High Altitude Reusable Weather Balloon Based Autonomous UAV

By: Fraser Filice, Chuhan Qin, Vietca Vo

Supervisor: Dr. Steven S. Beauchemin

3380 Research Project

Submitted: April 30, 2014

Department of Computer Science

The University of Western Ontario

1151 Richmond Street, London, Ontario N6A 5B7.

Table of Contents

1. Introduction

Recent growth in 3D printing, personal UAV design, and microcontrollers, has led to these technologies becoming more affordable and consumer available. As a result the use of these technologies has led to a great deal of innovation in areas that were formerly restricted to government organizations and large corporations. 3D printing allows for the design and fabrication of complex custom parts quickly and cheaply. UAVs are becoming more and more intelligent, able to fly predetermined paths without any need for human feedback. They are capable of correcting for variations in weather conditions and other anomalies to reach a human defined destination. As a result there has been a recent push to develop smart UAVs for search and rescue applications as well as land and crop surveys. Both of these technologies have been made possible by smaller and more affordable microcontrollers.

The goal of this experiment is to incorporate these three technologies into a weather balloon to create a high altitude multipurpose platform suitable for use in various applications. Current weather balloons drift for several hundred kilometers and are barely salvageable after landing impact. It is important to combine a weather balloon's high altitude capabilities with artificial intelligence and a propulsion system to create a UAV capable of flying at such altitudes. This UAV design could be used for near space photography, astrophotography, and weather monitoring. High altitude UAVs could also be used for search and rescue or land survey, similar to their low altitude counterparts. Using a high altitude platform however would allow for sampling of much larger areas in a shorter period of time. Applications for high altitude UAVs could be extended into global communication systems. Due to the high altitudes achieved by weather balloons, line of sight could be achieved over large coverage areas. A network of solar powered high altitude UAVs could be used to create a network servicing remote areas without the need to launch expensive satellites.

Current FAA regulations limit the flight of unmanned vehicles and drones to 121.92 m AGL[1] (400 feet). This restriction is set in place to ensure that there is no interference with other aircraft. Weather balloons however are limited to a minimum altitude of 18,000 m (60,000 ft).[2] Weather balloons do not have a fixed ceiling as they fly well above air traffic. This allows for flight at altitudes only limited by the balloons and payload weight.

2. Experimental Protocols

The UAV is launched with various sensors and logging systems distributed over numerous microcontrollers. The most important sensors for the UAV are the compass and GPS. The compass returns the direction the UAV is facing, as well as its heading. The GPS returns the UAVs position. This data is logged and analysed to determine whether the selected propulsion system is effective in manipulating the UAV's position at altitude.

The UAV is designed to carry 3 Arduino microcontrollers (One for navigation and position logging, one for the autopilot system, one strictly for data logging with numerous instruments) and a Raspberry Pi computer to near space altitudes. These devices are connected to a number of sensors allowing for the real time monitoring and logging of data.

The Arduino selected for the purposes of data logging was an Arduino Mega. The Mega was selected as it provides 4x the RAM of an Arduino Uno. This is necessary as the number of sensors attached to this Arduino would exceed the capabilities of the less powerful UNO platform. This device had been paired with a GY-80 sensor package. The GY-80 includes a Barometric Pressure and Temperature Sensor (BMP-085), 3-axis accelerometer (ADXL345), 3-axis gyroscope (L3G4200D), and 3-axis digital compass (HMC5883L). All devices on the GY-80 sensor board communicate with the UNO using the I2C protocol. This protocol allows numerous sensors to be connected to the same two analog data pins on the UNO. The Mega was also outfitted with a second temperature sensor (DS18B20 Waterproof Temperature Sensor) which was positioned to obtain a battery (T3000.3S.20/9265) temperature from the heated batteries. (Figure 2) This sensor communicates using the OneWire protocol, which similarly allows numerous sensors to be connected using a single digital pin. All this incoming data was logged on a Lexar 8GB SD card using an Adafruit SD Logging Shield. The SD Logging Shield was designed for use with an Arduino UNO and required minor modification to work on an Arduino Mega. (Figure 1a)

The Arduinos Selected for the Autopilot System were Arduino UNOs. Arduino UNO microcontrollers have limited memory size, with 32KB of flash memory and 2KB of SRAM. Although it is desired to have navigation, position logging, and autopilot systems on the same board, it was not technically possible, resulting in having the systems on two different Arduinos,

working closely with each other using I2C protocol. This configuration was selected over a single Arduino Mega, as the Mega is not compatible with the GPS Shield. Navigation and position logging system uses an Arduino UNO with Adafruit Ultimate GPS Logger Shield, which integrates both a GPS chip (MTK3329) and MicroSD card logging module on the same shield. The GPS chip outputs standard GPGGA (Global Positioning System Fix Data) and GPRMC (Recommended minimum specific GPS/Transit data) sentences that belong to NMEA standard at the rate of 1Hz or 5 Hz. Besides retrieving and logging position, the board that has the GPS shield also calculates the distance between current position and destination using Haversine formula, and sends coordinates and altitude to the board that controls the autopilot. The autopilot will then correct its direction with a Grove 6-axis Accelerometer/Compass (SEN05072P) and move toward the desired coordinates.

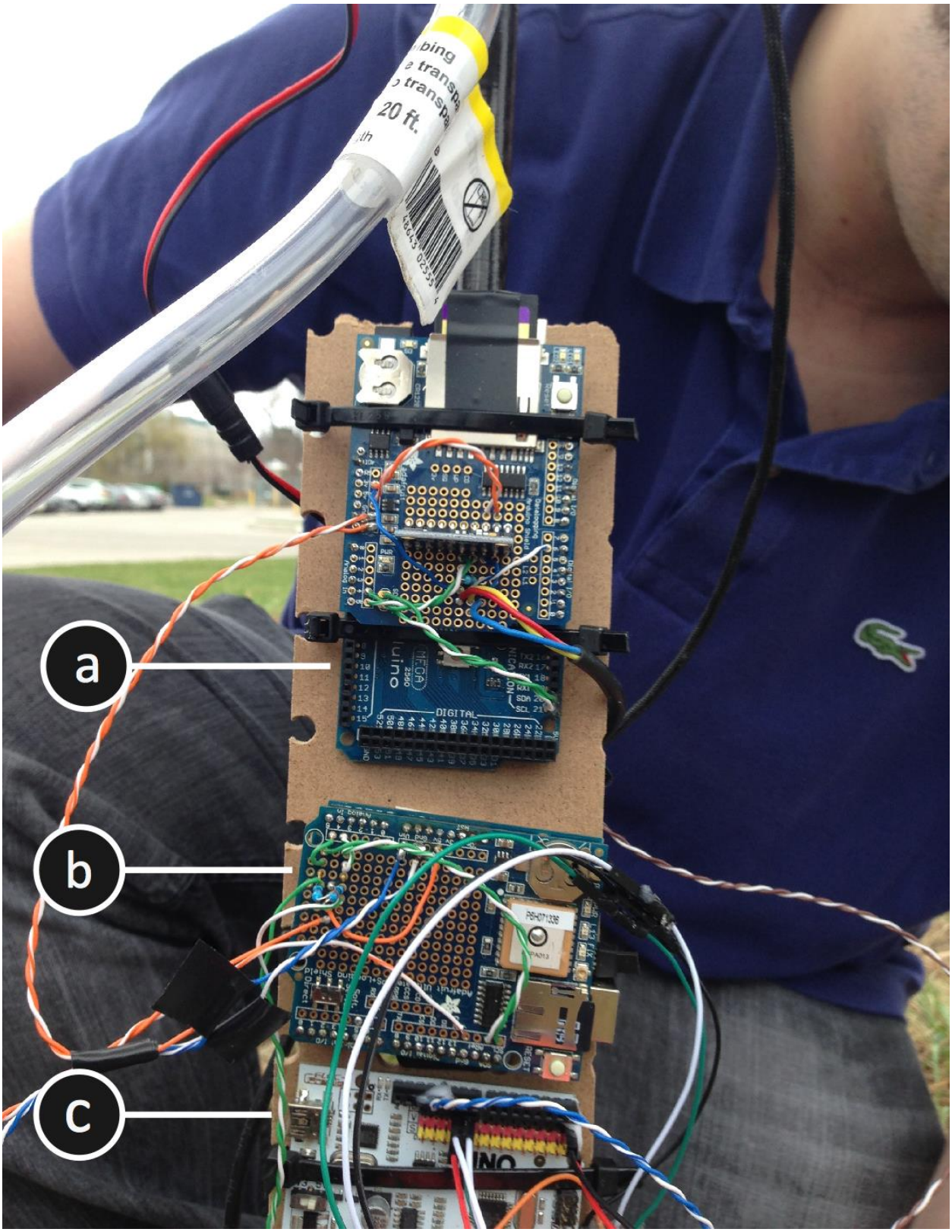

Figure 1 - (a) Arduino Mega data logging microcontroller. (b) Arduino Uno GPS navigation and position logging microcontroller. (c) Elecfreaks Arduino UNO autopilot microcontroller.

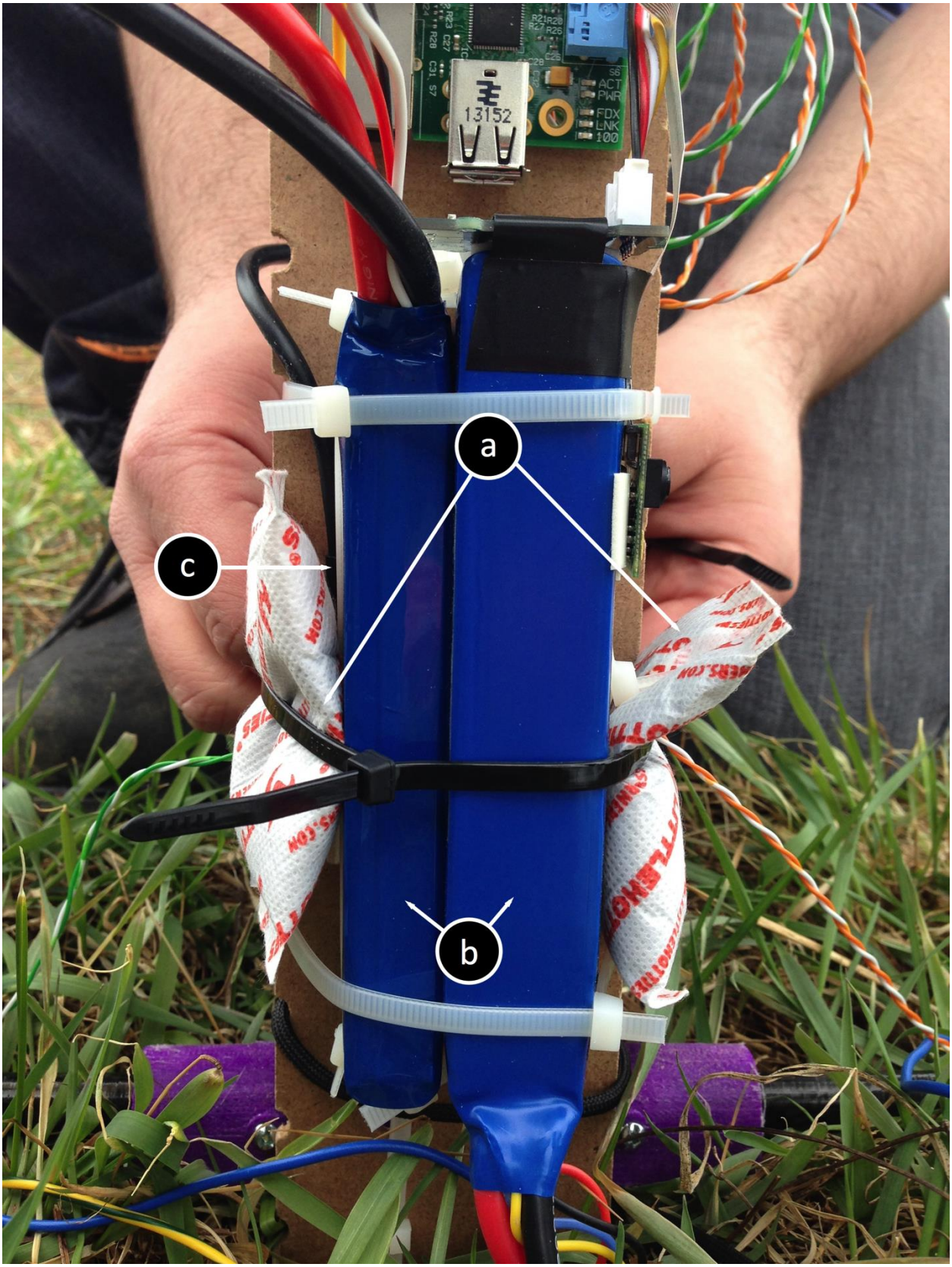

Figure 2 – (a) Chemical Hand Warmers. (b) Turnigy Batteries. (c) DS18B20 Waterproof Temperature Sensor

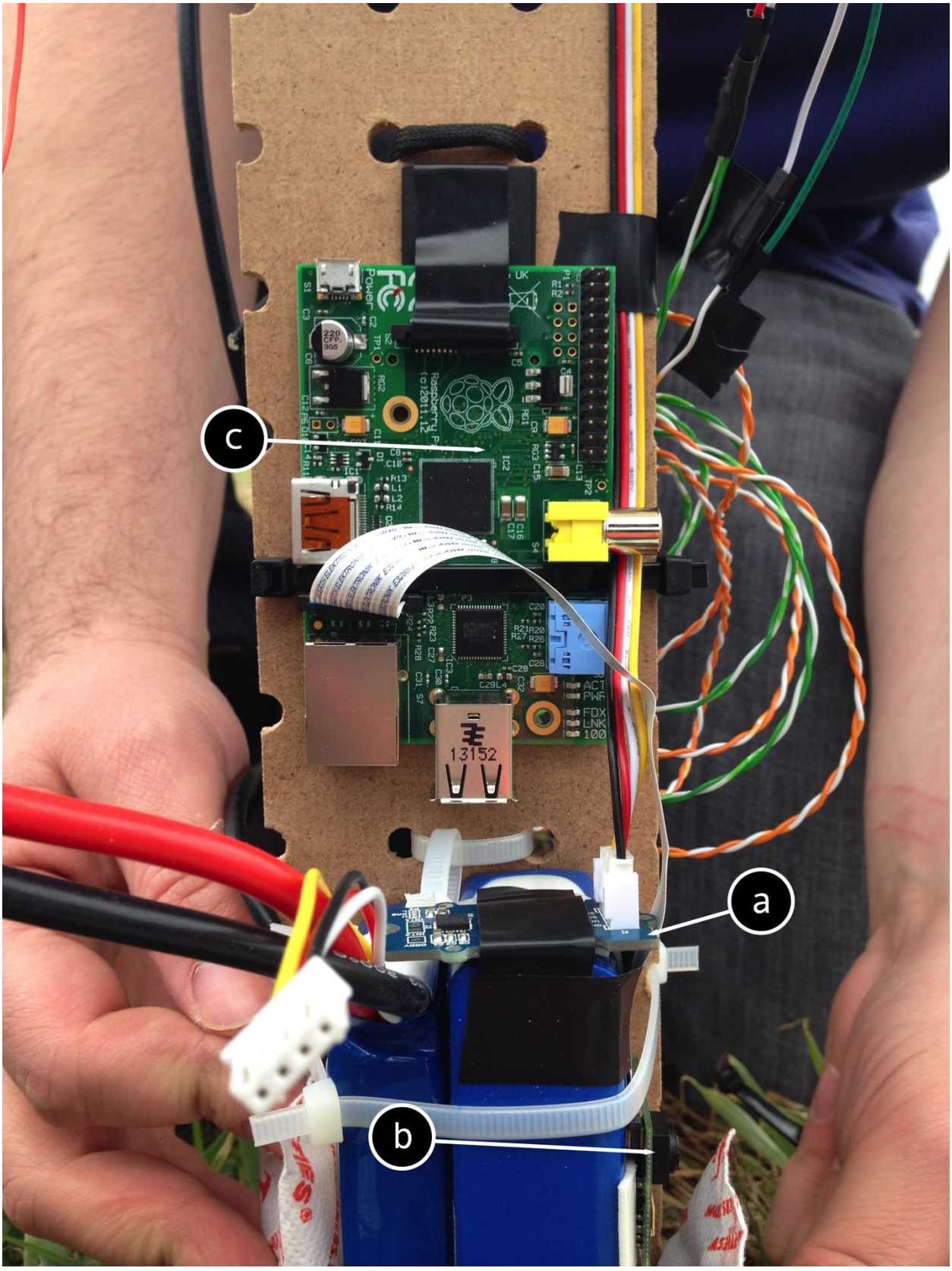

Figure 3 – (a) 3-Axis Digital Compass. (b) RPI Camera Board. (c) Raspberry Pi

The Raspberry Pi computer had been coupled with RPI Camera Boards. This camera board is an open source camera with open source drivers. This device was used for near space time-lapse photography, as well as to document the ascent and decent of the UAV.

The UAV uses a Meizu MX2 cellphone that runs an app called Where's My Droid on Android. The app detects keyword "WMD" in incoming text messages and takes certain actions according to the command. If a GPS location is requested, the phone will send four text messages, each contains request acknowledgement, coordinates with accuracy and bearing with speed, a link to Google Maps with current location, and approximate street. The app also has the ability to ring, take pictures and send them to the requestor, and lock the phone.

3. The Balloon

The frame of the UAV was built from Carbon fiber tubes (Rock West Composites) and custom 3D printed PLA plastic parts. This makes the frame extremely light and durable. The 3D parts were designed using 3D CAD software (Figure 4a) and exported to compatible *.stl file format. The parts were then printed on an Ultimaker 3D Printer (Figure 4b) owned and maintained by UnLab London Ontario. The 3D printed parts were then coated in a silicone based aquarium adhesive for additional strength. The parts were snugly fitted together and bolted to ensure a strong rigid frame. (Figure 5)

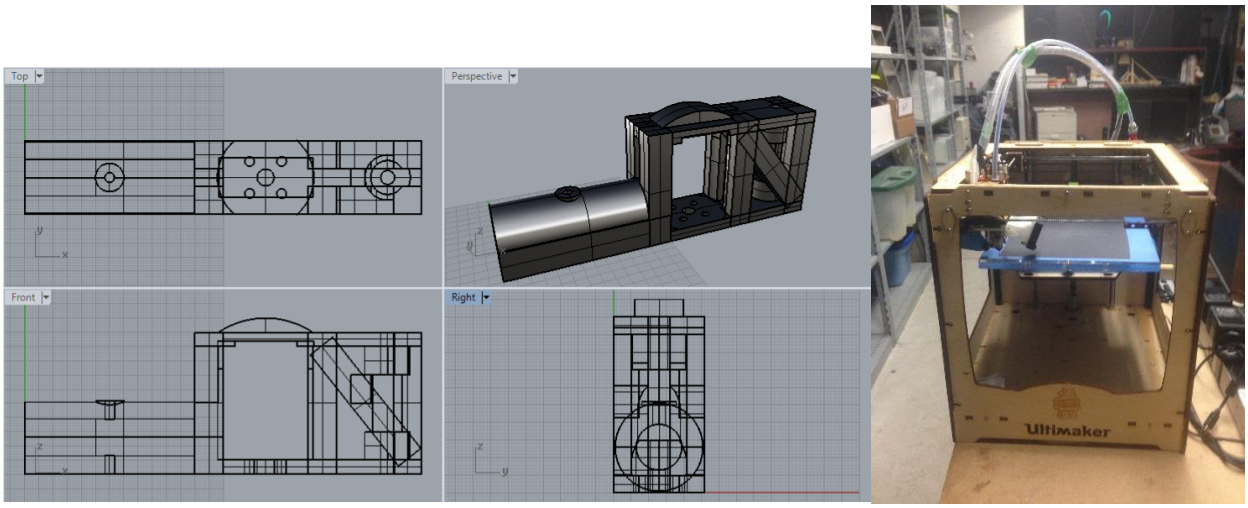

Figure 4 – (a) (left) Schematic and 3D perspective images of motor mount. (b) (right) Ultimaker 3D printer owned and maintained by UnLab London.

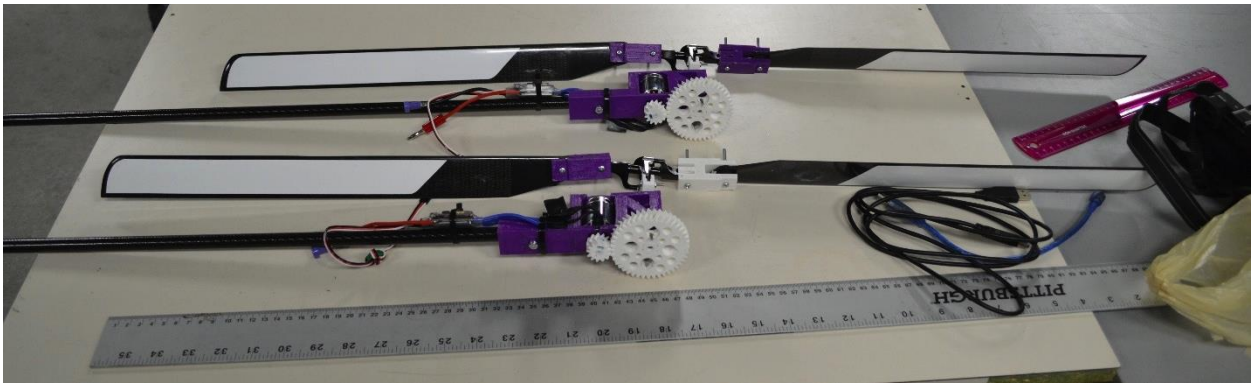

Figure 5 – Fully assembled blade assemblies for the UAV propulsion system mounted to the carbon fiber structural tubes.

The balloon's propulsion system is designed to be controlled by an Elecfreaks Freaduino UNO rev 1.8.1. The Freaduino was selected over the traditional Arduino due to its higher current output available on the 3.3V and 5.0V rails. The Solenoid controlled pressure release valve required an output current of 500mA to operate at 3.6V, which exceeds the current that can be provided by a normal Arduino. The Elecfreaks model is capable to providing 800mA on the 3.3V rail and 2A on the 5V rail more than providing the necessary current. This also allows one microcontroller to control a greater number of high current devices without being overloaded. This is an important feature as the same Arduino controlled the two servo ESC, monitored data from an electronic compass and communicated with the GPS and data logging Arduino via I2C.

Navigation positioning was obtained through the use of the Adafruit Ultimate GPS Logger Shield and Grove 6-axis Accelerometer/Compass.

The propulsion system for the UAV included consumer available model helicopter parts from Turnigy 450 Competition Series Helicopters, as well as 3D printed mounting brackets and gear systems. The UAV was designed to have two sets of blades at either end of the frame to allow for better manipulation of the UAV's heading with variations in blade rpm. To drive the blades, Turnigy HeliDrive SK3 Competition Series motors were coupled with Turnigy AE-45A Brushless ESCs (Electronic Servo Controllers). The ESCs were wired in parallel to a 3-cell 11.1V 5000mAh Turnigy Lithium Polymer battery to drive the motors. All other electronics received power from a second 11.1V 3000mAh battery. The ESCs were also wired individually to the Leonardo to obtain PWM speed control signals. Herringbone gears were 3D printed at a gear ratio of 43:13 to provide more torque to the large blade assembly and reduce the speed of the blades.

HK-450 PRO Flybarless DFC Rotor Heads were used with 425mm TIG Carbon Fiber Z-Weave Main Blades. A custom swashplate was 3D printed for the rotor head to fix the blade angle, as well as adapters for attaching the blades to the rotor heads. Steel bushings were fabricated and inserted into the motor mount to allow for less resistance on the rotor head drive shaft, and prevent melting of the PLA plastic at operating temperatures.

The balloon is an uncoloured 1200-gram sounding balloon with a nozzle lift of 2240 grams. Its recommended diameter at release is 179 cm and its bursting pressure is 7.3 hPa (approximately 33.2 kilometers above sea level) with a size of 863 cm.

A 180-gram parachute is attached to the bottom of the main structure, which has a rated descent rate of 3.8 – 4.0 metres per second at 1000-gram payload.

The UAV is designed to reach a defined altitude and level out at that altitude, never reaching its rated burst altitude. This is accomplished through the use of an electronically actuated air tight valve. (Figure 6a) Rather than being tied off, a combination PVC and ABS pipe was snugly inserted into the neck of the balloon and sealed with PL Premium Construction Adhesive. (Figure 6b) The pipe was capped, and two holes drilled in the cap to accommodate the electronic valve fitting and a mechanical higher volume fill valve. The valves were sealed using a silicone based adhesive. To interface with the Elecfreaks UNO, an H-Bridge (Texas Instruments L293D) was used to allow for polarity switching of 5.0V pulses to open and close the valve. The use of the 5V rail causes the motor to be briefly over-volted, as it is rated for use at 3.6V. However this only occurs for extremely brief periods. Testing done at 3.3V yielded inconsistent results when trying to open or close the valve due to limitations in the selected H-bridge. 5V however worked consistently with no noticeable heating of the solenoid or audible damage to the motor.

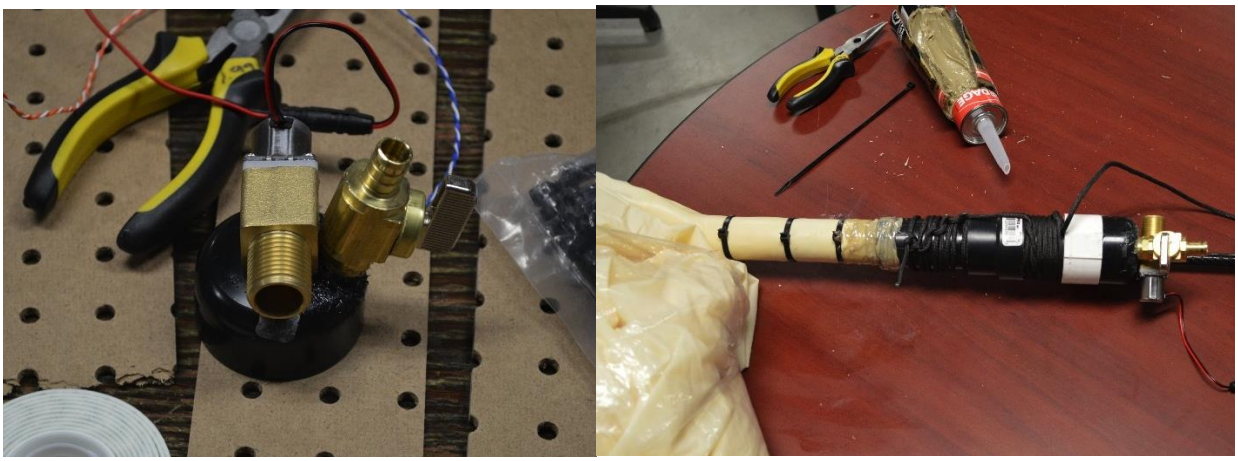

Figure 6 – (a) (left) ABS pipe cap with electronically actuated solenoid valve (center) and higher internal volume mechanical ball valve fill port (right) sealed with silicone based aquarium grade adhesive. (b) (right) Fully assembled ABS and PVC pipe, fitted and sealed into the neck of the balloon. Mounted to the UAV frame with PLA collar (white plastic), and secondary parachute cord safety harness (rated 200lbs).

4. The Flight

The launch date was selected based on predictions from the weather balloon predictor that analyzes data from Global Forecasting System of National Oceanic and Atmospheric Administration[4]. This prediction method uses current weather pattern predictions at various altitudes to determine the path a balloon will take given various other input parameters. When approaching completion of the weather balloon it was determined that a small launch window was available on April 22, 2014. The balloon was completed, assembled, filled with helium and prepared for launch. The balloon was then released at 4:00pm, and an estimate of the balloon path was taken at that time. (Figure 7) The wind during launch time was 20 kilometers per hour pushing east, and predicted to increase to 40 kilometers per hour easterly flow by 6:00pm.

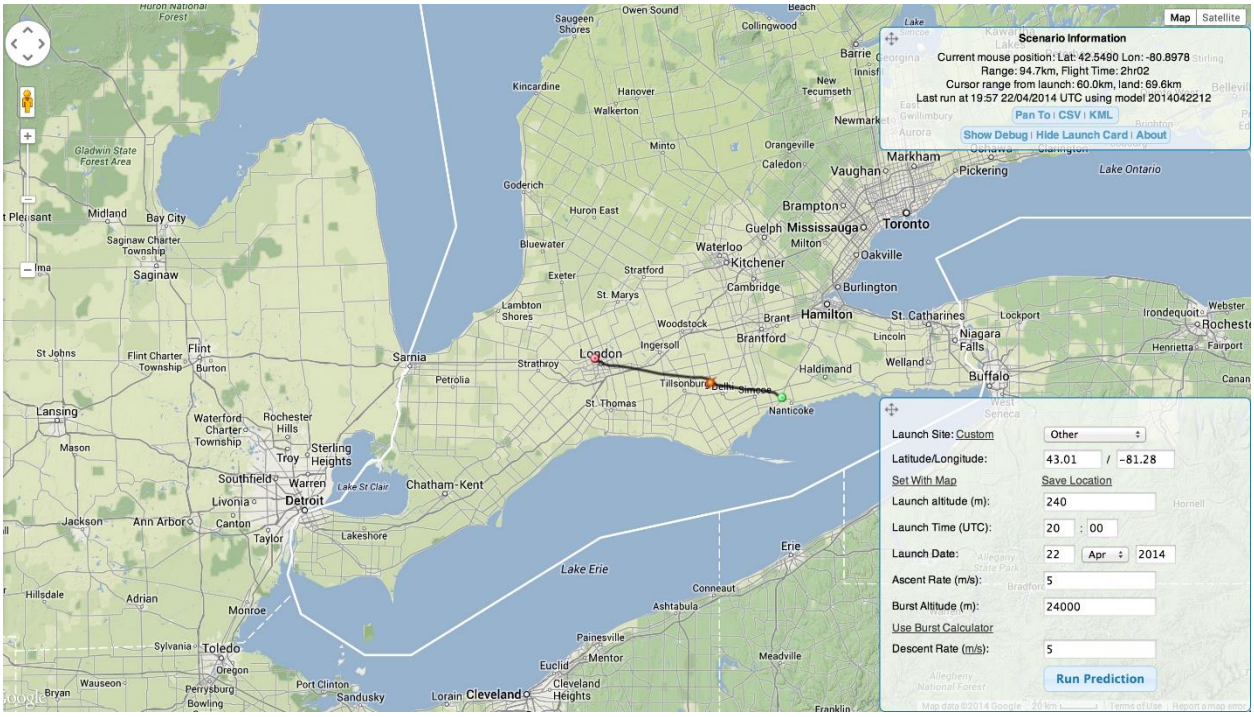

Figure 7 – Export of predicted weather balloon path at time of launch. Predictions provided by CUFS Landing Predictor[4].

One of the major concerns of this project had always been the weight limitations of balloons of this size. Upon filling the primary balloon it was discovered that while there was enough force generated by the helium to lift the payload, it would not be enough to achieve the desired altitude. The remaining Helium was used to fill a second balloon for a tandem design. (Figure 8) The second balloon was tied to the frame of the UAV using 200lb rated parachute cord. The second balloon was not filled to capacity, and as a result would not be capable of lifting the payload alone. In the event the primary balloon was not capable of venting gas due to a failure and burst altitude was reached, the primary balloon would burst and the UAV would begin a slow descent to land. This way the inability to control the volume of the second balloon does not limit the UAV's ability to adjust its altitude or land. These modifications alter the original design to make the second balloon the primary decent mechanism, and the parachute a secondary safety descent mechanism.

Figure 8 – (a) (left) Launch of the High altitude UAV in UWO Research Park. (b) (right) High Altitude UAV in flight. UAV still below safety threshold, propellers not in operation.

During flight, calculations are made to adjust the UAV's orientation and flight path.

$$\Delta\varphi = \ln\left(\tan\left(lat_B / 2 + \pi / 4 \right) / \tan\left(lat_A / 2 + \pi / 4 \right) \right)$$
$$\Delta lon = \text{abs}\left(lon_A - lon_B \right)$$
$$\text{bearing}: \ \theta = \text{atan2}\left(\Delta lon , \ \Delta\varphi \right)$$

Note: 1) ln = natural log 2) if $\Delta lon > 180°$ then $\Delta lon = \Delta lon \ (\text{mod } 180)$.

This is the formula used to calculate the desired bearing.[5] Longitude and latitude B is the set default landing location, while longitude and latitude A is the current position of the balloon itself. The calculated desired bearing is compared to the current bearing given by the 3-axis compass.

Figure 9 – The UAV is designed to correct its heading when faces with and offset greater than 20 degrees.

Both UAV blade motors will run as long as the current bearing is within a 20 degree offset of the desired bearing. For example, if the desired bearing is 230 degrees, both UAV helicopter motors will continue to run as long as the current bearing of the UAV is between 210 to 250 degrees.

If the current bearing is less than the sum of the desired bearing and its offset, then only the left motor runs to turn the UAV right. Likewise, if the current bearing is more than the sum of the desired bearing and its offset, then only the right motor will run to turn the UAV left.

In summary:

If (desiredBearing - offset ≤ currentBearing ≤ desiredBearing + offset)

then power motor1 and moter2

Else If (currentBearing < desiredBearing - offet)

then power motor1

else if (currentBearing > desiredBearing + offset)

then power motor2

5. Experimental Results

On launch day, the UAV was prepared for launch and tested. All electronic components powered on and reported as designed. The cell phone tracking system responded to test messages reporting the launch coordinates. When the UAV exceeded the range of the cell phone network, it failed to continue reporting its position. This occurred before line of sight with the device was lost. One additional message was obtained as the UAV was leaving London Ontario. (Figure 10) The UAV was travelling at 30.0MPH, bearing east.

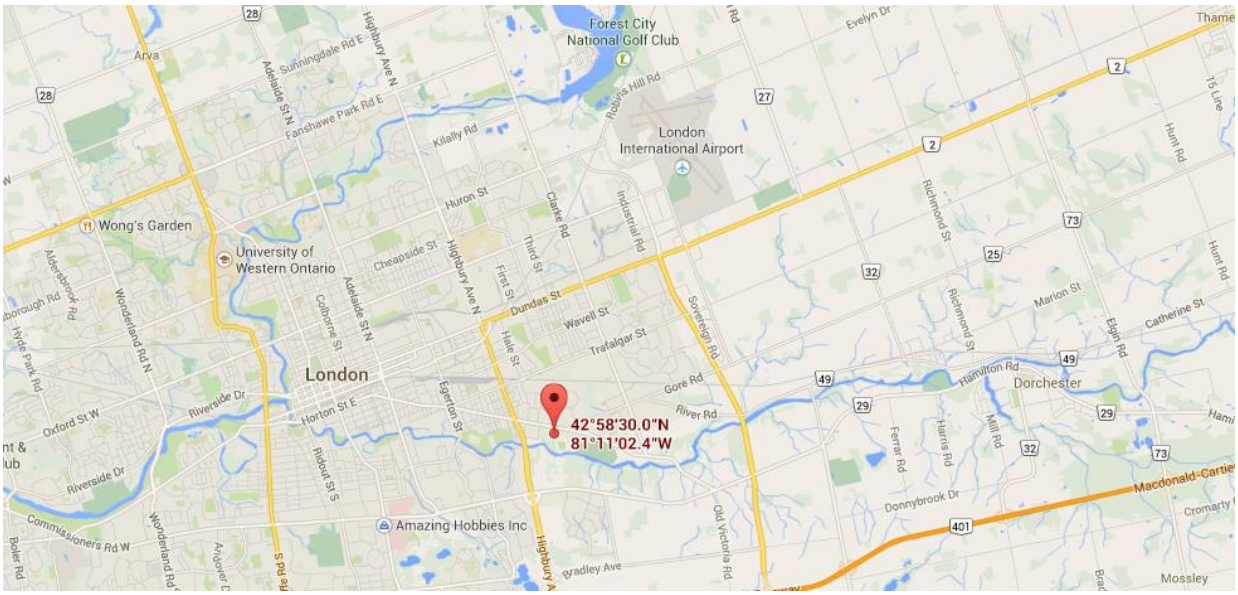

Figure 10 – Coordinates of the UAV as it left London Ontario heading east at 30.0MPH.

Numerous requests for the UAV location were queued and the service was monitored for 8 days. No further messages were obtained from the UAV. At this time the cell phone battery would certainly have died, and the phone would have ceased operation. The UAV was labelled with contact information in the event it is found by someone, however no calls have been received yet.

It is possible the cell phone shut down at altitude due to extreme heat or cold and failed to report as a result. It is also possible that the balloon landed in a location where cell phone service is limited or not available. A third possibility is that the UAV leveled out at cruising altitude, and due to a failure shut down before it was set to land. This would cause the valve to be sealed in the closed position. As a result, the UAV would not land until the helium slowly bled away from both balloons. In this third event it is not probable the UAV will ever be recovered, as the flight time could place it anywhere in the northern hemisphere.

6. Conclusion

Several features of the current design can be improved, including platform structure, two-way communication, electronic component design, and energy sustainability.

Use of this design is severely limited by the weight of the payload being carried and the structure itself. For practical future use, the size of the balloon needs to be scaled up, or a

multi-balloon platform would need to be created. Moreover, materials used in the structure should be as lightweight as possible. Sufficient improvement on this feature will allow the use of larger cameras or telescopes for near space photography or astrophotography.

Reliable communication with the UAV is of crucial importance. The current design uses a cellphone and the reliability is severely limited by the quality of signal reception. In order to achieve reliable and more integrated way of data transmission, using two-way satellite communication modules will be highly recommended, and communication systems should be redundant.

Arduino microcontrollers are used for prototyping, thus they are not optimized for highly sophisticated UAVs. Using smaller form factor Arduino Nano would reduce the weight and size of the microcontroller while maintaining all the same pins and processing power of the full sized UNO. Another option would be to use purposed-designed hardware substantially decreasing unnecessary weight being carried and power consumed.

The current design deploys two batteries that are originally designed for competition-grade remote control helicopters. Although they are two of the most suitable batteries on consumer market, they could only support the propellers to run at full speed for less than an hour. To achieve long-term stratospheric flights, solar panels or other sources of energy are undoubtedly necessary.

7. References

1. Federal Aviation Administration. Unmanned Aircraft Operations in the National Airspace System. http://www.faa.gov/about/initiatives/uas/reg/media/frnotice_uas.pdf (accessed Apr 14, 2014).
2. Nav Canada. Aeronautical Information Services Canadian NOTAM Procedures Manual. Section 5.5.13. http://www.navcanada.ca/EN/media/Publications/NOTAM-Manual-EN.pdf (accessed Apr 14, 2014).
3. Conover, W. C., C. J. Wentzien, 1955: WINDS AND TEMPERATURES TO FORTY KILOMETERS. *J. Meteor.*, **12**, 160–164. doi: http://dx.doi.org/10.1175/1520-0469(1955)012<0160:WATTFK>2.0.CO;2

4. UK High Altitude Society. CUFS Landing Predictor 2.5. http://predict.habhub.org/ (accessed April 22, 2014)
5. SunEarthTools.com. Distance. http://www.sunearthtools.com/tools/distance.php#txtDist_3 (accessed Apr 14, 2014).

Appendix

Arduino Mega Data Logger

```
//Arduino 1.0+ Only
//Arduino 1.0+ Only

#include "SD.h"
//#include "RTClib.h"
#include <Wire.h>
#include <ADXL345.h>
#include <HMC5883L.h>
#include <OneWire.h>

#define BMP085_ADDRESS 0x77  // I2C address of BMP085

// for the data logging shield, we use digital pin 10 for the SD cs line
const int chipSelect = 10;
const int chipSelect2 = 53;
// the logging file
File logfile;

//for L3G4200D
#define CTRL_REG1 0x20
#define CTRL_REG2 0x21
#define CTRL_REG3 0x22
#define CTRL_REG4 0x23
#define CTRL_REG5 0x24

//for BMP085
const unsigned char OSS = 0;  // Oversampling Setting
```

```
// Calibration values for BMP085
int ac1;
int ac2;
int ac3;
unsigned int ac4;
unsigned int ac5;
unsigned int ac6;
int b1;
int b2;
int mb;
int mc;
int md;

// b5 is calculated in bmp085GetTemperature(...), this variable is also used in bmp085GetPressure(...)
// so ...Temperature(...) must be called before ...Pressure(...).
long b5;

//for ADXL345
ADXL345 adxl; //variable adxl is an instance of the ADXL345 library

//for L3G4200D
int L3G4200D_Address = 105; //I2C address of the L3G4200D

int x;
int y;
int z;

//for hmc5883L
HMC5883L compass;
```

```
//for blinking pin 13 led
boolean ledon = true;

//for measuring time since the start of operation
uint32_t timeSinceStart; //good for 49.7 days
//uint64_t timeSinceStart; //good for 584942417.4 years

// DS18S20 Temperature chip i/o
OneWire ds(6);  // on pin 3

void setup(){
  Serial.begin(19200);
  Wire.begin();

  //prep the SD Card
  sdInit();
  //start the real time clock
  PrintCurrentTime();
  Serial.print("freeRam: ");
  Serial.println(freeRam());

  bmp085Calibration();
  adxl345Calibration();

  Serial.println("starting up L3G4200D");
  logfile.println("starting up L3G4200D");
  setupL3G4200D(2000); // Configure L3G4200  - 250, 500 or 2000 deg/sec
  delay(1500); //wait for the sensor to be ready
```

```
  compass = HMC5883L(); //new instance of HMC5883L library
  setupHMC5883L(); //setup the HMC5883L

  Serial.println();
  logfile.println();
}

void loop()
{
  PrintCurrentTime();

  pressureSensor();
  accelerometer();
  gyro();
  hallSensor();
  TempSensor();

  Serial.print("freeRam: ");
  Serial.println(freeRam());
  logfile.print("freeRam: ");
  logfile.println(freeRam());
  Serial.println();
  logfile.println();

  logfile.flush();
}

/**
Method to initialize the SD card on the Adafruit SD Logger Shield
*/
```

```
void sdInit()
{
   // initialize the SD card
   Serial.print("Initializing SD card...");
   // make sure that the default chip select pin is set to
   // output, even if you don't use it:
   pinMode(10, OUTPUT);
   pinMode(53, OUTPUT);
   // see if the card is present and can be initialized:
   if (!SD.begin(chipSelect)) {
     Serial.println("Card failed, or not present");
     // don't do anything more:
     return;
   }
   Serial.println("card initialized.");
   // create a new file
   char filename[] = "LOGGER00.TXT";
   for (uint8_t i = 0; i < 100; i++) {
     filename[6] = i/10 + '0';
     filename[7] = i%10 + '0';
   // create a new file
     if (! SD.exists(filename)) {
       // only open a new file if it doesn't exist
       Serial.print("filename: ");
       Serial.println(filename);
       logfile = SD.open(filename, FILE_WRITE);

       break; // leave the loop!
     }
   }
```

```
   if (! logfile) {
     error("couldnt create file");
   }
}

/**
Method to get the current time and print it
*/
void PrintCurrentTime()
{
    // log milliseconds since starting
   uint32_t timeSinceStart = millis();
   logfile.print("Millis since start: ");
   logfile.println(timeSinceStart); // milliseconds since start
   Serial.print("Millis since start: ");
   Serial.println(timeSinceStart); // milliseconds since start
}

/*Based largely on code by  Jim Lindblom

 Get pressure, altitude, and temperature from the BMP085.
 Serial.print it out at 9600 baud to serial monitor.
 */
void pressureSensor()
{
  float temperature = bmp085GetTemperature(bmp085ReadUT()); //MUST be called first
  float pressure = bmp085GetPressure(bmp085ReadUP());
  float atm = pressure / 101325; // "standard atmosphere"
  float altitude = calcAltitude(pressure); //Uncompensated caculation - in Meters
```

```
  Serial.print("Temperature: ");
  logfile.print("Temperature: ");
  Serial.print(temperature, 2); //display 2 decimal places
  logfile.print(temperature, 2); //display 2 decimal places
  Serial.println("deg C");
  logfile.println("deg C");

  Serial.print("Pressure: ");
  Serial.print(pressure, 0); //whole number only.
  Serial.println(" Pa");
  logfile.print("Pressure: ");
  logfile.print(pressure, 0); //whole number only.
  logfile.println(" Pa");

  Serial.print("Standard Atmosphere: ");
  Serial.println(atm, 4); //display 4 decimal places
  logfile.print("Standard Atmosphere: ");
  logfile.println(atm, 4); //display 4 decimal places

  Serial.print("Altitude: ");
  Serial.print(altitude, 2); //display 2 decimal places
  Serial.println(" m");
  logfile.print("Altitude: ");
  logfile.print(altitude, 2); //display 2 decimal places
  logfile.println(" m");

  delay(1000); //wait a second and get values again.
}

// Stores all of the bmp085's calibration values into global variables
```

```
// Calibration values are required to calculate temp and pressure
// This function should be called at the beginning of the program
void bmp085Calibration()
{
  ac1 = bmp085ReadInt(0xAA);
  ac2 = bmp085ReadInt(0xAC);
  ac3 = bmp085ReadInt(0xAE);
  ac4 = bmp085ReadInt(0xB0);
  ac5 = bmp085ReadInt(0xB2);
  ac6 = bmp085ReadInt(0xB4);
  b1 = bmp085ReadInt(0xB6);
  b2 = bmp085ReadInt(0xB8);
  mb = bmp085ReadInt(0xBA);
  mc = bmp085ReadInt(0xBC);
  md = bmp085ReadInt(0xBE);
}

// Calculate temperature in deg C
float bmp085GetTemperature(unsigned int ut){
  long x1, x2;

  x1 = (((long)ut - (long)ac6)*(long)ac5) >> 15;
  x2 = ((long)mc << 11)/(x1 + md);
  b5 = x1 + x2;

  float temp = ((b5 + 8)>>4);
  temp = temp /10;

  return temp;
}
```

```
// Calculate pressure given up
// calibration values must be known
// b5 is also required so bmp085GetTemperature(...) must be called first.
// Value returned will be pressure in units of Pa.
long bmp085GetPressure(unsigned long up){
  long x1, x2, x3, b3, b6, p;
  unsigned long b4, b7;

  b6 = b5 - 4000;
  // Calculate B3
  x1 = (b2 * (b6 * b6)>>12)>>11;
  x2 = (ac2 * b6)>>11;
  x3 = x1 + x2;
  b3 = (((((long)ac1)*4 + x3)<<OSS) + 2)>>2;

  // Calculate B4
  x1 = (ac3 * b6)>>13;
  x2 = (b1 * ((b6 * b6)>>12))>>16;
  x3 = ((x1 + x2) + 2)>>2;
  b4 = (ac4 * (unsigned long)(x3 + 32768))>>15;

  b7 = ((unsigned long)(up - b3) * (50000>>OSS));
  if (b7 < 0x80000000)
    p = (b7<<1)/b4;
  else
    p = (b7/b4)<<1;

  x1 = (p>>8) * (p>>8);
  x1 = (x1 * 3038)>>16;
```

```
  x2 = (-7357 * p)>>16;
  p += (x1 + x2 + 3791)>>4;

  long temp = p;
  return temp;
}

// Read 1 byte from the BMP085 at 'address'
char bmp085Read(unsigned char address)
{
  unsigned char data;

  Wire.beginTransmission(BMP085_ADDRESS);
  Wire.write(address);
  Wire.endTransmission();

  Wire.requestFrom(BMP085_ADDRESS, 1);
  while(!Wire.available())
    ;

  return Wire.read();
}

// Read 2 bytes from the BMP085
// First byte will be from 'address'
// Second byte will be from 'address'+1
int bmp085ReadInt(unsigned char address)
{
  unsigned char msb, lsb;
```

```
  Wire.beginTransmission(BMP085_ADDRESS);
  Wire.write(address);
  Wire.endTransmission();

  Wire.requestFrom(BMP085_ADDRESS, 2);
  while(Wire.available()<2)
    ;
  msb = Wire.read();
  lsb = Wire.read();

  return (int) msb<<8 | lsb;
}

// Read the uncompensated temperature value
unsigned int bmp085ReadUT(){
  unsigned int ut;

  // Write 0x2E into Register 0xF4
  // This requests a temperature reading
  Wire.beginTransmission(BMP085_ADDRESS);
  Wire.write(0xF4);
  Wire.write(0x2E);
  Wire.endTransmission();

  // Wait at least 4.5ms
  delay(5);

  // Read two bytes from registers 0xF6 and 0xF7
  ut = bmp085ReadInt(0xF6);
  return ut;
```

```
}

// Read the uncompensated pressure value
unsigned long bmp085ReadUP(){

  unsigned char msb, lsb, xlsb;
  unsigned long up = 0;

  // Write 0x34+(OSS<<6) into register 0xF4
  // Request a pressure reading w/ oversampling setting
  Wire.beginTransmission(BMP085_ADDRESS);
  Wire.write(0xF4);
  Wire.write(0x34 + (OSS<<6));
  Wire.endTransmission();

  // Wait for conversion, delay time dependent on OSS
  delay(2 + (3<<OSS));

  // Read register 0xF6 (MSB), 0xF7 (LSB), and 0xF8 (XLSB)
  msb = bmp085Read(0xF6);
  lsb = bmp085Read(0xF7);
  xlsb = bmp085Read(0xF8);

  up = (((unsigned long) msb << 16) | ((unsigned long) lsb << 8) | (unsigned long) xlsb) >> (8-OSS);

  return up;
}

float calcAltitude(float pressure){
```

```
  float A = pressure/101325;
  float B = 1/5.25588;
  float C = pow(A,B);
  C = 1 - C;
  C = C /0.0000225577;

  return C;
}

void adxl345Calibration()
{
  adxl.powerOn();

  //set activity/ inactivity thresholds (0-255)
  adxl.setActivityThreshold(75); //62.5mg per increment
  adxl.setInactivityThreshold(75); //62.5mg per increment
  adxl.setTimeInactivity(10); // how many seconds of no activity is inactive?

  //look of activity movement on this axes - 1 == on; 0 == off
  adxl.setActivityX(1);
  adxl.setActivityY(1);
  adxl.setActivityZ(1);

  //look of inactivity movement on this axes - 1 == on; 0 == off
  adxl.setInactivityX(1);
  adxl.setInactivityY(1);
  adxl.setInactivityZ(1);

  //look of tap movement on this axes - 1 == on; 0 == off
  adxl.setTapDetectionOnX(0);
```

```
 adxl.setTapDetectionOnY(0);
 adxl.setTapDetectionOnZ(1);

 //set values for what is a tap, and what is a double tap (0-255)
 adxl.setTapThreshold(50); //62.5mg per increment
 adxl.setTapDuration(15); //625µs per increment
 adxl.setDoubleTapLatency(80); //1.25ms per increment
 adxl.setDoubleTapWindow(200); //1.25ms per increment

 //set values for what is considered freefall (0-255)
 adxl.setFreeFallThreshold(7); //(5 - 9) recommended - 62.5mg per increment
 adxl.setFreeFallDuration(45); //(20 - 70) recommended - 5ms per increment

 //setting all interupts to take place on int pin 1
 //I had issues with int pin 2, was unable to reset it
 adxl.setInterruptMapping( ADXL345_INT_SINGLE_TAP_BIT,   ADXL345_INT1_PIN );
 adxl.setInterruptMapping( ADXL345_INT_DOUBLE_TAP_BIT,   ADXL345_INT1_PIN );
 adxl.setInterruptMapping( ADXL345_INT_FREE_FALL_BIT,    ADXL345_INT1_PIN );
 adxl.setInterruptMapping( ADXL345_INT_ACTIVITY_BIT,     ADXL345_INT1_PIN );
 adxl.setInterruptMapping( ADXL345_INT_INACTIVITY_BIT,   ADXL345_INT1_PIN );

 //register interupt actions - 1 == on; 0 == off
 adxl.setInterrupt( ADXL345_INT_SINGLE_TAP_BIT, 1);
 adxl.setInterrupt( ADXL345_INT_DOUBLE_TAP_BIT, 1);
 adxl.setInterrupt( ADXL345_INT_FREE_FALL_BIT,  1);
 adxl.setInterrupt( ADXL345_INT_ACTIVITY_BIT,   1);
 adxl.setInterrupt( ADXL345_INT_INACTIVITY_BIT, 1);
}

void accelerometer()
```

```
{
  //Boring accelerometer stuff
  int x,y,z;
  adxl.readAccel(&x, &y, &z); //read the accelerometer values and store them in variables  x,y,z

  // Output x,y,z values - Commented out
  Serial.print("acc: ");
  Serial.print("x=");
  Serial.print(x);
  Serial.print("y=");
  Serial.print(y);
  Serial.print("z=");
  Serial.println(z);
  logfile.print("acc: ");
  logfile.print("x=");
  logfile.print(x);
  logfile.print("y=");
  logfile.print(y);
  logfile.print("z=");
  logfile.println(z);
  delay(1000);


  //Fun Stuff!
  //read interrupts source and look for triggerd actions

  //getInterruptSource clears all triggered actions after returning value
  //so do not call again until you need to recheck for triggered actions
   byte interrupts = adxl.getInterruptSource();
```

```
// freefall
if(adxl.triggered(interrupts, ADXL345_FREE_FALL)){
  Serial.println("freefall");
  logfile.println("freefall");
  //add code here to do when freefall is sensed
}

//inactivity
if(adxl.triggered(interrupts, ADXL345_INACTIVITY)){
  Serial.println("inactivity");
  logfile.println("inactivity");
   //add code here to do when inactivity is sensed
}

//activity
if(adxl.triggered(interrupts, ADXL345_ACTIVITY)){
  Serial.println("activity");
 logfile.println("activity");
   //add code here to do when activity is sensed
}

//double tap
if(adxl.triggered(interrupts, ADXL345_DOUBLE_TAP)){
  Serial.println("double tap");
  logfile.println("double tap");
   //add code here to do when a 2X tap is sensed
}

//tap
if(adxl.triggered(interrupts, ADXL345_SINGLE_TAP)){
```

```
    Serial.println("tap");
    logfile.println("tap");
     //add code here to do when a tap is sensed
  }
}

void gyro()
{
  getGyroValues();  // This will update x, y, and z with new values

  Serial.print("gyro: ");
  logfile.print("gyro: ");

  Serial.print("X:");
  Serial.print(x);
  logfile.print("X:");
  logfile.print(x);

  Serial.print(" Y:");
  Serial.print(y);
  logfile.print(" Y:");
  logfile.print(y);

  Serial.print(" Z:");
  Serial.println(z);
  logfile.print(" Z:");
  logfile.println(z);

  delay(100); //Just here to slow down the serial to make it more readable
}
```

```
void getGyroValues(){

  byte xMSB = readRegister(L3G4200D_Address, 0x29);
  byte xLSB = readRegister(L3G4200D_Address, 0x28);
  x = ((xMSB << 8) | xLSB);

  byte yMSB = readRegister(L3G4200D_Address, 0x2B);
  byte yLSB = readRegister(L3G4200D_Address, 0x2A);
  y = ((yMSB << 8) | yLSB);

  byte zMSB = readRegister(L3G4200D_Address, 0x2D);
  byte zLSB = readRegister(L3G4200D_Address, 0x2C);
  z = ((zMSB << 8) | zLSB);
}

int setupL3G4200D(int scale){
  //From  Jim Lindblom of Sparkfun's code

  // Enable x, y, z and turn off power down:
  writeRegister(L3G4200D_Address, CTRL_REG1, 0b00001111);

  // If you'd like to adjust/use the HPF, you can edit the line below to configure CTRL_REG2:
  writeRegister(L3G4200D_Address, CTRL_REG2, 0b00000000);

  // Configure CTRL_REG3 to generate data ready interrupt on INT2
  // No interrupts used on INT1, if you'd like to configure INT1
  // or INT2 otherwise, consult the datasheet:
  writeRegister(L3G4200D_Address, CTRL_REG3, 0b00001000);
```

```
  // CTRL_REG4 controls the full-scale range, among other things:

  if(scale == 250){
    writeRegister(L3G4200D_Address, CTRL_REG4, 0b00000000);
  }else if(scale == 500){
    writeRegister(L3G4200D_Address, CTRL_REG4, 0b00010000);
  }else{
    writeRegister(L3G4200D_Address, CTRL_REG4, 0b00110000);
  }

  // CTRL_REG5 controls high-pass filtering of outputs, use it
  // if you'd like:
  writeRegister(L3G4200D_Address, CTRL_REG5, 0b00000000);
}

void writeRegister(int deviceAddress, byte address, byte val) {
    Wire.beginTransmission(deviceAddress); // start transmission to device
    Wire.write(address);       // send register address
    Wire.write(val);        // send value to write
    Wire.endTransmission();     // end transmission
}

int readRegister(int deviceAddress, byte address){

    int v;
    Wire.beginTransmission(deviceAddress);
    Wire.write(address); // register to read
    Wire.endTransmission();

    Wire.requestFrom(deviceAddress, 1); // read a byte
```

```
  while(!Wire.available()) {
    // waiting
  }

  v = Wire.read();
  return v;
}

void hallSensor()
{
 float heading = getHeading();
 Serial.print("Heading: ");
 Serial.println(heading);
 logfile.print("Heading: ");
 logfile.println(heading);
 delay(100); //only here to slow down the serial print
}

void setupHMC5883L(){
 //Setup the HMC5883L, and check for errors
 int error;
 float scaleVal = 0.88;
 error = compass.SetScale(scaleVal); //Set the scale of the compass.
 if(error != 0){
  Serial.println(compass.GetErrorText(error)); //check if there is an error, and print if so
  logfile.println(compass.GetErrorText(error)); //check if there is an error, and print if so
 }
```

```
  error = compass.SetMeasurementMode(Measurement_Continuous); // Set the measurement mode to
Continuous
  if(error != 0){
    Serial.println(compass.GetErrorText(error)); //check if there is an error, and print if so
    logfile.println(compass.GetErrorText(error)); //check if there is an error, and print if so
  }
}

float getHeading(){
  //Get the reading from the HMC5883L and calculate the heading
  MagnetometerScaled scaled = compass.ReadScaledAxis(); //scaled values from compass.
  float heading = atan2(scaled.YAxis, scaled.XAxis);

  Serial.print("Mag:x:");
  Serial.print(scaled.XAxis);
  Serial.print(" y:");
  Serial.print(scaled.YAxis);
  Serial.print(" z:");
  Serial.println(scaled.ZAxis);
  logfile.print("Mag:x:");
  logfile.print(scaled.XAxis);
  logfile.print(" y:");
  logfile.print(scaled.YAxis);
  logfile.print(" z:");
  logfile.println(scaled.ZAxis);

  // Correct for when signs are reversed.
  if(heading < 0) heading += 2*PI;
  if(heading > 2*PI) heading -= 2*PI;
```

```
  return heading * RAD_TO_DEG; //radians to degrees
}

/**
Method produces an error for Adafruit SD Logger
*/
void error(char *str)
{
  Serial.print("error: ");
  Serial.println(str);

  while(1);
}

/**
Method to interface with the DS18S20 Temperature sensor via OneWire
*/
void TempSensor()
{
  byte i;
  byte present = 0;
  byte data[12];
  byte addr[8];

  int HighByte, LowByte, TReading, SignBit, Tc_100, Whole, Fract;

  if ( !ds.search(addr)) {
     Serial.print("No more addresses.\n");
     ds.reset_search();
     return;
```

```
}

Serial.print("R=");
for( i = 0; i < 8; i++) {
  Serial.print(addr[i], HEX);
  Serial.print(" ");
}

if ( OneWire::crc8( addr, 7) != addr[7]) {
   Serial.print("CRC is not valid!\n");
   return;
}

if ( addr[0] == 0x10) {
   Serial.print("Device is a DS18S20 family device.\n");
}
else if ( addr[0] == 0x28) {
   Serial.print("Device is a DS18B20 family device.\n");
}
else {
   Serial.print("Device family is not recognized: 0x");
   Serial.println(addr[0],HEX);
   return;
}

ds.reset();
ds.select(addr);
ds.write(0x44,1);        // start conversion, with parasite power on at the end

delay(1000);     // maybe 750ms is enough, maybe not
```

```
// we might do a ds.depower() here, but the reset will take care of it.

present = ds.reset();
ds.select(addr);
ds.write(0xBE);         // Read Scratchpad

Serial.print("P=");
Serial.print(present,HEX);
Serial.print(" ");
for ( i = 0; i < 9; i++) {           // we need 9 bytes
  data[i] = ds.read();
  Serial.print(data[i], HEX);
  Serial.print(" ");
}
Serial.print(" CRC=");
Serial.print( OneWire::crc8( data, 8), HEX);
Serial.println();

LowByte = data[0];
HighByte = data[1];
TReading = (HighByte << 8) + LowByte;
SignBit = TReading & 0x8000;  // test most sig bit
if (SignBit) // negative
{
  TReading = (TReading ^ 0xffff) + 1; // 2's comp
}
Tc_100 = (6 * TReading) + TReading / 4;    // multiply by (100 * 0.0625) or 6.25

Whole = Tc_100 / 100;  // separate off the whole and fractional portions
Fract = Tc_100 % 100;
```

```
 if (SignBit) // If its negative
 {
   Serial.print("-");
 }
 Serial.print("Temperature Probe = ");
 Serial.print(Whole);
 Serial.print(".");
 if (Fract < 10)
 {
   Serial.print("0");
 }
 Serial.print(Fract);

 Serial.print(" C\n");
}



//if available ram is less than 300, card init will fail
   int freeRam ()
   {
   extern int __heap_start, *__brkval;
   int v;
   return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
   }
```

Arduino UNO GPS and Data Logging

```
// GPS logging and transmission using I2C on Adafruit Ultimate GPS Logger Shield
// Author: Chuhan Qin
```

```
// April, 2014



#include <Adafruit_GPS.h>
#include <SoftwareSerial.h>
#include <SD.h>
#include <avr/sleep.h>
#include <SPI.h>
#include <Wire.h>

SoftwareSerial mySerial(8, 7);
Adafruit_GPS GPS(&mySerial);

// Set GPSECHO to 'false' to turn off echoing the GPS data to the Serial console
// Set to 'true' if you want to debug and listen to the raw GPS sentences
#define GPSECHO  true
/* set to true to only log to SD when GPS has a fix, for debugging, keep it false */
#define LOG_FIXONLY false

// Set the pins used
#define chipSelect 10
#define ledPin 13

File logfile;

// Return available ram size in int.
// If this goes below 320, the logfile won't open.
int freeRam()
{
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}


// read a Hex value and return the decimal equivalent
uint8_t parseHex(char c) {
  if (c < '0')
    return 0;
  if (c <= '9')
    return c - '0';
  if (c < 'A')
    return 0;
  if (c <= 'F')
    return (c - 'A')+10;
}
```

```
void setup() {

  // connect at 115200 so we can read the GPS fast enough and echo without dropping chars
  Serial.begin(115200);
  Serial.println(freeRam());
  Wire.begin(2);              // join i2c bus with address #4
  Wire.onRequest(requestEvent);

  pinMode(ledPin, OUTPUT);

  // make sure that the default chip select pin is set to
  // output, even if you don't use it:
  pinMode(10, OUTPUT);

  // see if the card is present and can be initialized:
  if (!SD.begin(chipSelect)) {
    Serial.println("f"); // Failed
    exit(2);
  }

  // connect to the GPS at the desired rate
  GPS.begin(9600);

  // uncomment this line to turn on RMC (recommended minimum) and GGA (fix data) including altitude
  GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);
  //GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCONLY);

  // Set the update rate
  GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ);   // 1 or 5 Hz update rate

  // Turn off updates on antenna status, if the firmware permits it
  GPS.sendCommand(PGCMD_NOANTENNA);

}

void loop() {

  //Serial.print("R");
  Serial.println(freeRam());

  logfile = SD.open("0", FILE_WRITE);
  if( ! logfile ) {
    Serial.println("C"); // Error
    exit(3);
  }

  // Retrive gps sentence
```

```
  while(gps_receive()==0);

  requestEvent();

  logfile.close();
  delay(1000);

}


// Read from the GPS chip, need to call numerous time before a full sentence is received
byte gps_receive(){
  char c = GPS.read();

  // if a sentence is received, we can check the checksum, parse it...
  if (GPS.newNMEAreceived()) {

    if (!GPS.parse(GPS.lastNMEA()))   // this also sets the newNMEAreceived() flag to false
      return 0;  // we can fail to parse a sentence in which case we should just wait for another

    if (LOG_FIXONLY && !GPS.fix) {
       Serial.print("No Fix");
       return 0;
    }

    char *stringptr = GPS.lastNMEA();
    uint8_t stringsize = strlen(stringptr);
    if (stringsize != logfile.write((uint8_t *)stringptr, stringsize))    //write the string to the SD file
      exit(4);
    if (strstr(stringptr, "RMC"))   logfile.flush();
    return 1;
  }
  return 0;
}


// distance using Haversine formula
double get_distance(){
  // Richmond St. and Twelve Mile Rd, ~ 12 km from Western
  double latitude_destination  =  43.11;
  double longitude_destination = -81.33;

  double latitude_current  = get_gps_data(1);
  double longitude_current = get_gps_data(2) * -1;

  if(latitude_current == 0.00 && longitude_current == 0.00){
    Serial.println("E");
    return 0.00;
```

```
  }
  else{
    // Print current coordinates after conversion
    //Serial.print("Lat:  ");  Serial.println(latitude_current);
    //Serial.print("Lon: ");   Serial.println(longitude_current);

    float dist_calc   = 0;
    float dist_calc2  = 0;
    float diflat      = 0;
    float diflon      = 0;

    diflat = radians(latitude_current-latitude_destination);
    latitude_destination = radians(latitude_destination);
    latitude_current = radians(latitude_current);
    diflon = radians((longitude_current)-(longitude_destination));

    dist_calc   = (sin(diflat/2.0)*sin(diflat/2.0));
    dist_calc2  = cos(latitude_destination);
    dist_calc2 *= cos(latitude_current);
    dist_calc2 *= sin(diflon/2.0);
    dist_calc2 *= sin(diflon/2.0);
    dist_calc  += dist_calc2;

    dist_calc=(2*atan2(sqrt(dist_calc),sqrt(1.0-dist_calc)));

    dist_calc *= 6371000.0; //Converting to meters
    Serial.print("D:");
    Serial.println(dist_calc);
    return dist_calc;
  }
}


// I2C communication
// Sends current coordinates to master board
// The hemisphere flags are set to N and W automatically to save spaces
void requestEvent(){

  char data[12] = {};
  char temp[5] = {};

  itoa((int)GPS.latitude, temp,10);
  strcat(data, temp);

  itoa((int)GPS.longitude, temp,10);
  strcat(data, temp);

  strcat(data, "\n");
```

```
  Wire.write(data);
}


/* End code */
```

Elecfreaks Arduino UNO Autopilot System

```
// Autopilot
// Author: Don Vo
// April, 2014


//70-540 work as valid values
  //0 is break

#include <Servo.h>
#include <Wire.h>
#include <math.h>
#include <cmath.h>
#include <stdio.h>

#define ACCELE_SCALE 2  // accelerometer full-scale, should be 2, 4, or 8

/* LSM303 Address definitions */
#define LSM303_MAG  0x1E  // assuming SA0 grounded
#define LSM303_ACC  0x18  // assuming SA0 grounded

#define X 0
#define Y 1
#define Z 2

/* LSM303 Register definitions */
#define CTRL_REG1_A 0x20
#define CTRL_REG2_A 0x21
#define CTRL_REG3_A 0x22
#define CTRL_REG4_A 0x23
#define CTRL_REG5_A 0x24
#define HP_FILTER_RESET_A 0x25
#define REFERENCE_A 0x26
#define STATUS_REG_A 0x27
#define OUT_X_L_A 0x28
#define OUT_X_H_A 0x29
#define OUT_Y_L_A 0x2A
#define OUT_Y_H_A 0x2B
```

```
#define OUT_Z_L_A 0x2C
#define OUT_Z_H_A 0x2D
#define INT1_CFG_A 0x30
#define INT1_SOURCE_A 0x31
#define INT1_THS_A 0x32
#define INT1_DURATION_A 0x33
#define CRA_REG_M 0x00
#define CRB_REG_M 0x01//refer to the Table 58 of the datasheet of LSM303DLM
#define MAG_SCALE_1_3 0x20//full-scale is +/-1.3Gauss
#define MAG_SCALE_1_9 0x40//+/-1.9Gauss
#define MAG_SCALE_2_5 0x60//+/-2.5Gauss
#define MAG_SCALE_4_0 0x80//+/-4.0Gauss
#define MAG_SCALE_4_7 0xa0//+/-4.7Gauss
#define MAG_SCALE_5_6 0xc0//+/-5.6Gauss
#define MAG_SCALE_8_1 0xe0//+/-8.1Gauss
#define MR_REG_M 0x02
#define OUT_X_H_M 0x03
#define OUT_X_L_M 0x04
#define OUT_Y_H_M 0x07
#define OUT_Y_L_M 0x08
#define OUT_Z_H_M 0x05
#define OUT_Z_L_M 0x06
#define SR_REG_M 0x09
#define IRA_REG_M 0x0A
#define IRB_REG_M 0x0B
#define IRC_REG_M 0x0C
#define PI 3.14159    // 3.14159

/* Global variables */
int value = 0; // set values you need to zero
Servo firstESC, secondESC; //Create as much as Servoobject you want. You can controll 2 or more Servos
at the same time
unsigned long start, finished, elapsed = 0;
boolean valveOpen = false;
int minSpeed = 70;
int maxSpeed = 500;
double offset = 30.00;
boolean direction = false;
int flightDuration = 0;

// compass variables
int accel[3];  // we'll store the raw acceleration values here
int mag[3];  // raw magnetometer values stored here
float realAccel[3];  // calculated acceleration values here
double desiredBearing = 0.00;
double currentBearing = 0.00;

// Latitude and longitude
```

```
  // latA & lonA is the current coordinates
  // latB and lonB is the coordinates of where the balloon is expected to land
  double latA = 0.00;
  double lonA = 0.00;

  double latB = 43.135185;
  double lonB = -81.338175;

  double altitude = 0.0;

  // start time
  uint32_t startTime = 0;

  // current time
  uint32_t currentTime = 0;

  // bootup time
  uint32_t bootTime = 0;


void setup() {
  // SETUP MOTORS
  firstESC.attach(9);    // attached to pin 9 with motor 1
  secondESC.attach(10);  // attached to pin 10 with motor 2
  Serial.begin(115200);    // start serial at 9600 baud
  Serial.println("0=minThrottle, 1023=maxThrottle");
  value = 1023;
  firstESC.write(1023);
  secondESC.write(1023);

  // valve code
  pinMode(11,OUTPUT);
  digitalWrite(11,LOW);
  pinMode(12,OUTPUT);
  digitalWrite(12,LOW);


  bootTime = GetCurrentTime();

  // GPS code
   Wire.begin();  // Start up I2C, required for LSM303 communication
  initLSM303(ACCELE_SCALE);  // Initialize the LSM303, using a SCALE full-scale range
}


void initLSM303(int fs)
{
  LSM303_write(0x27, CTRL_REG1_A);  // 0x27 = normal power mode, all accel axes on
```

```
  if ((fs==8)||(fs==4))
    LSM303_write((0x00 | (fs-fs/2-1)<<4), CTRL_REG4_A);  // set full-scale
  else
    LSM303_write(0x00, CTRL_REG4_A);
  LSM303_write(0x14, CRA_REG_M);  // 0x14 = mag 30Hz output rate
  LSM303_write(MAG_SCALE_1_3, CRB_REG_M); //magnetic scale = +/-1.3Gauss
  LSM303_write(0x00, MR_REG_M);  // 0x00 = continouous conversion mode
}

void printValues(int * magArray, int * accelArray)
{
  /* print out mag and accel arrays all pretty-like */
  Serial.print(accelArray[X], DEC);

  Serial.print("\t");
  Serial.print(accelArray[Y], DEC);
  Serial.print("\t");
  Serial.print(accelArray[Z], DEC);
  Serial.print("\t\t");

  Serial.print(magArray[X], DEC);
  Serial.print("\t");
  Serial.print(magArray[Y], DEC);
  Serial.print("\t");
  Serial.print(magArray[Z], DEC);
  Serial.println();
}

float getHeading(int * magValue)
{
  // see section 1.2 in app note AN3192
  float heading = 180*atan2(magValue[Y], magValue[X])/PI;  // assume pitch, roll are 0

  if (heading <0)
    heading += 360;

  return heading;
}

float getTiltHeading(int * magValue, float * accelValue)
{
  // see appendix A in app note AN3192
  float pitch = asin(-accelValue[X]);
  float roll = asin(accelValue[Y]/cos(pitch));

  float xh = magValue[X] * cos(pitch) + magValue[Z] * sin(pitch);
  float yh = magValue[X] * sin(roll) * sin(pitch) + magValue[Y] * cos(roll) - magValue[Z] * sin(roll) *
cos(pitch);
```

```
  float zh = -magValue[X] * cos(roll) * sin(pitch) + magValue[Y] * sin(roll) + magValue[Z] * cos(roll) *
cos(pitch);
  float heading = 180 * atan2(yh, xh)/PI;

  if (yh >= 0)    return heading;
  else    return (360 + heading);
}

void getLSM303_mag(int * rawValues)
{
  Wire.beginTransmission(LSM303_MAG);
  Wire.write(OUT_X_H_M);
  Wire.endTransmission();
  Wire.requestFrom(LSM303_MAG, 6);
  for (int i=0; i<3; i++)
    rawValues[i] = (Wire.read() << 8) | Wire.read();
  int temp;
  temp = rawValues[Y];
  rawValues[Y] = rawValues[Z];
  rawValues[Z] = temp;
}

void getLSM303_accel(int * rawValues)
{
  rawValues[Z] = ((int)LSM303_read(OUT_X_L_A) << 8) | (LSM303_read(OUT_X_H_A));
  rawValues[X] = ((int)LSM303_read(OUT_Y_L_A) << 8) | (LSM303_read(OUT_Y_H_A));
  rawValues[Y] = ((int)LSM303_read(OUT_Z_L_A) << 8) | (LSM303_read(OUT_Z_H_A));
  // had to swap those to right the data with the proper axis
}

byte LSM303_read(byte address)
{
  byte temp;

  if (address >= 0x20)
    Wire.beginTransmission(LSM303_ACC);
  else
    Wire.beginTransmission(LSM303_MAG);

  Wire.write(address);

  if (address >= 0x20)
    Wire.requestFrom(LSM303_ACC, 1);
  else
    Wire.requestFrom(LSM303_MAG, 1);
  while(!Wire.available())
    ;
  temp = Wire.read();
```

```
  Wire.endTransmission();

  return temp;
}

void LSM303_write(byte data, byte address)
{
  if (address >= 0x20)
    Wire.beginTransmission(LSM303_ACC);
  else
    Wire.beginTransmission(LSM303_MAG);

  Wire.write(address);
  Wire.write(data);
  Wire.endTransmission();
}
/*********************************************/

void idle() {
   Serial.println("Idle Mode");
}

void test() {
    Serial.println("Entering Testing Mode ...");
    Serial.println("Step back!");
    delay(2000);
    firstESC.write(70);  // tests fan blades
    secondESC.write(70);  // tests fan blades
    delay(8000);
    Serial.println("Exiting Test Mode..");
    delay(3000);
    value = 0;  // Brings the program back to Idle Mode
}


void land() {
// open the valve
  valveSwitch();
}


void getLocation() {
boolean locFix = false;
boolean gibberish = false;
boolean readFin = false;
char coord[18] = {};
int i = 0;
```

```
  Wire.begin();
  Serial.println("lggdollllll");
  Wire.requestFrom(16,14);    // request 16 bytes from slave device #16
  Serial.println("enters wire");

  while(Wire.available() > 0 && gibberish == false && readFin == false)    // slave may send less than
requested
  {
    char c = Wire.read(); // receive a byte as character

    if ( 48 <= (int)c <= 57) {
      Serial.println(c);
      coord[i] = c;
    }
    else if (c == '\n') {
      coord[i] = c;
      readFin = true;
    }
    else {
      Serial.println("lol");
      gibberish = true;
      latA = 9999.00;
      return;
    }
    i++;
  }
  latA = (((double)coord[0] - 48) * 1000) + (((double)coord[1] - 48) * 100) + (((double)coord[2] - 48) * 10)
+ (((double)coord[3] - 48) * 1);
  lonA = (((double)coord[4] - 48) * 1000) + (((double)coord[5] - 48) * 100) + (((double)coord[6] - 48) * 10)
+ (((double)coord[7] - 48) * 1);

  int j = 8;
  boolean gotAlt = false;
  altitude = 0.0;
  int altLength = 0;

  while (gotAlt == false && coord[j] != '\n') {
   if (48 <= (int)coord[j] <= 57) {
     altLength++;
     j++;
   }
   else {
     gotAlt = true;
   }
  }

  // reset j to the index of the first altitude number
```

```
 j = 8;

 Serial.println("here");
 while (altLength > 0) {
  altitude = altitude + (((double)coord[j] - 48) * pow(10, altLength - 1));
  j++;
  altLength--;
 }

 // converts lon and lat to 0 - 90 scale
 latA = latA / 100;
 lonA = lonA / (-100);
 Serial.println(latA);
 Serial.println(lonA);
 Serial.println(altitude);
}

void vent() {

 while ((int)altitude >= 24000) {
  // Opens the valve
  //black wire
  digitalWrite(11,LOW);
  //red wire
  digitalWrite(12,HIGH);
  delay(500);
  digitalWrite(12,LOW);
 }

 // Closes the valve
  //red wire
  digitalWrite(12,LOW);
  //black wire
  digitalWrite(11,HIGH);
  delay(500);
  digitalWrite(11,LOW);
}


void launch() {
 while ((int)altitude != 0) {

  if ((int)altitude < 300) {  // stops motors if altitude is less than 300m
    firstESC.write(0);
    secondESC.write(0);
  }
  else if (((int)(currentTime - startTime))/1000 > 7200 && (int)altitude < 18000) {  // lands if below 18km
for longer than 2 hours
```

```
    land();
   }
   else if ((int)altitude > 24000) {  // releases helium if the balloon is over 24km
     vent();
   }
   else if (((int)(currentTime - startTime)) / 1000 > 7200) {  // lands the drone if it's been flying for over 3
hours
    land();
   }
   else {  // runs when drone is 70m+ altitude
    fly();
   }
  }
}


/**
Method to get the current time and print it
*/
uint32_t GetCurrentTime()
{
    // log milliseconds since starting
   uint32_t time = millis();
   Serial.print("Millis since start: ");
   Serial.println(currentTime); // milliseconds since start
   return time;
}

// opens or closes the valve based on its current position

void valveSwitch()
{
 if(valveOpen == false)
 {
   //red wire
   digitalWrite(12,LOW);
   //black wire
   digitalWrite(11,HIGH);
   delay(500);
   digitalWrite(11,LOW);
   Serial.println("valve is now closed");
 }
 else
 {
   //black wire
   digitalWrite(11,LOW);
   //red wire
   digitalWrite(12,HIGH);
```

```
   delay(500);
   digitalWrite(12,LOW);
   Serial.println("valve is now open");
  }
}


void getBearing() {

  getLocation();
  while ( latA == 9999.00 ) {  // calls getLocation() again if it was unable to previously
   getLocation();
  }

  /* This is the formula to calculate the desired bearing based on current position and desired position
   Δϕ = ln( tan( latB / 2 + π / 4 ) / tan( latA / 2 + π / 4) )
  Δlon = abs( lonA - lonB )
  bearing :  θ = atan2( Δlon ,  Δϕ )

  Note: 1) ln = natural log      2) if Δlon > 180°  then   Δlon = Δlon (mod 180).
*/
double deltaPi = log( tan( (double)latB / 2 + PI /4 ) / tan( (double)latA / 2 + PI / 4 ));
double deltaLon = abs( (double)lonA - (double)lonB );

if ( deltaLon > 180.00 ) {
   deltaLon = (double)((int)deltaLon % 180);
}
  desiredBearing = atan2( deltaLon , deltaPi );
}



void fly() {
  // Gets the desired bearing
  getBearing();

  // Fixes orientation if the drone points off the original launch direction by an offset of 35 degrees
  while (currentBearing < offset - desiredBearing) {
   // turn motor 1 on (left), motor 2 off (right)
   firstESC.write(100);
   secondESC.write(0);
  }

  while (currentBearing > offset + desiredBearing) {
   firstESC.write(0);  // tests fan blades
   secondESC.write(100);  // tests fan blades
  }
```

```
  while (offset - desiredBearing < currentBearing < offset + desiredBearing) {
    // if the drone is between both offsets, turn the blades on
     firstESC.write(100);  // tests fan blades
     secondESC.write(100);  // tests fan blades
  }
}




void loop() {

//First connect your ESC WITHOUT Arming. Then Open Serial and follo Instructions

  firstESC.write(value);
  secondESC.write(value);
  Serial.println(value);

  delay(3000);


/** Compass code **/

Serial.println("***************");
  getLSM303_accel(accel);  // get the acceleration values and store them in the accel array
  Serial.println("here");
  while(!(LSM303_read(SR_REG_M) & 0x01))
    ;  // wait for the magnetometer readings to be ready
  getLSM303_mag(mag);  // get the magnetometer values, store them in mag
  printValues(mag, accel);  // print the raw accel and mag values, good debugging
  Serial.println("Acceleration of X,Y,Z is");
  for (int i=0; i<3; i++)
  {
    realAccel[i] = accel[i] / pow(2, 15) * ACCELE_SCALE;  // calculate real acceleration values, in units of g
Serial.print(realAccel[i]);
Serial.println("g");
  }
  currentBearing = (double)getHeading(mag);
  Serial.println("current bearing: ");
  Serial.print(currentBearing);
/** End of compass code **/




delay(1000);
  bootTime = GetCurrentTime()/1000;
  Serial.println(bootTime);
```

```
 if (bootTime > 600) {
  Serial.println("entering launch process");
  startTime = GetCurrentTime();
  launch();
 }



 if(Serial.available())
 {
  value = Serial.parseInt();    // Parse an Integer from Serial
  Serial.println(value);  //prints value motor set to

  if (value == 0) {
   idle();
   Serial.println(GetCurrentTime());
   getLocation();
  }
  else if (value == 1) {
   test();
  }
  else if (value == 2) {
   if (valveOpen == true) {
    valveOpen = false;
    value = 0;
   }
   else {
   valveOpen = true;
   value = 0;
  }
    valveSwitch();
  }
  else if (value == 3) {
   startTime = GetCurrentTime();
   launch();
  }
  else {  // random values entered will keep the program in Idle Mode
   value = 0;
  }
 }
}
```