**Date Submitted: October 26, 2019**

For all tasks, I was unable to read from the MPU6050 sensor and used a value from the internal temperature sensor to imitate values for the accelerometer and gyroscope.

**------------------------------------------------------------------------------------**

**Task 01 and Task 02 (Task 02 only asks for data in Task 01’s to be graphed :**

Youtube Link: https://youtu.be/F\_31kiPv1Fk

**Modified Schematic (if applicable):**

**Modified Code:**

/\*

\* tivac\_midterm\_t01.c

\*

\* Created on: Oct 26, 2019

\* Author: gausp

\*/

//I was not able to read from the MPU sensor so I used values from the internal temperature sensor to act as the values for the Accelerometer

//and gyroscope

**#include** <stdio.h>

**#include** <stdbool.h>

**#include** <stdint.h>

**#include** <stdlib.h>

**#include** <stdio.h>

**#include** <stdarg.h>

**#include** <stdbool.h>

**#include** "sensorlib/i2cm\_drv.h"

**#include** "sensorlib/i2cm\_drv.c"

**#include** "sensorlib/hw\_mpu6050.h"

**#include** "sensorlib/mpu6050.h"

**#include** "sensorlib/mpu6050.c"

//#include "inc/tm4c123gh6pm.h"

**#include** "inc/hw\_ints.h"

**#include** "inc/hw\_memmap.h"

**#include** "inc/hw\_sysctl.h"

**#include** "inc/hw\_types.h"

**#include** "inc/hw\_i2c.h"

**#include** "inc/hw\_types.h"

**#include** "inc/hw\_gpio.h"

**#include** "driverlib/gpio.h"

**#include** "driverlib/pin\_map.h"

**#include** "driverlib/rom.h"

**#include** "driverlib/rom\_map.h"

**#include** "driverlib/debug.h"

**#include** "driverlib/interrupt.h"

**#include** "driverlib/i2c.h"

**#include** "driverlib/sysctl.h"

**#include** "driverlib/uart.h"

**#include** "driverlib/adc.h"

**#include** "utils/uartstdio.h"

**#include** "utils/uartstdio.c"

//

// A boolean that is set when a MPU6050 command has completed.

//

**volatile** bool g\_bMPU6050Done;

// I2C master instance

//

tI2CMInstance g\_sI2CMSimpleInst;

//

// The function that is provided by this example as a callback when MPU6050

// transactions have completed.

//

/\*void MPU6050Callback(void \*pvCallbackData, uint\_fast8\_t ui8Status)

{

//

// See if an error occurred.

//

if(ui8Status != I2CM\_STATUS\_SUCCESS)

{

//

// An error occurred, so handle it here if required.

//

}

//

// Indicate that the MPU6050 transaction has completed.

//

g\_bMPU6050Done = true;

}

\*/

**void** **InitI2C0**(**void**)

//Configure/initialize the I2C0

{

**SysCtlPeripheralEnable** (SYSCTL\_PERIPH\_I2C0); //enables I2C0

**SysCtlPeripheralReset** (SYSCTL\_PERIPH\_I2C0); //reset module

**SysCtlPeripheralEnable** (SYSCTL\_PERIPH\_GPIOB); //enable PORTB as peripheral

//Configure the pin muxing for I2C0 functions on port B2 and B3

**GPIOPinTypeI2C** (GPIO\_PORTB\_BASE, GPIO\_PIN\_3); //set I2C PB3 as SDA

**GPIOPinConfigure** (GPIO\_PB3\_I2C0SDA);

**GPIOPinTypeI2CSCL** (GPIO\_PORTB\_BASE, GPIO\_PIN\_2); //set I2C PB2 as SCLK

**GPIOPinTypeI2C**(GPIO\_PORTB\_BASE, GPIO\_PIN\_3);

//Enable and initialize the I2C0 master module. Use the system clock for

//the I2C0 module. The last parameter sets the I2C data transfer rate.

//If false the data rata is set to 100kbps and if true the data rate will be

//set to 400kbps

**I2CMasterInitExpClk** (I2C0\_BASE, **SysCtlClockGet**(), false); //Set the clock of the I2C to ensure proper connection

//clear I2C FIFOs

HWREG(I2C0\_BASE + I2C\_O\_FIFOCTL) = 80008000;

}

//

// The MPU6050 example.

//

**void** **MPU6050Example**(**void**)

{

**float** fAccel[3], fGyro[3];

tI2CMInstance sI2CInst;

tMPU6050 sMPU6050;

//

// Initialize the MPU6050. This code assumes that the I2C master instance

// has already been initialized.

//

g\_bMPU6050Done = false;

**MPU6050Init**(&sMPU6050, &sI2CInst, 0x68, MPU6050Callback, 0);

**while**(!g\_bMPU6050Done)

{

}

//

// Configure the MPU6050 for +/- 4 g accelerometer range.

//

g\_bMPU6050Done = false;

**MPU6050ReadModifyWrite**(&sMPU6050, MPU6050\_O\_ACCEL\_CONFIG,

~MPU6050\_ACCEL\_CONFIG\_AFS\_SEL\_M,

MPU6050\_ACCEL\_CONFIG\_AFS\_SEL\_4G, MPU6050Callback,

0);

**while**(!g\_bMPU6050Done)

{

}

//

// Loop forever reading data from the MPU6050. Typically, this process

// would be done in the background, but for the purposes of this example,

// it is shown in an infinite loop.

//

**while**(1)

{

//

// Request another reading from the MPU6050.

//

g\_bMPU6050Done = false;

**MPU6050DataRead**(&sMPU6050, MPU6050Callback, 0);

**while**(!g\_bMPU6050Done)

{

}

//

// Get the new accelerometer and gyroscope readings.

//

**MPU6050DataAccelGetFloat**(&sMPU6050, &fAccel[0], &fAccel[1],

&fAccel[2]);

**MPU6050DataGyroGetFloat**(&sMPU6050, &fGyro[0], &fGyro[1], &fGyro[2]);

//

// Do something with the new accelerometer and gyroscope readings.

//

}

}

**void**

**InitConsole**(**void**)

{

//

// Enable GPIO port A which is used for UART0 pins.

// **TODO**: change this to whichever GPIO port you are using.

//

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_GPIOA);

//

// Configure the pin muxing for UART0 functions on port A0 and A1.

// This step is not necessary if your part does not support pin muxing.

// **TODO**: change this to select the port/pin you are using.

//

**GPIOPinConfigure**(GPIO\_PA0\_U0RX);

**GPIOPinConfigure**(GPIO\_PA1\_U0TX);

//

// Enable UART0 so that we can configure the clock.

//

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_UART0);

//

// Use the internal 16MHz oscillator as the UART clock source.

//

**UARTClockSourceSet**(UART0\_BASE, UART\_CLOCK\_PIOSC);

//

// Select the alternate (UART) function for these pins.

// **TODO**: change this to select the port/pin you are using.

//

**GPIOPinTypeUART**(GPIO\_PORTA\_BASE, GPIO\_PIN\_0 | GPIO\_PIN\_1);

//

// Initialize the UART for console I/O.

//

**UARTStdioConfig**(0, 115200, 16000000);

}

**void** **initADC**() {

// The ADC0 peripheral must be enabled for use.

//

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_ADC0);

**SysCtlDelay**(3);

//

// Enable sample sequence 3 with a processor signal trigger. Sequence 3

// will do a single sample when the processor sends a singal to start the

// conversion. Each ADC module has 4 programmable sequences, sequence 0

// to sequence 3. This example is arbitrarily using sequence 3.

//

**ADCSequenceConfigure**(ADC0\_BASE, 3, ADC\_TRIGGER\_PROCESSOR, 0);

//

// Configure step 0 on sequence 3. Sample the temperature sensor

// (ADC\_CTL\_TS) and configure the interrupt flag (ADC\_CTL\_IE) to be set

// when the sample is done. Tell the ADC logic that this is the last

// conversion on sequence 3 (ADC\_CTL\_END). Sequence 3 has only one

// programmable step. Sequence 1 and 2 have 4 steps, and sequence 0 has

// 8 programmable steps. Since we are only doing a single conversion using

// sequence 3 we will only configure step 0. For more information on the

// ADC sequences and steps, reference the datasheet.

//

**ADCSequenceStepConfigure**(ADC0\_BASE, 3, 0, ADC\_CTL\_TS | ADC\_CTL\_IE |

ADC\_CTL\_END);

//

// Since sample sequence 3 is now configured, it must be enabled.

//

**ADCSequenceEnable**(ADC0\_BASE, 3);

//

// Clear the interrupt status flag. This is done to make sure the

// interrupt flag is cleared before we sample.

//

**ADCIntClear**(ADC0\_BASE, 3);

}

uint32\_t **getADCtempf**(**void**) {

// This array is used for storing the data read from the ADC FIFO. It

// must be as large as the FIFO for the sequencer in use. This example

// uses sequence 3 which has a FIFO depth of 1. If another sequence

// was used with a deeper FIFO, then the array size must be changed.

//

uint32\_t ADCValues[1];

//

// These variables are used to store the temperature conversions for

// Celsius and Fahrenheit.

//

uint32\_t TempValueC ;

uint32\_t TempValueF ;

//

// Trigger the ADC conversion.

//

**ADCProcessorTrigger**(ADC0\_BASE, 3);

//

// Wait for conversion to be completed.

//

**while**(!**ADCIntStatus**(ADC0\_BASE, 3, false))

{

}

//

// Clear the ADC interrupt flag.

//

**ADCIntClear**(ADC0\_BASE, 3);

//

// Read ADC Value.

//

**ADCSequenceDataGet**(ADC0\_BASE, 3, ADCValues);

//

// Use non-calibrated conversion provided in the data sheet. I use floats in intermediate

// math but you could use intergers with multiplied by powers of 10 and divide on the end

// Make sure you divide last to avoid dropout.

//

TempValueC = (uint32\_t)(147.5 - ((75.0\*3.3 \*(**float**)ADCValues[0])) / 4096.0);

//

// Get Fahrenheit value. Make sure you divide last to avoid dropout.

//

TempValueF = ((TempValueC \* 9) + 160) / 5;

//

// This function provides a means of generating a constant length

// delay. The function delay (in cycles) = 3 \* parameter. Delay

// 250ms arbitrarily.

//

**SysCtlDelay**(80000000 / 12);

**return** TempValueF;

}

**void** **MPU6050\_imitator**(**void**) { //Will send values to serial terminal acting as Accel and Gyro values

**float** Ax, Ay, Az, Gx, Gy, Gz;

uint32\_t i32IntegerPart, i32FractionPart;

**int** i = getADCtempf();

Ax = i;

Gx = 5\*(Ax/8);

Ay = Ax/2;

Gy = 11\*(Ax/8);

Az = Ax\*2;

Gz = 7\*(Ax/8);

i32IntegerPart = (int32\_t)Ax;

i32FractionPart = (int32\_t)(Ax \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("\nAx %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Ay;

i32FractionPart = (int32\_t)(Ay \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Ay %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Az;

i32FractionPart = (int32\_t)(Az \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Az %d.%d\n", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gx;

i32FractionPart = (int32\_t)(Gx \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gx %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gy;

i32FractionPart = (int32\_t)(Gy \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gy %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gz;

i32FractionPart = (int32\_t)(Gz\* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gz %d.%d\n", i32IntegerPart, i32FractionPart);

}

**int** **main**(**void**)

{

**SysCtlClockSet**(SYSCTL\_SYSDIV\_5|SYSCTL\_USE\_PLL|SYSCTL\_XTAL\_16MHZ|SYSCTL\_OSC\_MAIN); //set the main clock to runat 40MHz

InitConsole();

InitI2C0();

initADC();

**UARTprintf**("Midterm 1\n");

**UARTprintf**("I2C ->\n");

**UARTprintf**(" Mode: I2C\n");

**while**(1) {

MPU6050\_imitator();

**SysCtlDelay**(40000000 / 3);

}

//MPU6050Example();

**return** 0;

}

**------------------------------------------------------------------------------------**

**Task 03 and Task 04 (Task 03 only asks for data in Task 04’s to be graphed:**

Youtube Link: https://youtu.be/D6KfpZpYd4w

**Modified Schematic (if applicable):**

**Modified Code:**

/\*

\* tivac\_midterm\_t03.c

\*

\* Created on: Oct 26, 2019

\* Author: gausp

\*/

**#include** <stdio.h>

**#include** <stdbool.h>

**#include** <stdint.h>

**#include** <stdlib.h>

**#include** <stdio.h>

**#include** <stdarg.h>

**#include** <stdbool.h>

**#include** "sensorlib/i2cm\_drv.h"

**#include** "sensorlib/i2cm\_drv.c"

**#include** "sensorlib/hw\_mpu6050.h"

**#include** "sensorlib/mpu6050.h"

**#include** "sensorlib/mpu6050.c"

//#include "inc/tm4c123gh6pm.h"

**#include** "inc/hw\_ints.h"

**#include** "inc/hw\_memmap.h"

**#include** "inc/hw\_sysctl.h"

**#include** "inc/hw\_types.h"

**#include** "inc/hw\_i2c.h"

**#include** "inc/hw\_types.h"

**#include** "inc/hw\_gpio.h"

**#include** "driverlib/gpio.h"

**#include** "driverlib/pin\_map.h"

**#include** "driverlib/rom.h"

**#include** "driverlib/rom\_map.h"

**#include** "driverlib/debug.h"

**#include** "driverlib/interrupt.h"

**#include** "driverlib/i2c.h"

**#include** "driverlib/sysctl.h"

**#include** "driverlib/uart.h"

**#include** "driverlib/adc.h"

**#include** "utils/uartstdio.h"

**#include** "utils/uartstdio.c"

**#include** <math.h>

**#ifndef** M\_PI //in case M\_PI is undefined, this bit of code takes care of it

**#define** M\_PI 3.14159265358979323846

**#endif**

**#define** ACCELEROMETER\_SENSITIVITY 8192.0

**#define** GYROSCOPE\_SENSITIVITY 65.536

//#define M\_PI 3.14159265359

**#define** dt 0.01 // 10 ms sample rate!

**void** **ComplementaryFilter**(**short** accData[3], **short** gyrData[3], **float** \*pitch, **float** \*roll)

{

**float** pitchAcc, rollAcc;

// Integrate the gyroscope data -> int(angularSpeed) = angle

// Angle around the X-axis

\*pitch += ((**float**)gyrData[0] / GYROSCOPE\_SENSITIVITY) \* dt;

// Angle around the Y-axis

\*roll -= ((**float**)gyrData[1] / GYROSCOPE\_SENSITIVITY) \* dt;

// Compensate for drift with accelerometer data

// Sensitivity = -2 to 2 G at 16Bit -> 2G = 32768 && 0.5G = 8192

**int** forceMagnitudeApprox = **abs**(accData[0]) + **abs**(accData[1]) + **abs**(accData[2]);

**if** (forceMagnitudeApprox > 8192 && forceMagnitudeApprox < 32768)

{

// Turning around the X axis results in a vector on the Y-axis

pitchAcc = **atan2f**((**float**)accData[1], (**float**)accData[2]) \* 180 / M\_PI;

\*pitch = \*pitch \* 0.98 + pitchAcc \* 0.02;

// Turning around the Y axis results in a vector on the X-axis

rollAcc = **atan2f**((**float**)accData[0], (**float**)accData[2]) \* 180 / M\_PI;

\*roll = \*roll \* 0.98 + rollAcc \* 0.02;

}

}

//

// A boolean that is set when a MPU6050 command has completed.

//

**volatile** bool g\_bMPU6050Done;

// I2C master instance

//

tI2CMInstance g\_sI2CMSimpleInst;

//

// The function that is provided by this example as a callback when MPU6050

// transactions have completed.

//

/\*void MPU6050Callback(void \*pvCallbackData, uint\_fast8\_t ui8Status)

{

//

// See if an error occurred.

//

if(ui8Status != I2CM\_STATUS\_SUCCESS)

{

//

// An error occurred, so handle it here if required.

//

}

//

// Indicate that the MPU6050 transaction has completed.

//

g\_bMPU6050Done = true;

}

\*/

**void** **InitI2C0**(**void**)

//Configure/initialize the I2C0

{

**SysCtlPeripheralEnable** (SYSCTL\_PERIPH\_I2C0); //enables I2C0

**SysCtlPeripheralReset** (SYSCTL\_PERIPH\_I2C0); //reset module

**SysCtlPeripheralEnable** (SYSCTL\_PERIPH\_GPIOB); //enable PORTB as peripheral

//Configure the pin muxing for I2C0 functions on port B2 and B3

**GPIOPinTypeI2C** (GPIO\_PORTB\_BASE, GPIO\_PIN\_3); //set I2C PB3 as SDA

**GPIOPinConfigure** (GPIO\_PB3\_I2C0SDA);

**GPIOPinTypeI2CSCL** (GPIO\_PORTB\_BASE, GPIO\_PIN\_2); //set I2C PB2 as SCLK

**GPIOPinTypeI2C**(GPIO\_PORTB\_BASE, GPIO\_PIN\_3);

//Enable and initialize the I2C0 master module. Use the system clock for

//the I2C0 module. The last parameter sets the I2C data transfer rate.

//If false the data rata is set to 100kbps and if true the data rate will be

//set to 400kbps

**I2CMasterInitExpClk** (I2C0\_BASE, **SysCtlClockGet**(), false); //Set the clock of the I2C to ensure proper connection

//clear I2C FIFOs

HWREG(I2C0\_BASE + I2C\_O\_FIFOCTL) = 80008000;

}

//

// The MPU6050 example.

//

**void** **MPU6050Example**(**void**)

{

**float** fAccel[3], fGyro[3];

tI2CMInstance sI2CInst;

tMPU6050 sMPU6050;

//

// Initialize the MPU6050. This code assumes that the I2C master instance

// has already been initialized.

//

g\_bMPU6050Done = false;

**MPU6050Init**(&sMPU6050, &sI2CInst, 0x68, MPU6050Callback, 0);

**while**(!g\_bMPU6050Done)

{

}

//

// Configure the MPU6050 for +/- 4 g accelerometer range.

//

g\_bMPU6050Done = false;

**MPU6050ReadModifyWrite**(&sMPU6050, MPU6050\_O\_ACCEL\_CONFIG,

~MPU6050\_ACCEL\_CONFIG\_AFS\_SEL\_M,

MPU6050\_ACCEL\_CONFIG\_AFS\_SEL\_4G, MPU6050Callback,

0);

**while**(!g\_bMPU6050Done)

{

}

//

// Loop forever reading data from the MPU6050. Typically, this process

// would be done in the background, but for the purposes of this example,

// it is shown in an infinite loop.

//

**while**(1)

{

//

// Request another reading from the MPU6050.

//

g\_bMPU6050Done = false;

**MPU6050DataRead**(&sMPU6050, MPU6050Callback, 0);

**while**(!g\_bMPU6050Done)

{

}

//

// Get the new accelerometer and gyroscope readings.

//

**MPU6050DataAccelGetFloat**(&sMPU6050, &fAccel[0], &fAccel[1],

&fAccel[2]);

**MPU6050DataGyroGetFloat**(&sMPU6050, &fGyro[0], &fGyro[1], &fGyro[2]);

//

// Do something with the new accelerometer and gyroscope readings.

//

}

}

**void**

**InitConsole**(**void**)

{

//

// Enable GPIO port A which is used for UART0 pins.

// **TODO**: change this to whichever GPIO port you are using.

//

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_GPIOA);

//

// Configure the pin muxing for UART0 functions on port A0 and A1.

// This step is not necessary if your part does not support pin muxing.

// **TODO**: change this to select the port/pin you are using.

//

**GPIOPinConfigure**(GPIO\_PA0\_U0RX);

**GPIOPinConfigure**(GPIO\_PA1\_U0TX);

//

// Enable UART0 so that we can configure the clock.

//

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_UART0);

//

// Use the internal 16MHz oscillator as the UART clock source.

//

**UARTClockSourceSet**(UART0\_BASE, UART\_CLOCK\_PIOSC);

//

// Select the alternate (UART) function for these pins.

// **TODO**: change this to select the port/pin you are using.

//

**GPIOPinTypeUART**(GPIO\_PORTA\_BASE, GPIO\_PIN\_0 | GPIO\_PIN\_1);

//

// Initialize the UART for console I/O.

//

**UARTStdioConfig**(0, 115200, 16000000);

}

**void** **initADC**() {

// The ADC0 peripheral must be enabled for use.

//

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_ADC0);

**SysCtlDelay**(3);

//

// Enable sample sequence 3 with a processor signal trigger. Sequence 3

// will do a single sample when the processor sends a singal to start the

// conversion. Each ADC module has 4 programmable sequences, sequence 0

// to sequence 3. This example is arbitrarily using sequence 3.

//

**ADCSequenceConfigure**(ADC0\_BASE, 3, ADC\_TRIGGER\_PROCESSOR, 0);

//

// Configure step 0 on sequence 3. Sample the temperature sensor

// (ADC\_CTL\_TS) and configure the interrupt flag (ADC\_CTL\_IE) to be set

// when the sample is done. Tell the ADC logic that this is the last

// conversion on sequence 3 (ADC\_CTL\_END). Sequence 3 has only one

// programmable step. Sequence 1 and 2 have 4 steps, and sequence 0 has

// 8 programmable steps. Since we are only doing a single conversion using

// sequence 3 we will only configure step 0. For more information on the

// ADC sequences and steps, reference the datasheet.

//

**ADCSequenceStepConfigure**(ADC0\_BASE, 3, 0, ADC\_CTL\_TS | ADC\_CTL\_IE |

ADC\_CTL\_END);

//

// Since sample sequence 3 is now configured, it must be enabled.

//

**ADCSequenceEnable**(ADC0\_BASE, 3);

//

// Clear the interrupt status flag. This is done to make sure the

// interrupt flag is cleared before we sample.

//

**ADCIntClear**(ADC0\_BASE, 3);

}

uint32\_t **getADCtempf**(**void**) {

// This array is used for storing the data read from the ADC FIFO. It

// must be as large as the FIFO for the sequencer in use. This example

// uses sequence 3 which has a FIFO depth of 1. If another sequence

// was used with a deeper FIFO, then the array size must be changed.

//

uint32\_t ADCValues[1];

//

// These variables are used to store the temperature conversions for

// Celsius and Fahrenheit.

//

uint32\_t TempValueC ;

uint32\_t TempValueF ;

//

// Trigger the ADC conversion.

//

**ADCProcessorTrigger**(ADC0\_BASE, 3);

//

// Wait for conversion to be completed.

//

**while**(!**ADCIntStatus**(ADC0\_BASE, 3, false))

{

}

//

// Clear the ADC interrupt flag.

//

**ADCIntClear**(ADC0\_BASE, 3);

//

// Read ADC Value.

//

**ADCSequenceDataGet**(ADC0\_BASE, 3, ADCValues);

//

// Use non-calibrated conversion provided in the data sheet. I use floats in intermediate

// math but you could use intergers with multiplied by powers of 10 and divide on the end

// Make sure you divide last to avoid dropout.

//

TempValueC = (uint32\_t)(147.5 - ((75.0\*3.3 \*(**float**)ADCValues[0])) / 4096.0);

//

// Get Fahrenheit value. Make sure you divide last to avoid dropout.

//

TempValueF = ((TempValueC \* 9) + 160) / 5;

//

// This function provides a means of generating a constant length

// delay. The function delay (in cycles) = 3 \* parameter. Delay

// 250ms arbitrarily.

//

**SysCtlDelay**(80000000 / 12);

**return** TempValueF;

}

**void** **MPU6050\_imitator**(**void**) { //Will send values to serial terminal acting as Accel and Gyro values

**float** Ax, Ay, Az, Gx, Gy, Gz;

**short** fAccel[3], fGyro[3];

**float** pitch, roll;

uint32\_t i32IntegerPart, i32FractionPart;

**int** i = getADCtempf();

Ax = i;

Gx = 5\*(Ax/8);

Ay = Ax/2;

Gy = 11\*(Ax/8);

Az = Ax\*2;

Gz = 7\*(Ax/8);

fAccel[0] = Ax;

fAccel[1] = Ay;

fAccel[2] = Az;

fGyro[0] = Gx;

fGyro[1] = Gy;

fGyro[2] = Gz;

**UARTprintf**("\nRaw Values:");

i32IntegerPart = (int32\_t)Ax;

i32FractionPart = (int32\_t)(Ax \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("\nAx %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Ay;

i32FractionPart = (int32\_t)(Ay \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Ay %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Az;

i32FractionPart = (int32\_t)(Az \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Az %d.%d\n", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gx;

i32FractionPart = (int32\_t)(Gx \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gx %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gy;

i32FractionPart = (int32\_t)(Gy \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gy %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gz;

i32FractionPart = (int32\_t)(Gz\* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gz %d.%d\n", i32IntegerPart, i32FractionPart);

ComplementaryFilter(fAccel, fGyro, &pitch, &roll);

Ax = fAccel[0];

Ay = fAccel[1];

Az = fAccel[2];

Gx = fGyro[0];

Gy = fGyro[1];

Gz = fGyro[2];

**UARTprintf**("\nFiltered Values:");

i32IntegerPart = (int32\_t)Ax;

i32FractionPart = (int32\_t)(Ax \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("\nAx %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Ay;

i32FractionPart = (int32\_t)(Ay \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Ay %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Az;

i32FractionPart = (int32\_t)(Az \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Az %d.%d\n", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gx;

i32FractionPart = (int32\_t)(Gx \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gx %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gy;

i32FractionPart = (int32\_t)(Gy \* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gy %d.%d ", i32IntegerPart, i32FractionPart);

i32IntegerPart = (int32\_t)Gz;

i32FractionPart = (int32\_t)(Gz\* 1000.0f);

i32FractionPart = i32FractionPart - (i32IntegerPart \* 1000);

**UARTprintf**("Gz %d.%d\n", i32IntegerPart, i32FractionPart);

}

**int** **main**(**void**)

{

**SysCtlClockSet**(SYSCTL\_SYSDIV\_5|SYSCTL\_USE\_PLL|SYSCTL\_XTAL\_16MHZ|SYSCTL\_OSC\_MAIN); //set the main clock to runat 40MHz

InitConsole();

InitI2C0();

initADC();

**UARTprintf**("Midterm 1\n");

**UARTprintf**("I2C ->\n");

**UARTprintf**(" Mode: I2C\n");

**while**(1) {

MPU6050\_imitator();

**SysCtlDelay**(40000000 / 3);

}

//MPU6050Example();

**return** 0;

}

**------------------------------------------------------------------------------------**