,      2,      5,      8};
char midbitmask[16]   = {0xFF, 0x1F, 0xFF, 0x7F, 0x0F, 0xFF, 0x3F, 0x07, 0xFF,
    0x1F, 0xFF, 0x7F, 0x0F, 0xFF, 0x3F, 0x07};
char highbitshift[16] = {0,      0,      9,      0,      9,
    10,      0,      0,      0,      0,      9,
    0,      0,      10,      0,      0};
char highbitmask[16]  = {0,      0,      0x03, 0,      0,
    0x01, 0,      0,      0,      0x03, 0,
    0,      0x01, 0,      0};

if(sbus_frame_in[0]!=0xF0){
    return 1; // This frame is not starting with the
correct value
}
for(i=0;i<16;i++){
    channel_out[i] =
((sbus_frame_in[byte_num[i]]&highbitmask[i])<<highbitshift[i])+((sbus_frame_in[byte_num[i]+1]&
midbitmask[i])<<midbitshift[i])+((sbus_frame_in[byte_num[i]+2]&lowbitmask[i])>>lowbitshift[i])
;
}
endian_swap(channel_out,11,16);

```

```

    return 0;
} // end sbus_frame_reader

void sbus_init( int calval ){
    /* initializes the MSP430 to read in sbus over the serial pins
     * and enables various timers and interrupts for possible use
     */

    // Variables
    volatile int temp=0;
    // Set baud rate to 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400,
    460800, 921600
    // use index of 0 1 2 3... corresponding to the rates above
    long cal_temp;

    cal_temp = calval;
    cal_temp *= 118;
    cal_temp /= 1000;
    BCSCCTL1 = CALBC1_16MHZ; // Set DCO
    DCOCTL = CALDCO_16MHZ;
    P1SEL |= (BIT1+BIT2); // P3.4,5 = USCI_A0 TXD/RXD
    P1SEL2 |= (BIT1+BIT2);
    DCOCTL = 0; // Select lowest DCOx and MODx settings
    UCA0CTL0 |= UCSPB + UCPEN + UCPAR; // 2 Stop Bits, Enable Parity, Even Parity
    UCA0CTL1 |= UCSSEL_2; // SMCLK
    UCA0BR0 = cal_temp; // 16MHz/160 = 100kbs SBUS baud //115 for MSP1
    // 128 because of two stop bits Needs to be calibrated to the individual oscillator. can
    be between 118 and 136
    UCA0BR1 = 0; // 100kbs
    UCA0MCTL = UCBSR0; // Modulation UCBSRx = 1
    UCA0CTL1 &= ~UCSWRST; // **Initialize USCI state machine**
    IE2 |= UCA0RXIE; // Enable USCI_A0 RX interrupt
    CCTL0 = CCIE; // CCR0 interrupt enabled
    CCR0 = 50000;
    TACTL = TASSEL_2 + MC_2; // SMCLK, contmode

    __bis_SR_register(GIE); // interrupts enabled
} // end sbus_init

void sbus_write_fast_string(int vals, int vale){
    /* using values from tx_data_str a string
     * is written for the new sbus transmit
     * protocol
     */

    // Variables
    tx_ptr = vals; // vals is starting pointer
    e_tx_ptr = vale; // vale is the ending value

    UCA0TXBUF = tx_data_str[tx_ptr]; // fill sbus buffer with values from tx_data_str
    IE2 |= UCA0TXIE; // Uses interrupts to send out bytes
} // end sbus_write_fast_string

#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
    tout_counter++;
    if (tout_counter>11)

```

```

        tout_counter = 11;
        CCR0 += 50000;
    }

#pragma vector = USCIAB0TX_VECTOR
__interrupt void USCI0TX_ISR(void)
{
    if (IE2&UCA0TXIE){
        //portion of sbus_write_fast_string
        tx_ptr++;
        if (tx_ptr<e_tx_ptr)
            UCA0TXBUF = tx_data_str[tx_ptr];
        else{
            IE2 &= ~UCA0TXIE;
        }
    }
}

// Place data in RX-buffer and set flag
#pragma vector = USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
    volatile char temp;
    if(IFG2 & UCA0RXIFG){ // Receive data on
        sbus
        if (tout_counter>10){
            tout_counter = 0;
            rx_flag = 0;
            eos_flag = 1;
        }

        rx_data_str[rx_flag] = UCA0RXBUF;
        rx_flag++;
        if (rx_flag>sbus_max){ // maximum of
            characters starts at the beginning again
            rx_flag = 0;
            eos_flag = 2;
        }
    }
}

/*
 * SBUS_handler.h
 *
 * Created on: Jan. 10, 2019
 * Author: Bhill, tholliday
 */

#ifndef SBUS_HANDLER_H_
#define SBUS_HANDLER_H_

extern unsigned char tx_data_str[24], rx_data_str[24],rx_flag ,dec_str[7],eos_flag;

void endian_swap(int *,int, int);
void byte_endian_swap(char *, int);
void sbus_frame_maker(unsigned int *, char *,char);
int sbus_frame_reader(unsigned int *, char *);

```

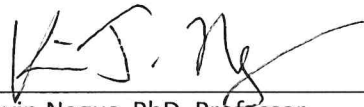
```
void sbus_init(int);  
void sbus_write_fast_string(int,int);  
  
#endif /* SBUS_HANDLER_H_ */
```


SIGNATURE PAGE

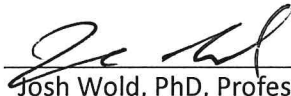
This is to certify that the thesis prepared by Tyler Holliday entitled "Modeling and Prototyping a Modular, Low-Cost Collision Avoidance System for UAVs" has been examined and approved for acceptance by the Department of Electrical Engineering, Montana Technological University, on this 22nd day of July, 2020.



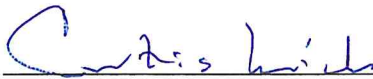
Bryce Hill, PhD, Associate Professor
Department of Electrical Engineering
Chair, Examination Committee



Kevin Negus, PhD, Professor
Department of Electrical Engineering
Member, Examination Committee



Josh Wold, PhD, Professor
Department of Electrical Engineering
Member, Examination Committee



Curtis Link, PhD, Professor Emeritus
Department of Geophysical Engineering
Member, Examination Committee