Auto Drone: Modifying a DIY Drone Kit for Autonomous Flight

Sam Ginsberg
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/mems500

Recommended Citation
https://openscholarship.wustl.edu/mems500/138

This Final Report is brought to you for free and open access by the Mechanical Engineering & Materials Science at Washington University Open Scholarship. It has been accepted for inclusion in Mechanical Engineering and Materials Science Independent Study by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.
Auto Drone: Modifying a DIY Drone Kit for Autonomous Flight

Author: Sam Ginsberg
Advisor: Guy Genin (not an author of this report)
Abstract:

With the addition of an Arduino microprocessor, a basic drone build was modified to obtain autonomous flight. By wiring Arduino and receiver in series with each other, the Arduino draws power from the main drone battery. The signal from the receiver is intercepted by the Arduino and a manually coded pulse position modulation signal is sent out to the flight controller. Through experimental testing using Betaflight configurator, it was found that the PPM signal requires 35000 µs of end pause between two frames of data. Any less, lead to unsteady errors in the flight controller’s ability to read the coded signal. By adding more time between frames there would be a decrease in performance caused by additional input delay and latency in the internal commands reaching the flight controller. Aux switches were utilized on the transmitter to code a manual to automatic switch, such that a drone pilot can quickly and easily transition in and out of the coded autonomous flight mode. The flight mode coded is based on feedback control using a GPS module as the primary sensor. The flight mission consists of keeping the drone at a level altitude with an additional switch on the transmitter that adds 5 ft to the altitude. A drone pilot could keep the drone in place without requiring constant input into the transmitter and simply flip a switch back and forth to raise then lower the drone’s desired altitude. When coding in a specific error from the expected value in the control loop, the throttle, pitch, roll, and yaw values recorded in Betaflight corresponded to the coded correction model. The GPS module altitude data proved too imprecise in testing to do flight testing outdoors. With more time, a barometer altimeter could be added to replace the GPS module as the sensor in the feedback loop. With the success of the core code and wiring of the Arduino, new flight missions can be coded and simply inserted into the existing Arduino code.
**Introduction:**
Through this report I will discuss how an Arduino microprocessor was successfully coded and wired into a drone to intercept normal flight signals from the drone pilot and send a manually coded signal. With this ability an autonomous flight mission was coded and digitally tested. Hundreds of iterations of code resulted in the code as seen in Appendix A. The final test of the code can be seen in Figure 1.

![Figure 1: Final Digital Test of Autonomous Code](image)

As seen in the graph above, initially the throttle, pitch, yaw, and roll values are steady and non-zero. With the manual flip of a switch on the transmitter to position two, all the values besides throttle return to normal. When flipping the switch to the third position the throttle increases by 100. The switch can go in between positions two and three and the throttle will move up and down by 100 with it. Finally, when the switch returns to position one, the other values are unblocked and are free to move to different values from normal.

Microscale unmanned aerial vehicles, or drones, are increasingly prevalent in multiple areas of our society. The size and cost allow for anybody to either purchase or build their own drones to fly. By modifying a basic drone building kit with a microcontroller, the drone can receive flight instructions from either the pilot or the microcontroller on board. The goal of this project is to implement code and wiring that would allow anyone to simply plug in the microcontroller to their drone and fly with an autonomous flight mode. These drones are often used for photography or film and this technology would allow for people to capture aerial footage without requiring a drone pilot. For example, if an individual wanted to record themselves walking, they could use the autonomous flight mode to hover the drone and slowly move along the walking path. I will be coding a simple autonomous mission that will keep the drone hovering with an option to elevate the hover by a given height.

In order to implement a coded mission, the flight dynamics of drones must be evaluated. I am working with quad rotary wing drones due to their ability to hover and fly in any direction,
expanding the possibilities of future autonomous missions. Unlike fixed wing drones that use motors to generate horizontal thrust and airfoils to generate lift, rotary wing drones such as quadcopters generate all their lift from vertical motors. Each of the four motors also generate a moment normal to the drone body. A free body diagram of a quadcopter can be seen in Figure 2 below.

![Quadcopter Free Body Diagram with roll (φ), pitch (θ), and yaw (ψ) angles and unit axes](image)

**Figure 2: Quadcopter Free Body Diagram with roll (φ), pitch (θ), and yaw (ψ) angles and unit axes [2].**

There are two coordinate systems of unit axes in Figure 2. The NED subscripts refer to the global North, East, and Down directions, while the $x_B$, $y_B$, and $z_B$ refer to the axes relative to the body of the drone. Lastly $e_V$ is the unit vector in the direction of the drone velocity [2]. Each motor labels 1 through 4 generates a thrust $T$ in the negative $z_B$ direction and a moment parallel to this axis. Motors 1 and 3 generate counterclockwise moments, while motors 2 and 4 generate clockwise moments. The opposite motor directions cause these moments to cancel each other out when all four thrusts are equal keeping the drone from rotating about the $z_B$ axis. By using differential thrust in these four motors the drone is able to control its roll, pitch, and yaw. For example, decreasing $T_1$ and $T_3$ and increasing $T_2$ and $T_4$, the net thrust is constant but the increased moments in the clockwise direction cause the drone to yaw. For the drone to be in vertical equilibrium, the thrust in the global down direction needs to be equal to the weight. Therefore, the thrust required to hover the drone is less than that of the thrust needed when pitched.
The microcontroller needs to be wired such that the signal sent from the remote-control transmitter is intercepted by the microcontroller. Without altering the drone, the normal signal path can be seen in Figure 3.

Figure 3: Signal Path for RC Drone

The transmitter is the physical controller the pilot uses to fly the drone. This transmits a signal to the receiver on the drone body, which in turn passes the flight instructions to the flight controller, the brain of the drone. The receiver gives values for throttle, pitch, roll, yaw, and a number of auxiliary values depending on the transmitter. The flight controller takes these values and alters the speed of each motor to fulfil the given input commands. I will insert a microcontroller between the receiver and flight controller so all of the flight commands the flight controller reads will come from the microcontroller. In order for the flight controller to properly read the commands, the coded signal needs to mimic the normal receiver signal perfectly. I will be using a pulse position modulation, or PPM, signal due to the simplicity of it. PPM signals consist of multiple pulses of current of consistent length at various positions that encode multiple channels of information. The length between leading edges of two pulses correspond to one channel. Therefore, for N channels of information, there is a set of N + 1 pulses [3]. The transmitter I will be using has six channels and therefore there will be seven pulses per transfer of flight commands. At the end of the last pulse, there is an extended dead time such that the device knows that the next pulse is the start of a new pulse group. A sample set of pulses for six channels can be seen in Figure 4 below.
The blue lines below the graph of pulses indicate what would be the recorded time lengths for each of the 6 channels. The pulse lengths in most RC drones are 500 $\mu s$ with minimum and maximum pause lengths of 500 $\mu s$ and 1500 $\mu s$ respectively. The minimum channel length is therefore 1000 $\mu s$ and the maximum is 1500 $\mu s$. There is no uniform space between the end and start of pulse groups. The core function of the microcontroller will be to read the PPM signal from the receiver and generate a PPM signal to send to the flight controller. By adding more code, the autonomous flight mode and manual to automatic switch will be added.

In order to code and fly the autonomous mission, a feedback control system will be implemented. The core function of these systems is to achieve a desired output based on measured outputs [1]. There are three main components to these control systems, a controller, sensor, and system. The sensor measures the output of the system and relays it back to the controller, which in turn changes the input to the system based on these results [1]. For my autonomous flight mission, the microcontroller will be the controller, the drone itself is the system, and the sensor will be a GPS module. With a desired altitude, based on the measured altitude from the GPS, the controller will change the input throttle value for the drone.

**Experimental Methods:**

The equipment used in constructing and modifying the drone can be seen in Table 1 below. The drone components were purchased together in the Tyro 119 kit manufactured by Eachine. Throughout this project safety precautions were taken while building and testing the drone components. Drones require high voltage lithium polymer batteries that can ignite if short circuited.
### Table 1  Equipment Used in Building and Modifying Drone

<table>
<thead>
<tr>
<th>Equipment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>250 mm Racing Frame Kit</td>
<td>Carbon Fiber Drone Frame</td>
</tr>
<tr>
<td>Eachine 2407 1850KV Motor</td>
<td>2 CW and 2 CCW Motors</td>
</tr>
<tr>
<td>Eachine GPS F4 Flight Controller</td>
<td>Drone Flight Controller</td>
</tr>
<tr>
<td>Eachine 40A 4In1 BLHeli_S ESC</td>
<td>Electronic Speed Controller and Power Distribution Board</td>
</tr>
<tr>
<td>DALProp TJ6045 3-blade propeller</td>
<td>2 CW and 2 CCW Propellers</td>
</tr>
<tr>
<td>5.8G 40CH 0/25/200/600mW VTX</td>
<td>Video Transmitter</td>
</tr>
<tr>
<td>Caddx.us Turbo f2 Camera</td>
<td>First Person View Camera</td>
</tr>
<tr>
<td>BN-220T GPS Module</td>
<td>GPS Module</td>
</tr>
<tr>
<td>URUAV 22.2V 1300mAh 100C 6S Lipo battery</td>
<td>Drone Battery</td>
</tr>
<tr>
<td>Flysky FS-i6X 6CH 2.4GHz</td>
<td>Transmitter</td>
</tr>
<tr>
<td>Flysky FS-iA6B</td>
<td>Receiver</td>
</tr>
<tr>
<td>WYCTIN 40 Tin Lead Rosin Core Solder Wire</td>
<td>Solder Wire</td>
</tr>
<tr>
<td>Weller WLC100 40-Watt</td>
<td>Soldering Iron</td>
</tr>
<tr>
<td>Etekcity Digital Multimeter</td>
<td>Digital Multimeter</td>
</tr>
<tr>
<td>Heat Shrink Tubing</td>
<td>Heat Shrink Tubing</td>
</tr>
<tr>
<td>Arduino Uno Rev3</td>
<td>Microcontroller</td>
</tr>
<tr>
<td>Breadboard Jumper Wires</td>
<td>Male-Male and Female-Male Wires</td>
</tr>
<tr>
<td>Wire Strippers</td>
<td>Wire Strippers</td>
</tr>
<tr>
<td>Betaflight Configurator</td>
<td>Computer Program for Drone Configuration</td>
</tr>
<tr>
<td>Arduino-2</td>
<td>Arduino Coding Program</td>
</tr>
<tr>
<td>Safety Glasses and Fan</td>
<td>Safety Equipment for Soldering</td>
</tr>
</tbody>
</table>

The first step in construction of the drone is soldering the battery lead and motor wires to the ESC and power distribution board. The battery leads connect red to positive and black to negative. The clockwise motors are connected at position 1 and 4 labeled on the ESC and the other motors take the other diagonal. It doesn’t matter which order the three motor wires are attached because when testing motor spin direction, they can be easily changed. While soldering safety glasses were worn at all times and a fan was running to disperse fumes. Figure 5 below shows the completed soldering for the board placed on the base of the carbon fiber frame.
The next step is to stack the flight controller on top of the ESC attaching the two with the wires provided with the boards. The flight controller has wire attachments for the camera, receiver, GPS, and video transmitter. The video transmitter and camera are not used in the autonomous flight but are still connected for the manual flight mode if the pilot requires first person view. Figure 6 shows the completed unmodified drone build.

The receiver wires can be seen on the left of Figure 6 leaving the drone body resting next to the battery connection. The three wires are battery, ground, and signal (red, black, and yellow respectively). When plugging in the battery the voltage given to the receiver can be measured using a digital multimeter. The digital multimeter should be used to test for short circuits before connecting the battery. The capacitor connected to the battery pads seen in Figure 5 became detached in my initial build which caused the battery to fry the ESC and one of the motors. Because of this incident, I had to acquire new hardware and rebuild the main drone body. In the
new build additional solder was added to reinforce the capacitor connection. The receiver connection has a measured voltage of 5V. The Arduino board also runs on 5V so it is wired in series with the receiver so they both receive the required power to work. The additional wire connectors soldered together to connect all three devices can be seen in Figure 7 below.

![Ground and Battery Wires for Series Connection of Receiver and Arduino](image)

**Figure 7: Ground and Battery Wires for Series Connection of Receiver and Arduino**

The wires have male ends to connect to the flight controller and one male and one female end for the Arduino and receiver respectively. The signal wire runs from the receiver into the Arduino and an additional signal wire runs from the Arduino to the flight controller. Regardless of manual or autonomous flight the signal to the flight controller comes directly from the Arduino.

The GPS connector consists of two signal wires and two for power. The signal wires were cut, and new male end attachments were added such that the power still comes from the flight controller, but the signal can be sent into the Arduino for feedback control. With these two wiring modifications the microcontroller is able to receive power from the drone battery, data from the GPS module, signals from the receiver, and send signals to the flight controller. The drone with modifications can be seen in Figures 7.
The green and yellow cut wires on the left of Figure 8 are the original signal wires for the GPS which now go into the Arduino. The wires soldered together to run in series in Figure 7 can be seen implemented in Figure 8 as well. With the wiring completed a safe method of testing needs to be implemented. For all of the autonomous code testing Betaflight Configurator is used. This program is used to get drones ready to fly and program the flight controller. They key function to test the implementation of the Arduino is the Receiver page. Here sliders for each receiver channel and a graph of all the values can be seen. With this the live flight commands the Arduino is sending can be seen and verified. Figure 9 A and B show sliders and graph used in testing from Betaflight.

![Arduino Wiring Modifications for Autonomous Flight Control System](image)

**Figure 8: Arduino Wiring Modifications for Autonomous Flight Control System**

**Figure 9A: Channel Sliders on Betaflight Configurator**
Figure 9B: Channel Graph on Betaflight Configurator

Note the colors in Figure 9B match with the slider colors on Figure 9A. The stacking of Roll, Pitch, Yaw and the Aux channels are causing a singular brown line. The Y axis of the graph corresponds to the channel value read and the X axis is time in ms. With these two methods the direct input the flight controller reads can be visualized without actually flying the drone. Without testing in this way, in a case of a coding error where the throttle goes to max value when turned on, the drone would potentially fly away uncontrollably. In the Arduino code two libraries were downloaded to assist reading the incoming PPM signal and GPS signals. These libraries can be seen in Appendix B.

Results and Discussion:

To generate the PPM signal a digital pin on the Arduino is set to fluctuate from HIGH to LOW in the required timing. The first iteration of the code worked writing a Pulse() function, which set the pin output to HIGH, waited for 500 $\mu s$, and set the output back to LOW. Delta timing is used to accomplish the precise timing of a pulse. The C++ function micros() returns the current time in microseconds since the Arduino started running the program. By saving the time in microseconds, a while loop can be generated that loops until the difference between micros() and the saved start time is 500 $\mu s$. A second function genPPM() was written to space out the pulses with the proper timing. Here a for loop is coded that goes through each pulse and waits, using the same method as described above, based on the desired channel value. Lastly an extended wait is at the end to signal a new frame of data. Using Betaflight it could be seen that values on the sliders were around expected but were significantly unsteady. The values were fluctuating by upwards of $\pm 100 \mu s$. The Pulse() function was eliminated and the code was inserted into genPPM() when needed to minimize time spent jumping between functions in the code. This worked at minimum values for each channel but when all six were set to the max
value, 2000, channel 6 would have greater error. The function was altered once again to set the current time micros() directly before every waiting period. This way error wouldn’t build through the pulses and more negatively affect the last few channels. With this combination the values for all six channels only fluctuate by less than 10\(\mu s\), which is not perfect but small enough to have negligible effects while flying. The cause of this is likely small fractions of time it takes to run lines of code and imperfections in the micros() function itself. The resolution of micros() is 4 \(\mu s\) so depending on the start time of the pauses an additional few microseconds could be added or subtracted, leading to the slight unsteadiness of the genPPM() function.

In order to maximize the effectiveness of the genPPM() function, the pause at the end of the frame needs to be minimized. While longer pauses may lead to potentially more steady values, there will be an increased latency between when commands are sent from the and when they are received by the flight controller. At a given pause time, the maximum unsteady spike values are recorded from using the graph as seen in Figure 9B. Figure 10 shows the test for 17500 \(\mu s\) pause time.

![Figure 10: 17500 \(\mu s\) Pause Time Graphical Test Results: Steady Condition](image)

The maximum deviation is 200 under steady conditions for this case. Steady conditions are when the transmitter is in normal positions. Roll, Yaw, and Pitch should be at 1500, Throttle and the two Aux channels should be at 1000. These Aux channels correspond to switches on the transmitter. The two Aux channels and the Yaw fluctuate the most in this condition. Surprisingly the Throttle which appears in between Yaw and Aux 1 in Figure 9A is much steadier. In further investigation it can be seen that the channels 3 and 4 are switched in the sliders on Betaflight, meaning the three unsteady channels are the last three in the frame. Because error appears to be building even with the additional micros() calls as described above. A maximum condition is also tested where the most unsteady condition is found. Here all values are set to 2000 except
Aux 1 which is at 1000. 10 shows the results for the maximum test condition for the same pause length.

![Graphical Test Results: Maximum Condition](image1)

**Figure 11: 17500 µs Pause Time Graphical Test Results: Maximum Condition**

This test condition was found through trial and error and as seen in Figure 11 produces drastically unsteady results when compared to the condition in Figure 10. The maximum test condition yields very interesting results at certain pause values. Figure 12 shows the maximum test condition for a 20 ms pause.

![Graphical Test Results: Maximum Condition](image2)

**Figure 12: 20000 µs Pause Time Graphical Test Results: Maximum Condition**

Besides the small spike the values are steady enough to fly. However, they are not representing the desired condition. Aux 1 is at 1300 when it should be at 1000, Aux 2 is at 1000 when it should be at 2000. Roll, Pitch, and Yaw are near 1700 when they should all be at 2000. Table 2 contains the test results for all pause times under both conditions with a note about potential irregularities in the second testing condition as seen in Figure 12.
Table 2  
Pause Length Fluctuation Results

<table>
<thead>
<tr>
<th>Pause Time (µs)</th>
<th>Fluctuation Steady Condition</th>
<th>Fluctuation Maximum Condition</th>
<th>Note: Maximum Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>15000</td>
<td>400</td>
<td>500</td>
<td>Completely Unsteady</td>
</tr>
<tr>
<td>17500</td>
<td>200</td>
<td>500</td>
<td>Completely Unsteady</td>
</tr>
<tr>
<td>20000</td>
<td>100</td>
<td>500</td>
<td>Steady at Wrong Values</td>
</tr>
<tr>
<td>22500</td>
<td>100</td>
<td>500</td>
<td>All but Aux 2 Steady</td>
</tr>
<tr>
<td>25000</td>
<td>20</td>
<td>500</td>
<td>Steady with Large Spikes Roughly every 50 ms</td>
</tr>
<tr>
<td>27500</td>
<td>10</td>
<td>500</td>
<td>All but Aux 2 Steady</td>
</tr>
<tr>
<td>30000</td>
<td>10</td>
<td>500</td>
<td>All but Aux 2 Steady</td>
</tr>
<tr>
<td>32500</td>
<td>10</td>
<td>500</td>
<td>All but Aux 2 Steady</td>
</tr>
<tr>
<td>35000</td>
<td>10</td>
<td>10</td>
<td>Steady</td>
</tr>
<tr>
<td>37500</td>
<td>10</td>
<td>10</td>
<td>Steady</td>
</tr>
<tr>
<td>40000</td>
<td>10</td>
<td>10</td>
<td>Steady</td>
</tr>
</tbody>
</table>

At 35000 µs, the values for all channels in the PPM signal remain steady and as expected regardless of the condition they are put into. This is reassured in the next two pause times as well, confirming that the threshold pause is less than 35000. By using the two libraries in Appendix B to help read the input signals from the GPS and receiver, enough information is obtained to code the autonomous flight mode and control switch. The two Aux channels are set to two switches on the transmitter. Aux 2 is either at 1000 or 2000 and is used to arm the drone for flight as an additional safety precaution. Aux 1 is used to control the automatic and manual flight and has three positions, 1000, 1500, and 2000. The main iterative section of the Arduino code (void loop()), reads the PPM signal coming in from the receiver, consists of a series of logic statements, and finally calls the genPPM() function. The reader encodes an array that consist of the channel values of the incoming PPM signal and these values are either altered or unchanged based on the logic statements. Finally, genPPM() creates the PPM signal for the flight controller based on these array values. The first logic statement creates the manual to automatic switch in the code. If the Aux 1 switch is above 1200 and a number levelThrot is equal to zero, the drone enters the automatic mode and levelThrot is set equal to the current throttle input and the altitude is recorded from the GPS. If this condition is not met, another statement is activated if Aux 2 is less than 1200 and the levelThrot is not equal to zero. Here the recorded altitude and throttle are returned to zero and the drone exits the autonomous flight mode. Lastly, there is one more if
statement that tests if the current levelThrot value is not equal to zero. If it is not equal to zero, the drone is in the automatic mode and if in manual it will be skipped over. This is where the autonomous code is inserted and can be altered based on the mission.

The simple autonomous mission I am testing consists of locking the Yaw, Roll, and Pitch values such that the drone will stay level. The flight controller has a built in self leveling mode so keeping the values for these three channels at 1500, even with mild unsteady values, should keep the drone in the same position. The Throttle is much more sensitive for keeping the drone level so this value will be altered based on the altitude data. For this flight mission the pilot should hover the drone prior to switching into the autonomous mode. When initially switching into this mode, the current throttle and altitude are recorded. The throttle value is a guess for the exact level throttle. The throttle will increase or decrease based on the current altitude measurements. When shifting the Aux 1 channel to 2000, the recorded level altitude is increased by a desired height. With this, a pilot could flip the switch to have the drone autonomously hover, flip the switch further to raise the drone by a desired height and return back to the middle switch to lower the hover to the initial level height. Two potential correction methods based on the difference in altitude can be seen in Figure 13 below.

![Figure 13: Correction Methods for Keeping Level Altitude](image)

The two correction slopes have a maximum absolute value of 100. This keeps the drone from accelerating too rapidly given large altitude differences. To maximize the flight efficiency an experimental value for the maximum value and which of the two correction methods needs to be found by flight testing. Before flight testing, the GPS module needs to be examined. While
sitting outside for maximum satellite coverage, the GPS collected altitude data while stationary. The printed GPS data can be seen in Table 3 below.

### Table 3  GPS Module Recorded Data

<table>
<thead>
<tr>
<th>Sats</th>
<th>HDOP</th>
<th>Latitude (deg)</th>
<th>Longitude (deg)</th>
<th>Fix Age</th>
<th>Date (YYYY-MM-DD)</th>
<th>Time (HH:MM:SS)</th>
<th>Altitude (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688415</td>
<td>-73.994529</td>
<td>431</td>
<td>12/03/2020</td>
<td>21:40:18:561</td>
<td>31.90</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994529</td>
<td>536</td>
<td>12/03/2020</td>
<td>21:40:19:652</td>
<td>32.60</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994529</td>
<td>613</td>
<td>12/03/2020</td>
<td>21:40:20:729</td>
<td>33.00</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994529</td>
<td>690</td>
<td>12/03/2020</td>
<td>21:40:21:805</td>
<td>33.50</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994537</td>
<td>765</td>
<td>12/03/2020</td>
<td>21:40:22:881</td>
<td>33.60</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994537</td>
<td>6</td>
<td>12/03/2020</td>
<td>21:40:24:20</td>
<td>34.10</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994537</td>
<td>86</td>
<td>12/03/2020</td>
<td>21:40:25:216</td>
<td>34.40</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994537</td>
<td>255</td>
<td>12/03/2020</td>
<td>21:40:26:386</td>
<td>34.30</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994537</td>
<td>423</td>
<td>12/03/2020</td>
<td>21:40:27:553</td>
<td>34.30</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994537</td>
<td>537</td>
<td>12/03/2020</td>
<td>21:40:28:653</td>
<td>34.10</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994529</td>
<td>615</td>
<td>12/03/2020</td>
<td>21:40:29:731</td>
<td>35.90</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688419</td>
<td>-73.994529</td>
<td>691</td>
<td>12/03/2020</td>
<td>21:40:30:806</td>
<td>33.50</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688423</td>
<td>-73.994529</td>
<td>765</td>
<td>12/03/2020</td>
<td>21:40:31:881</td>
<td>33.70</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688423</td>
<td>-73.994529</td>
<td>7</td>
<td>12/03/2020</td>
<td>21:40:33:21</td>
<td>33.80</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688426</td>
<td>-73.994529</td>
<td>85</td>
<td>12/03/2020</td>
<td>21:40:34:214</td>
<td>34.40</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>40.688426</td>
<td>-73.994529</td>
<td>256</td>
<td>12/03/2020</td>
<td>21:40:35:384</td>
<td>34.40</td>
</tr>
</tbody>
</table>

With a consistent number of satellites in view, the GPS recorded an altitude ranged from 31.9 meters to 34.4 meters of 16 seconds. The GPS altitude data is far too inconsistent to accurately level. The autonomous flight mission coded with the linear correction and a 5 ft increased hover with the third switch position was tested on Betaflight and can be seen in Appendix A. While the switch is in manual position the sliders react in real time to the physical flight inputs. When flipping the switch to the autonomous mode the Throttle stays constant and the Roll, Pitch, and Yaw return to 1500 regardless of current value. From this point while the switch is in the middle position any manual input change has no effect on the sliders, confirming the manual flight mode has been completely disconnected. When switching the switch to the max position, with no accurate GPS altitude the Arduino adds a full 5ft and accordingly the Throttle increases by 100. Returning to the middle the Throttle drops by 100 and when returning to the first position the sliders respond as they do in normal manual flight.

The Arduino’s ability to generate a PPM signal was accurately tested and proved effective. The manual to automatic switch was also successful in the code. Unfortunately, the GPS altitude data is inaccurate, and the leveling flight mode could not be tested safely in real flight conditions. Due to limited time, in part caused by the broken capacitor and time spent ordering a new ESC and rebuilding the main drone body, further testing and development could not be completed. With more time, a barometer sensor could replace the GPS for altitude data to test the hovering correction methods. The GPS still can still be used for accurate horizontal
positioning data and could act as a second sensor in the control loop to not only fly to a specific altitude, but a physical position. Another potential for further research is trying to extract data from the flight controller itself to the Arduino. There is a built-in barometer and accelerometer in the flight controller, but they are in the center of the flight controller board. If data could be extracted an additional barometer would not need to be attached to the Arduino, an individual signal wire could be soldered onto the board and sent to the Arduino. Another future development would be utilizing the first-person view camera to try and code the flight to follow or fix about an object. A different more powerful microcontroller would need to be used that uses a different programming language such as Python.

**Conclusion:**

Through experimentation the wiring of the Arduino intertwined the flight controller, receiver, and GPS effectively powered every device and sent the signals between each part where they needed to go. The code itself was able to generate a PPM signal with steady values using a pause time of 35 ms between frames of data, minimizing the input delay when flying. A switch on the transmitter was coded to generate a manual to automatic switch such that a drone pilot could enter and exit manual and autonomous flight instantly. The autonomous flight code worked as intended while testing on the Betaflight configurator however actual flight tests could not be conducted due to inconsistent GPS altitude data. If the GPS inaccurate but remained precise the flight mode would still be effective, however the data is both inaccurate and imprecise rendering the information useless and the control loop incomplete. By using barometer data as an altimeter, the control loop would have a precise sensor allowing the drone to be flight tested and the specific values of the autonomous flight mode to be altered to maximize speed and steadiness of the flight. The core Arduino code is effective in creating a switch between manual and autonomous flight and sending an accurate signal to the flight controller. The specifics of the autonomous flight mode could be further explored and perfected by adding additional sensors and generating a multi-sensor control loop.
Sources:


Appendix A: Auto Drone Code

```c
#include <PMReader.h>
#include <TinyGPS++.h>
#include <SoftwareSerial.h>

static const int RXPin = 9, TXPin = 8;
static const uint32_t GPSbaud = 9600;

TinyGPSPlus gps;
SoftwareSerial ss(RXPin, TXPin);

// Initialize Variables.
int interruptPin = 3;
int channelAmount = 6;
PMReader ppm(interruptPin, channelAmount);

int outPin = 6;
long ch[7];
long levelThrot = 0;
long levelAlt = 0;
long deltaAlt = 0;
long deltaThrot = 0;

unsigned long ppmMC;

void setup() {  
  Serial.begin(9600);
  pinMode(6, OUTPUT);
  ss.begin(GPSbaud);
}

void loop() {  
  // Read Receiver Signal
  for (int channel = 1; channel <= channelAmount; ++channel) {
    unsigned long value = ppm.latestValidChannelValue(channel, 0);
    ch[channel-1]=value;
  }

  // Manual/Automatic Switch
  if (ch[5] > 1200 && levelThrot != 0) {
    levelThrot = ch[3];
    levelAlt = gps.altitude.feet();
  } else if (ch[5] > 1200 && levelThrot == 0) {
    levelThrot = 0;
    levelAlt = 0;
  }

  // Autonomous Flight Mode
  if (levelThrot != 0) {
    deltaAlt = gps.altitude.feet() - levelAlt;
    if (ch[5] > 1800) {
      deltaThrot = deltaAlt / 5;
    } else if (deltaThrot == 0) {  
      deltaThrot = 0;
      if (deltaThrot == 1800) {
        ch[3] = levelThrot;
        ch[4] = 1500;
        ch[5] = 1500;
        ch[6] = 1500;
      }  
      ppmMC = ppmMC + ppmMC;
    }
  } else {
    ppmMC = ppmMC - ppmMC;
    digitalWrite(outPin, HIGH);
    while (micros() - ppmMC > 500) {
      digitalWrite(outPin, LOW);
    }
    ppmMC = ppmMC - ppmMC;
    digitalWrite(outPin, HIGH);
    while (micros() - ppmMC > 500) {
      digitalWrite(outPin, LOW);
    }
  }
}
```

Appendix B: Additional Libraries Used

#ifndef __TinyGPSPlus_h
#define __TinyGPSPlus_h

#if defined(ARDUINO) && ARDUINO >= 100
#include "Arduino.h"
#else
#include "WProgram.h"
#endif
#include <Limits.h>

#define _TINY_GPS Version "1.8.2" // software version of this library
#define _TINY_GPS_PER_Knot 1.15077945
#define _TINY_GPS_PER_Knot 0.51444444
#define _TINY_GPS_PER_Knot 1.092
#define _TINY_GPS_METRES_PER_METER 8.0002137112
#define _TINY_GPS_METRES_PER_METER 8.001
#define _TINY_GPS_FIELD_SIZE 15

struct RadDegrees
{
    uint16_t deg;
    uint32_t millimeters;
    bool negative;
}
public:
    RadDegrees() : deg(0), millimeters(0), negative(false) {}  
};

struct TinyGPSLocation
{
    friend class TinyGPSFix;
    public:
    bool isValid() const { return valid; }  
    bool isUpdated() const { return updated; }  
    uint32_t apel() const { return valid ? millis() - lastCommitTime : (uint32_t)1U * 16384; }  
    const RadDegrees latitude() const { updated = false; return radLat(); }  
    const RadDegrees longitude() const { updated = false; return radLong(); }  
    double lat()  
    double lng()  
    TinyGPSLocation() : valid(false), updated(false) {}  

    private:
    bool valid, updated;  
    RadDegrees radLat, radLong, radNorth, radEastLat, radEastLong;  
    uint32_t lastCommitTime;
    void commit();  
    void setTrim Südeast(6);  
};

struct TinyGPSData
{
    friend class TinyGPSFix;
    public:
    bool isValid() const { return valid; }  
    bool isUpdated() const { return updated; }  
    uint32_t age() const { return valid ? millis() - lastCommitTime : (uint32_t)1U * 16384; }  
    uint32_t valcnt() { updated = false; return dna; }  
    uint32_t year()  
    uint16_t month()  
    uint8_t day()  
    TinyGPSData() : valid(false), updated(false), data(0) {}  

    private:
    bool valid, updated;  
    uint32_t year, month, day;  
    uint32_t lastCommitTime;
    void commit();  
};

struct TinyGPTime
{
    friend class TinyGPSFix;
    public:
    bool isValid() const { return valid; }  
    bool isUpdated() const { return updated; }  
    uint32_t age() const { return valid ? millis() - lastCommitTime : (uint32_t)1U * 16384; }  
    uint32_t valcnt() { updated = false; return time; }  
    uint8_t hour();  
    uint8_t minute()  
    uint8_t second();  
    TinyGPTime() : valid(false), updated(false), time(0) {}  

    private:
    bool valid, updated;  
    uint32_t year, month, day;  
    uint32_t lastCommitTime;
    void commit();  
};
struct TinyGPSDecimal
{
    friend class TinyGPSPlus;
public:
    bool isvalid() const { return value; }
    bool isupdated() const { return updated; }
    uint32_t age() const { return valid ? millis() - lastCommitTime : (uint32_t)(300.0L * MAX); }
    int32_t value() { updated = false; return value; }
    TinyGPSDecimal() : valid(false), updated(false), value(0) {};
};

struct TinyGPSInteger
{
    friend class TinyGPSPlus;
public:
    bool isvalid() const { return value; }
    bool isupdated() const { return updated; }
    uint32_t age() const { return valid ? millis() - lastCommitTime : (uint32_t)(300.0L * MAX); }
    int32_t value() { updated = false; return value; }
    TinyGPSInteger() : valid(false), updated(false), value(0) {};
};

struct TinyGPSSpeed : TinyGPSDecimal
{
    double knots() { return value * 1852.0L / 1000.0L; }
    double mph() { return _GPS_MPS_PER_KNOT * value + 3.6L; }
    double mps() { return _GPS_MPS_PER_KNOT * value + 3.6L; }
    double kmph() { return _GPS_KMPH_PER_KNOT * value + 3.6L; }
};

struct TinyGPSCourse : public TinyGPSDecimal
{
    double deg() { return value * 180.0L / 360.0L; }
};

struct TinyGPSAltitude : TinyGPSDecimal
{
    double meters() { return value + 0.0L; }
    double miles() { return _GPS_MILES_PER_METER * value + 3.6L; }
    double kilometers() { return _GPS_KM_PER_METER * value + 3.6L; }
    double feet() { return _GPS_FEET_PER_METER * value + 3.6L; }
};

struct TinyGPSWMOID : TinyGPSDecimal
{
    double hdp() { return value * 1852.0L / 1000.0L; }
};

class TinyGPSPlus
{
    TinyGPSPlus();
public:
    TinyGPSPlus(TinyGPSPlus &gps, const char *sentencName, int termNumber);
    void begin(TinyGPSPlus &gps, const char *sentencName, int termNumber);
    bool isupdated() const { return updated; }
    bool isvalid() const { return value; }
    uint32_t age() const { return valid ? millis() - lastCommitTime : (uint32_t)(300.0L * MAX); }
    const char *value() { updated = false; return buffer; }
private:
    voidComm();
    void setConst(char *term);
    char *buffer(GPS_MAX_FIELD_SIZE + 11);
    char *buffer(GPS_MAX_FIELD_SIZE + 11);
    unsigned long lastCommitTime;
    bool valid, updated;
    const char *sentenceName;
    int termNumber;
    friend class TinyGPSPlus;
    TinyGPSCustom next;
};