Stable Hovering for Quadcopter Using PID Algorithm

Final Project Report

December 9, 2016
E155

Tina Zhu and Josephine Wong

Abstract:

Many RC hobbyists enjoy flying quadcopters, or small RC helicopter with four motors and propellers, recreationally because they are easy to stabilize and maneuver. Users usually only control pitch, roll and yaw of the quadcopter to control its movement, and between user inputs the quadcopter should be able to hover safely by keeping angular positions and velocity zero. This project prototypes a quadcopter which can perform that sort of state control to hover at a certain height, but be controlled over SSH instead of radio control. To do so, an 9DOF IMU and ultrasound sensor was used to collect sensor values, data was processed to eliminate drift, and the correct motor values were sent to a module to control motor velocity.

Introduction

The motivation of our project was to see if we could tackle the design challenges of collecting and processing sensor data correctly to create a stable, hovering quadcopter.

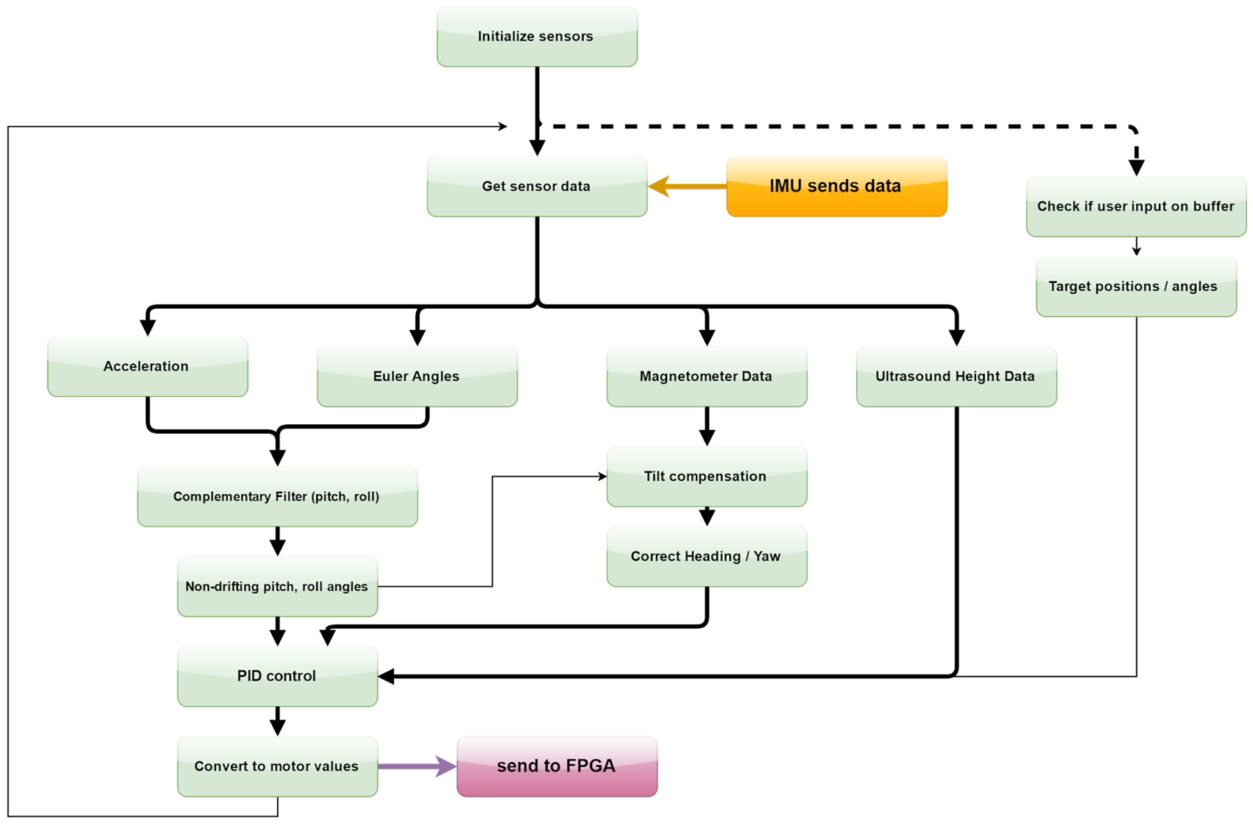

Figure 1: Block diagram of logic on the Pi, describes steps in data processing and how the Pi is interfaced with the FPGA and IMU

As shown by the above diagram, our system could be separated into three main parts: sensor data collection, drift compensation and PID control, and sending that output to the motor speed controllers. The first was handled by the IMU which sent data to the Pi over I2C, the second was handled by the Pi, and the third was handled by the FPGA which was given values over SPI.

New Hardware

The new hardware in this project included electronic speed controllers or ESCs, one for each of our 4 brushless motors. The ESCs accepted a simple 50Hz signal with a pulse from on time of 1-2ms, with 1ms representing 0% throttle and 2ms representing 100% throttle.

Another new hardware in this project was the Adafruit 9DOF IMU, which sent accelerometer, gyroscope and magnetometer data through I2C. It contained a LSM303 module that measured acceleration and magnetic field and a L3GD20 module that measured angular velocity. The IMU is powered by 5V, and since the maximum current draw of the modules is 6.1mA, it can be powered from one of the Raspberry Pi's 5V pins. Module datasheets are [3] (LSM303) and [4] (L3GD20).

Our second sensor was the HR-SR04 Ranging Detector, an ultrasound sensor that can measure height. The sensor has a trigger and an echo pin. When the Pi sends a high signal to the trigger, the detector emits a pulse. It waits for the pulse to come back before setting the echo pin high. The elapsed time can be determined and converted to height. The sensor's maximum sampling period is 60ms and measuring range is from 2 centimeters to 4 meters. The sensor runs on 5V and can be powered from the Pi since it only draws 15mA. The ultrasound detector's datasheet is [5].

Mechanical hardware used in this project included the quadcopter frame, four propeller blades, four 1000kV brushless motors, and a 4500mAh 3S lithium-polymer battery.

Schematics

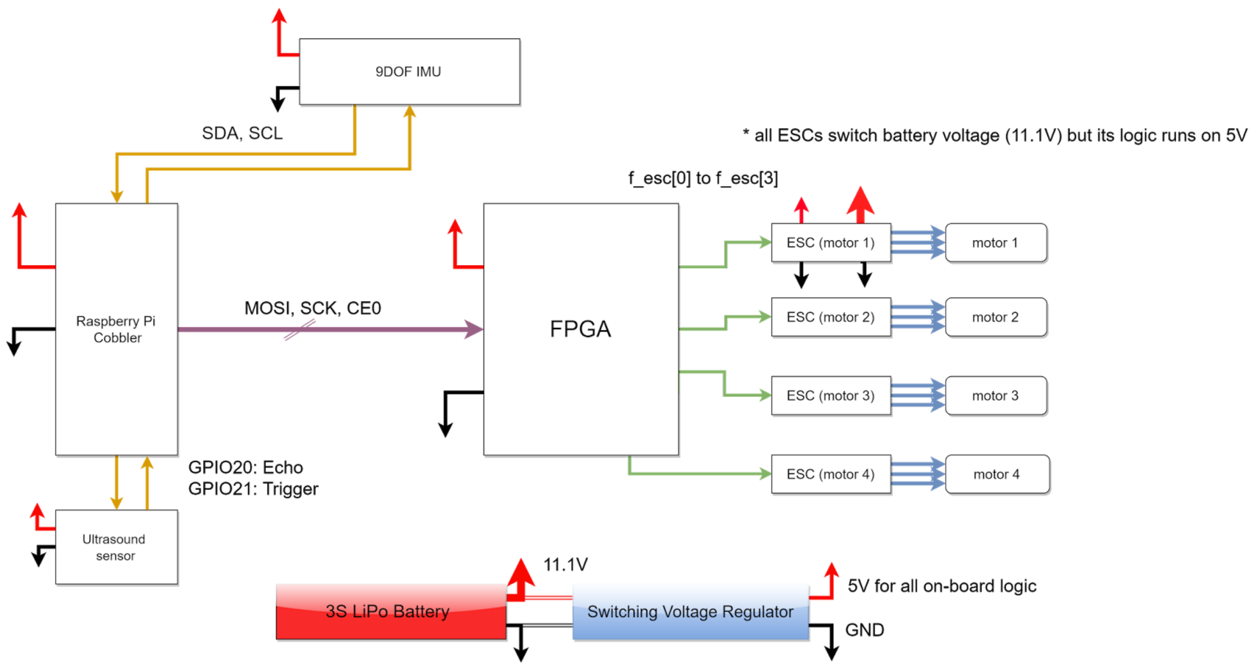

Figure 2: schematic of electronics on quadcopter

As formerly described, the Pi was interfaced with the 9DOF IMU over I2C and the FPGA with SPI. The ultrasound sensor trigger and echo signals were also connected to the Pi via GPIO pins. The Pi, IMU, ultrasound sensor, and ESCs were all powered by a 5V voltage regulator, which was powered on by the 3S, 11.1V lipo battery.

Microcontroller Design

The microcontroller takes sensor data from the IMU and ultrasound detector to determine its current height and angular position. Then, it performs a PID algorithm that compares the current state with the desired state and calculates the necessary motor speeds, encoded as values between 0 and 255, to get the quadcopter closer to the desired state. It sends these motor numbers to the FPGA via SPI.

The major modules include a BASIC.h library that provides access to the Pi registers and contains functions to perform SPI and I2C communication; sensor.h library that interfaces with the sensors to collect data; esc_output.h library that performs SPI to send motor numbers to the FPGA; and the main function, pid.c, that uses these libraries in a PID algorithm to control the quadcopter.

BASIC.h

Most of this library was already developed in previous labs. The major addition for this project was the I2C protocol. One I2C master, BSC1, was added to the memory map and four functions were written to allow for register manipulation.

The i2cInit method initialized I2C communication. It set pins 2 and 3, the pins corresponding to BSC1, to ALT0 and the clock frequency to a provided value.

The i2cSlaveAdr method set the slave address.

The i2cClearBits method cleared FIFO and reset the TRANSFER_DONE, ERR-ACK, and CLKT bits in the STATUS register. This method was called before every IMU read.

The i2cPowerOn method was created specifically to enable the three slaves of the Adafruit IMU. It populates FIFO with the slave's CTRL register address and the appropriate bits for the register to power on the slave. Then, it writes out those values to set the CTRL register.

The i2cRead method reads data from each slave. First, the method writes out the slave's sub-address. After checking that the transfer is active, it sets the data length (the number of bytes to read) and starts reading. After the read is done, it pulls the data from FIFO and stores them in the provided array.

An important reference when writing these functions was the source code for the I2C part of the bcm2835 library [1].

sensor.h

This library performs all sensor data collection. It has a initialization function that turns on the accelerometer, gyroscope, and magnetometer slaves on the IMU. These devices are initially powered off and their CTRL registers must be set to the appropriate 8-bit values to turn them on.

The library contains a getHeight method that sends a high signal to the ultrasound's trigger pin and tracks how long it takes for the echo pin to read high. It multiplies the elapsed time by the speed of sound and divides by two to determine the distance between the sensor and ground in meters.

The getIMU method calls i2cRead from BASIC.h for each slave on the IMU. Each read collects 6 bits of data, a most-significant and a least-significant byte for each dimension (x, y, and z). The function converts the 16-bit values to acceleration, angular velocity, and magnetic field strength values based the sensors' resolution. Note: by default, the output for each accelerometer axis is buffered with 0000 at the least-significant end [2].

Finally, the getData method calls getHeight and getIMU and collects the data in a Sensor structure that it sends to the calling function.

<u>esc_output.h</u>

This library handles SPI communication from the Pi to the FPGA. Its initialization function sets the SPI clock frequency and assigns pin modes. It has a writeOut method that takes four motor numbers from 0-255, one for each ESC, and calls spiSendReceive from BASIC.h to send those values to the FPGA.

<u>complementary.h</u>

This header file contains the complementary filter code, which eliminated drift from our pitch and roll data. A more serious method to reduce drift would be to use a Kalman filter, which may have better performance than a complementary filter but much too complicated and tedious to implement. This was implemented by obtaining pitch and roll from both the accelerometer and the gyroscope, and this code merged the two values together. This could be done by simply adding both with their previous values and merging the values together with appropriate weights.

Without doing so, due to integrated error and sensor drift, the pitch and roll would quickly climb to extraordinarily high values in a minute of run time.

We also included a function here, tilt_compensation, which compensated for pitch and roll because otherwise, finding the yaw from the magnetometer would require assuming the quadcopter was always parallel to the ground, which may not always be the case.

The complementary filter was tested by running five minutes of the main loop, printing the sensor values, and leaving the quadcopter on a stable surface. The pitch and roll stayed around zero and did not begin climbing up like before.

pid.c

This is our main function which, using the header files that were included, collects data from the sensors, collects user input, processes data with the complementary filter, executes a PID algorithm to control the quadcopter, and sends data out to the motors.

Wifi and Static IP Address

To allow for remote communication, the Pi had a Wifi adapter and a static IP address. To enable wifi connection, network credentials were entered into the Pi’s /etc/wpasupplicant/wpasupplicant.conf [6]. To set the static ip address, after experimentation with a number of different approaches, we used a Wifi connection program called wpa_gui to set the desired IP address.

FPGA Design

The FPGA hardware's purpose was to receive 4 8-bit numbers, corresponding to motor throttle, by SPI and then create 4 ESC signals from them. The design of the SPI was identical to the one used in class, but slightly revised to be unidirectional and only to expect 32 bits at a time.

After the 4 8-bit numbers were obtained, they were saved onto a register to be used by the ESC signal module, which was somewhat similar to a regular PWM module. It consisted of a counter with a reset which went high whenever count time exceeded the period, and its output was from a comparator comparing the count time to the on time. The period count had to be equal to clock * 20 milliseconds, and for maximum resolution, the on time count was the 8-bit motor number, which varied the on time of the ESC signal.

Then, each increment of 1 corresponded to 1/256 of the maximum motor throttle. Since ESC signal's on time varied from 1ms for no throttle and 2ms for maximum throttle, each increment stood for 1/256 of that 1ms millisecond on time. Because there were 256 increments per 1ms, a 256kHz clock was created from the system clock with a simple counter and was used to clock the PWM module, so that a full 8-bit count from 0 to 255 would take a millisecond. The full period then was 256kHz * 20ms, or 5120 counts.

Even in the final revision, in the main module two pins were used for debugging the hardware. One was the slow 256kHz clock and the other was the SPI module reset, which went high whenever 32 bits were received.

Results

Our quadcopter was able to spin its motors and, unfortunately not during hardware demo due to solder connection, lift off the ground. A test function was written that sent 120 to all motors. When this function was ran during the demo, three of the motors spun up. However, during testing before hardware demo, the function was able to spin all four motors. When the motor number was changed to 170, the quadcopter was able to lift off the ground. A faulty solder connection probably caused the fourth motor to fail to spin.

The Pi was able to collect sensor data. It printed reasonable acceleration, angular velocity, and magnetometer values from the IMU and height values from the ultrasound detector.

Also, all electronics fit on-board the quadcopter. The Pi and FPGA were powered using a non-linear 5V regulator that took 11.1V from the battery while the motors were powered directly with the battery. The Pi had a Wifi adapter that allowed for remote access to run its flight functions. The setup for a static IP address was mostly reliable but failed at times. If SSH with the assigned IP address did not work, the Pi was connected to a monitor to determine its current IP address.

A PID algorithm was written to control the quadcopter so that it could stably hover at a certain height. To test the algorithm, the quadcopter was tilted at different angles. The motor number outputs calculated by the algorithm were observed to change accordingly to correct for the tilt. This algorithm has not yet been tested in flight.

One of the most difficult parts of the design was implementing the I2C protocol since there were few sources that described how one would implement the protocol for the Pi. Most sources used existing libraries to perform I2C. Moreover, an understanding of I2C communication did not lend directly to implementation since some aspects of the communication process were implied. For example, the slave and master sent acknowledges automatically. These acknowledge bits did not need to be sent manually in code.

Another difficult part was constructing the quadcopter. Because all the electronics needed to be on-board, we ran into issues with space and finding ways to power everything from one battery. We experimented with making a protoboard and using different connectors. Ultimately, we settled with two breadboards that held the FGPA, Pi, and

IMU. Jumper wires connected the ESCs and ultrasound, which was strapped onto one arm, to male headers on the breadboards. This allowed for secure wiring connection and minimized space so that everything could fit on the quadcopter.

References

[1] McCauley, Mike. *I2C Access Source Code*. http://www.airspayce.com/mikem/bcm2835/group__i2c.html#gacb378490c2f6c295072f2da6eae2275a

[2] *LSM303DLHA Protocol Hints*. https://www.pololu.com/product/2124

[3] *LSM303DLHC Datasheet*. https://cdn-shop.adafruit.com/datasheets/LSM303DLHC.PDF

[4] *L3GD20 Datasheet*. https://cdn-shop.adafruit.com/datasheets/L3GD20.pdf

[5] *HC-SR04 Datasheet*. http://www.micropik.com/PDF/HCSR04.pdf

[6] Khan, Hamzah. *Connecting to Claremont-WPA over Wifi on Pi.* [email to class]

Parts List

Part	**Source**	**Vendor Part #**	**Price**
4 x 1045 10”x4.5” Carbon Fiber Propellers	Ebay	N/A	$21.98
4 x 10”x5.5” Propellers	Pegasus Hobbies	N/A	$12.00
4500mAh 3S Lipo Battery	Ebay	N/A	$24.99
4 x 980KV Brushless Motor and 30A Speed Controller Set	Ebay	N/A	$55.19
YoCoo 4-Axis Quadcopter Frame	Amazon	N/A	$15.99
9DOF IMU	Adafruit	1714	$19.95
HC-SR04 Ranging Detector	Amazon	N/A	$5.25

Appendices

```
//--------------------- FPGA motor controller  --------------------//
# quad_motor_control.sv
# Author: Tina Zhu
# Email: tzhu@g.hmc.edu
# Date:12/5/2016
# takes in 4 motor numbers over SPI, outputs 4 ESC signals

////////////////////////////////////////////
// Takes SPI, outputs motor control signals
// {f1, f2, f3, f4} = front - left - right - back motors
// FPGA clk is 40MHz
////////////////////////////////////////////
module quad_motor_control(input logic clk, input logic reset, input logic sck, input logic sdi, input logic cs,
                                                                              output logic[3:0] f_esc, output
logic slowclk, output done);
        //logic slowclk;
        //logic done;
        logic[7:0] f1_motor, f2_motor, f3_motor, f4_motor;
        logic[31:0] allf;

        // Divide 40MHz FPGA clock into 256kHz clock (divide by about 2^7)
        generate_acc_divclk#8 clk_div1(clk, slowclk);

        spi_interface spi1(reset, sck, sdi, cs, allf, done);

        register f1_motor_reg(clk, done, allf[31:24], f1_motor);
        register f2_motor_reg(clk, done, allf[23:16], f2_motor);
        register f3_motor_reg(clk, done, allf[15:8], f3_motor);
        register f4_motor_reg(clk, done, allf[7:0], f4_motor);

        ESCcontrol motor1(slowclk, cs, f1_motor, f_esc[0]);
        ESCcontrol motor2(slowclk, cs, f2_motor, f_esc[1]);
        ESCcontrol motor3(slowclk, cs, f3_motor, f_esc[2]);
        ESCcontrol motor4(slowclk, cs, f4_motor, f_esc[3]);
endmodule

////////////////////////////////////////////
// spi_interface
//   Half-duplex SPI interface. Shifts in 4 8-bit numbers,
//   Can only receive, can't send (hence no sdo)
////////////////////////////////////////////
module spi_interface(
                                    input logic reset,
                                    input  logic sck,
        input  logic sdi,
        input  logic cs,
        output logic [31:0] motor_number,
                                    output logic done);
```

```
  logic [5:0] count;
            logic should_shift;

            // then deassert load, wait until done
            // apply (4 * 8 bit = 32) sclks to shift in motor_number[0]
  always_ff @(posedge sck)
            begin
            if (should_shift)  motor_number[31:0] = {motor_number[30:0], sdi};
            end

            // Counter here to count number of motor numbers done
            counter#6 motor_num_counter(sck, cs|done, count);

            assign done = (count == 6'b100000); // = 32

            assign should_shift = (~cs) & (count < 6'b100000);
endmodule

module register#(parameter N = 8)(input logic clk, input logic en, input logic [N-1:0] din, output logic [N-1:0]
dout);
            always_ff@(posedge clk)
            begin
                        if(en) dout <= din;
            end
endmodule

module ESCcontrol#(parameter COUNT_20MS = 16'd10240, COUNT_1MS = 16'd512)(input logic clk, input logic
reset, input logic[7:0] motor_reg, output logic ESCout);
            // ESC neutral is at 1ms pulse, further counts bring up duty cycle

            logic reset_count;
            logic[15:0] count;
            //logic[12:0] t1, t2;
            // Use 256kHz clock to manage ESC counter, big enough to hold up to counts to 20ms
            counter#16 counter1(clk, reset_count|reset, count);

            // assign t1 = count + countoffset;
            // assign t2 = motor_reg + countoffset;
            assign reset_count = (count == COUNT_20MS);
            assign ESCout = (count < ((motor_reg<<1) + COUNT_1MS));
endmodule

//// Counter with neg-edge reset
//module nr_counter#(parameter N = 16)(input logic clk, reset, output logic[N-1:0] count);
//          always_ff@(negedge clk, negedge reset)
//          begin
//                      if(~reset && clk) count <= 0;
//                      else count <= count + 1;
//          end
//endmodule
```

```
module counter#(parameter N = 16)(input logic clk, reset, output logic[N-1:0] count);
        initial begin
                count <= 0;
        end
        always_ff@(posedge clk, posedge reset)
        begin
                if(reset) count <= 0;
                else count <= count + 1;
        end
endmodule

module en_counter#(parameter N = 16)(input logic clk, en, output logic[N-1:0] count);
        initial begin
                count <= 0;
        end
        always_ff@(posedge clk)
        begin
                if(en) count <= count + 1;
        end
endmodule

module nr_counter#(parameter N = 16)(input logic clk, output logic[N-1:0] count);
        initial begin
                count <= 0;
        end
        always_ff@(posedge clk)
        begin
                count <= count + 1;
        end
endmodule

// Slowed clock by a factor of 2^N
module generate_divclk#(parameter N = 4)(input logic clk, output logic slowclk);
        logic[N-1:0] count;
        nr_counter#(N) cnt(clk, count);
        assign slowclk = count[N-1];
endmodule

// Slowed clock by more accurate method
// 40MHz -> 256kHz means 156.25 ticks, half on half off 78ticks each
module generate_acc_divclk#(parameter N = 8)(input logic clk, output logic slowclk);
        logic[N-1:0] count;
        logic reset;
        counter#(N) cnt(clk, reset, count);
        assign reset = (count == 8'b1001_1100);
        assign slowclk = (count < 8'b0100_1110);
endmodule
```

```
//---------------------Testing Function for ESCs--------------------//
 # testbench.sv
# Author: Tina Zhu
# Email: tzhu@g.hmc.edu
# Date:12/5/2016
# simulate 4 test motor numbers sent over SPI
////////////////////////////////////////////
// testbench
//   Tests quad_motor_control
////////////////////////////////////////////

module testbench();
   logic clk, reset, sck, sdi, cs;
   logic [31:0] motor_number;
   logic [8:0] i;
   logic [3:0] f_esc;
   logic [3:0] test_nums;

   // device under test
   quad_motor_control dut(clk, reset, sck, sdi, cs, f_esc);

   // test case
   initial begin
           // Test case
      motor_number <= 32'hAAFF0077;
   end

   // generate clock and load signals
   initial
      forever begin
         clk = 1'b0; #1;
         clk = 1'b1; #1;
      end

   initial begin
          reset = 1'b0;
          #1;
          reset = 1'b1;
          #1;
          reset = 1'b0;
          i = 0;
          test_nums = 0;
          cs = 1'b1;
          sdi = 1'b0;
          sck = 1'b0;
   end

   // shift in test vectors, wait until done, and shift out result
   always @(posedge clk)
           begin
                              if (i<32)
                              begin
                                     #1; sdi = motor_number[31-i];
```

```
                        cs = 1'b0;
                        #1; sck = 1; #2; sck = 0;
                        i = i + 1;
                end
                else if (i == 32)
                begin
                        #1;
                        cs = 1'b1;
                        #3000000; // wait
                        reset = 1'b1;
                        #1;
                        reset = 1'b0;
                        #1;
                        test_nums = test_nums + 1;
                i = 0;
                end
                else if (test_nums >= 2)
                begin
                        $display("Testbench finished running.");
    $stop();
                end
  end

endmodule

//--------------------- PID controller --------------------//
# pid.c
# Author: Tina Zhu
# Email: tzhu@g.hmc.edu
# Date:12/5/2016
# Control loop, performs drift compensation, collect user input, PID control, prints out motor values 0-255

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <math.h>

#include "sensor.h"
#include "complementary.h"
#include "esc_output.h"
#include "BASIC.h"

using uint_8 = char;

typedef struct Errors {
        float angX;
        float angY;
        float angZ;
} Errors;

typedef struct Angles {
        float angX; // pitch
```

```
            float angY; // roll
            float angZ; // yaw
} Angles;

typedef struct Output{
            float w1;
            float w2;
            float w3;
            float w4;
} Output;

#define GX_TARGET      0.0f
#define GY_TARGET      0.0f
#define GZ_TARGET      0.0f

#define KP          3.0f
#define KI          5.5f
#define KD          4.0f

#define IXX        0.005f
#define IYY        0.005f
#define IZZ        0.02f

#define B   0.0000001f
#define K   0.000003f
#define L   0.15f  //quadcopter radius in m

#define MOTOR_KV    1110.0f

#define GRAVITY                              9.80665f
#define UPDATE_PERIOD                   0.100f
#define UPDATE_PERIOD_US      100000f
#define MASS                          1.18f
#define M_PI               3.14159265359

void main(void) {

            // For handling user input
            fd_set readfds;
    FD_ZERO(&readfds);

    struct timeval timeout;
    timeout.tv_sec = 0;
    timeout.tv_usec = 0;

    char user_input[50];

            // Declare Sensor struct
            Sensor sensor;
            // Initialize everything to zero
            Errors e = {0, 0, 0};
            // Declare angle structs to hold wx, wy, wz for theta(local) and omega(global)
            Angles theta, thetadot, thetaInt = {0, 0, 0};
                                              // Angles omega, omegadot = {.angX = 0, .angY = 0, .angZ = 0};
```

```
        // Declare Output struct
        Output output;
        // Numbers for motor output
        uint_8 motorNumbers[4] = {0, 0, 0, 0};
        // For checking delay in PID loop
        int micros = 0;

        // setup ESCs
        setup_ESCs();
        // setup sensor
        initializeSensor();

        while(1)
        {
                micros = getTime();

                sensor = getData();

                thetadot.angX = sensor.wx;
                thetadot.angY = sensor.wy;
                thetadot.angZ = sensor.wz;

                output = pid_controller(sensor, &thetadot, &theta, &thetaInt, e);
                send_motor_values(output, motorNumbers);

                // Check for user input only when there is extra time
                while((getTime() - micros) < UPDATE_PERIOD_US)
                {
                        FD_SET(STDIN_FILENO, &readfds);
                        if (select(1, &readfds, NULL, NULL, &timeout))
                        {
                                scanf("%s", user_input);
                        // printf("Message: %s\n", message);
                        }

                        if(user_input[0] == 'u')
                        {
                                // do something...
                        }
                        // printf("...\n");
                        // delayMicrosecs(10);
                }
        }

}

Output pid_controller(Sensor sensor, Angles *thetadot, Angles *theta, Angles *thetaInt, Errors e)
{
// Calculate total thrust
        float thrust = MASS * GRAVITY/ (K * cos(theta->angX * theta->angY));

// Prevent large thetaInt change / windup
        if(abs(thetaInt->angX) > 0.01 || abs(thetaInt->angY) > 0.01 || abs(thetaInt->angZ) > 0.01)
        {
                thetaInt->angX = 0;
                thetaInt->angY = 0;
                thetaInt->angZ = 0;
        }

// Calculate all errors
        e.angX = (KD * thetadot->angX) + (KP * theta->angX) - (KI * thetaInt->angX);
        e.angY = (KD * thetadot->angY) + (KP * theta->angY) - (KI * thetaInt->angY);
        e.angZ = (KD * thetadot->angZ) + (KP * theta->angZ) - (KI * thetaInt->angZ);

// Calculate all outputs (motor rpm -> esc voltage)
        Output outs = calc_outputs(e, thrust);

// Save past pitch and roll for use in filter later
        float preThetaX = theta->angX;
        float preThetaY = theta->angY;

// Update the integrals (angle, angleIntegrated)
        theta->angX = theta->angX + (UPDATE_PERIOD*thetadot->angX);
        theta->angY = theta->angY + (UPDATE_PERIOD*thetadot->angY);
        // theta.angZ = theta.angZ + (UPDATE_PERIOD*thetadot.angZ);
        complementary_filter(&(theta->angX), &(theta->angY), preThetaX, preThetaY, sensor.ax, sensor.ay,
sensor.az, UPDATE_PERIOD);
        theta.angZ = tilt_compensation(theta->angX, theta->angY, sensor.mx, sensor.my, sensor.mz);

        thetaInt->angX = thetaInt->angX + (UPDATE_PERIOD*theta->angX);
        thetaInt->angY = thetaInt->angY + (UPDATE_PERIOD*theta->angY);
        thetaInt->angZ = thetaInt->angZ + (UPDATE_PERIOD*theta->angZ);

        return outs;
}

Output calc_outputs(Errors e, float thrust)
{
        Output motors;
        motors.w1 = thrust/4 - (2*B*e.angX*IXX + e.angZ*IZZ*K*L)/(4*B*K*L);
        motors.w2 = thrust/4 - e.angZ*IZZ/(4*B) - (e.angY*IYY)/(2*K*L);
        motors.w3 = thrust/4 - (-2*B*e.angX*IXX + e.angZ*IZZ*K*L)/(4*B*K*L);
        motors.w4 = thrust/4 + e.angZ*IZZ/(4*B) + (e.angY*IYY)/(2*K*L);

        // These are angular velocities in rad/s, convert to voltage using kV (rpm/V)
        // Convert to throttle level (V/total V) a percentage
        motors.w1 = motors.w1 /(2.0*M_PI*60.0f) / MOTOR_KV / 11.1f;
        motors.w2 = motors.w2 /(2.0*M_PI*60.0f) / MOTOR_KV / 11.1f;
        motors.w3 = motors.w3 /(2.0*M_PI*60.0f) / MOTOR_KV / 11.1f;
        motors.w4 = motors.w4 /(2.0*M_PI*60.0f) / MOTOR_KV / 11.1f;

        return motors;
}

void send_motor_values(Output motors, uint_8* motor_numbers)
{
        motor_numbers[0] = (uint_8)(motors.w1*255); // cast floats to ints
        motor_numbers[1] = (uint_8)(motors.w2*255);
```

```
            motor_numbers[2] = (uint_8)(motors.w3*255);
            motor_numbers[3] = (uint_8)(motors.w4*255);

            writeOut(motornumbers);
}

float abs(float x)
{
            if(x < 0) { x = -x; }
            return x;
}


//--------------------- Drift compensation --------------------//
# complementary.c
# Author: Tina Zhu
# Email: tzhu@g.hmc.edu
# Date:12/5/2016
# Performs complementary filter to combine accelerometer data with gyroscope data, elimintates drift
# Also contains code to find yaw by compensating for pitch and roll, transform values to parallel plane to ground

#include <math.h>

void complementary_filter(float *thetaX, float *thetaY, float prevThetaX, float prevThetaY,
                                                        float x_accel, float y_accel, float z_accel, float dt);
float tilt_compensation(float thetaX, float thetaY, int mx, int my, int mz);

/*
   complementaryFilter() - Filters out drift for wx, wy
      according to http://www.olliw.eu/2013/imu-data-fusing/
   "The complementary filter fuses the accelerometer and integrated gyro data by passing the former
   through a 1st-order low pass and the latter through a 1st-order high pass filter and adding the outputs." (from
paper)
   equation 2.1: angle = 0.2*(prev_angle + accel_ang*dt) + 0.98*gyro_ang  */
void complementary_filter(float *thetaX, float *thetaY, float prevThetaX, float prevThetaY,
                                                        float x_accel, float y_accel, float z_accel, float dt)
{
   float gyro_angX = *thetaX;
            float gyro_angY = *thetaY;

   float accel_angX = atan(y_accel/(y_accel*y_accel + z_accel*z_accel));
   *thetaX = gyro_angX*0.98 + 0.02*(prevThetaX + accel_angX*dt);

   float accel_angY = atan(x_accel/(x_accel*x_accel + z_accel*z_accel));
   *thetaY = gyro_angY*0.98 + 0.2*(prevThetaY + accel_angY*dt);
}

/*
   tiltCompensation() - Finds the yaw
      according to http://franciscoraulortega.com/pubs/Algo3DFusionsMems.pdf page4
   A magnetometer works well when it is completely parallel to the Earth's surface.
   If not and the platform is tilted, we must transform the readings to a parallel plane first
   and then find the yaw.
   mx, my, mz - output of magnetometer   */
float tilt_compensation(float thetaX, float thetaY, int mx, int my, int mz)
{
   float x_Heading = mx*cos(thetaY) + my*sin(thetaY)*sin(thetaX) + mz*sin(thetaY)*cos(thetaX);
   float y_Heading = my*cos(thetaX) + mz*sin(thetaX);
   float yaw = atan(-y_Heading/x_Heading);
        return yaw;
}


//---------------------Testing Function for ESCs to begin hovering, test hardware --------------------//
# SPI_test.c
# Author: Tina Zhu
# Email: tzhu@g.hmc.edu
# Date:12/5/2016
# Test function for motors: sends incrementing motor numbers, up to 120, to all four ESCs to hover the quadcopter

#include "esc_output.h"

void main(void) {
        setup_ESCs();
        //reset_ESCs();
        char motorNumbers[4] = {0,0,0,0};
        writeOut(motorNumbers);

        for(int i =0; i<120; ++i)
        {
                for(int j=0; j<4; ++j)
                {
                        motorNumbers[j] = i;
                }
                        writeOut(motorNumbers);
                        delayMicrosecs(100000);
        }
              for(int k=0; k<4; ++k)
         {
               motorNumbers[k] = 0;
         }
        writeOut(motorNumbers);
}

//---------------------esc_output.h library--------------------//
// test.c
// Author: Tina Zhu
// Email: tzhu@g.hmc.edu
// Date: 12/5/2016
// Send 4 8-bit numbers over SPI
////////////////////////////////////////////////

// #includes
#include "BASIC.h"

// Constants
```

```
#define RESET_FPGA 25
#define SPI_SETTINGS  (1<<25) // LEN_LONG

// Function Prototypes
void setup_ESCs(void);
void reset_ESCs(void);
void writeOut(char* motor_numbers);

// Functions

//Initialize SPI communication
void setup_ESCs(void)
{
		char motor_numbers[4] = {0, 0, 0, 0};
		pioInit();
		spiInit(256000, 0);		// Initialize SPI and reset pin
		pinMode(RESET_FPGA, OUTPUT);
		writeOut(motor_numbers); // Set motor numbers to 0
}

// Test function: send all high, then all 0
void reset_ESCs(void)
{
		char motor_numbers[4] = {255, 255, 255, 255};
		char motor_numbers2[4] = {0, 0, 0, 0};

		writeOut(motor_numbers); // write 1st set of motor numbers
		delayMicrosecs(200000); // wait 2 seconds
		writeOut(motor_numbers2); // reset motor numbers to 0
}

// Write motor values to FPGA
void writeOut(char* motor_numbers)
{
  // Send pulse per write
  digitalWrite(RESET_FPGA, 0);
  digitalWrite(RESET_FPGA, 1);
  digitalWrite(RESET_FPGA, 0);

  // Send motor numbers to FPGA through SPI
  for(int i=0; i<4; ++i)
  {
		spiSendReceive(motor_numbers[i]);
  }
}

//---------------------sensor.h library--------------------//
# sensor.h
# By Josephine Wong
# 09 December 2016 jowong@g.hmc.edu
# library to collect IMU and ultrasound data

#ifndef _INCLUDE_SENSOR_
```

```
#define _INCLUDE_SENSOR_

#include <stdio.h>
#include "BASIC.h"

#define ACCRES	0.00981f
#define GYRORES		0.0001526f

// Sensor structure to store sensor data
typedef struct Sensor {
		float ax;
		float ay;
		float az;
		float wx;
		float wy;
		float wz;
		int mx;
		int my;
		int mz;
		float alt;
} Sensor;

void initializeSensor(void) {
		pioInit();

		// pin assignments for ultrasound
		pinMode(21, OUTPUT);
		pinMode(20, INPUT);

		// initialize i2c pins and clk frequency
		i2cInit(200000);

		// power on accelerometer
		i2cClearBits();
		i2cSlaveAdr(0b0011001);
		i2cPowerOn(0b00100000, 0b00110111);

		// power on gyroscope
		i2cClearBits();
		i2cSlaveAdr(0b1101011);
		i2cPowerOn(0b00100000, 0b00001111);

		// power on magnetometer
		i2cClearBits();
		i2cSlaveAdr(0b0011110);
		i2cPowerOn(0b00000010, 0b00000000);
}

// ULTRASOUND DATA METHODS

void getHeight(Sensor *sensor) {
		// Send high pulse to trigger
		digitalWrite(21, 1);
		delayMicrosecs(10);
		digitalWrite(21, 0);

		// Wait for echo to go high
		while(!digitalRead(20)){};

		// Find the elapsed time and height
		int time1 = getTime();
		while(digitalRead(20)){};
		int time2 = getTime();

		float range = (float) (time2 - time1) / 1000000 * 340 / 2; // in meters
		printf("range = %f\n", range);

		sensor->alt = range;
}

// IMU DATA METHODS

// Converts 2's complement 12-bit data to 32-bit data
int convert12to32(char mb, char lb) {
		int x = (mb >> 7 == 0) ? ((mb << 4) | (lb >> 4)) : (0xFFFFF000 | ((mb << 4) | (lb >> 4)));
		return x;
}

// Converts 2's complement 16-bit data to 32-bit data
int convert16to32(char mb, char lb) {
		int x = (mb >> 7 == 0) ? ((mb << 8) | lb) : (0xFFFF0000 | ((mb << 8) | lb));
		return x;
}

void getIMU(Sensor *sensor) {
		char accData[6] = {1,2,3,4,5,6};
		char gyroData[6] = {7,8,9,10,11,12};
		char magData[6] = {13,14,15,16,17};

		// Read from accelerometer
		i2cClearBits();
		i2cSlaveAdr(0b0011001);
		while(!i2cRead(0b10101000, accData, 6)){};

		// Read from gyroscope
		printf("get\n");
		i2cClearBits();
		i2cSlaveAdr(0b1101011);
		while(!i2cRead(0b10101000, gyroData, 6)){};

		// Read from magnetometer
		i2cClearBits();
		i2cSlaveAdr(0b0011110);
		while(!i2cRead(0b00000011, magData, 6)){};

		// Convert to 32-bit 2's complement values
		int rawax = convert12to32(accData[1], accData[0]);
		int raway = convert12to32(accData[3], accData[2]);
```

```
        int rawaz = convert12to32(accData[5], accData[4]);
        int rawwx = convert16to32(gyroData[1], gyroData[0]);
        int rawwy = convert16to32(gyroData[3], gyroData[2]);
        int rawwz = convert16to32(gyroData[5], gyroData[4]);
        int rawmx = convert16to32(magData[0], magData[1]);
        int rawmy = convert16to32(magData[4], magData[5]);
        int rawmz = convert16to32(magData[2], magData[3]);

        // Convert to appropriate units using sensor resolution
        sensor->ax = rawax * ACCRES;
        sensor->ay = raway * ACCRES;
        sensor->az = rawaz * ACCRES;
        sensor->wx = rawwx * GYRORES;
        sensor->wy = rawwy * GYRORES;
        sensor->wz = rawwz * GYRORES;
        sensor->mx = rawmx;
        sensor->my = rawmy;
        sensor->mz = rawmz;

        // Remove print statements later
        /*printf("accData = %d, %d, %d\n", ((accData[1] << 2) | accData[0] >> 6),
                ((accData[3] << 2) | accData[2] >> 6),
                ((accData[5] << 2) | accData[4] >> 6));
        printf("gyroData = %d, %d, %d\n", ((gyroData[1] << 2) | gyroData[0] >> 6),
                ((gyroData[3] << 2) | gyroData[2] >> 6),
                ((gyroData[5] << 2) | gyroData[4] >> 6));
        printf("accData = %d, %d, %d, %d, %d, %d\n", accData[0], accData[1],
                accData[2], accData[3], accData[4], accData[5]);
        printf("gyroData = %d, %d, %d, %d, %d, %d\n", gyroData[0], gyroData[1],
                gyroData[2], gyroData[3], gyroData[4], gyroData[5]);*/
}

// Get IMU and height data
Sensor getData(void) {
        Sensor sensor;

        getIMU(&sensor);
        getHeight(&sensor);

        return sensor;
}
#endif

//---------------------BASIC.h library-------------------//
 // basic.h
// By Josephine Wong
// 06 October 2016 jowong@g.hmc.edu
// library for basic GPIO, system timer, and SPI operations

#ifndef _INCLUDE_BASIC_
#define _INCLUDE_BASIC_

#include <sys/mman.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

/////////////////////////////////////////////////////////////////////
// Constants
/////////////////////////////////////////////////////////////////////

//GPIO FSEL Types
#define INPUT    0
#define OUTPUT1
#define ALT0      4
#define ALT1      5
#define ALT2      6
#define ALT3      7
#define ALT4      3
#define ALT5      2

#define GPFSEL   ((volatile unsigned int *) (gpio + 0))
#define GPSET    ((volatile unsigned int *) (gpio + 7))
#define GPCLR    ((volatile unsigned int *) (gpio + 10))
#define GPLEV    ((volatile unsigned int *) (gpio + 13))
#define SPICS      ((volatile unsigned int *) (spi + 0))
#define SPIFIFO  ((volatile unsigned int *) (spi + 1))
#define SPICLK   ((volatile unsigned int *) (spi + 2))
#define I2CCTRL              ((volatile unsigned int *) (i2c + 0))
#define I2CSTAT              ((volatile unsigned int *) (i2c + 1))
#define I2CDLEN              ((volatile unsigned int *) (i2c + 2))
#define I2CA        ((volatile unsigned int *) (i2c + 3))
#define I2CFIFO  ((volatile unsigned int *) (i2c + 4))
#define I2CCLK   ((volatile unsigned int *) (i2c + 5))

// Physical addresses
#define BCM2836_PERI_BASE        0x3F000000
#define GPIO_BASE              (BCM2836_PERI_BASE + 0x200000)
#define TIME_BASE                     (BCM2836_PERI_BASE + 0x3000)
#define SPI_BASE                      (BCM2836_PERI_BASE + 0x204000)
#define I2C_BASE                      (BCM2836_PERI_BASE + 0x804000)
#define BLOCK_SIZE (4*1024)
#define TIME_BLOCK_SIZE        34
#define SPI_BLOCK_SIZE 24
#define I2C_BLOCK_SIZE 24

// Pointers that will be memory mapped when pioInit() is called
volatile unsigned int *gpio; //pointer to base of gpio
volatile unsigned int *sys_timer; //pointer to base of system timer
volatile unsigned int *spi; // pointer to base of spi
volatile unsigned int *i2c; // pointer to base of i2c

void pioInit() {
            int  mem_fd;
            void *reg_map;
```

```
        // /dev/mem is a psuedo-driver for accessing memory in the Linux filesystem
        if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
            printf("can't open /dev/mem \n");
            exit(-1);
        }

        reg_map = mmap(
          NULL,             //Address at which to start local mapping (null means don't-care)
  BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  GPIO_BASE);       // Offset to GPIO peripheral

        if (reg_map == MAP_FAILED) {
  printf("gpio mmap error %d\n", (int)reg_map);
  close(mem_fd);
  exit(-1);
 }

        gpio = (volatile unsigned *)reg_map;

        reg_map = mmap(
          NULL,             //Address at which to start local mapping (null means don't-care)
  TIME_BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  TIME_BASE);       // Offset to system timer peripheral

        if (reg_map == MAP_FAILED) {
  printf("timer mmap error %d\n", (int)reg_map);
  close(mem_fd);
  exit(-1);
 }

        sys_timer = (volatile unsigned *)reg_map;

        reg_map = mmap(
          NULL,             //Address at which to start local mapping (null means don't-care)
  SPI_BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  SPI_BASE);       // Offset to SPI peripheral

        if (reg_map == MAP_FAILED) {
  printf("spi mmap error %d\n", (int)reg_map);
  close(mem_fd);
  exit(-1);
 }

        spi = (volatile unsigned *)reg_map;
		reg_map = mmap(
		  NULL,			//Address at which to start local mapping (null means don't-care)
	I2C_BLOCK_SIZE,		//Size of mapped memory block
	PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
	MAP_SHARED,		// This program does not have exclusive access to this memory
	mem_fd,			// Map to /dev/mem
	I2C_BASE);		// Offset to I2C peripheral

		if (reg_map == MAP_FAILED) {
	printf("i2c mmap error %d\n", (int)reg_map);
	close(mem_fd);
	exit(-1);
   }

		i2c = (volatile unsigned *)reg_map;
}

// Sets a pin to a certain function
void pinMode(int pin, int function) {
		int reg = pin/10;
		int offset = pin%10 * 3;
		GPFSEL[reg] &= ~((0b111 & ~function) << offset);
		GPFSEL[reg] |= ((0b111 & function) << offset);
}

// Reads the value of a pin
int digitalRead(int pin) {
		int reg = pin/32;
		int depth = pin%32;
		return (GPLEV[reg] >> depth) & 1;
}

// Write high or low to a pin
void digitalWrite(int pin, int value) {
		int reg = pin/32;
		int depth = pin%32;
		if(value == 0) {
			GPCLR[reg] |= (0x1 << depth);
		} else {
			GPSET[reg] |= (value << depth);
		}
}

// Causes delay for a certain number of microseconds
void delayMicrosecs(unsigned int micros) {
		sys_timer[5] = sys_timer[1]+micros;
		sys_timer[0] = 0b0100;
		while(!(sys_timer[0] & 0b0100));
}

// Accesses lower 32-bits of system timer
int getTime() {
		return sys_timer[1];
}
```

```
// Initializes SPI pins, clock, and settings
void spiInit(int frequency, int settings) {
	pinMode(8, ALT0);
	pinMode(9, ALT0);
	pinMode(10, ALT0);
	pinMode(11, ALT0);
	SPICLK[0] = 250000000/frequency;
	SPICS[0] = settings;
	SPICS[0] |= 1 << 7;
}

// Sends and receives a byte through SPI
char spiSendReceive(char send) {
	SPIFIFO[0] = send;
	while(!((SPICS[0] >> 16) & 1));
	return SPIFIFO[0];
}

// Obsolete: Sends and receives 16-bits through SPI
short spiSendReceive2(short send) {
	short receive;
	SPICS[0] |= 1 << 7;
	receive = spiSendReceive(send);
	SPICS[0] &= 0xFFFFFF7F;
	return receive;
}

// Sends and receives 16-bits through SPI
short spiSendReceive16(short send) {
	short receive;
	SPICS[0] |= 1 << 7;
	receive = spiSendReceive((send & 0xFF00) >> 8);
	receive = (receive << 8) | spiSendReceive(send & 0xFF);
	SPICS[0] &= 0xFFFFFF7F;
	return receive;
}

// Initializes I2C pins and clock speed
void i2cInit(int frequency) {
	pinMode(2, ALT0);
	pinMode(3, ALT0);
	I2CCLK[0] = 250000000/frequency;
}

// Sets I2C slave address
void i2cSlaveAdr(char adr) {
	I2CA[0] = adr;
}

// Clears FIFO and error bits in STAT
void i2cClearBits() {
	I2CCTRL[0] = 0x00000020;
	I2CSTAT[0] = 0x00000302;
```

```
}

// Powers on IMU slave given by subadr
int i2cPowerOn(char subadr, char value) {
//          printf("%d\n", subadr);
//          printf("%d\n", value);
            I2CDLEN[0] = 2;// Write 2 bytes
            I2CCTRL[0] = 0x00008000; // Enable BSC
//          printf("%d\n", subadr);
//          printf("%d\n", value);
            I2CFIFO[0] = subadr; // Add subaddress and its value to FIFO
            I2CFIFO[0] = value;
            I2CCTRL[0] = 0x00008080; // Start writing

            while(!(I2CSTAT[0] & 0x00000002)) {}; // Wait for DONE

            printf("%d\n", I2CSTAT[0]);
            return 1;
}

// Reads from a certain number of bytes from a slave of the IMU
int i2cRead(char subadr, char * data, short numBytes) {
            printf("in i2cr\n");
            int i = 0;
            int i2c_byte_wait_us = ((float)I2CCLK[0]/250000000)*9*1000000;

            //Set data length = 1
            I2CDLEN[0] = 1;
            // Enable BSC
            I2CCTRL[0] = 0x00008000;
            // Write subaddress to FIFO
            I2CFIFO[0] = subadr;
            // Enable BSC and ST: BSC necessary?
            I2CCTRL[0] = 0x00008080;

            // Poll TA until active
            while(!(I2CSTAT[0] & 0x00000001)) {
                        //printf("entered\n");
                        if(I2CSTAT[0] & 0x00000002)
                                    break;
                        //printf("polling\n");
            }
            // Set data length = 6 bytes
            I2CDLEN[0] = numBytes;
            // Change to read, enable BSC and ST: BSC necessary?
            I2CCTRL[0] = 0x00008081;
            printf("stuck 1\n");
            // Wait for bytes to start coming back
            delayMicrosecs(i2c_byte_wait_us * 3);
            // Collect data until DONE
/*          while(!(I2CSTAT[0] & 0x00000002)) {
                        while(i < numBytes && (I2CSTAT[0] & 0x00000020)) {
                                    char c = *I2CFIFO;
                                    data[i] = c;
                        i++;
                }
                if(I2CSTAT[0] & 0x00000100) {
                        printf("failed to ack\n");
                }
                if(I2CSTAT[0] & 0x00000200) {
                        printf("hold scl too long\n");
                }
        }
        // Collect remaining number of bytes after DONE
        while(i < numBytes && (I2CSTAT[0] & 0x00000020)) {
                char c = *I2CFIFO;
                data[i] = c;
                i++;
        }/*
        // Wait for DONE
*/      while(!(I2CSTAT[0] & 0x00000002)) {};

        // Collect data while FIFO is populated
        while(i < numBytes) {
                if(I2CSTAT[0] & 0x00000020) {
                        //data[i] = I2CFIFO[0];
                        char c = *I2CFIFO;
                        data[i] = c;
                        i++;
                }
        }
        int result = I2CSTAT[0];
        printf("result=%d\n", result);
        I2CSTAT[0] &= 0x00000002;
        return 1;
}

int printStuff() {
        return 3;
}
#endif

//---------------------Code to Enable Wifi on Pi-------------------//
// By Josephine Wong
// 09 December 2016 jowong@g.hmc.edu
// The following was added to /etc/wpasupplicant/wpasupplicant.conf to allow Wifi connection

network={
  ssid="Claremont-WPA"
  key_mgmt=WPA-EAP
  group=CCMP TKIP
  eap=PEAP
  identity="hmc\jowong"
  password="<my password>"
  phase2="MSCHAPV2"
}
```