pTera

Pod-Based Autonomous Vehicles Group

Final Report

Computer & Electrical Engineering 129

Senior Design Project

Advising Director Patrick E. Mantey

Harikrishna Kuttivelil, Sargis Yonan, Niraj Raniga, Nikola Panayotov,
Tianyi Zhu, Jessica Herrera, Justin Lee, Serena Mak

Jack Baskin School of Engineering

University of California, Santa Cruz

Table of Contents

Introduction

pTera is a flexible software and hardware framework intended to collect and visualize data from a pod of drones that work in unison to survey a user-selected area. This system can be extended to handle any number of drones and any type of sensor. This tool is expected to serve a wide variety of area-scanning applications, including fire prevention and observation, terrain analysis, and search-and-rescue.

pTera was built without any third party developed frameworks; resulting in a very efficient and cost effective alternative for retrieving sensor data from a given geographical area autonomously. The drones are designed to use a machine learning algorithm to quickly search a given map area while relaying information towards a ground base computer. The ground system consists of a central station involved in receiving and processing status packets from the drones, sending out command packets to the drones, and managing the database. It can be connected to a web interface as well. The control packets sent from the ground station to the drones are based on multiple factors, including data from status packets and input from the web interface.

pTera Data Structure and Flow

Introduction

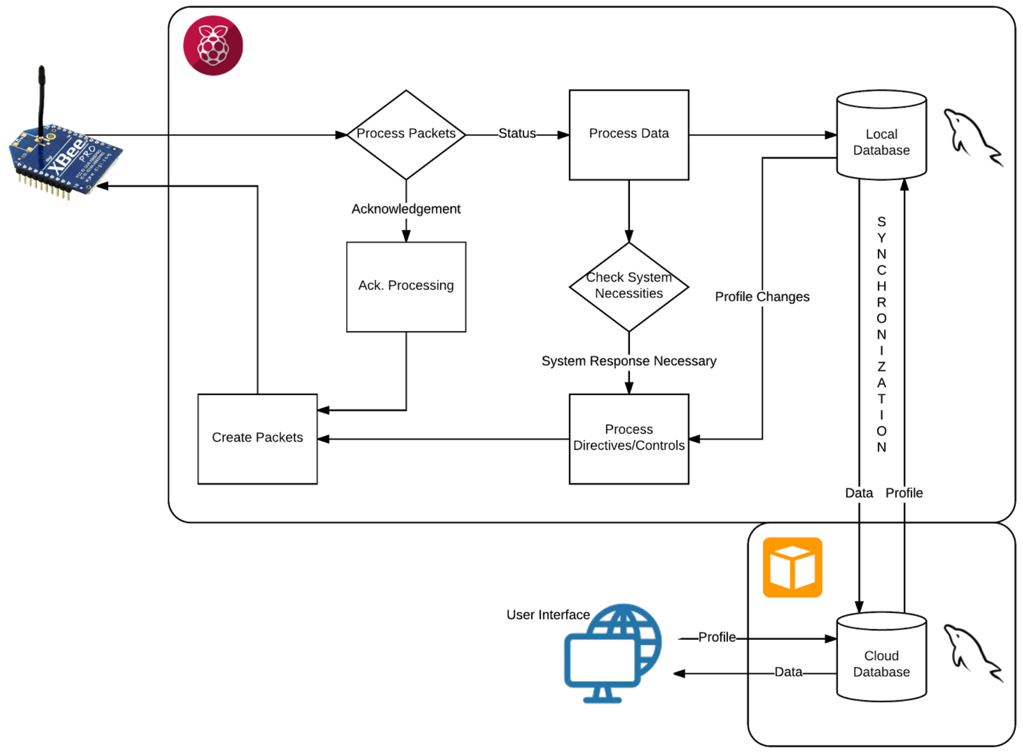
The pTera ground station model is a multipurpose structure that integrates a directive hub and data aggregation process that accomplishes four main objectives:

- Receive and parse incoming/outgoing packets
- Store status data into the local and cloud databases (local-to-cloud synchronization)
- Store profile changes and user configurations (cloud-to-local synchronization)
- Relay messages between other network entities
- Create control packets and directives

Overview

Each of the above objectives utilizes a specific platform for its function and corresponding programs or utilities to run in that platform.

The diagram below illustrates the structure and flow of our model.



The next few sections will detail the various components of this configuration.

Platform Hardware	***Raspberry Pi 3***
Communications Hardware	***XBee***
Local Database	***MySQL (hosted by nginx)***
Programming	***Python***
Cloud Platform	***Amazon Web Services***
Cloud Database	***MySQL (hosted by AWS)***

Hardware and Configuration

This model not only handles data, but acts as a ground station and handles most communications that occurs across the pTera network. A **Raspberry Pi 3** is at the core of the system, combined with the provided Raspbian OS, the Pi acts as both a database server as well as a command and data processing hub. The Raspberry Pi 3 is powerful, energy efficient and is able to host the software needed to develop our form of communication for our system. It is small enough to be portable and important when used during the flight. Additionally, the Pi 3's processor allows it to handle incoming and outgoing commands and requests in order to communicate with the drones, as well as larger data types such as images or video. While USB ports are available and are utilized in this system, its GPIO pins offer additional flexibility and I/O options.

XBee Network
XBee offers a solution for mesh-style network connections that work for distances up to a mile apart. XBee also offers a wide range of additional features that make parsing and receiving data less of a burden on our system.

The XBee network involves multiple drones in the sky as well as the ground station. Each of these vehicles have their own XBee module and are constantly sending data to each other. The ground station's primary network responsibility is to receive sensor, time, and location data from the drones in the air.

For the ground station, the XBee is connected via USB serial port to the Pi. This connection is explained in the following section.

UART Configuration for XBee
The Pi offers two primary methods of UART configuration. The first is through UART connection pins on the GPIO array.

The other is via USB. Using this method is a lot easier, especially because we are using an XBee adapter to properly interface the XBee module to a USB port. To utilize this connection, the Pi stores incoming data into the *ttyusb0*/*ttyusb1* folder. This will be referenced later when discussing the serial interfacing code.

Database and Architecture
Local Server: nginx and MySQL
To host the database for the ground station, nginx (a web server tool) was used to create a web server. There are well-documented guides on how to create a server on a Raspberry Pi with nginx, making it an appealing choice. MySQL was used for the actual database server and phpMyAdmin was used for the management interface to edit the database due to its simplicity in setup primarily. The web server location was left blank since nginx was used to create a local web server.

MySQL Integration
MySQL code is integrated into Python scripts that manage the data processing through a Python library dedicated for this use. This allows for automated interaction with the database, allowing real-time data to be added continuously.

Web Server: Amazon Web Services and MySQL
To host the database on the cloud, we used Amazon Web Services (AWS). Within the AWS options, we selected to create a MySQL server as this would make database manipulation code compatible with either the local or cloud databases. By using AWS, we also allowed the end user to access their data from any device, either through our MySQL workbench or through our provided web interface.

Data Architecture

The database on the ground station consists of separate tables for each drone. In every table, the time at which the data was received, along with the coordinates (longitude, latitude, altitude) is recorded. Additionally, a pair of columns is added for the number of sensors specified. These columns are used to record the raw sensor data, as well as a computed score based on the sensor reading to determine how “interesting” the data is.

Processing and Flow

All programs and utilities are part of a suite that was used to describe a particular data architecture. This data architecture as well as the flow of the data stream will be detailed in the following sections.

Overview

There are three distinct flows as part of this model: the data flow, the directive flow, and the acknowledgement flow.

The XBee first receives packets from the other network entities (in this case, the UAVs). It passes along this data to the Raspberry Pi through a USB connection. The packets are first parsed and based on the operational code, the packet is sent to the appropriate flow.

Data Flow

The data flow entails the process of receiving statuses from the drones and properly storing that data. The flow begins with receiving parsed packets containing important information including sensor data, location data, and battery status. Parsed information is then handled by a data processing thread of the main aggregation program. The processing of this information includes calculating the moving average of the last “n” samples, where the “n” is configured by the user. This process automatically removes outliers and bandpasses the center 80%. The processed information is then stored into the local database. Any new data stored into the local database is also queued to be stored into the cloud database, provided that a connection to the internet is available. Until the connection is established, this data is stored in program memory.

Directive Flow

The directive flow entails both UAV system responses and enacting profile or directive changes from the user end. There are two entry points into this flow. The first is if processed data from the data flow realizes a need for system change (ex. Drone out of range, low battery levels, etc.). The second is if a user makes a change using the interface (ex. New area to be scanned, manual request for data, manual reassignment of responsibilities).

In either case, the appropriate response is created and the solution is sent to a packet creation script. The directive is encoded with the control operation and code and sent to the XBee, which is then set to transmit the packet to the other network entities.

Acknowledgement Flow

The acknowledgement flow entails listening and sending acknowledgements of certain packets. Upon initialization of the entire system, UAVs will broadcast themselves and their information, to which the ground station will send an acknowledgement to each UAV in a “handshake” behavior. The UAV will then recognize this acknowledgement, completing the individual initialization of each UAV to the database.

During program operation, the directive flow may result in control packets being sent to the UAVs. The ground station will listen for an acknowledgement back from the respective UAVs to stop transmitting control packets.

Program Descriptions

Python was selected as the main language of our programs, due to its high flexibility and ease of use. The use of Python also enabled us to use a number of different libraries to easily interconnect with other utilities, such as configuration management and MySQL.

Descriptions of the programs and functions are shown in appendix A.

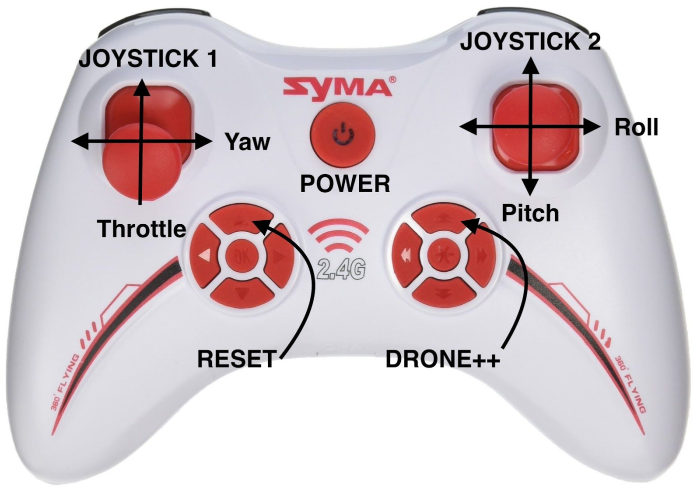
pTera Universal Manual Drone Controller

Overview

pTera has a universal manual controller that overrides all autonomous drone flight instantly for any selected drone on the system. A manual override is required by FAA regulations. The manual controller uses the existing case for the Syma X12 Mini Nano 6-Axis Gyro 4Channel RC Quadcopter controller. The controller works exactly like a normal drone controller works but it communicates with any specific drone part of the framework.

Controls / How it works

The controller joysticks and buttons are connected to an Arduino Mini. The Arduino Mini within the controller shell is constantly updating a manual override control packet which is transmitted via an Xbee module. All additional components fit within the handles of the manual controller chassis. Below is a visual representation of the controller along with a key for all the controls. As you can see, the manual controller is a standard drone controller, so the controller is very intuitive for any user with any drone experience to use.



Joystick 1: Controls drone yaw and throttle
Joystick 2: Controls drone roll and pitch
Drone++: Changes connected drone to next drone
Reset: Sets connected drone to drone 1

Internal Controller Wiring

The controller joysticks and buttons are connected the the Arduino Mini. The Arduino Mini creates manual control packets that are sent towards the XBee.

MAX_DRONES Macro

The manual controller is preset to control up to 3 drones. The controller can communicate with any drone within the framework. As the number of drones within the pTera framework expands, a macro for MAX_DRONES must be updated to the number of total drones to be manually controlled.

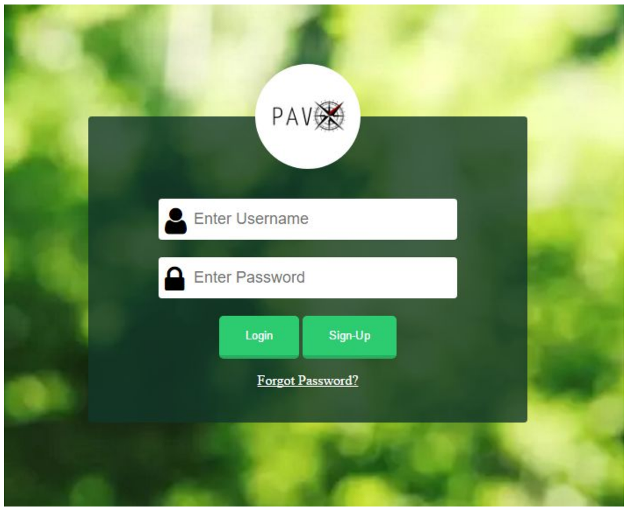
pTera User Interface

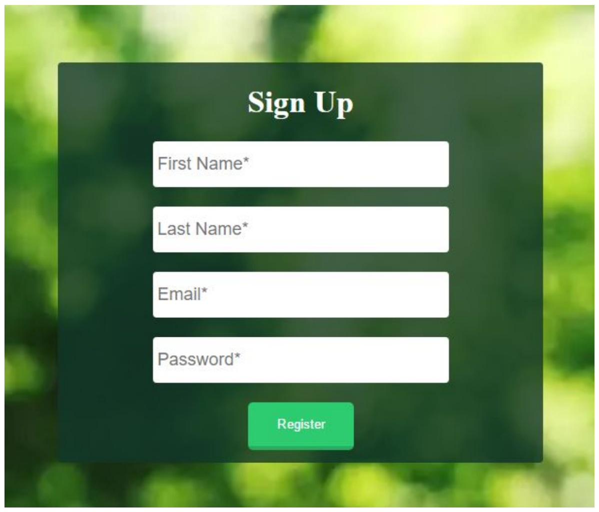
Overview

After structuring MySQL database, the next major goal was to visualize and have the capabilities to interact with our data, either by reading, recording, or loading new values. Our UI possesses functionalities to communicate and redirect commands, mainly serving as a central command station for our project. We decided to use PHP as our main language since it was the fastest to learn given our minimal experience on web design, and had straightforward interfacing to MySQL. Our web UI includes other user-friendly features such as real-time tracking, displaying a heat map representing our points of interests and the ability to view raw data through our visualization to create further analysis of our collected data.

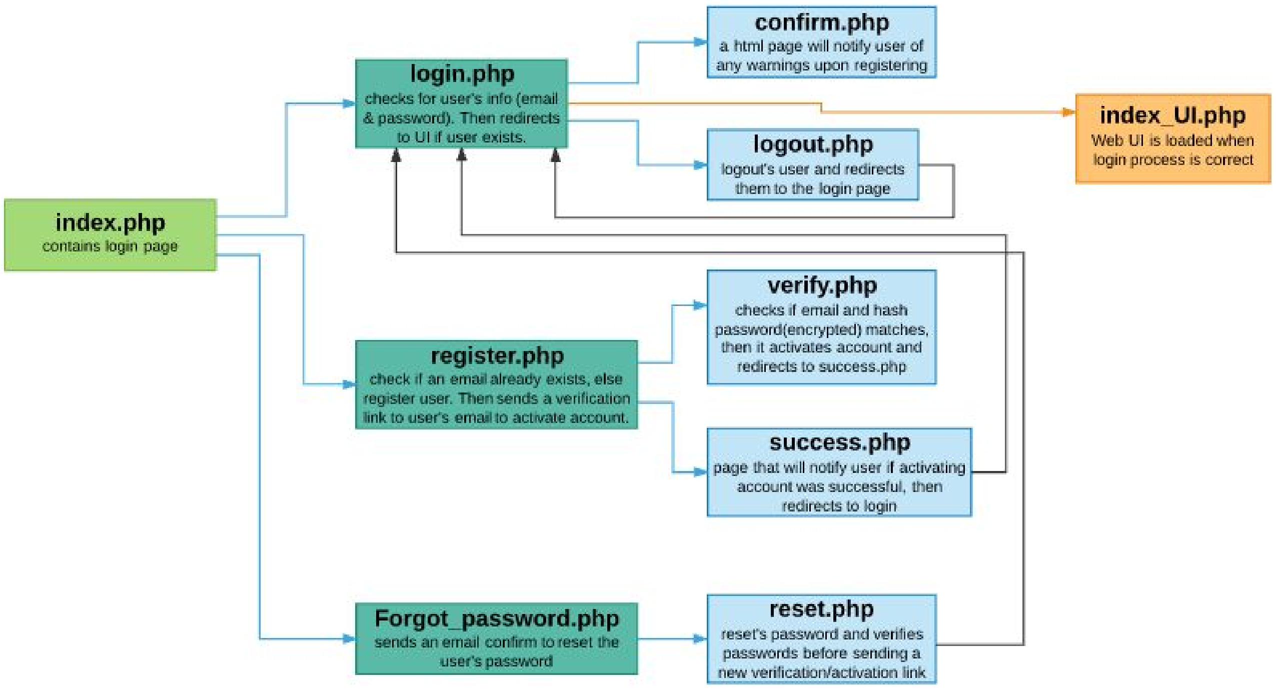
Layout

The interface allows for the integration of user control, database, and networks to allow the user to communicate with the drones. The UI was designed to be dynamic in order to stay flexible along with our database. We created a login page where the user can register in order to access our web UI. In case the user forgets their password, they can simply change their password by selecting the ‘Forgot Password?’ link which sends an email link to the user, allowing them to change their password.

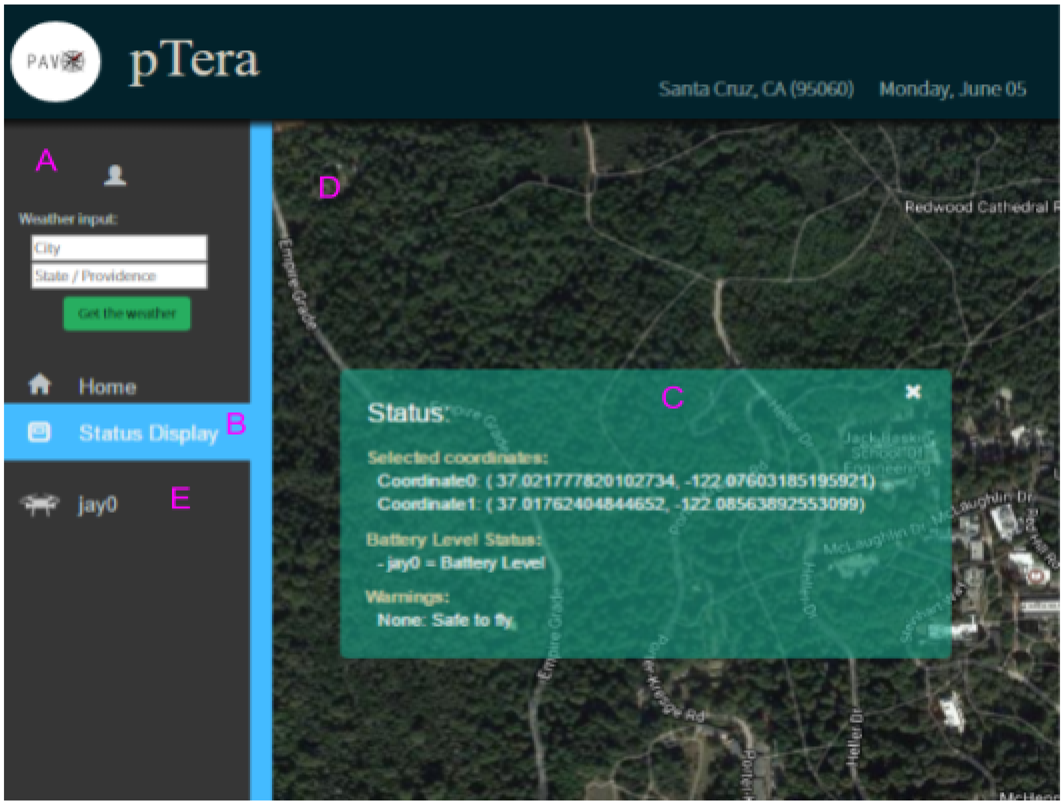




The diagram below represents the login structure and process of what needs to get done before accessing our web UI. It also has a description of what each file does to secure the process when logging on. We wanted to keep track of the users accessing our web UI and also create a secure way of limiting who is able to use it, since the user is able to send commands/control the drones.


index.php
contains login page
login.php
checks for user's info (email & password). Then redirects to UI if user exists.
confirm.php
a html page will notify user of any warnings upon registering
index_UI.php
Web UI is loaded when login process is correct
logout.php
logout's user and redirects them to the login page
verify.php
checks if email and hash password(encrypted) matches, then it activates account and redirects to success.php
register.php
check if an email already exists, else register user. Then sends a verification link to user's email to activate account.
success.php
page that will notify user if activating account was successful, then redirects to login
Forgot_password.php
sends an email confirm to reset the user's password
reset.php
reset's password and verifies passwords before sending a new verification/activation link

Below is a labelled screenshot of what users see when the web UI is first accessed. Each of the labels represent the drone's different functionalities which will be discussed.



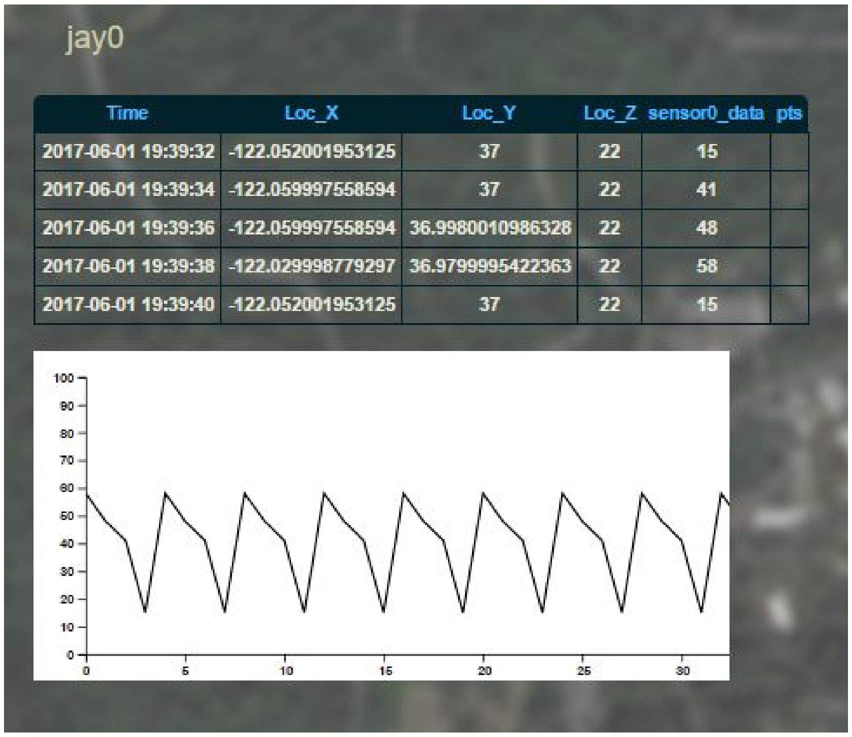
Status:

Selected coordinates:
Coordinate0: (37.021777820102734, -122.07603185195921)
Coordinate1: (37.017624044844652, -122.08563892553099)

Battery Level Status:
- jay0 = Battery Level

Warnings:
None: Safe to fly.

A. The tab selection represents the different databases that we have created, including our home base and the drone's database. The drone's tab contains each drones' information (including time stamp, longitude, latitude, points of interest, raw sensor data, etc.). This tab also holds our data visualization used to make further analysis on our collected sensor reading/raw data.
B. We enabled user control to type in the location so our weather parsing program can identify the area of interest and determine if any severe weather conditions will occur.
C. We created a window display that keeps track of all the drones' battery levels and monitors the weather and the area that the user selected to scan. It also helps to see the last location where a drone loses its signal.
D. The "start" button initializes streaming of live data by retrieving the most recent GPS location based on the timestamp from every drone in our database. There is a helper function that will create markers on our Google map. This layout is only displayed on the home tab. We included another button specifically for selecting the user's desired area of interest and will verify if the scan is possible given the drones' current battery levels, before uploading the coordinates onto our database.
E. Our data visualization presents our drones' information depending on which drone the user selects from the dropdown menu.



Time	Loc_X	Loc_Y	Loc_Z	sensor0_data	pts
2017-06-01 19:39:32	-122.052001953125	37	22	15	
2017-06-01 19:39:34	-122.059997558594	37	22	41	
2017-06-01 19:39:36	-122.059997558594	36.9980010986328	22	48	
2017-06-01 19:39:38	-122.029998779297	36.9799995422363	22	58	
2017-06-01 19:39:40	-122.052001953125	37	22	15	

Database Integration

For our UI development, PHP and HTML were used to display data from our local and cloud database. We decided to use PHP which has the capability of connecting and communicating with our server giving us control over the design/style for our UI. The descriptions of each PHP file will be expanded on in Appendix C, UI Functions.

Reading from database

An important note that is essential for our UI was figuring out how the browser (UI) is able to communicate with our database (server). We used AJAX, **A**synchronous **Ja**vascript and **X**ML, to have our web UI more interactive and to create dynamic content (such as retrieving and displaying data from our database) without reloading the page. XML, also known as extensible markup language, stores data in plain text format that can be used to store, transfer or exchange data over the internet between different applications. A user can continue to use our web UI while the client program (wrapper files) requests information from our server in the background. JavaScript and the XMLHttpRequest (an API) object provide a method for exchanging data asynchronously between the browser and server.

The HTTP request has event handlers that will listen for the status of the event changes to be complete, that way when the response is done it can continue to execute any other HTTP requests left in the program. This is really useful when updating the status, Google map section and when we want to visualize and analyze our data on the same webpage. This process required Javascript to execute a function call that starts an HTTP request and calls one of our wrapper programs. The wrapper programs are written using PHP to make a SQL query to retrieve or upload the data to and from our database. The wrapper program then returns an XML response containing the HTML format and data to be printed onto our browser. The descriptions of the the individual database files will be expanded on in Appendix C, UI Functions.

Loading coordinates and flight status to database

We created different parser programs(wrapper files) that pertain to one functionality, such as loading or creating the map marker and displaying the markers on the Google Map. It makes it easier to debug, which was most likely the case every time we tested our UI.

Map functionality

Forecast

As mentioned before, we implemented a weather forecast that can alert our drones on whether it is safe for them to fly. We decided to use the underground weather API since it's user friendly and open source. The conditions from an API call creates a json file with the current location's weather, current temperature, wind speeds, humidity, barometric pressure, and visibility. Useful information that will benefit our drones. We then created a parser file (*weather_getInfo.php*) that will help parse the contents and information that we need from the generated json file. Further information can be found in Appendix C.

Heatmap

The heatmap mentioned earlier is displayed as a layer on the Google Map. The map is initially off, and can be toggled to display or not by clicking a button. Most of the functionality for the heatmap is handled automatically by the Google Maps API, all that is needed is to pass in the latitude and longitude into the array of heatmap data and it will be updated. The heatmap is periodically refreshed, so that it obtains any new values and updates the map. Further information can be found in Appendix C.

User control

Since we have everything setup, we then enabled user control so our UI became our central station to redirect commands for our project. Once the user selects the scan button to scan the area there will also be a verification check when the user selects the check button. On the Drones side, they will receive a signal with the coordinate data and will determine how the area will be distributed evenly, by assigning each drone a specific scanning area on the map. The user can also live stream the drone's location which are represented by the google markers when selecting the "start" button.

Visualization

To visualize the data, the Javascript library D3.js was used in conjunction with the data obtained from visualization.php. The D3.js adds a graphical element to the page and draws a data-bound line with the derived points value from the sensor readings on the y-axis and the how recent the value is on the x-axis (most recent, second most recent, etc.). There is no efficient way stream the data live, so instead the data refreshed every few seconds.

Communication Networks

Network Protocol/Hardware Selection Process

Our initial thought process to finding a suitable wireless technology for drone to drone communication was to discover an existing UAV (unmanned aerial vehicle) communication framework. Naturally, the next step that followed involved coming up with a short list of hard requirements, as well as extensive research. The below list showcases some of the essential features that we required.

- Reliable transmission and long range
- Cheap hardware
- Easy to use hardware
- Simple manipulation

MAVLink

One of the first technologies we examined was a communications protocol called MAVLink (Micro Air Vehicle Link). MAVLink gave us a method of reliable transmission as well as an extremely long range as it was originally designed for use with rocket hobbyists. However, even though it satisfied one item from the list above, it failed in every other aspect. The hardware that we would have to purchase was custom made, and the hardware itself was difficult and bulky to configure. Furthermore, MAVLink required a plethora of C, C++, and Python libraries to use, much too complex and necessary for us.

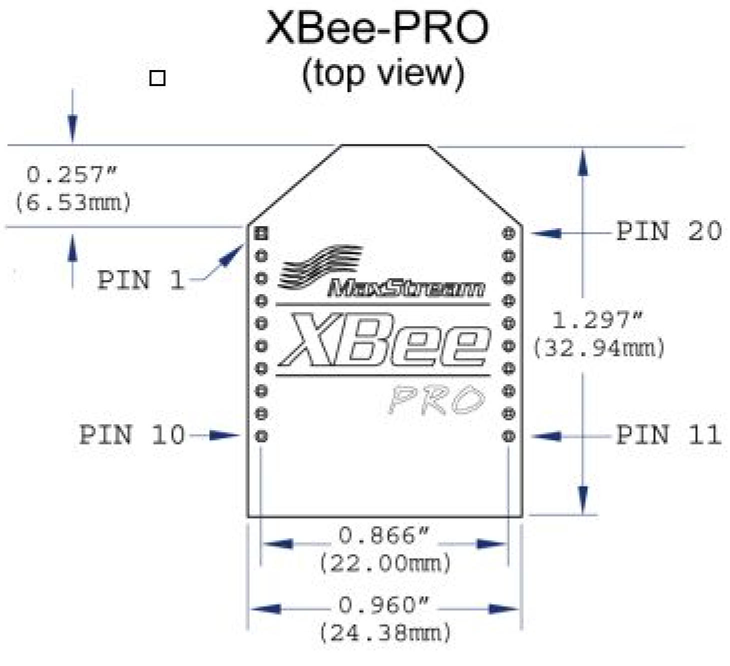
XBee

After doing further research, we realized that MAVLink's communication protocol completely depended on XBee, a wireless 2.4 GHz communication protocol. XBee satisfied every requirement that we had. The transmission was purported to be quite reliable, and the datasheets guaranteed a 1-mile range given a clear line of sight. The hardware was also relatively cheap, coming out to around ~$50/module. Furthermore, the hardware was incredibly easy to setup. Through four wires, we could obtain complete control of the XBee module. XBee modules can also be very easily manipulated from the command line, using a set of commands known as AT commands.

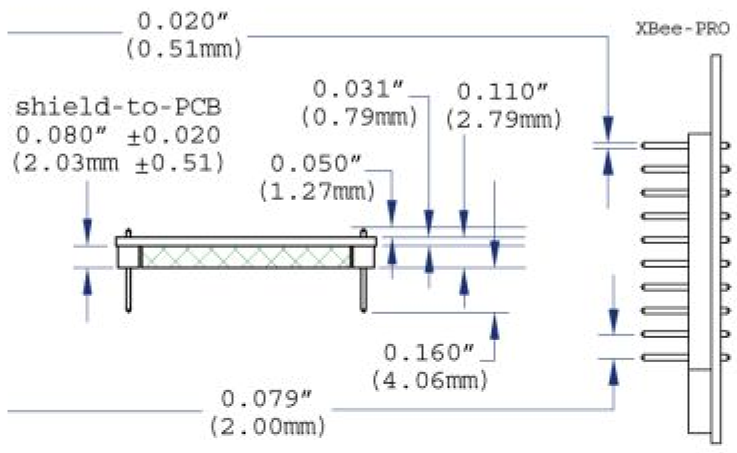
Physical Layer

Dimensions

The XBee module comes with its own flexible antenna as well as 20 configurable pins. The below is a datasheet schematic of the dimensions of the XBee module.



Top down View



Side View

Pin Connections

Of the twenty available pins, only four were of use to us. Pin 1 is VCC, whereas pin 10 is GND. Both of those pins were used for power, connected to a 3.3V power supply. The only other two pins that we used were pins 2 and 3, which were used for Data Out and Data In, respectively.

Limitations

Data Rate

The XBee Pro module has a data rate of 250,000 bytes per second. This is vastly more than what we require, as the largest packets that we send out have a size of no more than 200 bytes. As such, bandwidth is of no concern for us during this project. In terms of expandability however, the XBee Pro module is capable of streaming high quality videos with high latency. At that point it may be conceivable to attempt to use multiple XBee modules per drone to “daisy-chain” the bandwidth to bypass the limitations.

Distance

The XBee Pro module is rated for up to a mile difference between two modules, given a clear line of sight (meaning two obstacles in the way). During our field tests, we confirmed that we were receiving clear packets with little to no latency up to about ¾ of a mile. As such, for our demonstration purposes this is acceptable, as we will be scanning very small areas.

Vehicle to Vehicle Communication

Hardware/XBee Integration

The XBee modules are mounted onto each individual drone using pin headers that we designed on our PCB. The PCB has slots for all twenty pins of the XBee, but only four of those pins are routed to anything (VCC, ground, data out, and data in).

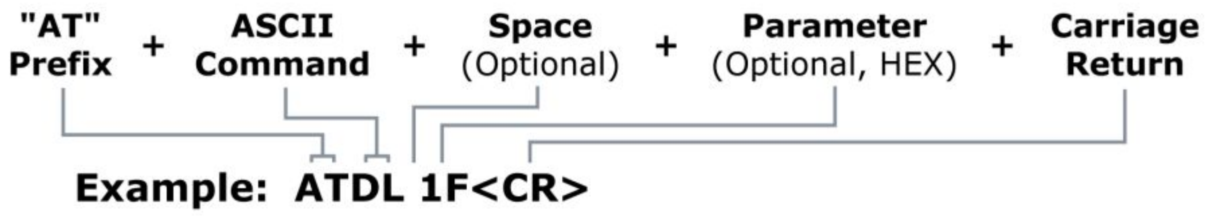
XBee Use in pTera System

There are many settings that can be configured for each XBee module. The most important setting is the configurable PAN ID. ZigBee devices communicate on a UDP network sharing the same PAN IDs. All of our drones and base station share the the PAN ID 0x3332. The XBee modules communicate on a connectionless network, where every module is always “listening” and receives every transmitted packet within its ranges. The XBee modules have Data In and Data Out pins that work as RX and TX pins respectively. Every data byte sent via hardware UART from our microcontroller to the RX pin will be sent wirelessly to the XBee network. Our drones’ microcontrollers and base station’s Raspberry Pi receive data from the TX pin of the XBee modules. The XBee modules also have collision avoidance when receiving packets from multiple XBee modules at the same time and ensure proper packet handling.

AT Commands

The XBee modules are easily configured through software. AT commands are used to configure and read the current settings for the connected module. As stated earlier, everything sent to the Data In pin to the XBee module is transmitted on the XBee network. The XBee module must enter Command Mode to access the local module configurations. To enter AT Command Mode, we send “+++” to the Data In pin of the XBee module.

We built a custom XBee AT Command library that enters the XBee into command mode and retrieves and sets information. Utilizing the library, the modules can return signal strength, module identifiers, discover nearby modules, as well as reset the module names. Every AT command must be in the following form:



The AT commands we found necessary are located in Appendix D.

UAV: Mechanical Hardware

The drones that we use have all been custom made to fit our application need.

Parts List

- Drone Frame (xt250 acrylic)
- 9 degrees of freedom inertial mass unit
- Barometer
- NEO 6M GPS breakout
- Custom Flight Controller
- 3 cell Lipo Battery
- Omron Temperature Sensor
- XBee Module Pro

Mechanical Constrictions & Decisions

One of the purposes of this project was to come up with an efficient, cheap, and easy to make drone design. To this end, the first challenge was to find a suitable material that was lightweight and cheap. An easy solution for this was to use acrylic. Acrylic was cheap, easy to obtain, and easy to work with, as the laser cutter provided to us could very easily cut through acrylic. There were several other options including impact resistance acrylic (too expensive), lightweight acrylic (non-flexible), as well as carbon fiber (too expensive).

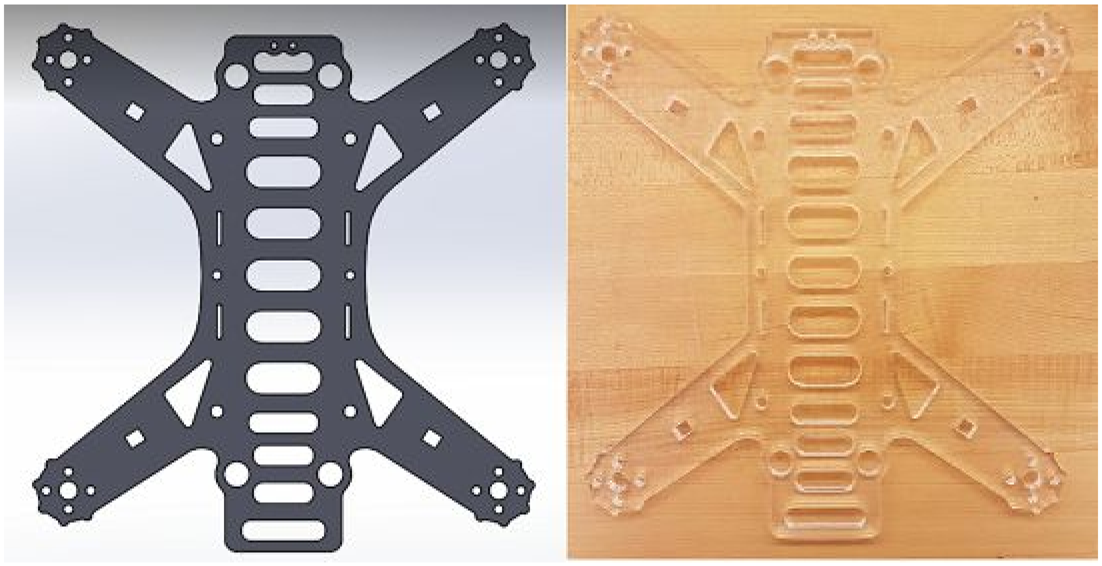
Next, we designed to utilize a four-rotor drone design as opposed to more than four, simply because four rotor drones are efficient, as well as easy to easier to design than other multi rotor designs.

Finally, we had to consider mechanical interferences such as vibration. To solve this potential issue, we built custom PCB frames using a thinner acrylic, as well as rubber mounting grommets to dampen vibration.

Frame Design

Main Frame

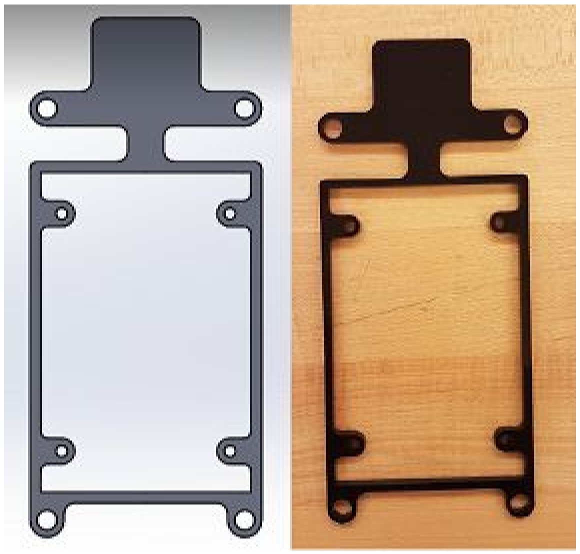
The main frame of our design consists of a horizontal platform that can mount four motors, various controllers, and our main flight controller PCB.



As can be seen from the above image, there are four “wings”. Each end of the wing contains a collection of holes which are used for mounting our motors. There are also four larger holes in the center of the frame for mounting our PCB acrylic bracket. Additionally, as much additional material was removed as possible without compromising the integrity of the frame (hence the myriad of oval holes).

PCB Mount

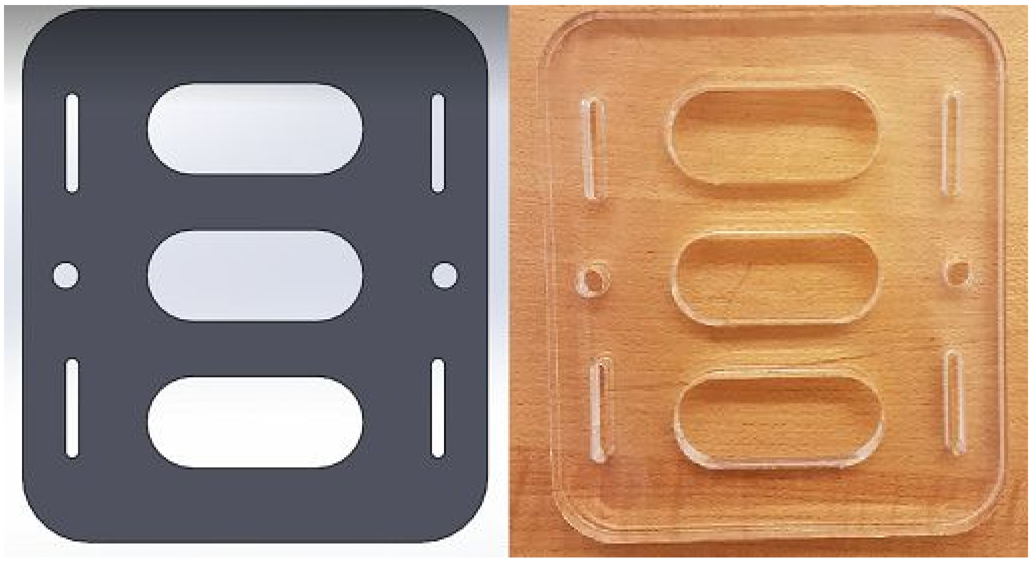
Since the motors are mounted on the main frame, they introduce significant amounts of vibration into the main frame. To solve this, we designed a PCB mount which will hold our flight control PCB away from the main frame.



This allows us to use materials such as rubber grommets to dampen the vibration coming from the main frame. This part was made with a slightly thinner acrylic, to add as little weight as possible to the drone. It also has a large rectangular area for mounting our GPS hardware on to.

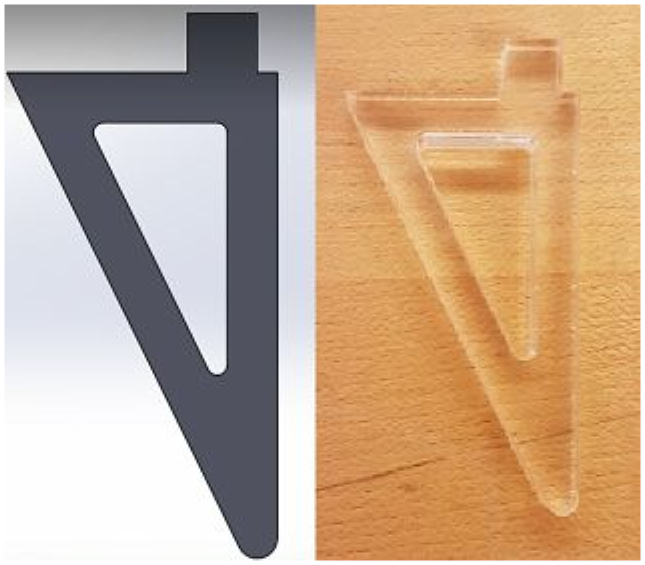
Battery/Sensor Mount

To hold the battery in snugly to the drone, we created a simple acrylic plate to hold the battery in. This works by using Velcro to strap the Battery/Sensor Mount, battery, and Main Frame together. This mount also has another use in that it is the primary method for affixing bottom facing sensors, using any of the precut holes in the acrylic.



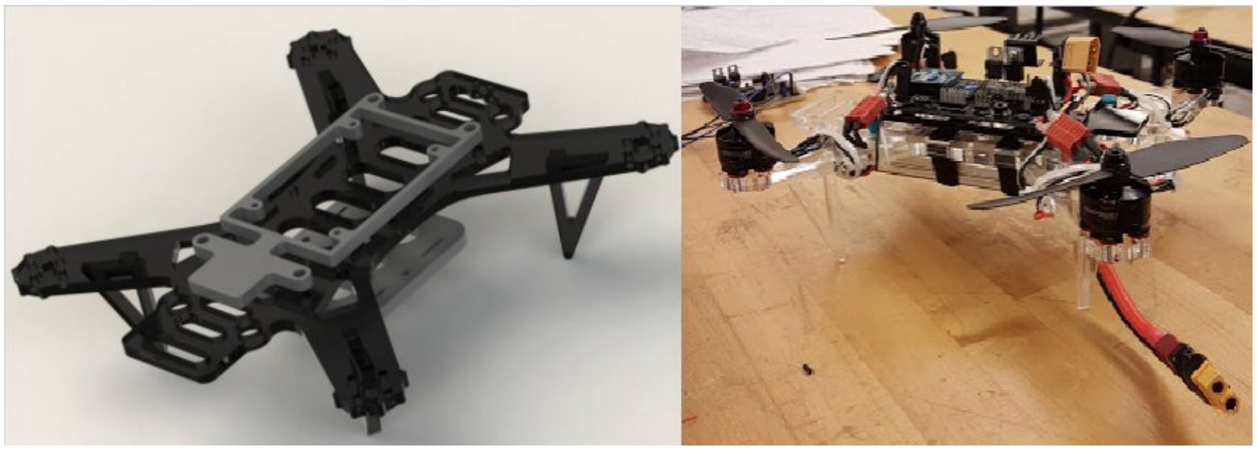
Legs

Finally, we designed a set of four legs that are attached to the main frame with acrylic glue. This legs were designed to be large, but also small enough so that they did not add significant weight. The legs themselves were hollowed out to prevent buckling that occurs from hard drone landings.



Final Frame

Our final drone design is a highly effective, yet simple design. All the parts together cost about $130 which is quite inexpensive for a drone that has this magnitude of capabilities.



Motors and Speed Controllers

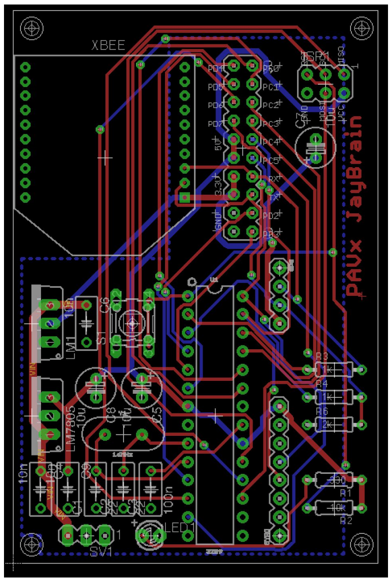
For the drone we are using brushless motors for efficiency. These motors are powered using 3 wire signals which works by sending sinusoidal waves each with a specific phase difference from the other. Luckily, it is a common practise to use an additional speed controller to maintain a constant output to the motors. These make it easy to program with the microcontroller as they act similar to servos; the duty cycle is directly proportional to the speed. The motors are rated at 2300 KV, which represents the amount of back emf generated per volt of input. In our case this value is small enough that it will not cause fluctuation in our digital circuitry.

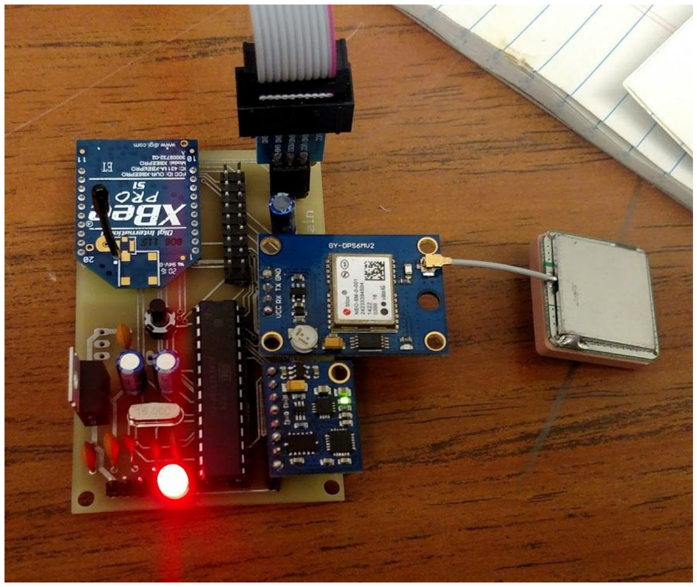
UAV: Electrical Hardware

Flight Controller Overview

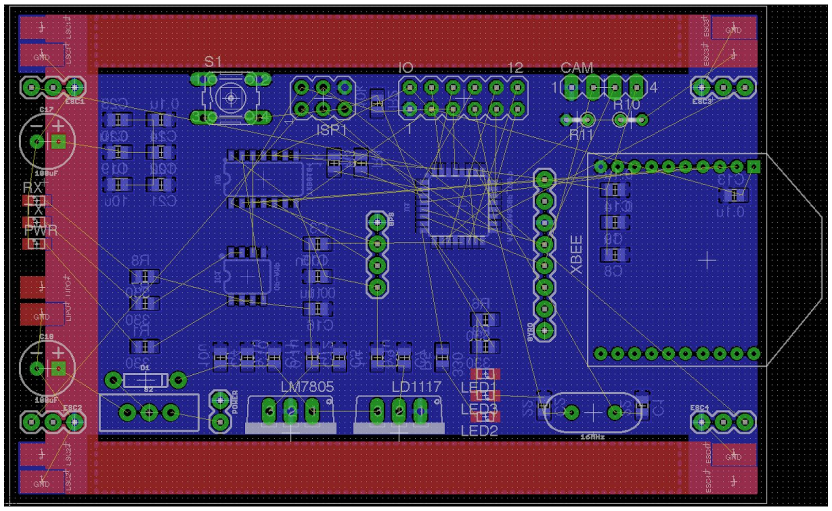
For this project we decided to pursue a custom solution for the flight controller. The main microcontroller is the Atmel ATMEGA 328p. This is a pretty common microcontroller found on many Arduinos. We felt that it would be a nice challenge, and great learning exercise, to try and fit the entire controller on this chip. This meant that we have to be careful with our code and optimize well. In a way this is forcing us to write good firmware.

The initial requirements for the first iteration of the PCB was to create a development platform. This was meant to be a platform to test our initial firmware. As a result we had extra pins routed to pins so that they could be easily accessed by the oscilloscope. Below are images of the development PCB.

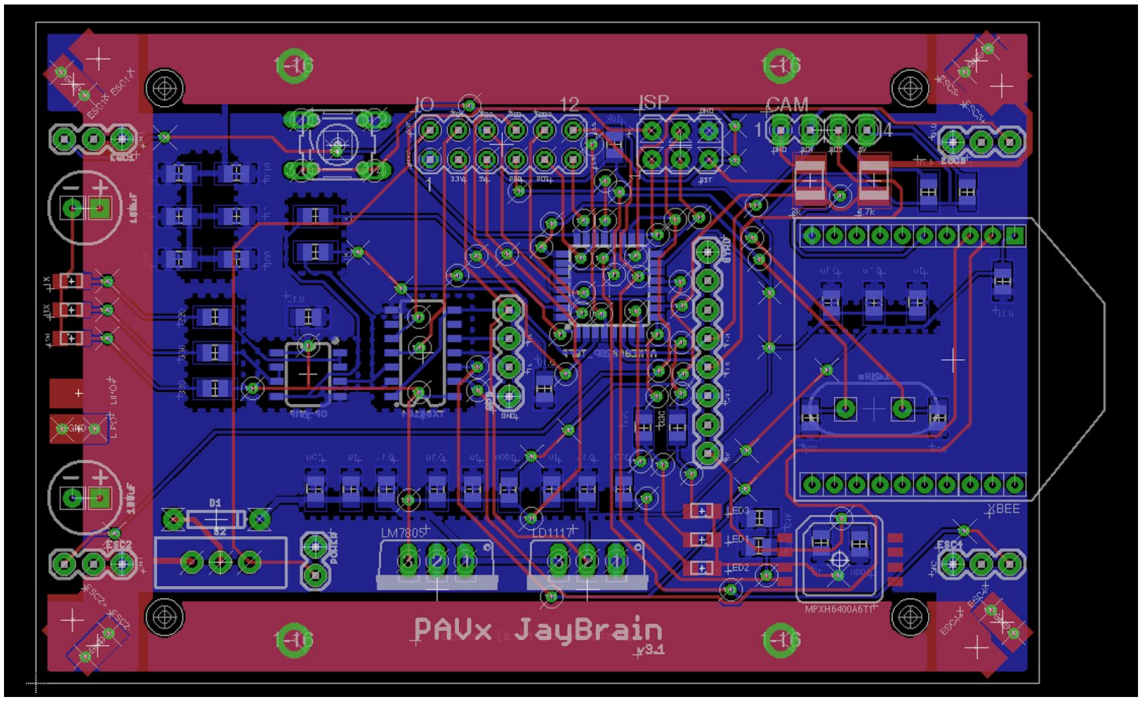


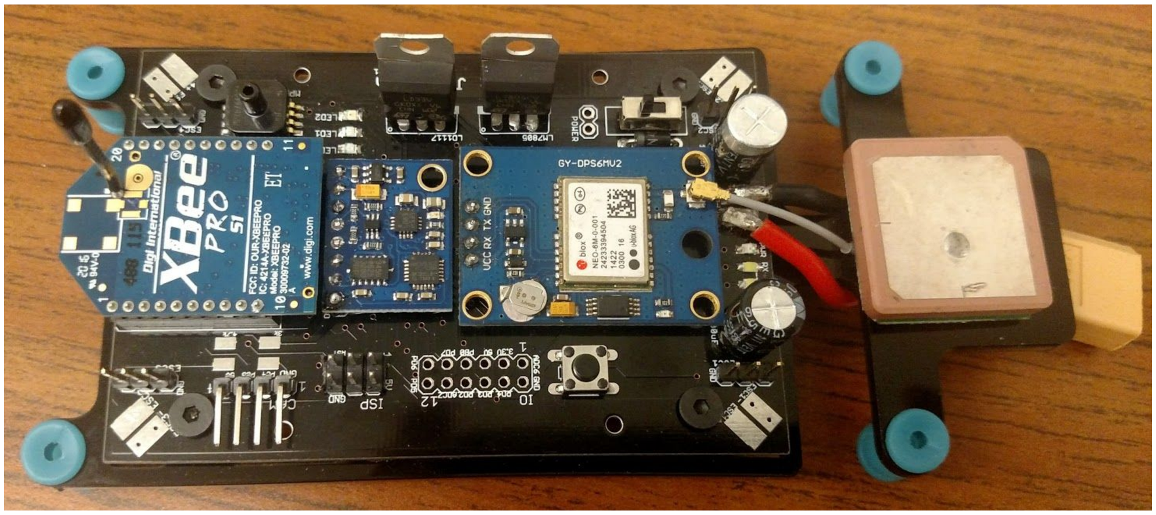


For the second iteration PCB, we looked at a more production-oriented design. Now that we have the majority of the firmware, there is no need to breakout all the pins. In addition, surface-mount devices were going to be used to decrease the electromagnetic noise resulting from through hole components. Along with the flight controller, the PCB will also house the power distribution board to minimize the number of components. Below is a picture of the proposed second iteration PCB.



Lastly, the third iteration of the PCB fixed minor spacing problems in the first PCB and added a barometer sensor to accurately predict the altitude. The final version of the PCB is shown below in both layout and physical form.





Power Distribution

The addition of power distribution within the PCB helped reduce the overall clutter when making the drone. As seen by figure above, the red copper pours indicate the power lines to the speed controllers for the motors. An important consideration is to ensure that there is enough copper pour to support the maximum current the speed controller can sink, which is 25A. Rather than make the copper pours larger, another pour was added to the opposite side to keep the overall footprint small while being able to transport a large amount of current. Since we expect fluctuation, an array of ceramic capacitors were added for quick response, in addition to a large electrolytic for slower, but larger spikes.

Power Cutoff

A safety procedure that we have implemented is a low voltage battery detector. Since we are using Lithium Polymer batteries, it is important to keep an eye on their battery levels to prevent damage. This is done through a set of high resistance ½ Watt resistors, which step down the voltage to a range that the microcontroller can accept. From here the ADC reads the value and lets the system deal with it. Ideally we would like the drone to stop what it's doing and immediately head home for recharge.

Sensor Port

The flight controller has an open I2C bus that allows for any I2C device to be used with it. This allows for modularity when trying to use the drone for different applications. Currently this is used for an IR sensor to detect hot spots. By placing the pins used for the I2C sensors on certain pins of the microcontroller, we are also able to expand the capabilities of our sensing interface to UART, SPI, ADC, and I2C compatible sensors.

High Speed Digital Design Considerations

Since the clock on this microcontroller is running at 8MHz, there will be unexpected consequences of high speed digital design. Although transition times are expected to be instant, at high clock rates, these become more apparent and Gaussian in shape. To get around this, all digital devices are run on a synchronous clock. To avoid ground bounce, all devices have a quick route back to ground thanks to a ground plane.

Board Bringup

Since the all the soldered chips and sensors came straight from the manufacturer, they were not initialized. In order to begin writing code for these flight controllers, we needed to initialize them to our bare metal environment. The Atmega 328p contains fuses which can be set to alter its startup behaviour. For our environment, we had to ensure that the microcontroller was using the external crystal and that it maintained its EEPROM after fresh flash. In addition to this, we ran an initialization program which took care of preliminary calibration of the gyro and accelerometer. The program would gather a given amount of data points and average them out to get the proper offsets needed to use on IMU data. These values were stored in EEPROM. When running the main script, one of the first things the program does is grab these values from EEPROM and stores them locally. This dramatically cut down our startup time as we no longer had to reinitialize after every new reboot.

Future Considerations

For future flight controller boards, it would be nice to have the inertial measurement unit as a standalone chip rather than a breakout board. If time permitted the MPU-9250 chip would have replaced the GY-85 breakout board. This would help avoid calibration each time due to constant offsets. For future expansion, an AVR Atmega 2560 would be added alongside the Atmega 328p. The 2560 would handle all the sensor input and communication while the 328p would handle the controls.

UAV: Software

GPS

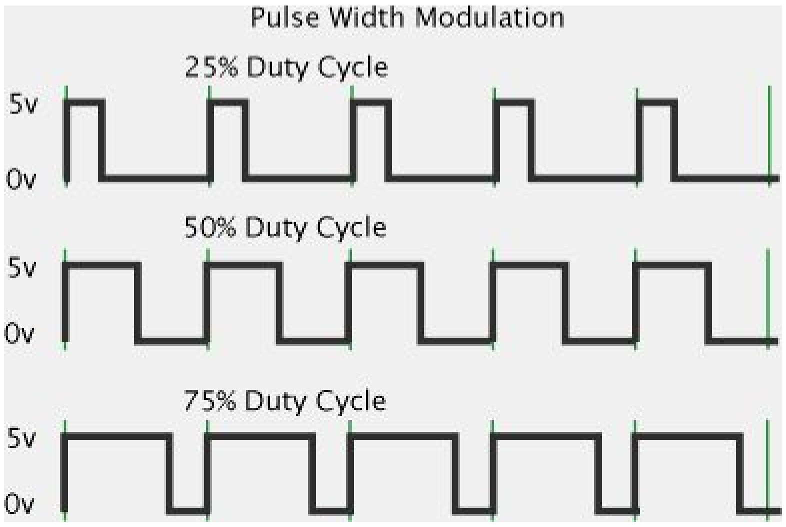
The GPS supplies the position and speed data by parsing the desired information found in the NMEA sentence types. The GPS defaults to sending new data every second. We chose to read two sentence types (RMC and GGA) to gather the specified information. This method is more practical than polling because we do not need to take time sending requests to the GPS. The GPS reads the packets one character at a time in order to prevent the system from being blocked while waiting for a complete sentence. The sentence state variable keeps track of the reading status to know when a full sentence is ready for parsing. The module can be configured to only output the desired sentence types to avoid clutter. Timestamps from GPS sentences are checked to align the information from the different types to assure data accuracy. The GPS is configured beforehand to communicate at a baud rate of 4800 to meet the specifications of our current microcontroller limitations.

Software UART

As mentioned above, it was a challenge to fit everything onto the microcontroller. Since the XBee uses the hardware UART, we were left with the objective of creating a software UART for the GPS. Through this process we learned to optimize the interrupts to act quick and only when needed. The software UART works in full duplex operation and has been tested thoroughly with a baud rate of 4800.

PWM

To control the motors, pulse width modulation (PWM) was used. To achieve this, we used two timers (timer0, timer1) from the 328p microcontroller. Using the output comparison register, we were able to trigger a pulse at any desired frequency with a given duty cycle. For this application we used the 328p's fast PWM mode to achieve 8kHz. For the speed controller, the duty cycle represents the throttle of the motor. A low duty cycle indicates a low speed whereas a high duty cycle indicates a high speed. Below is a diagram illustrating this.



System Tick

Our microcontroller features three timers that are occupied by motor controls and software UART applications. The system tick is a software counter driven by the compare interrupt used for the software UART. The timer is therefore scaled by the rate of the compare interrupt which was calculated to be 68µs per tick. Multiple compare values were used to set flags at various periods. The compare values are increased by its period when the timer reaches the compare value. The system tick is robust and designed to handle edge cases such as overflows and missed ticks. The timer system was used for handling general periodic events. Timers were initialized by presetting the compare values to the desired period before starting the system tick counter. On compare matches, the corresponding flag is set. The timers are used by calling a flag check function and flag clear function.

Packets

Packets were sent and received through the XBee module connected to the UART pins. As with the GPS data, we continuously checked for incoming data waiting to receive a complete packet. The packet format began with a distinct header and ending byte that helped us detect a complete packet. We had several types of packets that were characterized by a unique opcode.

- Manual Control (0) - yaw, pitch, roll, throttle values from manual controller
- Status Packet (1) - drone sensor and location data
- Control Packet (2) - base station to drone command packet
- Initialization (3) - drone to ground confirmation
- Initialization Acknowledge (4) - ground to drone confirmation

These packets had a corresponding packet handler that was called upon receiving an EOT byte using a function pointer. The source and destination were defined inside of the packet to create the desired communication connection. Each device listens and reacts to the packets that are addressed to them and ignores the others.

Debugging Techniques

Often we would run into issues that require further information about the system. After the board bringup, we ensured that UART worked with the XBee. This allowed us to wirelessly send messages to a base station. This is equivalent to using printf(). As we kept using this tool, we realized that a single print statement took up valuable runtime which affected our control loop. To combat this we then switched to only printing whenever it crashed versus continuously. In addition to this, digital pins were set high and low so that runtimes could be analyzed on an oscilloscope.

UAV: Controls

We originally intended on developing a full state space model of each UAV system in the pod, and applying a robust controller for flight stabilization, but because we were constantly in a rapid-prototyping phase throughout the project, we had to settle for a simple PID (Proportional Integral Derivative) control loop for flight stabilization. The feedback into our control system was provided by our inertial measurement unit and or GPS sensor. Our IMU was equipped with a gyroscope, accelerometer, and magnetometer which we read the values off of on an I2C bus where all three ICs lived. Our GPS sensor sent and received NMEA packets over our virtualized software USART.

Attitude Estimation

We used a PID loop for both position and rate of change of position to stabilize about an attitude and position. We had to estimate our attitude using all three integrated circuits on our IMU. The role of the IMU is to measure the position of the drone in 3D space. This was tested using three implementations using both the accelerometer and gyroscope. The first method simply used the accelerometer to compute the angles. An accelerometer measures the force of gravity on the drone. The problem with this is that every small force on the drone will disturb the measurement.

For the second implementation, we used a sensor fusion method to estimate roll and pitch from an accelerometer and gyroscope; a method known as the complementary filter. To do this, we first converted our gyro readings into degrees, and then we integrate the result to estimate a degree from the gyro's rate of change. We then take the 2-norm of the X, Y, Z converted gyro reading to get an estimated absolute magnitude, while taking the sign of the Z direction into account for directionality. From the accelerometer reading, we take the 2-norm on the X, Y, and Z readings from the accel IC, while taking the X and Y signs into consideration for directionality sake. We then convert that radial magnitude vector into a degree magnitude vector using a simple proportional conversion metric. Our output pitch and roll values in degrees is a linear combination of the two estimated gyro and accel calculations. Our yaw is estimated using a simple formula: $arctan\frac{magnetometer_Y}{magnetometer_X}\frac{180\ \mathrm{deg}}{\pi}$

Our last implementation took this a step further and implemented an explicit complementary filter that requires accelerometer, gyroscope and magnetometer outputs for a non-linear estimation based on quaternions.

Below is a table that indicates the approximate runtime of each method on our flight controller:

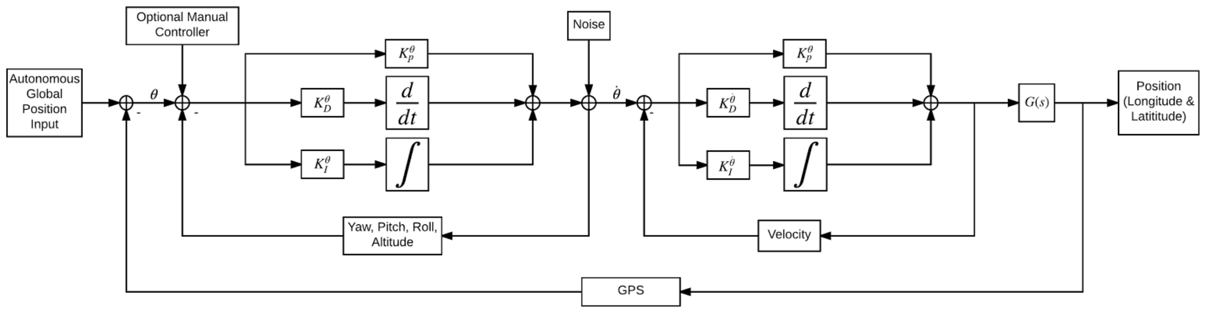
	Execution Time
Method 1: Simple accelerometer based angles	5.2ms
Method 2: Complimentary Filter	6.5ms
Method 3: Mahony Filter	12.8.ms

Flight Stabilization

Now that our attitude is estimated, we will use the values to stabilize our attitude about some reference position. To tune the gains for the this PID loop, we set the reference attitude to a yaw, pitch, and roll of 0 degrees, and measured the yaw, pitch, and roll step responses and biases to different inputs and tuned the parameters until the UAV was able to balance itself in the air. Once we were satisfied with the results, we had the output of those three PIDs go into a rate of change PID which had its loop closed around the gyroscope raw rate of change. We then tuned these parameters until the UAV was able to maintain its attitude and rough position in the air. This kept the UAV from drifting too much and as a result would alleviate instability from oscillations in the position PID controller. The final PID loop was close about these two PID loops and a GPS sensors with a simple proportional gain that would essentially help us bang-bang control along Earth coordinates. The desired attitudes into the controller are predetermined to have the aircraft pitch and roll forward and backward to move from coordinate to coordinate.

Because the controller differentiates and integrates, the time step Δt has to be close to exact in the constant code definition for the value. We had our control loops running off of set flags from an interrupt based timer. To measure the close to exact value, we set a digital GPIO pin high at the beginning of our attitude estimation and toggled it at the end of the control loop. We measured the pin toggling on an oscilloscope to measure the frequency of the attitude estimator (which involved an integral for the gyro), and the PID loops. We ran the control loops roughly at 250 Hz, which seemed to be a good enough frequency. We then updated the motors every iteration of the loop from the innermost PID (the “rate” PID) to move the aircraft to a desired position.

Below is a diagram explaining our final control scheme for the drone. As seen, the first PID loop takes in raw angles and is used as the desired set point for the rate PID’s. The measured rates, coming from the gyro, are compared to the desired to output the final error that will be used to actuate the motors. On top of all this is a simple GPS control loop that helps move the drone from position to position.


Optional Manual Controller
Noise
Autonomous Global Position Input
θ
$\dot{\theta}$
K_P^θ
K_D^θ
K_I^θ
$\frac{d}{dt}$
$\int$
$G(s)$
Position (Longitude & Latititude)
Yaw, Pitch, Roll, Altitude
Velocity
GPS

Conclusion

Our objective in this project was to develop a full and comprehensive software and hardware system that we expected to consist of fully autonomous flying UAVs, a ground station that would collect and process data from and send controls to the UAVs, and a refined easy-to-use interface that allows users to launch the system and manage it all while they're able to monitor the live feed of position and status data.

Unfortunately, as of June 2017, we were not able to completely accomplish our goal. However, we made significant progress towards that goal. We were able to manage a fully functional datapath from onboard sensors to the ground station to visualization and mapping in the user interface. We also established a solid network architecture and a full-featured communication protocol.

While we made huge accomplishments in developing our own custom flight controller, we still have some work to do, as the UAVs are still not stable enough to fly on their own. We expected this development to be challenging, but it was a rewarding endeavor in terms of the knowledge and expertise gained by pursuing the challenge. And while our user interface is functional and aesthetic, we can still push for more ease of use and functionality in conjunction with database and ground station operations.

Moving forward, this project will continue as we retain some members of the original project and invite new members to help finish, improve, and refine our current system thus far. Once we have accomplished decent flight stabilization we will begin field testing of the capabilities of the UAVs' flight systems, the behaviors of the onboard sensors during flight, and the limits of our network and effects on communication. We also want to develop learning capabilities for the system using machine learning and user training to "teach" our system what it should be interested in and what it should scan.

While the project is incomplete, we have achieved a robust and powerful framework for multi-AV systems. There is still work to be done on the UAV side and the framework would benefit heavily from continued improvement and refinement. Therefore, we are choosing to continue this project to accomplish our initial objective and to implement future functionality objectives as well.

Appendix A: Ground Station Programs

Database Initialization Script: db_initialize.py

The initialization program resets the database if it was already populated, and creates a new database with n amount of tables, specified by the number of drones in the configuration file. The tables are initialized with the 4 core columns (time, longitude, latitude, and altitude), along with a pair of raw sensor data and score columns for number of sensors specified by the config file.

MySQL Database Interface: dbi.py

The dbi.py file serves as a library of functions to communicate with the MySQL database. These functions include adding an entry to the database, fetch the most recently added entry, fetching all of the entries in the database, executing a specific MySQL command, closing the database to prohibit further writes, and writing all database entries to a log file. These functions use the database host and login information from the accessory files to connect to the database, and execute a specific command.

Serial Interface: dbz.py

The dbz.py file interfaces the database with the information transmitted from the XBee modules through ZigBee. Since there can be a variable number of sensors, the length of the packet is computed in order to determine how many bytes to read from the XBee receiver. The serial port to which the XBee module is connected is polled, and a string of the appropriate packet length is obtained. A general format is used for the data packets, and the string is then parsed accordingly, and saved into a database entry object. This object is then used to upload the data to the database server.

Struct Definitions: dbs.py

This file defines the sensor object used in saving the parsed packet into a structure, and then placed into the database as an entry. As mentioned earlier, the database entry struct consists of time, longitude, latitude, altitude, and n pairs of raw sensor data and derived score value, where n is the number of sensors per drone.

Aggregation Core: db_main.py

This is the main program that runs to implement the data architecture described previous. It samples at a user-specified rate and records a configurable average of those samples before comprehending and processing the data, then storing such data and results in the database.

Database Synchronization: db_sync.py

This contains functions used to continuously compare the local database on the ground station with the online cloud-hosted database. If there is any discrepancy in either database, the difference is noted, and the missing entries are queued to be written. This synchronization is bi-directional. That is, the local database can be ahead of the online one when drones are sending data packets to the ground station, and the online database can be more up to date than the local database when a user is interacting with the UI.

Writing to the Cloud: db_queue.py

This file contains the actual functionality for writing from one database to another. Whenever a database must be updated, the entries to be written are placed in a queue. The queued entries are written to the desired database until the queue is empty once again.

Accessory Files

Configuration File: db_conf

This is used as a configuration file for the database and all of its interfacing programs. This file specifies where the database is hosted and what user to sign in as. It also holds core variables to be used by the wrapper programs in adding data to the database. These include the number of drones, number of sensors per drone, number of data samples to record before making an entry in the database, and the sampling rate.

Credentials File: creds.py

The file creds.py holds private information for the database. Currently, it holds only the password to access the database.

Parsing helpers: a2d.py

This file contains miscellaneous functions used to assist in the conversion from ASCII values received from the XBee to be converted into the appropriate data type. Additional debugging helper functions related to parsing and storing the data values received are also here.

Appendix B: Manual Controller Wiring

<u>Controller -> arduino wiring:</u>
Vertical left potentiometer: pin A0
Horizontal left potentiometer: pin A1
Vertical right potentiometer: pin A2
Horizontal right potentiometer: pin A3
Left up button: pin 4
Right up button: pin 3

<u>XBee -> arduino wiring:</u>
Din: pin 1 (tx pin)
VCC: 3.3v pin
GND: gnd pin

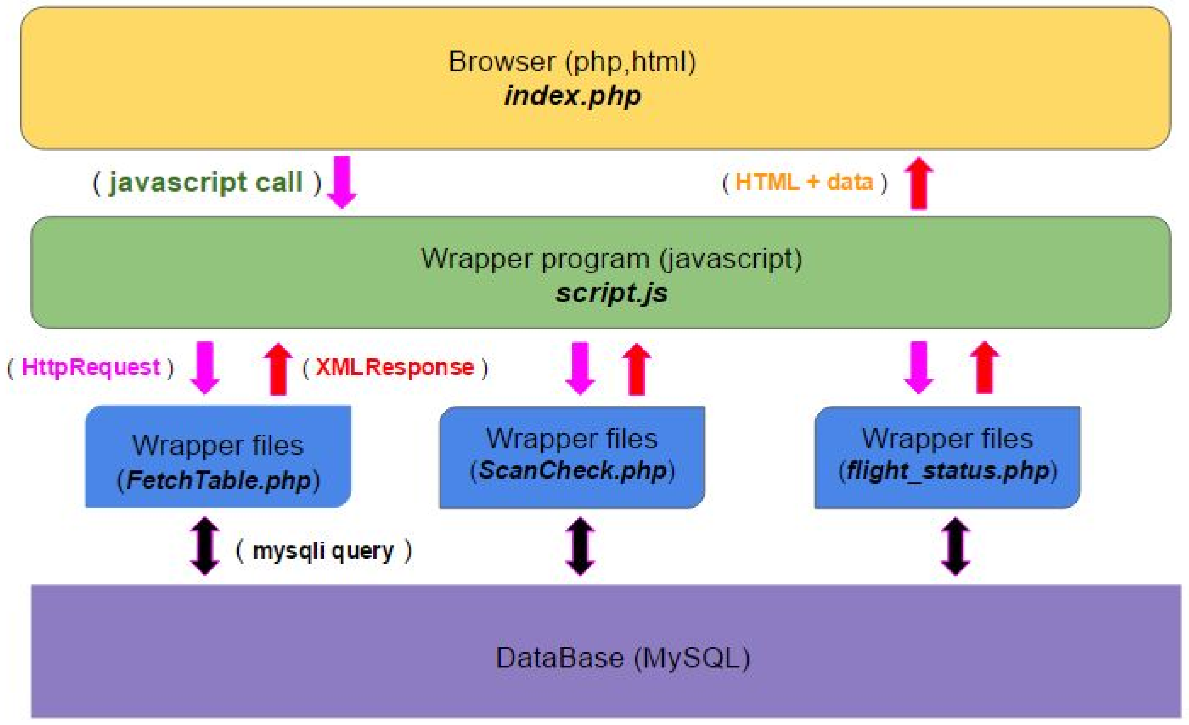
Appendix C: UI Functions

index_UI.php

Our UI is dynamic so it will create as many tabs that correspond to number of databases and tables that each one has.

Script.js

This file contains many wrapper and helper programs that are executed under different conditions depending on the user. We also have timer events being called when streaming starts and also when the weather conditions are monitored. The diagram below is a structure of our web UI and the layout of how it communicate with our server/ MySQL database.



FetchTable.php

The function ***showTable()*** in script.js will executed this file, passing the datatable and database variables, so the program knows where to connect to and what data to retrieve. This file will serve primarily to visualize our data.

weather_getInfo.php

The function ***LoadWeather()*** in script.js will execute this file passing a the city and state as input variables. The program will then determine which .json file to retrieve from the underground weather database. The gui representation for the weather forecast will mainly be in style.css but some html format will be contained in this file too. The response will update weather description at the top of the page.

HeatMap.php

This file is executed in script.js when ***hm_refresh()*** is called. The hm_refresh function reads the database with a specified threshold value. For example, if the call is hm_refresh(30), then the database is polled and returns all entries where the calculated points value is 30 or higher. Upon returning, an updated list of latitudes and longitudes are obtained to get a more current heatmap.

Visualization.php

Like the previous PHP files, this is executed when ***vis_getdata()*** is called to update the visualization. The last n most recent entries (usually 40 in our case) are pulled from the database and are used as the data points for the database.

Flight_status.php

The function ***check_flight()*** in script.js executes this file and verifies if drones are safe to fly, and then will continuously check every 10min when start(streaming) is selected and stops checking when the user stops streaming. This will just update the status box.

createMarkers.php

The function ***initMap()*** in script.js executes this file. This program will iterate through all of our drones(tables) and retrieve the latest timestamp from where we select the GPS coordinates and create XML nodes as markers that contain name,gps coordinates, and its point of interest. The markers are then parsed in the javascript, creating marker elements which are inserted onto the map. These are constantly being created whenever we stream for live feed on our drones.

ScanCheck.php

The function ***check_scan()*** in script.js executes this file. This program will do the computational math based on each drones battery levels to check if the area selected can be scanned. Any warnings will be updated on the status box which will display the coordinates inserted into the database, and the battery levels at computation time.

Appendix D: AT Commands Table

AT Command	Name	Description
+++	Enters Xbee into Command Mode	The XBee module confirms that it is in command mode by sending “OK” back on its Data Out pin.
ATDB <enter>	Signal Strength	Returns Signal strength of its communication with all XBee modules within range in dB.
ATND <enter>	Node Discover	Discovers and reports all RF modules found and calls ATMY, ATSH, ATSL, ATDB, ATNI
ATNI <enter>	Node Identifier	Returns XBee module name
ATNI “new name” <enter>	Change Node Identifier	Changes XBee module “new name”
ATID <enter>	PAN ID	Returns current PAN ID
ATFR <enter>	Soft Reset	Used when restarting the drone and base station.

Appendix E: Power Budget

Device	Power
Atmega 328p	16.32mA @ 5V = 81.6mW
GPS	10mA @ 3.3V = 33mW
GYro	6mA @ 3.3V = 19.8mW
Accel	0.145mA @ 3.3V = 0.5mW
Mag	0.1mA @ 3.3V = 0.33mW
IR	5mA @ 5V = 25mW

Xbee	300mA @ 3.3V = 990mW
Total	1150.2mW
Actual	0.154mW @ 11.1V = 1,709.4mW

The power budget is based on individual chips. This does not include the power consumed by the additional breakout boards or power lost due to thermal dissipation. Additional sources of power include various circuit elements such as LED's.

Of course the power consumption of the flight controller is trivial compare to the motors which draw the majority of the current. One motor typically goes from 20W to 65W depending on the throttle. This gave us a flight time of around 15 minutes given a 3 Cell Lipo battery.

Appendix F: Itemized Budget

ITEM	**QUANTITY**	**PRICE**
ATMEGA328P-PU DIP28	10	$24.20
XBee Pro 60mW Wire Antenna - Series 1 (802.15.4)	5	$207.75
Waveshare XBee USB Adapter USB Communication Board with Xbee Interface Supports XBee Connectivity	1	$12.99
Acrylic (Black and Clear)	2	$95.00
HiLetGo GY-NEO6MV2 Flight Controller NEO-6M GPS Module	1	$13.49
Teenitor 100 Pack SS-12D00G3 High Knob 3P 2 Position 1P2T SPDT Vertical Slide Switch 0.5 Amp	1	$5.99
Various Screws, Nuts	~	$10.00
ARRIS 30A OPTO Brushless ESC	4	$52.98
Zippy Compact LiPo Battery	1	$27.11
Usmile XT60 Power LiPo Distribution Board	1	$12.39
Andoer NEO-6M GPS Module	1	$17.98
WYHP 9DOF IMU GY-85	1	$11.99
Kinghard 1-8S LiPo Voltage Checker/Warning Buzzer	1	$3.51
iMAX B6-AC LiPo/NiMH 3S Battery Balance Charger	1	$29.99
XT60 Charge Cable	1	$4.46

Poster Printing From BELS	1	$35.00
Brushless Motors	4	$63.98
PCB: Development Board V2	1	$102.84
SainSmart Sensor Modules	2	$18.99
Xbee Sockets	2	$5.39
Jumper Wires	1	$7.99
Female Headers	1	$6.64
Buttons	1	$5.40
Uxcell Crystal Oscillators	1	$5.51
IC Chip Socket Adaptor	1	$5.67
Breakout boards for Xbee	1	$14.56
Pi and SD	1	$67.41
Omron Electronic Components D6T8L06 Sensor, Thermal, Mems	1	$61.15
ATMEGA328P-AU	1	$14.80
TI TXB0104	1	$20.36
PCB: Development Board V3	1	$30.51
10uF, 1uF, 0.1uF Caps	1	$20.26
TI TXB0104	1	$20.36
MPXHZ6400AC6T1 (Pressure Sensor)	1	$67.39
TPlug Connector	1	$7.99
Rubber Damper	1	$8.50
Rubber Gromet	1	$5.81
LM7805 Voltage Regulator	1	$7.99
Heat Sinks	1	$4.95
1/8" Black Acrylic	1	$10.50
PCB: Development Board V3.1	1	$30.51
Lipo Batteries	1	$58.48
Motors	1	$69.00
IMU X2	1	$15.49
GPS: Neo 6M	1	$13.49
Lock Nuts	1	$9.75
Propellers	1	$16.99
Xbee SOckets	1	$3.95
USBASP	1	$11.99

Omron Electronic Components D6T8L06 Sensor, Thermal, Mems	2	$164.00
IR Cables Omron	3	$38.97
Pi Casing	1	$10.99
Voltage Regulator (LD1117V33)	1	$8.28
Hobbymate Quadcopter Kit (Multirotor)	2	$116.00
Avatar RC Geniune Gemfan Propellers (pack of 2)	2	$17.98
AmazonBasics High-Speed HDMI Cable	1	$5.99
Xbee Pro 60 mW Wire Antenna Series 1	2	$78.90
Finware 10-pair ST60 Male-Female Bullet Connectors	1	$8.99
Total		$1829.53