UAV: Autonomous Flight

By Gabriel Khaselev and Jordan Singleton

Abstract

In this project, we have assembled an aircraft with some autonomous capabilities. The implications of autonomous flight are substantial. Without pilots, flight can be mechanically and deterministically optimized; this could greatly benefit all aircraft applications. We implemented an autonomous aircraft by adding sensors and a processing unit to a Multiplex EasyStar II, a small remote-controlled aircraft kit. These sensors, including a barometer, accelerometer, and gyroscope, monitor the state of the aircraft. The state data is processed on the airplane and used to send signals to the plane's controlling motors. We have also built a mechanism for our onboard computer to read signals sent by the plane's remote receiver to the servos, in order to retain remote control capabilities. Although our autonomous tests have not been successful, we have implemented a system where the airplane is controllable by software, and we have gathered data from wind tunnel experiments.

Table of Contents

Introduction

The purpose of this project is to construct a fixed-wing aircraft with a degree of autonomy. We decided to do this using a model airplane kit, to which we could add an embedded computer and sensors. We decided that the aircraft should retain its remote control capabilities, so that we could log sensor data while flying the plane manually, and so that manual control could be regained if the autonomous program was not functioning correctly.

We used a Multiplex Easy Star II airplane kit, which we selected due to its reputedly stable flight and durability. To this plane we mounted a processor, an accelerometer, a gyroscope, and an altitude meter (barometer). For our processor we used a development board called a BeagleBone Black. We selected this board because it combines a reasonably small size and light weight with relatively high processing power, and also has many I/O ports. This unit runs our custom software, which processes input from the sensors, and output signals to the aircraft's motors. We also used 2 MSP430 microcontrollers to read the signal from the plane's remote receiver, and send the signal in a form readable to the BeagleBone Black.

Technical Discussion

I. Airplane Kit Overview

We are using a Multiplex Easy Star II aircraft kit. The Easy Star II uses a elapor foam frame (length 32 inches, wingspan 54 inches), and is reputedly highly durable and easy to fly. The plane flies at roughly 15 to 20 miles per hour, which is reasonably slow for a model airplane of its size, and which allowed us to perform testing on campus. This model also has the reputation of very stable flight and

relative ease of use for beginner pilots.

The standard setup of the plane uses a 2.4 GHz tx/rx receiver. This unit receives a signal from the controller, and outputs 4 PWM (pulse width modulation) channels. These PWM channels are connected to a rudder servo, an elevator servo, an Electronic Speed Control (controlling the motor driving the propellor). Optionally, the last channel is connected to two aileron servos; these ailerons are always moved together.

Figure 1: The Multiplex Easystar II

II. Basic Flight Theory

The primary forces that act on an aircraft are lift, drag, thrust, lateral force, and gravity. Lift is generated by the pressure differential that is formed by air rushing past the plane's wings. Lift should be nearly equal to the weight of the airplane. Drag is caused by the resistance formed by the air to the plane moving through it. Thrust is a forward acting force, in our case generated by the propellor, which should counteract drag. Lateral force should be very low, but can be caused by crosswinds.

A basic equation for the aerodynamic forces acting on an aircraft is given by:

$$F_A = -i_w D + j_w Q + k_w L$$

Where:

D = Drag

Q = Lateral Force

L = Lift

An equation for lift is given by:

$$L = \oint pn \cdot k \, dA$$

Where:

p is the pressure value

n is the unit vector normal to the wing

k is the unit vector normal to the freestream direction

Similar equations can be written for drag and lateral force.

Causing lift is a pressure differential formed by air rushing past the airplane's surfaces; as it does this, the air's flow forms curved streamlines. The basic relationship between the pressure and these streamlines is given by:

$dp/dR = \rho v^2/R$

Where:

R is the radius of curvature

p is the pressure

ρ is the air density

v is the velocity

The Easy Star's servos together manipulate the *control surfaces* of the aircraft (Figure 2). These control surfaces are used to adjust three critical parameters in flight dynamics: yaw, pitch, and roll.

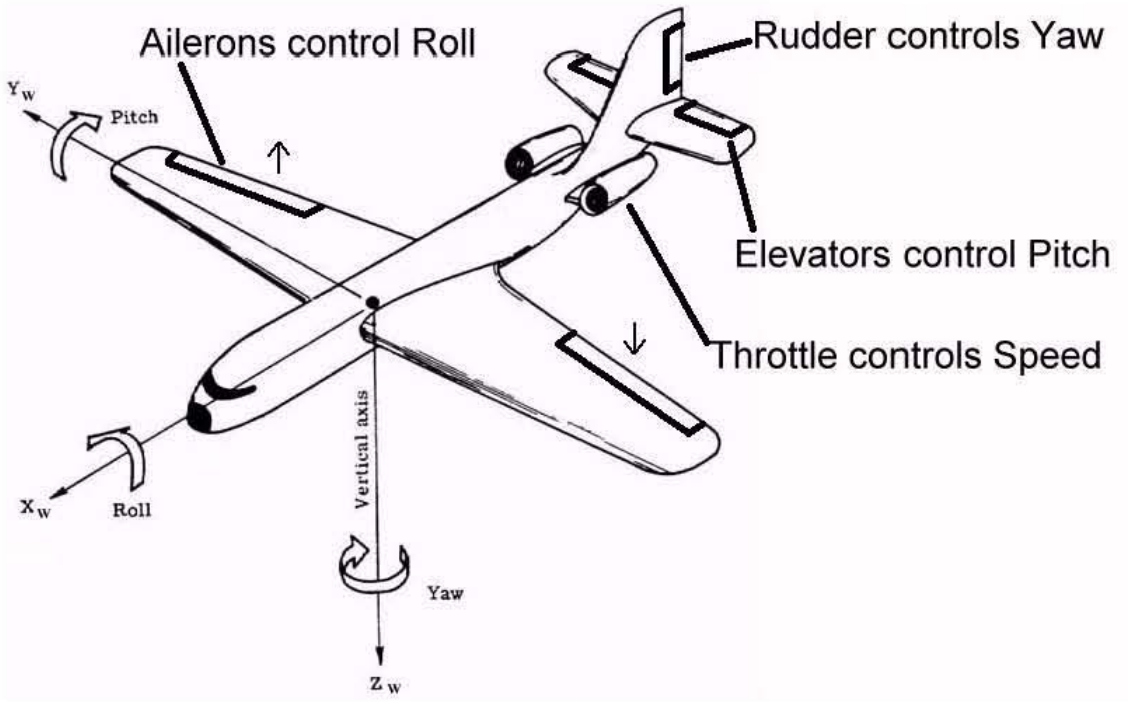

Figure 2 - Airplane Control Surfaces

Yaw, pitch, and roll are measured in angles of rotation about the axes of the aircraft. These axes runs through the center of mass of the aircraft. Ailerons are used to adjust roll, the rudder controls yaw, and the elevator controls pitch. Turning the aircraft can be accomplished using ailerons, rudder, or both; the roll generated by the ailerons causes the aircraft to bank, resulting in a curved flight path.

The throttle, in our case a single motor controlled through the ESC, controls the aircraft speed. The speed of the aircraft (relative to the air) determines how fast air rushes past the other control surfaces, and therefore the amount that each surface adjusts its respective parameter. The speed also determines the amount of lift generated by air rushing over the wings; therefore it acts in concert with the elevator to control aircraft ascent and descent.

We can examine our pressure relationship equation to see that inclining one of our control surfaces towards a given direction will cause an aerodynamic force opposite that direction. The airflow will curve to follow the direction of the control surface, so that the streamlines of greater radius are on the side away from the direction the surface is pointed. These lines of greater radius have a lower pressure, so the force on the aircraft points toward them. Therefore pointing the elevator up causes a downward force on the aircraft's tail, and a corresponding pitching moment. Similar effects are produced by ailerons and rudder.

III. Overview of our System

To create an autonomous airplane, we implemented control the servos and the ESC with our embedded computer, the Beaglebone Black (BBB). The BBB either translates sensor data into appropriate motor (and servo) controls, or it sends the motor and servos signals based on input that it receives from the radio receiver. MSP430 microcontrollers are used to read signal from the radio receiver, and send a signal to the BBB. The sensors providing input are the camera, barometer, accelerometer, and gyroscope. The control surfaces of the airplane are the propellor, the elevator, the rudder, and two ailerons. The BBB is the central component of the control system, which receives sensory input, runs the input processing software, and sends signals to the motors. The propellor is set by a variable speed motor to function as a throttle, and servos set the elevator, rudder, and ailerons. These surfaces are used to set the roll, pitch, and yaw of the aircraft. A diagram of the control structure is shown in figure 3.

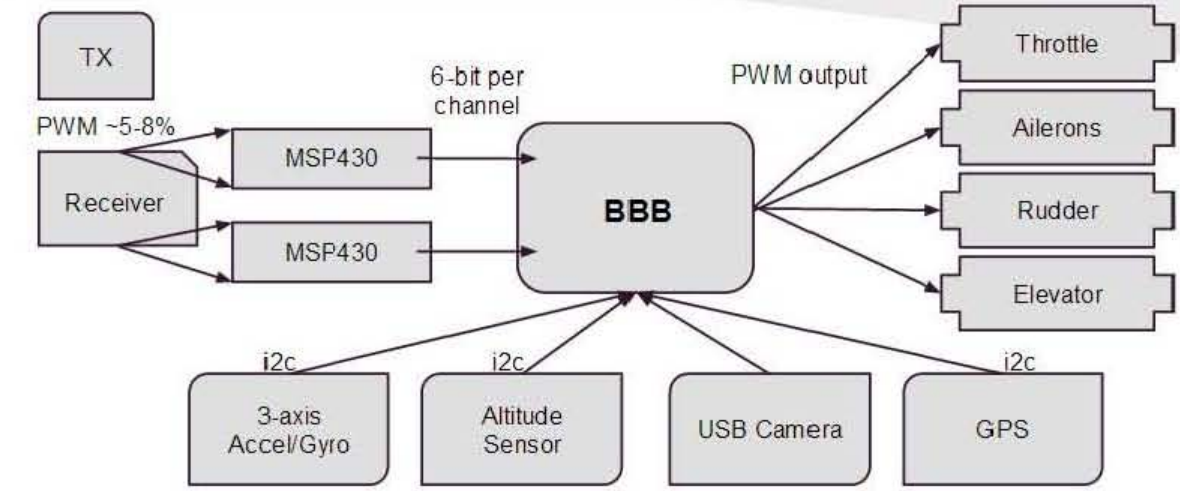

Figure 3 - Simple Diagram of Control Structure
Note: GPS and USB Camera not currently in use

IV. Hardware Details

The plane is controlled by a motor for the propellor, and four servos for the elevator, rudder, and two ailerons. The motor runs at 200 watts, and is connected to an electronic speed control (ESC). This ESC is controlled via pulse width modulation (PWM) on its input; to control servos autonomously we send input signals from the BeagleBone's built-in PWM modules. Four servos are also used for plane control (for elevator, rudder, and ailerons); these servos run at 5 V, .5 A, and can be controlled using the BBB's output pins and similar PWM techniques.

We used a 11.1 V LiPo battery with a 2200 mAh capacity, which outputs a continuous 66 amps (and can output a burst up to 132 amps). This battery can power a standard Multiplex Easy Star II (without peripherals) for about 45 minutes - when considering the power drawn from the servos and the onboard computer our battery life was closer to 30 minutes. To power the BBB, which runs at 5V, we use a DC-to-DC converter spliced to a standard 5V jack.

The BeagleBone Black is the piece of hardware that stores the software, runs the software, receives data from the sensors, and outputs controls to the motors. It contains a 1 GHz ARM processor, 512 MB of RAM, and 2 GB of onboard storage. Its weight is approximately 40g, and it uses a 5 V (.35 A) power supply, so it can be mounted on the aircraft with minimal effects on the aircraft's weight and power consumption. It contains I2C ports, analog inputs, GPIO ports, and many other IO peripherals. Our sensors are connected to the BeagleBone Black to help us determine state

and send output signals to the control motors. The BeagleBone also receives input from a 2.4GHz RC transmitter for emergency manual control; this is implemented through MSP430 microcontrollers.

The sensors we use consist of an accelerometer, a gyroscope, and an altitude meter; these are interfaced with the BeagleBone Black via the standard I2C protocol. The accelerometer and gyroscope are combined into the same device, which provides a triple axis accelerometer and a triple axis gyroscope, both with low drift. The altitude meter is a pressure sensor (with calibration used to translate into altitude), with a pressure measurement resolution that translates to an altitude resolution of 30cm. This is a sufficiently small resolution that allows us to monitor aircraft elevation effectively. Both of these devices are lightweight and consume little power, each weighing under 20g and running at about 3V. For our camera, we have a USB camera, connected to and powered by the Beaglebone Black's USB port. This camera provides 1.3 megapixel resolution at 30 FPS, but the camera is not used in our current setup.

Before creating our autonomous program we implemented a PWM capture program on two MSP430G2553s. The PWM duty cycle values are sent to the Beaglebone, so that the Beaglebone is able to then send these signals to the servos, resulting in standard manual control. By using two MSP430's (with two clocks each) we were able to capture PWM readings from the four output channels of the transmitter. By routing these signals through the Beaglebone before they are sent to servos, we intended to implement logging of the signal data, as well as a "kill switch" functionality. The term "kill switch" refers to the ability to take over manual control in the event that the autonomous program malfunctions. The MSP430s take PWM input from the transmitter and output a 6-bit digital reading of the duty cycle on each channel. A 6-bit resolution proved to be accurate enough for the range of duty cycles sent from the transmitter, which range from ~4.5-9.5%, at 45 Hz. With 6 bits, this translates to a resolution of approximately .08 % duty cycle.

V. Software

Initialization

The first part of the software work consisted of setting up the BeagleBone Black's environment. The BBB runs Angstrom Linux (3.8.13), this allows us to run C,C++, Javascript, or Python code easily. We make use of several open source libraries for interfacing components with the BeagleBone IO model; in particular we use two python libraries, one for performing general IO such as outputting pwm to servos, the other is for easily interfacing with i2c devices. Ultimately we found that there are several nuances to working with the beaglebone black, such as setting up the device tree to open all of its GPIO ports for use.

Kill-Switch

An important step to the project was to implement a "kill-switch" system so that we could easily regain manual control of the airplane in case the execution of the autonomous routines faltered. Ideally

this system would be capable of capturing the PWM signal received from the remote control transmitter, sending the values to the Beaglebone and outputting the same controls to the appropriate servos. The PWM capture was implemented on an MSP430G2553 in C.

To capture PWM, we use a timer on the MSP430 and detect rising a falling edges on an input pin while counting timer cycles between signals. When the signal goes high we begin to count the pulse length, then after the signal goes low we continue to increment the length of the period of the signal. Finally we divide these two numbers to determine the duty cycle. By enabling Capture-Compare Interrupts for each timer we are able to store the period and pulse width counts. To send the data to the beaglebone it is quantized into a 6 bit number, using a range of 000000 to 111111 to represent 4.5 to 9.5 gives us a resolution of 0.08, which is accurate enough for this purpose. This six bit number is sent across digital pins of the MSP430 and read in on six GPIO pins on the beaglebone.

Figure 4 - Msp430 pin configuration

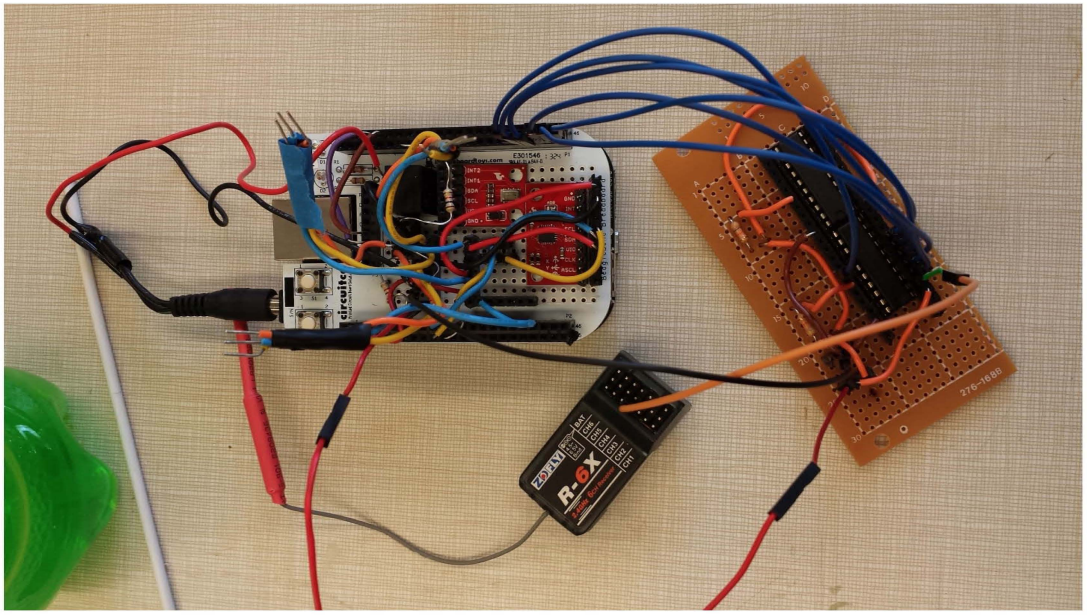

Figure 5 - MSP430 to Beaglebone connection for one channel

The orange wire runs the receiver input into the MSP430 on the beige board. There are six blue wires running the MSP430 output into the Beaglebone GPIO pins.

The same procedure was used for the MSP430's second timer to capture another PWM channel. To capture all four channels simultaneously we used two MSP430's.

To enable use of so many GPIO pins on the Beaglebone Black, we had to modify the system's device tree overlay, which specifies which pins are used for which purpose. To remove the HDMI overlay, we had to edit the uEnv.txt file in the mmcblk0p1 partition, which is referenced when loading overlays at startup. This was necessary because removing these overlays after startup triggers a bug in the BBB's kernel, causing a kernel panic. Ultimately we were unable to use this system to capture more than one channel, as discussed in the results section.

See the appendix for the MSP430 code and for the modified uEnv.txt.

I2C Interface

Inter-Integrated Circuit (I2C) is a system for peripherals to interface with embedded systems. Our Beaglebone Black reads data from the sensors via this system. Implementing this communication proved to be more complicated than we anticipated, as the sensors must be configured carefully. To interface with the accelerometer, we made use of a python class that we found on GitHub: https://github.com/cTn-dev/PyComms

Our own code can be found in the appendix.

Data Acquisition

Before building autonomous routines for the aircraft, we decided to build a data acquisition program, which would log the sensor data as well as control signals, during manually controlled flight. We thought that having this data (sensors and control outputs) for manual flights would help us design the autonomous program that would translate the sensor data into control outputs (and then roll, pitch, and yaw of the airplane). Our date acquisition program writes duty cycle values and sensors values to .csv files.

This program was implemented in Python using the Adafruit BBIO library. This library proved to be essential for interfacing with the Beaglebone IO pins easily and effectively.

This program was set to run on startup using systemd, a linux API that can be used to set "service" bash scripts to be run at startup. The logger bash script simply launches the python script. We also implemented a simple python script, which launches our main data acquisition program when a button on the Beaglebone protoboard is clicked. By launching this program at startup, we are able to plug the BeagleBone into the battery within the plane, then click the button to start data gathering just before flight.

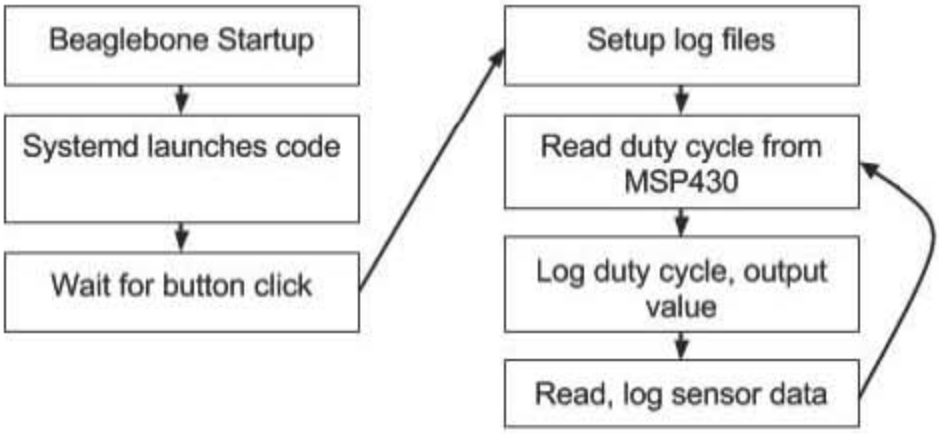

Figure 6 - Control flow of data logging program

Autonomous control

Our first plan for an autonomous routine was to maintain stable flight; this essentially requires the beaglebone to output servo controls to maintain the same Roll, Pitch, Yaw, and Altitude during flight. This is difficult because wind and air turbulence can quickly alter these values. A strong wind can cause flight of the styrofoam plane to be very unpredictable and hard to manage.

To implement stable flight, or indeed any controlled autonomous flight, we needed to implement a control loop that determines how to manipulate the plane's control surfaces based on sensor input. We implemented two control loops: a proportional control loop and an integral control loop. The proportional control loop sets the angle of a servo in proportion to the angle of the corresponding flight parameter. The integral control loop adds to or subtracts from the angle of the servo, with the value added or subtracted proportional to the angle of the flight parameter. There are max and min values that we enforce for the duty cycles sent to the servos, based on the physical characteristics of the plane.

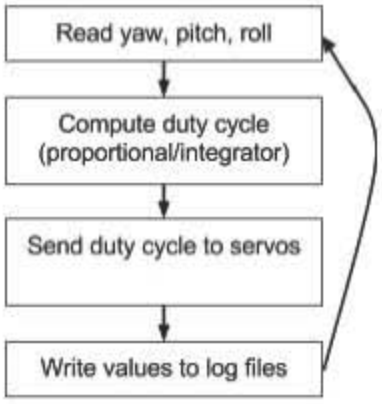

Figure 7 - Controlling the aircraft

Ideally, the autonomous program would consist of several key elements: image processing from the camera, a pre-set autonomous routine to be performed, and logical guidelines for how to perform movements - for how to maintain level flight, or implement turns.

Results

We have implemented sensor communication with the BBB, modular code to read sensor data and to control servos including two types of control loops, and modular code that is used for data acquisition.

We have implemented the MSP430 code so that they accurately read PWM, and so that the BBB can accurately control a single servo based on input from a MSP430. However we have encountered an issue on the BBB, where when more than one 6-bit channel is attached to the GPIO inputs for this purpose, the BBB goes into an error protection mode and the file system becomes read only. We can successfully read from all GPIO ports when an input other than the MSP430 is used. This error needs further investigation, or more work must be done to implement a workaround.

However, we can still log accelerometer data while flying the plane, with a configuration such that the radio receiver is routed directly to the servos. We can also wire the plane such that it is controlled autonomously with the BBB - however this is dangerous without a kill-switch to regain manual control in the event of a malfunction.

We have performed several tests which evaluate our control loops. These tests were performed in the Swarthmore College wind tunnel at realistic flight speeds. Due to the size constraints of the wind tunnel, the wings were removed from the plane and testing was performed on the elevator only. These tests were not devised to prove autonomous functionality, but rather to observe the the plane's reaction, and to incrementally test control loops. Testing methods include static displacement correction, dynamic pitch response, and integrator control.

Static Displacement Correction

Static displacement correction involves first balancing the plane within the wind tunnel. Next we apply a pitching moment to the plane by placing a 180g weight either in front or behind the center of gravity. We measure the pitch angle that the plane is displaced to, then we start our proportional control loop, and finally measure the angle that the plane corrects to.

In our first round of testing, we balanced the plane then applied a pitching moment five inches in front of the center of gravity. In the second round the pitching moment is applied six inches behind the center of gravity. The tables below show the angle of the plane before and after the proportional correction is applied, additionally we show the duty cycle sent to the elevator servo.

5" forward CG	15 MPH	20 MPH	25 MPH
pre-adjust (deg)	13.5	9.3	5.8
post-adjust (deg)	10.2	6.2	3.1
PWM duty cycle	7.6	7.1	6.8

6" behind CG	15 MPH	20 MPH	25 MPH
pre-adjust (deg)	-10.7	-5.3	-3.7
post-adjust (deg)	-7.2	-4	-2.5
PWM duty cycle	6.1	6.3	6.4

Table 1 - Results of Static Displacement Correction Testing

The results above show that our proportional correction scheme is unable to correct the pitch of the plane to zero degrees; equilibrium is reached at an angle significantly different from zero. We can see that controls are more effective at higher wind speeds, this is intuitive because a higher wind speed will apply a larger corrective force on the elevator. Additionally, there is a slight difference between the correction of the plane when the pitching moment is applied on different parts of the plane. In general, it is more difficult to correct displacement where the nose points down.

Dynamic Pitch Response

This test involved balancing the plane within the wind tunnel and then capturing accelerometer readings as the elevator is moved to a pre-set deflection. We are effectively collecting data on the dynamic response of the plane as its pitch changes, and adjusts to a new equilibrium; no control loop is used. These tests were also performed at different wind speeds but in each case the elevator is moved to the maximums of its range, this corresponds to 8% and 5% PWM signals respectively.

The graphs below depict the pitch response of the plane. The pitching angle in degrees is plotted against time in seconds.

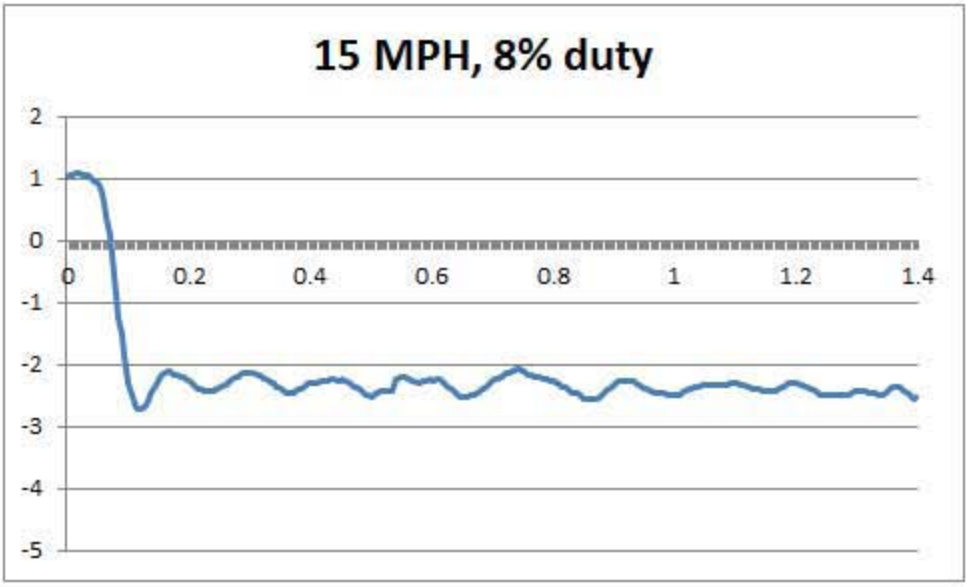

Figure 8 - 15 MPH, 8% duty cycle
Standard deviation after 0.2s ~ 0.12 degrees

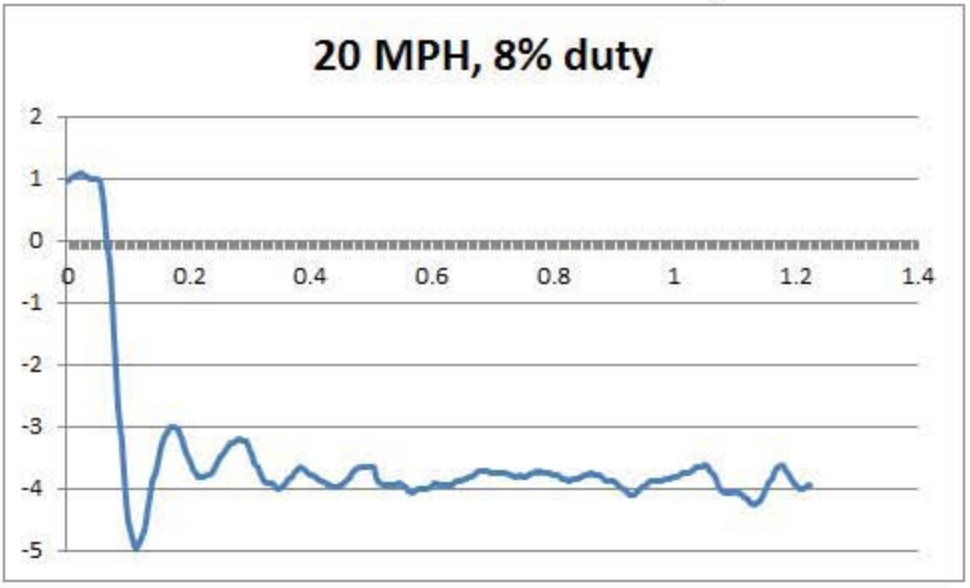

Figure 9 - 20 MPH, 8% duty cycle
Standard deviation after 0.2s ~ 0.28

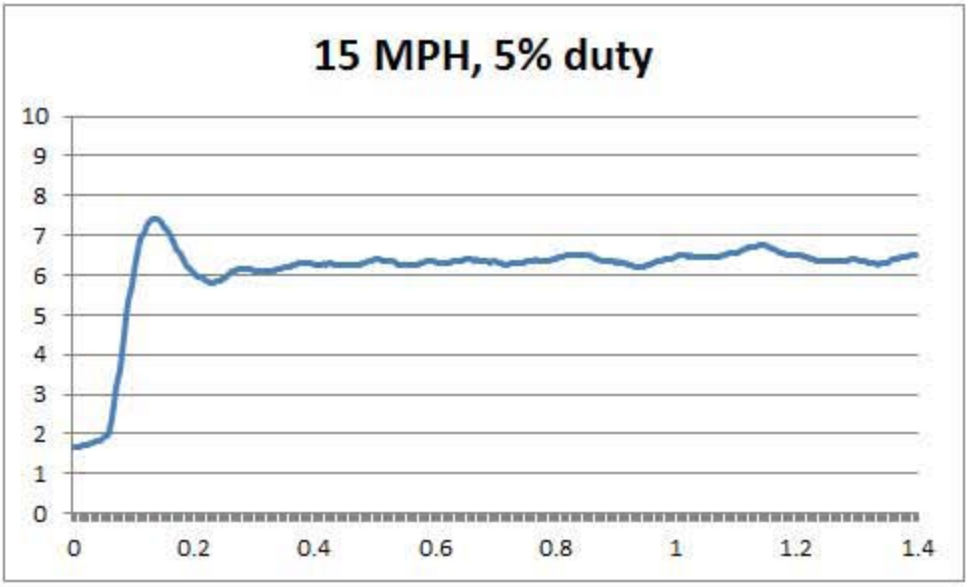

Figure 10 - 15 MPH, 5% duty cycle
Standard deviation after 0.2s ~ 0.16

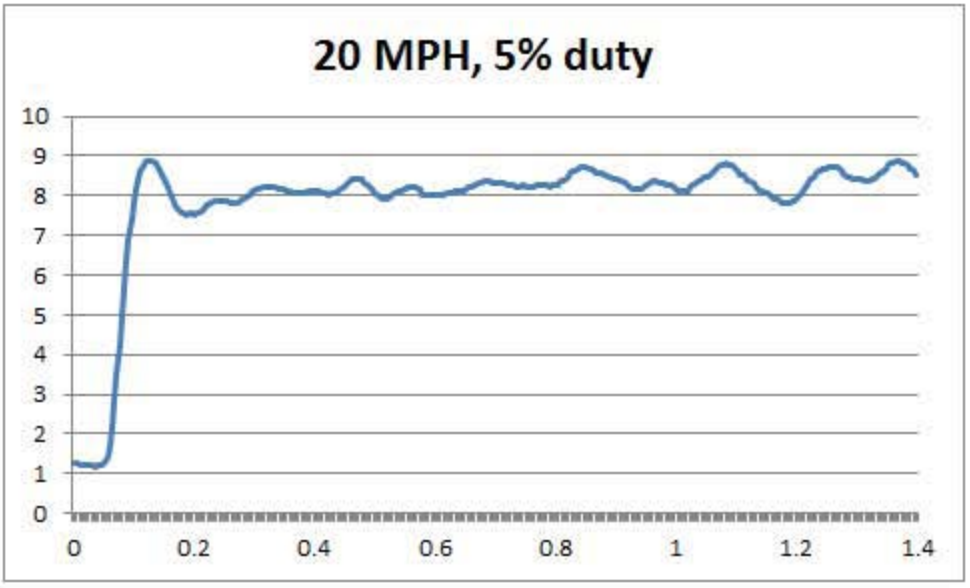

Figure 11 - 20 MPH, 5% duty cycle
Standard deviation after 0.2s ~ 0.26

In these results, we can see that the plane's pitch is quickly adjusted after the control surface is moved. In most of the graphs above, a stable pitch is achieved within 0.2 seconds. The overshoot in the plane response is approximately one degree; this is negligible with respect to pitching effects on the plane. The response is slightly underdamped, followed by minor fluctuations. This noise tends to have a standard deviation of about 0.2 degrees, which is also negligible within the scope of this plane's flight. We again observe the difference in the response of the plane between correcting upwards and downwards.

Integrator Control

In this test, we used a control loop with an integrator, which continuously adjusts the pitch to zero degrees. Again we used a 180 gram weight to provide a pitching moment to change the angle of the plane. This angle is detected by the accelerometer and a corrective control is applied. As this control loop is applied, the pitch angle and the duty cycle sent to the elevator are logged.

The graphs below depict excerpts from our integrator tests. Duty cycle (red) and pitch angle (blue) are plotted against time in seconds.

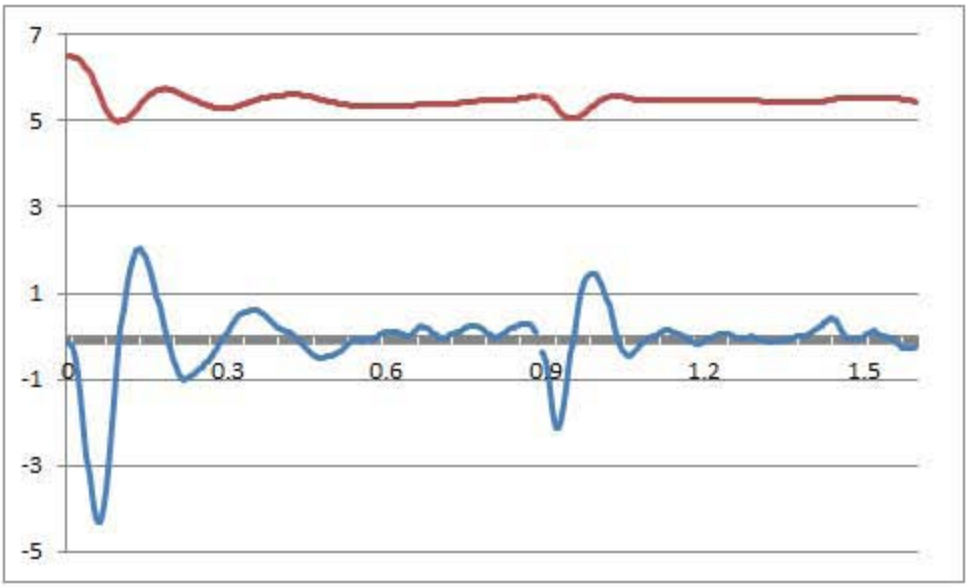

Figure 12 - Integrator Response 1

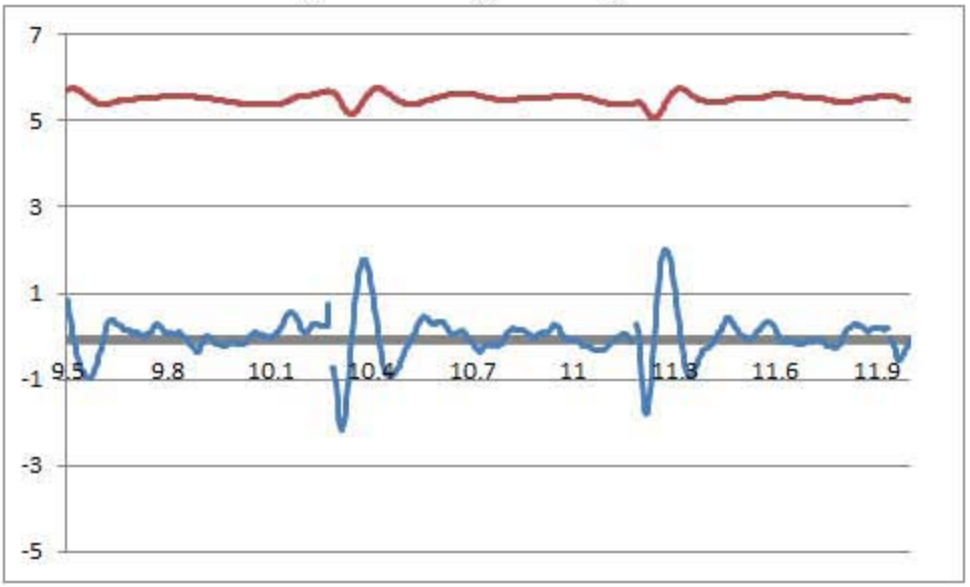

Figure 13 - Integrator Response 2

The results from integrator testing show that the control loop is capable of correcting the pitch of the aircraft to zero. This control method can be applied to other control surfaces in addition to the elevator to maintain stable flight. Similarly, an integrator can be set to adjust a flight parameter to some non-zero value, which could be used for banking or turning instead of straight flight. Several such integrators can be combined and manipulated in a state machine to implement autonomous routines. While the integrator is effective in correcting the pitch of the plane within the wind tunnel, it was unclear if those results would translate to real autonomous flight.

Partially Autonomous Test

Our final test was to give autonomous control to one control surface of the plane, while wiring other servos directly to the radio receiver. We decided to give autonomous control to the ailerons, to stabilize roll; we thought that this would be a reasonable parameter to let the control loop stabilize, and we also thought that manual control of the other channels would be sufficient for flight. We ran the tests using out integrator control loop, with an adjusted gain. Our tests were not successful, as the plane would not maintain flight for long without crashing. Often it seemed that the plane never recovered from takeoff, which consisted of us throwing the plane by hand. It was impossible to do this without an initial roll, which was not stabilized before a crash. Wind was also sometimes a factor in failed tests; we did not have time to wait for a perfectly still day.

Conclusion

We have created a system that will allow for both manual and autonomous control of the aircraft, which has been outfitted with an embedded computer. We were able to program routines to automatically manipulate flight controls based on sensor input. We were also able to implement PWM duty cycle capturing on an MSP430 for two simultaneous inputs.

We programmed two control loops that worked towards straight and level flight. We tested these first in a wind tunnel, and then with a partially autonomous test. Out integrator control loop worked well in the wind tunnel, but did not translate into real flight. The next steps in this project would involve more partially autonomous tests, in which we tweak our control loop, and ensure that control surfaces are level at takeoff.

There is a great deal of work still to be done to implement real autonomous flight. This would be implemented in degrees, starting with basic level flight, then adding tolerance for windy and variable conditions, and implementing controlled turns. The camera could be integrated into the control system to provide more advanced autonomous techniques that use computer vision. A GPS could be integrated for strategic path and location planning. The range of possibilities for this technology is very expansive and desires further exploration.

Appendices

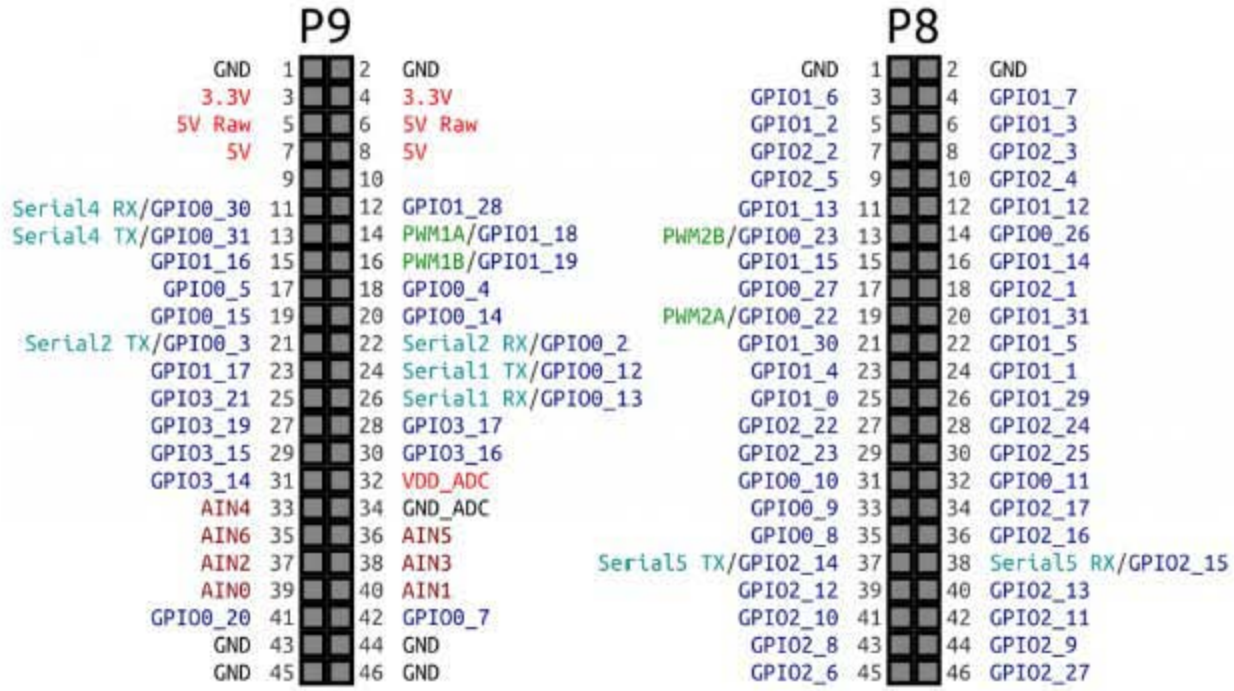

Figure 14 - Beaglebone Black Pinout

P9_1	Ground
P9_3	3.3V for I2C, Servos
P9_14	PWM Out
P9_19	I2C Clk
P9_20	I2C Data
P9_21	PWM Out
P9_42	PWM Out
P8_8	Button In
P8_10	Button Out
P8_13	PWM Out
P8_23 - P8_46	Lines in from MSP430

Table 2 - Beaglebone Pins Used

i2c_alt_class.py - class for interfacint with altitude sensor

```
from Adafruit_I2C import Adafruit_I2C
import time
class altSensor:
  def __init__(self, alt_address = 0x60):
    self.addr = 0x60
    self.alt = Adafruit_I2C(self.addr)
    #set to altimeter with OSR = 128
    self.alt.write8(0x26,0xB8)
    #enable data flags in PT_DATA_CFG
    self.alt.write8(0x13,0x07)
    #set up polling for data
    #set active
    self.alt.write8(0x26,0xB9)
  def getAlt(self):
    alt = self.alt
    #read STATUS reg
    status = alt.readU8(0x00)
    #TODO change infinite while?
    while(True):
      if(status & 0x08):
        p_msb = alt.readU8(0x01)     #bits 12-19
        p_csb = alt.readU8(0x02)     #bits 4-11
        p_lsb = alt.readU8(0x03)     #bits 0-3
        t_msb = alt.readU8(0x04)     #bits 4-11
        t_lsb = alt.readU8(0x05)     #bits 0-3
        meters = (p_msb << 8) | (p_csb)
        fraction = 0
        if(0b10000000 & p_lsb):
          fraction += 2**(-1)
        if(0b01000000 & p_lsb):
          fraction += 2**(-2)
        if(0b00100000 & p_lsb):
          fraction += 2**(-3)
        if(0b00010000 & p_lsb):
          fraction += 2**(-4)
```

```
    return meters + fraction
def getAlt2(self):
  alt = self.alt
  #read STATUS reg
  status = alt.readU8(0x00)
  #TODO change infinite while?
  ii = 0
  while(True):
    ii += 1
    if(ii >= 50):
      return 0
    if(status & 0x08):
      p_msb = alt.readU8(0x01)     #bits 12-19
      p_csb = alt.readU8(0x02)     #bits 4-11
      p_lsb = alt.readU8(0x03)     #bits 0-3
      t_msb = alt.readU8(0x04)     #bits 4-11
      t_lsb = alt.readU8(0x05)     #bits 0-3
      meters = (p_msb << 8) | (p_csb)
      fraction = 0
      if(0b10000000 & p_lsb):
        fraction += 2**(-1)
      if(0b01000000 & p_lsb):
        fraction += 2**(-2)
      if(0b00100000 & p_lsb):
        fraction += 2**(-3)
      if(0b00010000 & p_lsb):
        fraction += 2**(-4)
      return meters + fraction
def printLoop(self, pause = 0):
  alt = self.alt
  while(True):
    #read STATUS reg
    status = alt.readU8(0x00)
    if(status & 0x08):
      p_msb = alt.readU8(0x01)     #bits 12-19
      p_csb = alt.readU8(0x02)     #bits 4-11
      p_lsb = alt.readU8(0x03)     #bits 0-3
      t_msb = alt.readU8(0x04)     #bits 4-11
      t_lsb = alt.readU8(0x05)     #bits 0-3
```

```
            meters = (p_msb << 8) | (p_csb)
            print "m: "
            print bin(p_msb), bin(p_csb)
            print bin(meters)
            print meters
            fraction = 0
            if(0b10000000 & p_lsb):
               fraction += 2**(-1)
            if(0b01000000 & p_lsb):
               fraction += 2**(-2)
            if(0b00100000 & p_lsb):
               fraction += 2**(-3)
            if(0b00010000 & p_lsb):
               fraction += 2**(-4)
            print "f: "
            print fraction
            print "p: "
            print (p_msb << 12) | (p_csb << 4) | (p_lsb >> 4)
            print "t: "
            print (t_msb << 4) | (t_lsb >> 4)
            time.sleep(pause)
        #alt = altSensor()
        #alt.printLoop()
```

pwm_class.py - basic class for interfacing with a servo

```
        import Adafruit_BBIO.PWM as PWM
        import time

        class ServoPWM:
            def __init__(self, servo_pin, dc_min=3, dc_max=14.5, freq=45):
                self.servo_pin = servo_pin
                self.dc_min = dc_min
                self.dc_max = dc_max
                self.dc_span = dc_max - dc_min
                self.freq = freq
                PWM.start(self.servo_pin, (dc_max-dc_min)/2, freq)
            def mainloop(self):
                #PWM.start(servo_pin, (100-duty_min),freq)
```

```
            while True:
                set_duty = '1'
                set_duty = raw_input("enter 2 for input in duty cycle: ")
                angle = raw_input("angle 0 to 180 or x to exit: ")
                if angle == 'x':
                    PWM.stop(self.servo_pin)
                    PWM.cleanup()
                    break
                angle_f = float(angle)
                duty = ((angle_f/180)*self.dc_span+self.dc_min)
                if(set_duty == '2'):
                  duty = angle_f
                PWM.set_duty_cycle(self.servo_pin,duty)
        def setAngle(self, angle):
            angle_f = float(angle)
            duty = ((angle_f/180)*self.dc_span + self.dc_min)
            PWM.set_duty_cycle(self.servo_pin, duty)
        def setDuty(self, duty):
            PWM.set_duty_cycle(self.servo_pin, duty)

    if __name__ == "__main__":
      testPWM = ServoPWM("P9_21")
      testPWM.mainloop()
```

bb_class.py - “Bit banging” class that handles reading data sent from the MSP430

```
    import Adafruit_BBIO.GPIO as gpio
    import time
    import pwm_class
    class bitBanger:
      def __init__(self, port_list, pwm_port, thresh = .3, bits = 6):
        self.prev_duty = 5
        self.bits = bits
        self.port_list = port_list
        self.pwm_port = pwm_port
        self.thresh = thresh
        for i in range(0,6):
          gpio.setup(self.port_list[i], gpio.IN)
        self.servo = pwm_class.ServoPWM(self.pwm_port)
```

```
def read_set(self):
  raw = 0
  for i in range(0, self.bits):
    if( gpio.input(self.port_list[i]) ):
      raw += 2**i
  if(raw == 0):
    return 0
  else:
    duty = 4.5 + raw*5.0/63
    if( abs(self.prev_duty - duty) > self.thresh):
      #print "!!!!"
      self.prev_duty = duty
      return duty
    else:
      self.prev_duty = duty
      #print duty
      self.servo.setDuty(duty)
      return duty
def read(self):
  raw = 0
  for i in range(0, bits):
    if( gpio.input(self.port_list[i]) ):
      raw += 2**i
  if(raw == 0):
    return 0
  else:
    duty = 4.5 + raw*5.0/63
    return duty
def read_loop(self):
  while(1):
    raw = 0
    for i in range(0, bits):
      if( gpio.input(port_list[i])):
        print port_list[i]
        raw += 2**i
    if(raw==0):
      print "000000000000000"
    else:
      duty = 4.5 + raw*5.0/63
```

```
            print duty
            time.sleep(.1)
```

log_redirect.py - uses other classes to log accelerometer and control values, while also controlling servos based on input from MSP430

```
import time
import math
import os.path
#import mpu6050
import mpu6050_2 as mpu6050
from bb_class import bitBanger
from i2c_alt_class import *
#time interval to wait each loop
time_space = 0#.05
#number of loops to exit after
max_loops = 40000
loop = 0
#setup bitBangers
#lsb to msb
ports_ailerons = ["P8_46", "P8_44", "P8_42", "P8_40", "P8_38", "P8_36"]
ports_rudder =  ["P8_45", "P8_43", "P8_41", "P8_39", "P8_37", "P8_35"]
ports_elevator = ["P8_34", "P8_32", "P8_30", "P8_28", "P8_26", "P8_24"]
ports_esc = ["P8_33", "P8_31", "P8_29", "P8_27", "P8_25", "P8_23"]
servo_port_ailerons = "P9_42"
servo_port_rudder = "P8_13"
servo_port_elevator = "P9_21"
esc_port = "P9_14"
bb_ailerons = bitBanger(ports_ailerons, servo_port_ailerons)
bb_rudder = bitBanger(ports_rudder, servo_port_rudder)
bb_elevator = bitBanger(ports_elevator, servo_port_elevator)
bb_esc = bitBanger(ports_esc, esc_port)
class AccData:
  def __init__(self):
    self.x = 0.0
    self.y = 0.0
    self.z = 0.0
    self.yaw = 0.0
    self.pitch = 0.0
```

```
    self.roll = 0.0
    self.m = 0.0
# Setup logging files
acc_fname = "logs/acc"
ypr_fname = "logs/ypr"
alt_fname = "logs/alt"
pwm_fname = "logs/pwm"
fnum = 0
while(1):
  if(not os.path.isfile(acc_fname + str(fnum) + ".csv")):
    if(not os.path.isfile(ypr_fname + str(fnum) + ".csv")):
      if(not os.path.isfile(alt_fname + str(fnum) + ".csv")):
        if(not os.path.isfile(pwm_fname + str(fnum) + ".csv")):
          break
  fnum += 1
acc_fname = acc_fname + str(fnum) + ".csv"
acc_f = open(acc_fname, "w")
ypr_fname = ypr_fname + str(fnum) + ".csv"
ypr_f = open(ypr_fname, "w")
alt_fname = alt_fname + str(fnum) + ".csv"
alt_f = open(alt_fname, "w")
pwm_fname = pwm_fname + str(fnum) + ".csv"
pwm_f = open(pwm_fname, "w")
print "Saving logs numbered: ", fnum
acc_f.write("ts: " + str(time_space) + "\nx,y,z\n")
ypr_f.write("ts: " + str(time_space) + "\nyaw,pitch,roll\n")
alt_f.write("ts: " + str(time_space) + "\nm??\n")
pwm_f.write("ts: " + str(time_space) + "\nailerons,rudder,elevator,esc\n")
a_data = AccData()
alt = altSensor()
# Accelerometer initialization
mpu = mpu6050.MPU6050()
mpu.dmpInitialize()
mpu.setDMPEnabled(True)
# get expected DMP packet size for later comparison
packetSize = mpu.dmpGetFIFOPacketSize()
#function to fill in a_data
def getAccelerometerData():
    # Get INT_STATUS byte
```

```
    mpuIntStatus = mpu.getIntStatus()
    if mpuIntStatus >= 2: # check for DMP data ready interrupt (this should happen
frequently)
        # get current FIFO count
        fifoCount = mpu.getFIFOCount()
        # check for overflow (this should never happen unless our code is too
inefficient)
        if fifoCount == 1024:
            # reset so we can continue cleanly
            mpu.resetFIFO()
            print 'FIFO overflow!'
        # wait for correct available data length, should be a VERY short wait
        fifoCount = mpu.getFIFOCount()
        while fifoCount < packetSize:
            fifoCount = mpu.getFIFOCount()
        result = mpu.getFIFOBytes(packetSize)
        #read accel data
        q = mpu.dmpGetQuaternion(result)
        g = mpu.dmpGetGravity(q)
        ypr = mpu.dmpGetYawPitchRoll(q, g)
        accel = mpu.readAccel()
        a_data.x = accel[0]
        a_data.y = accel[1]
        a_data.z = accel[2]
        a_data.yaw = ypr['yaw'] * 180 / math.pi
        a_data.pitch = ypr['pitch'] * 180 / math.pi
        a_data.roll = ypr['roll'] * 180 / math.pi
#altitude data
def getAltData():
  a_data.m = alt.getAlt2()
while(1):
  getAccelerometerData()
  getAltData()
  ail_pwm = bb_ailerons.read_set()
  rud_pwm = bb_rudder.read_set()
  ele_pwm = bb_elevator.read_set()
  print ail_pwm, rud_pwm#, ele_pwm
  esc_pwm = bb_esc.read_set()
  acc_f.write(str(a_data.x) + "," + str(a_data.y) + "," + str(a_data.z) + "\n")
```

```
        ypr_f.write(str(a_data.yaw) + "," + str(a_data.pitch) + "," + str(a_data.roll) + "\n")
        alt_f.write(str(a_data.m) + "\n")
        pwm_f.write(str(ail_pwm) + "," + str(rud_pwm) + "," + str(ele_pwm) + "," +
      str(esc_pwm) + "\n")
        if time_space != 0:
          time.sleep(time_space)
        loop += 1
        if(loop >= max_loops):
          exit()
```

button.py - example of code that sets up a button press to launch a script

```
      import Adafruit_BBIO.GPIO as gpio
      import sys
      import os

      in_pin = "P8_8"
      out_pin = "P8_10"

      gpio.setup(in_pin, gpio.IN)
      gpio.setup(out_pin, gpio.OUT)
      gpio.output(out_pin, gpio.LOW)
      while(1):
        if(gpio.input(in_pin) == 0):
          break
      os.system("python ail_only.py")
```

auto.py excerpt - an example of a proportional control loop used by the elevator

```
      if(abs(a_data.pitch - prev_pitch) > 5):
       prev_pitch = a_data.pitch
       continue
      else:
       prev_pitch = a_data.pitch
       if(a_data.pitch > 0):  #nose pointing down: larger adjustment (gain)
         duty = (a_data.pitch + 15) * 3 / 30 + 5
       else:
         duty = (a_data.pitch + 30) * 3 / 60 + 5
       if(duty > 9):
```

```
        duty = 9
      if(duty < 4.5):
        duty = 4.5
      ele_servo.setDuty(duty)
```

integral.py excerpt - class implementing an integral control loop

```
class integral_control:
  def __init__(self, target_angle = 0, gain=.8):
    self.ta = target_angle #not implemented
    self.prev_angle_in = 0
    self.cur_duty = 6.5
    self.gain = gain
  def get_duty(self, angle):
    self.cur_duty += self.scale_angle(angle)
    if(self.cur_duty > 9):
      self.cur_duty = 9
    if(self.cur_duty < 4.5):
      self.cur_duty = 4.5
    return self.cur_duty
  def scale_angle(self, angle):
    d = 6.5 - ((-1*angle+45)*3/90 + 5)
    return d * self.gain
```

logger.sh - shell script added to service “logger” in systemd - executed on startup

```
#!/bin/bash
cd /home/root/E90/logger/
/usr/bin/python log_both.py > logs/last.log
```

uEnv.txt - modified file disabling HDMI overlay

```
optargs=quiet drm.debug=7 capemgr.disable_partno=BB-BONELT-HDMI,BB-BONE-EMMC-2G
```

PWMcapture.c - Captures 2 PWM input signals on MSP430

```
#include <msp430.h>

unsigned int new_cap_3 = 0;
unsigned int old_cap_3 = 0;
```

```
unsigned int cap_diff_3 = 0;
unsigned int diff_array_3[16];
unsigned int index_3 = 0;
float duty_3 = 0;

unsigned int new_cap_2 = 0;
unsigned int old_cap_2 = 0;
unsigned int cap_diff_2 = 0;
unsigned int diff_array_2[16];
unsigned int index_2 = 0;
float duty_2 = 0;

int p1;
int p2;
int bits[6];

// function prototypes
void output_duty(float duty, int flag, int overflow);

/*
 * main.c
 */
int main(void) {
	volatile unsigned int i;

	WDTCTL = WDTPW | WDTHOLD;                                   // Stop
watchdog timer

	P1DIR = 0xFD;                                                 // set
1.0,2,3,4,5,6,7 to output (1.1 for TA0)
	P2DIR = 0x3E;                                                 // set
2.1,2,3,4,5 to output (2.0 for TA1)

	P1SEL = 0x02;                                  // Set P1.1 to TA0
	P2SEL = 0x01;                                                 // Set
P2.0 to TA1

	TA0CCTL0 = CM_3 | SCS | CCIS_0 | CAP | CCIE;      // both edges +
CCI0A (P1.1) + Capture Mode + Interrupt
	TA1CCTL0 = CM_3 | SCS | CCIS_0 | CAP | CCIE;      // both edges +
CCI0A (P2.0) + Capture Mode + Interrupt

	TA0CTL = TASSEL_2 | MC_2;                  // SMCLK + Continuous
Mode
	TA1CTL = TASSEL_2 | MC_2;                             // SMCLK +
continuous mode

	_BIS_SR(LPM0_bits + GIE);                             // LPM0 + Enable
```

```
global ints

      return 0;
}

/*
 *  ======== Timer0_A0 Interrupt Service Routine ========
 */
#pragma vector=TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR_HOOK(void)
{

      new_cap_3 = TACCR0;
      cap_diff_3 = new_cap_3 - old_cap_3;
      if(new_cap_3 > old_cap_3){
            cap_diff_3 = new_cap_3 - old_cap_3;
      }else{
            cap_diff_3 = new_cap_3 + (65535 - old_cap_3);
      }
      diff_array_3[index_3] = cap_diff_3;
      old_cap_3 = new_cap_3;
      index_3++;

      if(index_3%2 == 0){
            int pulse;
            if(diff_array_3[index_3] < diff_array_3[index_3-1]){
                  pulse = diff_array_3[index_3];
            }else{
                  pulse = diff_array_3[index_3-1];
            }
            int period = diff_array_3[index_3] +
diff_array_3[index_3-1];
            duty_3 = (float) pulse/period;
            if(duty_3 > 0.03 && duty_3 < 0.10){ output_duty(duty_3, 1,
0); }
      }

      if(index_3>15){ index_3 = 0;}
}

/*
 *  ======== Timer1_A0 Interrupt Service Routine ========
 */
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR_HOOK(void)
{

      new_cap_2 = TA1CCR0;
```

```
	if(new_cap_2 > old_cap_2){
		cap_diff_2 = new_cap_2 - old_cap_2;
	}else{
		cap_diff_2 = new_cap_2 + (65535 - old_cap_2);
	}
	diff_array_2[index_2] = cap_diff_2;
	old_cap_2 = new_cap_2;
	index_2++;

	if(index_2%2 == 0){
		int pulse;
		if(diff_array_2[index_2] < diff_array_2[index_2-1]){
			pulse = diff_array_2[index_2];
		}else{
			pulse = diff_array_2[index_2-1];
		}
		int period = diff_array_2[index_2] +
diff_array_2[index_2-1];
		duty_2 = (float) pulse/period;
		if(duty_2 > 0.03 && duty_3 < 0.10){ output_duty(duty_2, 0,
0); }
	}

	if(index_2>15){ index_2 = 0;}
}

/*
 *  ======== Output Duty Cycle ========
 */
void output_duty(float duty, int flag, int overflow){

	float float_val = (duty*100 - 4.5)*63/5;   // 6 bit number
	int bin_val = (int) float_val;

	if(overflow){
		float_val = 0;
		bin_val = 0;
	}

	if(flag){
		p1 = 0;
		if(bin_val & BIT0){ p1 |= BIT0; }
		if(bin_val & BIT1){ p1 |= BIT2; }
		if(bin_val & BIT2){ p1 |= BIT3; }
		if(bin_val & BIT3){ p1 |= BIT4; }
		if(bin_val & BIT4){ p1 |= BIT5; }
		if(bin_val & BIT5){ p1 |= BIT6; }
		P1OUT = (p1 | (P1OUT & BIT7));
```

```
        }else{
                p2 = 0;
                P1OUT &= ~BIT7;
                if(bin_val & BIT0){ P1OUT |= BIT7; }
                if(bin_val & BIT1){ p2 |= BIT1; }
                if(bin_val & BIT2){ p2 |= BIT2; }
                if(bin_val & BIT3){ p2 |= BIT3; }
                if(bin_val & BIT4){ p2 |= BIT4; }
                if(bin_val & BIT5){ p2 |= BIT5; }
                P2OUT = p2;
        }
}
```