**Spring**

15

Senior Project

Trevor Yager

Quadcopter Aid in Rescue

08

**Fall**

Abstract 3

Introduction 4

Purpose of project 4

Why I wanted to do this project 4

Similar projects and my innovations 4

Coming sections 5

Plan of Action 6

Schedule 6

Implementation 7

Research 7

Construction 8

Coding 9

Features 11

Interrupts 11

servo.writeMicroseconds(uS) 12

MPU-6050 12

Complementary Filter 13

PID 13

Problems 15

Issues 15

Blockers 16

Conclusion 18

Improvements 18

References 19

Code 20

# Abstract

The purpose of this project was to build a drone capable of aiding in a natural disaster. If an earthquake were to happen, then first responders would need to be able to see the wreckage from many different vantage points quickly. This means that they would need something capable of being in the air and also capable of showing them what it sees. The quadcopter that I envisioned is able to quickly rise to large heights, stream the video it is seeing to a local computer via Wi-Fi or Bluetooth, and is also able to automatically stabilize in the air. In order to do this I am going to use an Arduino Mega that will be hooked up to multiple sensors (MPU6050) and a camera (Adafruit TTL with SD card). The end goal of this project would be to have a quadcopter that is able to sustain consistently stable flight.

# Introduction

## Purpose of project

Depending on the type of disaster that occurs, first responders need to be able to view the landscape and see where survivors are. In order to aid in this type of process I decided to go about building a quadcopter that will provide aerial support. This means that the quadcopter needs to be able to show the responders what it sees so they can decide how to go about their operations.

## Why I wanted to do this project

One thing that sparked my interest with this project is that I would be able to learn more about the hardware sides of things. While undergoing the Computer Science degree at CSU, I have mainly dealt with software only. We have not had the chance to do much with any physical hardware except when it came to working with routers and cables in Networking. This project was a chance for me to explore the world of hardware and see how hardware pieces communicate with each other. It also ended up being a chance for me to program how the pieces of hardware would end up talking to each other.

## Similar projects and my innovations

There have been similar projects done where Arduino boards have been used to create quadcopters, but these projects are mainly closed source. The source code that is publicly available on similar projects tends to differ from mine due to different pieces of hardware being connected to the board. The best way to go about this was to look at each piece of hardware separately and figure out how I wanted to combine them for my own project. One difference between my quadcopter and others is that mine will also be able to take pictures while it is airborne; most Arduino based quadcopters will only be used for recreational flying purposes.

## Coming sections

In the coming pages you will be able to read about how I went about this project and some of the challenges I faced. I will try to clearly lay out the project and how it works in a step-by-step walkthrough. This project has many moving parts and each one can be complicated to understand. In order for me to clearly describe what is going on, I will first give a large overview of the project and then in the later sections I will go into more detail on each of the main functions and pieces of hardware.

# Plan of Action

## Schedule

* Initial research
* Learn Arduino
* Order parts / construct quadcopter
* Enable transmitter/receiver
* Arm ESCs
* Calculate appropriate motor signals
* Camera / storing pictures
* Automatic stabilization

# Implementation

## Research

I started my project by first trying to learn the pieces of a quadcopter; I had never flown or seen one in person so I needed to understand all of the moving parts that go into building one. I started by first doing Google searches on quadcopters and seeing the parts that people were using; I would look at pictures and see how parts were connected to each other and then look up their purpose. I ended up not having to spend too much time doing this because there were guides publicly available when it came to building a quadcopter. I did have to do calculations on the weight and make sure that my motors would be able to handle what I had in mind for this project. The following is the list of parts that I used when constructing my quadcopter:

* Q450 V3 Glass Fiber Quadcopter Frame 450mm
* D2830-11 1000kv Brushless Motor
* ZIPPY Compact 4000mAh 3S 25C Lipo Pack
* TURNIGY Plush 30amp Speed Controller
* TURNIGY Accucel-6 50W 6A Balancer/Charger
* FS-TH9X 2.4GHz 9CH Transmitter/Receiver
* Cheerwing Carbon Nylon 10x4.5” Propellers
* MPU-6050
* TTL Serial JPEG Camera with NTSC Video
* Micro SD Card Breakout Board

I also had to do extensive research on Arduino boards because I had never experimented with one myself. Arduino is open source and gives users a way to experiment with microcontroller boards. These boards come in many shapes and sizes, each with their own hardware specifications, and support many types of add-ons. During my research, I had to figure out what type of board would be best for this project and also learn about the environment that I would be using to write the code in. Arduino comes with its own platform, Arduino Development Environment, which allows the users to write code and upload it to the board. After learning all about Arduino, I finally came to a decision on the type of board I wanted to use, the Arduino Mega 2560. This board has 15 PWM pins, compared to 6 PWM pins on the Uno, which ended up being a crucial in the way I implemented my project. The following specifications for the Arduino Mega 2560 have been pulled from the Arduino website:

|  |  |
| --- | --- |
| Microcontroller | ATmega2560 |
| Operating Voltage | 5V |
| Input Voltage (recommended) | 7-12V |
| Input Voltage (limits) | 6-20V |
| Digital I/O Pins | 54 (of which 15 provide PWM output) |
| Analog Input Pins | 16 |
| DC Current per I/O Pin | 40 mA |
| DC Current for 3.3V Pin | 50 mA |
| Flash Memory | 256 KB of which 8 KB used by bootloader |
| SRAM | 8 KB |
| EEPROM | 4 KB |
| Clock Speed | 16 MHz |

## Construction

After receiving all of the needed parts in the mail, I went about constructing the quadcopter. Many of the parts required modifications before I would be able to do anything; the battery needed to have a Deans connector soldered to the end of it, the ESCs needed to have banana connectors soldered to the motor side and all of the wires not on the motor side were soldered to the quadcopter frame. Coming off of the side of the quadcopter frame, I had to solder a Deans connector so that the frame would plug into the battery.

After I had the initial soldering down and all of the pieces connected together, I had to then make sure everything was going to stay in place during future flight. In order to achieve this, I used zip ties to tie down the ESCs, the receiver, and the battery.

Now that all of the initial parts were held down, I had to add the Arduino Mega, MPU-6050, TTL Camera, and Micro SD card breakout board. I was running very low with room on top of the frame so I ended up adding my own customizations to the frame. I attached a piece of wood to the top of the frame with a little over an inch between it and the top of the frame, and then I attached another piece of wood of the piece I just put on with about an inch between the two; there is now basically three levels to place items.

## Coding

During my time coding out this project, I tried many different ways going about each task; I did this in order to find out which method interacted the best with the quadcopter. Some of these tests involved different ways of writing to the ESCs, different ways of calculating values to send to the ESC, different ways of handling interrupts, and many more. This coding process for the quadcopter did involve some trial and error tactics, while at the same time it still required knowledge about the overall process.

This first thing I did was try to pick up values from the receiver; this is where I started to learn about interrupts. The receiver would constantly be receiving different values from the transmitter, and this meant that the Arduino board had to be alerted each time a new value came in. Because the transmitter values being sent are very important, the quadcopter needs to be adjusted in real time to these values. I ended up using a publicly available library to turn PWM pins on the Arduino board into pins that are capable of handling interrupts. In my code, I would then attach those pins to an interrupt method that would be called each time a new value was sent to that pin from the receiver.

Now that I was able to get the values the receiver was picking up, I had to be able to send these values to the motors; in order to do this, the ESCs had to be armed before they would send any voltage to the motors. In order to do this, I had to experiment with values to see what range had to be sent in microseconds (uS). After finding this out, I was then able to start sending many different values in microseconds to the ESCs to see how quickly the motors would react. The value range that I was using in microseconds was between 1040 and 1840; the higher the value, the faster the motors would spin. The value that ended up being sent to the ESCs to arm them was around 1150.

Now that the ESCs were armed and I could send values to the motors, I had to interpret the values being sent front the receiver and make a corresponding microsecond that would be appropriate for each motor; this means that I had to take the values coming from all four channels and then turn them into the appropriate value for each individual motor. The first step was to take the throttle channel because it is going to determine if I would send values that would rotate the motors at all; if the throttle value was below a certain value, then none of the motors should do anything. After looking at the throttle value, the corresponding yaw, pitch, and roll needed to be taken into account. The pitch is what determines if the quadcopter will go forwards or backwards, the roll will determine if it goes left or right, and yaw will determine if the quadcopter starts to turn to face a direction.

Now that I had the appropriate values being sent to each motor, I then went about adding the camera. This proved to be a pain at first because of conflicting library issues, but in the end I figured out a way to get the libraries to work with my code. The camera is very simple and will use a timer to take a picture every X amount of time and then store it to a micro SD card.

After calculating the values to send to the appropriate motor and getting the camera working, basic flight was then achieved. This meant that the quadcopter would take the values from the transmitter and send them to the motors. This is a very basic version of a quadcopter and depends entirely on the pilot. Flying under these condition would be difficult for even the most seasoned pilots, so I decided to add a sensor (MPU-6050) to the board. The point of adding this sensor was to allow the quadcopter to stabilize itself while it is in the air; if there is a gust of wind or the quadcopter starts to tilt a certain direction, then it needs to be able to adjust the motors without the pilot’s immediate attention. This ended up being the part of the project that I ended on. The PID algorithm was very complex and required extensive testing.

# Features

## Interrupts

The major backbone of my code is interrupts; they are in charge of constantly viewing the data flowing from the receiver, sensors, and TTL camera. The Arduino Mega comes with six, out-of-the-box, pins that are capable of handling interrupts. This can limit the board in terms of how many incoming streams of data it can have, but with the help of the PinChangeInt Library, I was able to change some of the PWM pins on the Mega to interrupt pins also. The following is an excerpt from the Arduino website talking about the PinChangeInt library:

“What are Pin Change interrupts? The ATmega processor at the heart of the Arduino has two different kinds of interrupts: “external”, and “pin change”. There are only two external interrupt pins, INT0 and INT1, and they are mapped to Arduino pins 2 and 3. These interrupts can be set to trigger on RISING or FALLING signal edges, or on low level.”

This quote is saying that whenever the signal changes in a certain way, then the associated function will be called to handle the interrupt. There is also a third signal that can be used to call a function and that is the CHANGE signal; this is the signal I used when defining my PinChangeInts.

Something that is very important to keep in mind when using interrupts is to make sure the functions are small and fast. When an interrupt is taking place, then nothing else is being done in the meantime; this means that my code had to be fast and efficient when handling these incoming values. Also, global variables are useful when handling interrupts so that they can be accessed from anywhere within my code.

Another note is that these variables should, usually, be volatile. Declaring a variable to be volatile will reduce the chance of using “bad data” throughout your code. The following is a quote from the Arduino website talking about volatile variables and an instance of when they should be used:

“A variable should be declared volatile whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called an interrupt service routine.”

If you are using a global variable that is used for calculations and interrupts, then it could change halfway through a calculation do to an interrupt; we declare the variable to be volatile so this does not happen.

## servo.writeMicroseconds(uS)

The Arduino servo library has two ways of writing to servos, or motors, write() and writeMicroseconds(). There are two main differences between the two and it is the value that is used to write to the servo. When using write() the value will be an angle between 0 and180; when using writeMicroseconds() the value will be a microsecond between 700 and 2300. The reason I chose to use writeMicroseconds(), instead of write(), is because of how accurate it is. The range is much larger and allows for the calculations to be more precise; this means that achieving a stable flight will be much more probable.

## MPU-6050

“The InvenSense MPU-6050 sensor contains a MEMS accelerometer and a MEMS gyro in a single chip. It is very accurate, as it contains 16-bits analog to digital conversion hardware for each channel. Therefor it captures the x, y, and z channel at the same time. The sensor uses the I2C-bus to interface with the Arduino.”

The above quote is taken from Arduino website and gives a quick, but detailed, account of what this sensor is. This board is very new and advanced because it can do all of the computations on the chip itself, and it combines a gyro and accelerometer on the same board. A very large thank you goes out to jrowberg/i2cdevlib for reverse engineering this sensor when there is little to no documentation available for the sensor. Jrowberg also developed a library for the MPU-6050 that can be used in Arduino project (I used it myself). The library allows for a user to extract the yaw, pitch, and roll from the sensor when needed, among other things.

This sensor plays a critical role when it comes to making the quadcopter stabilize on its own. Without an IMU, the brain of the quadcopter, the Arduino Mega 2560, will not know where the quadcopter sits in space. By extracting the yaw, pitch, and roll from the MPU-6050, we can then perform calculations that will try and level the quadcopter out.

## Complementary Filter

When implementing the IMU (inertial measurement unit), MPU-6050, I learned that the values being read from the sensor are not always dependable. Somewhat frequently, the MPU-6050 would send very skewed data; the sensor would sometimes send data indicating it was not level while it was remaining in the same resting position for the duration of the test. This “false data” being sent is called “noise”; it can be the caused by motor vibrations, wind, or jolts to the quadcopter frame.

Another issue that I encountered with the MPU-6050 is that the gyro values tending to drift over time, and the accelerometer values were not very accurate to begin with; this is where the complementary filter comes into play. The following excerpt is from Pieter-Jan’s website and gives a quick overview of why this algorithm is needed when using an IMU (inertial measurement unit):

“The complementary filter gives us a "best of both worlds" kind of deal. On the short term, we use the data from the gyroscope, because it is very precise and not susceptible to external forces. On the long term, we use the data from the accelerometer, as it does not drift. In it's most simple form, the filter looks as follows: ”

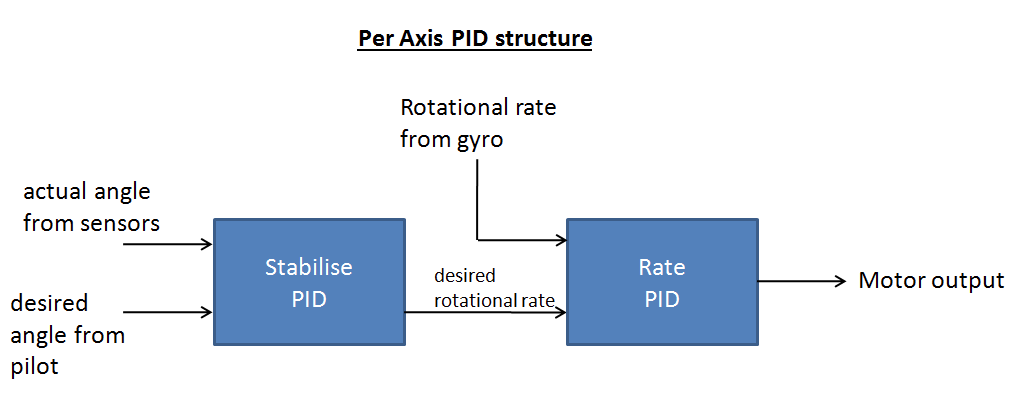
http://www.pieter-jan.com/images/equations/CompFilter_Eq.gif

As you can see, the math for these types of computations can be very complicated and have many parts, but it is very important to implement some type of filter in order to ensure a smooth and stable flight. This filter is able to do eliminate the noise that is coming from the MPU-6050 and also ensures that we use values in the future that will accurately represent where the quadcopter is in space.

## PID

PID is a complex algorithm that can be used in quadcopters in order to achieve a type of flight determined by the pilot. With a little understanding of how the algorithm works, the programmer/pilot can adjust the setting to achieve a flight that is more sensitive to changes in the sensor, or to achieve a flight that relies mainly on the pilot’s input. The following quote on the PID algorithm comes from Oscar:

“PID (proportional-integral-derivative) is a closed-loop control system that try to get the actual result closer to the desired result by adjusting the input. Quadcopters or multicopters use PID controller to achieve stability.”

Arduino has its own PID library that I decided to use in my own code. The variables that are used in this function come from MPU-6050, the receiver, and also three different variables that will determine how sensitive the compute function should be to the MPU-6050; depending on the settings of the function, it will then calculate a value that determines where the quadcopter needs to be based on current errors (P), past errors (I), and future errors (D). The compute functions needs to be ran every iteration in the main loop of the Arduino; a good representation of how this looks comes from Oscar again. 

After the PID compute function has ran, we will then use the output variable to determine the value we need to send to the motor. This value depends on if the pilot is trying to change the yaw, pitch, or roll of the quadcopter.

# Problems

## Issues

Interrupts ended up taking me a little longer than I would have liked; everything in my project revolves around interrupts, so taking the time to get them working properly and efficiently was worth the extra work. I ended up finding a way to do my interrupts that would work in main loop so that I would turn off interrupts, copy the current values being used in interrupts, and then turn interrupting back on; this gave me the flexibility to use the most recent interrupt values when I needed to.

Towards the end of building the quadcopter I decided to add the camera to the board and see if I could get it to work. The first thing I tried to do was test the camera on the board all by itself; this means that I wired the TTL camera and the micro SD card storage to the Arduino board. I then imported the library for the camera and SD card and uploaded it to the board and did my testing. All went well and the camera was storing the images properly. The next step was to add the camera to my current code and see if it would still work. After I added the code and did an initial compile of the code I was presented with multiple errors; the errors were pointing me to two specific libraries and saying that they were referencing the same register. In order to fix this I had to go into both libraries and compare the two. After much research, comparing, and time I learned that it was a problem with how the timers work in the Arduino board and libraries.

The way that interrupts work on the Arduino is that a timer is associated with certain sets of pins; this means that moving one pin over can change the timer that the Arduino board is using; the problem was that the PinChangeInt library was using the timer associated with the pins that the SoftwareSerial library was using for the camera. In order to fix the problem, I was going to have to edit both of these libraries so that they would play nice together. I copied both of the current libraries and renamed them to MyPinChangeInt and MySoftwareSerial. The big change with these two new libraries is that I did a global name change of variables and methods; this means that all occurences of PinChangeInt and SoftwareSerial were change to MyPinChangeInt and MySoftwareSerial, respectively. When looking at the pins and timers being defined in MyPinChangeInt, I noticed that it was defining a set of pins that I was not using; after noticing this, I removed all pins that were not being used by my project and then added my two custom libraries back to the Arduino libraries. I then altered my current code to import the new libraries and, sure enough, the code compiled correctly.

The MPU-6050 has a buffer that holds the values and uses the FIFO method. Something that had to be paid attention to was that the sensor would sometimes have a FIFO overflow and this would cause issues with the rest of the code; this meant that I had to dig a little deeper and see why the overflow kept occurring. After reading more about the hertz of the two boards, I had to modify them so that the sensor did not send too many values to the Mega at once. In order to do this, I had to add a line of code to my own code when I called Wire.begin(); the line that I added was ‘TWBR = 24;’. This increases the Mega from 200kHz mode @ 8MHz CPU to 400kHz mode @ 16MHz CPU; this increases the amount of operations the board will do (faster). I then went into the MPU6050\_6AXIS\_MOTIONAPPS20.h and changed the end of line 305 from 0x01 to 0x05. This will decrease the speed at which the sensor sends values to the Mega (slower). After doing both of these modifications, the sensor quit displaying FIFO overflow in the terminal.

## Blockers

The task of getting the quadcopter to auto-stabilize in the air was my largest and most complicated problem. I first started this by learning what a PID algorithm is. This algorithm has three main parts to it: P (Proportional) is present errors, I (Integral) is past errors, and D (derivative) is future errors. This algorithm will work by capturing where the quadcopter is in space my looking at the sensor values and comparing where it should be in space by looking at the transmitter/receiver values. It will then calculate a value that will later be used to determine how fast the motors should spin to move the quadcopter to the proper orientation. As you can imagine, the faster this runs the better stabilized the quadcopter will be. In the end I was not able to completely solve this problem. I ended up finding new ways to better increase the values and responses of my quadcopter, but I did not have enough time to finish the auto-stabilization.

# Conclusion

## Improvements

Completing the auto stabilization of the quadcopter would be of the upmost importance in the future. Without having this type of sensing in place, the quadcopter will depend solely on the pilot and will probably not stay in the air for long.

The coding could also be split into classes for better organization; right now it is spaghetti code, and is one long text file of code.

The camera should also be able to take pictures based off of the user triggering a switch on the transmitter. This would give the pilot the ability to take pictures when they wanted to, instead of just every X amount of seconds.

# References

<http://arduino.cc/>

<http://arduino.cc/en/Reference/attachInterrupt>

<http://blog.oscarliang.net/quadcopter-pid-explained-tuning/>

<https://github.com/adafruit/Adafruit-VC0706-Serial-Camera-Library>

<https://github.com/GreyGnome/PinChangeInt>

<https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050>

<http://playground.arduino.cc/code/TimedAction>

<http://playground.arduino.cc/Main/MPU-6050>

<http://rcarduino.blogspot.co.uk/2012/04/how-to-read-multiple-rc-channels-draft.html>

<http://www.geekmomprojects.com/mpu-6050-redux-dmp-data-fusion-vs-complementary-filter/>

<http://www.pieter-jan.com/node/11>

# Code

// Receiver

#include <MyPinChangeInt.h>

// Assign your channel in pins

#define CH1\_IN\_PIN 13

#define CH2\_IN\_PIN 12

#define CH3\_IN\_PIN 11

#define CH4\_IN\_PIN 10

// These bit flags are set in bUpdateFlagsShared to indicate which

// channels have new signals

#define CH1\_FLAG 1

#define CH2\_FLAG 2

#define CH3\_FLAG 4

#define CH4\_FLAG 8

// holds the update flags defined above

volatile uint8\_t bUpdateFlagsShared;

// shared variables are updated by the ISR and read by loop.

// In loop we immediatley take local copies so that the ISR can keep ownership of the

// shared ones. To access these in loop

// we first turn interrupts off with noInterrupts

// we take a copy to use in loop and the turn interrupts back on

// as quickly as possible, this ensures that we are always able to receive new signals

volatile float unCh1InShared;

volatile float unCh2InShared;

volatile float unCh3InShared;

volatile float unCh4InShared;

float unCh1In, unCh2In, unCh3In, unCh4In;

float unCh1InLast, unCh2InLast, unCh3InLast, unCh4InLast;

float v\_ac, v\_bd; // velocity of axes

// These are used to record the rising edge of a pulse in the calcInput functions

// They do not need to be volatile as they are only used in the ISR. If we wanted

// to refer to these in loop and the ISR then they would need to be declared volatile

float ulCh1Start;

float ulCh2Start;

float ulCh3Start;

float ulCh4Start;

// ESC

#define ESC1\_OUT\_PIN 7

#define ESC2\_OUT\_PIN 6

#define ESC3\_OUT\_PIN 5

#define ESC4\_OUT\_PIN 4

#define ESC\_MIN 1000

#define ESC\_MAX 1900

#define ESC\_ARM 1150

// MPU

#include "I2Cdev.h"

#include "MPU6050\_6Axis\_MotionApps20.h"

#include "Wire.h"

MPU6050 mpu;

/\* =========================================================================

NOTE: In addition to connection 3.3v, GND, SDA, and SCL, this sketch

depends on the MPU-6050's INT pin being connected to the Arduino's

external interrupt #0 pin. On the Arduino Uno and Mega 2560, this is

digital I/O pin 2.

\* ========================================================================= \*/

// MPU control/status vars

bool dmpReady = false; // set true if DMP init was successful

uint8\_t mpuIntStatus; // holds actual interrupt status byte from MPU

uint8\_t devStatus; // return status after each device operation (0 = success, !0 = error)

uint16\_t packetSize; // expected DMP packet size (default is 42 bytes)

uint16\_t fifoCount; // count of all bytes currently in FIFO

uint8\_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars

Quaternion q; // [w, x, y, z] quaternion container

VectorFloat gravity; // [x, y, z] gravity vector

float ypr[3]; // [yaw, pitch, roll] yaw/pitch/roll container and gravity vector

// Use the following global variables and access functions to help store the overall

// rotation angle of the sensor

unsigned long last\_read\_time;

float last\_x\_angle; // These are the filtered angles

float last\_y\_angle;

float last\_z\_angle;

float last\_gyro\_x\_angle; // Store the gyro angles to compare drift

float last\_gyro\_y\_angle;

float last\_gyro\_z\_angle;

void set\_last\_read\_angle\_data(unsigned long time, float x, float y, float z, float x\_gyro, float y\_gyro, float z\_gyro) {

last\_read\_time = time;

last\_x\_angle = x;

last\_y\_angle = y;

last\_z\_angle = z;

last\_gyro\_x\_angle = x\_gyro;

last\_gyro\_y\_angle = y\_gyro;

last\_gyro\_z\_angle = z\_gyro;

}

inline unsigned long get\_last\_time() {return last\_read\_time;}

inline float get\_last\_x\_angle() {return last\_x\_angle;}

inline float get\_last\_y\_angle() {return last\_y\_angle;}

inline float get\_last\_z\_angle() {return last\_z\_angle;}

inline float get\_last\_gyro\_x\_angle() {return last\_gyro\_x\_angle;}

inline float get\_last\_gyro\_y\_angle() {return last\_gyro\_y\_angle;}

inline float get\_last\_gyro\_z\_angle() {return last\_gyro\_z\_angle;}

// Use the following global variables

// to calibrate the gyroscope sensor and accelerometer readings

float base\_x\_gyro = 0;

float base\_y\_gyro = 0;

float base\_z\_gyro = 0;

float base\_x\_accel = 0;

float base\_y\_accel = 0;

float base\_z\_accel = 0;

// This global variable tells us how to scale gyroscope data

float GYRO\_FACTOR;

// This global varible tells how to scale acclerometer data

float ACCEL\_FACTOR;

// Variables to store the values from the sensor readings

int16\_t ax, ay, az;

int16\_t gx, gy, gz;

const float RADIANS\_TO\_DEGREES = 57.2958; //180/3.14159

unsigned long t\_now;

float gyro\_x, gyro\_y, gyro\_z;

float accel\_x, accel\_y, accel\_z;

float accel\_angle\_x, accel\_angle\_y, accel\_angle\_z;

float dt;

float gyro\_angle\_x, gyro\_angle\_y, gyro\_angle\_z;

float unfiltered\_gyro\_angle\_x, unfiltered\_gyro\_angle\_y, unfiltered\_gyro\_angle\_z;

const float alpha = 0.96;

float angle\_x, angle\_y, angle\_z;

// Buffer for data output

char dataOut[256];

// ================================================================

// === INTERRUPT DETECTION ROUTINE ===

// ================================================================

volatile bool mpuInterrupt = false; // indicates whether MPU interrupt pin has gone high

void dmpDataReady() {

mpuInterrupt = true;

}

// ================================================================

// === CALIBRATION\_ROUTINE ===

// ================================================================

// Simple calibration - just average first few readings to subtract

// from the later data

void calibrate\_sensors() {

int num\_readings = 10;

// Discard the first reading (don't know if this is needed or

// not, however, it won't hurt.)

mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

// Read and average the raw values

for (int i = 0; i < num\_readings; i++) {

mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

base\_x\_gyro += gx;

base\_y\_gyro += gy;

base\_z\_gyro += gz;

base\_x\_accel += ax;

base\_y\_accel += ay;

base\_y\_accel += az;

}

base\_x\_gyro /= num\_readings;

base\_y\_gyro /= num\_readings;

base\_z\_gyro /= num\_readings;

base\_x\_accel /= num\_readings;

base\_y\_accel /= num\_readings;

base\_z\_accel /= num\_readings;

}

// Motors

#include <Servo.h>

Servo servoMotorA, servoMotorC; // Motors on the Y axis (pitch)

Servo servoMotorB, servoMotorD; // Motors on the X axis (roll)

float va, vb, vc, vd; //velocities

// PID Algorithm

#include <PID\_v1.h>

// #define PITCH\_P\_VAL 4.1f

#define PITCH\_P\_VAL 0.0f

#define PITCH\_I\_VAL 0.0f

#define PITCH\_D\_VAL 0.0f

// #define PITCH\_I\_VAL 0.01f

// #define PITCH\_D\_VAL 0.003f

#define ROLL\_P\_VAL 0.0f

#define ROLL\_I\_VAL 0.0f

#define ROLL\_D\_VAL 0.0f

#define YAW\_P\_VAL 0.0f

#define YAW\_I\_VAL 0.0f

#define YAW\_D\_VAL 0.0f

#define PITCH\_MIN -200

#define PITCH\_MAX 200

#define ROLL\_MIN -100

#define ROLL\_MAX 100

#define YAW\_MIN -180

#define YAW\_MAX 180

#define PID\_PITCH\_INFLUENCE 100

#define PID\_ROLL\_INFLUENCE 100

#define PID\_YAW\_INFLUENCE 100

float bal\_ac = 0, bal\_bd = 0, bal\_axes = 0;

PID yawReg(&last\_z\_angle, &bal\_axes, &unCh4In, YAW\_P\_VAL, YAW\_I\_VAL, YAW\_D\_VAL, DIRECT);

PID pitchReg(&last\_y\_angle, &bal\_ac, &unCh2In, PITCH\_P\_VAL, PITCH\_I\_VAL, PITCH\_D\_VAL, DIRECT);

PID rollReg(&last\_x\_angle, &bal\_bd, &unCh1In, ROLL\_P\_VAL, ROLL\_I\_VAL, ROLL\_D\_VAL, DIRECT);

// Cam and SD

#include <TimedAction.h>

#include <MySoftwareSerial.h>

#include <Adafruit\_VC0706.h>

#include <SPI.h>

#include <SD.h>

TimedAction timedAction = TimedAction(30000, takePicture);

// SD card chip select line varies among boards/shields:

// Arduino Mega w/hardware SPI: pin 53

#define chipSelect 53

// Using hardware serial on Mega: camera TX conn. to RX1,

// camera RX to TX1, no SoftwareSerial object is required:

Adafruit\_VC0706 cam = Adafruit\_VC0706(&Serial1);

// DEBUG

// #define DEBUG

// #define OUTPUT\_RECEIVER\_VALUES

// #define OUTPUT\_READABLE\_COMPLIMENTARY

// #define OUTPUT\_BAL\_AC\_BD

// #define OUTPUT\_MOTOR\_VARIABLES

void setup()

{

#ifdef DEBUG

Serial.begin(115200);

#endif

// Pin configuration

pinMode(A0, OUTPUT);

pinMode(A1, OUTPUT);

pinMode(A2, OUTPUT);

pinMode(A3, OUTPUT);

pinMode(A4, OUTPUT);

pinMode(A5, OUTPUT);

pinMode(A6, OUTPUT);

pinMode(A7, OUTPUT);

digitalWrite(A0, HIGH); // Cam PWR

digitalWrite(A1, HIGH); // SD PWR

digitalWrite(A2, HIGH); // RC PWR

digitalWrite(A3, HIGH); // MPU PWR

digitalWrite(A4, LOW); // RC GND

digitalWrite(A5, LOW); // ESC GND

digitalWrite(A6, LOW); // ESC GND

digitalWrite(A7, LOW); // ESC GND

// attach servo objects, these will generate the correct

// pulses for driving Electronic speed controllers, servos or other devices

// designed to interface directly with RC Receivers

servoMotorA.attach(ESC1\_OUT\_PIN, ESC\_MIN, ESC\_MAX);

servoMotorB.attach(ESC2\_OUT\_PIN, ESC\_MIN, ESC\_MAX);

servoMotorC.attach(ESC3\_OUT\_PIN, ESC\_MIN, ESC\_MAX);

servoMotorD.attach(ESC4\_OUT\_PIN, ESC\_MIN, ESC\_MAX);

delay(100);

// arm ESCs

servoMotorA.write(ESC\_ARM);

servoMotorB.write(ESC\_ARM);

servoMotorC.write(ESC\_ARM);

servoMotorD.write(ESC\_ARM);

delay(5000);

initCamAndSD();

initMPU();

// get calibration values for sensors

calibrate\_sensors();

set\_last\_read\_angle\_data(millis(), 0, 0, 0, 0, 0, 0);

initRegulators();

// using the MyPinChangeInt library, attach the interrupts

// used to read the channels

PCintPort::attachInterrupt(CH1\_IN\_PIN, calcCh1, CHANGE);

PCintPort::attachInterrupt(CH2\_IN\_PIN, calcCh2, CHANGE);

PCintPort::attachInterrupt(CH3\_IN\_PIN, calcCh3, CHANGE);

PCintPort::attachInterrupt(CH4\_IN\_PIN, calcCh4, CHANGE);

}

void initCamAndSD() {

// When using hardware SPI, the SS pin MUST be set to an

// output (even if not connected or used). If left as a

// floating input w/SPI on, this can cause lockuppage.

pinMode(53, OUTPUT); // SS on Mega

#ifdef DEBUG

Serial.println("-----------------");

Serial.println("VC0706 Camera snapshot test");

#endif

// see if the card is present and can be initialized:

if (!SD.begin(chipSelect)) {

#ifdef DEBUG

Serial.println("Card failed, or not present");

#endif

// don't do anything more:

return;

}

// Try to locate the camera

if (cam.begin()) {

#ifdef DEBUG

Serial.println("Camera Found:");

#endif

} else {

#ifdef DEBUG

Serial.println("No camera found?");

#endif

return;

}

#ifdef DEBUG

// Print out the camera version information (optional)

char \*reply = cam.getVersion();

if (reply == 0) {

Serial.print("Failed to get version");

} else {

Serial.print(reply);

}

#endif

// Set the picture size - you can choose one of 640x480, 320x240 or 160x120

// Remember that bigger pictures take longer to transmit!

//cam.setImageSize(VC0706\_640x480); // biggest

cam.setImageSize(VC0706\_320x240); // medium

//cam.setImageSize(VC0706\_160x120); // small

#ifdef DEBUG

// You can read the size back from the camera (optional, but maybe useful?)

uint8\_t imgsize = cam.getImageSize();

Serial.print("Image size: ");

if (imgsize == VC0706\_640x480) Serial.println("640x480");

if (imgsize == VC0706\_320x240) Serial.println("320x240");

if (imgsize == VC0706\_160x120) Serial.println("160x120");

#endif

}

void initMPU() {

Wire.begin();

TWBR = 24; // set 400kHz mode @ 16MHz CPU or 200kHz mode @ 8MHz CPU

// initialize device

#ifdef DEBUG

Serial.println(F("Initializing I2C devices..."));

#endif

mpu.initialize();

#ifdef DEBUG

// verify connection

Serial.println(F("Testing device connections..."));

Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") : F("MPU6050 connection failed"));

#endif

// load and configure the DMP

#ifdef DEBUG

Serial.println(F("Initializing DMP..."));

#endif

devStatus = mpu.dmpInitialize();

// supply your own gyro offsets here, scaled for min sensitivity

//386 86 1864 -31 -63 -7

mpu.setXGyroOffset(-31);

mpu.setYGyroOffset(-63);

mpu.setZGyroOffset(-7);

mpu.setXAccelOffset(386);

mpu.setYAccelOffset(86);

mpu.setZAccelOffset(1864);

// make sure it worked (returns 0 if so)

if (devStatus == 0) {

// turn on the DMP, now that it's ready

#ifdef DEBUG

Serial.println(F("Enabling DMP..."));

#endif

mpu.setDMPEnabled(true);

// enable Arduino interrupt detection

#ifdef DEBUG

Serial.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));

#endif

attachInterrupt(0, dmpDataReady, RISING);

mpuIntStatus = mpu.getIntStatus();

// set our DMP Ready flag so the main loop() function knows it's okay to use it

#ifdef DEBUG

Serial.println(F("DMP ready! Waiting for first interrupt..."));

#endif

dmpReady = true;

// Set the full scale range of the gyro

uint8\_t FS\_SEL = 0;

//mpu.setFullScaleGyroRange(FS\_SEL);

// get default full scale value of gyro - may have changed from default

// function call returns values between 0 and 3

uint8\_t READ\_FS\_SEL = mpu.getFullScaleGyroRange();

Serial.print("FS\_SEL = ");

Serial.println(READ\_FS\_SEL);

GYRO\_FACTOR = 131.0/(FS\_SEL + 1);

// get default full scale value of accelerometer - may not be default value.

// Accelerometer scale factor doesn't reall matter as it divides out

uint8\_t READ\_AFS\_SEL = mpu.getFullScaleAccelRange();

Serial.print("AFS\_SEL = ");

Serial.println(READ\_AFS\_SEL);

//ACCEL\_FACTOR = 16384.0/(AFS\_SEL + 1);

// Set the full scale range of the accelerometer

//uint8\_t AFS\_SEL = 0;

//mpu.setFullScaleAccelRange(AFS\_SEL);

// get expected DMP packet size for later comparison

packetSize = mpu.dmpGetFIFOPacketSize();

} else {

// ERROR!

// 1 = initial memory load failed

// 2 = DMP configuration updates failed

// (if it's going to break, usually the code will be 1)

#ifdef DEBUG

Serial.print(F("DMP Initialization failed (code "));

Serial.print(devStatus);

Serial.println(F(")"));

#endif

}

#ifdef DEBUG

Serial.println("-----------------");

#endif

}

void initRegulators(){

pitchReg.SetMode(AUTOMATIC);

pitchReg.SetOutputLimits(-PID\_PITCH\_INFLUENCE, PID\_PITCH\_INFLUENCE);

rollReg.SetMode(AUTOMATIC);

rollReg.SetOutputLimits(-PID\_ROLL\_INFLUENCE, PID\_ROLL\_INFLUENCE);

yawReg.SetMode(AUTOMATIC);

yawReg.SetOutputLimits(-PID\_YAW\_INFLUENCE, PID\_YAW\_INFLUENCE);

}

void loop()

{

// local copy of variables on first function call to hold local copies

static uint8\_t bUpdateFlags;

// if programming failed, don't try to do anything

if (!dmpReady) return;

// wait for MPU interrupt or extra packet(s) available

while (!mpuInterrupt && fifoCount < packetSize) {}

// check shared update flags to see if any channels have a new signal

if(bUpdateFlagsShared) {

noInterrupts(); // turn interrupts off quickly while we take local copies of the shared variables

// take a local copy of which channels were updated

bUpdateFlags = bUpdateFlagsShared;

if(bUpdateFlags & CH1\_FLAG)

{

// unCh1In = unCh1InShared; // roll

unCh1In = map(unCh1InShared, ESC\_MIN, ESC\_MAX, ROLL\_MIN, ROLL\_MAX);

}

if(bUpdateFlags & CH2\_FLAG)

{

//unCh2In = unCh2InShared; // pitch

unCh2In = map(unCh2InShared, ESC\_MIN, ESC\_MAX, PITCH\_MIN, PITCH\_MAX);

}

if(bUpdateFlags & CH3\_FLAG)

{

unCh3In = unCh3InShared; // velocity

}

if(bUpdateFlags & CH4\_FLAG)

{

// unCh4In = unCh4InShared; // yaw

unCh4In = map(unCh4InShared, ESC\_MIN, ESC\_MAX, YAW\_MIN, YAW\_MAX);

}

// clear shared copy of updated flags as we have already taken the updates

bUpdateFlagsShared = 0;

interrupts(); // we have local copies of the inputs, so now we can turn interrupts back on

// as soon as interrupts are back on, we can no longer use the shared copies, the interrupt

// service routines own these and could update them at any time. During the update, the

// shared copies may contain junk. Luckily we have our local copies to work with

}

// do any processing from here onwards

// only use the local values unCh1In, unCh2In, unCh3In, unCh4In

// the shared variables unCh1InShared, unCh2InShared, unCh3InShared, unCh4InShared are

// always owned by the interrupt routines and should not be used in loop

if((unCh1In < ESC\_MIN) || (unCh1In > ESC\_MAX)) unCh1In = unCh1InLast;

if((unCh2In < ESC\_MIN) || (unCh2In > ESC\_MAX)) unCh2In = unCh2InLast;

if((unCh3In < ESC\_MIN) || (unCh3In > ESC\_MAX)) unCh3In = unCh3InLast;

if((unCh4In < ESC\_MIN) || (unCh4In > ESC\_MAX)) unCh4In = unCh4InLast;

unCh1InLast = unCh1In;

unCh2InLast = unCh2In;

unCh3InLast = unCh3In;

unCh4InLast = unCh4In;

#ifdef OUTPUT\_RECEIVER\_VALUES

Serial.print("ch1, ch2, ch3, ch4");

Serial.print("\t\t");

Serial.print(unCh1In);

Serial.print("\t\t");

Serial.print(unCh2In);

Serial.print("\t\t");

Serial.print(unCh3In);

Serial.print("\t\t");

Serial.println(unCh4In);

#endif

// Keep calculating the values of the complementary filter angles for comparison with DMP here

// Read the raw accel/gyro values from the MPU-6050

mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

// Get time of last raw data read

t\_now = millis();

// Remove offsets and scale gyro data

gyro\_x = (gx - base\_x\_gyro)/GYRO\_FACTOR;

gyro\_y = (gy - base\_y\_gyro)/GYRO\_FACTOR;

gyro\_z = (gz - base\_z\_gyro)/GYRO\_FACTOR;

accel\_x = ax; // - base\_x\_accel;

accel\_y = ay; // - base\_y\_accel;

accel\_z = az; // - base\_z\_accel;

accel\_angle\_x = atan(-accel\_y/sqrt(pow(accel\_x,2) + pow(accel\_z,2)))\*RADIANS\_TO\_DEGREES;

accel\_angle\_y = atan(accel\_x/sqrt(pow(accel\_y,2) + pow(accel\_z,2)))\*RADIANS\_TO\_DEGREES;

accel\_angle\_z = 0;

// Compute the (filtered) gyro angles

dt = (t\_now - get\_last\_time())/1000.0;

gyro\_angle\_x = gyro\_x\*dt + get\_last\_x\_angle();

gyro\_angle\_y = gyro\_y\*dt + get\_last\_y\_angle();

gyro\_angle\_z = gyro\_z\*dt + get\_last\_z\_angle();

// Compute the drifting gyro angles

unfiltered\_gyro\_angle\_x = gyro\_x\*dt + get\_last\_gyro\_x\_angle();

unfiltered\_gyro\_angle\_y = gyro\_y\*dt + get\_last\_gyro\_y\_angle();

unfiltered\_gyro\_angle\_z = gyro\_z\*dt + get\_last\_gyro\_z\_angle();

// Apply the complementary filter to figure out the change in angle

const float alpha = 0.98;

angle\_x = alpha\*gyro\_angle\_x + (1.0 - alpha)\*accel\_angle\_x;

angle\_y = alpha\*gyro\_angle\_y + (1.0 - alpha)\*accel\_angle\_y;

// angle\_z = gyro\_angle\_z; //Accelerometer doesn't give z-angle

// Update the saved data with the latest values

set\_last\_read\_angle\_data(t\_now, angle\_x, angle\_y, angle\_z, unfiltered\_gyro\_angle\_x, unfiltered\_gyro\_angle\_y, unfiltered\_gyro\_angle\_z);

#ifdef OUTPUT\_READABLE\_COMPLIMENTARY

Serial.print("CMP:\t");

Serial.print(get\_last\_z\_angle(), 2);

Serial.print("\t:\t");

Serial.print(get\_last\_y\_angle(), 2);

Serial.print("\t:\t");

Serial.println(get\_last\_x\_angle(), 2);

#endif

// reset interrupt flag and get INT\_STATUS byte

mpuInterrupt = false;

mpuIntStatus = mpu.getIntStatus();

// get current FIFO count

fifoCount = mpu.getFIFOCount();

// check for overflow (this should never happen unless our code is too inefficient)

if ((mpuIntStatus & 0x10) || fifoCount == 1024) {

// reset so we can continue cleanly

mpu.resetFIFO();

#ifdef DEBUG

Serial.println(F("MPU FIFO overflow!"));

#endif

// otherwise, check for DMP data ready interrupt (this should happen frequently)

} else if (mpuIntStatus & 0x02) {

// wait for correct available data length, should be a VERY short wait

while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

// read a packet from FIFO

mpu.getFIFOBytes(fifoBuffer, packetSize);

// track FIFO count here in case there is > 1 packet available

// (this lets us immediately read more without waiting for an interrupt)

fifoCount -= packetSize;

// display Euler angles in degrees

mpu.dmpGetQuaternion(&q, fifoBuffer);

mpu.dmpGetGravity(&gravity, &q);

mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

}

rollReg.Compute();

pitchReg.Compute();

yawReg.Compute();

#ifdef OUTPUT\_BAL\_AC\_BD

Serial.print("ac, bd");

Serial.print("\t\t");

Serial.print(bal\_ac);

Serial.print("\t\t");

Serial.println(bal\_bd);

#endif

v\_ac = (abs(-100+bal\_axes)/100)\*unCh3In;

v\_bd = ((100+bal\_axes)/100)\*unCh3In;

va = abs(((-100+bal\_ac)/100)\*v\_ac);

vb = abs(((-100+bal\_bd)/100)\*v\_bd);

vc = ((100+bal\_ac)/100)\*v\_ac;

vd = ((100+bal\_bd)/100)\*v\_bd;

if(unCh3In < ESC\_ARM){

va = ESC\_MIN;

vb = ESC\_MIN;

vc = ESC\_MIN;

vd = ESC\_MIN;

}

if(va < ESC\_MIN) va = ESC\_ARM;

if(va > ESC\_MAX) va = ESC\_MAX;

if(vb < ESC\_MIN) vb = ESC\_ARM;

if(vb > ESC\_MAX) vb = ESC\_MAX;

if(vc < ESC\_MIN) vc = ESC\_ARM;

if(vc > ESC\_MAX) vc = ESC\_MAX;

if(vd < ESC\_MIN) vd = ESC\_ARM;

if(vd > ESC\_MAX) vd = ESC\_MAX;

servoMotorA.write(va);

servoMotorB.write(vb);

servoMotorC.write(vc);

servoMotorD.write(vd);

#ifdef OUTPUT\_MOTOR\_VARIABLES

Serial.print("v, va, vb, vc, vd");

Serial.print("\t");

Serial.print(unCh3In);

Serial.print("\t");

Serial.print(va);

Serial.print("\t");

Serial.print(vb);

Serial.print("\t");

Serial.print(vc);

Serial.print("\t");

Serial.println(vd);

#endif

timedAction.check();

bUpdateFlags = 0;

}

void takePicture(){

#ifdef DEBUG

Serial.println("Snap in 3 secs...");

#endif

if (! cam.takePicture()) {

#ifdef DEBUG

Serial.println("Failed to snap!");

#endif

} else {

#ifdef DEBUG

Serial.println("Picture taken!");

#endif

}

// Create an image with the name IMAGExx.JPG

char filename[13];

strcpy(filename, "IMAGE00.JPG");

for (int i = 0; i < 100; i++) {

filename[5] = '0' + i/10;

filename[6] = '0' + i%10;

// create if does not exist, do not open existing, write, sync after write

if (! SD.exists(filename)) {

break;

}

}

// Open the file for writing

File imgFile = SD.open(filename, FILE\_WRITE);

// Get the size of the image (frame) taken

uint16\_t jpglen = cam.frameLength();

#ifdef DEBUG

Serial.print("Storing ");

Serial.print(jpglen, DEC);

Serial.print(" byte image.");

#endif

int32\_t time = millis();

pinMode(8, OUTPUT);

// Read all the data up to # bytes!

while (jpglen > 0) {

// read 32 bytes at a time;

uint8\_t \*buffer;

uint8\_t bytesToRead = min(64, jpglen); // change 32 to 64 for a speedup but may not work with all setups!

buffer = cam.readPicture(bytesToRead);

imgFile.write(buffer, bytesToRead);

#ifdef DEBUG

byte wCount = 0; // For counting # of writes

if(++wCount >= 64) { // Every 2K, give a little feedback so it doesn't appear locked up

Serial.print('.');

wCount = 0;

}

Serial.print("Read "); Serial.print(bytesToRead, DEC); Serial.println(" bytes");

#endif

jpglen -= bytesToRead;

}

imgFile.close();

#ifdef DEBUG

time = millis() - time;

Serial.println("done!");

Serial.print(time); Serial.println(" ms elapsed");

Serial.println("-----------------");

#endif

}

// simple interrupt service routine

void calcCh1(){

// if the pin is high, its a rising edge of the signal pulse, so lets record its value

if(digitalRead(CH1\_IN\_PIN) == HIGH)

{

ulCh1Start = micros();

}

else

{

// else it must be a falling edge, so lets get the time and subtract the time of the rising edge

// this gives use the time between the rising and falling edges i.e. the pulse duration.

unCh1InShared = (uint16\_t)(micros() - ulCh1Start);

// use set flag to indicate that a new signal has been received

bUpdateFlagsShared |= CH1\_FLAG;

}

}

void calcCh2(){

if(digitalRead(CH2\_IN\_PIN) == HIGH)

{

ulCh2Start = micros();

}

else

{

unCh2InShared = (uint16\_t)(micros() - ulCh2Start);

bUpdateFlagsShared |= CH2\_FLAG;

}

}

void calcCh3(){

if(digitalRead(CH3\_IN\_PIN) == HIGH)

{

ulCh3Start = micros();

}

else

{

unCh3InShared = (uint16\_t)(micros() - ulCh3Start);

bUpdateFlagsShared |= CH3\_FLAG;

}

}

void calcCh4(){

if(digitalRead(CH4\_IN\_PIN) == HIGH)

{

ulCh4Start = micros();

}

else

{

unCh4InShared = (uint16\_t)(micros() - ulCh4Start);

bUpdateFlagsShared |= CH4\_FLAG;

}

}