www.smeedaviation.com

Technical Manual

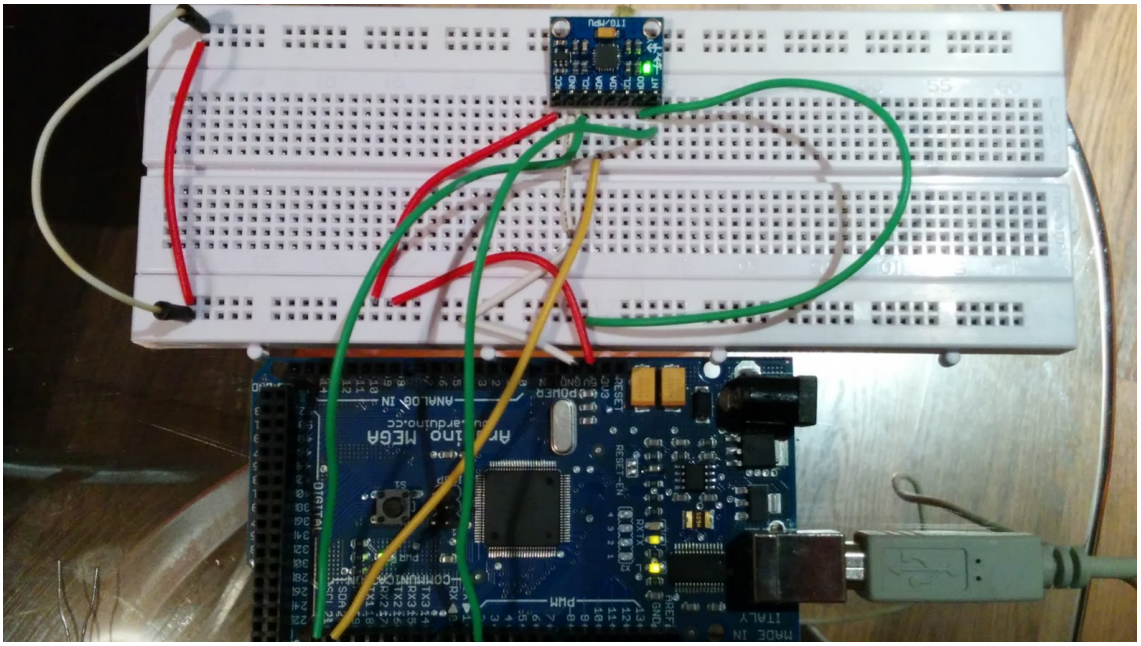
The plan from the beginning of the project was to purchase a pre-built Quadcopter and to disassemble then reconstruct it with our own microcontroller replacing the initial control unit. We spent a few weeks deciding on which pre-built Quadcopter had the components necessary to fulfil the aims of our project, chose this model, (http://www.banggood.com/WLtoys-V262-Cyclone-2_4G-4CH-6-Axis-RC-Quadcopter-With-Camera-RTF-p-77720.html) and ordered it in mid-November.
After we placed the order, we were informed that it was being shipped from China and we raised some concerns that it may face some trouble passing through customs due to the the nature of some of the electrical components. Despite being assured the delivery would be made within 20 days, 3 weeks after we placed the order we were informed that the product had, in fact, been rejected at customs because of the battery included. The company offered to resend it, which seemed futile as the same thing would probably just happen again. This was quite a major setback to our project, as we had specifically chosen this Quadcopter due to its suitability to the

aims of our project. We had also lost a significant time waiting for the Quadcopter which we hadn't fully accounted for in our project plan.

By mid December, we were behind schedule and had to act quickly to obtain a Quadcopter we could work with and make some progress on our project. The model we ended up picking was (http://www2.ripmax.net/Item.aspx?ItemID=A-U817), which we purchased from a High Street retailer. We tested the Quadcopter to make sure it worked properly before we removed the control Unit.

An important component of our system was the Gyroscope/Accelerometer, which accurately measures the angles of direction on the chip, and also the force of gravity and momentum. To give our Quadcopter the opportunity to maintain stability, we needed to be able to interpret the serialised output from the gyroscope and, eventually, control the motors based on the readings from the gyroscope.

Reading the gyroscope and accelerometer



Hardware

After researching the best value 3-axis gyroscope and accelerometer we decided to use the InvenSense MPU-6050. We purchased the GY-521 breakout board which allows us to easily interface with the MPU-6050 using the I2C bus. The wiring for this was easy, this is how we did it:

GY-521 connection	Arduino Mega connection

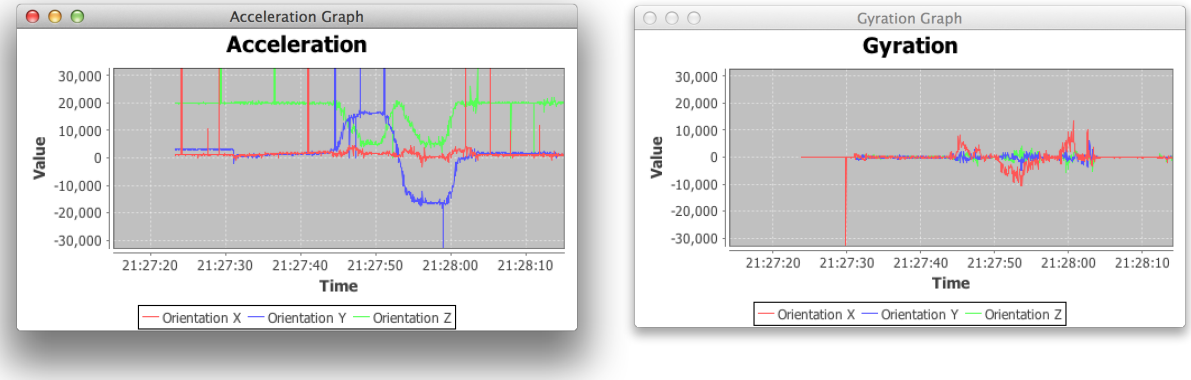
VCC	5v power source
GND	Gound
SCL	SCL pin 21
SDA	SDA pin 20
ADO	Gound

Software

As this sensor is very popular in embedded systems, there is a tutorial and source code available on the Arduino website http://playground.arduino.cc/Main/MPU-6050. We followed this tutorial and modified the sketch to output to serial the sensor values in the format we wanted (ax:0;ay:0;az:0;gx:0;gy:0;gz:0;).

Visualisation

To better understand the gyroscope and accelerometer data we used a java charting library called JFreeChart to plot real time line graphs of accelerometer x, y and z values and the gyroscope x, y and z values. Holding the quadcopter and slowly pivoting it around its Y axis 90° in one direction, back to level, and then 90° in the other direction produces accelerator and and gyroscope data as seen below.



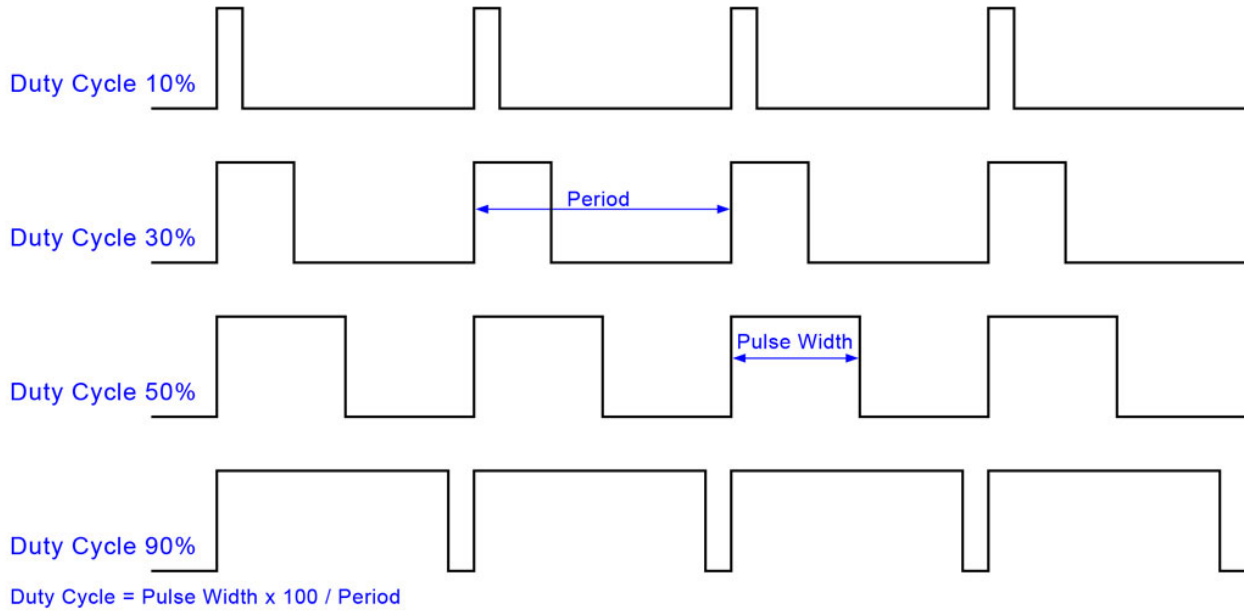
These graphs were produced when the quadcopters rotors were turned off. When the rotors were turned on the accelerometer data was hard to interpret because the vibrations caused big fluctuations. Utilising the graphs we clearly identified that the accelerometer is affected by vibrations and the gyroscope tends to drift over time.

The accelerometer and gyroscope data had to be combined to reduce the inherent negative effects of both sensors. The

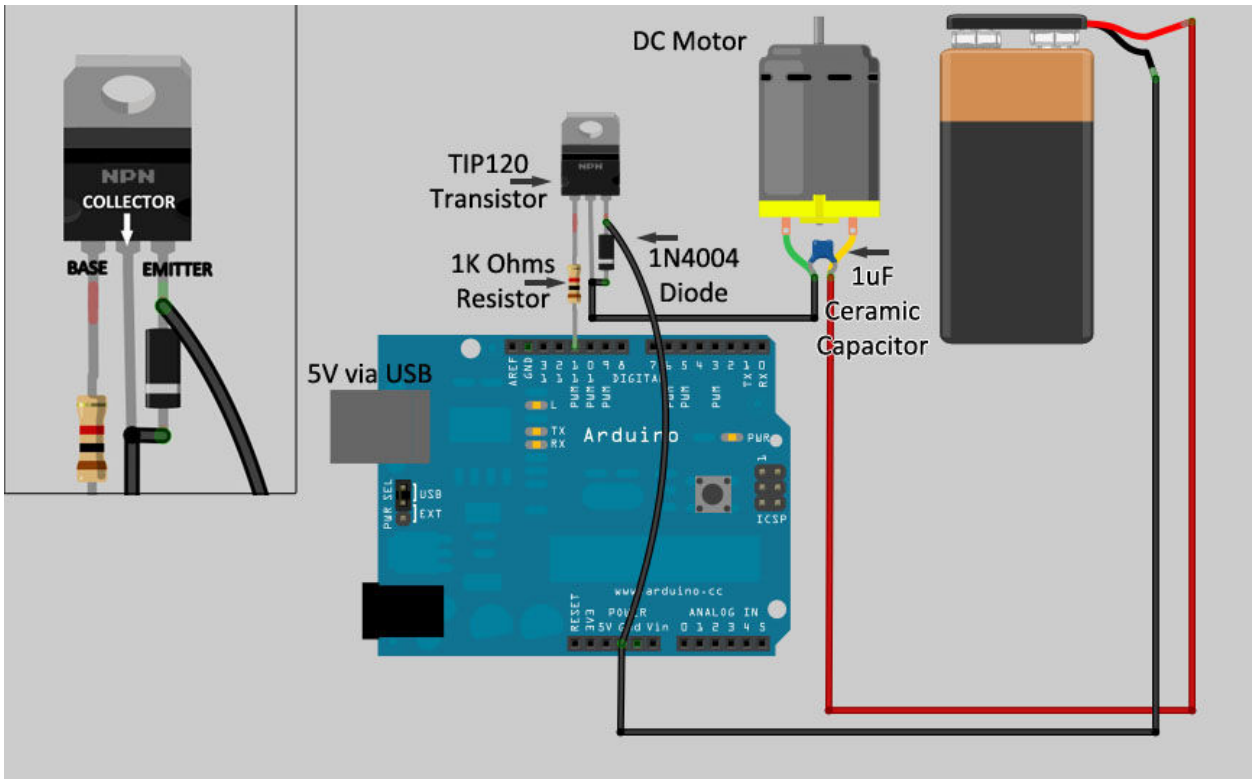
Controlling the motors

Hardware

Fine grain control of the each motor of the quadcopter is naturally required for stability and maneuverability. The way to achieve this is by using pulse width modulation (PWM). PWM switches the power on and off very quickly to adjust the speed of the motors. Full power using PWN would have the pulse width (Duty Cycle) being the same width as the period, zero power would have a duty cycle of 0%.



The Arduino Mega has 12 pins that support PWM, these are pins 2-13. Connecting the motors to ground and a PWN pin although simple is flawed. That approach would not protect the arduino from sparks caused by the bush motors, also would not provide enough power to the motors, and finally, does not protect against a possible high voltage spike called flyback. Flyback occurs when high voltage is suddenly removed from an inductive circuit like a motor. To solve this we used the PWM coming out from the Arduino to switch on and off a more powerful circuit using TP-120 transistors. To remove the possible sparks coming from the bushes in the motors, capacitors are added between the positive and negative of the motor. Finally to protect the transistor from flyback, a diode is added between the collector and emitter. This diode is often referred to as a flyback diode.



This circuit was reproduced 4 times to control each motor.

Arduino pin	Motor
8	Front Left
9	Front Right
10	Back Right
11	Back Left

Software

With the high power motor circuit wired up, we modified the Arduino sketch to set the PWM pins over the serial communication. We decided on the format “W,X,Y,Z;” where W,X,Y,Z values are between 0-255 with 0 being 0% duty cycle and 255 being 100%. These map to the pins 8-11 respectively.

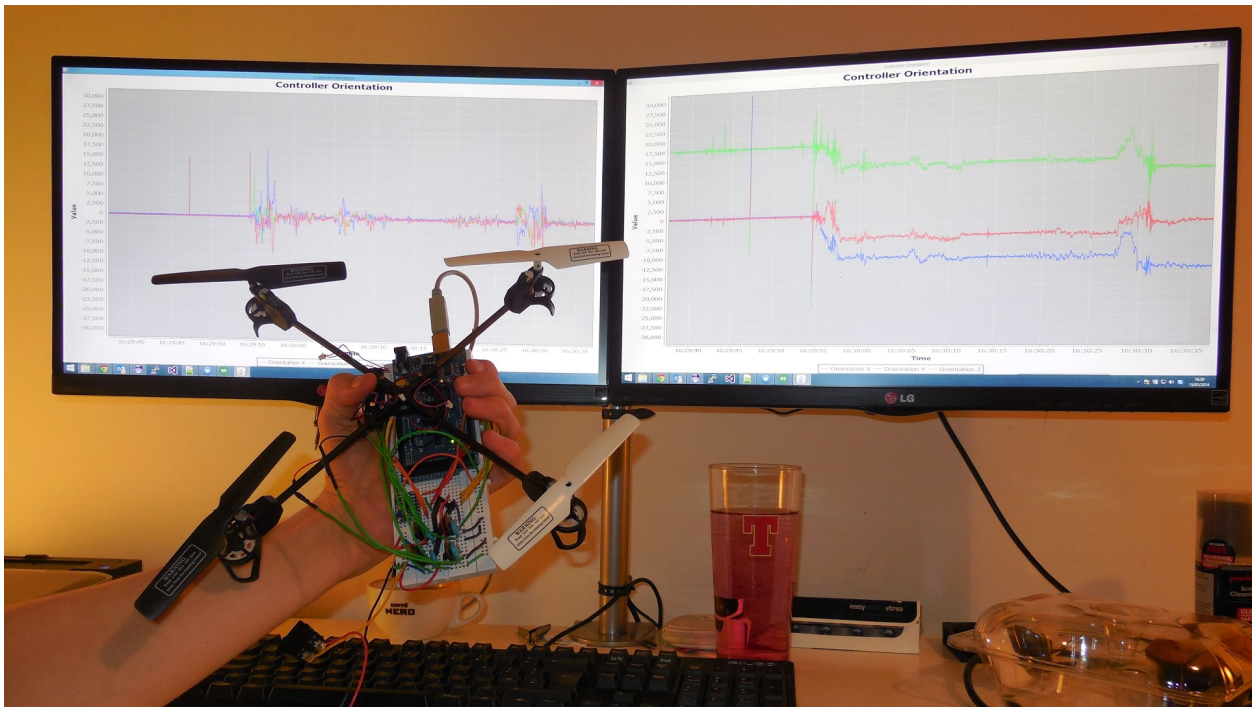
```
void serialEvent() {
  while (Serial.available()) {

    inbyte = Serial.read();
    if(inbyte >= '0' & inbyte <= '9') {
      serialDataIn += inbyte;
```

```
    }
    if (inbyte == ',' || inbyte == ';'){
      int value = serialDataIn.toInt();
      data[counter] = value;
      counter = counter + 1;
      serialDataIn = "";
    }
    if(inbyte ==  '\r' || inbyte ==  '\n'){
      serialDataIn = "";
      counter = 0;
    }
  }
}
```

Another paragraph, not related to motors

At this stage, we started writing some code in Eclipse to read in the data from the gyroscope, which was initially in quite a cryptic form. We handled the data and decided that textual representation wasn't a suitable way to display the data. Instead, we used some Java Graphing APIs and used a real-time line graph to visualise both the acceleration and gyration data.



After configuring the electronics and output of the gyroscope, the team moved onto the other main component(s) of the Quadcopter, the motors which power the rotors.

Controller Design, Trouble with Frameworks

In our idea/design discussions we decided to work with a Dualshock 6-Axis PS3 Bluetooth remote control. However this presented us with a series of difficulties. Namely the proprietary

PS3 Bluetooth protocol. It was just too much work for our relatively simple needs. The frameworks we could find were bloated/clunky or even questionable in regards to security. Having scrapped the PS3 remote idea we decided to try our hands with an Xbox 360 remote. Our success with the Xbox remote was similarly dismal and we quickly scrapped that idea.

Running out of options (and owned console controllers) we got a Nintendo Wiimote couriered to us and proceeded to explore the capabilities of it and the Nunchuk controller. Andrew found a relatively simple framework, **WiiuseJ** (https://code.google.com/p/wiiusej/), which nicely packages up all the functionality required for reading signals from Wiimotes. Using this framework (under the GNU LGPL) we simply had to create a Java class that implements the *WiimoteListener* interface. This one class allows us to capture what buttons are being pressed. Below is a screenshot of our code that indicate the different numerical values received from the Wiimote.

```
/*
 *
 *  1.  Thrust (B) = 4      Thrust A
 *  2.  Thrust (A) = 8      Thrust B
 *
 *  3.     Forward = 2048   Forward
 *  4.        Left = 256    Left
 *  5.       Right = 512    Right
 *  6.        Back = 1024   Back
 *
 *  7.     (1 + 3) = 2052   Thrust A + Forward
 *  8.     (1 + 4) = 260    Thrust A + Left
 *  9.     (1 + 5) = 516    Thrust A + Right
 * 10.     (1 + 6) = 1028   Thrust A + Back
 *
 * 11. (1 + 3) + 4 = 2308   Thrust A + Forward + Left
 * 12. (1 + 3) + 5 = 2564   Thrust A + Forward + Right
 *
 * 13. (1 + 6) + 4 = 1284   Thrust A + Back + Left
 * 14. (1 + 6) + 5 = 1540   Thrust A + back + Right
 *
 * 15.     (1 + 2) = 12     Thrust A + Thrust B
 *
 */
 private static final short BUTTON_A       = 4;     // 000000000100
 private static final short BUTTON_B       = 8;     // 000000001000
 private static final short BUTTON_LEFT    = 256;   // 000100000000
 private static final short BUTTON_RIGHT   = 512;   // 001000000000
 private static final short BUTTON_BACK    = 1024;  // 010000000000
 private static final short BUTTON_FORWARD = 2048;  // 100000000000
```

With these values we were originally executing a large switch/case statement that affected the model based on the values pressed. For example: if button A and Left were pressed the value would be 260, which would then update our control models thrust and bank-left values. However Tom (Smeedaviation CEO) noticed that these were all powers of 2 and suggested we use logical AND operations to decide which buttons were being pressed and in turn how to affect the model (see below).

```
boolean buttonA       = (BUTTON_A       & pressed) == BUTTON_A;
```

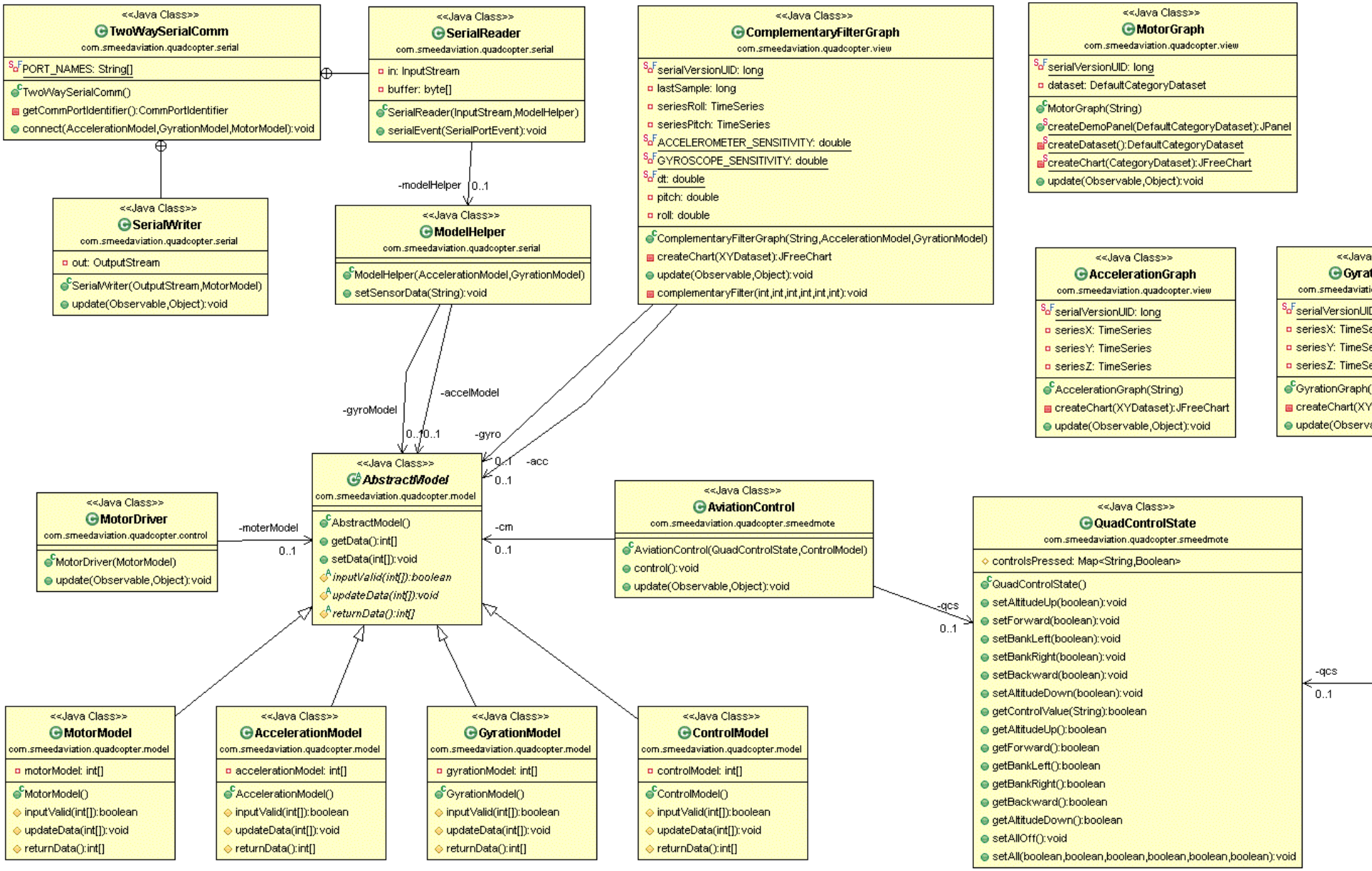
```
boolean buttonB       = (BUTTON_B       & pressed) == BUTTON_B;
boolean buttonLeft    = (BUTTON_LEFT    & pressed) == BUTTON_LEFT;
boolean buttonRight   = (BUTTON_RIGHT   & pressed) == BUTTON_RIGHT;
boolean buttonBack    = (BUTTON_BACK    & pressed) == BUTTON_BACK;
boolean buttonForward = (BUTTON_FORWARD & pressed) == BUTTON_FORWARD;
```

This ‘setAll’ method lets the model know what buttons have been pressed which then updates the Quadcopter stability and engine control classes using the Observer design pattern.

Dissection of issues and cause of issues

Software design

The java application is centered around 4 model objects, these hold the state of the user imput, acceleration, gyration and motor power. Accessing any of these model objects is synchronized to avoid race conditions and all model objects are observable.
Using the observer design pattern, the motor driver it notified when the user input has changed, trigger it update the motor values accordingly


<<Java Class>>
TwoWaySerialComm
com.smeedaviation.quadcopter.serial
PORT_NAMES: String[]
TwoWaySerialComm()
getCommPortIdentifier():CommPortIdentifier
connect(AccelerationModel,GyrationModel,MotorModel):void
<<Java Class>>
SerialReader
com.smeedaviation.quadcopter.serial
in: InputStream
buffer: byte[]
SerialReader(InputStream,ModelHelper)
serialEvent(SerialPortEvent):void
<<Java Class>>
ComplementaryFilterGraph
com.smeedaviation.quadcopter.view
serialVersionUID: long
lastSample: long
seriesRoll: TimeSeries
seriesPitch: TimeSeries
ACCELEROMETER_SENSITIVITY: double
GYROSCOPE_SENSITIVITY: double
dt: double
pitch: double
roll: double
ComplementaryFilterGraph(String,AccelerationModel,GyrationModel)
createChart(XYDataset):JFreeChart
update(Observable,Object):void
complementaryFilter(int,int,int,int,int,int):void
<<Java Class>>
MotorGraph
com.smeedaviation.quadcopter.view
serialVersionUID: long
dataset: DefaultCategoryDataset
MotorGraph(String)
createDemoPanel(DefaultCategoryDataset):JPanel
createDataset():DefaultCategoryDataset
createChart(CategoryDataset):JFreeChart
update(Observable,Object):void
<<Java Class>>
SerialWriter
com.smeedaviation.quadcopter.serial
out: OutputStream
SerialWriter(OutputStream,MotorModel)
update(Observable,Object):void
-modelHelper
0..1
<<Java Class>>
ModelHelper
com.smeedaviation.quadcopter.serial
ModelHelper(AccelerationModel,GyrationModel)
setSensorData(String):void
<<Java Class>>
AccelerationGraph
com.smeedaviation.quadcopter.view
serialVersionUID: long
seriesX: TimeSeries
seriesY: TimeSeries
seriesZ: TimeSeries
AccelerationGraph(String)
createChart(XYDataset):JFreeChart
update(Observable,Object):void
-gyroModel
-accelModel
0..1
0..1
-gyro
-acc
0..1
0..1
<<Java Class>>
AbstractModel
com.smeedaviation.quadcopter.model
AbstractModel()
getData():int[]
setData(int[]):void
inputValid(int[]):boolean
updateData(int[]):void
returnData():int[]
<<Java Class>>
MotorDriver
com.smeedaviation.quadcopter.control
MotorDriver(MotorModel)
update(Observable,Object):void
-moterModel
0..1
-cm
0..1
<<Java Class>>
AviationControl
com.smeedaviation.quadcopter.smeedmote
AviationControl(QuadControlState,ControlModel)
control():void
update(Observable,Object):void
-qcs
0..1
<<Java Class>>
QuadControlState
com.smeedaviation.quadcopter.smeedmote
controlsPressed: Map<String,Boolean>
QuadControlState()
setAltitudeUp(boolean):void
setForward(boolean):void
setBankLeft(boolean):void
setBankRight(boolean):void
setBackward(boolean):void
setAltitudeDown(boolean):void
getControlValue(String):boolean
getAltitudeUp():boolean
getForward():boolean
getBankLeft():boolean
getBankRight():boolean
getBackward():boolean
getAltitudeDown():boolean
setAllOff():void
setAll(boolean,boolean,boolean,boolean,boolean,boolean):void
-qcs
0..1
<<Java Class>>
MotorModel
com.smeedaviation.quadcopter.model
motorModel: int[]
MotorModel()
inputValid(int[]):boolean
updateData(int[]):void
returnData():int[]
<<Java Class>>
AccelerationModel
com.smeedaviation.quadcopter.model
accelerationModel: int[]
AccelerationModel()
inputValid(int[]):boolean
updateData(int[]):void
returnData():int[]
<<Java Class>>
GyrationModel
com.smeedaviation.quadcopter.model
gyrationModel: int[]
GyrationModel()
inputValid(int[]):boolean
updateData(int[]):void
returnData():int[]
<<Java Class>>
ControlModel
com.smeedaviation.quadcopter.model
controlModel: int[]
ControlModel()
inputValid(int[]):boolean
updateData(int[]):void
returnData():int[]