**TITLE: SENDING IMU DATA TO THINGSPEAK**

GOAL:

* Interface IMU (all 4 chips onboard) with ATmega328P through I2C
* Send sensor data serially from ATmega328P to ESP8266
* Use LUA Floating Point firmware to upload sensor data
* Display all 10 values from IMU on Thingspeak (IoT service)

DELIVERABLES:

The project deliverables desired were to be able to display all of the sensor data available from the 10 DOF IMU onto an IoT service, such as Thingspeak. The data will be converted from its raw data to the appropriate measurement. Being able to upload every value available to users provides monitoring of anything from temperature and altitude through the barometer to movement and positioning from the other three sensors/chips. While there is a 15 second delay on what can be uploaded with LUA to Thingspeak, it is still very useful for certain applications that the IMU offers.

LITERATURE SURVEY:

Since there was use of all 4 chips available on the 10 DOF IMU, there was not one specific sensor that was focused on and used for a general application. The accelerometer is measured with a range of +- 2G, gyro measuring sensitivities up to +- 2,000 degrees per second, compass able to measure a full 360 degrees with a proper magnetic field, temperature sensor within 1 degree, and measurement of altitude based on a pressure within 100 pascals. The variety of applications available with the 4 sensors range from simple at-home devices such as a thermostat to even wireless controllers for other devices through the use of multiple WiFi boards. The limitations of more complex devices that need real time monitoring and updates will be restrained by upload times of the IoT service being used or even the firmware that is being used on the ESP8266 board. Finding services with faster allowed uploads or using a different firmware can minimize this issue though. Obviously the max ranges measurable by the sensors will allow some limitations but it is expected with a relatively low cost for the entire board.

COMPONENTS:

All 4 chips on this board are communicated by through I2C. The SCL is set to run at 400kHz in fast mode as opposed to the 100kHz normal mode. All of the sensors can run at a maximum of 400kHz which is why I went ahead and chose this speed, although the barometer can run up to 3.4 MHz. The write/read addresses for each of the sensors is as follows:

**HMC5883L (compass)** **0x1E** – 0x3C write, 0x3D read

**ADXL345 (accelerometer)** **0x53** – 0xA6 write, 0xA7 read

**ITG3200 (gyro)** **0x68** – 0xD0 write, 0xD1 read (although this sensor has a pin that can be tied to VCC causing the slave address to change)

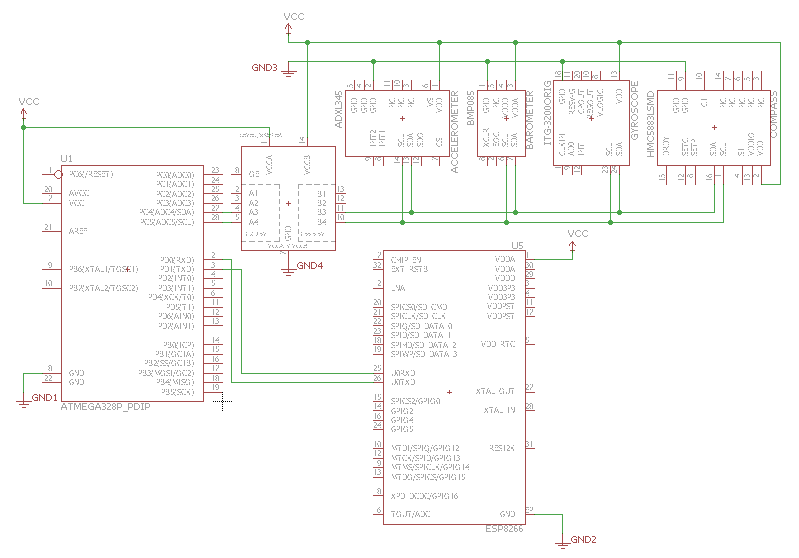
**BMP085 (barometer) 0x77** – 0xEE write, 0xEF read

Since Thingspeak has a 15 second limit on data being uploaded, there is no issue with the ATmega328P sending data too slow and creating blank uploads, but there is the issue of everything being sent in between the 15 second intervals not being uploaded whatsoever. This is mainly an issue that can be resolved by averaging out from the ATmega328P but the majority of values being read would not make sense averaging out such as accelerometer or gyro data. Taking the average movement over a 15 second interval could just lead to a close value of 0 a majority of the time. This is why I figured that knowing the exact measurements at the time of uploading would be the best approach and no averaging was taken into account.

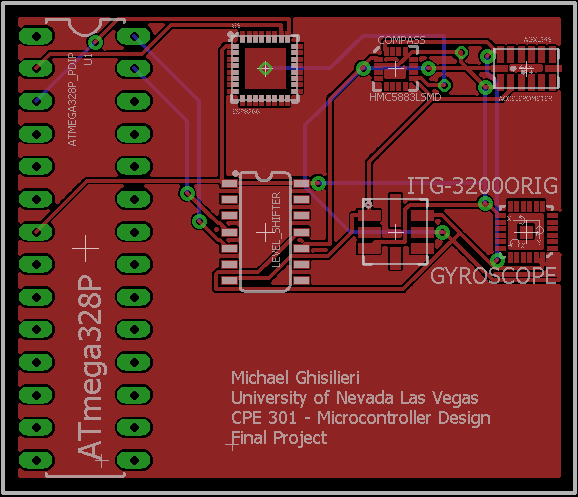
The ATmega328P handles not only all of the reading of values but also the calculations and conversions needed to get the actual measurements that aren’t in raw data. As mentioned in the literature survey, each of the sensors has a sensitivity/accuracy range that is measures within.

SCHEMATIC:

This is a screenshot in “zoom to fit” view in Eagle which blurred most of the pin names in the schematic but a clear shot of each of the components is available in the drive folder. The 10 DOF IMU wasn’t available from any libraries I found so I just used each sensor individually on the schematic.



PCB:



IMPLEMENTATION:

* IMU connected to ATmega328P through I2C (only SDA/SCL needed, no INT on board)
* ATmega328P connected to ESP8266 through RX/TX
* LUA will be flashed with new WiFi SSID and password if in a new location
* I2C SCL set to 400kHz and baud rate set to 9600
* Each value read in every half of a second in the order of compass, accelerometer, gyro, and barometer
* Raw data converted immediately after being read in and stored as floats for greater accuracy in terms of data and display on charts through Thingspeak
* Data is also output serially to the ESP8266 every half of a second and LUA will be updating the variables storing these values but Thingspeak will only be updated every 15 seconds
* 2 Thingspeak channels needed since max of 8 fields per channel and 10 values from IMU
* Default graphs used but display of last 5 values only shown for a smaller range of values which is easier for users to view

SNAPSHOTS/SCREENSHOTS:

Video link that provides both component explanation on breadboard as well as full demonstration of data being uploaded to Thingspeak: <https://youtu.be/EI9FAQbZcrw>

Images folder for photos described here: <https://drive.google.com/folderview?id=0BzVSchvmDqtrakN2ZXNJaTVHWUU&usp=sharing>

3 images are shown to just give more shots of what the video covers, although the screenshots don’t show all 10 values (since my compass doesn’t work properly and I was borrowing the other 10 DOF board for the video), just the temperature, altitude, gyroscope values, and accelerometer values.

As mentioned previously, a cleaner shot of the schematic is shown with the VCC and GND cut off for some of the chips.

An image of the breadboard without the FTDI chip (only used for programming) attached to the WiFi.

Screenshot of ESPlorer to show messages sent back from ESP8266 saying that posting to Channel 1 and Channel 2 of Thingspeak was posted OK.

CODE:

**First file is the C code on the ATmega328P:**

/\*

\* I2C.c

\*

\* Created: 4/26/2016 4:29:32 PM

\* Author : Michael

\*/

// TWBR Bit Rate Register

// TWCR Control Register (start/stop bits, enable, interrupt)

// TWDR Data Register (data for transmit/receive)

// TWSR Status Register

#define *F\_CPU* 16000000UL // ATmega328 Xplained board runs at 16MHz

#include <avr/io.h> // library for I/O

#include <util/delay.h> // library for \_delay\_ms function

#include <stdlib.h> // library for printf function

#include <stdio.h> // library for printf function

#include <math.h> // for atan2 and pow functions

void compass\_init(); // initializes compass registers

void i2c\_readcompass(); // reads values for compass

void compass2float(); // converts compass values from raw data

void accelo\_init(); // initializes accelorometer registers

void i2c\_readaccelo(); // reads values for accelorometer

void accelo2float(); // converts accelorometer values from raw data

void gyro\_init(); // initializes gyro registers

void i2c\_readgyro(); // reads values for gyro

void gyro2float(); // converts gyro values from raw data

void baro\_calibrations(); // reads barometer calibration values

void read\_temperature(); // reads and calculates raw temperature

void temp2float(); // converts raw temperature to Fahrenheit

void read\_pressure(); // reads and calculates raw pressure

void altitude2float(); // converts raw pressure into altitude

int USART0SendByte(char , *FILE* \*stream); // send a byte through UART

int USART0ReceiveByte(*FILE* \*stream); // receive byte through UART

void delay (); // delay for given time

// variables for each value being read from sensors

float x\_compass, y\_compass, z\_compass, heading;

float x\_accelo, y\_accelo, z\_accelo;

float temp\_gyro, x\_gyro, y\_gyro, z\_gyro;

// used for final temperature and altitude values from barometer

float real\_temp, altitude;

// set of variables for calibration values and calculations using the barometer

long AC1, AC2, AC3, B1, B2, MB, MC, MD;

unsigned long AC4, AC5, AC6;

long T, P, UT, UP, X1, X2, X3, B3, B5, B6;

unsigned long B4, B7;

//set stream pointer

*FILE* usart0\_str = *FDEV\_SETUP\_STREAM*(USART0SendByte, USART0ReceiveByte, *\_FDEV\_SETUP\_RW*);

// stops i2c communication

void i2c\_stop()

{

TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTO);

}

// writes value to current i2c slave address

// and waits for an acknowledgment

void i2c\_write(int data)

{

TWDR = data;

TWCR = (1<<TWINT)|(1<<TWEN);

while (!(TWCR & (1<<TWINT)));

}

// starts i2c communication

void i2c\_start(void)

{

TWCR = (1<< TWINT) | (1<<TWSTA) | (1<< TWEN);

while (!(TWCR & (1<<TWINT)));

}

// initializes i2c registers in ATmega328P

void i2c\_init(void)

{

TWSR = 0x00; //set prescaler to 0

TWBR = 0x0C; //SCL freq. is 400k for fast mode of devices

TWCR = 0x04; //enable TWI module

}

// read byte with ACK

*uint8\_t* i2c\_readACK(void)

{

TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWEA);

while ((TWCR & (1<<TWINT)) == 0);

return TWDR;

}

//read byte with NACK

*uint8\_t* i2c\_readNACK(void)

{

TWCR = (1<<TWINT)|(1<<TWEN);

while ((TWCR & (1<<TWINT)) == 0);

return TWDR;

}

int main(void)

{

// libprintf\_flt.a and check box for vprintf

// used to print floating point numbers with printf

// set stream pointer to standard stream

*stdin* = *stdout* = &usart0\_str;

UBRR0L = 0x67; // set based on equation using baud rate of 9600

UCSR0B = (1 << TXEN0); // Enable transmitter

UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // 8 bit data

// Initializations of i2c, compass, acceloremeter, and gyro

i2c\_init();

compass\_init();

accelo\_init();

gyro\_init();

// to allow i2c to be ready

*\_delay\_us*(100);

while(1) {

i2c\_readcompass(); // read compass values

compass2float(); // convert raw values to actual float data

i2c\_readaccelo(); // read accelorometer values

accelo2float(); // convert raw values to actual float data

i2c\_readgyro(); // read gyro values

gyro2float(); // convert raw values to actual float data

baro\_calibrations(); // read barometer calibration registers

read\_temperature(); // read and calculate raw temperature

read\_pressure(); // read and calculate raw pressure

temp2float(); // convert raw temperature to actual float data

altitude2float(); // convert raw pressure to altitude float data

/\* These four lines of code output to a regular terminal serially

to display the contents of every sensor in a labeled manner.

printf("Heading: %.3f, Z Magnetic Field: %.3f \n\r", heading, z\_compass);

printf("X Acc: %.3f, Y Acc: %.3f, Z Acc: %.3f\n", x\_accelo, y\_accelo, z\_accelo);

printf("X Gyro: %.3f, Y Gyro: %.3f, Z Gyro: %.3f\n\r", x\_gyro, y\_gyro, z\_gyro);

printf("Altitude: %.3f, Temperature: %.3f\n\r\n\r", altitude, real\_temp);

\*/

// prints out all 10 sensor values with 3 bit precision

// this is the exact format for the data that LUA will be expecting the values in

*printf*("%.3f %.3f %.3f %.3f %.3f %.3f %.3f %.3f %.3f %.3f\n", heading, z\_compass,

x\_accelo, y\_accelo, z\_accelo, x\_gyro, y\_gyro, z\_gyro, real\_temp, altitude);

// delay for 500ms

delay();

}

return 0;

}

void compass\_init()

{

i2c\_start();

i2c\_write(0x3C); // write mode

i2c\_write(0x00); // configuration register A

i2c\_write(0x70); // 8 samples, 15 Hz

i2c\_stop();

i2c\_start();

i2c\_write(0x3C); // write mode

i2c\_write(0x01); // configuration register B

i2c\_write(0xA0); // Gain of 4.7

i2c\_stop();

}

void i2c\_readcompass()

{

i2c\_start();

i2c\_write(0x3C); // write mode

i2c\_write(0x02); // mode register

i2c\_write(0x01); // single-measurement mode

i2c\_stop();

*\_delay\_ms*(8); // delay 8 ms for monitor status

i2c\_start();

i2c\_write(0x3D); // read mode

x\_compass = i2c\_readACK()<<8; // MSBX

x\_compass += i2c\_readACK(); // LSBX

z\_compass = i2c\_readACK()<<8; // MSBZ

z\_compass += i2c\_readACK(); // LSBZ

y\_compass = i2c\_readACK()<<8; // MSBY

y\_compass += i2c\_readNACK(); // LSBY

i2c\_stop();

}

void compass2float()

{

// Converts X and Y values to angle in direction.

// Converts Z value to magnetic field from earth.

heading = *atan2*(x\_compass, y\_compass); // arctan of x/y

float newPI = 2\**M\_PI*; // stores common 2\*M\_PI as float

// adjust heading to proper sign

if(heading < 0)

heading += newPI;

if(heading > newPI)

heading -= newPI;

// convert radians to degrees

heading = heading \* 180;

heading /= *M\_PI*;

// gain of 4.7 so divide Z-Axis by this value for magnetic field

z\_compass /= 4.7;

}

void accelo\_init()

{

i2c\_start();

i2c\_write(0xA6); // write mode

i2c\_write(0x2D); // POWER\_CTL register

i2c\_write(0x08); // turn on measuring

i2c\_stop();

}

void i2c\_readaccelo()

{

// MSBX value

i2c\_start();

i2c\_write(0xA6); // write mode

i2c\_write(0x32);

i2c\_start();

i2c\_write(0xA7); // read mode

x\_accelo = i2c\_readACK();

// LSBX value

i2c\_start();

i2c\_write(0xA6); // write mode

i2c\_write(0x33);

i2c\_start();

i2c\_write(0xA7); // read mode

x\_accelo += i2c\_readACK()<<8;

// MSBY value

i2c\_start();

i2c\_write(0xA6); // write mode

i2c\_write(0x34);

i2c\_start();

i2c\_write(0xA7); // read mode

y\_accelo = i2c\_readACK();

// LSBY value

i2c\_start();

i2c\_write(0xA6); // write mode

i2c\_write(0x35);

i2c\_start();

i2c\_write(0xA7); // read mode

y\_accelo += i2c\_readACK()<<8;

// MSBZ value

i2c\_start();

i2c\_write(0xA6); // write mode

i2c\_write(0x36);

i2c\_start();

i2c\_write(0xA7); // read mode

z\_accelo = i2c\_readACK();

// LSBZ value

i2c\_start();

i2c\_write(0xA6); // write mode

i2c\_write(0x37);

i2c\_start();

i2c\_write(0xA7); // read mode

z\_accelo += i2c\_readNACK()<<8;

i2c\_stop();

}

void accelo2float()

{

// Accelorometer using +-2G measurement

// So factor will be (range/1024) = 4/1024 or 0.0039

x\_accelo \*= 0.0039;

y\_accelo \*= 0.0039;

z\_accelo \*= 0.0039;

}

void gyro\_init()

{

i2c\_start();

i2c\_write(0xD0); // write mode

i2c\_write(0x16); // Configuration Register

i2c\_write(0x1B); // +- 2000 deg/sec scale

i2c\_stop();

}

void i2c\_readgyro()

{

i2c\_start();

i2c\_write(0xD0); // write mode

i2c\_write(0x1B); // address of first value

i2c\_start();

i2c\_write(0xD1); // read mode

temp\_gyro = i2c\_readACK()<<8; // TEMP\_H (not used)

temp\_gyro += i2c\_readACK(); // TEMP\_L (not used)

x\_gyro = i2c\_readACK()<<8; // MSBX

x\_gyro += i2c\_readACK(); // LSBX

y\_gyro = i2c\_readACK()<<8; // MSBY

y\_gyro += i2c\_readACK(); // LSBY

z\_gyro = i2c\_readACK()<<8; // MSBZ

z\_gyro += i2c\_readNACK(); // LSBZ

i2c\_stop();

}

void gyro2float()

{

// Gyro has sensitivity of 14.375 LSBs per deg/sec

// Divide by 14.375 to get proper rotation in deg/sec

x\_gyro /= 14.375;

y\_gyro /= 14.375;

z\_gyro /= 14.375;

}

void baro\_calibrations()

{

i2c\_start();

i2c\_write(0xEE); // write mode

i2c\_write(0xAA); // first calibration register

i2c\_start();

i2c\_write(0xEF); // read mode

AC1 = i2c\_readACK()<<8; // AC1\_H

AC1 += i2c\_readACK(); // AC1\_L

AC2 = i2c\_readACK()<<8; // AC2\_H

AC2 += i2c\_readACK(); // AC2\_L

AC3 = i2c\_readACK()<<8; // AC3\_H

AC3 += i2c\_readACK(); // AC3\_L

AC4 = i2c\_readACK()<<8; // AC4\_H

AC4 += i2c\_readACK(); // AC4\_L

AC5 = i2c\_readACK()<<8; // AC5\_H

AC5 += i2c\_readACK(); // AC5\_L

AC6 = i2c\_readACK()<<8; // AC6\_H

AC6 += i2c\_readACK(); // AC6\_L

B1 = i2c\_readACK()<<8; // B1\_H

B1 += i2c\_readACK(); // B1\_L

B2 = i2c\_readACK()<<8; // B2\_H

B2 += i2c\_readACK(); // B2\_L

MB = i2c\_readACK()<<8; // MB\_H

MB += i2c\_readACK(); // MB\_L

MC = i2c\_readACK()<<8; // MC\_H

MC += i2c\_readACK(); // MC\_L

MD = i2c\_readACK()<<8; // MD\_H

MD += i2c\_readNACK(); // MD\_L

i2c\_stop();

}

void read\_temperature()

{

long temp = 0; // temporary variable for calculations

i2c\_start();

i2c\_write(0xEE); // write mode

i2c\_write(0xF4); // measurement selection

i2c\_write(0x2E); // temperature mode

i2c\_stop();

*\_delay\_ms*(5); // datasheet requests delay

i2c\_start();

i2c\_write(0xEE); // write mode

i2c\_write(0xF6);

i2c\_start();

i2c\_write(0xEF); // read mode

temp = i2c\_readACK(); // TEMP\_H

UT = temp<<8;

UT += i2c\_readNACK(); // TEMP\_L

i2c\_stop();

// Equation from datasheet to calculate true temperature

// X1 = (UT - AC6) \* AC5 / 2^15

// X2 = MC \* 2^11 / (X1 + MD)

// B5 = X1 + X2

// T = (B5 + 8) / 2^4

X1 = UT-AC6;

X1 = X1\*AC5;

X1 = X1/32768;

X2 = MC\*2048;

temp = X1+MD;

X2 = X2/temp;

B5 = X1+X2;

T = B5+8;

T = T/16;

}

void temp2float()

{

// Since raw temperature is given in 0.1 C

// Divide by 10 to get true Celsius temperature.

// Then apply F = C\*1.8 + 32 to convert to Fahrenheit

real\_temp = T\*1.8;

real\_temp = real\_temp/10.0;

real\_temp = real\_temp+32.0;

}

void read\_pressure()

{

long temp = 0; // temporary variable for calculations

i2c\_start();

i2c\_write(0xEE); // write mode

i2c\_write(0xF4); // measurement selection

i2c\_write(0x34); // pressure mode

i2c\_stop();

*\_delay\_ms*(5); // datasheet requests delay

i2c\_start();

i2c\_write(0xEE); // write mode

i2c\_write(0xF6);

i2c\_start();

i2c\_write(0xEF); // read mode

temp = i2c\_readACK(); // MSB

UP = temp<<16;

temp = i2c\_readACK(); // LSB

UP += temp<<8;

UP += i2c\_readNACK(); // XLSB

UP = UP >> 8; // request to >> (8-oss) but oss = 0

i2c\_stop();

// Equation from datasheet to calculate true pressure

// B6 = B5 - 4000

// X1 = (B2 \* (B6 \* B6 / 2^12)) / 2^11

// X2 = AC2 \* B6 / 2^11

// X3 = X1 + X2

// B3 = ((AC1\*4+X3) << oss + 2) / 4

// X1 = AC3 \* B6 / 2^13

// X2 = (B1 \* (B6 \* B6 / 2^12)) / 2^16

// X3 = ((X1 + X2) + 2) / 2^2

// B4 = AC4 \* (unsigned long)(X3 + 32768) / 2^15

// B7 = ((unsigned long)UP - B3) \* (50000 >> oss)

// if (B7 < 0x80000000) { p = (B7 \* 2) / B4 }

// else { p = (B7 / B4) \* 2 }

// X1 = (p / 2^8) \* (p / 2^8)

// X1 = (X1 \* 3038) / 2^16

// X2 = (-7357 \* p) / 2^16

// p = p + (X1 + X2 + 3791) / 2^4

B6 = B5-4000;

X1 = B6\*B6;

X1 = X1/4096;

X1 = X1\*B2;

X1 = X1/2048;

X2 = AC2\*B6;

X2 = X2/2048;

X3 = X1+X2;

B3 = AC1\*4;

B3 = B3+X3;

B3 = B3+2;

B3 = B3/4;

X1 = AC3\*B6;

X1 = X1/8192;

X2 = B6\*B6;

X2 = X2/4096;

X2 = X2\*B1;

X2 = X2/65536;

X3 = X1+X2;

X3 = X3+2;

X3 = X3/4;

B4 = (unsigned long)(X3+32768);

B4 = B4\*AC4;

B4 = B4/32768;

B7 = (unsigned long)(UP-B3);

B7 = B7\*50000;

if(B7 < 0x80000000) {

P = B7\*2;

P = P/B4;

}

else {

P = B7/B4;

P = P\*2;

}

X1 = P/256;

X1 = X1\*P;

X1 = X1/256;

X1 = X1\*3038;

X1 = X1/65536;

X2 = (-7357\*P);

X2 = X2/65536;

temp = X1+X2;

temp = temp+3791;

temp = temp/16;

P = P+temp;

}

void altitude2float()

{

// Altitude equation given in another library

// but given in datasheet by

// 44330 \* (1 - (p/po)^(1/5.255))

float temp = 0;

temp = P/(float)100339;

altitude = 1 - *pow*(temp, 0.1903);

altitude /= 0.0000225577;

altitude \*= 3.28084;

}

int USART0SendByte(char u8Data, *FILE* \*stream)

{

if(u8Data == '\n')

{

USART0SendByte('\r', 0);

}

//wait while previous byte is completed

while(!(UCSR0A&(1<<UDRE0))){};

// Transmit data

UDR0 = u8Data;

return 0;

}

int USART0ReceiveByte(*FILE* \*stream)

{

*uint8\_t* u8Data;

// Wait for byte to be received

while(!(UCSR0A&(1<<RXC0))){};

u8Data=UDR0;

//echo input data

USART0SendByte(u8Data,stream);

// Return received data

return u8Data;

}

void delay () {

int i; // declare i for counter

for(i = 0; i < 50; i++) { // loop 50 times

*\_delay\_ms*(10); // delay 10 ms

} // total delay = 50\*10ms = 500ms

}

**Second file is the LUA code saved onto the ESP8266:**

wifi.setmode(wifi.STATION); -- sets operation mode

wifi.sta.config("SSID" ,"password"); -- configure wifi connection (took mine out)

-- variables for storing compass values

heading = 0

z\_compass = 0

-- stores accelerometer values

x\_accelo = 0

y\_accelo = 0

z\_accelo = 0

-- stores gyro values

x\_gyro = 0

y\_gyro = 0

z\_gyro = 0

-- stores barometer values

real\_temp = 0

altitude = 0

-- uart is constantly receiving data

-- when a "\n" character is received (which I include in my printf from 328P)

-- this block of code will execute

uart.on("data", "\n",

function(data) -- if "quit" is typed to ESP8266, UART callback reset

if (string.match(data, "quit")) then

print("Quitting...")

uart.on("data")

end

-- expects this exact format for the incoming serial data

-- each (-\*%d+.%d+) will basically mean

-- negative sign, any number of digits, "." for decimal point, any number of digits

-- "+" means at leasts one or more digit needs to appear

heading, z\_compass, x\_accelo, y\_accelo, z\_accelo, x\_gyro, y\_gyro, z\_gyro, real\_temp, altitude = data:match("(-\*%d+.%d+) (-\*%d+.%d+) (-\*%d+.%d+) (-\*%d+.%d+) (-\*%d+.%d+) (-\*%d+.%d+) (-\*%d+.%d+) (-\*%d+.%d+) (-\*%d+.%d+) (-\*%d+.%d+)%s\*")

end,

0)

function postThingSpeak(level)

connout = nil

connout = net.createConnection(net.TCP, 0)

-- when sending data, this will output if the data was

-- posted correctly (not necessarily meaning the correct data was sent though)

connout:on("receive", function(connout, payloadout)

if (string.find(payloadout, "Status: 200 OK") ~= nil) then

print("Posted Channel 1 OK");

end

end)

connout:on("connection", function(connout, payloadout)

-- print to terminal that posting is currectly occurring

print ("Posting Channel 1 ...");

-- this whole block will set the API\_KEY and each GET or POST

-- line will output the specified variable to the fieldX specified

connout:send("GET /update?api\_key=MJK010IW6EPLZ776&field1=" .. heading

.. "GET /update?api\_key=MJK010IW6EPLZ776&field2=" .. z\_compass

.. "GET /update?api\_key=MJK010IW6EPLZ776&field3=" .. x\_accelo

.. "GET /update?api\_key=MJK010IW6EPLZ776&field4=" .. y\_accelo

.. "GET /update?api\_key=MJK010IW6EPLZ776&field5=" .. z\_accelo

.. "GET /update?api\_key=MJK010IW6EPLZ776&field6=" .. x\_gyro

.. "GET /update?api\_key=MJK010IW6EPLZ776&field7=" .. y\_gyro

.. "GET /update?api\_key=MJK010IW6EPLZ776&field8=" .. z\_gyro

.. " HTTP/1.1\r\n"

.. "Host: api.thingspeak.com\r\n"

.. "Connection: close\r\n"

.. "Accept: \*/\*\r\n"

.. "User-Agent: Mozilla/4.0 (compatible; esp8266 Lua; Windows NT 5.1)\r\n"

.. "\r\n")

end)

-- closes connection with host

connout:on("disconnection", function(connout, payloadout)

connout:close();

collectgarbage();

end)

connout:connect(80,'api.thingspeak.com')

-- this second block of similar code does exactly what the first block

-- of code will be doing except that this posts to the second channel

-- with a different API\_KEY. setting the API\_KEY on each GET/POST line

-- in the first block caused issues and the API\_KEY wouldn't update so

-- adding this setting request to connect and POST to a new channel was

-- the simple fix around that

connout = nil

connout = net.createConnection(net.TCP, 0)

connout:on("receive", function(connout, payloadout)

if (string.find(payloadout, "Status: 200 OK") ~= nil) then

print("Posted Channel 2 OK");

end

end)

connout:on("connection", function(connout, payloadout)

print ("Posting Channel 2 ...");

connout:send("GET /update?api\_key=UHKR6IHYZRBGTCTG&field1=" .. real\_temp

.. "GET /update?api\_key=UHKR6IHYZRBGTCTG&field2=" .. altitude

.. " HTTP/1.1\r\n"

.. "Host: api.thingspeak.com\r\n"

.. "Connection: close\r\n"

.. "Accept: \*/\*\r\n"

.. "User-Agent: Mozilla/4.0 (compatible; esp8266 Lua; Windows NT 5.1)\r\n"

.. "\r\n")

end)

connout:on("disconnection", function(connout, payloadout)

connout:close();

collectgarbage();

end)

connout:connect(80,'api.thingspeak.com')

end

-- this timer setting simply repeats the function specified

-- which is the main function to post data to both channels

-- and it occurs every 15000 milliseconds or 15 seconds since

-- that is the delay required when uploading to Thingspeak

tmr.alarm(1, 15000, 1, function() postThingSpeak(0) end)

REFERENCE:

LUA code based off of this tutorial: <http://captain-slow.dk/2015/04/16/posting-to-thingspeak-with-esp8266-and-nodemcu/>

I2C library wasn’t used but functions were grabbed from: <http://www.embedds.com/programming-avr-i2c-interface/>

Conversion of compass values: <http://bluelemonlabs.blogspot.com/2013/08/arduino-simple-compass-with-hmc5883l.html>

Datasheets provided under “Useful Links” section on: <http://www.robotshop.com/en/imu-10-dof-16g-3-axis-accelerometer-2000--s-gyromagnetometerbarometer.html>