IOT ATMOSPHERIC GAS SENSING AERIAL VEHICLE

A Project
Presented to the
Faculty of
California State Polytechnic University, Pomona

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science
In
Electrical Engineering

By
Vlad F. Cretu
2020
SIGNATURE PAGE

PROJECT: IOT ATMOSPHERIC GAS SENSING AERIAL VEHICLE

AUTHOR: Vlad F. Cretu

DATE SUBMITTED: Spring 2020

Department of Electrical and Computer Engineering

Dr. James Kang
Project Advisor

Dr. Salomon Oldak
Faculty Director of Graduate Studies,
Academic Programs

Dr. Thomas Ketseoglou
Graduate Program Coordinator
ACKNOWLEDGEMENTS
ABSTRACT

Unmanned Aerial vehicles have evolved drastically over the past decade from being used strictly for the military toward corporate and civilian uses. Some of these uses include surveillance, traffic monitoring, search and rescue, as well as agriculture and farming. This project will focus on building a fully functioning quadcopter which can be controlled with or without the user. The emphasis will be on flight stability, object avoidance and overall flight. The project includes a custom-built internet of things (IOT) transmitter/receiver, as well as a custom-built flight controller with PID compensation for drone stabilization and flight. The transmitter is user friendly with touch screen compatibility and can display telemetry and housekeeping data to the user. This would be the perfect project in terms of skill level, budget, and overall goal. The project also attempts to find how well an IOT control scheme works relative to the traditional RF system.
# TABLE OF CONTENTS

Signature Page ................................................................................................................... iii
Acknowledgments .............................................................................................................. iv
Abstract ........................................................................................................................... v
List of Tables ................................................................................................................... viii
List of Figures ................................................................................................................... ix
Chapter 1: Introduction ........................................................................................................1
Chapter 2: Basic Theory ......................................................................................................2
Chapter 3: Recent Progress in the Area ...............................................................................3
Chapter 4: Electrical Design and Results .............................................................................4
  Section 1: Microcontroller ............................................................................................... 4
  Section 2: Electrical Schematics ...................................................................................... 4
  Section 3: Motors ............................................................................................................. 6
  Section 4: IMU ................................................................................................................ 10
  Section 5: PID ................................................................................................................ 17
Chapter 5: Firmware and IOT Controller ............................................................................4
  Section 1: Web Server .................................................................................................... 21
  Section 2: Web Developer and Controller ..................................................................... 23
Chapter 6: Improvements and Future Work ......................................................................26
  Section 1: Improvements ................................................................................................ 26
  Section 2: Future Work .................................................................................................. 27
Chapter 7: Conclusion ........................................................................................................28
References .......................................................................................................................... 29
Appendix A ........................................................................................................................ 30
Appendix B ........................................................................................................................ 31
Appendix C ........................................................................................................................ 32
Appendix D ........................................................................................................................ 35
Appendix E ........................................................................................................................ 36
Appendix F ........................................................................................................................ 37
Appendix G ........................................................................................................................ 38
Appendix H ........................................................................................................................ 43
Appendix I ........................................................................................................................ 44
Appendix J ........................................................................................................................ 45
LIST OF TABLES
LIST OF FIGURES

Figure 1: Teensy 4.0 microcontroller ................................................................. 4
Figure 2: Electrical Schematics ................................................................. 6
Figure 3: Motor and ESC ................................................................. 7
Figure 4: Calibrated Motors I ................................................................. 8
Figure 5: Calibrated Motors II ................................................................. 9
Figure 6: Motor Speed Resolution ................................................................. 9
Figure 7: Motor Height Offset ................................................................. 10
Figure 8: IMU product selection ................................................................. 11
Figure 9: Euler Angles ................................................................. 13
Figure 10: Accelerometer Median Filter .............................................................. 15
Figure 11: Gyroscope and Accelerometer Theory ............................................................ 16
Figure 12: Adjusted Roll ................................................................. 16
Figure 13: PID Controller Theory ................................................................. 19
Figure 14: Web Server Overview ................................................................. 23
Figure 15: Drone Controller in Browser ................................................................. 25
CHAPTER 1: Introduction

An ongoing problem that looms our civilization is the ever-approaching climate change. Climate change already affects us and every year the world experiences a new heat record in terms of the warmest month, along with more intense weather patterns and irregular phenomena. The glaciers are melting, forests are on fire, and massive extinctions are occurring rapidly. These big changes require a big response. A step that must be taken is to identify the airborne chemicals that reside in our atmosphere. Besides the levels of CO2 gasses, there are numerous other hazardous gasses that are interesting to study. These are gases that people breathe into their lungs each day, pollute our water supplies, and can certainly damage public health over a longer period. The main scope of these drones would be to map out these aerial plumes at different elevations and gain an understanding of the actions are needed to minimize the pollutants that end up in our atmosphere. The pollutants sent into the atmosphere are not fully understood, as they are highly toxic. According to the National Fire Protection Association, while all types of hazardous spills are difficult to manage, airborne releases of toxic chemical agents in gaseous form, commonly used within industrial processes, are particularly challenging [1]. If a gaseous substance is spilled, these drones will assist with detection, area of spread, and increase response time to thoroughly clean the area and minimize environmental impact. The project accomplished the development of an IOT receiver/transmitter with a custom-built flight controller. A touchscreen display is used to control the roll and pitch of the drone to aid in positional movement and display to the user housekeeping and telemetry data. There are more improvements to be made which will be discussed in more detail in the Future Work section.
CHAPTER 2: Basic Theory

Due to the cost and allotted time, the focus of the project is to build a custom flight controller with fast processing and IOT capable transmitter and receiver. This is for several reasons. The drone will incorporate an IOT module, so the cost without the controller will be significantly less. It will also make it easier to send data to the user and display it on our touchscreen devices. The third reason for this change is that it is much more accessible to fly this drone since it is not bound by having a controller. Simply a touchscreen device that can connect to the internet and open an html file via Chrome browser will be able to access the drone and control it. There are also heavy drawbacks to this method as well. Of these, the most important is decreased flight range, and a lower data transfer rate. For this first prototype of this project, these are not as important if the drone achieves flight. The main components of this project will be the ESP-32 web module, the inertial measurement unit (IMU), the motors and electronic speed controllers (ESC). The IMU ESP-32 will allow the creation of an access point web server that mobile devices can connect to. The IMU will give us positional data for the drone. And will allow us to control the roll and pitch during flight. The motors will be controlled by the ESC’s, or electronic speed controllers. These will ensure that the motors will spin in the correct direction with the correct amount of RPM. A lot of software will have to be written to control all these elements and to incorporate a PID controller for flight stabilization. The project and parts will be explained in more detail in chapters 4 and 5.
CHAPTER 3: Recent Progress in Area of Interest

This very project has recently been the topic of heavy research. As of many years there are numerous companies and research projects dedicated to studying our changing climate. Drones have been proven to be very reliant mode of plume tracking. A lot of emerging technologies assist in aiding with forest fires. They can detect small fires before they become a threat and alert first responders to contain and extinguish it early. This drone gas sensing technology has been studied and prototyped many times before. As early as 2014, in [2], a quadcopter was armed with sensors to find and monitor hazardous gasses in the atmosphere. 2016 also had an abundance of new emerging technology in the area such as the use of a drone to monitor and track hazardous clouds. The data would be sent to a base computer for later analysis [3]. In that same year, [4], a drone was used to monitor gas concentrations in underground pipes and caves, while streaming the data to a mobile viewer. This provided data in real time. Most recently however, and most like the work done in this paper was in 2018. Unlike previous work, [5] integrated an IOT module for better positional tracking of gaseous plumes. IOT also allowed for the synchronization of multiple drones for better coverage and data accuracy. IOT gave the possibility of this drone swarm, which allowed for “Drones to prove a visual reference point for plume movement that enables on-scene plume proximity and trajectory over time”. Climate study and atmospheric gasses will continue to be studied with more emerging technology and new science data from current analysis.
CHAPTER 4: Electrical Design and Results

Section 1: Microcontroller

For this type of project, the selected microcontroller is a Teensy 4.0 by PJRC. The Teensy houses an ARM-Cortex M7 which offers serious processing power for its listing price. The Teensy stands at 35.6mm by 17.8mm; which is the same size as the Arduino Nano; It is packaged small and delivers 600MHz clock speed, 3 SPI busses, 3 I2C busses, and 32 GPIO pins with 31 being PWM-able outputs. The microcontroller was chosen due to these specifications, so that it can handle any task that is needed without breaking the user having to optimize code to meet performance requirements. This microcontroller can be exchanged with almost any other with lower performance without running into any performance issues. This paper will highlight how the second microcontroller used, and ESP-32, can also take over the entirety of the functions by itself, thereby the Teensy is not needed. For the purposes of this project however, the Teensy remains and takes care of anything that is thrown it.

Figure 1: Teensy 4.0 microcontroller

Section 2: Electrical Schematics

The electrical components used are essential to this project. They are represented in the
electrical schematics diagram in Figure 2. We have a barrel jack input which we use to power the entire system. A 3 or 4 cell LI-PO battery to power the system, but for the purposes of this project a 60W power cable was used to test out and optimize the electrical components and the drone stabilization. At the input of our power, we have a current sensor which measures the entire current drawn by the electrical components including the motors and ESC’s. These components will be explained in more detail in a future section. The input voltage is stepped down from 15V or 12V to 5V using a PTN78000WAZ. This 5V is to power the electronics and the microcontroller. As mentioned above, the microcontroller used is a Teensy 4.0. This is connected to the secondary microcontroller ESP-32, whose sole purpose is to host an access point web server. The ESP and Teensy communicate through UART, sending data from the Server to the main microcontroller. The IMU used is an ICM 20602 from TDK InvenSense. It is coupled to the teensy through an SPI bus. Four dedicated PWM outputs are used to connect to the ESC’s which control each motor. This will also be explained in a later section. The schematics can be seen below.
Section 3: Motors

Brushless DC motors are preferred due to their high performance and durability over time. The motors used are high performance, light, brushless motors at a low price point. The motors are by DYS Samguk 2206 series, capable of delivering 2700kV. Brushless motors work by having a magnetic rotor. This rotor spins as the coils around the stator are energized producing a magnetic field. This magnetic field in turn spins the rotor around. Thus, changing the order of coils that are energized and the time they are energized is based on the DC potential applied. This is where the ESC’s or electronic speed controllers, come into play. ESC’s energize the coils consecutively in a desired direction thus making the motor spin. The faster the ESC shifts the applied voltage, the faster the motor will spin. Figure 3 shows how the entire process works. In the code, a PWM signal of low frequency is generated, which is then sent to the ESC to be processed.
to the opening and closing of its internal transistors to spin the motor. The ESC’s used are BL-Heli32. They are chosen due to being cheap and very robust. A good ESC will have a failsafe and is able to protect the motor from harm. Figure 3 shows how our microcontroller produces a PWM signal between 1000 to 2000 microseconds to spin the motor.

The downside of these microcontrollers and ESC’s is that the RPM of each motor differs slightly from each other. As such, there needs to be a method of calibration to match the speed of each motor to each other. An easy way to calibrate these motors to work in sync is to use a microphone and record the period that the motor takes for one full rotation. To gather this data, a simple piece of tape was adjusted to the motor shaft. This way, whenever the tape hits the microphone, a high amplitude in sound indicates a full revolution. The noise capture software used was audacity. Audacity also allows to choose exactly the window we want to focus on and then display the time for that window. This way multiple periods can be averaged for a more accurate representation of motor speed. Finally, the number in the window can be divided into 60 to get the speed in rotations per minute. This RPM value will be the main way we set the speed of the motor. The data gathered from these motors can be plotted in Microsoft excel, which we can use to generate a linear trendline to get a conversion from microseconds to RPM. Figure 4

Figure 3: Motor and ESC

The downsides of these microcontrollers and ESC’s is that the RPM of each motor differs slightly from each other. As such, there needs to be a method of calibration to match the speed of each motor to each other. An easy way to calibrate these motors to work in sync is to use a microphone and record the period that the motor takes for one full rotation. To gather this data, a simple piece of tape was adjusted to the motor shaft. This way, whenever the tape hits the microphone, a high amplitude in sound indicates a full revolution. The noise capture software used was audacity. Audacity also allows to choose exactly the window we want to focus on and then display the time for that window. This way multiple periods can be averaged for a more accurate representation of motor speed. Finally, the number in the window can be divided into 60 to get the speed in rotations per minute. This RPM value will be the main way we set the speed of the motor. The data gathered from these motors can be plotted in Microsoft excel, which we can use to generate a linear trendline to get a conversion from microseconds to RPM. Figure 4
shows how each motor has been calibrated, the linear equation it generated, and the coefficient of determination to show how accurate our linear equation is to the captured data.

Figure 4: Calibrated Motors I
Figure 5: Calibrated Motors II

Figure 5 shows the calibrated motors stacked on top of each other. This highlights just how different ESC #3 is compared to the others. The generated trendline equation brings the different motors to the same slope, however, there is loss of resolution in motor 3 since it is very different from the others. This could cause problems moving forward due to resolution change as the PID sensitivity would need to be higher. More on this will be discussed in an upcoming section. Figure 6 shows the resolution of each motor. For a precision application such as a drone, flight stability requires that the motors spin at the same speed. That is the reason for calibrating the motors using this process.

<table>
<thead>
<tr>
<th>ESC/Motor Resolutions (RPM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
</tr>
<tr>
<td>29</td>
</tr>
</tbody>
</table>

Figure 6: Motor Speed Resolution

Due to lack of breadboarding space, the components stacked up too high in the corners. This stopped some of the propellers spinning since they would hit exactly where some of
the components were. To fix this issue and avoid making a new electronics board, a 3-D printer was used to print custom designed motor offsets to lift the propellers and clear the lower electronics space. The motor offsets provide a 6 mm offset and were designed in SolidWorks. The 3-D printer used to print them was a Prusa i3. Which allows for greater print resolution. The material used was PLA plastic. Figure 7 shows how the offset SolidWorks design.

Figure 7: Motor Height Offset

Section 4: IMU

The most essential part of a drone is the inertial measurement unit (IMU). The IMU stabilizes the drone and gives feedback to control the drone movement. The IMU used is a new micro-Electromechanical System (MEMS) designed by TDK InvenSense, ICM20602. The ICM20602 houses a 6 degree of freedom, 3 axis accelerometer and gyroscope. Figure 8 compares the different products performance including the well
known and renowned IMU’s MPU6000, MPU6500, and MPU6050. The attraction toward picking out the ICM20602 is compelling, as its specs show that it has a much lower accelerometer and gyroscope noise than the three aforementioned IMU’s. On top of this, the price point for the ICM20602 is much lower than the others. In direct comparison to the MPU6050 and similarly to the MPU6000/MPU6500, the ICM20602 provides faster sampling rates through SPI communication, with speeds up to 10Mhz. This is much higher than the 400kHz maximum sampling of the MPU6050. The reason for higher sampling is due to higher averages and lower noise. The signal can be collected much more often and averaged, thus resulting in a much better signal that the processor can use.

Figure 8: IMU product selection
The IMU used communicates over I2C or SPI. To use SPI mode, we connect the address pin as our chip select. The settings register must be loaded with the correct byte of information to keep the IMU from reverting back into I2C mode. The resolution settings for the accelerometer and gyroscope must first be set before reading, in the accelerometer and gyroscope setting registers, respectively. Our goal is to read the raw accelerometer and gyroscope values in the X, Y, and Z axis. This returns 2 bytes of information that needs to be concatenated to form a complete data set. A total of 12 registers must be read to get positional information from the IMU. This raw data must be converted into information based on roll, pitch, and, yaw. Roll is the rotation across the X-axis of the drone, while pitch and yaw are rotations on the Y and Z axis, respectively. To achieve this information, the accelerometer must make use of Euler Angles. According to the article “Autocalibration of MEMS accelerometer” [7], figure 9 shows the Euler angle formulas and the depiction of vector diagram. According to [6], the accelerometer measures the projections due to gravity. These projections are utilized to compute the angular position. This is described by the Euler angles. The vectors dictate how the gravity affects the position angle that is desired. The gyroscope on the other hand, measures rotational velocity or rate of change of the angular position over time, along the X, Y and Z axis. To get angular positioning of the gyroscope, the angular velocity must be integrated over time. The error in integration causes the gyroscope to drift over time, so both pieces of information must be used (gyroscope and accelerometer) for stable output.
Figure 2: Orientation of the MEMS accelerometer expressed both with the orientation angles ($\phi$ and $\rho$) and with the Euler angles ($\alpha, \beta, \gamma$). Because the value of $\alpha$ is irrelevant, it is arbitrarily set to 0; consequently, the nodes’ axis $N$ coincides with the $X$-axis.

$$\varphi = \arctan \left( \frac{a_x}{\sqrt{a_y^2 + a_z^2}} \right), \quad \rho = \arctan \left( \frac{a_y}{\sqrt{a_x^2 + a_z^2}} \right).$$

Figure 9: Euler Angles

The IMU that was used performed much more poorly than expected. The gyroscope suffers from large drift, which is about 5 degrees of drift per minute, after optimization and filtering. This could be negligible for short term flight, however the true problem of is much worse. The main problems is the sudden offset of the gyroscope when more abrupt drone movement takes place. The offset could be small ~5 degrees, or large ~25 degrees offset from our 0 degrees starting point. On the other end, the accelerometer does not seem to fare any better. Even at the lowest resolution, the noise levels on the accelerometer is too high with a +/- 40-degree oscillation of the zero-degree axis. As shown in the product selection in figure 8, the ICM20602 has a much higher sensitivity to
noise than the other modes. This is what makes the ICM20602 a poor part selection despite the lower noise and faster sampling. The negatives far outweigh the positives in this case. Due to these problems, the standard Kalman filter or complementary filter do not work. The Kalman Filter and Complementary filter are very similar, in which they both take inputs of two data sets and combine them to get a better output. This does not work for this application since both inputs are flawed. Therefore, a new system must be put in place. This is where the high sampling speed becomes a positive advantage to compromise for the poor performance. The accelerometer is averaged every millisecond, for approximately 300 samples. Then these samples are stored in 133-255 values of the averages into an array. The array is then passed through a bubble sort which selects the median value and stores in into a variable. This median filter is essential to bring down the +- 40 degrees of noise that is generated by the spin of the motors. Figure 10 shows the comparison between filtered and unfiltered accelerometer data. Although much better than before, the drawback of this filter is the very slow response time of only 4-7.5Hz. Thus, it is somewhat accurate, but very slow to react. The gyroscope is very much still needed to compensate for this.
The gyroscope must also go through some filtering or compensation. To avoid drift, or even worse, a shift in offset of the gyroscope; the accelerometer must be used. Figure 11 depicts how the gyroscope and accelerometer are used. The gyroscope is sampled as often as possible, and every new accelerometer reading will be used to subtract the gyroscope, ensuring that there is no drift or offset.
The theory is demonstrated in the results in Figure 12. The figure tracks the values of the gyroscope, accelerometer, error, and roll. Roll is the value that is most important since it is the rotation about the X-axis.

Figure 11: Gyroscope and Accelerometer Theory

Figure 12: Adjusted Roll

The figure shows how the gyroscope is smooth, but at the end of the figure, it has suffered an offset that pushed it to -15 degrees. The accelerometer is very low resolution, but it always has the correct value. While the roll combines the gyroscope data and the subtracted error between accelerometer and gyroscope. The figure also shows only one
axis that is used, roll. The other that is essential is the pitch, which works the same as roll with very similar results. The roll and pitch are both used; however, the yaw is omitted from this design. According to [7], since the angle $\alpha$ expresses a rotation around the absolute vertical axis, it cannot be measured by MEMS. It remains undetermined, and it can be set equal to zero. Thereby is no accelerometer output that can be used to control the direction in the Z-axis, and the gyroscope alone is too error prone to reliably measure the yaw change. It is better to omit the yaw than to try to compensate for it.

Section 5: PID

A PID controller was implemented to stabilize the drone and achieve flight. The PID was implemented using the common Ziegler-Nichols method. The proportional gain, integral, and derivative are all used in order to achieve the best stability possible. The poor performance of the IMU is detrimental of what could have been a stable drone without any oscillation. The poor feedback outweighs how well the PID works and the best results have been when the drone PID is purposely programmed to oscillate. As mentioned previously, the drone needs two PID controllers, one for the roll and one for the pitch. As such the Ziegler method needs to be implemented in two instances. The controllers P, I, and D components however are going to be the same for both directions. The P controller is the first and most important variable that needs to be set. The P component focuses on the instantaneous error at a point in time. The P constant is multiplied by the error difference between the user set point and the drone IMU reading. This is in degrees. The higher the P value, the more aggressive the drone response becomes. Setting the P value too high causes the drone to oscillate. According to the Ziegler-Nichols theory of PID compensation, the P value must be first chosen to be as close to oscillation as possible.
The integral constant represents the sum of all errors in a direction and is representative of how the error changes over time. The I variable is indicative of how the error changes over time. The integral component complements the P component and dampens the drone response and lowers the overshoot. It also makes the PID controller a 2\textsuperscript{nd} order system, which means that the error should be zero when stable. The third and final component is not necessary for most controllers, but for this situation it is very useful. The D component is a response to how fast the error changes, and it is helpful to dampen sudden positional drone changes and keep the drone upright. For this situation, since the IMU cannot be trusted completely, the Derivative gain allows the drone to mitigate any sudden large movements in of the roll and pitch. These values are then added together and subtracted from the total throttle. The throttle of the system is the offset from 0. It determines the altitude of the drone. The roll and pitch compensation are added or subtracted from this throttle offset, depending on the direction that the drone needs to be adjusted toward. The PID compensation process is very arduous especially when stabilizing a drone, where the variables need to be perfectly tuned for stability. On top of this, the IMU output error vastly makes this task even more complex, as the positional feedback cannot be reliably trusted. Figure 12 shows a standard PID controller block diagram. The complexity of this system arises from the fact that a quadcopter is “a MIMO system, but in this work it is considered as a combination of SISO systems, in order to divide the problem for each movement (yaw, pitch, and roll). As mentioned, these division was tested individually.
Due to the current sensor that is used in the schematics, it was easy to estimate how much power the drone would use. The AC/DC converter used to power the drone allows only 5A of current at 12 volts for a total of 60W. This limits how much throttle we can apply to the motors and how much height the drone can achieve. Exceeding 5A would turn off the power supply due to the supply built in failsafe. Using the data from the current monitor, the results are plotted below in figure 13. The data can be linearized using excel and the total power needed for around 19000 RPM is calculated. This would mean that it is necessary to have a power supply that is able to output 9.2A or 111W at 12-volt input. Thus, a 3cell Li-Po battery would be enough to power the drone. Using the battery calculator on forest-blue multi-copter website, allowed comparison between various modifications in batteries. The most important parameters for this project were battery price, and battery flight time. With the 10A current requirement, the OVONIC 3s, 2200 mAh battery would allow a total of 21 minutes of flight with 85 cents per total flight time. This seemed like the best deal. Unfortunately, after weeks or so of using the battery, the main Li-Po cell failed reducing it to a 2S battery with 0 output. This rendered the battery useless and the need to shift the drone back to AC/DC converter was necessary to
continue the PID Calibration.
CHAPTER 5: Firmware and IOT Controller

Section 1: Web Server

As mentioned previously in the microcontroller selection section, the IDE used for writing the firmware of this project is the Arduino IDE. The libraries for the electrical components are written specifically for this project and contained in their individual drivers. The libraries for SPI, AP web server, and UART are standard built in Arduino libraries that are used. The code follows very specific pattern of creating classes and functions that interact between each other and send data back and forth. The two microcontrollers, the Teensy 4.0, and the ESP-32 both use Arduino data. The ESP-32 contains AP web server code and parser of data, as well as a UART transmitter that sends the data from the user controller to the main microprocessor, the Teensy 4.0. The data sent to and from the web server will be covered in a future version. The Teensy program has several different functions. Namely the UART receiver, which can also work as a transmitter, the IMU data, the motor and ESC controllers, and the PID controller which ties everything together. The receiver communicates with the ESP-32 and sends data back and forth between the microcontrollers. The motors driver handles all communication with the ESC’s, such as setting the motor speed and initializing the motors. The IMU driver communicates with the ICM20602 chip through SPI, obtaining data from the gyroscope and accelerometer. It also includes the bubble sort algorithm with a median filter and averaging of the samples. The goal of the IMU driver is to generate the pitch and roll of the system. The Controller driver connects all of these pieces together and makes decisions on the motor speed, throttle, and direction based on the information from each sensor and the user input. The controller driver also
implements a PID controller to stabilize the drone. All these components work together, and the Controller driver is the brain of the operation and makes all the decisions. This briefly sums up how the firmware architecture is organized and how everything is linked together. The code itself can be viewed in the Appendix section of this document.

As mentioned previously, and the communication with the drone is through a web server. Figure 14 shows the overview of how the web server works and the data flow. The Teensy collects data from the drone and communicates with the ESP-32. The ESP-32 communicates back to the Teensy, as well as hosts a web server set up as an access point. The user can connect to the access point the same as through a Wi-Fi router, with a specific network name and password. According to [9], a web site can be hosted by a web server and provides request services for clients. This however may use a vast amount of memory. To circumvent the memory challenge from loading onto our ESP32, we download the front-end data on the device we use to control the server. Once connected, the user can download an app which supports the web server on their phone or open the link that is stored on their device with google chrome. The functionality of the google chrome is done with additional languages. This is explained in more detail in the next section.
Section 2: Web Developer and Controller

The front end of the user controller is designed using a different program. For this, the use of sublime text editor is essential. The sublime text editor allows for text manipulation based on the specific language. It can be used for C++, python, as well as the languages that this program is based on, HTML, CSS, and JavaScript. All these languages combined are essential for the controller to work. The three mentioned languages are used by web designers to create the web pages which are visited by millions of times a day. For this project, the same architecture will be used but at a much more basic level. To understand how the three languages, interact with each other, the analogy of a car is a useful tool for understanding. The HTML serves as the frame of the car. Once it is completed, the skeleton underneath is unquestionable that it is that of a car, however it does not look pleasing, and offers no functionality. This is the bare bones or
foundation of the car. However, the CSS language adds the look of the car. After this is applied, the car looks exactly like it would be able to be driven off the lot, without the ability to do so. The engine, or the overall functionality of the car is found in the JavaScript component. This is the engine of the car. When all three elements are combined or linked in the code, it creates a functioning webpage that looks pleasant as well. Figure 15 shows the finished browser controller. It uses JavaScript elements that combined to make the joysticks. The browser looks empty when it loads, but it is half split for two joysticks. This is the data sent to the access point server and processed by the ESP-32. When the user touches the screen, throttle, pitch, and roll can be accessed through the left and right joysticks. The joysticks are also analog, meaning they give proportional change based on how far the joystick is dragged. The data processing and the get requests of the html page are done using the AJAX protocol. AJAX is a technique for accessing a web server through a web page. It allows for faster communication due to data sending without refreshing of the webpage. According to [10], the asynchronous protocol of Ajax introduces parallelism between client and server. An ajax page can change in response to user interactions or asynchronous messages received from the server. This is exactly how the communication scheme was implemented.
Figure 15: Drone Controller in Browser
CHAPTER 6: Improvements and Future Work

Section 1: Improvements

The project did not meet the entire goal that it set out; however, it is on the correct path to move forward. There are many improvements that should be made, yet more time and budget is needed. The most crucial component that should be improved upon is the IMU. The current IMU is too sensitive to the vibration of the motors, thus affecting drone stability and flight. The best improvement in this area is to look toward swapping out with a better 6-axis motion tracking device. Given how difficult of a problem that mechanical noise due to vibration causes in IMU sensor reading, it would be a great idea to add an additional 3-axis of positional detection. Mainly this would be in the form of a magnetometer. This is extremely useful for any error correction as a redundancy check over the gyroscope and accelerometer. The magnetometer can be part of a 9-axis chip, or a standalone magnetometer sensor. Fixing the IMU is the number one problem that should fixed for. Additionally, a new electronic speed controller would be a welcome addition for increasing the resolution of the third motor. This would also help greatly improve drone stability by PID compensation. By increasing the step size of the PWM duty cycle, drone maneuvering allows for more precise modifications toward roll and pitch to keep drone to a higher stabilization by dampening the percent overshoot. After making these changes, the PID will need to be tuned again to achieve better flight performance. A new battery will need to be purchased, since the main cell of the LI-PO battery has been damaged. Moving the conversation toward the web server side, additional telemtry could be added such as motor speed and a power button to allow just to start. As more sensors are integrated, which will be talked about in the future work
section, these too should also be displayed for the user. Battery level and total flight time can also be displayed with a functional battery and the on-board current sensor. These are the current improvements that should be made to the drone with more time and more budget. The web server telemetry display and battery housekeeping are nice-to-haves, the must-have improvements are to integrate a new IMU and implement a PID controller.

Section 2: Future work

The overall scope of the project has not been achieved. The atmospheric gas sensors have not been integrated into the system. These are the next step after the improvements. The BMP-388 ambient pressure sensor must first be integrated. This will allow for the user to achieve more positional data for the drone and be able to implement an altitude maintaining feature. If the goal of the project is to create a 3-D map of gasses in the atmosphere, then this ambient pressure sensor is very important. The next steps after the pressure sensor is to add gas detection sensors. These should be able to detect, to a various degree of certainty, the type of gases in a particular area, as well as the intensity of the gasses. Fixing the improvements section, integrating these two pieces of hardware with the system, and running tests to confirm that everything is working, would be sufficient to say the scope of what this project set out to do to be finished. However, for additional information, if this device is used to gather data and to truly run analysis on climate change and aerial plume leaking first response situations, the improvements that can be made are endless. A GPS can be added, Camera, and other types of telemetry to aid in classifying atmospheric gasses.
CHAPTER 7: Conclusion

The project set out to build an internet of things (IOT) aerial flight controller, which would detect atmospheric gasses. The project did not fully accomplish all that it set out to do in the current time frame. There are some parts which failed and other parts which are simply underperforming for what the project requires. The improvements section in chapter 6 describes in detail what needs to be repurchased, such as the battery, and what needs to be replaced. The current IMU greatly underperforms when exposed to the vibration of the motors. What has been achieved is a working IOT receiver/transmitter using the ESP-32. Using a touchscreen device with WI-FI capability, the user can set the roll and pitch of the drone to control the positional movement. Housekeeping data, such as motor speed and battery information can be sent back to the device and displayed on screen for the user. By looking at the future work section, it is shown that this project has a long way to go to meet initial set goals.
REFERENCES


#include "UART.h"
#include "WifiServer.h"
// the board is the nodeMCU_32s

void setup() {
    Serial.begin(115200);
    while(!$Serial);
    Serial.println();
    Serial.println("Configuring access point...");
    wifi.initWIFI();
    uart.init_UART();
}

void loop() {
    wifi.updateWIFI();
    //uart.send_UART("hello2! "); // for debugging
}
APPENDIX B: ESP-32 Ajax.h

/*
 * Ajax Command Parser,
 * this program takes the ethernet input and analyzes it in communicating with the teensy.
 */

#ifndef Ajax_H
#define Ajax_H

#include "Arduino.h"
#include "WifiServer.h"
#include <WiFiClient.h>
#include <WiFi.h>
class Ajax
{
  public:
    //constructor
    Ajax();

    void setClient(WiFiClient); // Set the Ethernet Client
    void readHTTP(String); // Read the HTTP Request
    void sendIDs( void ); // Send the ID List
    void sendValues( void ); // Send Value List
    void parseCommand(String); // Parse Command
    void groupCommand(String, String); // Sends Command to appropriate Group
    void updateGauges(void);

    WiFiClient cl;

    String Command = "";
    String Value = "";

  private:
    //elapsedMillis WebServerTimer;
};

extern Ajax ajax;

#endif
APPENDIX C: ESP-32 Ajax.cpp

#include "Ajax.h"
#include "Arduino.h"
#include <WiFiClient.h>
#include <WiFi.h>

/* Create External Object */
Ajax ajax;

const bool L_VALVE_OUTPUT = false;    // set to false for less verbose output
const bool DATA_MONITOR_OUTPUT = false;   // set to false for less verbose output
const bool POWER_SWITCH_OUTPUT = false;  // set to false for less verbose output
const bool HVPS_OUTPUT = false; // set to false for less verbose output

/* Constructor */
Ajax::Ajax(){}

void Ajax::setClient(WiFiClient CL){
  cl = CL;
  Serial.println("Client has been set");
}

void Ajax::readHTTP(String HTTP){
  if (HTTP.indexOf("elementIDs") > 0) {
    sendIDs();                  // First we pass the ID's to the Ajax
    Serial.println(HTTP);
    cl.println("id1,id2,id3,  
               1,2,3,4,  
               a,b,c,d,e,f,g");
  }
  if (HTTP.indexOf("elementValues.ajax") > 0){
    sendValues();                // The Values will be updated every second unless
                                  // specifically called
  }
  if (HTTP.indexOf("cmd") > 0){
    parseCommand(HTTP);         // This is where we look for commands nad parse
                                 // them
  }
}

void Ajax::sendIDs(){

  // Concatonate ID's for different Groups
// String toSend_IDs = 
LatchingValves_IDs+Reporting_IDs+PowerSwitches_IDs+HVPSinterface_IDs+TECinterface_IDs+Opticalinterface_IDs;
String toSend_IDs = "This_is_an_ID,id2,"; // change this for optics
cl.println("Id1,id2,");
}

void Ajax::sendValues()
{
    //update the gauges;
    updateGauges();

    String toSend_Vals = "print ID list";

    cl.println(5); // send the list to the client
    //Serial.println(toSend_Vals);
}

void Ajax::parseCommand(String data)
{
    // Find the positions of delimeters
    int beginning_of_command = (data.indexOf("="));
    int end_of_command = (data.indexOf("_"));
    int end_of_value = (data.indexOf("&"));

    // Empty the previous Command and Value Strings
    Command = "";
    Value = "";

    // Concatinate to get the string of command
    for(int i = beginning_of_command+1; i < end_of_command; i++){
        Command += data[i];
    }

    // Concatinate to get the string of value
    for(int j = end_of_command+1; j < end_of_value; j ++){
        Value += data[j];
    }

    // uncomment to display the command through Serial
    //Serial.println(Command);
    //Serial.println(Value);

    // Send the newly parsed data to be grouped by type
    groupCommand(Command, Value);
}
void Ajax::groupCommand(String Command, String Value) {
    // uncomment to see the command being sent.
    // Serial.println("command = " + Command + " Value = " + Value);

    // Send information to appropriate group
    if (Command.indexOf("TECinput") > -1) {
        // TECinterface(Command, Value);
    }
    else {
        // Convert the value from string to number;
        int Val = Value.toInt();

        // match the command to its appropriate subgroup. Send to that specific function.
        if (Command.indexOf("Latching") > -1) {
            // LatchingValves(Command, Val); // Send to Latching Valves
        }
        else {
            Serial.println("Command NOT Found :( ");
        }
    }
}

void Ajax::updateGauges(void) {
    return;
}
APPENDIX D: ESP-32 UART.h

/*
UART Communication between ESP32 and teensy 4.0
RX AND TX should be connected to the GPIO16 and GPIO17 respectively
*/

#ifndef UART_H
#define UART_H

#include "Arduino.h"

#define RX 16
#define TX 17
#define UART_BAUD 57600

class UART{
public:
    UART(); // Constructor
    void init_UART(void); // initialize the UART Serial Transfer
    void send_UART(String); // sends the data to through UART

private:
   
};

extern UART uart;

#endif
#include "UART.h"
#include <HardwareSerial.h>

/* External Object */
UART uart;
HardwareSerial mySerial(1);

/* Constructor */
UART::UART(){

void UART::init_UART(){
  mySerial.begin(UART_BAUD, SERIAL_8N1, RX, TX);
}

void UART::send_UART(String _toSend){
  mySerial.print(_toSend);        // check why "print" works, but "write" does not
}
APPENDIX F: ESP-32 WifiServer.h

#ifndef WifiServer_H
#define WifiServer_H

#include "Arduino.h"
#include <WiFi.h>
#include <WiFiClient.h>

class WIFI{
public:
    WIFI();                         // Constructor
    void initWIFI(void);            // Initialize the Web server.
    void updateWIFI(void);  // Run the web server continously. Called in main loop.

    void ajaxGetRequest(String);  // function that deals with the ajax get request for data

private:
    String HTTP_req;  // stores the HTTP request
    String requestURL;  // URL of pending get request; blank if nothing pending
    uint32_t currentIP;  // Hold the values of the IP address
    void parseCommand(String); // here we parse the incoming commands.
    void groupCommand(String,String); // group the commands, extension of the parser.
    void direction(String,float);  // the command that handles the direction of the drone.
    int WebServerTimer;             // timer which keeps track of connection timeout
    WiFiClient client;

    String list_ID = "dx,";
    String list_Val = String(dx)+"",";

    String Command = "";
    String Value = "";

    String dx,dy,TH; // holds the values of the input from the GUI. (roll, pitch, throttle)
    String testString; // temporary for debugging
    int start1, end1, end2, end3;

};
extern WIFI wifi;

#endif
APPENDIX G: ESP-32 WifiServer.cpp

#include "Ajax.h"
#include "UART.h"
#include "WifiServer.h"
#include <WiFi.h>
#include <WiFiClient.h>
#include <WiFiAP.h>

/* External Object */
WIFI wifi;

/* Debugging */
const bool VERBOSE_WEB_OUTPUT = false; // set to false for less verbose output

/* Defined Constants */
#define LED_BUILTIN 2  // Set the GPIO pin where you connected your test LED or
comment this line out if your dev board has a built-in LED

// Set these to your desired credentials.
const char *ssid = "Drone-Fi";
const char *password = "password";

WiFiServer server(80);
IPAddress ip{169,254,125,60};  // Change ip address here, must be the same as in the
JAVASCRIPT file

/* Constructors */
WIFI::WIFI()

void WIFI::initWIFI()

  WiFi.mode(WIFI_AP);
  WiFi.softAP(ssid, password);
  Serial.println("Wait 100 ms for AP_START...");
  delay(100);
  Serial.println("Set softAPConfig");
  IPAddress Ip(192, 168, 1, 1);
  IPAddress NMask(255, 255, 255, 0);
  WiFi.softAPConfig(ip, ip, NMask);
  IPAddress myIP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(myIP);
  Serial.println(ip);
server.begin();
Serial.println("Server Started");
}

void WIFI::updateWIFI(){
    // Check if our module is connected to the correct IP address. If not => reconnect
    // This code is very important to maintain connection with instrument
    //---------------------------------------------
    // currentIP = WiFi.localIP();
    // if (currentIP != initialized_ip){ // uncomment to use the web server!!!!
    //     if(VERBOSE_WEB_OUTPUT){
    //         Serial.println(WiFi.localIP());
    //         Serial.println("Resetting IP Adress . . .");
    //     }
    //     initWIFI();
    // }
    // if(IPAddress myIP != ip){
    //     initWIFI();
    // }
    //----------------------------------------------------------------------------
    client = server.available(); // listen for incoming clients
    if (client) { // if you get a client,
        //Serial.println("New Client."); // print a message out the serial port
        bool currentLineIsBlank = true;
        while (client.connected()) { // loop while the client's connected
            if (client.available()) { // if there's bytes to read from the client,
                char c = client.read(); // read a byte, then
                //Serial.print(c);
                HTTP_req += c; // save the HTTP request 1 char at a time
                if (c == \n & currentLineIsBlank) { // if the byte is a newline character
                    // if the current line is blank, you got two newline characters in a row.
                    // that's the end of the client HTTP request, so send a response:

                    // send a standard http response header
                    client.println("HTTP/1.1 200 OK");
                    client.println("Access-Control-Allow-Origin: *");
                    client.println("Connection: close");
                    client.println("n");

                    // ajax.setClient(client);
                    // ajax.readHTTP(HTTP_req);
                    ajaxGetRequest(HTTP_req); // send the data to a
HTTP_req = "";
break;
}
if(c == "\n") { // if you got a newline, then
  clear currentLine:
  currentLineIsBlank = true;
}
else if(c != 'r') { // if you got anything else
  but a carriage return character,
  currentLineIsBlank = false;
}
}
}
client.stop(); // close the connection:
}

// Check what data is requested by the Web Server
void WIFI::ajaxGetRequest(String http){
  if (http.indexOf("elementIDs") > 0) { // The Id list will be asked once, or each
time a page is refreshed
    client.println(list_ID);
  }
  else if(http.indexOf("elementValues.ajax") > 0){
    client.println(list_Val); // The Values will be updated every second
      unless specifically called
  }
  else if(http.indexOf("cmd") > 0){
    parseCommand(http);
  }
}

void WIFI::parseCommand(String data){
  //Serial.println(data); // print the raw data for debugging

  // check if the incoming data is specifically for the drone command (Throttle, roll, or
pitch)
  if(data.indexOf("TXY")){

    // get the delimiters.
    start1 = data.indexOf("\n");
    //   end1 = data.indexOf(";");
    //   end2 = data.indexOf(";\n");
    end3 = data.indexOf("!");

    testString = "";

for(int i = start1; i < end3+1; i++)
    testString += data[i];
    //uart.send_UART(data[i]);
}

// // empty the previous strings
// TH = "";
// dx = "";
// dy = "";
//
// // get the throttle
// for(int i = start1+1; i < end1; i++)
//    TH += data[i];  // concatonate the data
// }
//
// // get the X setPoint
// for(int i = end1+1; i < end2; i++)
//    dx += data[i];                // concatonate the data
// }
//
// // get the Y setPoint
// for(int i = end2+1; i < end3; i ++)
//    dy += data[i];             // concatonate the data
// }

uart.send_UART(testString);  // send the commands to the
teensy
Serial.println(testString);  // print the string for debugging.

// Find the positions of delimeters
int beginning_of_command = (data.indexOf("="));
int end_of_command = (data.indexOf("_"));
int end_of_value = (data.indexOf("&"));

// Empty the previous Command and Value Strings
Command = "";
Value = "";

// Concataenate to get the string of command
for(int i = beginning_of_command+1; i < end_of_command; i++)
    Command += data[i];
}

// Concataenate to get the string of value
for(int j = end_of_command+1; j < end_of_value; j ++){
    Value += data[j];
}

// uncomment to display the command through Serial
//Serial.println(Command);
//Serial.println(Value);

// Send the newly parsed data to be grouped by type
//groupCommand(Command, Value);
APPENDIX H: Teensy Main

/******************************************

Main
DRONE FIRMWARE
CAL POLY POMONA

******************************************

/***************************************************************************/

/* Local Includes */
#include "Control.h"
#include "IMU.h"
#include "Motors.h"
#include "SerialCom.h"
#include "UART.h"
#define SERIAL_BAUD  115200                   // baud rate of serial comms

/* setup */

void setup() {
    Serial.begin(SERIAL_BAUD);                     // setup Serial Communication
    Serial.println("****************************************");
    Serial.println("DRONE FIRMWARE V4.1, 2020-03-28");
    Serial.println("****************************************");
    uart.init_UART(); // initialize the UART between ESP32 and Teensy
    motor.begin();  // initialize the ESC's and the motors
    imu.begin(); // initialize the IMU
    control.begin();
}

/* loop */

void loop() {
    uart.receive_UART(); // handle the command reading from the ESP32
    parser.checkSerial(); // handle Command requests
    imu.updateIMU();  // update the IMU values
    control.update(); // updates the desired values and
    stability of the drone
}
APPENDIX I: Teensy UART.h

/*
UART Communication between ESP32 and teensy 4.0
RX AND TX should be connected to the GPIO16 and GPIO17 respectively
We are using RX/TX #5 on the Teensy 4.0
*/

#ifndef UART_H
#define UART_H
#include "Arduino.h"
#define UART_BAUD 57600

class UART{
public:
UART();                  // Constructor
void init_UART(void); // initialize the UART Serial Transfer
void send_UART(String); // sends the data to through UART
void receive_UART(void);                      // listens to incoming Serial data and parses it accordingly
int getThrottle();                            // gets the value of the desired throttle
int getRollSetpoint();  // gets the most recent value of the desired roll
int getPitchSetpoint();    // gets the most recent value of the desired pitch

private:
String TH, dx, dy; // variables to store Throttle, change in X, and change in Y. incoming commands.
String incomingString; // concatenates the incoming bytes of characters in to a cohesive string which we can manipulate for desired data.
char incomingChar; // binary incoming data over serial from the ESP32
int start1,end1,end2,end3;  // delimiters necessary for parsing the incoming data.
};

extern UART uart;
#endif
#include "UART.h"

/* Externa*/
UART uart;

/* Constructor */
UART::UART(){

void UART::init_UART(){
    Serial5.begin(UART_BAUD);                 // The baud rate has to be the same as in the
    // The baud rate has to be the same as in the
    ESP node-mcu
}

void UART::send_UART(String _toSend){
    mySerial.print(_toSend); // check why "print" works, but "write" does not
    //}
}

void UART::receive_UART(){
    if (Serial5.available()>0) {
        incomingChar = char(Serial5.read());
        incomingString += incomingChar;

        if(incomingChar == '!' )
            //Serial.println(incomingString); // print here to debug incoming information

            // load the delimeters
            start1 = incomingString.indexOf("\^") ;
            end1 = incomingString.indexOf(":" );
            end2 = incomingString.indexOf("; ");
            end3 = incomingString.indexOf("!");

            // empty the previous strings
            TH = "";
            dx = "";
            dy = "";

            // get the throttle
            for(int i = start1+1; i < end1; i++){
                TH += incomingString[i]; // concatonate the data
            }

            // get the X setPoint
            for(int i = end1+1; i < end2; i++){
                dx += incomingString[i]; // concatonate the data
            }

}
// get the Y setPoint
for(int i = end2+1; i < end3; i ++){
    dy += incomingString[i];  // concatenate the data
}

    incomingString = "";
}
}
}

int UART::getThrottle(){
    return TH.toInt();  // convert the string to an integer and return it.
}

int UART::getRollSetpoint(){
    return dx.toInt();
}

int UART::getPitchSetpoint(){
    return dy.toInt();
}
APPENDIX K: Teensy SerialCOM.h

/* Serial Communication V2.0: Combined with StringSplitter Library
* this program will contain the Serial Communication protocol and parser.
* Designed to work with both new line and carriage return input
* 
* Function: StringSplitter(String s, char c, unsigned int l);
* Where s = input string, c = desired delimiter, and l = the number of elements
* To allow a longer string input, adjust the MAX value
*/
#ifndef SerialCom_H
#define SerialCom_H
#include "Arduino.h"

class SerialParser{
public:
  SerialParser();                                         // Constructor
  void checkSerial(void);  // waits for and reads the incoming
  command line data
  void StringSplitter(String s, char c, unsigned int l);  // splits the string with a desired
demiliter and a length
  int getItemCount();                                     // gets the item count of the string
  String getItemAtIndex(int index);  // gets the string at the desired index
private:
  static const unsigned int MAX = 10;
  String op[MAX];
  int count = 0;
  int countOccurencesOfChar(String s, char c);
  void ParseData(String _str[]);                          // Parse the delimited data
  String SerialString = ""; // holds the concatonated characters from
  the incoming Serial.
  char SerialCharacter;                   // holds the incoming Serial data character
  by character.
  #define num_of_arguments 10  // holds the number of arguments
  expected as a command.
  String delimitedString[num_of_arguments]; // holds the array of delimited
  string from the incoming Serial.
  int itemCount;                              // holds the number of delimited string
  Values in the incoming String.
  byte StringToByte(String); // converts the String byte to a binary
  byte to be used.
};
extern SerialParser parser;
#endif
#include "Motors.h"
#include "SerialCom.h"

/* External Object */
SerialParser parser;

/* Debugging */
const bool VERBOSE_OUTPUT_PARSER = false; // set to false for less verbose output

/* Constructor */
SerialParser::SerialParser()
{

void SerialParser::checkSerial()
{
    SerialString = "";
    for(int i = 0; i < num_of_arguments; i++)
    {
        delimitedString[i] = ' ';
    }
    // clear any data in the string
    while(Serial.available()) { // check for serial
        SerialCharacter = Serial.read(); // read the incoming character
        SerialString.concat(SerialCharacter); // add the character to our current string
    }
    SerialString = SerialString.replace("\r\n","" ).trim(); // removes new line and/or carriage return characters.
    if (SerialString != "") { // if we have a string,
        if(VERBOSE_OUTPUT_PARSER){Serial.println(SerialString);} // print it. (optional)
        StringSplitter(SerialString, ' ', num_of_arguments); // new
        StringSplitter(string_to_split, delimiter, limit)
        itemCount = getItemCount(); // count the number of items
        if(VERBOSE_OUTPUT_PARSER){Serial.println("Item count: "+String(itemCount));}
        for(int i = 0; i < itemCount; i++)
        {
            delimitedString[i] = getItemAtIndex(i); // get the space delimited command
            if(VERBOSE_OUTPUT_PARSER){Serial.println("Item @ index " + String(i) + ":" + String(delimitedString[i]));}
        }
    }
    // if(VERBOSE_OUTPUT_PARSER){
    //     for(int i = 0; i < num_of_arguments; i++)
    //     {
    //         delimitedString[i] = ' '; // clear any data in the string
    //     }
    //     while(Serial.available()) { // check for serial
    //         SerialCharacter = Serial.read(); // read the incoming character
    //         SerialString.concat(SerialCharacter); // add the character to our current string
    //     }
    //     SerialString = SerialString.replace("\r\n","" ).trim(); // removes new line and/or carriage return characters.
    //     if (SerialString != "") { // if we have a string,
    //         if(VERBOSE_OUTPUT_PARSER){Serial.println(SerialString);} // print it. (optional)
    //         StringSplitter(SerialString, ' ', num_of_arguments); // new
    //         StringSplitter(string_to_split, delimiter, limit)
    //         itemCount = getItemCount(); // count the number of items
    //         if(VERBOSE_OUTPUT_PARSER){Serial.println("Item count: "+String(itemCount));}
    //         for(int i = 0; i < itemCount; i++)
    //         {
    //             delimitedString[i] = getItemAtIndex(i); // get the space delimited command
    //             if(VERBOSE_OUTPUT_PARSER){Serial.println("Item @ index " + String(i) + ":" + String(delimitedString[i]));}
    //         }
    //     }
    // }
// Serial.println(String(i)+": "+String(delimitedString[i]));
// }
// }
ParseData(delimitedString);
}
}

byte SerialParser::StringToByte(String _input){
  byte out = 0;
  uint8_t L = _input.length();
  //Serial.println("Input String length: "+String(L));
  for(int i = 0; i < _input.length(); i++){
    if(_input.charAt(i) == '1'){ bitWrite(out,(L-1)-i,1);}// Serial.println("1 at: "+String(i));}
  //Serial.println(out);
  return out;
}

void SerialParser::StringSplitter(String s, char c, unsigned int limit){
  count = countOccurencesOfChar(s, c) + 1;  // fix for no elements, ie return input string
  if(count <= 1 || limit <= 1){
    count = 1;
    op[0] = s;
    return;
  }
  if(count > limit) count = limit;
  if(count > MAX) count = MAX;
  String d = String(c);
  String first;
  String second = s;
  int current = 0;
  while(second.indexOf(d) > -1){
    if(current >= (count - 1)){
      //current++;
      break;
    }
    for (int i = 0; i < second.length(); i++) {
      if (second.substring(i, i + 1) == d) {
        first = second.substring(0, i);
        second = second.substring(i + 1);
        if(first.length() > 0)
op[current++] = first;
break;
}
}
}

//current = (current < MAX - 1) ? current : MAX;
if(second.length() > 0)
  op[current] = second;
//else
  // --count;
}

int SerialParser::countOccurencesOfChar(String s, char c){
  int size = 0;
  for(int x = 0; x < s.length(); (s[x] == c) ? size++ : 0, x++);
  return size;
}

int SerialParser::getItemCount(){
  return count;
}

String SerialParser::getItemAtIndex(int index){
  if((index >= 0) && (index < count))
    return op[index];
  else
    return "";
}

/****************************************************************************
//-------------------------------This is where we parse the commands-----------------
-----------------------------------Only Modify below this line---------------------------------
-//-------------------------------This is where we parse the commands-------------------
-----
//-----------------------------------Enter your Commands here----------------------------------
---
/****************************************************************************

void SerialParser::ParseData(String _str[]){
  if(_str[0].equalsIgnoreCase("hello") || _str[0].equalsIgnoreCase("hi") && itemCount == 1){
    Serial.println("Hello user");
  } /* --------------------------------------------------------------- Motor Commands */
  // set the speed of the motor
  else if(_str[0].equalsIgnoreCase("M") && itemCount == 3){
    motor.setSpeed(_str[1].toInt(),_str[2].toInt());
}
}  
// set speed in RPM  
else if(_str[0] == "S" && itemCount == 3){  
    motor.setSpeedRPM(_str[1].toInt(),_str[2].toInt());  
}  
// set throttle of all motors in RPM  
else if(_str[0].equalsIgnoreCase("T") && itemCount == 2){  
    motor.setThrottle(_str[1].toInt());  
}  
/* ----------------------------------------------------------------- Default Commands */  
else{  
    Serial.println("Command: "+String(SerialString)+" not found");  
}  
}
#ifndef Motors_H
#define Motors_H

#include "Arduino.h"

/* Defined Constants */
#define MAX_SIGNAL 2000
#define MIN_SIGNAL 1000
#define MOTOR1_PIN 0
#define MOTOR2_PIN 1
#define MOTOR3_PIN 2
#define MOTOR4_PIN 3

class Motor{
public:
    Motor();  // Constructor
    void begin(void); // set up the motors
    void setSpeed(int, int); // sets the speed of the motor, in raw speed [1000-2000]
    void setThrottle(int); // sets the throttle on all motors in RPM [0 - 12,000]
    void setSpeedRPM(int,int); // sets the speed of the motors in RPM using linear conversion

private:

};

extern Motor motor;

#endif
 APPENDIX N: Teensy Motors.cpp

#include "Control.h"
#include "Motors.h"
#include <Servo.h>
/* External Object */
Motor motor;
Servo motor1, motor2, motor3, motor4;  // initialize the servo objects for each motor

/* Debugging */
const bool VERBOSE_OUTPUT_MOTOR = false;  // set to false for less verbose output

/* Constructor */
Motor::Motor(){

void Motor::begin(){

delay(2000);

    // initialize Motor 1
    motor1.attach(MOTOR1_PIN);
    motor1.writeMicroseconds(MAX_SIGNAL);
    delay(2500);
    motor1.writeMicroseconds(MIN_SIGNAL);
    Serial.println("Motor 1 calibrating");
    delay(1000);

    // initialize Motor 2
    motor2.attach(MOTOR2_PIN);
    motor2.writeMicroseconds(MAX_SIGNAL);
    delay(2500);
    motor2.writeMicroseconds(MIN_SIGNAL);
    Serial.println("Motor 2 calibrating");
    delay(1000);

    // initialize Motor 3
    motor3.attach(MOTOR3_PIN);
    motor3.writeMicroseconds(1500);
    delay(2500);
    motor3.writeMicroseconds(MIN_SIGNAL);
    Serial.println("Motor 3 calibrating");
    delay(1000);
}
// initialize Motor 4
motor4.attach(MOTOR4_PIN);
motor4.writeMicroseconds(MAX_SIGNAL);
delay(2500);
motor4.writeMicroseconds(MIN_SIGNAL);
Serial.println("Motor 4 calibrating");
delay(1000);

Serial.println("Motor setup Finished");
}

void Motor::setSpeed(int _motorNUM, int _Speed) {
  if (_Speed <= 1700) {
    Serial.println(_motorNUM);
    Serial.println(_Speed);
    if (_Speed == 0) {
      _Speed = 1000;
    }
    if (_motorNUM < 0 || _motorNUM > 4 || _Speed < 1000 || _Speed > 1600) {
      Serial.println("Invalid Motor or Speed");
    } else {
      Serial.println(_Speed);
      Serial.println("Motor "+String(_motorNUM)+ " Speed: "+String(_Speed));
      switch(_motorNUM) {
        case 1:
          motor1.writeMicroseconds(_Speed);
          break;
        case 2:
          motor2.writeMicroseconds(_Speed);
          break;
        case 3:
          // if (_Speed > 1100) {
          //   break;
          // }
          motor3.writeMicroseconds(_Speed);
          break;
        case 4:
          motor4.writeMicroseconds(_Speed);
          break;
      }
    }
  }
  return;
}
void Motor::setThrottle(int _RPM){
    if(_RPM >= 0 && _RPM <= 19000){
        for(int i = 1; i < 5; i++){
            setSpeedRPM(i, _RPM);
        }
        control.Throttle = _RPM;
    }
    else{
        Serial.println("RPM value is invalid, [0-12000] only");
    }
}

void Motor::setSpeedRPM(int _motorNum, int _RPM){
    if(_RPM <= 9500){
        if(_motorNum < 0 || _motorNum > 4){
            Serial.println("Motor number error");
        }
        else{
            switch(_motorNum){
            case 1:
                motor1.writeMicroseconds(0.0344*_RPM + 1009.9);
                break;
            case 2:
                motor2.writeMicroseconds(0.0355*_RPM + 1003.6);
                break;
            case 3:
                // motor3.writeMicroseconds(0.0308*_RPM + 1033.6);
                // motor3.writeMicroseconds(0.0308*_RPM + 1033.6+5.5);
                break;
            case 4:
                // motor4.writeMicroseconds(0.0303*_RPM + 1036.3);
                break;
            }
        }
    }
}
APPENDIX O: Teensy IMU.h

#ifndef IMU_H
#define IMU_H
#include <elapsedMillis.h>

#include "Arduino.h"

/* Reg.Name  Address  Description */
#define GYRO_CONFIG  0x1B // holds the gyroscope Configuration
#define ACCEL_CONFIG  0x1C // holds the Accelerometer Configuration
#define ACCEL_XOUT_H 0x3B // this will read the x-axis accelerometer data, high byte
#define ACCEL_XOUT_L 0x3C
#define ACCEL_YOUT_H 0x3D
#define ACCEL_YOUT_L 0x3E
#define ACCEL_ZOUT_H 0x3F
#define ACCEL_ZOUT_L 0x40
#define GYRO_XOUT_H  0x43 // this will read the x-axis gyroscope data, high byte
#define GYRO_XOUT_L  0x44
#define GYRO_YOUT_H  0x45
#define GYRO_YOUT_L  0x46
#define GYRO_ZOUT_H  0x47
#define GYRO_ZOUT_L  0x48
#define XA_OFFSET_H  0x77
#define XA_OFFSET_L 0x78
#define YA_OFFSET_H  0x7A
#define YA_OFFSET_L 0x7B
#define ZA_OFFSET_H  0x7D
#define ZA_OFFSET_L 0x7E
#define CONFIG        0x1A  // configuration register. User should set the msb of this register LOW
#define PWR_MGMT_1  0x6B // power management register 1. This turns ON/OFF the functions of the chip
#define PWR_MGMT_2  0x6C // power management register 2. This turns ON/OFF the functions of the chip
#define WHO_AM_I 0x75 // used to identify the device. should return 0x12 when read
#define OFFSET  0x80  // This offset is necessary when addressing registers to write or read to the correct register. Very Important
#define I2C_IF  0x70  // disables the i2c mode when using SPI

/* Defined Constants */
#define CS 10
#define SPI_CLOCK      10000000 // spi clock frequency (10MHz)

class IMU{
public:
IMU();                   // Constructor
void begin(void);       // set up the motors
void updateIMU(void);   // runs the loop to update IMU values.
float getRoll(void);    // returns the roll of the imu
float getPitch(void);   // returns the pitch of the imu
float getYaw(void);

private:
elapsedMillis AccTimer = 0;                      // averaging the accelerometer for more
precise data
elapsedMillis Timer2 = 0;          // temporary timer

float get_GYRO_rawX(void);                       // read and return the raw gyroscope data
in the X direction
float get_GYRO_rawY(void);                       // read and return the raw gyroscope data
in the Y direction
float get_GYRO_rawZ(void);                       // read and return the raw gyroscope data
in the Y direction
float get_ACCEL_rawX(void);                      // read and return the raw Accelerometer
data in the X direction
float get_ACCEL_rawY(void);                      // read and return the raw Accelerometer
data in the Y direction
float get_ACCEL_rawZ(void);                      // read and return the raw Accelerometer
data in the Z direction

byte MSB,LSB;
float Acc_rawX, Acc_rawY, Acc_rawZ;              // Here we store the raw data read
float Acc_angle_x, Acc_angle_y;                  // Here we store the angle value obtained
with Acc data
float elapsedTime, currentTime, previousTime;
float GyroX, GyroY, GyroZ;
float roll =0, pitch = 0, yaw = 0;
float Ar = 0, Ap = 0;                            // accelerometer pitch and accelerometer roll
float Gr = 0, Gp = 0;                            // gyroscope pitch and gyroscope roll
float Er = 0, Ep = 0;                            // gyroscope drift error to compensate for pitch
and roll
float gyroAngleX, gyroAngleY, gyroAngleZ;
float gyroErrorX, gyroErrorY, gyroErrorZ;
float accelErrorX, accelErrorY;
float ACCEL_CALIBRATION_VALUE = 2048.0;           // this is for +-16g
float GYRO_CALIBRATION_VALUE  = 16.4;            // this is for 2000 dps check
the calibration to set the correct value here

float ACC_X_SUM, ACC_Y_SUM, ACC_Z_SUM;
int ACC_count;
#define ACC_Samples 20 // get as many samples in this amount of time

float SUM_X, SUM_Y, SUM_Z;
float Sum_gyroX, Sum_gyroY, Sum_gyroZ, SumRoll, SumPitch;
int count;

// capture 21 floats for averaging and median filter.

#define SIZE 133 // 7.52Hz // size of the array. Must always be odd number
    float accelY_arr[SIZE];
    float accelX_arr[SIZE];
    uint8_t index = 0; // holds position of array.
    void Sort(float arr1[], float arr2[]); // sorts by lowest absolute value
    float bubbleSort(float a[]); // bubble sort algorithm in order, will return median value

// best so far has been 75 samples at 2ms at 6.6Hz. better than 30 samples at 5ms with same frequency
// also really good is 70 samples at 2ms at 7Hz. Not as good but higher frequency
// previously best size 133 and 1ms sample at 7.52Hz

};

extern IMU imu;

#undef
APPENDIX P: Teensy IMU.cpp

#include "Arduino.h"
#include "IMU.h"
#include <SPI.h>

/* External Object */
IMU imu;

/* Debugging */
const bool VERBOSE_OUTPUT_IMU = false;  // set to false for less verbose output
const bool IMU_CALIBRATE = false;       // set to false for no calibration

/* Configure the SPI Comms */
SPISettings mySPI(SPI_CLOCK, MSBFIRST, SPI_MODE0);  // SPI initialization

/* Constructor */
IMU::IMU(){}

void IMU::updateIMU()
{

    // === Read accelerometer raw data ===
    Acc_rawX= get_ACCEL_rawX(); // read Accel_X raw
    Acc_rawY= get_ACCEL_rawY(); // read Accel_Y raw
    Acc_rawZ= get_ACCEL_rawZ(); // read Accel_Z raw

    // compute the X and Y values
    Acc_angle_x = (atan(Acc_rawY / sqrt(pow(Acc_rawX, 2) + pow(Acc_rawZ, 2))) * 180 / PI);//-accelErrorX;
    Acc_angle_y = (atan(-1 * Acc_rawX / sqrt(pow(Acc_rawY, 2) + pow(Acc_rawZ, 2))) * 180 / PI);//-accelErrorY;

    // alternate equation
    // Acc_angle_x = atan2(Acc_rawY, Acc_rawZ) * 180 / PI - accelErrorX;
    // Acc_angle_y = atan2(-1 * Acc_rawX, sqrt(Acc_rawY * Acc_rawY + Acc_rawZ * Acc_rawZ)) * 180 / PI - accelErrorY;

    // === Read gyroscope raw data ===

    GyroX = get_GYRO_rawX() - gyroErrorX;
    GyroY = get_GYRO_rawY() - gyroErrorY;
    GyroZ = get_GYRO_rawZ() - gyroErrorZ;
    previousTime = currentTime;   // Previous time is stored before the actual time read
    currentTime = micros();       // Current time actual time read
    elapsedTime = (currentTime - previousTime) / 1000000; // Divide by 1000 to get seconds

    gyroAngleX = gyroAngleX + GyroX * elapsedTime; // deg/s * s = deg
gyroAngleY = gyroAngleY + GyroY * elapsedTime;

Gr = gyroAngleY;
Gp = gyroAngleX;
roll = Gr - Er;
pitch = Gp - Ep;
yaw = yaw + GyroZ * elapsedTime;

if(VERBOSE_OUTPUT_IMU){
    if(Timer2 > 40){
        Serial.print(Gr); Serial.print(" ");
        Serial.print(Er); Serial.print(" ");
        Serial.print(Ar); Serial.print(" ");
        Serial.print(roll); Serial.print(" ");
        // Serial.print(pitch); Serial.print(" ");
        //Serial.print(millis()); Serial.print(" ");
        Serial.println();
        Timer2 = 0;
    }
}

if(AccTimer < 1){
    SUM_X+= Acc_angle_x;
    SUM_Y+= Acc_angle_y;
    count++;
}
else{
    accelY_arr[index] = SUM_X/count;
    accelX_arr[index] = SUM_Y/count;
    index++;
    if (index == SIZE){
        //Sort(accelY_arr, accelX_arr); // minimum abs value sort
        Ap = bubbleSort(accelY_arr);
        Ar = bubbleSort(accelX_arr);
        index = 0;
        Er = Gr-Ar; // adjust the roll offset
        Ep = Gp-Ap; // adjust the pith offset
        // if(VERBOSE_OUTPUT_IMU){
        /// Serial.print(Gr); Serial.print(" ");
        /// Serial.print(Gp-Ap); Serial.print(" ");
        /// Serial.print(yaw); Serial.print(" ");
        /// Serial.print(millis()); Serial.print(" ");
    }
void IMU::begin()
{
pinMode(CS, OUTPUT);
digitalWrite(CS, HIGH);
SPI.begin();

  // set the I2C_IF to disable. Protects the chip from randomly switching to i2c mode and losing communication
  digitalWrite(CS, LOW);
  SPI.beginTransaction(mySPI);
  SPI.transfer(I2C_IF);
  SPI.transfer(0x40);
  SPI.endTransaction();
  digitalWrite(CS, HIGH);

  // set the Power management 1
  digitalWrite(CS, LOW);
  SPI.beginTransaction(mySPI);
  SPI.transfer(PWR_MGMT_1);
  SPI.transfer(0x00);
  SPI.endTransaction();
  digitalWrite(CS, HIGH);

  // set the Power management 2
  digitalWrite(CS, LOW);
  SPI.beginTransaction(mySPI);
  SPI.transfer(PWR_MGMT_2);
  SPI.transfer(0x00);
  SPI.endTransaction();
  digitalWrite(CS, HIGH);

  // set the gyroscope configuration
  digitalWrite(CS, LOW);
  SPI.beginTransaction(mySPI);
  SPI.transfer(GYRO_CONFIG);
  //SPI.transfer(0x10);  // our GYRO_CALIBRATION_VALUE = 32.8
// SPI.transfer(0x00);       // our GYRO_CALIBRATION_VALUE = 131       ? check if this is correct
SPI.transfer(0x18);       // our GYRO_CALIBRATION_VALUE = 16.4
SPI.endTransaction();
digitalWrite(CS,HIGH);

// set the Accelerometer configuration
digitalWrite(CS,LOW);
SPI.beginTransaction(mySPI);
SPI.transfer(ACCEL_CONFIG);
//SPI.transfer(0x10);     // 4096
//SPI.transfer(0x00);     // 16384
SPI.transfer(0x18);  // 2048
SPI.endTransaction();
digitalWrite(CS,HIGH);

// begins the Gyroscope Initialization. This is to compute the offset values needed.
int count = 0;
float SUM_GYRO_X = 0;
float SUM_GYRO_Y = 0;
float SUM_GYRO_Z = 0;

float SUM_ACCEL_X = 0;
float SUM_ACCEL_Y = 0;
int Samples = 1000;

if(IMU_CALIBRATE){
    while(count < Samples){
        Acc_rawX= get_ACCEL_rawX(); // read Accel_X raw
        Acc_rawY= get_ACCEL_rawY(); // read Accel_Y raw
        Acc_rawZ= get_ACCEL_rawZ(); // read Accel_Z raw

        SUM_ACCEL_X += (atan(Acc_rawY / sqrt(pow(Acc_rawX, 2) + pow(Acc_rawZ, 2))) * 180 / PI);
        SUM_ACCEL_Y += (atan(-1 * Acc_rawX / sqrt(pow(Acc_rawY, 2) + pow(Acc_rawZ, 2))) * 180 / PI);

        previousTime = currentTime; // Previous time is stored before the actual time read
        currentTime = micros(); // Current time actual time read
        elapsedTime = (currentTime - previousTime) / 1000000; // Divide by 1000 to get seconds

        SUM_GYRO_X += get_GYRO_rawX();
        SUM_GYRO_Y += get_GYRO_rawY();
        SUM_GYRO_Z += get_GYRO_rawZ();
    }
}
count++;
}
gyroErrorX = SUM_GYRO_X/count;
gyroErrorY = SUM_GYRO_Y/count;
gyroErrorZ = SUM_GYRO_Z/count;

accelErrorX = SUM_ACCEL_X/count;
accelErrorY = SUM_ACCEL_Y/count;
}
delay(1000);
Serial.println("IMU Initialization Complete");
//Serial.println(gyroErrorX);
//Serial.println("Count: "+String(count));

// // read a register configuration. This is for debugging the setup conditions. The value read is displayed in decimal format
// digitalWrite(CS,LOW);
// SPI.beginTransaction(mySPI);
// SPI.transfer(GYRO_CONFIG+OFFSET);
// SPIreading = SPI.transfer(0x00);
// SPI.endTransaction();
// digitalWrite(CS,HIGH);
// Serial.println("GYRO_CONFIG: "+String(SPIreading));

gyroAngleX = 0;
gyroAngleY = 0;
roll = 0;
pitch = 0;
}

float IMU::get_GYRO_rawX(){
digitalWrite(CS,LOW);
SPI.beginTransaction(mySPI);
SPI.transfer(GYRO_XOUT_H+OFFSET);
MSB = SPI.transfer(0x00);
LSB = SPI.transfer(0x00);
SPI.endTransaction();
digitalWrite(CS,HIGH);
float VAL = (MSB << 8 | LSB);  // division by 182.0 gives angle between 0 and 360 degrees
if(VAL > 32760.0){
  VAL = VAL-65535.0;      // subtract to get the negative degree
}
return VAL/GYRO_CALIBRATION_VALUE;  // ranges from 0 to 1998 because division by 32.8
float IMU::get_GYRO_rawY() {
    digitalWrite(CS, LOW);
    SPI.beginTransaction(mySPI);
    SPI.transfer(GYRO_YOUT_H + OFFSET);
    MSB = SPI.transfer(0x00);
    LSB = SPI.transfer(0x00);
    SPI.endTransaction();
    digitalWrite(CS, HIGH);
    float VAL = (MSB << 8 | LSB);  // division by 182.0 gives angle between 0 and 360 degrees
    if (VAL > 32760.0) {
        VAL = VAL - 65535.0;    // subtract to get the negative degree
    }
    return VAL / GYRO_CALIBRATION_VALUE;  // ranges from 0 to 1998 because division by 32.8
}

float IMU::get_GYRO_rawZ() {
    digitalWrite(CS, LOW);
    SPI.beginTransaction(mySPI);
    SPI.transfer(GYRO_ZOUT_H + OFFSET);
    MSB = SPI.transfer(0x00);
    LSB = SPI.transfer(0x00);
    SPI.endTransaction();
    digitalWrite(CS, HIGH);
    float VAL = (MSB << 8 | LSB);  // division by 182.0 gives angle between 0 and 360 degrees
    if (VAL > 32760.0) {
        VAL = VAL - 65535.0;    // subtract to get the negative degree
    }
    return VAL / GYRO_CALIBRATION_VALUE;  // ranges from 0 to 1998 because division by 32.8
}

float IMU::get_ACCEL_rawX() {
    digitalWrite(CS, LOW);
    SPI.beginTransaction(mySPI);
    SPI.transfer(ACCEL_XOUT_H + OFFSET);
    MSB = SPI.transfer(0x00);
    LSB = SPI.transfer(0x00);
    SPI.endTransaction();
    digitalWrite(CS, HIGH);
    float VAL = (MSB << 8 | LSB);  // division by 182.0 gives angle between 0 and 360 degrees
    if (VAL > 32760.0) {
        VAL = VAL - 65535.0;    // subtract to get the negative degree
    }
}
return VAL/ACCEL_CALIBRATION_VALUE;  // division by 182.0 gives angle between 0 and 360 degrees;
}

float IMU::get_ACCEL_rawY(){
digitalWrite(CS,LOW);
SPI.beginTransaction(mySPI);
SPI.transfer(ACCEL_YOUT_H+OFFSET);
MSB = SPI.transfer(0x00);
LSB = SPI.transfer(0x00);
SPI.endTransaction();
digitalWrite(CS,HIGH);
float VAL = (MSB << 8 | LSB);  // division by 182.0 gives angle between 0 and 360 degrees
if(VAL > 32760.0){
    VAL = VAL-65535.0;      // subtract to get the negative degree
}
return VAL/ACCEL_CALIBRATION_VALUE;  // division by 182.0 gives angle between 0 and 360 degrees;
}

float IMU::get_ACCEL_rawZ(){
digitalWrite(CS,LOW);
SPI.beginTransaction(mySPI);
SPI.transfer(ACCEL_ZOUT_H+OFFSET);
MSB = SPI.transfer(0x00);
LSB = SPI.transfer(0x00);
SPI.endTransaction();
digitalWrite(CS,HIGH);
float VAL = (MSB << 8 | LSB);  // division by 182.0 gives angle between 0 and 360 degrees
if(VAL > 32760.0){
    VAL = VAL-65535.0;      // subtract to get the negative degree
}
return VAL/ACCEL_CALIBRATION_VALUE;  // division by 182.0 gives angle between 0 and 360 degrees;
}

float IMU::getRoll(){
    if(roll < 180){
        return roll;
    } else{
        return 0;
    }
}

float IMU::getPitch(){

if(pitch < 180) {
    return pitch;
}
else{
    return 0;
}
}

float IMU::getYaw() {
if(yaw < 180) {
    return yaw;
}
else{
    return 0;
}
}

void IMU::Sort(float arr1[], float arr2[]) {
    Ap = arr1[1];
    Ar = arr2[1];
    for(int i = 1; i < SIZE; i++) {
        if(abs(arr1[i]) < abs(Ap)) {
            Ap = arr1[i];
        }
        if(abs(arr2[i]) < abs(Ar)) {
            Ar = arr2[i];
        }
    }
}

float IMU::bubbleSort(float a[]) {
    for(int i=0; i<(SIZE-1); i++) {
        for(int o=0; o<(SIZE-(i+1)); o++) {
            if(a[o] > a[o+1]) {
                int t = a[o];
                a[o] = a[o+1];
                a[o+1] = t;
            }
        }
    }
    return a[round(SIZE/2)];
    Serial.println(a[round(SIZE/2)]);
}

/////////////////////////////////////////////////////// original sort
//void IMU::Sort(float arr1[], float arr2[]){
// pitch = arr1[0];
// roll = arr2[0];
// for(int i = 1; i < SIZE; i++)
// if(abs(arr1[i]) < abs(Ap))
//   pitch = arr1[i];
// if(abs(arr2[i]) < abs(Ar))
//   roll = arr2[i];
// }
//}
APPENDIX Q: Teensy Control.h

/*
This controls the drone, setting the correct throttle(height) as well as PID stabilization
Guidelines for update_roll and update_pitch
check if we have a percent error of 1.5% or more.
if no: then we set roll_adjust or pitch_adjust to 0;
if yes: - increment roll_adjust or pitch_adjust.
    - check if we have a positive shift or negative and adjust the motors with that in
mind.
    - the equation for setting this should ALWAYS BE: motor.setSpeedRPM(Motor
#, throttle +/- roll_adjust, +/- pitch adjust);
*/

#ifndef Control_H
#define Control_H

#include "Arduino.h"
#include "elapsedMillis.h"

class Control{
public:
    Control();     // Constructor
    void begin();  // intialize
    void update(void);                // updates the controller
    void update_Throttle(void); // updates the throttle of the motors.
    void update_Controller(void);  // updates roll and pitch
    void moveDrone(int);     // moves the drone in the desired position.
    void update_setPoints();  // updates the desired setPoint of the drone.

    int Throttle;  // holds the desired throttle value in RPM [0-10000]

    void update_Controller2(void);  // 2nd way

    float PitchError = 0;
    float RollError = 0;
    float YawError = 0;

    float PitchErrorLast = 0;
    float RollErrorLast = 0;
    float YawErrorLast = 0;

    float RollProportional, PitchProportional, YawProportional;
    float RollIntegral =0, PitchIntegral =0;
    float RollDerivative = 0, PitchDerivative = 0;

};

#endif Control_H
float dt = 0, currentTime = 0, previousTime = 0;

#define KP 100 // oscillation at 70 and above
#define KI 0.00  // BEST 0.01
#define KD 0 // Derivative constant // BEST 1000

// Best Values, actually stable drone
// #define KP 20 // oscillation at 70 and above
// #define KI 0.001 // BEST 0.001
// #define KD 100 // Derivative constant // BEST 1000

//------------------------------
private:
  elapsedMillis ThrottleTimer; // times the throttle so that we dont change too fast or too slow allowing greater control of drone Throttle
  elapsedMillis ControlTimer; // timer for updating the controller for change in pitch and roll
  elapsedMillis setPointTimer; // timer for updating the desired setPoint of the drone

float ROLL = 0, PITCH = 0; // holds the values of the current roll and pitch
float YAW = 0;

float T[4]; // holds the motor Throttle to send to the motors. This will set how far each motor will spin.

int r1, r2, r3, r4, p1, p2, p3, p4; // holds the values for roll and pitch of each individual motor.

int roll_adjust, pitch_adjust, yaw_adjust; // how much change we need to make to overall throttle offset to keep the drone at the desired setpoint.

#define MAX_OFFSET 500 // this is the max offset of the roll and pitch. So the drone does not flip over or increase RPM indefinitely (500 initially)
#define OFFSET_ERROR 0.5 // error at which the system is acceptable and does not need change.

void resetRoll(); // reset the r adjustments
void resetPitch();

float rollSetpoint = 90;
float pitchSetpoint = 90;
float yawSetpoint = 0;
float rollSUM = 0, pitchSUM = 0;
int count = 0;

enum {MOVE_LEFT, MOVE_RIGHT, MOVE_FORWARD,
MOVE_BACKWARD};

};

extern Control control;

#undef
APPENDIX R: Teensy Control.cpp

```cpp
#include "Control.h"
#include "IMU.h"
#include "Motors.h"
#include "UART.h"

/* External Object */
Control control;

/* Debugging */
const bool VERBOSE_OUTPUT_CONTROLLER  = true; // set to false for less verbose output, prints out the offset values for roll and pitch

/* Constructor */
Control::Control() {};

void Control::begin() {
    Throttle = 0;
    resetPitch();
    resetRoll();
    ROLL = 90;
    PITCH = 90;
    delay(1000);
}

void Control::update() {
    update_Throttle();

    // choose one of the two controllers to run
    //update_Controller();
    update_Controller2();
}

void Control::update_Throttle() {
    if (ThrottleTimer >= 20) { // update every 8ms (can be changed)
        if (-uart.getThrottle() > 0) { // here we increase
            if (Throttle <= 15000) { // keep it to 5000 for now
                Throttle += 50; // increment by 10; this value will be changed and we can make a proportional controller if needed.
            }
            //motor.setThrottle(Throttle); // set the motor RPM
        }
    }
    else if (-uart.getThrottle() < 0) { // here we decrease
        if (Throttle != 0) {
        } else if (Throttle != 0) {
        }
    }
}
```

71
otherwise do nothing
    Throttle -= 50;                      // same as increment but we
decrement by the same value
    //motor.setThrottle(Throttle);        // set the motor RPM
}
}
ThrottleTimer = 0;                                         // reset the timer
//Serial.println(Throttle); // print the throttle RPM
speed for debugging
}
}

// 2nd pid
void Control::update_Controller2(){
  if(ControlTimer < 100){
    rollSUM += imu.getRoll()+90;
    pitchSUM += imu.getPitch()+90;
    count++;
  }
  else{
    previousTime = currentTime;
    currentTime = millis();
    dt = currentTime - previousTime;

    rollSetpoint = (uart.getRollSetpoint()/20)+90;  // -0.6 // obtain
    pitchSetpoint = (uart.getPitchSetpoint()/20)+90; // obtain the
    the desired roll setPoint from the user
    desired pitch setPoint from the user

    ROLL = rollSUM/count;

    rollSUM = 0;
    count = 0;

    if (ROLL > 1000){
      ROLL = 90;
    }
    //    ROLL = imu.getRoll()+90;          // read in the new
    roll value, add offset 90 degrees
    //    PITCH = imu.getPitch()+90;       // read in the new
    pitch value, add offset 90 degrees
    //    YAW = imu.getYaw();             // read in the new

    //    Serial.println("PITCH: "+String(PITCH));
PitchError = (PITCH - pitchSetpoint);          // calculate the error for the pitch
RollError  = (ROLL - rollSetpoint);            // calculate the error for the roll
YawError  = (YAW - yawSetpoint);              // calculate the error for the yaw

// roll PID controller
RollProportional = KP*RollError;               // calculate Roll P
RollIntegral += KI * RollError * dt;          // calculate Roll I
RollDerivative = KD*(RollError - RollErrorLast)/dt; // calculate Roll D
roll_adjust = RollProportional + RollIntegral + RollDerivative; // get the output roll adjust

// pitch PID controller
PitchProportional = KP*PitchError;            // calculate Pitch P
PitchIntegral += KI*(PitchError)*dt;          // calculate Pitch I
PitchDerivative = KD*(PitchError - PitchErrorLast)/dt; // calculate Pitch D
pitch_adjust = PitchProportional + PitchIntegral + PitchDerivative; // get the output pitch adjust

// yaw PID controller
YawProportional = KP*YawError;
yaw_adjust = YawProportional;
yaw_adjust = 0;

T[0] = Throttle - pitch_adjust - roll_adjust - yaw_adjust;
T[1] = Throttle + pitch_adjust - roll_adjust - yaw_adjust;
T[2] = Throttle + pitch_adjust + roll_adjust + yaw_adjust;
T[3] = Throttle - pitch_adjust + roll_adjust + yaw_adjust;

// T[0] = Throttle-roll_adjust - yaw_adjust;
// T[1] = Throttle-roll_adjust - yaw_adjust;
// T[2] = Throttle+roll_adjust + yaw_adjust;
// T[3] = Throttle+roll_adjust + yaw_adjust;

if(Throttle > 500){
    for(int i = 0; i < 4; i++){
        if(T[i] < 500){
            T[i] = 500;
        }
    }
}
if(VERBOSE_OUTPUT_CONTROLLER){
    //Serial.println(String(r1)+" "+String(r2)+" "+String(r3)+" "+String(r4)+
    "+String(p1)+" "+String(p2)+" "+String(p3)+" "+String(p4));
    //Serial.println(String(Throttle+r1+p1)+" "+String(Throttle+r2+p2)+
    "+String(Throttle+r3+p3)+" "+String(Throttle+r4+p4));
    // Serial.print(T[0])     ;Serial.print(" ");
    // Serial.print(T[1])     ;Serial.print(" ");
    // Serial.print(T[2])     ;Serial.print(" ");
    // Serial.print(T[3])     ;Serial.print(" ");
    // Serial.print(rollSetpoint)  ;Serial.print(" ");
    // Serial.print(pitchSetpoint) ;Serial.print(" ");
    Serial.print(ROLL) ;Serial.print(" ");
    // Serial.print(PITCH)  ;Serial.print(" ");
    // Serial.print(RollIntegral)  ;Serial.print(" ");
    // Serial.print(PitchIntegral) ;Serial.print(" ");
    // Serial.print(abs(PITCH-pitchSetpoint)*MAX_OFFSET/90);Serial.print(" ");
    // Serial.print(RollDerivative)     ;Serial.print(" ");
    Serial.println();
}

// set the motors here (in RPM)
for( int i = 0; i < 4; i++){
    motor.setSpeedRPM(i+1,Throttle);
}

// record the previous error
RollErrorLast  = RollError;
PitchErrorLast = PitchError;
YawErrorLast  = YawError;

ControlTimer = 0;
}

void Control::update_Controller(){
    if(ControlTimer > 150 ){                                                                 // if enough time has
        // set the motors here (in RPM)
        for( int i = 0; i < 4; i++){
            motor.setSpeedRPM(i+1,Throttle);
        }
        // record the previous error
        RollErrorLast  = RollError;
        PitchErrorLast = PitchError;
        YawErrorLast  = YawError;
        ControlTimer = 0;
    }
}
rollSetpoint = (uart.getRollSetpoint()/10)+90; // obtain the desired roll setPoint from the user
pitchSetpoint = (uart.getPitchSetpoint()/10)+90; // obtain the desired pitch setPoint from the user

// Serial.print(rollSetpoint);
// Serial.print(" ");
// Serial.println(pitchSetpoint);
ROLL = imu.getRoll()+90; // read in the new roll value, add offset 90 degrees
PITCH = imu.getPitch()+90; // read in the new pitch value, add offset 90 degrees

Roll = 2
Roll = 3

//Serial.println(abs((ROLL - rollSetpoint) / rollSetpoint) * 100);

// adjust for roll setPoint
if (abs((ROLL - rollSetpoint) / rollSetpoint) * 100 >= OFFSET_ERROR) {
    // check if we need to adjust the roll, we have an error of 1.5 degrees
    if(roll_adjust < MAX_OFFSET){
        // check what the max offset should be (initially 500)
        roll_adjust = abs(ROLL-rollSetpoint)*MAX_OFFSET/90; // increase the roll offset until we reach our point, unless we are at max adjustment
    }
    if(ROLL > rollSetpoint){ // means we leaning right and moving right
        moveDrone(MOVE_LEFT);
    } else{ // otherwise...
        moveDrone(MOVE_RIGHT);
    }
} else{
    resetRoll(); // reset the roll adjusts for each individual motor
}

// adjust for pitch setPoint
if (abs((PITCH - pitchSetpoint) / pitchSetpoint) * 100 >= OFFSET_ERROR) {
    // check if we need to adjust the roll, we have an error of 1.5 degrees
    Serial.println();
    if(pitch_adjust < MAX_OFFSET){
        // check what the max offset should be (initially 500)
        pitch_adjust = abs((PITCH-pitchSetpoint)*MAX_OFFSET/90); // increase the roll offset until we reach our point, unless we are at max adjustment
    }
if (PITCH > pitchSetpoint) { // means we leaning back and moving backwards
    moveDrone(MOVE_FORWARD);
} else {
    moveDrone(MOVE_BACKWARD);
}
else {
    resetPitch(); // reset the roll adjusts for each individual motor
}

if (VERBOSE_OUTPUT_CONTROLLER) {
    // Serial.println(String(r1) + " " + String(r2) + " " + String(r3) + " " + String(r4) + " " + String(p1) + " " + String(p2) + " " + String(p3) + " " + String(p4));
    // Serial.println(String(Throttle+r1+p1) + " " + String(Throttle+r2+p2) + " " + String(Throttle+r3+p3) + " " + String(Throttle+r4+p4));
    Serial.print(Throttle+r1+p1); Serial.print("  ");
    Serial.print(Throttle+r2+p2); Serial.print("  ");
    Serial.print(Throttle+r3+p3); Serial.print("  ");
    Serial.print(Throttle+r4+p4); Serial.print("  ");
    Serial.print(rollSetpoint) ; Serial.print(" ");
    Serial.print(pitchSetpoint) ; Serial.print(" ");
    Serial.print(ROLL) ; Serial.print(" ");
    Serial.print(PITCH) ; Serial.print(" ");
    // Serial.print(abs(PITCH-pitchSetpoint)*MAX_OFFSET/90); Serial.print(" ");
    Serial.println();
}

// set the motors here (in RPM)
motor.setSpeedRPM(1, Throttle+r1+p1);
motor.setSpeedRPM(2, Throttle+r2+p2);
motor.setSpeedRPM(3, Throttle+r3+p3);
motor.setSpeedRPM(4, Throttle+r4+p4);

    ControlTimer = 0;
}

// Control::update_setPoints()
void Control::resetRoll(){
    roll_adjust = 0;
    r1 = 0;
    r2 = 0;
    r3 = 0;
    r4 = 0;
}
void Control::resetPitch()
{
    pitch_adjust = 0;
    p1 = 0;
    p2 = 0;
    p3 = 0;
    p4 = 0;
}

void Control::moveDrone(int _movement)
{
    switch(_movement)
    {
    case MOVE_LEFT:
        Serial.println("moving left");
        r1 = - roll_adjust;
        r2 = - roll_adjust;
        r3 = roll_adjust;
        r4 = roll_adjust;
        break;
    case MOVE_RIGHT:
        Serial.println("moving right");
        r1 = roll_adjust;
        r2 = roll_adjust;
        r3 = - roll_adjust;
        r4 = - roll_adjust;
        break;
    case MOVE_FORWARD:
        Serial.println("moving forward");
        p1 = - pitch_adjust;
        p2 = pitch_adjust;
        p3 = pitch_adjust;
        p4 = - pitch_adjust;
        break;
    case MOVE_BACKWARD:
    {
        Serial.println("moving backward");
        p1 = pitch_adjust;
        p2 = - pitch_adjust;
        p3 = - pitch_adjust;
        p4 = pitch_adjust;
        break;
    }
    default:
        break;
    }
}
APPENDIX S: HTML

<html>
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, user-scalable=no, minimum-scale=1.0, maximum-scale=1.0">
<link href="joystick test css.css" rel="stylesheet">
</head>
<body>
<div id="container"></div>
<div id="info">
  DRONE GUI
  Vlad Cretu, CAL POLY POMONA
  Touch the screen and move
  -
  works with mouse too as debug
  <br/>
  <span id="result"></span>
</div>
<div>
  <span id="throttle"></span>
</div>
<div>
  <span id="dx">0</span>
</div>
<div>
  <span id="dy">0</span>
</div>

<!-- JavaScript Libraries -->
<script src="lib/jquery/jquery.min.js"></script>
<script src="lib/jquery/jquery-migrate.min.js"></script>
<script src="js/main.js"></script>
<script src="js/virtualjoystick.js"></script>
<script src="joystick test js.js"></script>
</body>
</html>
APPENDIX T: CSS

body {
  overflow : hidden;
  padding : 0;
  margin : 0;
  background-color: #BBB;
}

#info {
  position : absolute;
  top : 0px;
  width : 100%;
  padding : 5px;
  text-align : center;
}

#info a {
  color : #66F;
  text-decoration : none;
}

#info a:hover {
  text-decoration : underline;
}

#container {
  width : 100%;
  height : 100%;
  overflow : hidden;
  padding : 0;
  margin : 0;
  -webkit-user-select : none;
  -moz-user-select : none;
  -webkit-user-select : none;
  -moz-user-select : none;
}
APPENDIX U: JavaScript

// Drone WIFI Comunications
// Author: Vlad Cretu, California State University, Pomona
// Date: 01/13/2020
// Description: This is a section of javascript code use to communicate with an
// hosts a web server. The code has three mains routes of communication.

// 1) Status ID List - Web page sends an HTTP GET request to web server asking for
// the resource "elementIDs.ajax"
// to get the element IDs that the webserver wants to update later.It expects a series
// of comma seperate values that
// exactly match the element IDs of the html file for the web page.
//
// 2) Status Value List - Web page sends an HTTP GET request asking for the
// resource "elementValues.ajax"
// The webserver is expected to respond with a series of comma seperated values
// that will be used
// to update the corresponding element IDs from (1). Note that order must excelely
// match that used in (1).
//
// 3) Sending Commands - Web page sends an HTTP GET request askingfor the
// resource "elementValues.ajax"
// followed by a query string with a command encoded. The qeury string will have
// the following format
// ?cmd=<yourCommandGoesHere>&_=<yourTimestampGoesHere>
// Web server should process command, and the respond with by updating the GUI
// with element values to confirm
// if the command was executed or not

/*-------------------------------------------------------------------------------
Input Parameters
/*-------------------------------------------------------------------------------

// Input Parameters
var statusRequestPeriod = 1000;       // Time to wait in milliseconds before sending a
// new status request (25 and 50 when we fly)
var ajaxRequestTimeout  = 2000;       // Time in milliseconds since request sent
// before abandoning request
//var instrumentIP = "169.254.125.1";   // IP address of instruement to communicate
// with
var instrumentIP = "169.254.125.60";  // IP address of instruement to communicate
// with
//var instrumentIP = "169.254.125.80";  // alternate IP address for the instrument.
// Debugging Options
var debugAJAX = {
    printIncomingData: true,           // Print all data received by web page
    printPageRefreshConfirmation: false,        // Print when a page element update
    printOutgoingData: true,             // Print all data transmitted by web page
}

// Route GUI interaction to functions
//

// Controls the Joysticks for Drone Control
//
console.log("touchscreen is", VirtualJoystick.touchScreenAvailable() ? "available" : "not available");

var joystick  = new VirtualJoystick({
    container : document.getElementById('container'),
    mouseSupport : true,
   .strokeStyle : 'red',
    limitStickTravel: true,
    stickRadius : 200
});

joystick.addEventListener('touchStartValidation', function(){
    //console.log('touching')
    var touch = event.changedTouches[0];
    if( touch.pageX < window.innerWidth/2 ) return false;
    return true
})

//joystick.addEventListener('touchEnd', function(){
//    //console.log('not touching')
//})

var joystick2 = new VirtualJoystick({
    container : document.getElementById('container'),
    mouseSupport : true,
    }
strokeStyle: 'orange',
limitStickTravel: true,
stickRadius: 200
}

joystick2.addEventListener("touchStartValidation", function() {
  var touch = event.changedTouches[0];
  if (touch.pageX >= window.innerWidth/2) return false;
  return true
})

// send this information to our ESP32 every 100ms (initially, this is subject to change)
setInterval(function() {
  sendCommand("TXY^"+Math.round(joystick2.deltaY())+":"+Math.round(joystick.deltaX())+":"+Math.round(joystick.deltaY())+"!");
  //console.log(Math.round(joystick2.deltaY()));
}, 100);

// joystick2.addEventListener('touchEnd', function(){
//})

// prevent page zooming. important for 2 different joysticks.
//document.addEventListener("touchstart", function(e){
//  e.preventDefault();
//}, {passive: false});

// Add jQuery code here to handle GUI interactions (e.g. button clicks, for submissions), to get the data from it,
// and then initiate a command

// Whenever a button of the switch class is clicked
$('.switch').on('click', function(event) {
  event.preventDefault();

  // Read the current state of the switch
  var switchObject = this.childNodes[1];
```javascript
var id = switchObject.id;
var value = switchObject.checked; // Read state of switch before click

// Reset switch to initial state (since we only want this to change when the web server
// says it has actually changed)
switchObject.checked = value;

// Generate and send command
value = !value; // Get the value the user intends to send
command = id+"_"+(value? 1 : 0); // Create command string
sendCommand(command); // Send Command to instrument
}

// Handles any html form submissions
$('form').on('submit', function(event){
  event.preventDefault();
  var id = event.currentTarget[0].id;
  var value = event.currentTarget[0].value;
  event.currentTarget[0].value = null; // Clear current value in box to display
  submission results
  sendCommand(id+"_"+value);
});

//handles the range slider input. Whenever the range slider is dragged.
$('.slidecontainer').on('change', function(event){
  event.preventDefault();

  var rangeObject = this.childNodes[1];
  var id = rangeObject.id;
  var value = rangeObject.value; // Read state of slide before click
  sendCommand(id+"_"+value);
});

// Update GUI with information from Drone
//setting the date at the top:
//document.getElementById(dateholder).write(Date());

// -- Add code here to update each element of the web page as you want
// Should consist of a series of if/else statements to determine
// what type of element you are dealing with and how you want to handle it
function updateGuiElement(elementID, elementValues){
  updateLDstatus(elementID, elementValues);
}
```

//console.log(elementID)
//sendCommand("Throttle"+_"+joystick2.deltaY());
//sendCommand("dx"+_"+joystick.deltaX());
//sendCommand("dy"+_"+joystick.deltaY());

if(elementID.indexOf("PoweR")>-1){
    //document.getElementById(elementID+"feedback").innerHTML = elementValues;
}
else{
    //console.log(elementID);
    //document.getElementById(elementID).checked = parseInt(elementValues);
}

//method to write the date and time on the screen
//document.write(Date());


Refresh dynamic data elements of web page with object data
function refreshPageData(elementIDs, elementValues){

    // Make sure both datasets are consistent
    if(elementIDs.length != elementValues.length){
        alert("Error: Status list and data have different lengths("+elementIDs.length+" vs "+elementValues.length+")");
        requestStatusIDList(); // Request new status list in case it changed
        return false;
    }

    // Loop through IDs and update
    for(i = 0; i<elementIDs.length ; i++){
        if(debugAJAX.printPageRefreshConfirmation){
            console.log(elementIDs);
        }
        updateGuiElement(elementIDs[i], elementValues[i])
    }
}
// Handle Communications between GUI and instrument

// Three different functions which each handle a different type of information exchange between the instrument

// --------- Generate formatted URL from user entered IP ---------
// Function looks at the IP address of the webserver entered above and formats it into a proper URL that jQuery can interpret (if not already)

function generateServerURL(serverIPAddress)
{
    var serverURL = serverIPAddress;
    if(serverURL.indexOf("http://")<0)
    {
        serverURL = "http://"+serverURL;
    }
    if(serverURL.substring(serverURL.length-1)!="/")
    {
        serverURL+="/";
    }
    return serverURL;
}

// --------- Ask Instrument of ID Lists ---------
// Function sends a request of element ID list of values to update in the GUI.
// The data is expected to be in the format of element IDs separated by commas
// Note that it expects a comma at the very end as well. The ID list should be in the same order as the value list sent later

var statusIDList;

function requestStatusIDList()
{
    // Using the core $.ajax() method
    $.ajax(
    {
        url: generateServerURL(instrumentIP)+"elementIDs",
        timeout: ajaxRequestTimeout, // in milliseconds
        type: "GET",
        dataType : "text",
        crossDomain: true,
    })
    .done(function(returnedText)
    {
        if(debugAJAX.printIncomingData)
        {
            console.log("Processing ID Response -- Received Data is Shown Below");
            console.log(returnedText);
        }
    })
returnedText = returnedText.slice(0, returnedText.lastIndexOf(","));
statusIDList = returnedText.split(",");
setTimeout(requestStatusValueList, statusRequestPeriod);  // Start status requests
}

// Code to run if the request fails; the raw request and status codes are passed to the function
.fail(function( xhr, status, errorThrown ) {
    if(debugAJAX.printIncomingData)   {console.log( "The AJAX request failed" );}
    setTimeout(requestStatusIDList, statusRequestPeriod);  // Try again later
});

// Get list numeric status data; the data will update the element IDs listed in requestStatusIDList()
function requestStatusValueList(){

    // Using the core $.ajax() method
    $.ajax(
        url: generateServerURL(instrumentIP)+"elementValues.ajax",
        timeout: ajaxRequestTimeout,  // in milliseconds
        type: "GET",
        dataType : "text",
    );

    // Code to run if the request succeeds (is done); The response is passed to the function
    .done(function(returnedText)
    {
        if(debugAJAX.printIncomingData){
            console.log("Processing Val Response -- Received Data is Shown Below");
            console.log(returnedText);
        }
        returnedText = returnedText.slice(0, returnedText.lastIndexOf(","));
        statusValues = returnedText.split(",");
        displayConnectionStatus(true);
        refreshPageData(statusIDList, statusValues);
    });

    // Code to run if the request fails; the raw request and status codes are passed to the function
    .fail(function( xhr, status, errorThrown ) {
        if(debugAJAX.printIncomingData)   {console.log( "The AJAX request failed" );}
        displayConnectionStatus(false);
    });
// Code to run regardless of success or failure;
.always(function( xhr, status ) {
    setTimeout(requestStatusValueList, statusRequestPeriod);
});

// Take command string generate by web, add to url with time stamp
function sendCommand(commandString){

    // Using the core $.ajax() method
    $.ajax{
        url: generateServerURL(instrumentIP)+"cmd.ajax",
        timeout: ajaxRequestTimeout, // in milliseconds
        cache: false,
        data: "cmd="+commandString,
        type: "GET",
        dataType : "text",
        crossDomain: true,
    });

    // Code to run if the request succeeds (is done); The response is passed to the
    function
    .done(function(returnedText){
        returnedText = returnedText.slice(0, returnedText.lastIndexOf(","));
        statusValues = returnedText.split(",");
        if(debugAJAX.printOutgoingData) {console.log("Command: "+commandString);}
        //displayConnectionStatus(true);
        refreshPageData(statusIDList, statusValues);
    })

    // Code to run if the request fails; the raw request and status codes are passed to the
    function
    .fail(function( xhr, status, errorThrown ) {
        if(debugAJAX.printIncomingData)  {console.log( "The AJAX request failed" );}
        connectionStatus = false;
        //displayConnectionStatus(false);
    });
}

// Start communications
//displayConnectionStatus(false);
requestStatusIDList();