Zach Drone

Justin Armenta
Elias Duque
Jonah Morris
Grayson Vansickle

FINAL REPORT

REVISION – 1.0
5 December 2018

Table of Contents

Zach Drone

Elias Duque
Grayson Vansickle
Justin Armenta
Jonah Morris

Concept of Operations

REVISION – 1.0
5 December 2018

CONCEPT OF OPERATIONS
FOR
Zach Drone

PREPARED BY:
TEAM <17>

APPROVED BY:

_____
Project Leader Date

_____
Prof. S. Kalafatis Date

_____
T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
0.0	9/6/18	Grayson Vansickle		Draft Release
0.2	9/21/18	Justin Armenta		Added Introduction and Scenario(s) sections
0.4	9/24/18	Jonah Morris		Added Executive Summary and Operating Concept sections
0.6	9/24/18	Elias Duque		Added Impacts section
0.8	9/26/18	Grayson Vansickle		Added Analysis section
1.0	10/3/18	Grayson Vansickle		Edited and Formatted
1.2	12/5/18	Grayson Vansickle		Updated information

Table of Contents

List of Tables

No tables at this time.

List of Figures

1. Executive Summary

Navigating a new building as large as Zachry by yourself can be difficult. Traditional tours with a tour guide do not always work around an individual's schedule. With the Zach Drone, tourists can take a tour anytime to anywhere inside of Zachry. Using a phone application, tourists will be able to tell the drone to take them to one of three different pre-programmed automated tours. While on the tour the drone will narrate important information about the building such as where it is going next and details of different art pieces in the building. After twenty minutes of use, the drone will return to the tour starting point where it can be recharged for the next set of tourists.

2. Introduction

Zachry is the newest and most innovative of the Texas A&M engineering buildings. As the flagship of all the engineering buildings it is presented as a mixture of technology, innovation, and convenience. Since the building exemplifies these values, the tours given in the facility should convey these same values. This bold and innovative style is what inspired us to make a drone tour inside of the Zachry building. Guests will not have to wait for scheduled tours by people, but instead can ask for a quick and easy drone tour where the drone will tell customers key information about the building while also guiding its prospective tourists around the complex.

2.1. Background

Currently, tours are given by tour guides provided by the Texas A&M Engineering department's communication team. It usually takes around 30 minutes for these employees to give a tour, and they have to give these tours at least twice a day. These are hours wasted every day for them where they are giving tours instead of doing other helpful jobs for the department. It also can be an inconsistent tour if the tour guide forgets about any of the key locations or information about the building. Also, the tour times would not be flexible since a human has to run on a scheduled time to give the tour every day with no room for flexibility. A drone tour of Zachry would solve these problems by offering a more customizable and innovative experience. The drone would free up employees to do other more important jobs that the department is working on. It would also allow tourists to choose the destinations they would like to see and have all the information about that destination told to them by the drone. Lastly, it would allow customers to choose a more convenient time for them to get their tour instead of being told by Texas A&M when they can come see the tour.

2.2. Overview

The drone will consist of a custom built drone made by a previous engineering team, Raspberry Pi 3 B microcontroller, a speaker, six ultrasonic sensors, and a camera. The Raspberry Pi will be used to process information received by the ultrasonic sensors and the WiFi connection from the tourist's device. The microcontroller will connect to the camera to make sure the tourist's stay close while also guiding the tourist towards the next destination in the tour. The ultrasonic sensors will monitor how close the drone gets to obstacles while the microcontroller will give commands to the drone to either stop or change its course of direction if an obstacle gets too close. A speaker will be used to give information to the

customer about different destination points while also explaining where the drone is going next.

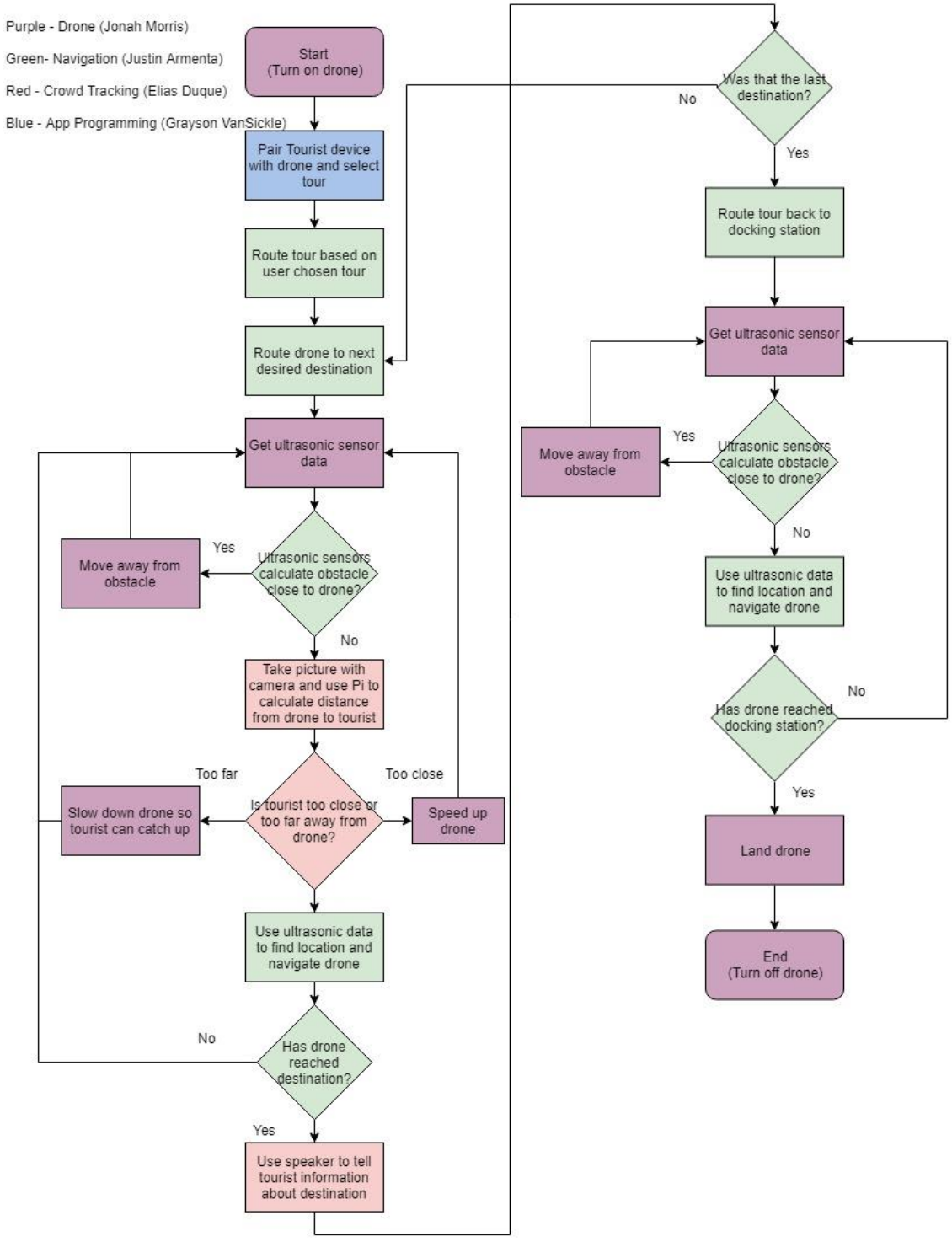


Figure 1: Drone Logic Overview

Figure 1 shown above displays the logic that the drone will follow throughout the process from the beginning of the tour to the end. Every function block is color coded based

on who will be in charge of that particular function. As shown above Jonah (Purple) will be in charge of the placement and function of the components on the drone, Grayson (Blue) will be in charge of creating an app for the WiFi connection between the drone and the tourist, Elias (Red) will be in charge of making sure the drone stays within a certain distance of the tourists, and Justin (Green) will be in charge of making sure the drone knows how to get from one destination to the next.

2.3. Referenced Documents and Standards

https://federaldroneregistration.com/?gclid=EAIaIQobChMIq9mDtIjN3QIVIIlpCh0-KAneEAAYASAAEgLrHPD_BwE
https://federaldroneregistration.com/know-before-you-fly
http://knowbeforeyoufly.org/
https://www.scienceabc.com/innovation/what-is-the-range-of-bluetooth-and-how-can-it-be-extended.html
https://zachry.tamu.edu/tours/
https://zachry.tamu.edu/

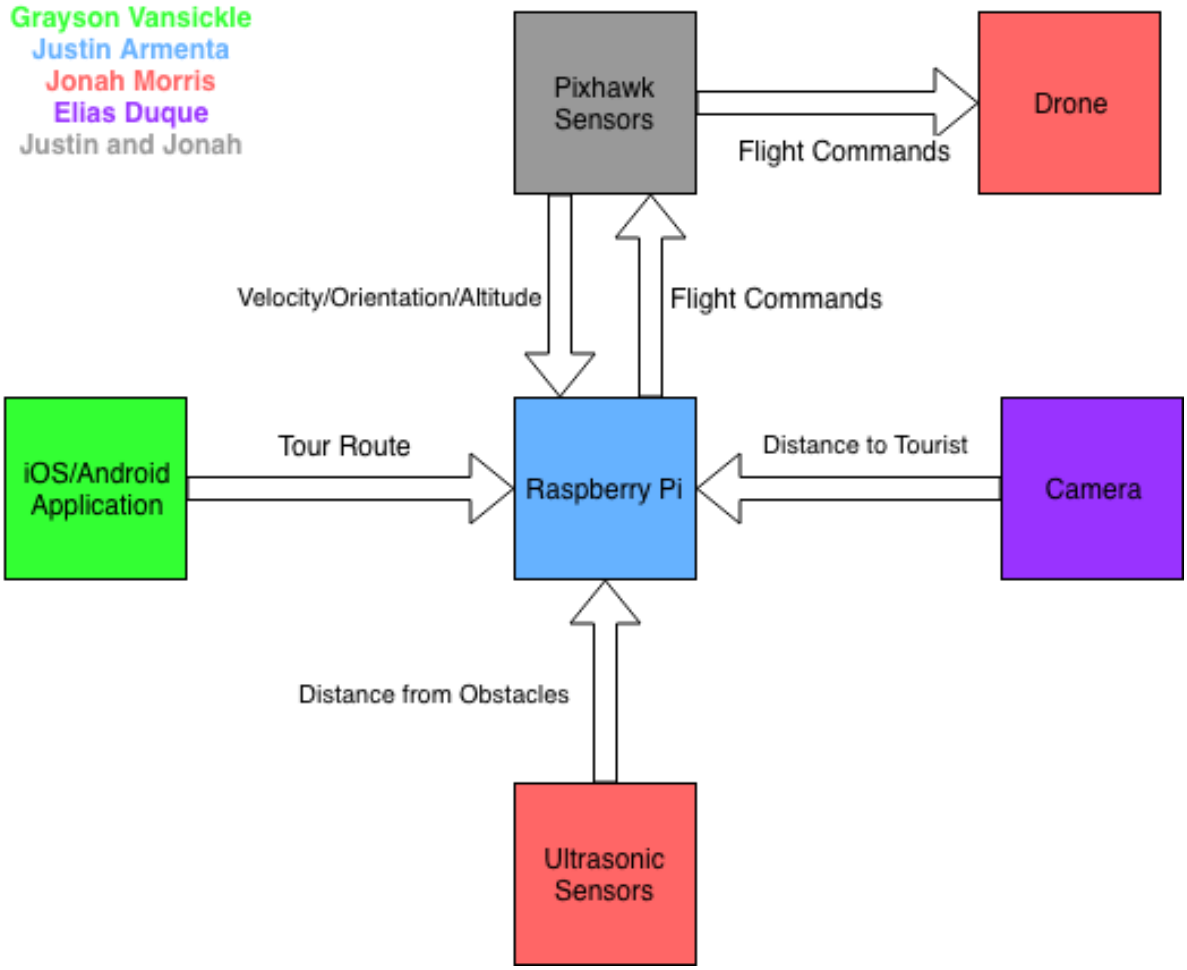


Figure 2: Overall System Diagram

3. Operating Concept

3.1. Scope

The Zach Drone is intended to fly inside Zachry where the drone will use ultrasonic sensors on the drone to navigate. The drone assumes that the tourists will not try to hit the drone or alter it's path while it is flying. It will be controlled using a microcontroller that can be connected to phones through WiFi using an app interface.

3.2. Operational Description and Constraints

The Zach Drone will be used by small or large groups to tour Zachry. The drone will start from the Anadarko Welcome Center where the tourists will connect to the drone with their phone through the phone app. The user will then pick what type of tour they would like to take. The drone will use the ultrasonic sensors on the drone and compare those values to the known dimensions of the building to navigate through the tour while using a speaker to playback important information about the building. A microcontroller will monitor sensors, use the camera, communicate with the flight controller and take input from the phone application.

The constraints for the Zach Drone are as follows:

- The drone must start and end at the Anadarko Welcome Center to charge.
- Tourist must have a smart-phone with the app downloaded.
- The drone must not be intentionally hit by the user or others in the building.
- The tour can only last up to twenty minutes before needing to recharge the drone.
- The speed of the drone cannot exceed that of walking speeds.
- Zach Drone will not be functional in an overcrowded building because camera may not be able to detect the group leader
- The drone must be given two hours in between tours to recharge the battery.
- Must take open stairs to travel between floors and cannot pass through doorways.

3.3. System Description

The Zach Drone is made up of four main subsystems detailed below:

Tourist Tracking & Speaker System: This system is used to track the tourists to make sure that they do not fall too far behind while taking the tour and to give the commentary about the various points of interest in Zachry. The system will consist of a camera[M1] to analyze and track a crowd leader to gauge the distance between the crowd and the drone. Additionally this system includes a speaker to give the commentary. The tracking system will communicate with the microcontroller to determine if the drone should slow down to allow the tourists to catch up or speed up if they get too close.

Drone & Sensors: This subsystem consists of the drone itself and the sensors mounted on it. Sensors will be used to avoid collisions with people and objects inside Zachry. It will accomplish this by having the microcontroller telling the flight controller to move in the opposite direction of whichever ultrasonic sensor is outputting a close reading. This system tells the microcontroller how close the drone is to objects on all six sides of the

drone. The microcontroller will determine if the drone will need to change paths and then communicate with the flight controller on the drone to tell it which direction to fly.

Zach Drone App: The drone will use a phone application to communicate with the tourists. This subsystem includes the user interface as well as the ability to talk to the microcontroller through a WiFi connection. The app will contain a connect-to-drone feature as well as options for what type of tour that the user wants to take. These options include an art tour, a classroom tour, and a standard tour. The user will also be able to stop or pause the tour from the app.

Tour Navigation: The Zach Drone will use ultrasonic sensors and the dimensions of the inside of Zachry to determine where the drone is at any given time. This system will receive signals from the ultrasonic sensors and then determine where the drone is within the building and in which direction the drone should fly based on this location.

3.4. Modes of Operations

There are three modes of operation for the drone. The first mode is the tour mode. During this mode, the drone will fly from destination to destination while guiding the tourist. The second mode of operation is the wait mode. The drone will enter this mode if the users are too far away for too long, the user presses the wait button on the app, or if connection is lost with the phone. In this mode, the drone will hover in air and wait for the condition to start again to be satisfied. The conditions for the drone to exit the wait mode and enter back into tour mode will be: the customer's device has connection and is in range of the drone after losing connection or that the start button has been clicked again to resume the tour after a user-initiated pause. The drone will only wait until the maximum length of the tour is reached. Once the maximum tour length of twenty minutes is reached, the drone has successfully completed the tour, or the user cancels the tour the drone will it enter the third mode of operation which is the return to start mode. In this mode the drone will navigate itself back to the Anadarko Welcome Center where it will be picked up to charge.

3.5. Users

The Zach Drone can be used by anyone who would like to take a tour of the new Zachry building without having to follow the tour schedule. The drone provides more flexibility to those who are taking a short visit to the Texas A&M campus. Anyone with a smartphone will be able to download the app and take a tour.

3.6. Support

Commonly asked questions will be listed in the app.

4. Scenario(s)

4.1. Small Personalized Groups

For tours with a small group of guests, the Zach Drone tour would be an ideal option. This would allow tourists to choose a personalized tour that meets their needs, whether it be looking at and listening about all the different art installations around Zachry or seeing what different learning and advising opportunities are available around the building. The drone

will be able to easily guide small groups by having them follow behind the drone at whatever pace they would like since the drone will wait on you.

4.2. Large School Groups

For large groups such as a school field trip the Zach Drone would be a great option. Since all large groups on school field trips or general tours of Texas A&M will have group leaders such as teachers or actual guides it will be very easy to control the pace of the group by making the drone follow the leader. That way the leader of the group can control at what pace the drone gives the tour. This will also add a bonus incentive for people to stay with the group because of the interest that the drone tour will bring. Instead of having a boring tour guide that many people will ignore, the drone could be as big of an attention getter as the building itself. In addition to two other preset tours, the drone will have standardized tours for school field trips or prospective student groups.

5. Analysis

5.1. Summary of Proposed Improvements

The use of the Zach Drone will allow tourists to have a more immersive experience while visiting the new Zachry Engineering Education Complex. The Zach Drone will give tourists the opportunity to custom fit a tour to whatever they preference using the Zach Drone App's multiple tour feature. Using the Zach Drone will allow tourists to tour Zachry whenever they feel instead of having to wait for a tour guide. The Zach Drone will show the tourists some of the innovative ways Texas A&M University is utilizing the new building.

5.2. Disadvantages and Limitations

The Zach Drone will be limited to the time of the tours since it will be running on a battery. The tour given by the Zach Drone will only be able to be a maximum of twenty minutes since that is the longest the battery can run before needing to be charged. The Zach Drone App is the main interface to communicate with the Zach Drone, and without a smartphone the tourists won't be able to select a tour.

5.3. Alternatives

- Camera vs. Bluetooth for crowd tracking
    - Using Bluetooth would be more unreliable due to the nature of Bluetooth connections.
    - Using Bluetooth signals would also lead to greater inaccuracies when calculating the distance between the drone and the tourists
    - Using Bluetooth signals would take less processing power than image processing.
- Ultrasonics vs ARUBA beacons for navigation
    - ARUBA signals would produce fuzzy data when used that is only up to 3m of accuracy which would cause the location of the drone to be incorrect.
    - Ultrasonics would only have a small degree of inaccuracy but could be difficult to use if the drone gets lost.
    - The ultrasonics would need the dimensions of all locations near the tour route to be used.

6. Impact(s)

6.1. Economic & Time Impacts

Electricity is the only upkeep cost the drone will have; more specifically the cost of the amount of electricity it takes to recharge the drone battery. Charging a battery for an hour or two has a negligible cost.

Currently though the true cost of the Zachry tours is time. Tour givers are departmental members that volunteer to have time taken out of their day to give these tours. A drone tour would return that time back to them; allowing them to be put to better use. In other settings this can save a company the cost of an entire position.

6.2. Social & Business Impacts

In the case of the Zachry building the Zach Drone will act as an engineering feat to entice potential students. Its secondary purpose is to “wow” students & show them what engineering is capable of. For other business settings it will allow companies to give tours where they otherwise could not spare time or money to have someone else do it. It could even be used in an accessibility sense if we were to switch the speaker out for a screen or set of LEDs, the drone could be used to direct deaf people. Applicable settings could be museums, airports, & many others.

6.3. Safety Impacts

The propellers on a drone can spin at thousands of RPM & can pose a hazard. The drone will be set to fly as high as it can leaving enough room to not hinder its performance. Sensors will detect people & the drone will attempt to maneuver away should someone attempt to touch it by flying in the opposite direction of whichever ultrasonic had a reading that was too close in proximity.

7. Citations

https://federaldroneregistration.com/?gclid=EAIaIQobChMIq9mDtIjN3QIVIIlpCh0-KAneEAAYASAAEgLrHPD_BwE
https://federaldroneregistration.com/know-before-you-fly
http://knowbeforeyoufly.org/
https://www.scienceabc.com/innovation/what-is-the-range-of-bluetooth-and-how-can-it-be-extended.html
https://zachry.tamu.edu/tours/
https://zachry.tamu.edu/

Zach Drone

Justin Armenta
Elias Duque
Jonah Morris
Grayson Vansickle

Functional System Requirements

REVISION – 1.2
5 December 2018

FUNCTIONAL SYSTEM REQUIREMENTS
FOR
Zach Drone

PREPARED BY:
TEAM <17>

_____
Author Date

APPROVED BY:

_____
Project Leader Date

_____
Prof. S. Kalafatis Date

_____
T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
0.0	9/6/18	Grayson Vansickle		Draft Release
0.2	9/24/18	Elias Duque		Added Introduction section
0.4	10/1/18	Grayson Vansickle		Added Applicable and Reference Documents section
0.6	10/1/18	Jonah Morris		Added System Definition subsection of Requirements section
0.8	10/1/18	Justin Armenta		Added Characteristics subsection of Requirements section
1.0	10/3/18	Grayson Vansickle		Edited and Formatted
1.2	12/5/18	Grayson Vansickle		Updated information

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1. Purpose and Scope

The purpose of the Zach Drone is to give guided tours to a crowd of people that have signed up for a viewing of the new Zachry Engineering building. It shall communicate with an app that will sync up with the phones of the tourists so that it shall keep track of them. The drone must maneuver the hallways of Zachry and must give a spoken tour. It should give information on where labs and classrooms are, as well as draw the crowd's attention to the installed art pieces. The drone shall detect via a camera that will track the crowd leader's location. When the tour is over the drone must return to the Anadarko Welcome Center so that it shall recharge before the next tour.

The drone must be able to fly using its own power for at least 15 minutes to give a fully-fledged tour. It must have a speaker to give commentary on the surroundings and shall provide relevant information. The ultrasonic sensors on the drone shall be used along with known dimensions of the building to find the drone's location and plot its course. The drone shall have ultrasonic sensors to prevent collisions with ceiling fixtures, walls, and pedestrians.

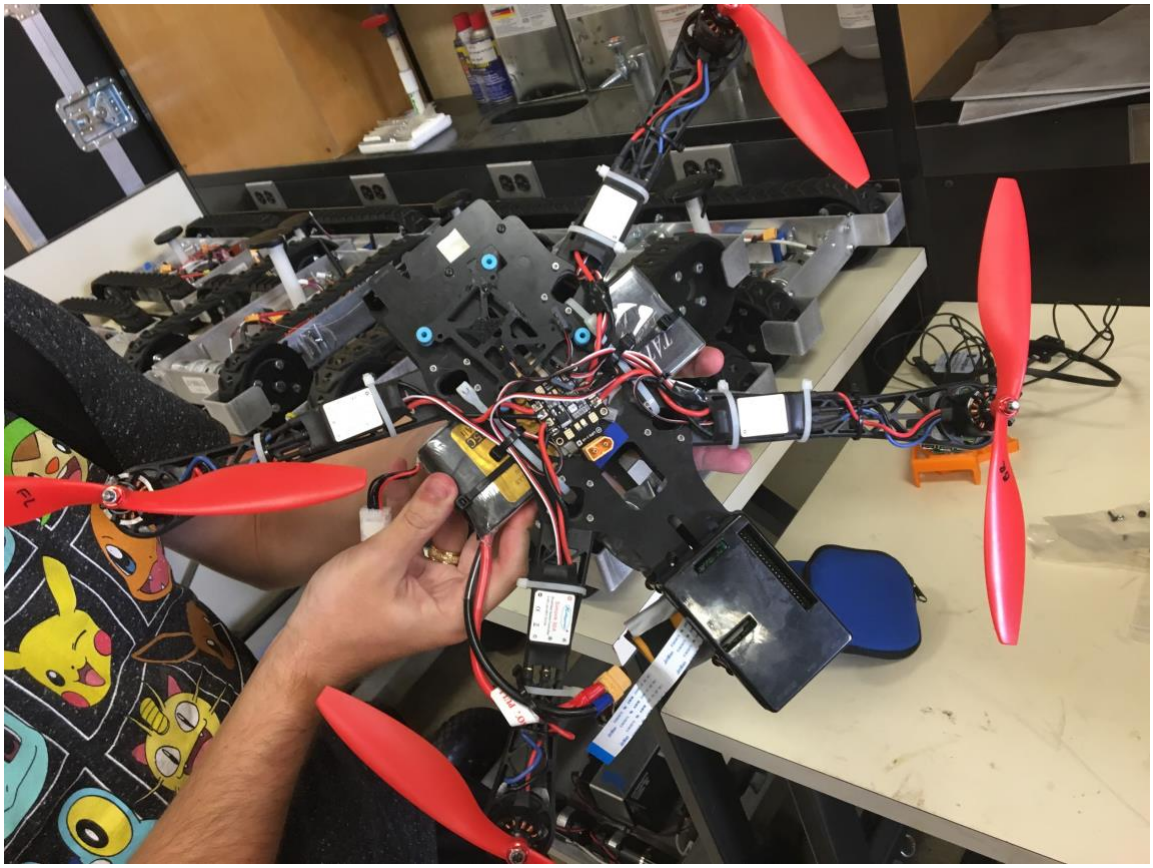


Figure 1 - In-progress Image of Drone

1.2. Responsibility and Change Authority

Jonah Morris is the team leader & shall be responsible for the oversight of the subsystems & ensuring that requirements are met according to the validation plan. Any changes to be made to the deliverable requirements must be approved by both Jonah Morris & Stavros Kalafatis.

Subsystem	Responsibility
Drone & Assembly	Jonah Morris
Navigation	Justin Armenta
Crowd Tracking	Elias Duque
Mobile Application	Grayson Vansickle

Table 1 - Subsystem Responsibility

2. Applicable and Reference Documents

2.1. Applicable Documents

The following documents, of the exact issue and revision shown, form a part of this specification to the extent specified herein:

Document Number	Revision/Release Date	Document Title
NFPA 70	2014	National Electrical Code
IEEE 802.15.4	2013	Standard for Low Rate WPAN
FAA Part 107	2018	Small Unmanned Aircraft Regulations

Table 2 - Applicable Documents

2.2. Reference Documents

The following documents are reference documents utilized in the development of this specification. These documents do not form a part of this specification, and are not controlled by their reference herein.

Document Title	Publisher
Raspberry Pi 3 B Data Sheet	Raspberry Pi
Pixhawk 2.1 Flight Controller Data Sheet	Pixhawk
Ultrasonic Rangefinder HRLV-EZ0 Data Sheet	Maxbotix
Python Standard Library Reference	Python
Swift Language Reference	Apple
Java SE Technical Documentation	Oracle

Table 3 - Reference Documents

2.3. Order of Precedence

In the event of a conflict between the text of this specification and an applicable document cited herein, the text of this specification takes precedence without any exceptions.

All specifications, standards, exhibits, drawings or other documents that are invoked as “applicable” in this specification are incorporated as cited. All documents that are referred

to within an applicable document are considered to be for guidance and information only, with the exception of ICDs that have their applicable documents considered to be incorporated as cited.

3. Requirements

The drone will give guided tours to tourists who have set up a tour of Zachry. The drone must be able to connect with the tourist's phone using a WiFi connection from an app, so that the user can set up locations they want to visit for the tour. The drone must be able to navigate through Zachry from one checkpoint to the next without colliding with objects.

This section defines the minimum requirements that the development item(s) must meet. The requirements and constraints that apply to performance, design, interoperability, reliability, etc., of the system are covered.

3.1. System Definition

The Zach Drone is an automated tour delivery system. Users connect to the drone using WiFi through a mobile app interface. They are given a lists of tours to choose from. Once the tour group has selected a tour, the drone will fly the route while using a speaker to playback important information about the building. For this project there are four major subsystems: mobile app, navigation, tour group tracking and the drone.

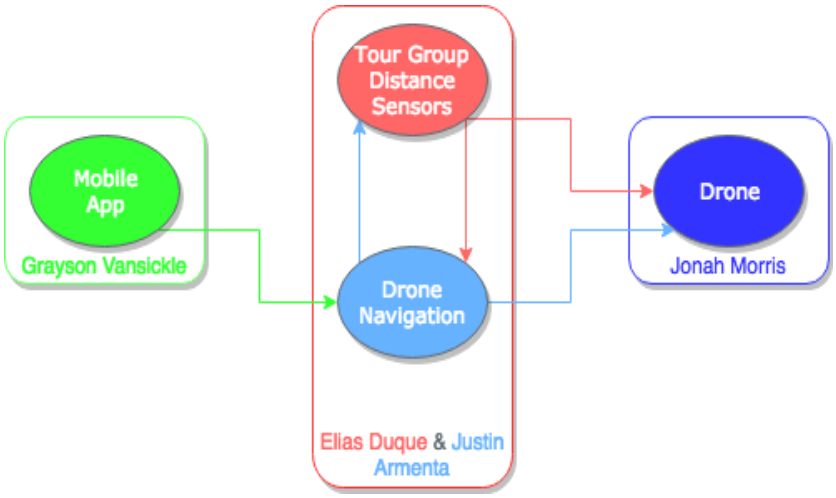


Figure 2 - Drone Subsystems & Responsibilities Block Diagram

The mobile application is the primary interface with the user. Once the user picks a tour to take, the mobile app will communicate with the navigation system and tell it where to go. The navigation system will use ultrasonic sensors on the drone to determine where it is, if there are obstacles, and what direction the drone should go next. The navigation system will then tell the drone's flight controller which direction to fly in. The drone will fly towards the waypoint while the group tracking system controls the speed of the drone based on the distance from the group.

3.2. Characteristics

3.2.1. Functional / Performance Requirements

3.2.1.1. Smartphone Application

An application shall be created on both Android and Apple platforms where the tourist can use their smartphone to access the app, and use it to select a tour route and when they would like a tour. The drone shall also have a speaker to tell the tourists information about Zachry. The app will continuously communicate with the drone allowing the user to change route or cancel.

3.2.1.2. Drone Functionality

The drone shall be able to fly for the full length of a 15 minute tour, while being able to avoid a collision with anything that comes near the drone. It will also be able to take off and land at the designated location inside of the Anadarko Welcome Center. The maximum time requirement of the tour is necessary since tours need to cover as much of the building as possible while also taking into account that our battery life is about 20 minutes. There also must be a designated take-off and landing location so people know where to go for the tour and a storage spot for the drone.

3.2.1.3. Navigation of Checkpoints

The drone shall be able to navigate, in an efficient manner, from any of the tour destinations to another successfully using the ultrasonic sensors on the drone as a guidance system. Without this capability the drone wouldn't be able to lead a tour group to all the spots that the group would like to see.

3.2.1.4. Crowd Guiding

The drone shall stay within about 15 feet of the crowd using a camera to keep track of a crowd leader and their distance. 15 feet is recommended such that the connection between the leader's mobile device and drone does not terminate which allows the tourists to pause or cancel at any moment and avoids forcing them to reconnect. If the crowd becomes closer than 8 feet the drone will speed up. If the crowd exceeds 25 feet away then the drone will slow down. If the drone is told to slow down twice consecutively then the drone will stop momentarily.

3.2.2. Physical Characteristics

3.2.2.1. Mass

The maximum total mass of the drone shall be less than 3.06 kg. The maximum combined thrust for four motors produces a total of 4.6 kg. To efficiently take off the motors must be able to produce thrust equal to 150% of the weight of the drone which here is 4.6 kg. 4.6 kg is 150% of 3.06 kg making it the maximum weight we should target.

Rationale: This is the maximum weight that can be carried by the drone such that it will fly at 75% throttle. 75% throttle simply means the motor can run at a higher thrust thanks to the stable conditions inside the building.

3.2.2.2. Volume Envelope

The volume envelope of the Zach Drone shall be less than or equal to 150 mm in height, 525.4 mm in width, & 525.4 mm in length.

Rationale: This is the smallest size that can house all components safely. Drone size affects weight so a smaller drone makes meeting out weight requirement easier.

3.2.3. Electrical Characteristics

3.2.3.1. Inputs

No combination of settings input by the user shall damage the Zach Drone, cause any malfunction within the system, nor reduce performance.

Rationale: The chance of user errors should be minimized by the design of the system.

3.2.3.1.1 Power Consumption

The maximum peak power for the system shall not exceed 1050 watts.

Rationale: The maximum power the drone motors can take is 259 watts per motor. The rest of the system requires less than 10 watts to power.

3.2.3.1.2 Voltage Level

The output voltage of the battery will be between 14.8 and 16.5 VDC. The minimum input voltage levels for the sensors, motors, microcontroller and pixhawk flight controller are 2.8 VDC, 4.5 VDC, 4.75 VDC and 4 VDC respectively.

Rationale: The nominal voltage levels for the sensors, motors, microcontroller and Pixhawk are 3.3 VDC, 5 VDC, 5 VDC and 5 VDC respectively.

3.2.3.1.3 External Commands

The Zach Drone shall document all external commands in the appropriate ICD.

Rationale: The ICD will capture all interface details from the low level electrical to the high level packet format.

3.2.3.1.4 Data Output

The Zach Drone shall include a speaker for playback to the tourist.

Rationale: This requirement was set by the customer.

3.2.3.1.5 Diagnostic Output

The Zach Drone shall include a mobile application interface for control and data logging.

Rationale: Provides the ability to manually control things for debugging.

3.2.3.2. Wiring

The Zach Drone shall follow the connector guidelines set forth in the National Electric Code. Article NEC 300 details general guidelines for wiring methods and materials.

Rationale: Conform to general wiring standards.

3.2.4. Communication Requirements

3.2.4.1. Phone App to Microcontroller Connection

The tourist's phone shall use the app to tell the drone which tour to take, so the drone can navigate to all checkpoints one at a time. The app will also have an option to "Cancel Tour" during the tour if the user becomes uninterested in the tour or has other obligations and has to leave. There will also be a "Pause" option, in case the user wants to look at something for a while or go to the bathroom, where the drone will hover idly until the user clicks the "Resume Tour" option or enough time has passed that the drone will automatically cancel the tour.

3.2.4.2. Microcontroller to Audio Connection

The microcontroller will use a USB 2.0 connection to send an audio signal to the speaker on the drone to tell the tourists information about destinations on the tour.

3.2.4.3. Microcontroller to Flight Controller Connection

The microcontroller will use a serial connection to connect with the flight controller to command the flight controller where to navigate to, how fast, when to avoid an obstacle, and other important flight commands.

3.2.4.4. Ultrasonic Sensors to Microcontroller Connection

The ultrasonic sensors will be connected to one of the GPIO ports on the Raspberry Pi. These sensors will be used to navigate around Zachry as well as detect if the drone is close to colliding with something as well as for navigation.

3.2.5. Environmental Requirements

The Zach Drone shall be designed to withstand and operate inside of the Zachry Engineering Education Complex.

Rationale: This is a requirement specified by our customer because this is where tours will be held.

3.2.5.1. Thermal Requirement

The Raspberry Pi 3 B on the Zach Drone will not exceed 70℃.

Rationale: The Raspberry Pi 3 B is rated for temperatures of 0℃ - 70℃ by the manufacturer.

3.2.6. Failure Propagation

The Zach Drone shall account for connection failures as well as objects getting near the drone. The drone will have a manual override to land if the user observes a malfunction.

3.2.6.1. Camera Tracking

3.2.6.1.1 Camera Failure

If the camera used to track the crowd should malfunction to where the image can no longer be processed then the camera will shut off and the crowd detection functionality will be disabled. The crowd will be alerted that there has been a malfunction but will continue.

3.2.6.1.2 Image Processing Failure

If the image processing detection and tracking should fail such that the leader can no longer be tracked then the drone will attempt to re-initialize the crowd leader. Should this attempt fail then the drone will report that crowd tracking has been disabled and will continue on the tour.

3.2.6.2. Object Proximity Detection

3.2.6.2.1 Ultrasonic Proximity Detection

If an object gets too close to the drone, the ultrasonic sensors, which will be placed on each side of the drone, will be used to detect how close the object is to the drone. The microcontroller will then tell the drone to move away from the object by going in the opposite direction of the ultrasonic that was triggered.

3.2.6.3. Navigation

3.2.6.3.1 Sensor Reading Errors

If there is a small one time failure of an ultrasonic sonic, the error will most likely be ignored and not taken into account due to how the algorithm for navigating the drone works, which expects that the sensors won't be perfect and may produce a false, one time, value. However if a sensor continuously is reading an incorrect value for multiple iterations, the drone will perform an emergency landing where it will announce that there has been a fatal error and the drone needs to land.

4. Support Requirements

The Zach Drone requires that the battery be charged for two hours in between uses. The Zach Drone also requires the user to have a smartphone with the Zach Drone app downloaded.

Appendix A Acronyms and Abbreviations

Zach	Zachry Engineering Education Complex
USB	Universal Serial Bus
℃	Degrees Celsius
GPIO	General Purpose Input Output
APP	Application
NEC	National Electric Code
VDC	Volts Direct Current
KG	Kilogram
%	Percentage

Appendix B Definition of Terms

Flight Controller: A microcontroller that includes gps, compass and gyroscope that controls the motors to fly the drone.

Ultrasonic Sensor: A distance sensor that uses ultrasonic sound waves to determine distance to the nearest object.

Zach Drone

Justin Armenta
Elias Duque
Jonah Morris
Grayson Vansickle

Interface Control Document

REVISION – 1.2
5 December 2018

INTERFACE CONTROL DOCUMENT
FOR
Zach Drone

PREPARED BY:
TEAM <17>

_____
Author Date

APPROVED BY:

_____
Project Leader Date

_____
Prof. Stavros Kalafatis Date

_____
T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description

0.0	9/6/18	Grayson Vansickle		Draft Release
0.2	9/23/18	Justin Armenta		Added Overview section
0.4	10/1/18	Grayson Vansickle		Added Communications Protocol section
0.6	10/1/18	Elias Duque		Added Physical Interface section
0.8	10/3/18	Jonah Morris		Added Electrical Interface section
1.0	10/4/18	Grayson Vansickle		Edited and Formatted
1.2	12/5/18	Grayson Vansickle		Updated information

Table of Contents

List of Tables

List of Figures

1. Overview

This document will cover the systems designed and used in the Zach Drone tour system. The drone that will be used to give tours of Zachry will be described on a functionally detailed level. The interface between the flight controller on the drone, the microprocessor, and all devices used to determine location will be described in terms of how they will function separately as well as how they will communicate with each other. The way the proximity sensors will interact with the microprocessor and flight controller will be addressed as well. The physical aspects of the drone such as its weight, dimensions, size, component location, and component size will also be described. Electrical aspects will be explained as well. This will include details such as power inputs and outputs for components, voltage, current and power levels at different points in the system, and signal interfaces for components. Lastly, the methods in which the user will interact with the drone will be explained.

2. References and Definitions

2.1. References

Document Title	**Revision/Release Date**	**Publisher**
National Electrical Code	2014	NECA
Raspberry Pi 3 B Data Sheet	2018	Raspberry Pi
Pixhawk 2.1 Data Sheet	2016	Pixhawk
HRLV-EZO Data Sheet	2015	Maxbotix
SimonK 30A ESC Data Sheet	Revision 1.0	Lynxmotion

Table 1 - References

2.2. Definitions

g.	Grams
kg.	Kilograms
m.	Meters
mm.	Millimeters
V.	Volts
A.	Amps
mA.	Milliamps
Ah.	Amp hour
mAh.	Milliampere hour
W.	Watt
ESC	Electronic Speed Controller

3. Physical Interface

3.1. Weight

Component	Weight	Amount	Total Weight
SunnySky X2212-13 KV980 II	69 g	4	276 g
1045 Propeller	7 g	4	28 g
SimonK 30A ESC	27 g	4	108 g
Matek PDB-XT60	11 g	1	11 g
Pixhawk 2.1 Flight Controller	50.5 g	1	50.5 g
Maxbotix Ultrasonic Rangefinders	4.23 g	6	25.38 g
Raspberry Pi 3 B	49.7 g	1	49.7 g
TATTU 4S 8 Ah 25C Li-Po Battery	785 g	1	785 g
Drone Frame	90 g	1	90 g

Table 2 - Physical Weight

3.2. Dimensions

Component	Length	Width	Height
SunnySky X2212-13 KV980 II	28 mm	28 mm	40 mm
1045 Propeller	254 mm	114.3 mm	6 mm
SimonK 30A ESC	216 mm	25 mm	8 mm
Matek PDB-XT60	50 mm	36 mm	24.2 mm
Pixhawk 2.1 Flight Controller	94.5 mm	44.3 mm	31.42 mm
Maxbotix Ultrasonic Rangefinders	22.36 mm	15.52 mm	20 mm
Raspberry Pi 3 B	87 mm	58.5 mm	18 mm
TATTU 4S 8 Ah 25C Li-Po Battery	165 mm	65 mm	53 mm
Drone Frame	500 mm	500 mm	150 mm

Table 3 - Physical Dimensions

3.3. Mounting Locations

3.3.1. Mounting Location for Control Electronics

Electronics responsible for the control & flight of the drone include the flight controller & the Raspberry Pi which will manage the navigation. For optimum performance & accuracy the flight controller must be placed in the direct center of the drone's mass. The Raspberry Pi will be placed behind the flight controller at the tail end of the drone.

3.3.2. Mounting Location for Power Electronics

Electronics responsible for the power input, output, & distribution include the power distributor & the battery itself. The battery will be mounted in the lower half of the middle of the drone's mass to keep it balanced. The power distributor must be in the middle of the drone below the flight controller to allow for the brushless motor controls to reach evenly.

3.3.3. Mounting Location for Motors

The motors must be placed at the furthest end of the drone arms. Propeller blades are attached to the shafts of the motors. The brushless motor controls will be mounted along the arms themselves stretching from the motors to the power distributor in the middle.

3.3.4. Mounting Location for Sensors

Ultrasonic sensors must be placed on all faces of the drone for maximum precision in collision avoidance. There will be one sensor per side: front, back, left, right, upward & downward.

4. Electrical Interface

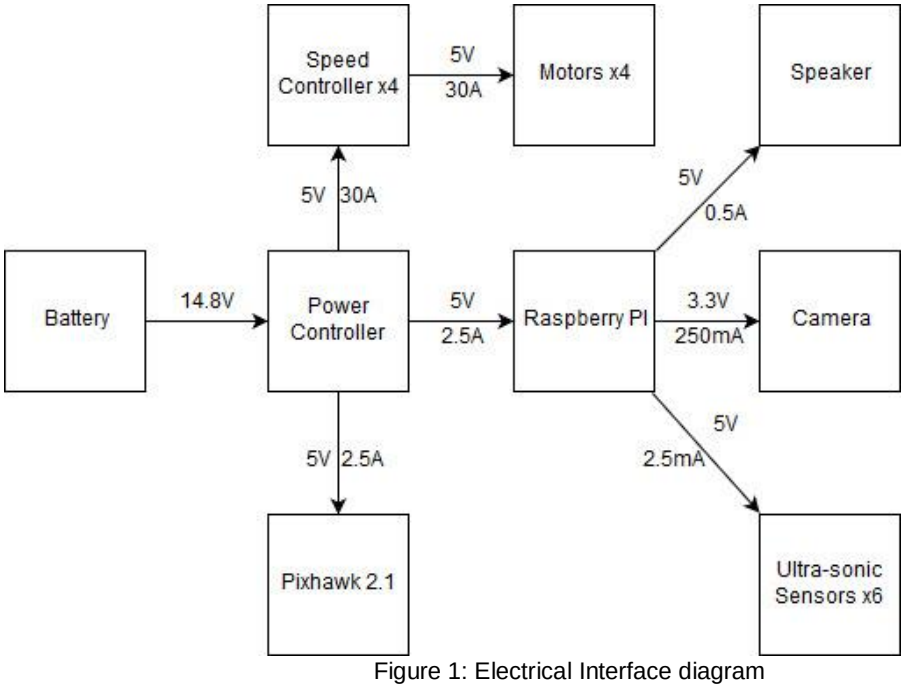


Figure 1: Electrical Interface diagram

4.1. Primary Input Power

The primary input power is the 14.8V 8000mAh battery pack that will be connected to the power distributor.

4.2. Voltage and Current Levels

Component	Voltage [V]	Current [A]	Power [W]
Pixhawk 2.1	5	2.5	12.5
Raspberry Pi	5	2.5	12.5
Speed Controller	5	2	10
Motor ESC	0-5	0-30	150 (max)
Speaker	5	0.5	2.5
Ultrasonic sensor	3.3	.0025	0.00825

Table 4 - Electrical Voltage, Current, & Power

Detailed in Table 4 above are the wattages consumed by each part of the drone in one hour. Taking the amount of components into account it is not unreasonable to see a total of 215 watt hours. It is unreasonable & unnecessary that our drone will fly for an hour; our target is 20 minutes. In a 20 minute window our drone will therefore use a third of the the total watt hour amount, equaling 71.7 watts. Every component is vital to normal operation of the drone except for the speaker. There is no lower power operation mode of the drone though the speaker only consumes power when in use during tours.

4.3. Signal Interfaces

The signal interfaces included are as follows:

- Serial port connection from Raspberry Pi to pixhawk
- Pixhawk PWM signal outputs to speed controllers
- Raspberry Pi 3.5mm audio jack to speaker
- Ultrasonic sensors to Raspberry Pi GPIO headers

4.4. User Control Interface

The user control interface is a mobile application. The user is given options for tour destinations and the application communicates these choices to the drone's Raspberry Pi via WiFi connection.

5. Communications / Device Interface Protocols

5.1. Audio Interface

The drone will have an attached USB 2.0 speaker connected to the 4-pole stereo output port on the Raspberry Pi 3 B. The speaker will be powered by the one of the Pi's four onboard USB 2.0 ports. This audio interface will be used to give the description of the point of interest the drone is currently located at.

5.2. Device Peripheral Interface

The sensors used by the drone will be attached to the Pi's extended 40-pin GPIO header. The Pixhawk flight controller will also be connected to the GPIO header. These ports will be used to communicate between the sensors and microcontroller.

5.3. WiFi Interface

The Raspberry Pi 3 B has onboard BCM43438 wireless LAN (WiFi). Using the Pi's WiFi connection, the Pi will be able to communication with the mobile application. The WiFi connection will comply with the IEEE 802.11 standards for wireless local area networks (WLANs).

6. References

Ultrasonic Sensors
https://www.maxbotix.com/documents/LV-MaxSonar-EZ_Datasheet.pdf

Raspberry Pi
https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md

Pixhawk Flight Controller
http://www.hex.aero/wp-content/uploads/2016/07/DRS_Pixhawk-2-17th-march-2016.pdf

SimonK Speed Controller
http://www.lynxmotion.com/images/document/PDF/Lynxmotion%20-%20SimonK%20ESC%20-%20User%20Guide.pdf

Execution Plan

	October 15	**October 29**	**November 12**	**November 26**	**Dec 3**
Drone Subsystem	Drone parts put together	Drone flies	Sensors added to drone and coded	Drone avoids collisions	Drone flight tested in Zachry
Navigation Subsystem	Raspberry Pi connects to beacons	All first floor dimensions are obtained	Algorithm for obtaining checkpoint is made	Model is made to simulate hallway	Simulation GUI is made to show drone
Tour Group Tracker	Get Pi camera feed	Recognize people	Create boundary update logic	React to distance by changing speed	Drone keeps distance
Mobile Application	Draft application layout	Start coding GUI	Connect to drone	Route selection	Send tour destinations to drone

Validation Plan

	Stage 1	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**
Drone Subsystem	Confirm drone and components power on	Confirm drone hovers in place	Confirm sensors give accurate readings	Confirm drone doesn't collide with anything	Confirm drone can fly a test path
Navigation Subsystem	Confirm Raspberry Pi connects to beacons	Confirm all wall changes and checkpoint lengths are found	Confirm algorithm outputs correct checkpoint when fed sensor data	Confirm model can create fake sensor readings and algorithm will do correct action	Confirm drone and map data match with the GUI output
Tour Group Tracker	Confirm video feed output	Confirm people in frame through boundary box	Confirm boundary update logic executes properly	Confirm range conditions execute (too far, too close, right)	Confirm real response is appropriate to distance measured
Mobile Application	Confirm group all approve of application layout	Confirm application can be loaded on iPhone and Android	Confirm application connects to drone	Confirm selection of tour is sent to drone	Confirm application tells drone the list of destinations for the selected tour

Zach Drone

Elias Duque

Tour Group Tracker Subsystem

REVISION – 1.2
5 December 2018

Table of Contents

List of Figures

List of Tables

1. Introduction

The purpose of this report is to detail the working aspects and the validation of the Tour Group Tracking subsystem of the Zachry Tour Drone. The Tour Group Tracking is primarily software based with the only hardware being the necessary camera apparatus and the microcontroller. This subsystem interacts the most with the navigation subsystem as the data from this subsystem will change the speed of the drone either through slowing down or speeding up in order to keep an appropriate distance with the group. The Tour Group Tracking subsystem augments the tour by making it more comfortable for tourists allowing them to go at their own pace. This report will cover aspects such as the design, data collection, accomplishments, and admissions of necessary future work.

2. Theory

The scope of the Tour Group Tracking subsystem is to create an appropriate response to the changing distance between the tour drone and the crowd of tourists. This response is necessary due to the variable pace that a crowd of people can move. As the drone reaches a point of interest it will begin to give commentary and without the appropriate responses may be giving the commentary to a crowd that has yet to catch up. This system monitors an appointed crowd leader and tracks them throughout the tour and uses them to gauge the distance of the overall crowd. OpenCV's human descriptor detectors and MOSSE tracker will be used in tandem to track both the actual person as well as their relative distance. Based on that relative distance the drone may generate a response that is sent from the microcontroller to the flight controller. The only time a response is not generated is when the crowd is an acceptable distance away or during a malfunction. Should the crowd leader get 25 feet or further from the drone it will send a signal to the flight controller for a slower throttle. If the slowdown response is sent twice consecutively then on the third slowdown response the program will instead tell the flight controller to hover in place and wait. Should the crowd leader come closer than 8 feet then the microcontroller will send a signal to the flight controller to up the throttle.

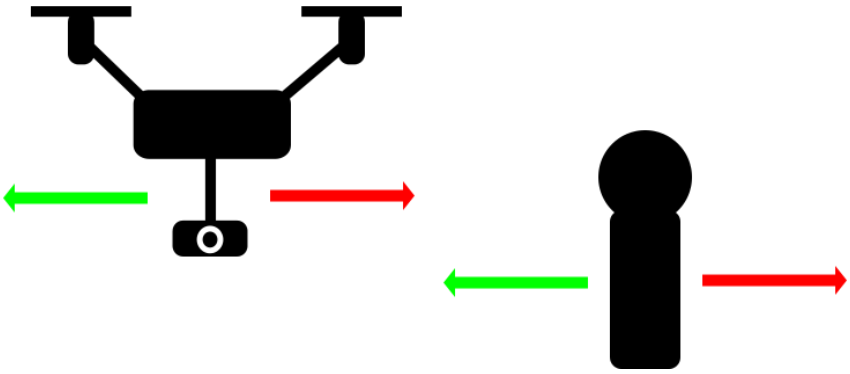


Figure 1: Conceptual Overview of System Response

3. Design

3.1. Hardware

The only hardware unique to the Tour Group Tracker subsystem would be the Raspberry Pi Camera Module v2. The other hardware that could be considered part of this system would be the Raspberry Pi microcontroller, PixHawk flight controller, and the drone itself with all its components. The main interaction that this subsystem has with those other components is the signal that will command the drone to change speed. Otherwise this subsystem is mostly divorced from the other hardware and could function (though not do anything) without them. The camera hardware consists of the the camera module itself and later down the line will also consist of the casing and mounting that is currently not necessary. The Raspberry Pi Camera Module v2 uses the designated camera ribbon cable port on the microcontroller and runs off 3.3 V and 250 mA of current. Exact data is difficult to find but with the lower resolutions the camera is running at these numbers are subject to change. The camera is capable of high resolution but will be running at a locked 640 x 480 pixels where the frames are further downsampled to 533 x 400 pixels for speed. To further increase speed the camera is run at a lower framerate which is currently 5 frames per second. Due to the camera being manufactured specifically for this microcontroller with the microcontroller having a specialized port just for it no other hardware is necessary for the operation of the camera besides obvious necessities such as a power source.

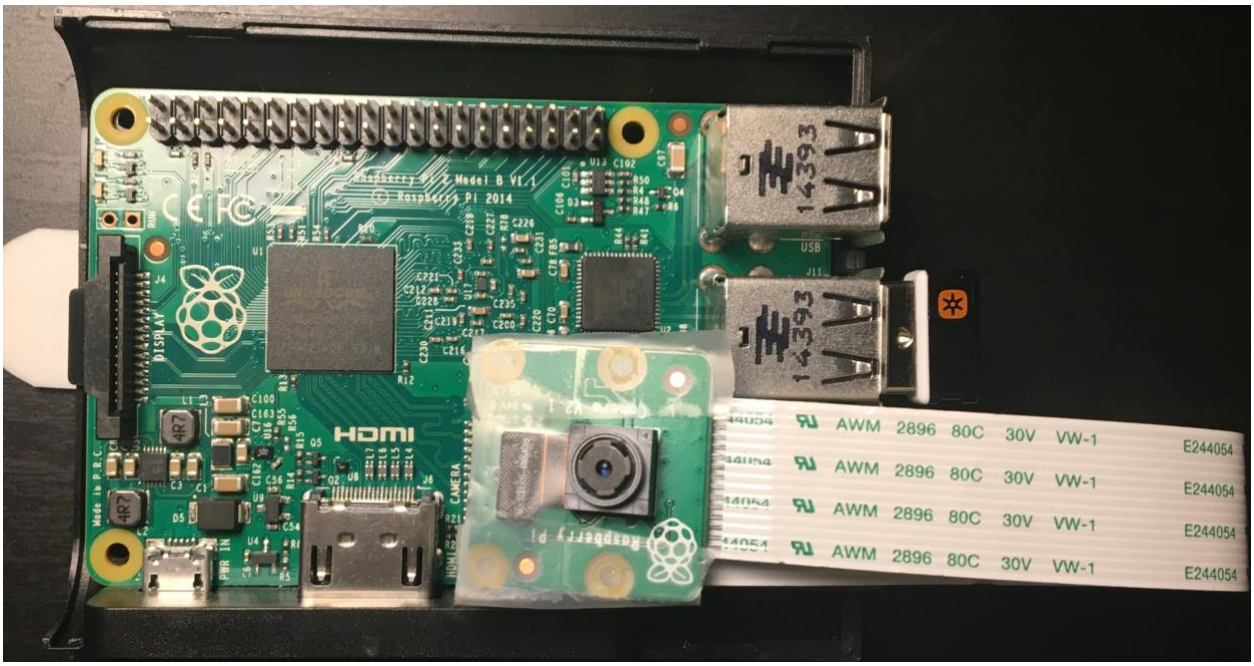


Figure 2: Camera Module attached to Microcontroller

3.2. Software - Detectandtrackvideo3.py

Since the Tour Group Tracker subsystem relies heavily on image processing to perform its duties it is very software intensive. Programming language Python was used for the coding

and OpenCV 4.0.0 was used for all steps of image processing. The actual image processing requires minimal coding but the extensiveness of the program comes from the logic required to update the tracker and preserve what we are tracking. To begin the tour an elected crowd leader will stand on a red X where the drone will get into its initial position at 15 feet away. OpenCV's Histogram of Ordered Gradients (HOG) detector is set to detect humans. The detection is performed and a boundary box around the crowd leader is drawn. This box has a width and height that are saved as references along with the equivalent area (width times height). A MOSSE tracker is generated and initialized with the generated boundary box. Movement of anything within this box will be monitored allowing us to stay on the crowd leader. The MOSSE tracker is fast but is less accurate than other trackers. Speed was the main determiner of the tracker eliminating most but the MOSSE was ultimately chosen for proving the best at occlusion out of the two to choose from. Having greater accuracy after occlusion means that we needn't worry if non-tourists walk in front of our crowd.

```
#create tracker
tracker = cv2.TrackerMOSSE_create()

# initialize the HOG descriptor/person detector
hog = cv2.HOGDescriptor() #create descriptor
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector()) #set descriptor to find people

#initialize crowd leader and their boundary box
(rects, weights) = hog.detectMultiScale(frame, winStride=(4, 4), padding=(8, 8), scale=1.05)
for (x, y, w, h) in rects:
    cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 0, 255), 2) #create a rectangle from detection

#initialize tracker with bounding box
success = tracker.init(frame, tuple(rects[0]))
```

Figure 3: Code Excerpt of Detection and Tracking (Non-relevant code was omitted for brevity)

Once the tracker is initialized the tour begins. After a predetermined amount of time, currently 10 seconds, the HOG detector is run once more. Testing of the natural inaccuracy and drift of the MOSSE tracker showed that for slow regular movement there was virtually none. This means that there was no necessity to update our tracker with a new box for that reason unlike the MedianFlow tracker tested (more on that later). However the only downside to the MOSSE tracker is that the boundary box does not dynamically update its own width and height like the MedianFlow does. So while updating the tracker with a new box is unnecessary for reasons of accuracy it is necessary to get an updated bounding box for the purpose of finding relative distance. An arbitrary time of 10 seconds was chosen and can be adjusted at a later time. After humans in the camera frame are detected the logic loop begins. Firstly we want to ensure that we keep track of the right person so we need to eliminate the detections that are not overlapping with our leader. A function was defined that checks the left and right edges of both the detection being checked and the current tracker box. If the left edge of either box is past the right edge of the other box then they do not intersect. If the right of of either box is left of the left edge of the other box then they do not intersect. Meeting neither of these conditions means the rectangles intersect. Rectangles that do not pass the intersection test are removed from the array. In the figure below, 'a' and 'b' are arrays for rectangles with elements 0 through 4

corresponding with the x coordinate, y coordinate, width in pixels, and height in pixels respectively.

```
#define rectangle intersection function
def intersection(a,b):
    if (a[0] > b[0]+b[2] or b[0] > a[0]+a[2]):
        print('first intersection test failed')
        return False
    elif (a[1] > b[1]+b[3] or b[1] > a[1]+a[3]):
        print('second intersection test failed')
        return False
    else:
        print('rectangles intersect')
        return True
```

Figure 4: Intersection Test Function

To finally determine which rectangle we go with we check the weight values associated with the detections. The OpenCV HOG Detector outputs a weight value along with each rectangle that estimates how confident the detector is that what it has detected is a human. This is done to allow us to finally select a rectangle and helps eliminate false positives. Once the rectangle has been decided the tracker is re-initialized using this new box. The width and height of this box are saved and used to generate the associated area. The new area is compared to the original box area from 15 feet away and the appropriate response is sent out. If the new area is 25% greater than the original then they are too close and the program sends the signal to speed up. If the new area is only 60% of the original area (40% smaller) then we are too far and the signal is sent to slow down. If neither of these conditions is met then the crowd is an acceptable distance and no action is taken. In the code below appropriate messages are printed to the console as we have not currently integrated the systems. The messages are placeholders. The last mention to be made is if detection fails. Should the HOG detector find no humans in the frame then the program will initialize nothing and continue the tracking that it was already performing.

```
        else: #if we do have detections...
            tracker = cv2.TrackerMOSSE_create() #overwrite our tracker with a new one

            success = tracker.init(frame, tuple(editRects[argmax(editWeights)]))

#now we need to see if we have to speed up or slow down, otherwise stay course
            newTrkRect = editRects[argmax(editWeights)] #grab our new tracker rectangle
            newTrkRectW = newTrkRect[2]
            newTrkRectH = newTrkRect[3]
            newTrkRectArea = newTrkRectW * newTrkRectH

        if newTrkRectArea/refArea > 1.25:
            print('too close! let\'s speed up.')
        elif newTrkRectArea/refArea < 0.6:
            print('too far! let\'s slow down.')
        else:
            print('just right :)')
```

Figure 5: Excerpt of Tracker Re-initialization and Distance Response (some omission of validation print functions)

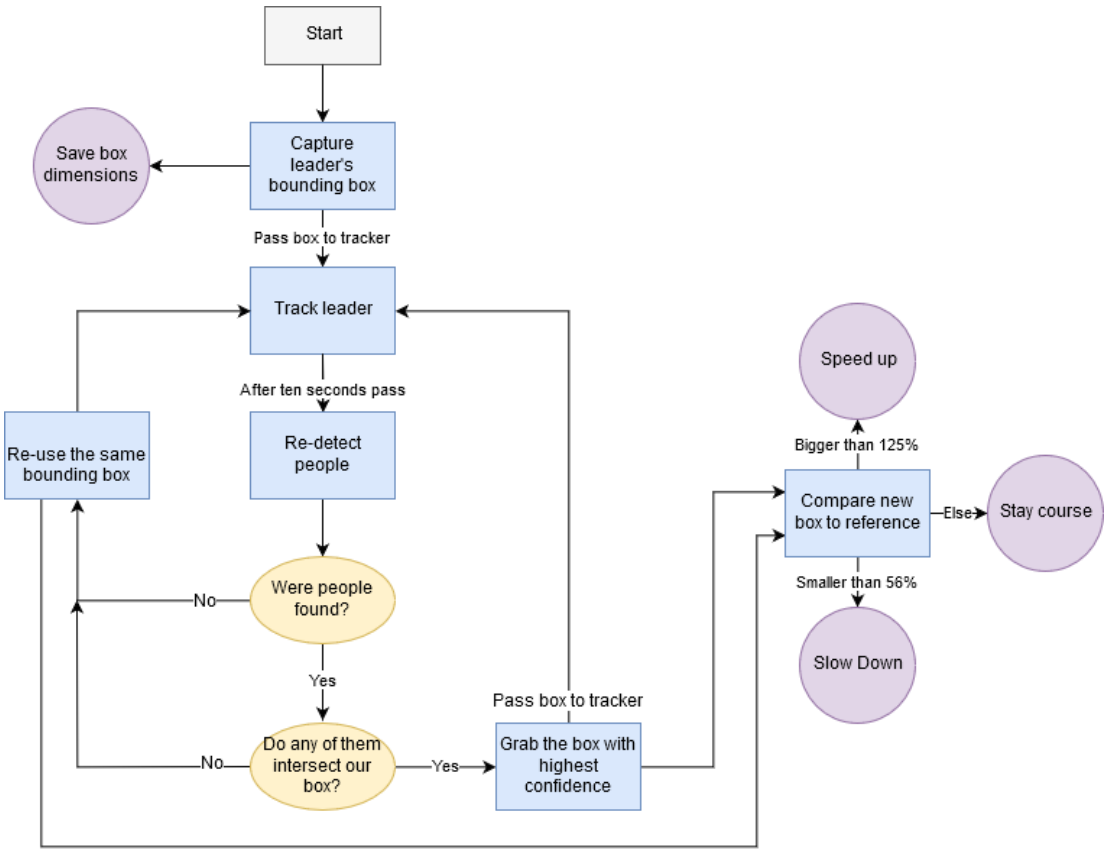


Figure 6: Flow Diagram of Crowd Tracker Program Logic

4. Operation

Since the entirety of the the Tour Group Tracker subsystem consists of image processing all steps of validation and execution are performed visually. Currently the program is set to display the images that it is processing on the monitor but when the systems are fully integrated those commands will be omitted since there is no screen.

As the code currently stands the way that tests are validated is through visual inspection of the output from both the console and the displayed images. During the initial calibration the program outputs an image of the first video frame with the detections drawn on it. The system continues on with a window displaying the frames with the tracking boundary drawn over it. When re-calibration begins another window is created showing all the new detections and the current tracker box. For each of these tests validation was a matter of visual confirmation. First a visual confirmation that the box detected the only person in the frame, secondly a visual confirmation that tracking is successful as the program iterates over the video frames, and lastly as the tracker is re-initialized a visual confirmation that the correct detection was used to reiterate the tracker. The other component, the distance response, is confirmed through the console. The ratio of the new box area to the reference area is output to the console so the following message of ‘just right’, ‘too far’, or ‘too close’ is cross-referenced with the conditions to confirm.

During normal operation the program will be validated through the real responses in the drone. With no screen to reference we must rely on what actually happens to confirm functions. Range responses and tracking can be checked through moving further and closer to the drone to see the response. If the appropriate action is taken then the tracker is tracking the correct person and the responses execute properly. Should the responses be inappropriate or there be no response then the tracking failed.

Once everything is complete and integrated the system will be unnoticeable for the tourists (save for the initial calibration) unless a malfunction occurs. Were one of the components of this subsystem to fail then the drone will still be able to continue the tour either after an attempt to recalibrate or a surrender should the camera itself fail. An announcement may be given stating the dynamic speed has been disabled and that the crowd must keep up.

5. Data Collection

Much of the data gathered is simply a yes or no to whether a condition executed properly. Being an image processing heavy subsystem means visual confirmation was required for everything tested.

5.1. Relative Distance Data

The first data to be collected was how the area of the boundary boxes changed with respect to the distance. The crowd leader’s boundary box area is used as a reference as opposed to an actual calculation of distance using focal length because of the variation of human height. By using the initial boundary box area we dynamically create a constant tailored to that person’s height and width as opposed to dictating an average that would “hopefully” work for everyone.

By using the box area instead of a direct distance a method needed to be established to determine how much they’ve changed distance. The ratio between the new area and the reference area was chosen. To determine this a test was performed where a teammate stood on a measuring tape and a test program was set to run such that the ratio of the box to the new area was calculated and printed. Several iterations were run such that the boundary box varied

naturally and an average box area for that distance was calculated. Finally, the percent difference between the boxes at the maximum and minimum distances were calculated for those averages, giving us an average percentage change from the reference at that distance.

Reference Area = 18432 px^2

Distance (ft)	Avg. Ratio to Reference
25	0.56
7	1.22

Table 1: Distance Ratios

5.2. Tracker Decision

The next set of data that needed to be collected was which tracker to be used. After narrowing down to the Median Flow and the MOSSE trackers due to their speed and efficiency it had to be chosen which tracker performed the best at its job. A set of cases were set up and each tracker was initialized to find their strengths and weaknesses. Each of them performed as expected. Median Flow is the least costly but fails under occlusion and large motion. This held true as during the occlusion and large motion tests the tracker failed. The data has been compiled in the following table.

Test / Tracker	MOSSE	Median Flow
Normal Motion	Pass	Pass
Occlusion (with movement)	Pass	Fail
Large Motion	Fail	Fail

Table 2: Tracker Test Cases

As can be seen in the table above both trackers did not do well with large movement. This is mainly because they are meant for slower normalized movement (such as walking down a hallway). As lamentable as a failing test is the choice is clear; the MOSSE tracker failed the least and thus was the obvious choice. Both succeeded in normal fluid motion but only the MOSSE passed the test where a teammate passed in front of the teammate being tracked. The MOSSE tracker successfully reset itself on the right person after being occluded. For now, large motion will be treated as an anomaly seeing as a crowd of tourists are not expected to make wild sweeping motions across the hallways.

5.3. Logic Validation and Performance Measurement

The tests for whether the logic for the various components of the subsystem were correct are purely boolean; did the correct thing happen? The code was run and the output images and console text analyzed to see if what should have happened in that scenario happened. This data is not wholly quantifiable beyond the fact that everything worked as expected. An example of a test is shown below. In the image we can see the output of the detector, the current tracker, and the console. In the console we see the correct output that the person being tracked is not too far away. We can also see the three humans detected previously and that in the above window the correct person (in the middle) is still being tracked.

Since image processing is difficult on the CPU and we are using a mere Raspberry Pi the total load on the CPU needed to be measured. At 5 fps, the program averaged about 30% load on the CPU with a brief spike to 53% when detection is performed.

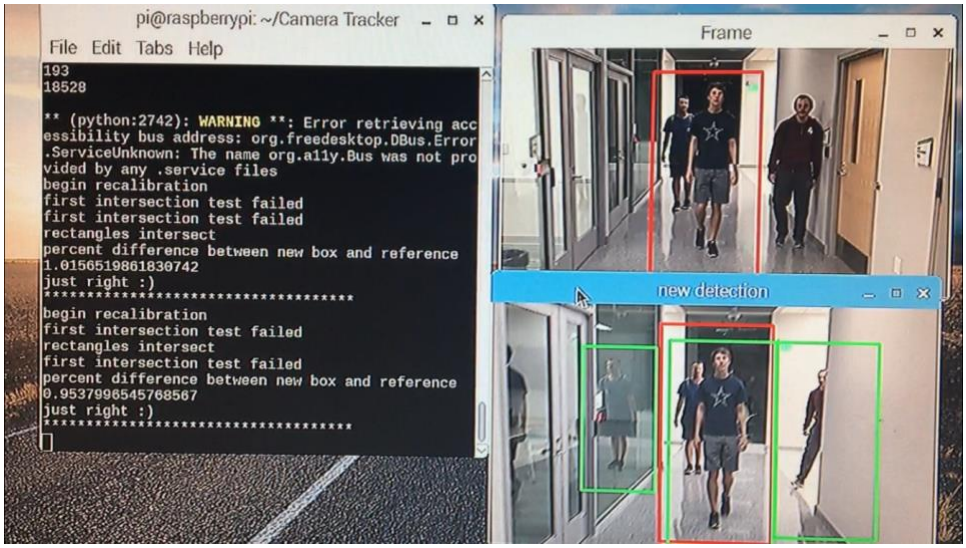


Figure 7: Example of Program Output, Used for Validation

6. Accomplishments

There were a number of landmark accomplishments in the development of the Tour Group Tracker subsystem. Small programs were written to test the various components that makeup the entire subsystem for testing and verification. Interfacing them together was another issue.

The first hurdle that had to be passed was actually choosing to use the camera for this subsystem. There were a few proposed ideas that were tossed around and considered before the camera due to the relative complexity. It seemed overkill to use a camera and image processing to simply detect how far away a crowd was from the drone. As it turned out the complexity was important as most other methods, namely Bluetooth, proved too simple to provide accurate measurements. Bluetooth signal to distance conversion was the initial choice for tracking the crowd distance. This proved disastrous as Bluetooth signals are not reliable. With all other choices stricken the camera was the only choice left.

Once the camera was chosen and the various components of the actual tracking were confirmed to work the problem of integration became apparent. Many hours were spent attempting to get the HOG human detector to interface with the tracking; mostly due to ignorance and a lack of documentation or examples. With plenty of work they eventually integrated.

The last big accomplishment was the logic for re-initializing the tracker. Many more hours were spent watching trial after trial fail due to an error in logic somewhere. The loops were built outside in with the beginning and ends of everything written first and the more complex inner functions added afterward. First the loop to break after a certain amount of time was created with the loop to perform new detections following. With new detections performed there needed to be a way to iterate through them and eliminate unnecessary ones. This as more conditions and checks were needed as the logic was amended and new cases needed to be considered.

In the end a working program that could detect a crowd leader, track them, reinitialize itself, and respond to their distance was created.

7. Future Work

While the program works well and takes many possible cases into consideration there is still more than can be done. The re-initialization logic could be slimmed down and less clunky with possible errors and shortcomings eliminated with more time. More conditions can be implemented for possible cases not yet considered. The greatest work that still must be finished is getting a live version of the program working. Currently the program reads a video as opposed to the camera itself. It was decided that getting the program working on videos first would be more beneficial as more consistent tests could be performed and validation would be a video away as opposed to requiring a computer desk in the middle of an empty hallway to perform the tracking and detection checks live.

8. Conclusion

A lot was learned this semester. What was originally thought to be a hardware heavy project turned out to be one where everyone ended up programming. I learned a lot about programming in general as well as Python specifically. Image processing was a subject I had never dipped my feet into and for this project I dove into the deep end. This required lot of research into how detection is done and what others have accomplished with it. I'd like to say that I have gained a few new skill sets through this class and I am proud of the work that I have accomplished. The Tour Group Tracker subsystem is by no means perfect and will definitely need more work for it to be as robust as would be necessary for something that will be used by Texas A&M. On the whole the project came out better than was expected at our low points. All of us encountered troubles that we did not expect but were able to adapt and overcome them to get to the point where, even with compromise, we achieved what we set out to do.

Zach Drone

Grayson Vansickle

Mobile Application Subsystem

REVISION – 1.2
5 December 2018

Table of Contents

List of Figures

List of Tables

No tables at this time.

1. Introduction

The purpose of the Mobile Application subsystem of the Zach Drone, which is an automated drone tour throughout the new Zachry Engineering Education Complex, is to allow the user to connect to the drone and tell it which tour to take. The mobile application will also allow the user to pause and resume the tours if someone in their tour group needs to use the bathroom or needs to stop for an extended period of time, and the user will be able to end the current tour and select a new tour. This subsystem will involve a user interface between the user and the drone and will be connected using a WiFi signal to communicate between the mobile application and the microcontroller on the drone, a Raspberry Pi 3 B. The mobile application will be available for the user to download on both Android and iOS platforms. Based on the user input, this subsystem interacts with the navigation subsystem to communicate which tour and tour destinations the user has picked. The user input is a major factor regarding the navigation of the drone, since different tours will need to be navigated differently in order to visit all of the destinations specific to the tour. The mobile application consists of a main page, an art tour page, a classroom tour page, a basic tour page and a drone page. The three tour pages will show the user which destinations are on the tour of the page they selected and will allow the user to start the tour on the page. The drone page will allow the user to connect and disconnect from the drone, along with pause, resume and end the current tour. The Raspberry Pi 3 B will have a message decoder running to ensure the drone performs the correct action in response to the user's input. The subsystem overview is shown in Figure 1. This document will describe the theory, design, operation, data collection and validation of the mobile application subsystem.

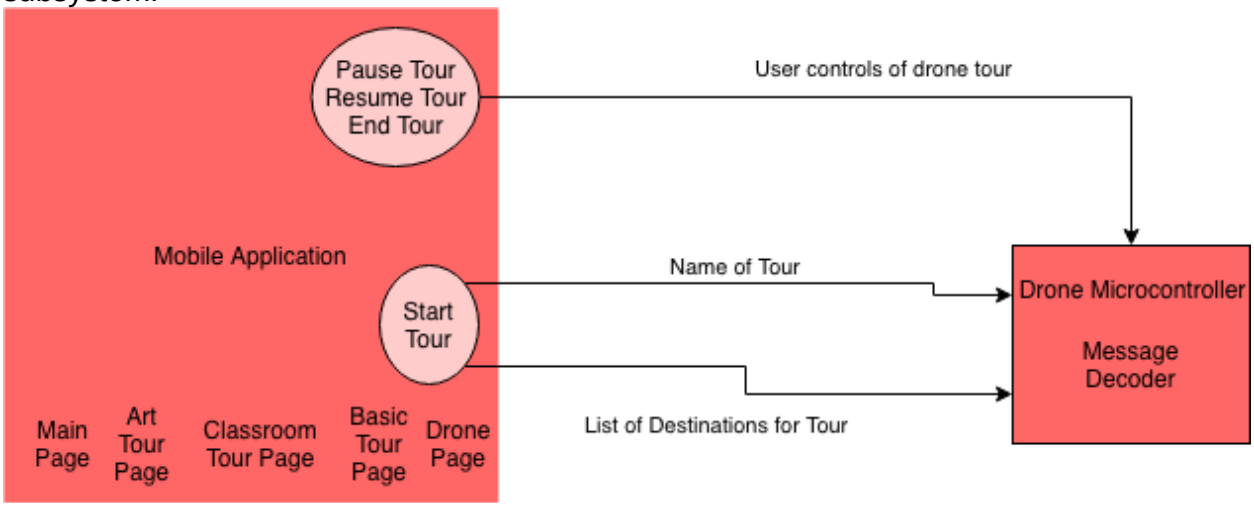


Figure 1: Mobile Application Subsystem

2. Theory

2.1. Mobile Application Code Logic

The mobile application will connect to the drone's onboard Raspberry Pi 3 B. This connection will allow the user to communicate with the drone to give it commands. The logic

used in the iOS and Android applications follow the diagram below. The logic in the figure gives the user some control of what the application sends to the drone but restricts them to an extent that they won't make the drone malfunction due to an overload of messages. The users must first connect to the drone before any other function of the application will work. After the user connects to the drone, they will then be able to start a tour and disconnect from the drone. The pause and end tour buttons only function after the user has started a tour. The resume tour button only functions after the user has paused a tour.

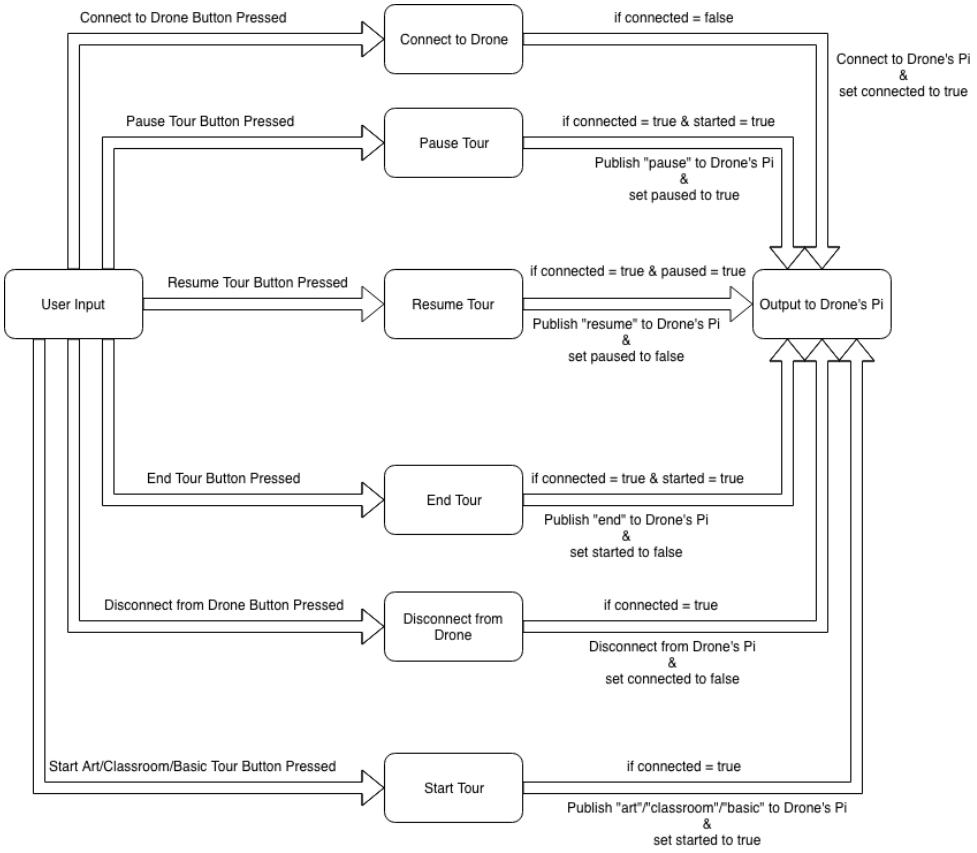


Figure 2: Application Logic used for the code

3. Design

3.1. Swift Code

The backend of the iOS application is written using Xcode in the Swift programming language. The logic of the the following swift files follows the logic defined in Figure 2 in the Theory section.

3.1.1. MainViewController.swift

MainViewController.swift contains only one function and that function is used to load the view. This function is needed so that the iOS application shows the correct page of the app. There is a print statement included in this function which was used when debugging that shows that the correct view was loaded.

```swift
import UIKit

class MainViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        print("Main View Loaded")
    }
}
```

Figure 3: Load MainViewController function

3.1.2. DroneViewController.swift

DroneViewController.swift contains a struct used for the client details and six functions. This struct contains the client itself and three boolean values used to make sure the user can online push certain buttons. The client uses the CocoaMQTT external library which is given a client id name, a host ip address and a port number. In this case the client id name is Zach Drone iOS Device, the host ip address is the ip address of the drone's Raspberry Pi 3 B on the WiFi network, and the port number is 1883. The three boolean values are connected, paused and started. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The six functions in DroneViewController.swift are load view, connect button, pause button, resume button, end button and disconnect button functions. The load view function is the same as the load view function in MainViewController.swift, but it loads the drone view instead of the main view.

```swift
struct Client {
    //home WIFI
    //static let mqttClient = CocoaMQTT(clientID: "Zach Drone iOS Device", host: "192.168.1.138", port: 1883)
    //Hotspot
    //static let mqttClient = CocoaMQTT(clientID: "Zach Drone iOS Device", host: "172.20.10.10", port: 1883)
    //Zach Guest
    //static let mqttClient = CocoaMQTT(clientID: "Zach Drone iOS Device", host: "10.236.61.140", port: 1883)
    //TAMU WPA
    static let mqttClient = CocoaMQTT(clientID: "Zach Drone iOS Device", host: "10.236.3.30", port: 1883)
    static var connected = Bool()
    static var paused = Bool()
    static var started = Bool()
}

class DroneViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        print("Drone View Loaded")
    }
```

Figure 4: Client struct and load view function defined

The connect button function is executed when the Connect to Drone button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value is true. If the connected value is true then the function will publish the message "connect" to the topic used for this project, "zach/drone/rpi/tour", and print "Already connected to drone" for debugging. There will be an alert that pops on the screen to tell the user that they have already connected to the drone. If the connected value is false then the function will connect the the

mqttClient defined in the Client struct, print the client's host name along with connect, and set the connected value to true.

```
//Executes when Connect to Drone button gets pressed
@IBAction func connectButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "connect")
        print("Already connected to drone")
        let alert = UIAlertController(title: "Did you already connect to the drone?", message: "You can only press the \"Connect to Drone\" button once.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else {
        Client.mqttClient.connect()
        print(Client.mqttClient.host + "  connect")
        Client.connected = true
    }
}
```

Figure 5: Connect button function defined

The pause button function is executed when the Pause Tour button is pressed. This function checks to see if the drone is already connected to first and a tour has already been started, if the connected and started boolean values are true. If the connected and started values are true then the function will publish the message "pause" to the topic used for this project, "zach/drone/rpi/tour", print the client's host name along with pause, and set the paused value to true. If the connected boolean value is true but the started boolean value is false then the function will print "Need to start a tour first" for debugging and an alert will pop on the screen to tell the user that they have to start a tour before pausing a tour. If the connected and started boolean values are false then the function will print "Need to connect to drone and start a tour first" for debugging and an alert will pop on the screen to tell the user that they have to connect to the drone and start a tour before pausing a tour.

```
//Executes when Pause Tour button gets pressed
@IBAction func pauseButton(_ sender: UIButton) {
    if(Client.connected && Client.started) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "pause")
        print(Client.mqttClient.host + "  pause")
        Client.paused = true
    }
    else if(Client.connected && !Client.started) {
        print("Need to start a tour first")
        let alert = UIAlertController(title: "Did you start a tour?", message: "You must press the \"Start Tour\" button before pressing \"Pause Tour\" button.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else {
        print("Need to connect to drone and start a tour first")
        let alert = UIAlertController(title: "Did you connect to the drone and start a tour?", message: "You must press the \"Connect to Drone\" and \"Start Tour\" buttons
            before pressing the \"Pause Tour\" button.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 6: Pause button function defined

The resume button function is executed when the Resume Tour button is pressed. This function checks to see if the drone is already connected to first and a tour has already been paused, if the connected and paused boolean values are true. If the connected and paused values are true then the function will publish the message "resume" to the topic used for this project, "zach/drone/rpi/tour", print the client's host name along with resume, and set the paused value to false. If the connected boolean value is true but the paused boolean value is false then the function will print "Need to pause tour first" for debugging and an alert will pop on the screen to tell the user that they have to pause a tour before resuming a tour. If the connected and paused boolean values are false then the function will print "Need to connect to drone and pause a tour first" for debugging and an alert will pop on the screen to tell the user that they have to connect to the drone and pause a tour before resuming a tour.

```
//Executes when Resume Tour button gets pressed
@IBAction func resumeButton(_ sender: UIButton) {
    if(Client.connected && Client.paused) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "resume")
        print(Client.mqttClient.host + "  resume")
        Client.paused = false
    }
    else if(Client.connected && !Client.paused) {
        print("Need to pause tour first")
        let alert = UIAlertController(title: "Did you pause a tour?", message: "You must press the \"Pause Tour\" button before pressing \"Resume Tour\" button.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else {
        print("Need to connect to drone and pause tour first")
        let alert = UIAlertController(title: "Did you connect to the drone and pause a tour?", message: "You must press the \"Connect to Drone\" and \"Pause Tour\" buttons
            before pressing the \"Resume Tour\" button.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 7: Resume button function defined

The end button function is executed when the End Tour button is pressed. This function checks to see if the drone is already connected to first and a tour has already been started, if the connected and started boolean values are true. If the connected and started values are true then the function will publish the message “end” to the topic used for this project, “zach/drone/rpi/tour”, print the client’s host name along with end, and set the started value to false. If the connected boolean value is true but the started boolean value is false then the function will print “Need to start a tour first” for debugging and an alert will pop on the screen to tell the user that they have to start a tour before ending a tour. If the connected and started boolean values are false then the function will print “Need to connect to drone and start a tour first” for debugging and an alert will pop on the screen to tell the user that they have to connect to the drone and start a tour before ending a tour.

```
//Executes when End Tour button gets pressed
@IBAction func endButton(_ sender: UIButton) {
    if(Client.connected && Client.started) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "end")
        print(Client.mqttClient.host + "  end")
        Client.started = false
    }
    else if(Client.connected && !Client.started) {
        print("Need to start a tour first")
        let alert = UIAlertController(title: "Did you start a tour?", message: "You must press the \"Start Tour\" button before pressing \"End Tour\" button.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else {
        print("Need to connect to drone and start a tour first")
        let alert = UIAlertController(title: "Did you connect to the drone and start a tour?", message: "You must press the \"Connect to Drone\" and \"Start Tour\" buttons
            before pressing the \"End Tour\" button.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 8: End button function defined

The disconnect button function is executed when the Disconnect from Drone button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value is true. If the connected value is true then the function will publish the message “disconnect” to the topic used for this project, “zach/drone/rpi/tour”, print the client’s host name and disconnect, disconnect the mqttClient defined the the Client struct, and set the connected value to false. If the connected value is false then the function will print “Need to connect to drone first” for debugging and an alert will pop on screen to tell the user that they have to connect to the drone before doing anything else.

```swift
//Executes when Disconnect from Drone button gets pressed
@IBAction func disconnectButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "disconnect")
        print(Client.mqttClient.host + "  disconnect")
        Client.mqttClient.disconnect()
        Client.connected = false
    }
    else {
        print("Need to connect to drone first")
        let alert = UIAlertController(title: "Did you connect to the drone?", message: "You must press the \"Connect to Drone\" button before pressing any other buttons.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 9: Disconnect button function defined

3.1.3. ArtViewController.swift

ArtViewController.swift contains two functions and an array of strings. The two functions are the view load and start art button functions. The array of strings contains the list of all the destinations that the art tour will visit. The load view function is the same as the load view function in MainViewController.swift, but it loads the art view instead of the main view.

```swift
class ArtViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        print("Art Tour View Loaded")

        // Do any additional setup after loading the view.
    }

    let destinations = ["1.  Anadarko Welcome Center", "2.  Smoke Painting #44 by Rosemarie Fiore", "3.  A-Gadda-Da-Vida by Jay Shinn", "4.
        Lathocyte by Andy Vogt", "5.  Transcending Realms; Chaos and Flow, Love and Fear by Lyndi Sales", "6.  Shapeshifting to Transcend Limbo
        by Lyndi Sales", "7.  What it Takes to Make by Daniel Rozin", "8.  Pulse by Daniel Canogar", "9.  Prototype for Stellar Interloper
        (Silver Surfer) by Inigo Manglano-Ovalle", "10. Infinitesimals by Rusty Scruby"]
```

Figure 10: Load view function and art tour destinations defined

The start art button function is executed when the Start Art Tour button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value from the Client struct in the DroneViewController.swift file is true. If the connected value is true then the function will publish the message “art” to the topic used for this project, “zach/drone/rpi/tour”, send all the destinations in the array of strings to the same topic, print the client’s host name from the Client struct in the DroneViewController.swift file and art, and set the started boolean value from the Client struct in the DroneViewController.swift file to true. If the connected value is false then the function will print “Need to connect to drone first” for debugging and an alert will pop on screen to tell the user that they have to connect to the drone before doing anything else.

```swift
//Executes when Start Art Tour button gets pressed
@IBAction func startArtButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "art")
        for destination in destinations {
            Client.mqttClient.publish("zach/drone/rpi/tour", withString: destination)
        }
        print(Client.mqttClient.host + "  art")
        Client.started = true;
    }
    else {
        print("Need to connect to drone first")
        let alert = UIAlertController(title: "Did you connect to the drone?", message: "You must press the \"Connect to Drone\" button on the
            Drone (far right) page before pressing any other buttons.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 11: Art start function defined

3.1.4. ClassroomViewController.swift

ClassroomViewController.swift contains two functions and an array of strings. The two functions are the view load and start classroom button functions. The array of strings contains the list of all the destinations that the classroom tour will visit. The load view function is the same as the load view function in MainViewController.swift, but it loads the classroom view instead of the main view.

```
class ClassroomViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        print("Classroom Tour View Loaded")

        // Do any additional setup after loading the view.
    }

    let destinations = ["1. Anadarko Welcome Center", "2. 106 - Virginia Brown Atrium", "3. 282 - Leach Learning Resource Center", "4. 212 - Small
        Learning Studio", "5. 343A", "6. 318 - Open Study Area", "7. 317 - Design Studio", "8. Large Learning Studios", "9. Fischer Engineering
        Design Center"]
```

Figure 12: Load view function and classroom tour destinations defined

The start classroom button function is executed when the Start Classroom Tour button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value from the Client struct in the DroneViewController.swift file is true. If the connected value is true then the function will publish the message “classroom” to the topic used for this project, “zach/drone/rpi/tour”, send all the destinations in the array of strings to the same topic, print the client’s host name from the Client struct in the DroneViewController.swift file and classroom, and set the started boolean value from the Client struct in the DroneViewController.swift file to true. If the connected value is false then the function will print “Need to connect to drone first” for debugging and an alert will pop on screen to tell the user that they have to connect to the drone before doing anything else.

```
//Executes when Start Classroom Tour button gets pressed
@IBAction func startClassroomButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "classroom")
        for destination in destinations {
            Client.mqttClient.publish("zach/drone/rpi/tour", withString: destination)
        }
        print(Client.mqttClient.host + "  classroom")
        Client.started = true
    }
    else {
        print("Need to connect to drone first")
        let alert = UIAlertController(title: "Did you connect to the drone?", message: "You must press the \"Connect to Drone\" button on the
            Drone (far right) page before pressing any other buttons.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 13: Classroom start function defined

3.1.5. BasicViewController.swift

BasicViewController.swift contains two functions and an array of strings. The two functions are the view load and start basic button functions. The array of strings contains the list of all the destinations that the basic tour will visit. The load view function is the same as the load view function in MainViewController.swift, but it loads the basic view instead of the main view.

```
class BasicViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        print("Basic Tour View Loaded")

        // Do any additional setup after loading the view.
    }

    let destinations = ["1.  Anadarko Welcome Center", "2.  Smoke Painting #44 by Rosemarie Fiore", "3.  Fischer Engineering Design Center", "4.
        Common Labs", "5.  A-Gadda-Da-Vida by Jay Shinn", "6.  HPE Tech Deck", "7.  481 - Academic Advising", "8.  Infinitesimals by Rusty Scruby",
        "9.  Photo Spot", "10. Large Learning Studios", "11. 343A", "12. 318 - Open Study Area", "13. 317 - Design Studio", "14. 212 - Small
        Learning Studio", "15. Transcending Realms; Chaos and Flow, Love and Fear by Lyndi Sales", "16. 280 - ConocoPhillips Atrium", "17. 282 -
        Leach Learning Resource Center", "18. Pulse by Daniel Canogar", "19. What it Takes to Make by Daniel Rozin", "20. Prototype for Stellar
        Interloper (Silver Surfer) by Inigo Manglano-Ovalle", "21. 284", "22. Starbucks - 271", "23. Shapeshifting to Transcend Limbo by Lyndi
        Sales", "24. Lathocyte by Andy Vogt", "25. South-facing window", "26. 106 - Virginia Brown Atrium"]
```

Figure 14: Load view function and basic tour destinations defined

The start basic button function is executed when the Start Basic Tour button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value from the Client struct in the DroneViewController.swift file is true. If the connected value is true then the function will publish the message “basic” to the topic used for this project, “zach/drone/rpi/tour”, send all the destinations in the array of strings to the same topic, print the client’s host name from the Client struct in the DroneViewController.swift file and basic, and set the started boolean value from the Client struct in the DroneViewController.swift file to true. If the connected value is false then the function will print “Need to connect to drone first” for debugging and an alert will pop on screen to tell the user that they have to connect to the drone before doing anything else.

```
//Executes when Start Basic Tour button gets pressed
@IBAction func startBasicButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "basic")
        for destination in destinations {
            Client.mqttClient.publish("zach/drone/rpi/tour", withString: destination)
        }
        print(Client.mqttClient.host + "  basic")
        Client.started = true
    }
    else {
        print("Need to connect to drone first")
        let alert = UIAlertController(title: "Did you connect to the drone?", message: "You must press the \"Connect to Drone\" button on the
            Drone (far right) page before pressing any other buttons.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 15: Basic start function defined

3.2. Java Code

The backend of the Android application is written using Android Studio in the Java programming language. The logic of the the following java files follows the logic defined in Figure 2 in the Theory section.

3.2.1. ArtActivity.java

ArtActivity.java contains three strings, three boolean values, and two main functions, on create and art functions. The three strings are used to pass the boolean values between the different pages within the application. The three strings are ART_CONNECT, ART_START and ART_PAUSE. The ART_CONNECT string is used to store the connected boolean value. The ART_START string is used to store the started boolean value. The ART_PAUSE string is used to store the started boolean value. The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused.

The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application. The art function has a string array that has all of the destination that the art tour will visit.

```
public class ArtActivity extends AppCompatActivity {

    public static final String ART_CONNECT = "com.example.graysonvansickle.zachdrone.ART_CONNECT";
    public static final String ART_START = "com.example.graysonvansickle.zachdrone.ART_START";
    public static final String ART_PAUSE = "com.example.graysonvansickle.zachdrone.ART_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;
```

Figure 16: Art Activity strings and booleans defined

The on create function starts the art activity layout xml file and displays this layout to the application. The function then sets the art tour item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named ART_CONNECT, ART_START or ART_PAUSE. If the intent has an ART_CONNECT extra then the connected boolean value is set to the value of the intent's ART_CONNECT extra. If the intent has an ART_START extra then the started boolean value is set to the value of the intent's ART_START extra. If the intent has an ART_PAUSE extra then the paused boolean value is set to the value of the intent's ART_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_art);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 1);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(ART_CONNECT))
        connected = getIntent().getBooleanExtra(ART_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + ART_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(ART_START))
        started = getIntent().getBooleanExtra(ART_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + ART_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(ART_PAUSE))
        paused = getIntent().getBooleanExtra(ART_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + ART_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 17: Art Activity on create function defined

If the user selected the Main page in the bottom navigation bar, then a new intent is made that connects the ArtActivity class to the MainActivity class. The connected boolean value is then added to this intent as an extra named MAIN_CONNECT. The started boolean value is then added to this intent as an extra named MAIN_START. The paused boolean value is then added to this intent as an extra named MAIN_PAUSE. The intent created from the ArtActivity class to the MainActivity class is then started so the application will then run the MainActivity.java file. If the user selected the Classroom Tour page in the bottom navigation bar, then a new intent is made that connects the ArtActivity class to the ClassroomActivity class. The connected boolean value is then added to this intent as an extra named CLASSROOM_CONNECT. The started boolean value is then added to this intent as an extra named CLASSROOM_START. The paused boolean value is then added to this intent as an extra named CLASSROOM_PAUSE. The intent created from the ArtActivity class to the ClassroomActivity class is then started so the application will then run the ClassroomActivity.java file. If the user selected the Basic Tour page in the bottom navigation bar, then a new intent is made that connects the ArtActivity class to the BasicActivity class. The connected boolean value is then added to this intent as an extra named BASIC_CONNECT. The started boolean value is then added to this intent as an extra named BASIC_START. The paused boolean value is then added to this intent as an extra named BASIC_PAUSE. The intent created from the ArtActivity class to the BasicActivity class is then started so the application will then run the BasicActivty.java file. If the user selected the Drone page in the bottom navigation bar, then a new intent is made that connects the ArtActivity class to the DroneActivity class. The connected boolean value is then added to this intent as an extra named DRONE_CONNECT. The started boolean value is then added to this intent as an extra named DRONE_START. The paused boolean value is then added to this intent as an extra named DRONE_PAUSE. The intent created from the ArtActivity class to the DroneActivity class is then started so the application will then run the DroneActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                Intent intent1 = new Intent( packageContext: ArtActivity.this, MainActivity.class);
                intent1.putExtra(MainActivity.MAIN_CONNECT, connected);
                intent1.putExtra(MainActivity.MAIN_START, started);
                intent1.putExtra(MainActivity.MAIN_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_dashboard: //Art Tour
                break;
            case R.id.navigation_classroom: //Classroom Tour
                Intent intent2 = new Intent( packageContext: ArtActivity.this, ClassroomActivity.class);
                intent2.putExtra(ClassroomActivity.CLASSROOM_CONNECT, connected);
                intent2.putExtra(ClassroomActivity.CLASSROOM_START, started);
                intent2.putExtra(ClassroomActivity.CLASSROOM_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_basic: //Basic Tour
                Intent intent3 = new Intent( packageContext: ArtActivity.this, BasicActivity.class);
                intent3.putExtra(BasicActivity.BASIC_CONNECT, connected);
                intent3.putExtra(BasicActivity.BASIC_START, started);
                intent3.putExtra(BasicActivity.BASIC_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_drone: //Drone
                Intent intent4 = new Intent( packageContext: ArtActivity.this, DroneActivity.class);
                intent4.putExtra(DroneActivity.DRONE_CONNECT, connected);
                intent4.putExtra(DroneActivity.DRONE_START, started);
                intent4.putExtra(DroneActivity.DRONE_PAUSE, paused);
                startActivity(intent4);
                break;
        }
        return false;
    }
});
```

Figure 18: Art Activity switch pages function defined

The art function first checks to see if the connected boolean value is true. If connected is true then the function will create a new MqttMessage and this message is then set to art, an alert will pop up at the bottom of the screen that says art to tell the user that the art tour is start, and the started boolean value is set to true. The function then publishes the message to the topic and client defined in the DroneActivity class. The function then goes through the array of destinations and makes a new MqttMessage with each destination and publishes that message to the topic and client defined in the DroneActivity class. If connected is false then an alert will pop up at the bottom of the screen to tell the user that they must connect to the drone before starting the art tour.

```
@RequiresApi(api = Build.VERSION_CODES.N)
public void art(View v) {
    if(connected) {
        MqttMessage message = new MqttMessage();
        message.setPayload("art".getBytes());

        System.out.println("bool: " + connected);

        Toast.makeText( context: ArtActivity.this,  text: "art", Toast.LENGTH_LONG).show();

        started = true;

        String[] destinations = new String[] {"1.  Anadarko Welcome Center",
                "2.  Smoke Painting #44 by Rosemarie Fiore",
                "3.  A-Gadda-Da-Vida by Jay Shinn",
                "4.  Lathocyte by Andy Vogt",
                "5.  Transcending Realms; Chaos and Flow, Love and Fear by Lyndi Sales",
                "6.  Shapeshifting to Transcend Limbo by Lyndi Sales",
                "7.  What it Takes to Make by Daniel Rozin",
                "8.  Pulse by Daniel Canogar",
                "9.  Prototype for Stellar Interloper (Silver Surfer) by Inigo Manglano-Ovalle",
                "10. Infinitesimals by Rusty Scruby"};

        try {
            DroneActivity.client.publish(DroneActivity.topic, message);
            for(String d : destinations){
                System.out.println(d);
                message = new MqttMessage();
                message.setPayload(d.getBytes());
                DroneActivity.client.publish(DroneActivity.topic, message);
            }
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else
        Toast.makeText( context: ArtActivity.this,  text: "Unable to Start Art Tour! Must Connect to Drone first!", Toast.LENGTH_LONG).show();
}
```

Figure 19: Art Activity art function defined

3.2.2. ClassroomActivity.java

ClassroomActivity.java contains three strings, three boolean values, and two main functions, on create and classroom functions. The three strings are used to pass the boolean values between the different pages within the application. The three strings are CLASSROOM_CONNECT, CLASSROOM_START and CLASSROOM_PAUSE. The CLASSROOM_CONNECT string is used to store the connected boolean value. The CLASSROOM_START string is used to store the started boolean value. The CLASSROOM_PAUSE string is used to store the started boolean value. The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application. The classroom function has a string array that has all of the destination that the classroom tour will visit.

```
public class ClassroomActivity extends AppCompatActivity {

    public static final String CLASSROOM_CONNECT = "com.example.graysonvansickle.zachdrone.CLASSROOM_CONNECT";
    public static final String CLASSROOM_START = "com.example.graysonvansickle.zachdrone.CLASSROOM_START";
    public static final String CLASSROOM_PAUSE = "com.example.graysonvansickle.zachdrone.CLASSROOM_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;
```

Figure 20: Classroom Activity strings and booleans defined

The on create function starts the classroom activity layout xml file and displays this layout to the application. The function then sets the classroom tour item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named CLASSROOM_CONNECT, CLASSROOM_START or CLASSROOM_PAUSE. If the intent has

an CLASSROOM_CONNECT extra then the connected boolean value is set to the value of the intent's CLASSROOM_CONNECT extra. If the intent has an CLASSROOM_START extra then the started boolean value is set to the value of the intent's CLASSROOM_START extra. If the intent has an CLASSROOM_PAUSE extra then the paused boolean value is set to the value of the intent's CLASSROOM_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_classroom);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 2);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(CLASSROOM_CONNECT))
        connected = getIntent().getBooleanExtra(CLASSROOM_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + CLASSROOM_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(CLASSROOM_START))
        started = getIntent().getBooleanExtra(CLASSROOM_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + CLASSROOM_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(CLASSROOM_PAUSE))
        paused = getIntent().getBooleanExtra(CLASSROOM_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + CLASSROOM_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 21: Classroom Activity on create function defined

If the user selected the Main page in the bottom navigation bar, then a new intent is made that connects the ClassroomActivity class to the MainActivity class. The connected boolean value is then added to this intent as an extra named MAIN_CONNECT. The started boolean value is then added to this intent as an extra named MAIN_START. The paused boolean value is then added to this intent as an extra named MAIN_PAUSE. The intent created from the ClassroomActivity class to the MainActivity class is then started so the application will then run the MainActivity.java file. If the user selected the Art Tour page in the bottom navigation bar, then a new intent is made that connects the ClassroomActivity class to the ArtActivity class. The connected boolean value is then added to this intent as an extra named ART_CONNECT. The started boolean value is then added to this intent as an extra named ART_START. The paused boolean value is then added to this intent as an extra named ART_PAUSE. The intent created from the ClassroomActivity class to the ArtActivity class is

then started so the application will then run the ArtActivity.java file. If the user selected the Basic Tour page in the bottom navigation bar, then a new intent is made that connects the ClassroomActivity class to the BasicActivity class. The connected boolean value is then added to this intent as an extra named BASIC_CONNECT. The started boolean value is then added to this intent as an extra named BASIC_START. The paused boolean value is then added to this intent as an extra named BASIC_PAUSE. The intent created from the ClassroomActivity class to the BasicActivity class is then started so the application will then run the BasicActivty.java file. If the user selected the Drone page in the bottom navigation bar, then a new intent is made that connects the ClassroomActivity class to the DroneActivity class. The connected boolean value is then added to this intent as an extra named DRONE_CONNECT. The started boolean value is then added to this intent as an extra named DRONE_START. The paused boolean value is then added to this intent as an extra named DRONE_PAUSE. The intent created from the ClassroomActivity class to the DroneActivity class is then started so the application will then run the DroneActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                Intent intent1 = new Intent( packageContext: ClassroomActivity.this, MainActivity.class);
                intent1.putExtra(MainActivity.MAIN_CONNECT, connected);
                intent1.putExtra(MainActivity.MAIN_START, started);
                intent1.putExtra(MainActivity.MAIN_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_dashboard: //Art Tour
                Intent intent2 = new Intent( packageContext: ClassroomActivity.this, ArtActivity.class);
                intent2.putExtra(ArtActivity.ART_CONNECT, connected);
                intent2.putExtra(ArtActivity.ART_START, started);
                intent2.putExtra(ArtActivity.ART_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_classroom: //Classroom Tour
                break;
            case R.id.navigation_basic: //Basic Tour
                Intent intent3 = new Intent( packageContext: ClassroomActivity.this, BasicActivity.class);
                intent3.putExtra(BasicActivity.BASIC_CONNECT, connected);
                intent3.putExtra(BasicActivity.BASIC_START, started);
                intent3.putExtra(BasicActivity.BASIC_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_drone: //Drone
                Intent intent4 = new Intent( packageContext: ClassroomActivity.this, DroneActivity.class);
                intent4.putExtra(DroneActivity.DRONE_CONNECT, connected);
                intent4.putExtra(DroneActivity.DRONE_START, started);
                intent4.putExtra(DroneActivity.DRONE_PAUSE, paused);
                startActivity(intent4);
                break;
        }
        return false;
    }
});
```

Figure 22: Classroom Activity switch pages function defined

The classroom function first checks to see if the connected boolean value is true. If connected is true then the function will create a new MqttMessage and this message is then set to classroom, an alert will pop up at the bottom of the screen that says classroom to tell the user that the classroom tour is start, and the started boolean value is set to true. The function then publishes the message to the topic and client defined in the DroneActivity class. The function then goes through the array of destinations and makes a new MqttMessage with each destination and publishes that message to the topic and client defined in the DroneActivity class. If connected is false then an alert will pop up at the bottom of the screen to tell the user that they must connect to the drone before starting the classroom tour.

```
public void classroom(View v) {
    if(connected) {
        MqttMessage message = new MqttMessage();
        message.setPayload("classroom".getBytes());

        System.out.println("bool: " + connected);

        Toast.makeText( context: ClassroomActivity.this,  text: "classroom", Toast.LENGTH_LONG).show();

        started = true;

        String[] destinations = new String[] {"1. Anadarko Welcome Center",
                "2. 106 - Virginia Brown Atrium",
                "3. 282 - Leach Learning Resource Center",
                "4. 212 - Small Learning Studio",
                "5. 343A",
                "6. 318 - Open Study Area",
                "7. 317 - Design Studio",
                "8. Large Learning Studios",
                "9. Fischer Engineering Design Center"};

        try {
            DroneActivity.client.publish(DroneActivity.topic, message);
            for(String d : destinations){
                System.out.println(d);
                message = new MqttMessage();
                message.setPayload(d.getBytes());
                DroneActivity.client.publish(DroneActivity.topic, message);
            }
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else
        Toast.makeText( context: ClassroomActivity.this,  text: "Unable to Start Classroom Tour! Must Connect to Drone first!", Toast.LENGTH_LONG).show();
}
```

Figure 23: Classroom Activity classroom function defined

3.2.3. BasicActivity.java

BasicActivity.java contains three strings, three boolean values, and two main functions, on create and basic functions. The three strings are used to pass the boolean values between the different pages within the application. The three strings are BASIC_CONNECT, BASIC_START and BASIC_PAUSE. The BASIC_CONNECT string is used to store the connected boolean value. The BASIC_START string is used to store the started boolean value. The BASIC_PAUSE string is used to store the started boolean value. The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application. The classroom function has a string array that has all of the destination that the basic tour will visit.

```
public class BasicActivity extends AppCompatActivity {

    public static final String BASIC_CONNECT = "com.example.graysonvansickle.zachdrone.BASIC_CONNECT";
    public static final String BASIC_START = "com.example.graysonvansickle.zachdrone.BASIC_START";
    public static final String BASIC_PAUSE = "com.example.graysonvansickle.zachdrone.BASIC_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;
```

Figure 24: Basic Activity strings and booleans defined

The on create function starts the basic activity layout xml file and displays this layout to the application. The function then sets the basic tour item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named BASIC_CONNECT, BASIC_START or BASIC_PAUSE. If the intent has an BASIC_CONNECT extra then the connected boolean value is set to the value of the intent's BASIC_CONNECT extra. If the intent has an BASIC_START extra then the started boolean value is set to the value of the intent's BASIC_START extra. If the intent has an BASIC_PAUSE extra then the paused

boolean value is set to the value of the intent's BASIC_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_basic);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 3);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(BASIC_CONNECT))
        connected = getIntent().getBooleanExtra(BASIC_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + BASIC_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(BASIC_START))
        started = getIntent().getBooleanExtra(BASIC_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + BASIC_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(BASIC_PAUSE))
        paused = getIntent().getBooleanExtra(BASIC_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + BASIC_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 25: Basic Activity on create function defined

If the user selected the Main page in the bottom navigation bar, then a new intent is made that connects the BasicActivity class to the MainActivity class. The connected boolean value is then added to this intent as an extra named MAIN_CONNECT. The started boolean value is then added to this intent as an extra named MAIN_START. The paused boolean value is then added to this intent as an extra named MAIN_PAUSE. The intent created from the BasicActivity class to the MainActivity class is then started so the application will then run the MainActivity.java file. If the user selected the Art Tour page in the bottom navigation bar, then a new intent is made that connects the BasicActivity class to the ArtActivity class. The connected boolean value is then added to this intent as an extra named ART_CONNECT. The started boolean value is then added to this intent as an extra named ART_START. The paused boolean value is then added to this intent as an extra named ART_PAUSE. The intent created from the BasicActivity class to the ArtActivity class is then started so the application will then run the ArtActivity.java file. If the user selected the Classroom Tour page in the bottom navigation bar, then a new intent is made that connects the BasicActivity class to the ClassroomActivity class. The connected boolean value is then added to this intent as an extra named CLASSROOM_CONNECT. The started boolean value is then added to this intent as an extra named CLASSROOM_START. The paused boolean value is then added to this intent as an extra named CLASSROOM_PAUSE. The intent created from the BasicActivity class to the

ClassroomActivity class is then started so the application will then run the ClassroomActivity.java file. If the user selected the Drone page in the bottom navigation bar, then a new intent is made that connects the BasicActivity class to the DroneActivity class. The connected boolean value is then added to this intent as an extra named DRONE_CONNECT. The started boolean value is then added to this intent as an extra named DRONE_START. The paused boolean value is then added to this intent as an extra named DRONE_PAUSE. The intent created from the BasicActivity class to the DroneActivity class is then started so the application will then run the DroneActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                Intent intent1 = new Intent( packageContext: BasicActivity.this, MainActivity.class);
                intent1.putExtra(MainActivity.MAIN_CONNECT, connected);
                intent1.putExtra(MainActivity.MAIN_START, started);
                intent1.putExtra(MainActivity.MAIN_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_dashboard: //Art Tour
                Intent intent2 = new Intent( packageContext: BasicActivity.this, ArtActivity.class);
                intent2.putExtra(ArtActivity.ART_CONNECT, connected);
                intent2.putExtra(ArtActivity.ART_START, started);
                intent2.putExtra(ArtActivity.ART_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_classroom: //Classroom Tour
                Intent intent3 = new Intent( packageContext: BasicActivity.this, ClassroomActivity.class);
                intent3.putExtra(ClassroomActivity.CLASSROOM_CONNECT, connected);
                intent3.putExtra(ClassroomActivity.CLASSROOM_START, started);
                intent3.putExtra(ClassroomActivity.CLASSROOM_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_basic: //Basic Tour
                break;
            case R.id.navigation_drone: //Drone
                Intent intent4 = new Intent( packageContext: BasicActivity.this, DroneActivity.class);
                intent4.putExtra(DroneActivity.DRONE_CONNECT, connected);
                intent4.putExtra(DroneActivity.DRONE_START, started);
                intent4.putExtra(DroneActivity.DRONE_PAUSE, paused);
                startActivity(intent4);
                break;
        }
        return false;
    }
});
```

Figure 26: Basic Activity switch pages function defined

The basic function first checks to see if the connected boolean value is true. If connected is true then the function will create a new MqttMessage and this message is then set to basic, an alert will pop up at the bottom of the screen that says basic to tell the user that the basic tour is start, and the started boolean value is set to true. The function then publishes the message to the topic and client defined in the DroneActivity class. The function then goes through the array of destinations and makes a new MqttMessage with each destination and publishes that message to the topic and client defined in the DroneActivity class. If connected is false then an alert will pop up at the bottom of the screen to tell the user that they must connect to the drone before starting the basic tour.

```
public void basic(View v) {
    if(connected) {
        MqttMessage message = new MqttMessage();
        message.setPayload("basic".getBytes());

        System.out.println("bool: " + connected);

        Toast.makeText( context: BasicActivity.this,  text: "basic", Toast.LENGTH_LONG).show();

        started = true;

        String[] destinations = new String[] {"1.  Anadarko Welcome Center",
                "2.  Smoke Painting #44 by Rosemarie Fiore",
                "3.  Fischer Engineering Design Center",
                "4.  Common Labs",
                "5.  A-Gadda-Da-Vida by Jay Shinn",
                "6.  HPE Tech Deck",
                "7.  481 - Academic Advising",
                "8.  Infinitesimals by Rusty Scruby",
                "9.  Photo Spot",
                "10. Large Learning Studios",
                "11. 343A",
                "12. 318 - Open Study Area",
                "13. 317 - Design Studio",
                "14. 212 - Small Learning Studio",
                "15. Transcending Realms; Chaos and Flow, Love and Fear by Lyndi Sales",
                "16. 280 - ConocoPhillips Atrium",
                "17. 282 - Leach Learning Resource Center",
                "18. Pulse by Daniel Canogar",
                "19. What it Takes to Make by Daniel Rozin",
                "20. Prototype for Stellar Interloper (Silver Surfer) by Inigo Manglano-Ovalle",
                "21. 284",
                "22. Starbucks - 271",
                "23. Shapeshifting to Transcend Limbo by Lyndi Sales",
                "24. Lathocyte by Andy Vogt",
                "25. South-facing window",
                "26. 106 - Virginia Brown Atrium"};

        try {
            DroneActivity.client.publish(DroneActivity.topic, message);
            for(String d : destinations){
                System.out.println(d);
                message = new MqttMessage();
                message.setPayload(d.getBytes());
                DroneActivity.client.publish(DroneActivity.topic, message);
            }
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else
        Toast.makeText( context: BasicActivity.this,  text: "Unable to Start Basic Tour! Must Connect to Drone first!", Toast.LENGTH_LONG).show();
}
```

Figure 27: Basic Activity basic function defined

3.2.4. DroneActivity.java

DroneActivity.java contains five strings, three boolean values, a MqttAndroidClient and six main functions. The three of the strings are used to pass the boolean values between the different pages within the application. The other two strings are used to store the client connection address and topic for the client. The six functions are on create, connect, pause, resume, end and disconnect functions. The three strings used to pass boolean values are DRONE_CONNECT, DRONE_START and DRONE_PAUSE. The two strings strings used to store the client information are MQTTHOST and topic. The DRONE_CONNECT string is used to store the connected boolean value. The DRONE_START string is used to store the started boolean value. The DRONE_PAUSE string is used to store the started boolean value. The MQTTHOST string is used to store the connection address for the client, which is a tcp connection to the ip address of the drone's Raspberry Pi 3 B and the port number 1883. The topic string is used to store the topic used for this project, which is "zach/drone/rpi/tour". The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-

function in it that deals with the changing the pages within the application. The classroom function has a string array that has all of the destination that the basic tour will visit.

```
public class DroneActivity extends AppCompatActivity {
    //static String MQTTHOST = "tcp://192.168.1.138:1883";    //home wifi
    //static String MQTTHOST = "tcp://172.20.10.10:1883";     //hotspot
    static String MQTTHOST = "tcp://10.236.3.30:1883";  //TAMU
    public static String topic = "zach/drone/rpi/tour";
    MqttMessage message = new MqttMessage();

    public static final String DRONE_CONNECT = "com.example.graysonvansickle.zachdrone.DRONE_CONNECT";
    public static final String DRONE_START = "com.example.graysonvansickle.zachdrone.DRONE_START";
    public static final String DRONE_PAUSE = "com.example.graysonvansickle.zachdrone.DRONE_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;

    public static MqttAndroidClient client;
```

Figure 28: Drone Activity strings and booleans defined

The on create function starts the drone activity layout xml file and displays this layout to the application. The function then sets the drone item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named DRONE_CONNECT, DRONE_START or DRONE_PAUSE. If the intent has an DRONE_CONNECT extra then the connected boolean value is set to the value of the intent's DRONE_CONNECT extra. If the intent has an DRONE_START extra then the started boolean value is set to the value of the intent's DRONE_START extra. If the intent has an DRONE_PAUSE extra then the paused boolean value is set to the value of the intent's DRONE_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_drone);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 4);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(DRONE_CONNECT))
        connected = getIntent().getBooleanExtra(DRONE_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + DRONE_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(DRONE_START))
        started = getIntent().getBooleanExtra(DRONE_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + DRONE_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(DRONE_PAUSE))
        paused = getIntent().getBooleanExtra(DRONE_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + DRONE_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 29: Drone Activity on create function defined

If the user selected the Main page in the bottom navigation bar, then a new intent is made that connects the DroneActivity class to the MainActivity class. The connected boolean value is then added to this intent as an extra named MAIN_CONNECT. The started boolean value is then added to this intent as an extra named MAIN_START. The paused boolean value is then added to this intent as an extra named MAIN_PAUSE. The intent created from the DroneActivity class to the MainActivity class is then started so the application will then run the MainActivity.java file. If the user selected the Art Tour page in the bottom navigation bar, then a new intent is made that connects the DroneActivity class to the ArtActivity class. The connected boolean value is then added to this intent as an extra named ART_CONNECT. The started boolean value is then added to this intent as an extra named ART_START. The paused boolean value is then added to this intent as an extra named ART_PAUSE. The intent created from the DroneActivity class to the ArtActivity class is then started so the application will then run the ArtActivity.java file. If the user selected the Classroom Tour page in the bottom navigation bar, then a new intent is made that connects the DroneActivity class to the ClassroomActivity class. The connected boolean value is then added to this intent as an extra named CLASSROOM_CONNECT. The started boolean value is then added to this intent as an extra named CLASSROOM_START. The paused boolean value is then added to this intent as an extra named CLASSROOM_PAUSE. The intent created from the DroneActivity class to the ClassroomActivity class is then started so the application will then run the ClassroomActivity.java file. If the user selected the Basic Tour page in the bottom navigation bar, then a new intent is made that connects the DroneActivity class to the BasicActivity class. The connected boolean value is then added to this intent as an extra named BASIC_CONNECT. The started boolean value is then added to this intent as an extra named BASIC_START. The paused boolean value is then added to this intent as an extra named BASIC_PAUSE. The intent created from the DroneActivity class to the BasicActivity class is then started so the application will then run the BasicActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                Intent intent1 = new Intent( packageContext: DroneActivity.this, MainActivity.class);
                intent1.putExtra(MainActivity.MAIN_CONNECT, connected);
                intent1.putExtra(MainActivity.MAIN_START, started);
                intent1.putExtra(MainActivity.MAIN_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_dashboard: //Art Tour
                Intent intent2 = new Intent( packageContext: DroneActivity.this, ArtActivity.class);
                intent2.putExtra(ArtActivity.ART_CONNECT, connected);
                intent2.putExtra(ArtActivity.ART_START, started);
                intent2.putExtra(ArtActivity.ART_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_classroom: //Classroom Tour
                Intent intent3 = new Intent( packageContext: DroneActivity.this, ClassroomActivity.class);
                intent3.putExtra(ClassroomActivity.CLASSROOM_CONNECT, connected);
                intent3.putExtra(ClassroomActivity.CLASSROOM_START, started);
                intent3.putExtra(ClassroomActivity.CLASSROOM_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_basic: //Basic Tour
                Intent intent4 = new Intent( packageContext: DroneActivity.this, BasicActivity.class);
                intent4.putExtra(BasicActivity.BASIC_CONNECT, connected);
                intent4.putExtra(BasicActivity.BASIC_START, started);
                intent4.putExtra(BasicActivity.BASIC_PAUSE, paused);
                startActivity(intent4);
                break;
            case R.id.navigation_drone: //Drone
                break;
        }
        return false;
    }
});
```

Figure 30: Drone Activity switch pages function defined

The connect function first checks to see if the connected boolean value is false. If connected is false then the function will create string for the client id of “Zach Drone Android Device”, create the MqttAndroidClient using the MQTTHOST and client id strings, set the connected boolean value to true, create a new MqttConnectOptions object, and then connect to the client using the options object. The function then checks if the connection was successful or not. If the connection was successful then the function creates a new MqttMessage that says connect, publishes that message to the topic defined earlier, and an alert will pop up at the bottom of the screen to tell the user the connection was successful. If the connection was unsuccessful then an alert will pop up at the bottom of the screen to tell the user the connection failed. If connected is true then an alert will pop up at the bottom of the screen to tell the user they are already connected to the drone.

```
public void connect(View v) {
    if(!connected) {
        String clientId = "Zach Drone Android Device";
        client = new MqttAndroidClient(this.getApplicationContext(), MQTTHOST, clientId);

        Toast.makeText( context: DroneActivity.this,  text: "connect", Toast.LENGTH_LONG).show();

        connected = true;

        MqttConnectOptions options = new MqttConnectOptions();

        try {
            IMqttToken token = client.connect(options);
            token.setActionCallback(new IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {
                    MqttMessage message = new MqttMessage();
                    message.setPayload("connect".getBytes());
                    try {
                        client.publish(topic, message);
                    } catch (MqttException e) {
                        e.printStackTrace();
                    }
                    Toast.makeText( context: DroneActivity.this,  text: "Connected!!", Toast.LENGTH_LONG).show();

                }

                @Override
                public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
                    Toast.makeText( context: DroneActivity.this,  text: "Connect Failed!!", Toast.LENGTH_LONG).show();
                }
            });
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else
        Toast.makeText( context: DroneActivity.this,  text: "Already Connected to Drone", Toast.LENGTH_LONG).show();
}
```

Figure 31: Drone Activity connect function defined

The pause function first checks to see if the connected and started boolean values are true, and the paused boolean value is false. If connected and started are true and pause is false then the function creates a new MqttMessage that says pause, an alert pops up at the bottom of the screen that says pause to tell the user the tour is paused, and the paused boolean value is set to true. The function then publishes the message to the topic defined earlier. If connected is true and started is false then an alert pops up at the bottom of the screen to tell the user to start a tour before pausing a tour. If connected and started are false then an alert pops up at the bottom of the screen to tell the user to connect to the drone and start a tour before pausing a tour.

```
public void pause(View v) {
    if(connected && started && !paused) {
        MqttMessage message = new MqttMessage();
        message.setPayload("pause".getBytes());

        Toast.makeText( context: DroneActivity.this,  text: "pause", Toast.LENGTH_LONG).show();

        paused = true;

        try {
            client.publish(topic, message);
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else if(connected && !started)
        Toast.makeText( context: DroneActivity.this,  text: "Unable to Pause Tour! Must Start a Tour first!", Toast.LENGTH_LONG).show();
    else if(paused)
        Toast.makeText( context: DroneActivity.this,  text: "Tour Already Paused",Toast.LENGTH_LONG).show();
    else {
        Toast.makeText( context: DroneActivity.this,  text: "Unable to Pause Tour! Must Connect to Drone and Start a Tour first!", Toast.LENGTH_LONG).show();
    }
}
```

Figure 32: Drone Activity pause function defined

The resume function first checks to see if the connected and paused boolean values are true. If connected and paused are true then the function creates a new MqttMessage that says resume, an alert pops up at the bottom of the screen that says resume to tell the user the tour is

resumed, and the paused boolean value is set to false. The function then publishes the message to the topic defined earlier. If connected is true and paused is false then an alert pops up at the bottom of the screen to tell the user to pause a tour before resuming a tour. If connected and paused are false then an alert pops up at the bottom of the screen to tell the user to connect to the drone and pause a tour before resuming a tour.

```
public void resume(View v) {
    if(connected && paused) {
        MqttMessage message = new MqttMessage();
        message.setPayload("resume".getBytes());

        Toast.makeText( context: DroneActivity.this,  text: "resume", Toast.LENGTH_LONG).show();

        paused = false;

        try {
            client.publish(topic, message);
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else if(connected && !paused)
        Toast.makeText( context: DroneActivity.this,  text: "Unable to Resume Tour! Must Pause Tour first!", Toast.LENGTH_LONG).show();
    else
        Toast.makeText( context: DroneActivity.this,  text: "Unable to Resume Tour! Must Connect to Drone and Pause Tour first!", Toast.LENGTH_LONG).show();
}
```

Figure 33: Drone Activity resume function defined

The end function first checks to see if the connected and started boolean values are true. If connected and started are true then then function creates a new MqttMessage that says end, an alert pops up at the bottom of the screen that says end to tell the user the tour is ended, and the started boolean value is set to false. The function then publishes the message to the topic defined earlier. If connected is true and started is false then an alert pops up at the bottom of the screen to tell the user to start a tour before ending a tour. If connected and started are false then an alert pops up at the bottom of the screen to tell the user to connect to the drone and start a tour before ending a tour.

```
public void end(View v) {
    if(connected && started) {
        MqttMessage message = new MqttMessage();
        message.setPayload("end".getBytes());

        Toast.makeText( context: DroneActivity.this,  text: "end", Toast.LENGTH_LONG).show();

        started = false;

        try {
            client.publish(topic, message);
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else if(connected && !started)
        Toast.makeText( context: DroneActivity.this,  text: "Unable to End Tour! Must Start a Tour first!", Toast.LENGTH_LONG).show();
    else {
        Toast.makeText( context: DroneActivity.this,  text: "Unable to End Tour! Must Connect to Drone and Start a Tour first!", Toast.LENGTH_LONG).show();
    }
}
```

Figure 34: Drone Activity end function defined

The disconnect function first checks to see if the connected boolean value is true. If connected ist true then the function will create a new MqttMessage and this message is then set to disconnect, an alert will pop up at the bottom of the screen that says disconnect to tell the user that the application was disconnected from the drone, and the connected boolean value is set to false. The function then publishes the message to the topic defined earlier. The function then checks if the disconnection was successful or not. If the disconnection was successful then an alert will pop up at the bottom of the screen to tell the user the disconnection was successful. If the disconnection was unsuccessful then an alert will pop up at the bottom of the screen to tell the user the disconnection failed. If connected is false then an alert will pop up at the bottom of the screen to tell the user that they must connect to the drone before disconnecting from the drone.

```
public void disconnect(View v) {
    if(connected) {
        MqttMessage message = new MqttMessage();
        message.setPayload("disconnect".getBytes());

        Toast.makeText( context: DroneActivity.this,  text: "disconnect", Toast.LENGTH_LONG).show();

        connected = false;

        try {
            client.publish(topic, message);
            IMqttToken token = client.disconnect();
            token.setActionCallback(new IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {
                    Toast.makeText( context: DroneActivity.this,  text: "Disconnected!!", Toast.LENGTH_LONG).show();
                }

                @Override
                public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
                    Toast.makeText( context: DroneActivity.this,  text: "Disconnect Failed!!", Toast.LENGTH_LONG).show();
                }
            });
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else {
        Toast.makeText( context: DroneActivity.this,  text: "Unable to Disconnect from Drone! Must Connect to Drone first!", Toast.LENGTH_LONG).show();
    }
}
```

Figure 35: Drone Activity disconnect function defined

3.2.5. MainActivity.java

MainActivity.java contains three strings, three boolean values, and one main function, on create function. The three strings are used to pass the boolean values between the different pages within the application. The three strings are MAIN_CONNECT, MAIN_START and MAIN_PAUSE. The MAIN_CONNECT string is used to store the connected boolean value. The MAIN_START string is used to store the started boolean value. The MAIN_PAUSE string is used to store the started boolean value. The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application.

```
public class MainActivity extends AppCompatActivity {

    public static final String MAIN_CONNECT = "com.example.graysonvansickle.zachdrone.MAIN_CONNECT";
    public static final String MAIN_START = "com.example.graysonvansickle.zachdrone.MAIN_START";
    public static final String MAIN_PAUSE = "com.example.graysonvansickle.zachdrone.MAIN_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;
```

Figure 36: Main Activity strings and booleans defined

The on create function starts the main activity layout xml file and displays this layout to the application. The function then sets the main item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named MAIN_CONNECT, MAIN_START or MAIN_PAUSE. If the intent has an MAIN_CONNECT extra then the connected boolean value is set to the value of the intent's MAIN_CONNECT extra. If the intent has an MAIN_START extra then the started boolean value is set to the value of the intent's MAIN_START extra. If the intent has an MAIN_PAUSE extra then the paused boolean value is set to the value of the intent's MAIN_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function

takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 0);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(MAIN_CONNECT))
        connected = getIntent().getBooleanExtra(MAIN_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + MAIN_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(MAIN_START))
        started = getIntent().getBooleanExtra(MAIN_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + MAIN_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(MAIN_PAUSE))
        paused = getIntent().getBooleanExtra(MAIN_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + MAIN_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 37: Main Activity on create function defined

If the user selected the Art Tour page in the bottom navigation bar, then a new intent is made that connects the MainActivity class to the ArtActivity class. The connected boolean value is then added to this intent as an extra named ART_CONNECT. The started boolean value is then added to this intent as an extra named ART_START. The paused boolean value is then added to this intent as an extra named ART_PAUSE. The intent created from the MainActivity class to the ArtActivity class is then started so the application will then run the ArtActivity.java file. If the user selected the Classroom Tour page in the bottom navigation bar, then a new intent is made that connects the MainActivity class to the ClassroomActivity class. The connected boolean value is then added to this intent as an extra named CLASSROOM_CONNECT. The started boolean value is then added to this intent as an extra named CLASSROOM_START. The paused boolean value is then added to this intent as an extra named CLASSROOM_PAUSE. The intent created from the MainActivity class to the ClassroomActivity class is then started so the application will then run the ClassroomActivity.java file. If the user selected the Basic Tour page in the bottom navigation bar, then a new intent is made that connects the MainActivity class to the BasicActivity class. The connected boolean value is then added to this intent as an extra named BASIC_CONNECT. The started boolean value is then added to this intent as an extra named BASIC_START. The paused boolean value is then added to this intent as an extra named BASIC_PAUSE. The intent created from the MainActivity class to the BasicActivity class is then

started so the application will then run the BasicActivity.java file. If the user selected the Drone page in the bottom navigation bar, then a new intent is made that connects the MainActivity class to the DroneActivity class. The connected boolean value is then added to this intent as an extra named DRONE_CONNECT. The started boolean value is then added to this intent as an extra named DRONE_START. The paused boolean value is then added to this intent as an extra named DRONE_PAUSE. The intent created from the MainActivity class to the DroneActivity class is then started so the application will then run the DroneActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                break;
            case R.id.navigation_dashboard: //Art Tour
                Intent intent1 = new Intent( packageContext: MainActivity.this, ArtActivity.class);
                intent1.putExtra(ArtActivity.ART_CONNECT, connected);
                intent1.putExtra(ArtActivity.ART_START, started);
                intent1.putExtra(ArtActivity.ART_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_classroom: //Classroom Tour
                Intent intent2 = new Intent( packageContext: MainActivity.this, ClassroomActivity.class);
                intent2.putExtra(ClassroomActivity.CLASSROOM_CONNECT, connected);
                intent2.putExtra(ClassroomActivity.CLASSROOM_START, started);
                intent2.putExtra(ClassroomActivity.CLASSROOM_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_basic: //Basic Tour
                Intent intent3 = new Intent( packageContext: MainActivity.this, BasicActivity.class);
                intent3.putExtra(BasicActivity.BASIC_CONNECT, connected);
                intent3.putExtra(BasicActivity.BASIC_START, started);
                intent3.putExtra(BasicActivity.BASIC_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_drone: //Drone
                Intent intent4 = new Intent( packageContext: MainActivity.this, DroneActivity.class);
                intent4.putExtra(DroneActivity.DRONE_CONNECT, connected);
                intent4.putExtra(DroneActivity.DRONE_START, started);
                intent4.putExtra(DroneActivity.DRONE_PAUSE, paused);
                startActivity(intent4);
                break;
        }
        return false;
    }
});
```

Figure 38: Main Activity switch pages function defined

3.3. Python Code

The message decoder for the mobile application subsystem is written in python and takes in messages from the Mosquitto server. These messages are then decoded in the python program and outputs an appropriate message that will be later sent to the navigation subsystem.

3.3.1. Client.py

Client.py contains two functions. These two functions are the message decoder and the connection status. The message decoder takes the message received from the client and then outputs the correct action based on the message received. The connection status function subscribes the client to the correct topic so that the client can see the messages that are published to this topic. The client is created using Eclipse Paho given the client name, which in this case is just Zach Drone Raspberry PI. The client is connected to the defined server address, which is the IP address of the Raspberry Pi on the WiFi. When the client is connected to the server, the program runs the connection status function. When the client receives a message from the server, the program runs the message decoder function. The client runs forever, unless the user connects to the Raspberry Pi and terminates the program.

```python
#Execute when a connection has been established to the MQTT server
def connectionStatus(client, userdata, flags, rc):
	#Subscribe client to a topic
	mqttClient.subscribe("zach/drone/rpi/tour")

#Execute when a message has been received from the MQTT server
def messageDecoder(client, userdata, msg):
	#Decode message received from topic
	message = msg.payload.decode(encoding='UTF-8')

	#Print out correct message
	if message == "art":
		print("\n\nArt Tour Selected!")
	elif message == "classroom":
		print("\n\nClassroom Tour Selected!")
	elif message == "basic":
		print("\n\nBasic Tour Selected!")
	elif message == "pause":
		print("\nTour Paused!")
	elif message == "resume":
		print("\nTour Resumed!")
	elif message == "end":
		print("\nTour Ended!")
	elif message == "connect":
		print("\nConnected to Drone!")
	elif message == "disconnect":
		print("\nDisconnected from Drone!")
	else:
		#print("Unknown message!")
		print(message)
```

```python
#Set client name
clientName = "Zach Drone Raspberry PI"

#Set MQTT server address
	#Home wifi
#serverAddress = "192.168.1.138"
	#Hotspot
#serverAddress = "172.20.10.10"
	#Zach Guest
#serverAddress = "10.236.61.140"
	#TAMU
serverAddress = "10.236.3.30"

#Instantiate Eclipse Paho as mqttClient
mqttClient = mqtt.Client(clientName)

#Set calling functions to mqttClient
mqttClient.on_connect = connectionStatus
mqttClient.on_message = messageDecoder

#Connect client to server
mqttClient.connect(serverAddress)

#Monitor client activity forever
mqttClient.loop_forever()
```

Figure 39: Defined functions (left) and variables (right) for the client.py code

3.4. Mobile Application Layout

The mobile application subsystem is dependent on the layout of the application to make the app easy for the user to interact with. The application takes input from the user and then outputs the correct action to the drone's microcontroller. The application is written for both Android and iOS platforms.

3.4.1. Android Application Layout

The layout of the Android application is written using the Android Studio application on a Macbook Pro. The application is separated into 5 different pages, which in Android Studio are written as xml files. On the main page there is just the name of the project and the team member's names. On the art tour, classroom tour and basic tour pages the list of all the destinations for that particular tour are listed in order and there is a button that will start that tour. On the drone page there are buttons to connect to and disconnect from the drone, along with buttons to pause, resume and end the current tour. All the buttons on the pages follow the app logic in figure 2 and will send the correct information to the message decoder on the drone's Raspberry Pi 3 B.

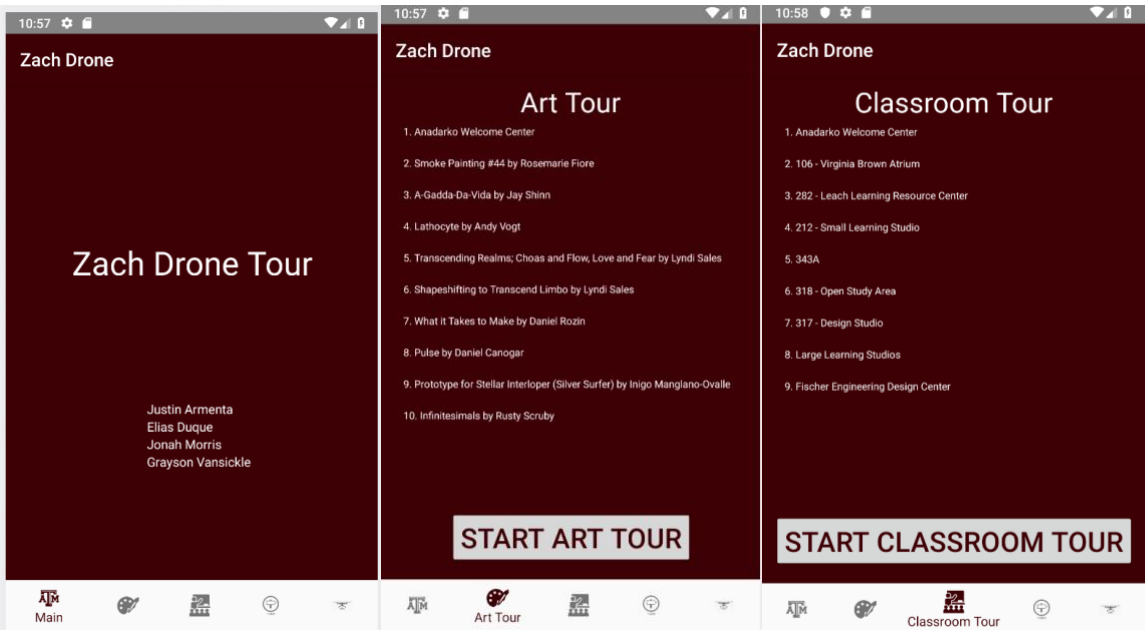
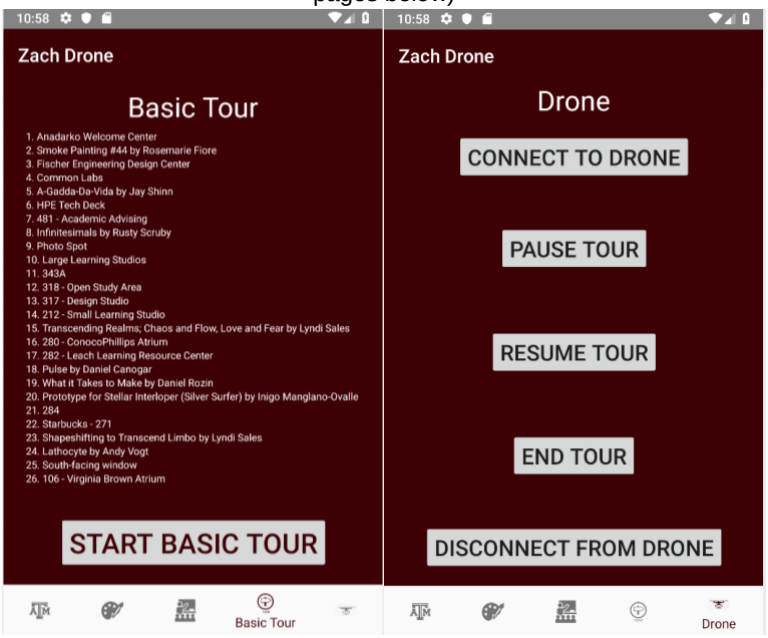


Figure 40: Android Application Layout (Main, Art Tour and Classroom Tour pages above; Basic Tour and Drone pages below)



3.4.2. iOS Application Layout

The layout of the iOS application is written using the Xcode application on a Macbook Pro. The application is separated into 5 different pages, which in Xcode are written one Main.storyboard file and given separate view controllers. On the main page there is just the name of the project and the team member's names. On the art tour, classroom tour and basic tour pages the list of all the destinations for that particular tour are listed in order and there is a button that will start that tour. On the drone page there are buttons to connect to and disconnect from the drone, along with buttons to pause, resume and end the current tour. All the buttons on

the pages follow the app logic in figure 2 and will send the correct information to the message decoder on the drone's Raspberry Pi 3 B.

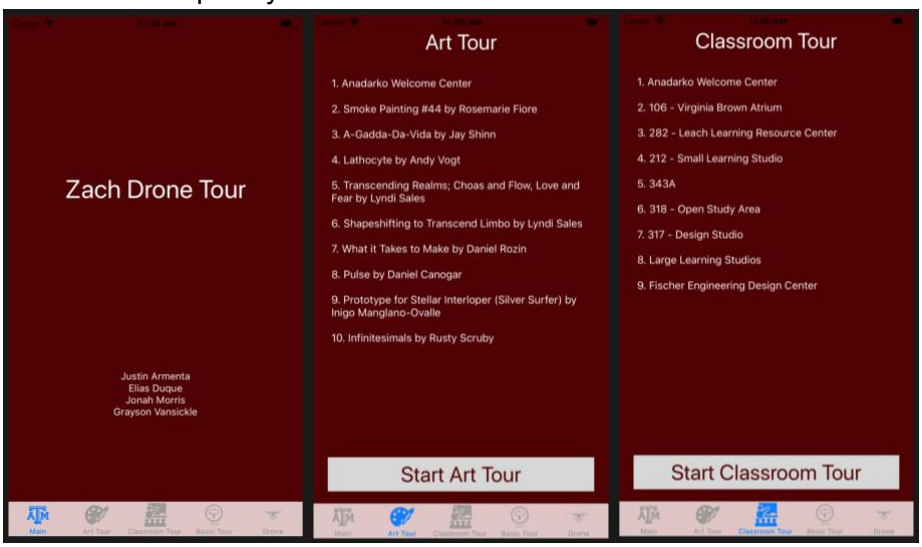
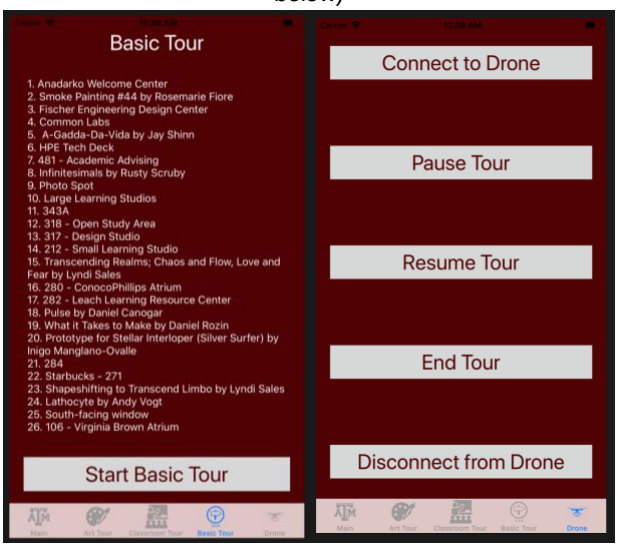


Figure 41: iOS Application Layout (Main, Art Tour and Classroom Tour pages above; Basic Tour and Drone pages below)



4. Operation

In order to show the mobile application subsystem working without connecting it to the other subsystems, statements were printed to the console to show the buttons working as expected. The statements were added to both the iOS and Android applications. A python code file was written to show that the buttons sent the right messages to the drone's Raspberry Pi 3 B.

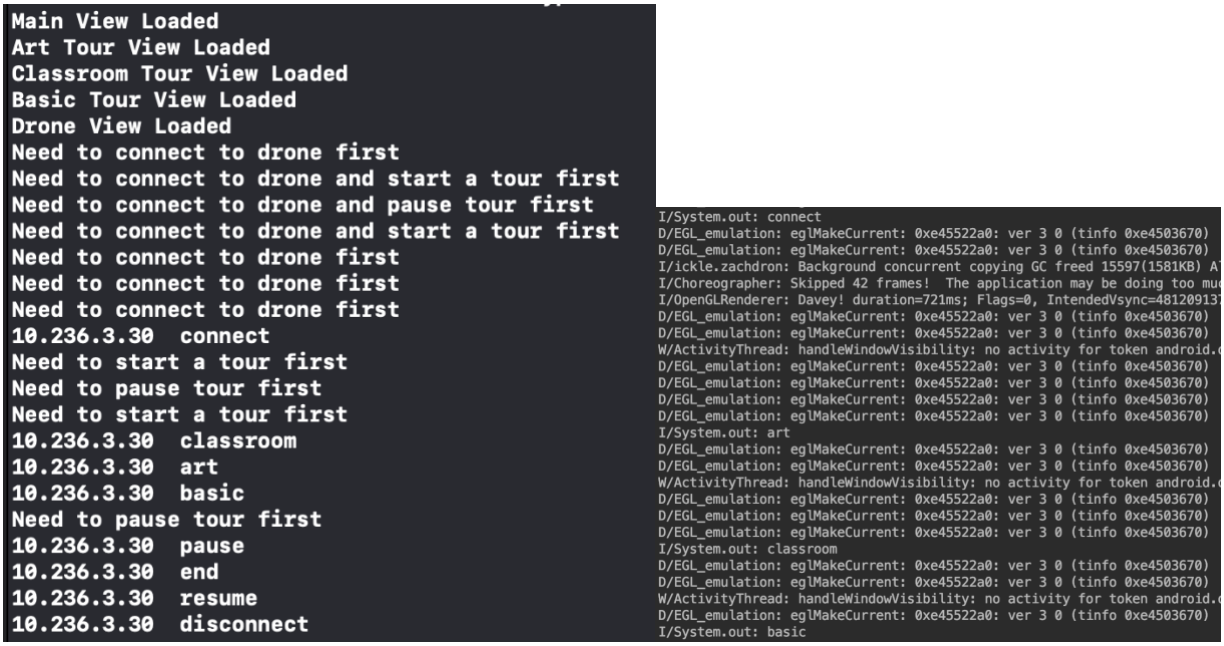


Figure 42: Terminal output for iOS (left) and Android (right) applications

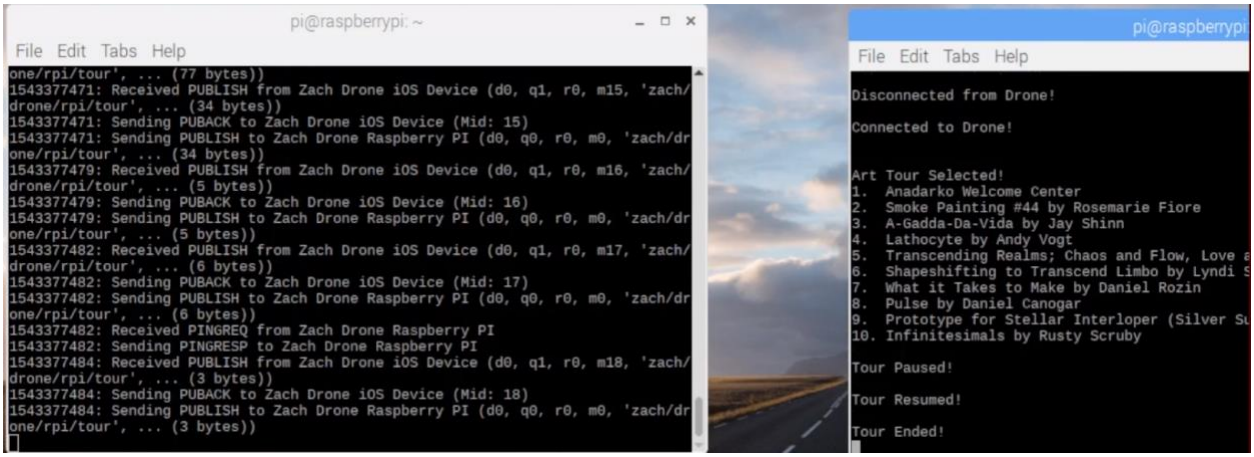


Figure 43: Terminal output for Raspberry Pi 3 B on drone

5. Data Collection

Since this application wasn't connected to any of the other subsystems, there wasn't any actual data since all the application does is send a string to the drone. The string sent to the drone will eventually be passed into the navigation subsystem's function so that the correct tour is started on the drone. However, when sending the destinations to the drone multiple times there was sometimes missing destinations on the drone.

6. Accomplishments

Throughout the semester, the mobile application subsystem reached many accomplishments. The mobile apps were able to be loaded onto test iOS and Android devices. Using the applications on the devices, the apps were able to connect to the drone's onboard

Raspberry Pi 3 B. The applications were also able to send the three different tours to the Pi and send commands to pause, resume and end a tour. The user was able to communicate with the drone and send it messages that will be used to control the drone.

7. Future Work

Although the mobile application was able to send messages to the drone's Raspberry Pi 3 B, there are some changes that needed to be made to make the mobile application easier to incorporate with the other subsystems. The application will need to made faster to connect to the drone and send messages to the drone. The message decoder function on the drone's Raspberry Pi needs to be changed to better decode the incoming messages on the drone. The main focus of next semester in ECEN 404 is to combine the mobile application with the other three subsystems of the Zach Drone project, the subsystem that will be most closely related to the mobile application is the navigation subsystem.

8. Conclusion

The iOS and Android applications performed very well and sent the messages to the drone quicker than I expected. The applications performed all of the functions it was supposed to and sent the correct messages to the drone. When the application is combined with the drone, there might need to be slight changes to the way the messages are sent or the message decoder to allow the drone to work efficiently. Looking forward to the next semester there is still work that needs to be done to make this a complete working project, but the mobile application subsystem was a great success for the amount of time in this semester. The task at hand for next semester isn't an easy one, but completing the mobile application takes one thing off the list that needs to be finished next semester.

Zach Drone

Justin Armenta

Navigation Subsystem Report

REVISION – 1.2
5 December 2018

Table of Contents

List of Figures

List of Tables

No tables in this document.

1. Introduction

The purpose of the navigation system is to determine the drone's location within Zachry and navigate to the next destination based on this information. This subsystem will consist of six ultrasonic sensors that will be placed on the top, bottom, front, back, left, and right of the drone,

as well as a Raspberry Pi 3 B to run the Python code, which takes the distance information from the sensors to navigate. Based on the current and previous values of the ultrasonic sensors the Raspberry Pi will determine if a new checkpoint has been reached using an algorithm that compares the sensor values to the map model or if it was a false reading and should be ignored. Based on the determined location a command for the forward speed, sideways speed, and up and down speed will be given to the flight controller. A model of Zachry has been written into the Raspberry Pi that gives the dimensions of all hallways that the drone will fly near. This will help the drone determine if it is at a new checkpoint or if a sensor value has made an error and should be ignored.

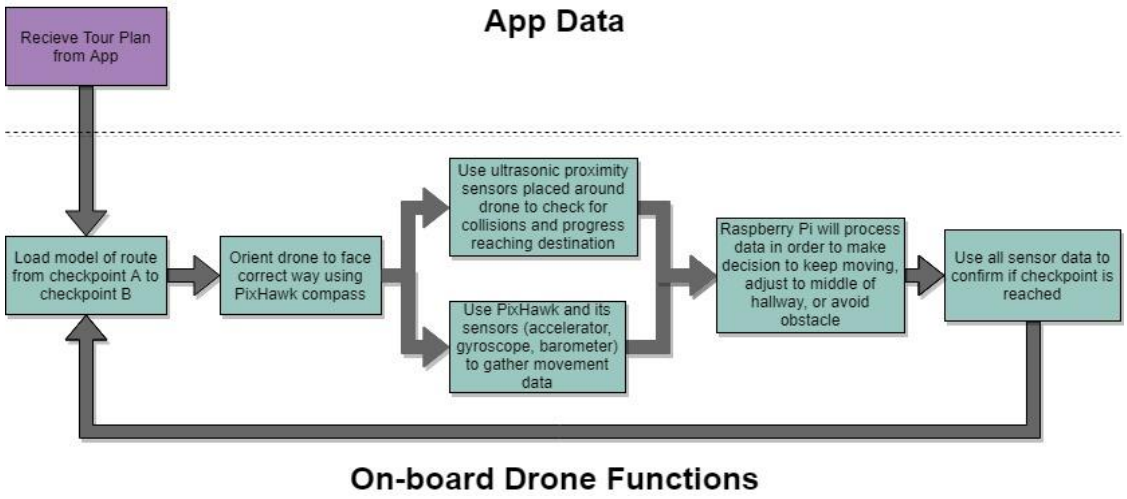


Figure 1: Navigation Subsystem

2. Theory

2.1. Tracking Location of Drone

All hallways within Zachry have been inserted into the Raspberry Pi with different areas called “checkpoints” that make up the hallway. A “checkpoint” is an area with unique dimensions within the hallway, such as the area before a divot for a doorway appears on the right side of the wall. The key thing to note about every checkpoint is that there is a significant change to a particular side of a wall. Checkpoints were created as a point where ultrasonic sensors can see a large enough change that a flag can be thrown which starts the process of determining if this change in value was a sensor misreading or a new checkpoint being reached. Dimensions on each checkpoint were manually entered in the Raspberry Pi such as: the change in the left and right wall compared to the previous checkpoint, the distance from the beginning of the destination to the beginning of the checkpoint, the ceiling height, and wall width at the beginning of the checkpoint. All of this data is used in a checkpoint algorithm to determine which checkpoint the drone is located in. The Raspberry Pi will check to see if there is greater than 0.6 m change in the ultrasonic sensors compared to their last readings. If a large difference is detected the rest of the algorithm is ran to determine the checkpoint the drone is in. Since the ultrasonic sensors can only be relied on up to about 4 m away and some hallways are much wider than 12 m, every destination has a “side reliance” which basically means the drone will

stay near one side of the hallway and rely on the ultrasonic sensor on the reliant side more than the other. The algorithm is shown below in figure 1.

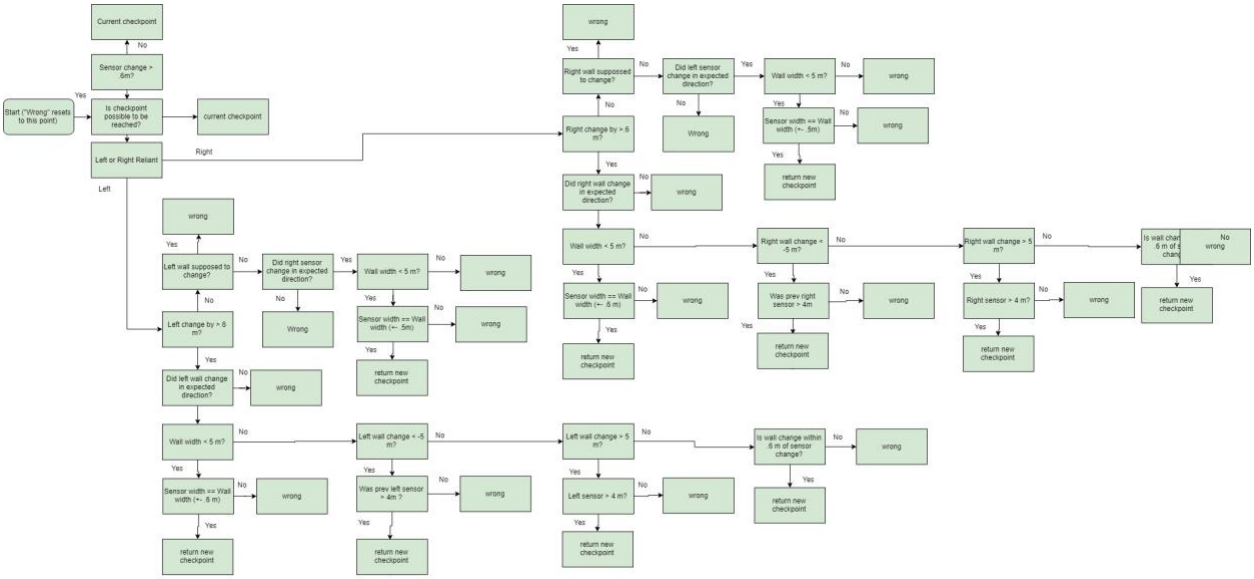


Figure 2: Checkpoint Algorithm

Once the last checkpoint has been reached within a destination, the “attraction” has been reached, which means the drone will give a speech about the attraction. At this point the drone will orient itself and load a new destination with all new checkpoints. When the last destination is reached this will signal the end of the tour.

2.2. Outputting Speed Commands to PixHawk

The drone speed is split into three different numbers based on which axes needs to move. The “z speed” is meant to be the forward/backward speed, the “x speed” is meant to be left/right speed, and the “y speed” is meant to be up/down speed. The z speed is defaulted to 2 (m/s) and will only change when an obstacle is detected in front of the drone, the destination has been reached and the drone needs to be stopped, or the tourists are too far/close and the drone needs to change speed. The y speed will be at 0 most of the time and it will be up to the flight controller to maintain altitude. The only time the y speed won’t be 0 is when the drone is actively changing floors and the drone needs to elevate to reach the next floor. The x speed will be determined by how close the drone thinks the left and right walls are to the drone. By default the x speed will be 0, but if the ultrasonic sensor on the left or right detects a wall within 1.2 m of the drone it will move away from the wall. Based on the side reliance it can also move towards the wall if there is supposed to be a wall closer than the sensor reads.

3. Design

3.1. Python Codes

The navigation subsystem is entirely in python code and takes ultrasonic inputs and outputs speeds to the flight controller. The two main programs that will be used are the Map_Model.py and Navigation_Test.py.

3.1.1. Map_Model.py

Map_Model.py contains only three actual functions. They all are very trivial functions that simply point to the next checkpoint or destination. The rest of the code is stating the dimensions of the building so Navigation_Test.py can call on that information later when the navigation function needs it to determine the current checkpoint. Dictionaries were used to create the attributes of the tour. The tour choice will be populated first between the art tour, class tour, or the basic tour. Then within a specific tour, all destinations within that tour's dictionary will be declared. Then all the checkpoints within the destination will be declared. Shown below in figure 3 is the different dictionaries being declared.

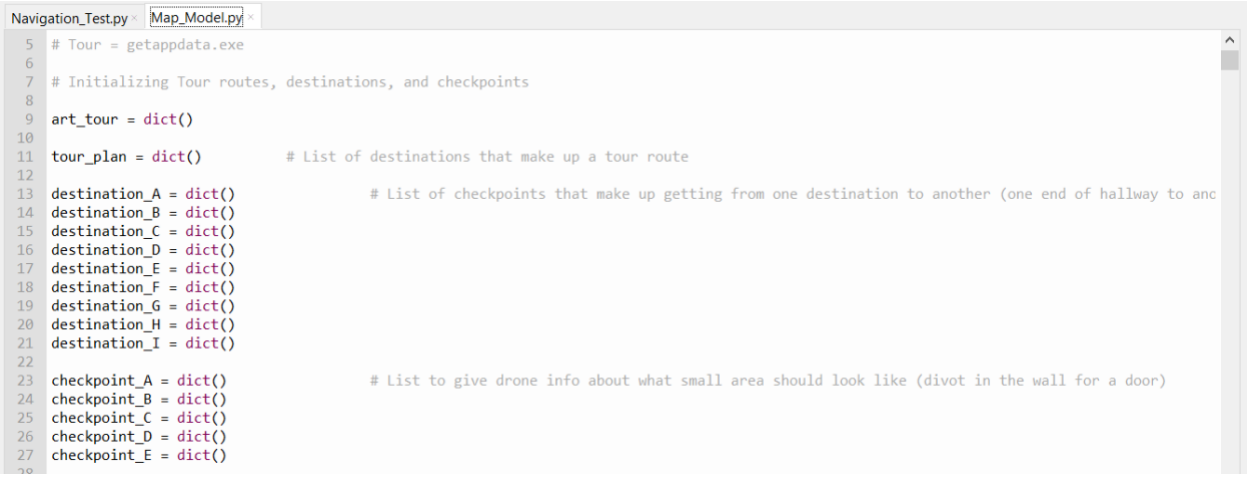


Figure 3: Declaring Dictionaries

In figure 4 the different attributes that go inside checkpoint and destination are shown. As you can see checkpoints are put inside unique destinations. Together any attribute of a checkpoint can be found as long as you have the tour, the destination, and the checkpoint.

Navigation_Test.py × Map_Model.py

```
checkpoint_D['Ceiling'] = 3.048
checkpoint_D['Desired Altitude'] = 2.438
checkpoint_D['Left wall change'] = .69215
checkpoint_D['Right wall change'] = 1.003
checkpoint_D['Wall width'] = 7.493
checkpoint_D['Z Position'] = 8.128

checkpoint_E['Floor'] = 0.0
checkpoint_E['Ceiling'] = 3.048
checkpoint_E['Desired Altitude'] = 2.438
checkpoint_E['Left wall change'] = 50
checkpoint_E['Right wall change'] = 7
checkpoint_E['Wall width'] = 7.493
checkpoint_E['Z Position'] = 11.938


destination_A['Compass Orientation'] = 45.0         # Compass direction drone should face before beginning flight path (in degrees)
destination_A['Orientation'] = 'Left'               # Will be used to tell drone which side sensor will have priority over the other when m
destination_A['Checkpoint A'] = checkpoint_A
destination_A['Checkpoint B'] = checkpoint_B
destination_A['Checkpoint C'] = checkpoint_C
destination_A['Checkpoint D'] = checkpoint_D
destination_A['Checkpoint E'] = checkpoint_E
destination_A['Checkpoint F'] = 'End'
```

Figure 4: Attributes of destinations and checkpoints

3.1.2. Navigation_Test.py

Navigation_Test.py is where the important functions associated to finding the location, determining the speeds, and changing from one destination to the next is found. Once the drone has taken off, the sensors gather data every .3 seconds. At that time the checkpoint algorithm is used, using the function named “checkpoint_checker”, to determine which checkpoint the drone is at. If a new checkpoint is found then the “current checkpoint” variable will become this new checkpoint and the drone will change it’s “guessed z position” variable to the value stated in the new checkpoint’s z position. This “guessed z position” also changes based on the z speed to keep track of the likely z position of the drone, unless a new checkpoint is found then the z position will be changed to the value stored inside the new checkpoint dictionary. This guessed z position variable is important when determining whether it is actually possible for the drone to be at the next checkpoint. Once the “checkpoint_checker” function is ran the “x_speed_chooser”, “z_speed_chooser”, and “y_speed_chooser” functions are ran. These functions take the current checkpoint and the ultrasonic sensor data in order to determine what speed each function should output to the flight controller.

4. Operation

In order to show the navigation subsystem working without flying the actual drone a simulation program had to be created to imitate the walls of the building, calculate what the sensors would output, and calculate where the drone would be based on speed outputs. A GUI was also created to show where the walls are and the drone would be. When the simulation started it was assumed that the drone would be placed in an exact location. The GUI began and the simulation looked like figure 5, where the drone is the red box and the black lines are the walls.

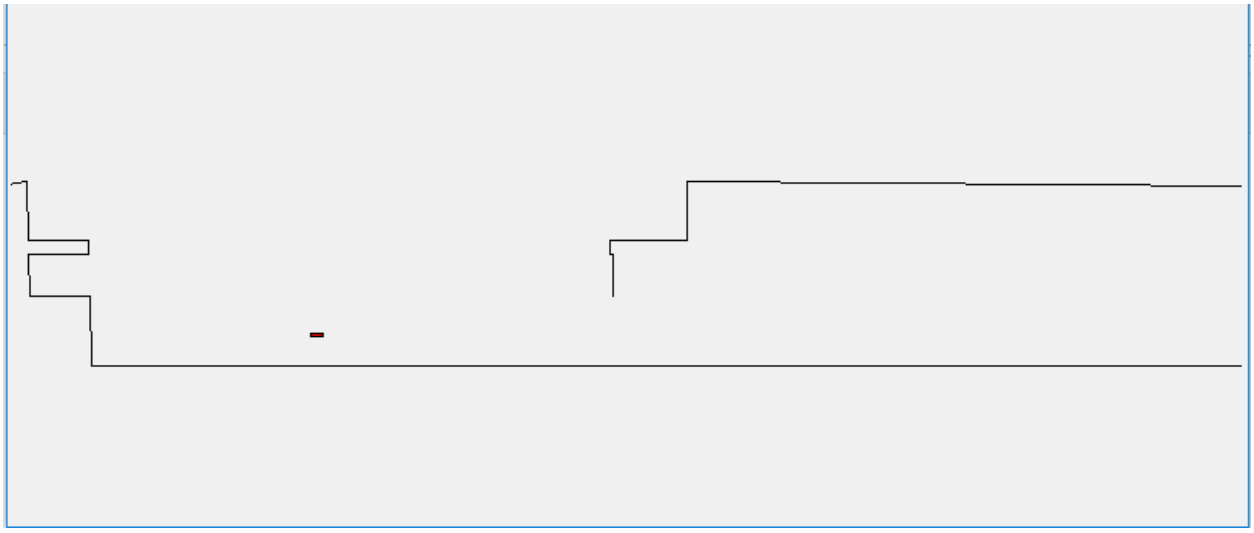


Figure 5: Beginning of navigation simulation

As the drone reached the first divot outward change of the left wall it changed in the correct way which was change to checkpoint B and the left sensor increased. As the drone hit the beginning of the checkpoints it changed correctly for wall checkpoints.

```
Time:  1.5
Left sensor:  1.5444340365731
Right sensor:  4.2798
Forward sensor:  4.2729
Z position:  4.39883
Y position:  2.21673
X position:  0.5017999999999999
True X Speed:  0.0282
Checkpoint:  Checkpoint A

Time:  1.8
Left sensor:  2.2875636522673997
Right sensor:  4.2504
Forward sensor:  4.2692000000000005
Z position:  5.00297
Y position:  2.2134
X position:  0.5102599999999999
True X Speed:  -0.0551
Checkpoint:  Checkpoint B
```

Figure 6: Data from transition from checkpoint A to checkpoint B

When the last checkpoint is reached, the new destination is loaded along with all of its new checkpoints. The drone's coordinates inside the simulation were then converted as the drone changed front facing direction based on compass orientation. This can be seen in the transition between figure 7 and figure 8.

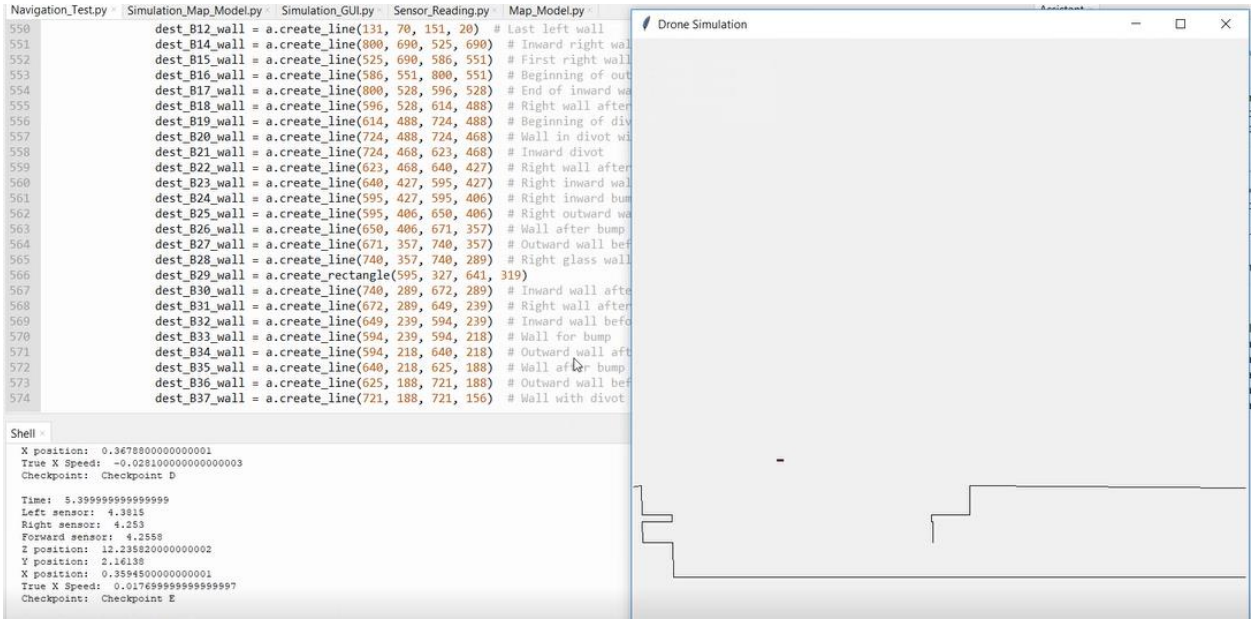


Figure 7: Results at the end of destination A of the art tour

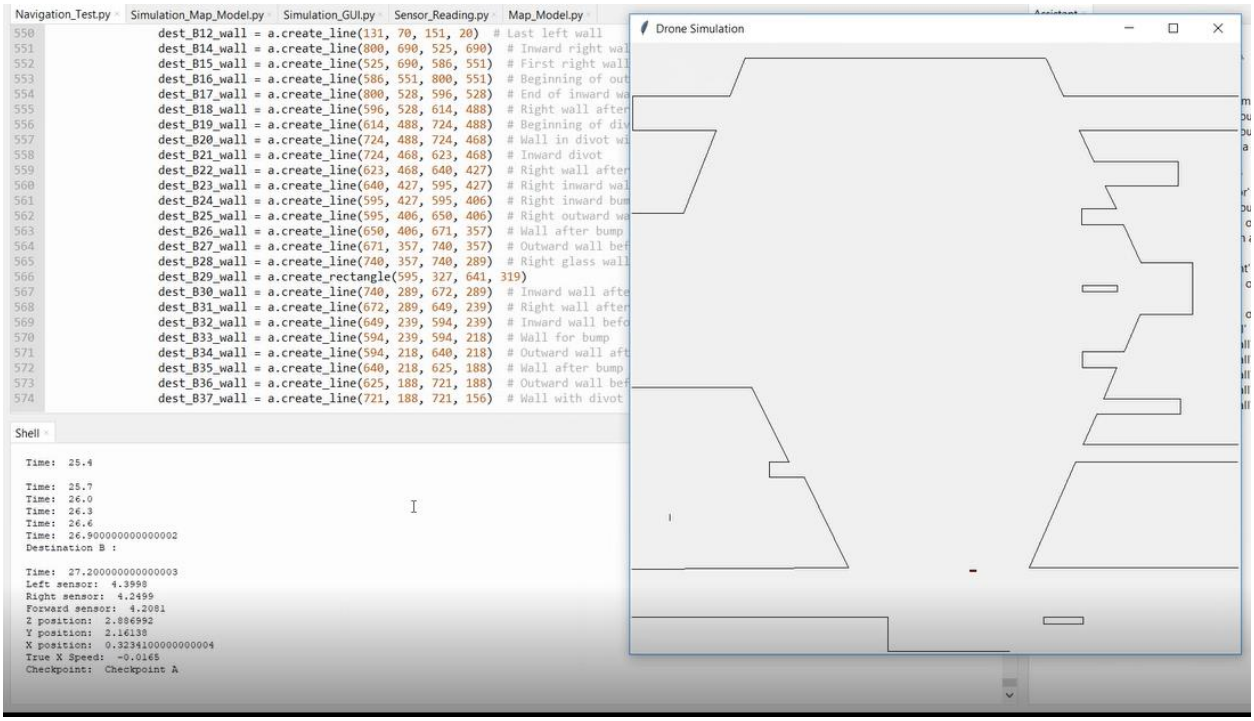


Figure 8: Results at the beginning of destination B of the art tour

The drone continues down the hallway in the same fashion as destination A, then it changes to destination C and returns back down the hallway. The rest of the simulation runs in this fashion going from destination to destination until the tour is over.

5. Data Collection

Since this was a simulation of how a code full of algorithms that only outputted commands functioned when data was given to it, there was no real data to be found. However, when the simulation was ran 10 times, there were no checkpoints missed and the drone chose the correct z, x, and y speed for the entirety of the simulation. This is significant because the speed and sensor values were given error elements to help give the code conditions that were closer to what would actually happen. The speed value was given an random error of up to ±10% and the sensor readings were given a multiple realistic conditions. First, if the sensor reading was less than 40 cm the sensor would output a value near 40 cm as it would in a real situation, if the sensor was greater than 4.2m it would give a random value from 4.2m to 4.56m, and if the sensors were any value in between there would be a random error of up to ± 6 cm. So although there was error that attempted to emulate the actual restrictions of the drone and sensors, the code still performed its function as intended in all the simulations.

6. Accomplishments

Over the course of the semester the navigations subsystem had many accomplishments. The code was able to take only the tour plan from the app and the outputs from the sensors and give correct speed commands and correct checkpoints. A simulation was also created so that future test can be ran if the code needs to be changed to account for unforeseen issues that may come up in the future. Dimensions of many parts of Zachry have been found in order to easier model different hallways in the future without having the remeasure. A GUI was also created that successfully simulated the flight path of the drone while also outputting which checkpoint the drone thought it was at. This accomplishment helped debug issues and helped visualize where the drone was in the building at any given time. A picture of the end of the tour with the output and GUI simulation is shown below in figure 9.

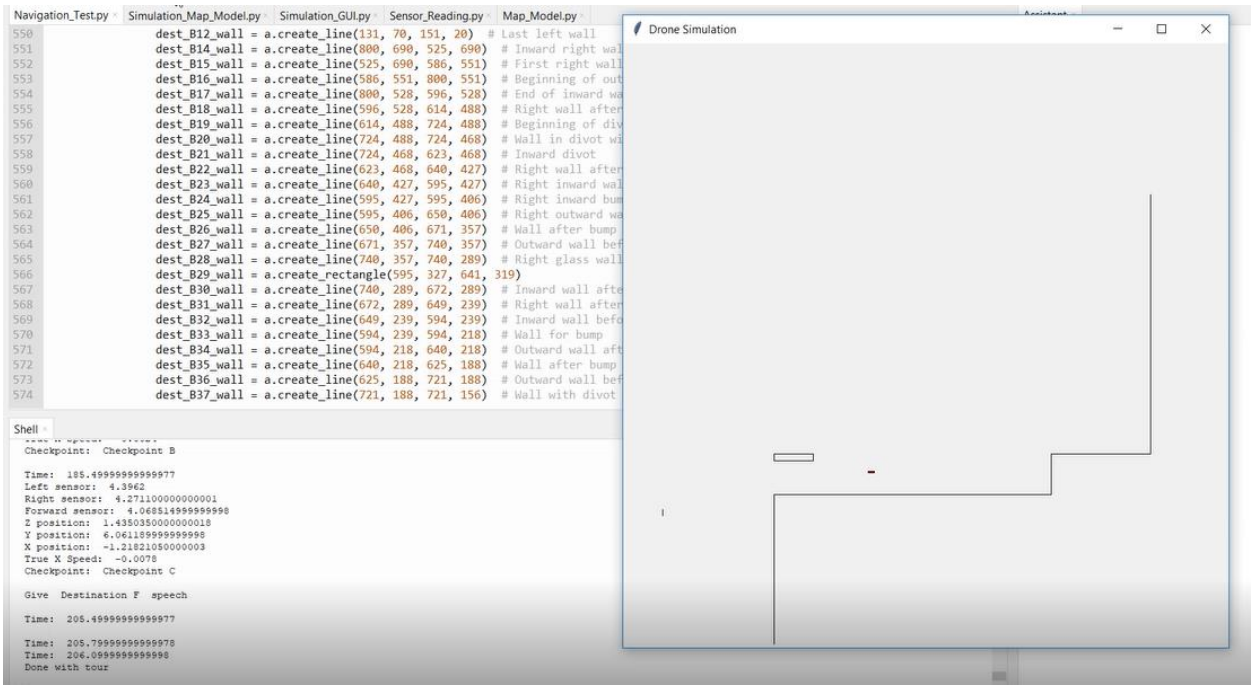


Figure 9: End of tour simulation result

7. Future Work

Although the simulation worked there may need to be changes to some error tolerances in the future when the actual drone is used instead of a simulation with fixed error bounds. The entire tour was never fully simulated because creating a model for all of Zachry would be a very long process. This is because there is no way to find the blueprint of Zachry since that information won't be released to the public until next year. So the only way to find the dimensions of the hallways was to hand measure it myself with a tape measure. This took me hours to do for the first floor and the very beginning of the second floor so doing all of the second and third floor became an impossible task because between measuring the walls, recording that data, and then inserting this data into the model that would take up dozens of hours. The code is also still written to coordinate with the simulation and will need to be tweaked when the actual flight controller and ultrasonics are interfacing with it.

8. Conclusion

The code for navigating the drone performed very well and exceeded my own expectations. It did everything it was supposed to do in terms of figuring out where it is within the building and outputting the correct speeds based on that information. However when this code is actually used with the drone it may need slight adjustments to run as well since even though error was added to the drone speed to account for realistic cases, we don't know how accurately the real drone will handle these speed commands. Looking forward there is still work

to be done making this a working project, but I believe that this subsystem was a huge success this semester. Looking back at choosing this project I believe that it is a lot for four undergraduate students to handle. This was an important lesson in learning just how much work, time, and brain power any real engineering project requires to become a success. Flying a drone autonomously with no GPS is a massive task in itself, and asking it to also lead a crowd, and get to each destination in a timely manner is massive undertaking.

Zach Drone

Jonah Morris

Drone Subsystem

REVISION – 1.2
5 December 2018

Table of Contents

List of Figures

List of Tables

1. Introduction

The Zach drone tour will need a functioning drone to implement a tour. The drone subsystem consists of all the of the hardware aspects of this project. This subsystem includes

the drone frame, battery, power converter, motors, electric speed converters, propellers, Pixhawk, Raspberry Pi and ultrasonic sensors.

2. Theory

The scope of this subsystem is to provide all of the hardware aspects of the project needed by the other subsystems. There are essentially two subsystems of this subsystem, the ultrasonic sensors and the drone.

2.1. Drone

Our drone uses a battery that is connected to a power distributor that supplies the motor ESCs, Pixhawk and Raspberry Pi with power. When powered on the Raspberry Pi controls the Pixhawk using the MAVlink protocol over serial connection giving it the direction and speed to move. The Pixhawk will then tell the electronic speed converters (ESC) how much trust to give the motors.

2.2. Ultrasonic Sensors

There are six ultrasonic sensors connected to the Raspberry Pi via GPIO pins. These sensors send out ultrasonic sound and measure the time it takes to receive the sound back. It then converts that time into a high voltage pulse that is high for 1 µs per 1 mm to the nearest detectable object is. This data is used by the navigation subsystem to determine where the drone is inside of the building based on a model.

3. Design

3.1. Power Connections

A single four cell Lipo battery will power the entire drone and all microcontrollers. The drone uses a TATTU 8000mAh lipo battery plugged into a PDB-XT60 power regulator through the Pixhawk's own power converter connection. The motor electronic speed converters are powered from the ESC tabs and the Raspberry Pi is powered from the 5V tab on the PDB-XT60. The six ultrasonic sensors are powered from the Raspberry Pi's 5V pin. Voltage and current flow is shown in the following figure.

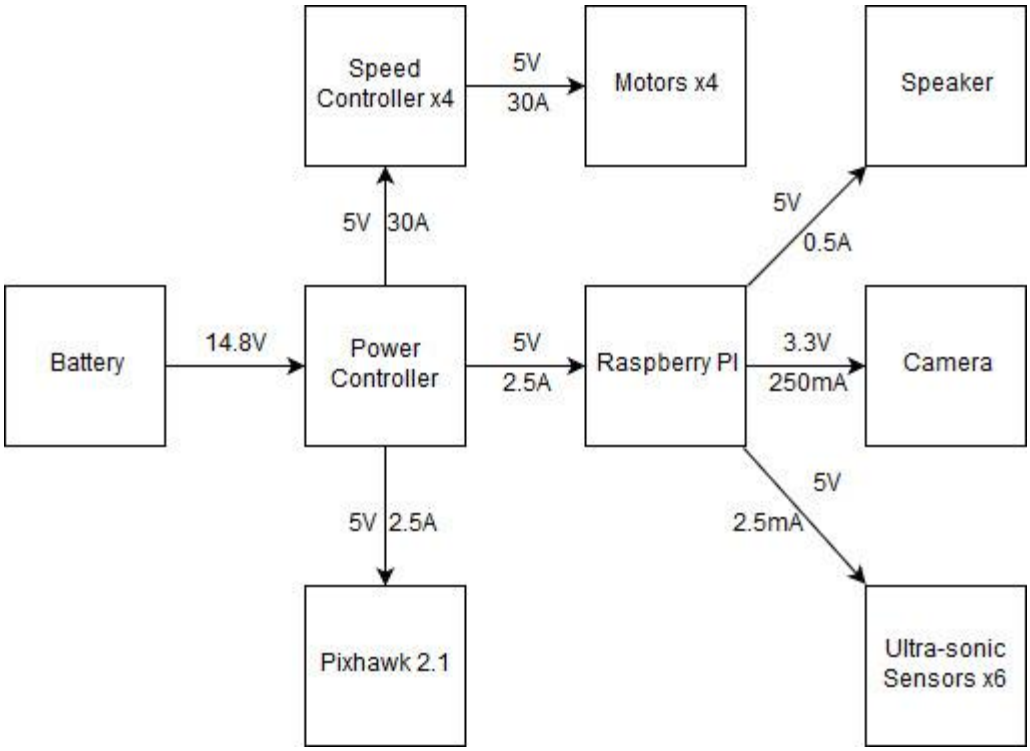


Fig. 1: Power connections of the drone subsystem with maximum voltage and current ratings

3.2. Ultrasonic Sensors

Ultrasonic sensors are used to detect the drones distance from nearby objects. The drone uses six Maxbotix HRLV-EZ0 ultrasonic rangefinders connected to the Raspberry Pi to determine the distance to the nearest object. These sensors are rated for distances of 30 cm to 5m. The Raspberry Pi uses these sensors in a trigger then receive relationship allowing for readings to only be requested when they are needed by the navigation subsystem. Ultrasonics sensor were chosen over infrared sensors because many walls inside of Zachry are glass which does not reflect infrared well. The following figure displays the connections made between the Raspberry Pi and ultrasonic sensors.

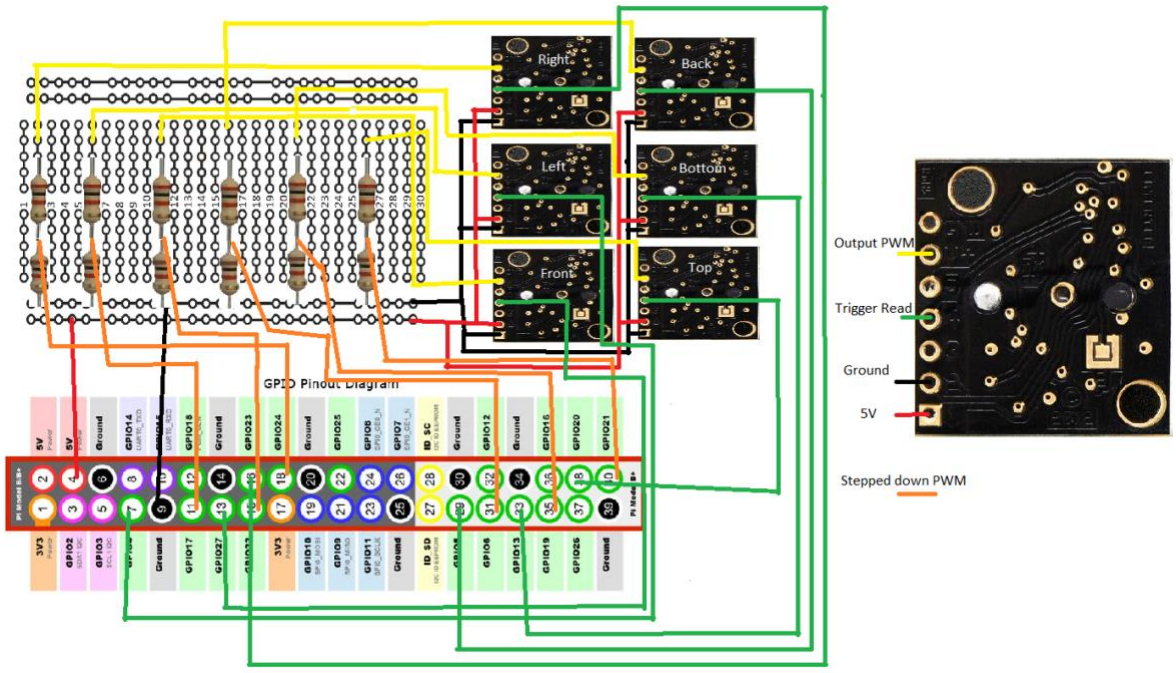


Fig. 2: Connections between raspberry pi GPIO pins and ultrasonic sensors.

The Raspberry Pi gathers ranging data by sending a 25us pulse to the trigger pin and waits for a PWM signal from the output pin. This signal is stepped down using a resistor divider to match the 3.3V GPIO input voltage. The signal output is 1us long per 1 mm distance sensed.

3.3. Flight Time

We would like our flight time of the drone to be as long as possible to allow for longer tours. To calculate our estimated time of flight we used the following formula:

$$Flight\ time\ (t) = \frac{Battery\ Capacity\ *\ Discharge}{Weight\ *\ Light\ Power} * Voltage$$

The combined weight of all components is 1.5kg. The calculated estimated time of flight is 22.3 minutes. We limit the time of tours to be 15 minutes so that the drone has time to fly back to the starting position before it's battery life fails.

3.4. Raspberry Pi to Pixhawk Connection

The Raspberry Pi tells the Pixhawk which direction to fly and how fast whereas the Pixhawk sends information to the Raspberry Pi about the orientation of the drone and its altitude. This is done by using a serial connection as show in the following figure.

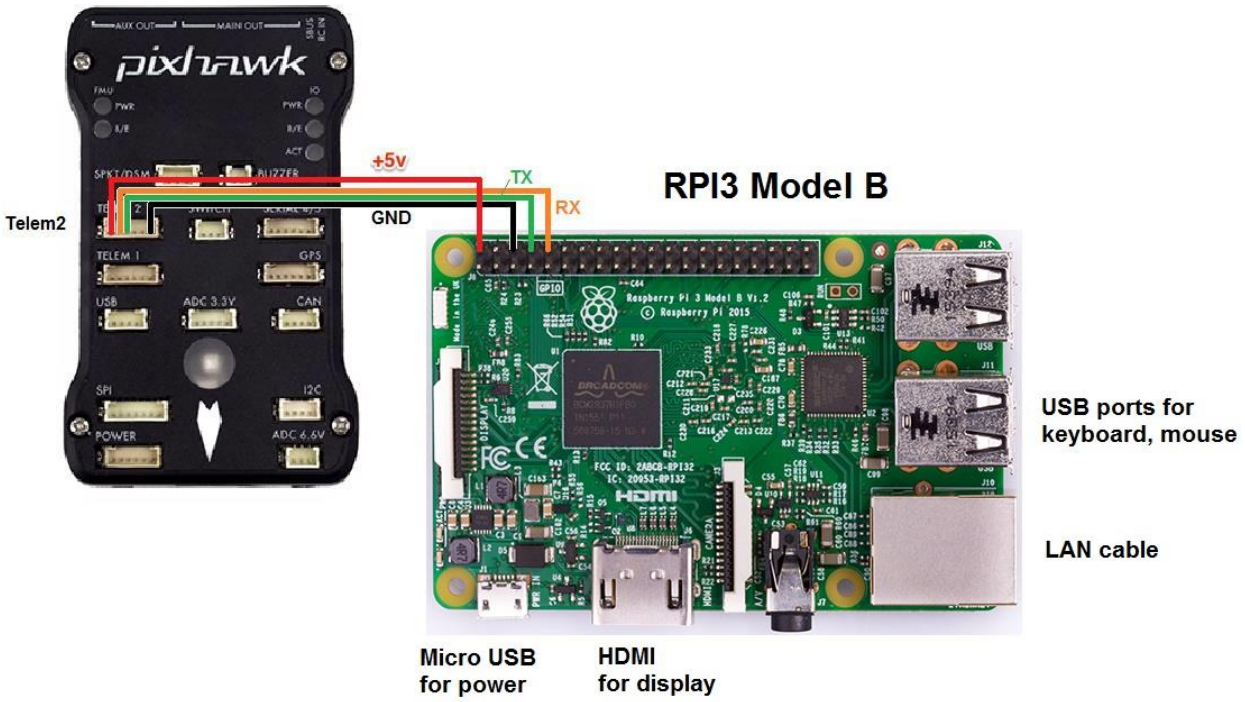


Fig. 3: Pixhawk to Raspberry Pi serial connection

This connection in conjunction with Dronekit and GUIDED_NOGPS flight mode will be used give directional commands to the drone for takeoff and flight.

4. Operation

4.1. Ultrasonic Sensors

The ultrasonic sensors were validated by first connecting one sensor to the Raspberry Pi and placing an object at incremental distances from the sensor and reading the outputs. The sensor proved to be within 5 centimeters of the expected value for up to 4.3 meters. Next, the six sensors were attached to the drone and connected to the Raspberry Pi. They were tested by placing the drone in the center of a room and placing objects around the drone and testing the Raspberry Pi output to the actual distance. Further testing of the six ultrasonic sensor during flight needs to be conducted.

4.2. Power connections

The power connections were tested by attaching all components to the drone and turning them all on using only the battery. The functionality of the Raspberry Pi to Pixhawk connection and ultrasonic sensors were tested again while the battery powered the system. Further testing of the battery life under load of the motors still needs to be tested and quantified.

4.3. Raspberry Pi to Pixhawk Connection

The connection between the Raspberry Pi and Pixhawk was validated by sending flight commands to the Pixhawk through the Raspberry Pi terminal. Changing flight modes and

arming the throttle were both tested and validated. Further testing of Guided_NOGPS flight commands to control the drone still need to be simulated and tested.

5. Data Collection

5.1. Ultrasonic Sensors

To characterize typical accuracy of the ultrasonic sensors I tested one sensor by placing it at the end of an empty table and placing a small box in front of it and taking many distance reading at different distances. Below is an excel sheet showing the average distance sensed, the width of the beam at that distance and the standard deviation.

	A	B	C	D	E	F	G	H	I	J
1	distance:	26.4 cm	62.98 cm	92.44 cm	122.96 cm	153.4 cm	183.88 cm	214.36 cm	305.8 cm	427.72 cm
2	values:	30.04	65.37	94.91	125.6	155.71	185.2	215.34	305.22	424.86
3		30.23	64.68	94.39	125.38	156.66	185.97	215.63	305.1	424
4		30.02	64.99	94.7	124.28	156.24	185.68	215.39	304.17	425.15
5		29.85	64.83	93.77	125.81	155.9	185.7	215.27	305.68	424.81
6		30.21	65.64	94.1	125.48	156.12	185.92	216.01	305.27	424.03
7		30.28	65.04	94.46	125.17	156.16	186.13	215.63	305.39	425.05
8		30.23	65.3	94.84	124.98	156.26	185.99	215.7	306.2	424.93
9		29.64	65.09	95.39	125.12	156.09	185.51	215.01	306.22	424.89
10		29.83	64.28	94.87	125.74	156.16	186.3	215.53	305.65	424.55
11		30.21	65.02	94.51	124.91	155.14	185.97	215.17	305.53	424.7
12		30.35	64.95	94.37	125.53	156	185.25	216.03	305.22	425.15
13		29.73	64.8	95.08	125.43	155.69	186.23	214.82	305.68	424.7
14		29.78	65.26	94.91	125.41	156.04	185.94	215.53	305.49	424.86
15		30.35	64.44	94.75	125.43	156.02	185.54	214.91	305.13	425.08
16		29.8	64.9	94.03	125.48	155.78	186.42	215.79	304.94	424.41
17		29.95	64.75	94.7	125.31	155.54	185.51	216.03	305.94	427.87
18		29.54	65.47	94.63	125.79	155.71	185.31	215.79	305.99	424.05
19		29.87	64.68	93.98	125.36	155.93	185.94	215.77	305.94	424.31
20		30.23	64.64	94.22	124.96	155.59	185.97	215.67	305.77	424.84
21		30.72	65.14	94.37	125.48	155.04	185.39	215.58	305.53	425.1
22		30.04	65.04	94.99	125.69	155.76	186.13	215.55	304.84	424.58
23	averages:	30.04286	64.96714	94.57	125.3495	155.8829	185.8095	215.531	305.4714	424.8533
24	beamwidth:		35.56	53.34	50.8	53.34	60.96	38.1	30.48	20.32
25	STD	0.278348	0.326892	0.397851	0.348486	0.363477	0.347953	0.338511	0.47896	0.761998

Table 1: Characteristics of the Maxbotix HRLV-EZO ultrasonic sensors.

Included in the data is the measured distance using a tape measure, the values output from the ultrasonic sensor 20 times, the averages, the beamwidth and the standard deviation. This data helped characterize the sensors and gave the navigation subsystem more guidelines as to how accurate its distance reading are going to be.

After attaching and connecting all six ultrasonic sensors I ran 8 tests with objects at different distance in the four x and y directions and compared them to expected results. The following table contains the list of the tests, the results and the results were within 5 cm accuracy.

Measure Distances	Distance Results	Within 5cm?

L = 120cm R = 120cm F = 120cm B = 120cm	L = 118.4cm R = 120.5cm F = 119.4cm B = 120.1cm	Yes
L = 180cm R = 180cm F = 180cm B = 180cm	L = 178.7cm R = 181.3cm F = 180.9cm B = 179.1cm	Yes
L = 250cm R = 250cm F = 250cm B = 250cm	L = 250.1cm R = 247.6cm F = 250.2cm B = 248.5cm	Yes
L = 400cm R = 400cm F = 400cm B = 400cm	L = 404.4cm R = 402.1cm F = 401.6cm B = 402.4cm	Yes
L = 60cm R = 60cm F = 150cm B = 150cm	L = 59.3cm R = 60.3cm F = 58.8cm B = 59.1cm	Yes
L = 20cm R = 20cm F = 20cm B = 20cm	L = 30.1cm R = 30.4cm F = 30.2cm B = 30.2cm	No, Range must be greater than 30cm
L = 200cm R = 250cm F = 120cm B = 180cm	L = 199.2cm R = 251.1cm F = 117.3cm B = 178.7cm	Yes
L = 90cm R = 90cm F = 60cm B = 60cm	L = 87.4cm R = 88.6cm F = 58.7cm B = 156.4cm	Yes

Table 2: This table shows the rangefinding data for four sensors placed on the drone.

6. Accomplishments

This semester I was successfully able to create all power connection and communication connections for the drone to be able to control its flight using the ultrasonic sensors, gyroscope and compass on the Pixhawk. I tested and validated that all of the work that I accomplished on

this subsystem was functional to the specification outlined in the CONOPS, FSR and ICD. I also wrote sample flight code code but was unable to test and validate this semester.

7. Future Work

Simulation and testing of actual flight without GPS will need to be conducted before any further work on navigation can be done. I will be working over the break to simulate flight with GUIDED_NOGPS mode so that at the start of next semester we can get the drone through some test flights. Further testing of the sensors will need to be conducted during flight to ensure that the propellers do not affect the accuracy of the sensors.

8. Conclusion

Creating this subsystem put into perspective the amount of work that goes into designing, testing and documenting a full project. This being my first time working with drones I did not know how difficult it would be to simulate and fly the drone without GPS. In hindsight I should have focused more on getting a working simulation and flight test for no gps flight before working on the sensors. Through working on this subsystem I learned the values of planning ahead and being thorough with research and design so that implementation comes more smoothly. I look forward to next semester where we will integrate our subsystems.