Instruction Manual
Programmable Flight Controller

Prepared for Reiland Systems

Internal Supervisor: Dr. Jason Gu

External Supervisor: Dr. Jason Rhinelander

Brendon Camm	Lucas Doucette	Dylan Humber
B00649176	B00685962	B00695554

Submitted: April 10, 2017

DALHOUSIE UNIVERSITY

This document was written in LaTeX. It was compiled with pdflatex. The source of the document may be viewed at:
https://github.com/Brendoncamm/SYP/edit/master/Documents/Instruction%20Manual/InstructionManual.tex

Contents

Abstract

The following instruction manual provides information regarding the design and usage of the flight controller. The required packages, libraries, installation procedures are outlined in the documentation. The function libraries provided are discussed in detail to ensure clarification.

1 Procedure

This section outlines the procedure of initializing the system as it currently stands. It is recommended that the connection to the Raspberry Pi to begin enabling the server script is done by use of Remote Desktop or an SSH connection through a terminal or PuTTY.

- Ensure Server.py and arduino.py scripts are located in the same folder on the RPi.
- Flash arduino with main.ino script.
- Connect controller to the base station computer.
- Ensure PS4Controller.py script uses the correct IP address correlating to the RPi to connect via Wi-Fi
- Provide power to the power distribution board on the drone and connect the RPi to the Arduino using a serial connection. At this time, motor initialization will begin.
- Run Server.py script to initialize the server followed by running PS4Controller.py on the base station to begin control signalling. The Arduino must be connected to the RPi before the Server.py script will run, PS4Controller.py requires the Server.py script to be running before it will run.

Please note as mentioned in the Arduino section of this manual, the drone's flight can be tested by hard coding set points instead of using a control input over the network. This is useful for quick performance testing.

2 Simulink Simulation

This section details how to open the SimuLink Simulation and how to change the existing control signals. The simulation still has a few issues which are discussed in detail within the report.

2.1 Opening the Simulation

To open the simulation, open the SimuLink project file. Once that has opened within the MatLab environment, open the Dynamic Simulation diagram. This sheet contains the entire simulation.

2.2 Running the Simulation

In its current state, the simulation is rendered through the Mechanics Explorer from Simscape. It is recommended to define test signals in the labeled section. From left to right the signals are Roll, Pitch, Yaw, and then altitude. The test signals should be normalized to between -1 and 1. This is to reflect how controllers such as USB joysticks or PS4 controllers work. Above this area is another highlighted area for limit values. These allow for the user to limit the maximum values of the control signal. For example if the altitude limit is set to 100 meters, and the control signal is at a value of 0.5, the controller will attempt to fly to 50 meters.

3 Raspberry Pi

This section details the server script for handling communications to the drone.

3.1 Required Software

The python programs outlined in this section should run natively within the Raspbian Operating system on the raspberry pi. It is required that they are run in Python3 or higher as Python 2 standard library does not contain all of the pre-requisites for the servers. Launching the server file in Python 2 will raise an exception. The Python code for this section may be examined in Appendix B.

3.2 Control Server

In this python file there are two servers based off of the *socketserver* library. The first handles the UDP packets that hold the controller signal from the GUI. Running the server file from terminal will allow this server to run on port 2222. The second server, for returning data from sensors is not enabled by default.

3.3 Communication Feedback Server

This server, included in the same file as the previous server, is a TCP server for responding to requests with a serialized dictionary containing data. To make use of this server it is required to supply it a list of functions that the server will run. Periodically it will iterate through this list and append the returned values of the function to the Data Dictionary.

4 Arduino

This section outlines the functionality of the Arduino software package and provides details with regards to minor customizability. The section also provides a list of additional requirements and references.

4.1 Required Software

The following is a list of the required library packages to use the Drone.cpp, Drone.h software. To install a library simply find the libraries folder located within the Arduino folder and unzip the downloaded libraries to that location.

- Drone library package (https://github.com/Brendoncamm/SYP/tree/master/Arduino)
- Adafruit 10 DOF library (https://github.com/adafruit/Adafruit10DOF)
- TimerOne library (https://github.com/PaulStoffregen/TimerOne)
- Arduino IDE (https://www.arduino.cc/en/main/software)

4.2 Software Breakdown

All discussed software can be found in Appendix C.

4.2.1 Main.ino Description

Currently, the main.ino file provided performs initializations of ESCs, the serial bus and the 10DOF IMU. The file then writes a byte to the serial bus which acts as a request to the Raspberry Pi to begin set point transmission. The file then collects current values from the IMU in terms of Yaw, Pitch, Roll and Altitude. PID values are calculated and combined as laid out in Simulink. The values returned from the combination are written to the motors.

The main file also includes sections currently commented out referring to interrupts. Ideally, an interrupt will be called to begin a PID calculation to ensure consistent sampling times. At this time this procedure has been unsuccessful. For testing purposes, set points can be hard coded instead of acquired from the serial connection, this is also shown as a commented section of the script.

Gain values for PID can be changed in the variable declaration section, in the future this is planned to be modifiable through the GUI. Due to time constraints and testing, several actions that would ideally be functions are directly coded in the main file such as writing values to the motors. Again, as a future recommendation functions for these actions should be created in the Drone.h and Drone.cpp files.

4.2.2 Functions Breakdown

Outlining the functions provided in the Drone library.

- Drone( ): Used to initialize all sensors, motors and communications (future).
- initSensors(): Initialize Inertial Measurement Unit.
- initESCs(Servo esc_1, Servo esc_2, Servo esc_3, Servo esc_4): Writes upper and lower control values to the ESCs for initialization.
- read_float(): Used to read bytes from the serial bus and package them as floating point values.
- send_float(); Used for testing, echoes values read from the serial bus back to the RPi.
- read_PS4Setpoints(float *PS4Yaw, float *PS4Pitch, float *PS4Roll, float *PS4Altitude): Utilizes read_float() to gather all set points available from the serial bus. This function requires further testing.
- ping_ultrasonic(): Uses ultrasonic sensor to determine altitude.
- get_currentPressure(): Takes barometric pressure readings and uses an averaging filter on the results.
- get_sensorAltitude(float startingPressure): Utilizes ping_ultrasonic() to determine altitude for altitudes less than 2.5m. Uses the startingPressure variable to compare to current barometric pressure reading and calculates relative altitude. In the future, a Kalman filter is recommended to combine the barometric pressure and accelerometer readings for a more accurate altitude.
- get_sensorRoll(): Acquires current roll reading from the IMU.
- get_sensorPitch(): Acquires current pitch reading from the IMU.
- get_sensorYaw(): Acquires current yaw reading from the IMU.

- PID_Calculate(float Setpoint, float SenseRead, float kp, float kd, float ki): Original PID loop, does not use Forward Euler, therefore not calibrated through Simulink.
- PID_FWD_EULER_Calculate(float Setpoint, float SenseRead, float kp, float kd, float ki, float Ts, float N): Forward Euler PID loop implementation.
- PR_Calibration(float PR_PID): Saturation limit requirements for pitch and roll PID loops as determined through simulations.
- Altitude_Calibration(float AltitudePID): Altitude saturation limits based upon Simulink requirements.

4.3 Additional Resources

For PID implementation, the following resources were extremely useful.

- http://www.controlsystemslab.com/doc/b4/pid.pdf
- http://controlsystemslab.com/discrete-time-pid-controller-implementation/

The 10 DOF IMU was learned through reading the following documentation.

- https://cdn-learn.adafruit.com/downloads/pdf/adafruit-10-dof-imu-breakout-lsm303-l3gd20-bmp180.pdf.

5 GUI

This section gives the user instructions on how to make use of the functionalities the GUI has to offer as well as outlines how to add additional functionality if desired. It begins by explaining the required software and libraries the user must have installed in order to use the GUI. The functionality of the pushbuttons on the GUI will be explained, the section will finish off with where to find helpful resources for QtCreator, PyQt5 and the other libraries used.

5.1 Required Software

The following is a list of the required software needed to run the GUI:

- QtDesigner (Included in QtCreator5.6)
- Python3

If your system does not have QtDesigner installed follow these steps:

1. Open your preferred browser and navigate to: https://www.qt.io/qt5-6/ and click "Download"
2. On the next page select "In-house deployment, private use, or student use" and click "Get Started"
3. On the next page select "No" and click "Get Started"

4. On the next page select "No" and click "Get Started"
5. On the next page select "Desktop/multiscreen application" and click "Get Started"
6. You should now be on a entirely different page outlining the Commerical and Open Source versions of Qt scroll down and select "Get your open source package"
7. On the next page click "Download Now" to download the installer
8. Once you have downloaded the installer run it
9. With the installer now running click "Next"
10. Enter your Qt account information, if you do not already have a Qt account you can make one within the installer enter this information and then click "Next"
11. You should now be on the Setup page on the installer click "Next"
12. Browse for an installation folder or click "Next" to use the default one provided, ensure the "Associate common file types with QtCreator" box is selected
13. On the next page click "Deselect All" at the bottom of the page and then select "Qt5.6" and "Tools" from the list. then click "Next"
14. On the next page select "I agree" and then click "Next"
15. On the next page click "Next"
16. Finally, click "Install"

If your system does not have Python3 installed follow these steps:

1. Open your preferred browser and navigate to https://www.python.org/downloads/
2. Click on "Python 3.x" where x is the version number. The version number is irrelevant as long as the first number is 3.
3. Scroll to the bottom of the page to "Files" and select the appropriate version for your operating system. For example, I am using a 64bit Windows OS I would click on "Windows x86-64 executable installer". Click on this to download the installer. Once it is downloaded run the installer
4. On the installer, click "Customize installation"
5. On the next page ensure every box is checked
6. On the next page ensure the following boxes are checked: "Associate files with Python", "Create shortcuts for installed applications", "Add Python to environment variables" and "Precompile standard library". Once this is done click "Install"

Once you have successfully installed QtCreator and Python3 you must install the following libraries using the "pip" command. This is done through the command window, open the command window and type, for example: "pip install PyQt5", this will install the latest version of PyQt5. Follow this process for each of the libraties listed below.

- PyQt5
- pyqtgraph
- numpy
- sip

5.2 Basic Functionality

This section outlines the functionality of the GUI. To switch pages simply click on the name of the page in the list on the right hand side of the GUI. The following sections outline the functionality of the pushbuttons on each of the two pages.

5.2.1 Home

- **Start**: Initializes communications between the Base Station and Raspberry Pi
- **Finish**: Closes the GUI

5.2.2 Controller

- **Update Axis**: Update the joystick Yaw, Pitch, Roll and Thrust inputs
- **Update Host**: Update the Host PC Name
- **Update Port**: Update the port number for the socket
- **Manual Control**: Initialize the Joystick to send manual control inputs
- **Update Connection**: Updates the Host and Port at the same time, upon completing this a new connection will be established using the new information

To see which definition is called when each of these buttons are clicked see the GUI code in Appendix A.1. How the Joystick initialization is coded can be viewed in Appendix A.2.

5.3 Additional Resources

If the user would like to add any additional functionality many excellent online resources exsist to aid them through the process. Below are some of the best that were used extensively when developing the GUI. They are broken down into QtCreator resources, PyQt5 Resources and Library resources.

5.3.1 QtCreator Resources

- The best resource for QtDesigner is the one provided by Qt. This can be found at http://doc.qt.io/qt-5/qtdesigner-manual.html

5.3.2 PyQt5 Resources

- A very nice video series describing many different aspects of PyQt is provided by sentdex on YouTube. The tutorial is based on PyQt4 but the principal is still valid, the only thing to remember is that when they use the ”QtGui” class in PyQt4 we use ”QtWidgets” in PyQt5. To find these videos go to YouTube and type in ”sentdex PyQt”.
- The reference guide for PyQt5 can be found at http://pyqt.sourceforge.net/Docs/PyQt5/

5.3.3 Library and general Python Resources

- For additional information regarding pygqtgraph (the library that allows for live plotting) visit http://www.pyqtgraph.org/documentation/
- For additional information regarding anything else you’re interested in for Python visit https://docs.python.org/3/library/index.html.

A GUI

A.1 GUI Code

Listing 1: GUI.py

```
import sys
from PyQt5 import QtCore, QtGui, uic, QtWidgets
import numpy as np
import time
import struct
import socket
from PS4_Controller import PS4Controller as PS4
import pyqtgraph as pg
import pyqtgraph.exporters

qtCreatorFile = "GUI.ui" # Enter QtDesigner file here.
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)



class GUI(QtWidgets.QMainWindow, Ui_MainWindow, QtWidgets.QMenu):
def __init__(self):

super(GUI,self).__init__()
#Qt initialization
QtWidgets.QMainWindow.__init__(self)
Ui_MainWindow.__init__(self)
self.page = QtWidgets.QStackedWidget()
self.setCentralWidget(self.page)
self.setupUi(self)
self.setWindowTitle("Drone")
self.setWindowIcon(QtGui.QIcon('smu.png'))

#Networking
self.getHost = socket.gethostname()
self.staticPort = '1247'

#Main Page buttons
self.start.clicked.connect(self.connection)
self.end.clicked.connect(self.stop)

#Listing widget, allows for the user to select a certain page
self.list.insertItem(0, 'Home')
self.list.insertItem(1, 'Controller')
self.list.currentRowChanged.connect(self.display) #Changes widget index to appropriate page

#Controller Page
self.axisVal.setText('1_2_3_4')
self.hostVal.setText(self.getHost)
self.portVal.setText(self.staticPort)
self.axisMenu.clicked.connect(self.axisSettings) #When "Update Axis" is clicked call definition axisSettings
self.hostMenu.clicked.connect(self.hostSettings) #When "Update host" is clicked call definition hostSettings
self.portMenu.clicked.connect(self.portSettings) #When "Update Port" is clicked call definition portSettings
self.updateConnect.clicked.connect(self.updateConnection) #When "Update Connection" is clicked call definition updateConnec
self.connectPS4.clicked.connect(self.connectController) #When "Manual Control" is clicked call definition connectController

#Live plotting Initializations
self.initplt()
self.plotcurve = pg.PlotDataItem()
self.plotwidget.addItem(self.plotcurve)
self.t = 0
self.update1()
self.timer = pg.QtCore.QTimer()
self.timer.timeout.connect(self.move)# Connects a timer to the "move" definition that allows for live plotting
self.timer.start(1000) # Poll for updates of new data ever 1000miliseconds (1 second)

def stop(self):
sys.exit(app.exec_())

def connection(self):
s = socket.socket()
host = self.getHost
port = int(self.staticPort)
status = s.connect_ex((host,port)) #Returns 0 if connect is successful, returns errno if not
if status: # Status = errno
self.thisworks.setText("Connection_Unsuccessful")
self.connectionStat.setText("Communications_have_not_been_established")
else: # Status = 0
print(status)
self.thisworks.setText("Connection_Successful")
self.connectionStat.setText("Communications_are_active")

def axisSettings(self):
cont = PS4()
#Input boxes when "Update Axis" is clicked
text, ok = QtWidgets.QInputDialog.getText(self, 'Axis_Value[0]', 'No_Spaces')
text1, ok = QtWidgets.QInputDialog.getText(self, 'Axis_Value[1]', 'No_Spaces')
```

```
text2, ok = QtWidgets.QInputDialog.getText(self, 'Axis Value[2]', 'No Spaces')
text3, ok = QtWidgets.QInputDialog.getText(self, 'Axis Value[3]', 'No Spaces')
axis = [int(text), int(text1), int(text2), int(text3)] # Make an array of the values from the input dialogs
self.axisVal.setText(str(axis))
cont.axis_order = axis # Set the axis order for the controller equal to the new settings

def display(self,i):
self.home.setCurrentIndex(i)

def hostSettings(self):
#Input box when "Update Host" is clicked
text, ok = QtWidgets.QInputDialog.getText(self,'Host', 'Host name or IP address')
newHost = str(text)
if newHost == '':
self.hostVal.setText(self.getHost)
else:
self.hostVal.setText(newHost)
return newHost

def portSettings(self):
#Input box when "Update Port" is clicked
text, ok = QtWidgets.QInputDialog.getText(self,'Port','Port number')
if ok:
print('success')
newPort = str(text)
if newPort == '':
self.portVal.setText(self.staticPort)
else:
self.portVal.setText(newPort)
return int(newPort)

def connectController(self):
new = PS4()
cont.axis_order = self.axisSettings()
print(str(cont.axis_order))
new.listen() #Accept data from the manual controller, calls the listen definition from PS4Controller.py

def updateConnection(self):
host1 = self.hostSettings()
port1 = self.portSettings()
s = socket.socket()
status = s.connect_ex((host1,port1)) #Returns 0 if connect is successful, returns errno if not

if status: #status = errno
self.connectionStat.setText("Update and Connection Unsuccessful")
self.thisworks.setText("Update and Connection Unsuccessful")
else: #status = 0
self.connectionStat.setText("Update and Connection Successful")
self.thisworks.setText("Update and Connection Successful")

def liveData(self):
graph_data = open('test.txt', 'r').read() # Open the text file to read in data
lines = graph_data.split('\n') # Read in data from different lines
xs = [] #Empty list
ys = [] #Empty List
for line in lines:
if len(line)>1:
x, y = line.split(' ,') #Read in data in the form of (x ,y)
xs.append(int(x))
ys.append(int(y))
return xs,ys

def initplt(self):
self.plotwidget = pg.PlotWidget() #Initate the plotting widget field
self.mplvl.addWidget(self.plotwidget) #Set the plotting widget field to populate the QVBoxLayout widget field
self.plotwidget.setLabel('left', 'Altitude [m]') # Y-Axis name
self.plotwidget.setLabel('bottom','Time[s]') #X-Axis name
self.show()

def update1(self):
#read in the data from the liveData definition in the form of two separate lists.
#list1 = x-values, list2 = y-values
list1,list2 = self.liveData()
self.plotcurve.setData(list1,list2) # Plot the data

def move(self):
self.t+=1 #Move the data 1 spot to the right
self.update1() # Call update1 definition to get the new data


if __name__ == '__main__':
app = QtWidgets.QApplication(sys.argv)
main = GUI()
main.show()
QtWidgets.QApplication.processEvents()
sys.exit(app.exec_())
```

A.2 Joystick Code

Listing 2: PS4Controller.py

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
#
# This file presents an interface for interacting with the Playstation 4 Controller
# in Python. Simply plug your PS4 controller into your computer using USB and run this
# script!
#
# NOTE: I assume in this script that the only joystick plugged in is the PS4 controller.
#       if this is not the case, you will need to change the class accordingly.
#
# Copyright    2015 Clay L. McLeod <clay.l.mcleod@gmail.com>
#
# Distributed under terms of the MIT license.

#TODO:
#   rewrite connection for new server
#   test

# import os
# import pprint
import pygame
import socket
import struct
import sys

if sys.version_info[0] < 3:
raise Exception('Lucas', 'not_compatible_with_Python_version_2')


class PS4Controller(object):
"""Class representing the PS4 controller. Pretty straightforward functionality."""

controller = None
axis_data = None
button_data = None
hat_data = None

def __init__(self, axis_order=[1, 2, 3, 4], hostname='raspberrypi', port=2222, limits = [10, 10, 1, 10]):
"""Initialize the joystick components"""

pygame.init()
pygame.joystick.init()
self.controller = pygame.joystick.Joystick(0)
self.controller.init()
self.hostname = hostname
self.port = port
self.limits = limits
if isinstance(axis_order, list):
self.axis_order = axis_order  # For changing how controller axes are bound
else:
raise Exception(TypeError, 'axis_order_must_be_list.')

def update_axes(self, axis_order):
self.axis_order = axis_order

def listen(self):
"""Listen for events to happen"""

if not self.axis_data:
self.axis_data = {0: float(0),
1: float(0),
2: float(0),
3: float(0),
4: float(-1),
5: float(-1)}  # Added explicity number of axes to avoid waiting for input

if not self.button_data:
self.button_data = {}
for i in range(self.controller.get_numbuttons()):
self.button_data[i] = False

if not self.hat_data:
self.hat_data = {}
for i in range(self.controller.get_numhats()):
self.hat_data[i] = (0, 0)


# host = '192.168.2.19' #ip of Server (PI)
host = socket.gethostbyname(self.hostname)  # if fails install samba on pi and reboot

while True:
for event in pygame.event.get():
if event.type == pygame.JOYAXISMOTION:
self.axis_data[event.axis] = round(event.value, 2)
elif event.type == pygame.JOYBUTTONDOWN:
self.button_data[event.button] = True
```

```
elif event.type == pygame.JOYBUTTONUP:
self.button_data[event.button] = False
elif event.type == pygame.JOYHATMOTION:
self.hat_data[event.hat] = event.value

# Insert your code on what you would like to happen for each event here!
# In the current setup, I have the state simply printing out to the screen.

# Defining Variables to send through the socket to the RPi, need to be strings

# axis_data=str(self.axis_data)
# button_data = str(self.button_data)
# hat_data = str(self.hat_data)

# Sending Data over a socket to the RPi
# print(str(self.axis_data))
# Isolate desired Axes

axes_data = [self.axis_data[self.axis_order[0]] * self.limits[0],
self.axis_data[self.axis_order[1]] * self.limits[1],
self.axis_data[self.axis_order[2]] * self.limits[2],
self.axis_data[self.axis_order[3]] * self.limits[3]]
byte_data = []  # To hold the axes data serialized to bytes
for axis in axes_data:
byte_data.append(struct.pack("f", axis))  # F for float

# Send the control input data in byte form over the to be sent over the socket
xmission_bytes = bytes().join(byte_data)
connection = socket.socket()
connection.connect((host, self.port)) # Make the connection to the RPi
connection.send(xmission_bytes)  # sending the controller data over the port
connection.close() # Whenever no control inputs, close socket
# print(xmission_bytes)

# os.system('cls')
# break
# s.send(button_data)
# s.send(hat_data)
# s.close()


if __name__ == "__main__":
ps4 = PS4Controller()
# ps4.init()
ps4.listen()
```

B Server Code

```
import socketserver
import sys
import arduino
import threading
import queue
from pickle import dumps as serialize
from subprocess import check_output

class SerialRequestHandler(threading.Thread):
    """
    Thread object for handling the serial bus connection.  When the arduino controller writes to the serial bus to
    signal    that it is ready for control data, the thread clears the serial bus and then writes the current state from
    the queue it was initialized with.

    :param stateq: Queue object for holding control state.  The queue should be of size one.  It is being used for
    its multi-thread implementation over its properties as a queue.
    """
    def __init__(self, stateq):
        """
        Initialize thread object.
        """
        super(self.__class__, self).__init__()
        self.stateq = stateq

    def run(self):
        """
        Creates object for handling the serial bus to the arduino and then writes to it when signaled to.
        """
        sbus = arduino.Arduino_Controller(9600)
        while True:
            ready = sbus.ready()
            if ready:
                sbus.serial_bus.read(ready)
                sbus.serial_bus.write(self.stateq.get())


class QuadControlHandler(socketserver.BaseRequestHandler):
    """
    Request handler that puts 16 received bytes into the queue for the SerialRequestHandler.

    :param request: Inherited from BaseRequestHandler
```

```
    :param client_address: Inherited from BaseRequestHandler
    :param server: Inherited from BaseRequestHandler
    :param stateq: Threading queue object of size 1.
    """
    def __init__(self, request, client_address, server, stateq):
        """
        Initialize request handler.
        """
        self.stateq = stateq
        super(self.__class__, self).__init__(request, client_address, server)
        return

    def handle(self):
        """
        Handles connection.  If queue is full it empties it, then reads the received bytestring to the queue.
        """
        if self.stateq.full():
            self.stateq.get() #Queue has size of 1, if full clear for new state
        recv = self.request.recv(16)
        self.stateq.put(recv)
        print('Received: ' + str(recv))
        return

class QuadControlServer(socketserver.UDPServer):
    """
    Handles UDP datagrams containing 16 Bytes

    :param server_address: Inherited from UDPServer, a tuple of (Address, Port)
    :param RequestHandlerClass: Should be QuadControlHandler
    """
    def __init__(self, server_address, RequestHandlerClass):
        """
        Initialize controller.
        """
        super(self.__class__, self).__init__(server_address, RequestHandlerClass)
        self.stateq = queue.Queue(1)
        return

    def serve_forever(self, poll_interval=0.5):
        """
        Overridden from UDPServer, added creating of SerialRequestHandler and start of that thread.
        """
        thread = SerialRequestHandler(self.stateq)
        thread.start()
        super(self.__class__, self).serve_forever(poll_interval)
        return

    def finish_request(self, request, client_address):
        """
        Overriden from UDPServer.  Adds passing queue to the request handler.
        """
        self.RequestHandlerClass(request, client_address, self, self.stateq)

class FeedbackCommHandler(socketserver.BaseRequestHandler):
    """
    Handles data requests from the GUI.
    """
    def __init__(self, request, client_address, server, data):
        """
        Overridden to bring datadictionary to request handler.
        """
        self.DataDictionary = data
        super(self.__class__, self).__init__(request, client_address, server)

    def handle(self):
        """
        Pickles datadictionary and sends it as a response.
        """
        byte_string = serialize(self.DataDictionary)
        self.request.send_all(byte_string)

class CommunicationServer(socketserver.TCPServer):
    """
    Server inherited from TCPServer, handles request for data as well as acquiring data.  Periodically runs functions
    from a list of DataFunctions and appends their returned values/object to a list of data.

    :param server_address: Inherited from UDPServer, a tuple of (Address, Port)
    :param RequestHandlerClass: Should be FeedbackComHandler
    :param DataFuntions:  A list of functions that acquire data.
    """
    def __init__(self, server_address, RequestHandlerClass, DataFuntions):
        """
        Initializes server.
        """
        self.DataFunctions = DataFuntions
        self.DataDictionary = {}
        for x in self.DataFunctions:
            self.DataDictionary[x[0]] = {'name' : x[0],
                                         'function' : x[1],
                                         'data' : []
                                        }
        super(self.__class__, self).__init__(server_address, RequestHandlerClass)
```

```python
    def service_actions(self):
        """
        Iterates through list of functions in self.DataFunctions and records them in the corresponding dictionary within
        self.DataDictionary.
        """
        for x in self.DataDictionary:
            x['data'].append(x['function']())

    def finish_request(self, request, client_address):
        """
        Overriden from TCPServer.  Adds passing data to the request handler.
        """
        self.RequestHandlerClass(request, client_address, self, self.DataDictionary)

if __name__ == '__main__':
    if sys.version_info[0] < 3:
        raise Exception('Version Error', 'Not compatible with Python version 2')

    HOST = check_output(['hostname', '-I']).strip()
    CONTROL_PORT = 2222
    COMMS_PORT = 4444

    test_serv = QuadControlServer((HOST, CONTROL_PORT), QuadControlHandler)
    test_serv.serve_forever()
```

C Arduino Library

Listing 3: Drone.h Arduino Header File

```cpp
/*
  Drone.h - Library for Drone Controller.
  Created by Lucas Doucette, February 17, 2017.
*/
#ifndef Drone_h
#define Drone_h

#include "Arduino.h"
#include <Wire.h>
#include <Servo.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_LSM303_U.h>
#include <Adafruit_BMP085_U.h>
#include <Adafruit_L3GD20_U.h>
#include <Adafruit_10DOF.h>

class Drone
{
  public:
    Drone();
    float get_sensorRoll();
    float get_sensorPitch();
    float get_sensorYaw();
    float get_sensorAltitude(float startingPressure);
    float get_currentPressure();
    float PID_Calculate(float Setpoint, float SenseRead, float kp, float kd, float ki );
    float PID_FWD_EULER_Calculate(float Setpoint, float SenseRead, float kp, float kd, float ki, float Ts, float N);
    float Altitude_Calibration(float AltitudePID);
    float PR_Calibration(float PR_PID);
    void read_PS4Setpoints(float *PS4Yaw, float *PS4Pitch, float *PS4Roll, float *PS4Altitude);
    void initSensors();
    void initESCs(Servo esc_1, Servo esc_2, Servo esc_3, Servo esc_4);
    float read_float();
    void send_float(float arg);
    float ping_ultrasonic();
};

#endif
```

Listing 4: Drone.cpp Arduino Library File

```
/*
  Drone.h - Library for Drone Controller.
  Created by Lucas Doucette, February 17, 2017.
*/

#include "Arduino.h"
#include "Drone.h"
#include <Wire.h>
#include <Servo.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_LSM303_U.h>
#include <Adafruit_LSM303.h>
#include <Adafruit_BMP085_U.h>
#include <Adafruit_L3GD20_U.h>
#include <Adafruit_10DOF.h>
#include <LiquidCrystal.h>

  /* Assign a unique ID to the sensors */
Adafruit_10DOF dof   = Adafruit_10DOF();
Adafruit_LSM303_Accel_Unified accel = Adafruit_LSM303_Accel_Unified(30301);
Adafruit_LSM303_Mag_Unified   mag   = Adafruit_LSM303_Mag_Unified(30302);
Adafruit_BMP085_Unified       bmp   = Adafruit_BMP085_Unified(18001);

sensors_event_t accel_event;
sensors_event_t mag_event;
sensors_event_t bmp_event;
sensors_vec_t   orientation;

float temperature;

Drone::Drone()
{

}

float Drone::ping_ultrasonic()
{


  long duration, m;



    //send low high low to ensure a clean ping

  pinMode(7, OUTPUT);
  digitalWrite(7, LOW);
  delayMicroseconds(2);
  digitalWrite(7, HIGH);
  delayMicroseconds(5);
  digitalWrite(7, LOW);


  pinMode(8, INPUT);
  duration = pulseIn(8, HIGH);


  //29 microseconds per cmeter, and round trip
  m=(duration/29)/2;

    if(m>1500)
    {

        m=0;

    }

  return m;




}

float Drone::get_sensorRoll()
{

    float sensorRoll;


  // Calculate pitch and roll from the raw accelerometer data

  accel.getEvent(&accel_event);
  if (dof.accelGetOrientation(&accel_event, &orientation))
  {
    // 'orientation' should have valid .roll and .pitch fields
    sensorRoll=orientation.roll;
    return sensorRoll;
```

```
  }
}

float Drone::get_sensorPitch()
{

  float sensorPitch;

  // Calculate pitch and roll from the raw accelerometer data

  accel.getEvent(&accel_event);
  if (dof.accelGetOrientation(&accel_event, &orientation))
  {
    // 'orientation' should have valid .roll and .pitch fields

     sensorPitch=orientation.pitch;
     return sensorPitch;

  }
}

float Drone::get_sensorYaw()
{

  float sensorYaw;

  // Calculate the heading using the magnetometer
  mag.getEvent(&mag_event);
  if (dof.magGetOrientation(SENSOR_AXIS_Z, &mag_event, &orientation))
  {
    // 'orientation' should have valid .heading data now
    sensorYaw=orientation.heading;
    return sensorYaw;

  }
}



float Drone::get_sensorAltitude(float startingPressure)
{


    float sensorAltitude, NowPressure;

    // kalman filtering variables

    float Variance=0.108923435;
    float varProcess = 1e-12;
    float Pc = 0.0;
    float G = 0.0;
    float P = 1.0;
    float Xp = 0.0;
    float Zp = 0.0;
    float Xe = 0.0;



    // Calculate the altitude using the barometric pressure sensor
    bmp.getEvent(&bmp_event);
    NowPressure=bmp_event.pressure;

    // Get ambient temperature in C
    bmp.getTemperature(&temperature);


    // Convert atmospheric pressure, SLP and temp to altitude
    sensorAltitude=bmp.pressureToAltitude(startingPressure,NowPressure,temperature);


    // kalman filter;
    Pc=P+varProcess;
    G=Pc/(Pc+Variance);
    P=(1-G)*Pc;
    Xp=Xe;
    Zp=Xp;
    Xe=ping_ultrasonic()/100;

    if (Xe >= 2.0)
    {
        //Enable barometric pressure above 2 meters
        Xe=G*(sensorAltitude-Zp)+Xp;
    }


    return Xe;



}
```

```
//must be called after initSensors();

float Drone::get_currentPressure()
{
    float currentPressure;

   //Initializing Pressure, Filtered for accuracy
   //Smooting Constant

   const int numReadings=1;
   float SmoothingVariable=0;
   float NowPressure=0;

    bmp.getEvent(&bmp_event);
    for(int i=0; i<numReadings; i++)
    {
        SmoothingVariable+=bmp_event.pressure;
        //delay(1);

    }

    currentPressure = SmoothingVariable/numReadings;
    SmoothingVariable=0;

    return currentPressure;

}

float Drone::PID_Calculate(float Setpoint, float SenseRead, float kp, float kd, float ki)
{

    //Define PID Variables
    float Error, DError, IError, LastTime, LastError, Output;

    //find current time
    unsigned long NowTime = millis();

    //calculate PID Error values
    Error = Setpoint-SenseRead;
    DError = (Error-LastError)/(NowTime-LastTime);
    IError += (Error*(NowTime-LastTime));

    //Calculate Output
    Output = kp*Error+ki*IError+kd*DError;

    //Save LastTime and LastError
    LastTime=NowTime;
    LastError=Error;


    //Requirement based on Simulink

    Output=(Output/9000)*(1860-1127);

    if(Output>=1200){

        Output = 1200;

    }

    if(Output<=1127)
    {
        Output=1127;
    }

    return Output;


}


float Drone::PID_FWD_EULER_Calculate(float Setpoint, float SenseRead, float kp, float kd, float ki, float Ts, float N)
{

    float e2, e1, e0, u2, u1, u0; // Variables for PID computation
    float a0, a1, a2, b0, b1, b2, ku1, ku2, ke0, ke1, ke2;


    // hand calculated forward euler coefficients
    a0 = (1 - N*Ts);
    a1 = (N*Ts - 2);
    a2 = (1);
    b0 = (kp*(1 - N*Ts)+ki*(N*Ts-1)+kd*N);
    b1 = (kp*(N*Ts-2)+ki*Ts-2*kd*N);
    b2 = (kp+kd*N);
    ku1 = a1/a0; ku2 = a2/a0; ke0 = b0/a0; ke1 = b1/a0; ke2 = b2/a0;

    /* Backward Euler for testing
    a0 = (1+N*Ts);
    a1 = -(2 + N*Ts);
    a2 = 1;
    b0 = kp*(1+N*Ts) + ki*Ts*(1+N*Ts) + kd*N;
```

```
    b1 = -(kp*(2+N*Ts) + ki*Ts + 2*kd*N);
    b2 = kp + kd*N;
    ku1 = a1/a0; ku2 = a2/a0; ke0 = b0/a0; ke1 = b1/a0; ke2 = b2/a0;
    */

    //update variables
    e2=e1;
    e1=e0;
    u2=u1;
    u1=u0;

    //calculate error and output
    e0 = Setpoint - SenseRead;
    u0 = - ku1*u1 - ku2*u2 + ke0*e0 + ke1*e1 + ke2*e2;


    //return output
    return u0;

}

float Drone::Altitude_Calibration(float AltitudePID)

{
    AltitudePID=(AltitudePID/9000)*(1860-1127);

    if (AltitudePID > 1860)
    {
        AltitudePID = 1860;
    }

    if (AltitudePID < 1127)
    {
        AltitudePID = 1127;
    }

    return AltitudePID;

}


float Drone::PR_Calibration(float PR_PID)

{
    if (PR_PID > 4500)
    {
        PR_PID = 4500;
    }

    if (PR_PID < -4500)
    {
        PR_PID = -4500;
    }

    return (PR_PID/9000);


}


void Drone::read_PS4Setpoints(float *PS4Yaw, float *PS4Pitch, float *PS4Roll, float *PS4Altitude)

{

    //Write serially to Pi to begin transmission.

    byte XMIT = 00000001;

    if (Serial.available())
    {
       Serial.write(XMIT);
    }



    while (Serial.available())
    {
        for(int i=0; i<4; i++)
        {
            *PS4Yaw=read_float();

        }

        for(int i=4; i<8; i++)
        {
            *PS4Pitch=read_float();

        }

        for(int i=8; i<12; i++)
        {
```

```
            *PS4Roll=read_float();

        }

        for(int i=12; i<16; i++)
        {
           *PS4Altitude=read_float();

        }

    }



}

void Drone::initSensors()
{

  if(!accel.begin())
  {
    /* There was a problem detecting the LSM303 ... check your connections */
    Serial.println(F("Ooops, no LSM303 detected ... Check your wiring!"));
    while(1);
  }
  if(!mag.begin())
  {
    /* There was a problem detecting the LSM303 ... check your connections */
    Serial.println("Ooops, no LSM303 detected ... Check your wiring!");
    while(1);
  }
  if(!bmp.begin())
  {
    /* There was a problem detecting the BMP180 ... check your connections */
    Serial.println("Ooops, no BMP180 detected ... Check your wiring!");
    while(1);
  }

}

void Drone::initESCs(Servo esc_1, Servo esc_2, Servo esc_3, Servo esc_4)
{


  //Initialization of the ESCs
  esc_1.writeMicroseconds(1860);
  esc_2.writeMicroseconds(1860);
  esc_3.writeMicroseconds(1860);
  esc_4.writeMicroseconds(1860);
  delay(3000);
  esc_1.writeMicroseconds(1060);
  esc_2.writeMicroseconds(1060);
  esc_3.writeMicroseconds(1060);
  esc_4.writeMicroseconds(1060);
  delay(3000);


}

void Drone::send_float (float arg) {
  byte * data = (byte *) &arg;
  Serial.write(data, sizeof(arg));
}

float Drone::read_float () {
 union{
   float a;
   unsigned char bytes[4];
 } data;
 for (int i=0; i<4; i++) {
   data.bytes[i] = Serial.read();
 }
 float test = data.a;
 return(test);
}
```

C.1 Arduino Main File

Listing 5: Main.ino Drone Flight Program

```
#include <Drone.h>
#include "TimerOne.h"

Drone drone;
Servo esc_1, esc_2, esc_3, esc_4;
int MotorPin1=6, MotorPin2=9, MotorPin3=5, MotorPin4=3;
float Yaw_Setpoint, Pitch_Setpoint, Roll_Setpoint, Altitude_Setpoint;
float Current_Yaw, Current_Pitch, Current_Roll, Current_Altitude;
float initial_pressure;
float kp_alt=17257.712, ki_alt=2752.7, kd_alt=12500, N_alt=20;
float kp_PR=-24.3498729396233, ki_PR=-15.1642535450127, kd_PR=-7.30148358076729, N_PR=44.9852981416911;
float AltitudePID, PitchPID, RollPID;
float Ts = 0.0001;
float Alt_Out, Pitch_Out, Roll_Out;
float Motor1_Output, Motor2_Output, Motor3_Output, Motor4_Output;


void setup() {

  esc_1.attach(MotorPin1);  //set up motor connections & Initialize
  esc_2.attach(MotorPin2);
  esc_3.attach(MotorPin3);
  esc_4.attach(MotorPin4);
  drone.initESCs(esc_1, esc_2, esc_3, esc_4);

  //Initialize Serial Bus and IMU Sensors

  Serial.begin(9600);
  drone.initSensors();
  initial_pressure=drone.get_currentPressure();

   //initialize timer and interrupt callback
  //Timer1.initialize(100);
  //Timer1.attachInterrupt(PID_Interrupt);



}


//Commented until troubleshooting is complete
/*void PID_Interrupt(){

AltitudePID = drone.PID_FWD_EULER_Calculate(Altitude_Setpoint, Current_Altitude, kp, kd, ki, Ts);

 esc_1.writeMicroseconds(AltitudePID);
 esc_2.writeMicroseconds(AltitudePID);
 esc_3.writeMicroseconds(AltitudePID);
 esc_4.writeMicroseconds(AltitudePID);

}

*/
void loop() {


//Get Control Setpoints
if (Serial.available()==0)
{
  Serial.print('c');
}

while (Serial.available()>0)
  {
    Yaw_Setpoint = drone.read_float();
    Pitch_Setpoint = drone.read_float();
    Altitude_Setpoint = drone.read_float();
    Roll_Setpoint = drone.read_float();
  }

/*
Altitude_Setpoint = 0.50;
Pitch_Setpoint=0;
Roll_Setpoint=0;
*/

//Gather IMU Data
Current_Yaw = drone.get_sensorYaw();
Current_Pitch = drone.get_sensorPitch();
Current_Altitude = drone.get_sensorAltitude(initial_pressure);
Current_Roll = drone.get_sensorRoll();

//Calculate PID's and calibrate outputs based on simulink requirements.

AltitudePID = drone.PID_FWD_EULER_Calculate(Altitude_Setpoint, Current_Altitude, kp_alt, kd_alt, ki_alt, Ts, N_alt);
```

```
Alt_Out = drone.Altitude_Calibration(AltitudePID);

PitchPID = drone.PID_FWD_EULER_Calculate(Pitch_Setpoint, Current_Pitch, kp_PR, kd_PR, ki_PR, Ts, N_PR);
Pitch_Out=drone.PR_Calibration(PitchPID);

RollPID = drone.PID_FWD_EULER_Calculate(Roll_Setpoint, Current_Roll, kp_PR, kd_PR, ki_PR, Ts, N_PR);
Roll_Out = drone.PR_Calibration(RollPID);


//PID Combination as required by simulink

Motor1_Output = Alt_Out + Pitch_Out - Roll_Out;
Motor2_Output = Alt_Out - Pitch_Out - Roll_Out;
Motor3_Output = Alt_Out + Pitch_Out + Roll_Out;
Motor4_Output = Alt_Out - Pitch_Out + Roll_Out;


//Output PID Results to motors

 esc_1.writeMicroseconds(Motor1_Output);
 esc_2.writeMicroseconds(Motor2_Output);
 esc_3.writeMicroseconds(Motor3_Output);
 esc_4.writeMicroseconds(Motor4_Output);


}
```