```
/* MPU6050 Basic Example with IMU
 by: Kris Winer
 date: May 10, 2014
 license: Beerware - Use this code however you'd like. If you
 find it useful you can buy me a beer some time.

 Demonstrate  MPU-6050 basic functionality including initialization, accelerometer trimming, sleep mode
functionality as well as
 parameterizing the register addresses. Added display functions to allow display to on breadboard
monitor.
 No DMP use. We just want to get out the accelerations, temperature, and gyro readings.

 SDA and SCL should have external pull-up resistors (to 3.3V).
 10k resistors worked for me. They should be on the breakout
 board.

 Hardware setup:
 MPU6050 Breakout --------- Arduino
 3.3V --------------------- 3.3V
 SDA ----------------------- A4
 SCL ----------------------- A5
 GND ---------------------- GND

  Note: The MPU6050 is an I2C sensor and uses the Arduino Wire library.
 Because the sensor is not 5V tolerant, we are using a 3.3 V 8 MHz Pro Mini or a 3.3 V Teensy 3.1.
 We have disabled the internal pull-ups used by the Wire library in the Wire.h/twi.c utility file.
 We are also using the 400 kHz fast I2C mode by setting the TWI_FREQ  to 400000L /twi.h utility file.
 */

#include "SPI.h"
#include <Wire.h>
//#include <Adafruit_GFX.h>
#include "ILI9341_t3.h"

// For the Adafruit shield, these are the default.
#define TFT_DC 9
#define TFT_CS 10

// Use hardware SPI (on Uno, #13, #12, #11) and the above for CS/DC
ILI9341_t3 tft = ILI9341_t3(TFT_CS, TFT_DC);

// Define registers per MPU6050, Register Map and Descriptions, Rev 4.2, 08/19/2013 6 DOF Motion sensor
fusion device
// Invensense Inc., www.invensense.com
// See also MPU-6050 Register Map and Descriptions, Revision 4.0, RM-MPU-6050A-00, 9/12/2012 for
registers not listed in
// above document; the MPU6050 and MPU 9150 are virtually identical but the latter has an on-board
magnetic sensor
//
#define XGOFFS_TC        0x00 // Bit 7 PWR_MODE, bits 6:1 XG_OFFS_TC, bit 0 OTP_BNK_VLD
#define YGOFFS_TC        0x01
#define ZGOFFS_TC        0x02
#define X_FINE_GAIN      0x03 // [7:0] fine gain
#define Y_FINE_GAIN      0x04
#define Z_FINE_GAIN      0x05
#define XA_OFFSET_H      0x06 // User-defined trim values for accelerometer
#define XA_OFFSET_L_TC   0x07
#define YA_OFFSET_H      0x08
#define YA_OFFSET_L_TC   0x09
#define ZA_OFFSET_H      0x0A
#define ZA_OFFSET_L_TC   0x0B
#define SELF_TEST_X      0x0D
#define SELF_TEST_Y      0x0E
#define SELF_TEST_Z      0x0F
#define SELF_TEST_A      0x10
#define XG_OFFS_USRH     0x13  // User-defined trim values for gyroscope; supported in MPU-6050?
#define XG_OFFS_USRL     0x14
#define YG_OFFS_USRH     0x15
#define YG_OFFS_USRL     0x16
#define ZG_OFFS_USRH     0x17
#define ZG_OFFS_USRL     0x18
#define SMPLRT_DIV       0x19
```

```
#define CONFIG           0x1A
#define GYRO_CONFIG      0x1B
#define ACCEL_CONFIG     0x1C
#define FF_THR           0x1D  // Free-fall
#define FF_DUR           0x1E  // Free-fall
#define MOT_THR          0x1F  // Motion detection threshold bits [7:0]
#define MOT_DUR          0x20  // Duration counter threshold for motion interrupt generation, 1 kHz
rate, LSB = 1 ms
#define ZMOT_THR         0x21  // Zero-motion detection threshold bits [7:0]
#define ZRMOT_DUR        0x22  // Duration counter threshold for zero motion interrupt generation, 16
Hz rate, LSB = 64 ms
#define FIFO_EN          0x23
#define I2C_MST_CTRL     0x24
#define I2C_SLV0_ADDR    0x25
#define I2C_SLV0_REG     0x26
#define I2C_SLV0_CTRL    0x27
#define I2C_SLV1_ADDR    0x28
#define I2C_SLV1_REG     0x29
#define I2C_SLV1_CTRL    0x2A
#define I2C_SLV2_ADDR    0x2B
#define I2C_SLV2_REG     0x2C
#define I2C_SLV2_CTRL    0x2D
#define I2C_SLV3_ADDR    0x2E
#define I2C_SLV3_REG     0x2F
#define I2C_SLV3_CTRL    0x30
#define I2C_SLV4_ADDR    0x31
#define I2C_SLV4_REG     0x32
#define I2C_SLV4_DO      0x33
#define I2C_SLV4_CTRL    0x34
#define I2C_SLV4_DI      0x35
#define I2C_MST_STATUS   0x36
#define INT_PIN_CFG      0x37
#define INT_ENABLE       0x38
#define DMP_INT_STATUS   0x39  // Check DMP interrupt
#define INT_STATUS       0x3A
#define ACCEL_XOUT_H     0x3B
#define ACCEL_XOUT_L     0x3C
#define ACCEL_YOUT_H     0x3D
#define ACCEL_YOUT_L     0x3E
#define ACCEL_ZOUT_H     0x3F
#define ACCEL_ZOUT_L     0x40
#define TEMP_OUT_H       0x41
#define TEMP_OUT_L       0x42
#define GYRO_XOUT_H      0x43
#define GYRO_XOUT_L      0x44
#define GYRO_YOUT_H      0x45
#define GYRO_YOUT_L      0x46
#define GYRO_ZOUT_H      0x47
#define GYRO_ZOUT_L      0x48
#define EXT_SENS_DATA_00 0x49
#define EXT_SENS_DATA_01 0x4A
#define EXT_SENS_DATA_02 0x4B
#define EXT_SENS_DATA_03 0x4C
#define EXT_SENS_DATA_04 0x4D
#define EXT_SENS_DATA_05 0x4E
#define EXT_SENS_DATA_06 0x4F
#define EXT_SENS_DATA_07 0x50
#define EXT_SENS_DATA_08 0x51
#define EXT_SENS_DATA_09 0x52
#define EXT_SENS_DATA_10 0x53
#define EXT_SENS_DATA_11 0x54
#define EXT_SENS_DATA_12 0x55
#define EXT_SENS_DATA_13 0x56
#define EXT_SENS_DATA_14 0x57
#define EXT_SENS_DATA_15 0x58
#define EXT_SENS_DATA_16 0x59
#define EXT_SENS_DATA_17 0x5A
#define EXT_SENS_DATA_18 0x5B
#define EXT_SENS_DATA_19 0x5C
#define EXT_SENS_DATA_20 0x5D
#define EXT_SENS_DATA_21 0x5E
#define EXT_SENS_DATA_22 0x5F
#define EXT_SENS_DATA_23 0x60
```

```
#define MOT_DETECT_STATUS 0x61
#define I2C_SLV0_DO      0x63
#define I2C_SLV1_DO      0x64
#define I2C_SLV2_DO      0x65
#define I2C_SLV3_DO      0x66
#define I2C_MST_DELAY_CTRL 0x67
#define SIGNAL_PATH_RESET  0x68
#define MOT_DETECT_CTRL   0x69
#define USER_CTRL        0x6A  // Bit 7 enable DMP, bit 3 reset DMP
#define PWR_MGMT_1       0x6B // Device defaults to the SLEEP mode
#define PWR_MGMT_2       0x6C
#define DMP_BANK         0x6D  // Activates a specific bank in the DMP
#define DMP_RW_PNT       0x6E  // Set read/write pointer to a specific start address in specified DMP
bank
#define DMP_REG          0x6F  // Register in DMP from which to read or to which to write
#define DMP_REG_1        0x70
#define DMP_REG_2        0x71
#define FIFO_COUNTH      0x72
#define FIFO_COUNTL      0x73
#define FIFO_R_W         0x74
#define WHO_AM_I_MPU6050 0x75 // Should return 0x68

// Using the GY-521 breakout board, I set ADO to 0 by grounding through a 4k7 resistor
// Seven-bit device address is 110100 for ADO = 0 and 110101 for ADO = 1
#define ADO 0
#if ADO
#define MPU6050_ADDRESS 0x69  // Device address when ADO = 1
#else
#define MPU6050_ADDRESS 0x68  // Device address when ADO = 0
#endif

// Set initial input parameters
enum Ascale {
  AFS_2G = 0,
  AFS_4G,
  AFS_8G,
  AFS_16G
};

enum Gscale {
  GFS_250DPS = 0,
  GFS_500DPS,
  GFS_1000DPS,
  GFS_2000DPS
};

// Specify sensor full scale
int Gscale = GFS_250DPS;
int Ascale = AFS_2G;
float aRes, gRes; // scale resolutions per LSB for the sensors

// Pin definitions
int intPin = 12;  // These can be changed, 2 and 3 are the Arduinos ext int pins
#define blinkPin 13  // Blink LED on Teensy or Pro Mini when updating
boolean blinkOn = false;

int16_t accelCount[3];  // Stores the 16-bit signed accelerometer sensor output
float ax, ay, az;       // Stores the real accel value in g's
int16_t gyroCount[3];   // Stores the 16-bit signed gyro sensor output
float gx, gy, gz;       // Stores the real gyro value in degrees per seconds
float gyroBias[3] = {0, 0, 0}, accelBias[3] = {0, 0, 0}; // Bias corrections for gyro and accelerometer
int16_t tempCount;   // Stores the real internal chip temperature in degrees Celsius
float temperature;
float SelfTest[6];

uint32_t delt_t = 0; // used to control display output rate
uint32_t count = 0;  // used to control display output rate

// parameters for 6 DoF sensor fusion calculations
float GyroMeasError = PI * (40.0f / 180.0f);     // gyroscope measurement error in rads/s (start at 60
deg/s), then reduce after ~10 s to 3
float beta = sqrt(3.0f / 4.0f) * GyroMeasError;  // compute beta
float GyroMeasDrift = PI * (2.0f / 180.0f);      // gyroscope measurement drift in rad/s/s (start at
```

```
0.0 deg/s/s)
float zeta = sqrt(3.0f / 4.0f) * GyroMeasDrift;  // compute zeta, the other free parameter in the
Madgwick scheme usually set to a small or zero value
float pitch, yaw, roll;
float deltat = 0.0f;                              // integration interval for both filter schemes
uint32_t lastUpdate = 0, firstUpdate = 0;         // used to calculate integration interval
uint32_t Now = 0;                                 // used to calculate integration interval
float q[4] = {1.0f, 0.0f, 0.0f, 0.0f};            // vector to hold quaternion

void setup()
{
  Wire.begin();
  Serial.begin(38400);

  // Set up the interrupt pin, its set as active high, push-pull
  pinMode(intPin, INPUT);
  digitalWrite(intPin, LOW);
  pinMode(blinkPin, OUTPUT);
  digitalWrite(blinkPin, HIGH);

  tft.begin(); // Initialize the display

  tft.setRotation(2); //  0 or 2) width = width, 1 or 3) width = height, swapped etc.


// Start device display with ID of sensor
  tft.fillScreen(ILI9341_WHITE);
  tft.setTextColor(ILI9341_BLACK); // Set pixel color; 1 on the monochrome screen
  tft.setTextSize(2);
  tft.setCursor(40,0); tft.print("MPU6050");
  tft.setCursor(0, 40); tft.print("6-DOF 16-bit");
  tft.setCursor(0, 60); tft.print("motion sensor");
  tft.setCursor(40,80); tft.print("60 ug LSB");

  delay(1000);

// Set up for data display
  tft.setTextSize(1); // Set text size to normal, 2 is twice normal etc.
  tft.setTextColor(ILI9341_BLACK); // Set pixel color; 1 on the monochrome screen
  tft.fillScreen(ILI9341_WHITE);   // clears the screen and buffer

  // Read the WHO_AM_I register, this is a good test of communication
  uint8_t c = readByte(MPU6050_ADDRESS, WHO_AM_I_MPU6050);  // Read WHO_AM_I register for MPU-6050
  tft.setCursor(20,0); tft.print("MPU6050");
  tft.setCursor(0,10); tft.print("I AM");
  tft.setCursor(0,20); tft.print(c, HEX);
  tft.setCursor(0,30); tft.print("I Should Be");
  tft.setCursor(0,40); tft.print(0x68, HEX);

  delay(1000);

  if (c == 0x68) // WHO_AM_I should always be 0x68
  {
    Serial.println("MPU6050 is online...");

    MPU6050SelfTest(SelfTest); // Start by performing self test and reporting values
    Serial.print("x-axis self test: acceleration trim within : "); Serial.print(SelfTest[0],1);
Serial.println("% of factory value");
    Serial.print("y-axis self test: acceleration trim within : "); Serial.print(SelfTest[1],1);
Serial.println("% of factory value");
    Serial.print("z-axis self test: acceleration trim within : "); Serial.print(SelfTest[2],1);
Serial.println("% of factory value");
    Serial.print("x-axis self test: gyration trim within : "); Serial.print(SelfTest[3],1);
Serial.println("% of factory value");
    Serial.print("y-axis self test: gyration trim within : "); Serial.print(SelfTest[4],1);
Serial.println("% of factory value");
    Serial.print("z-axis self test: gyration trim within : "); Serial.print(SelfTest[5],1);
Serial.println("% of factory value");

    if(SelfTest[0] < 1.0f && SelfTest[1] < 1.0f && SelfTest[2] < 1.0f && SelfTest[3] < 1.0f &&
SelfTest[4] < 1.0f && SelfTest[5] < 1.0f) {
    tft.fillScreen(ILI9341_WHITE);
    tft.setCursor(0, 30); tft.print("Pass Selftest!");
```

```
    delay(1000);

    calibrateMPU6050(gyroBias, accelBias); // Calibrate gyro and accelerometers, load biases in bias
registers
  tft.fillScreen(ILI9341_WHITE);

  tft.setCursor(20, 0); tft.print("MPU6050 bias");
  tft.setCursor(0, 8); tft.print(" x   y   z  ");

  tft.setCursor(0,  16); tft.print((int)(1000*accelBias[0]));
  tft.setCursor(24, 16); tft.print((int)(1000*accelBias[1]));
  tft.setCursor(48, 16); tft.print((int)(1000*accelBias[2]));
  tft.setCursor(72, 16); tft.print("mg");

  tft.setCursor(0,  24); tft.print(gyroBias[0], 1);
  tft.setCursor(24, 24); tft.print(gyroBias[1], 1);
  tft.setCursor(48, 24); tft.print(gyroBias[2], 1);
  tft.setCursor(66, 24); tft.print("o/s");

  delay(1000);

   initMPU6050(); Serial.println("MPU6050 initialized for active data mode...."); // Initialize device
for active mode read of acclerometer, gyroscope, and temperature
   }
   else
   {
    Serial.print("Could not connect to MPU6050: 0x");
    Serial.println(c, HEX);
    while(1) ; // Loop forever if communication doesn't happen
   }
  }
}

void loop()
{
   // If data ready bit set, all data registers have new data
  if(readByte(MPU6050_ADDRESS, INT_STATUS) & 0x01) {  // check if data ready interrupt
    readAccelData(accelCount);  // Read the x/y/z adc values
    getAres();

    // Now we'll calculate the accleration value into actual g's
    ax = (float)accelCount[0]*aRes;  // get actual g value, this depends on scale being set
    ay = (float)accelCount[1]*aRes;
    az = (float)accelCount[2]*aRes;

    readGyroData(gyroCount);  // Read the x/y/z adc values
    getGres();

    // Calculate the gyro value into actual degrees per second
    gx = (float)gyroCount[0]*gRes;  // get actual gyro value, this depends on scale being set
    gy = (float)gyroCount[1]*gRes;
    gz = (float)gyroCount[2]*gRes;

    tempCount = readTempData();  // Read the x/y/z adc values
    temperature = ((float) tempCount) / 340. + 36.53; // Temperature in degrees Centigrade
   }

    Now = micros();
    deltat = ((Now - lastUpdate)/1000000.0f); // set integration time by time elapsed since last filter
update
    lastUpdate = Now;
//    if(lastUpdate - firstUpdate > 10000000uL) {
//      beta = 0.041; // decrease filter gain after stabilized
//      zeta = 0.015; // increase gyro bias drift gain after stabilized
//    }
   // Pass gyro rate as rad/s
    MadgwickQuaternionUpdate(ax, ay, az, gx*PI/180.0f, gy*PI/180.0f, gz*PI/180.0f);

    // Serial print and/or display at 0.5 s rate independent of data rates
    delt_t = millis() - count;
    if (delt_t > 500) { // update LCD once per half-second independent of read rate
    digitalWrite(blinkPin, blinkOn);
```

```
/*
    Serial.print("ax = "); Serial.print((int)1000*ax);
    Serial.print(" ay = "); Serial.print((int)1000*ay);
    Serial.print(" az = "); Serial.print((int)1000*az); Serial.println(" mg");

    Serial.print("gx = "); Serial.print( gx, 1);
    Serial.print(" gy = "); Serial.print( gy, 1);
    Serial.print(" gz = "); Serial.print( gz, 1); Serial.println(" deg/s");

    Serial.print("q0 = "); Serial.print(q[0]);
    Serial.print(" qx = "); Serial.print(q[1]);
    Serial.print(" qy = "); Serial.print(q[2]);
    Serial.print(" qz = "); Serial.println(q[3]);
*/
  // Define output variables from updated quaternion---these are Tait-Bryan angles, commonly used in
aircraft orientation.
  // In this coordinate system, the positive z-axis is down toward Earth.
  // Yaw is the angle between Sensor x-axis and Earth magnetic North (or true North if corrected for
local declination, looking down on the sensor positive yaw is counterclockwise.
  // Pitch is angle between sensor x-axis and Earth ground plane, toward the Earth is positive, up
toward the sky is negative.
  // Roll is angle between sensor y-axis and Earth ground plane, y-axis up is positive roll.
  // These arise from the definition of the homogeneous rotation matrix constructed from quaternions.
  // Tait-Bryan angles as well as Euler angles are non-commutative; that is, the get the correct
orientation the rotations must be
  // applied in the correct order which for this configuration is yaw, pitch, and then roll.
  // For more see http://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles which
has additional links.
    yaw   = atan2(2.0f * (q[1] * q[2] + q[0] * q[3]), q[0] * q[0] + q[1] * q[1] - q[2] * q[2] - q[3] *
q[3]);
    pitch = -asin(2.0f * (q[1] * q[3] - q[0] * q[2]));
    roll  = atan2(2.0f * (q[0] * q[1] + q[2] * q[3]), q[0] * q[0] - q[1] * q[1] - q[2] * q[2] + q[3] *
q[3]);
    pitch *= 180.0f / PI;
    yaw   *= 180.0f / PI;
    roll  *= 180.0f / PI;

//    Serial.print("Yaw, Pitch, Roll: ");

    Serial.print(yaw, 2);
    Serial.print(", ");
    Serial.print(pitch, 2);
    Serial.print(", ");
    Serial.println(roll, 2);


//    Serial.print("average rate = "); Serial.print(1.0f/deltat, 2); Serial.println(" Hz");

    tft.fillScreen(ILI9341_WHITE);
    tft.setTextSize(2);
    tft.setCursor(0, 25); tft.print(" x   y    z  ");

    tft.setCursor(0,  45); tft.print((int16_t)(1000*ax));
    tft.setCursor(60, 45); tft.print((int16_t)(1000*ay));
    tft.setCursor(120, 45); tft.print((int16_t)(1000*az));
    tft.setCursor(192, 45); tft.print("mg");

    tft.setCursor(0,  65); tft.print((int16_t)(gx));
    tft.setCursor(60, 65); tft.print((int16_t)(gy));
    tft.setCursor(120, 65); tft.print((int16_t)(gz));
    tft.setCursor(192, 65); tft.print("o/s");

    tft.setCursor(0,  95); tft.print((int)(yaw));
    tft.setCursor(60, 95); tft.print((int)(pitch));
    tft.setCursor(120, 95); tft.print((int)(roll));
    tft.setCursor(192, 95); tft.print("ypr");

    tft.setCursor(0, 135); tft.print("rt: "); tft.print(1.0f/deltat, 2); tft.print(" Hz");


    blinkOn = ~blinkOn;
    count = millis();
}
```

```
  }

//===================================================================================================================
//====== Set of useful function to access acceleratio, gyroscope, and temperature data
//===================================================================================================================

void getGres() {
  switch (Gscale)
  {
        // Possible gyro scales (and their register bit settings) are:
        // 250 DPS (00), 500 DPS (01), 1000 DPS (10), and 2000 DPS  (11).
        // Here's a bit of an algorith to calculate DPS/(ADC tick) based on that 2-bit value:
    case GFS_250DPS:
          gRes = 250.0/32768.0;
          break;
    case GFS_500DPS:
          gRes = 500.0/32768.0;
          break;
    case GFS_1000DPS:
          gRes = 1000.0/32768.0;
          break;
    case GFS_2000DPS:
          gRes = 2000.0/32768.0;
          break;
  }
}

void getAres() {
  switch (Ascale)
  {
        // Possible accelerometer scales (and their register bit settings) are:
        // 2 Gs (00), 4 Gs (01), 8 Gs (10), and 16 Gs  (11).
        // Here's a bit of an algorith to calculate DPS/(ADC tick) based on that 2-bit value:
    case AFS_2G:
          aRes = 2.0/32768.0;
          break;
    case AFS_4G:
          aRes = 4.0/32768.0;
          break;
    case AFS_8G:
          aRes = 8.0/32768.0;
          break;
    case AFS_16G:
          aRes = 16.0/32768.0;
          break;
  }
}


void readAccelData(int16_t * destination)
{
  uint8_t rawData[6];  // x/y/z accel register data stored here
  readBytes(MPU6050_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);  // Read the six raw data registers into data array
  destination[0] = (int16_t)((rawData[0] << 8) | rawData[1]) ;  // Turn the MSB and LSB into a signed 16-bit value
  destination[1] = (int16_t)((rawData[2] << 8) | rawData[3]) ;
  destination[2] = (int16_t)((rawData[4] << 8) | rawData[5]) ;
}

void readGyroData(int16_t * destination)
{
  uint8_t rawData[6];  // x/y/z gyro register data stored here
  readBytes(MPU6050_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);  // Read the six raw data registers sequentially into data array
  destination[0] = (int16_t)((rawData[0] << 8) | rawData[1]) ;  // Turn the MSB and LSB into a signed 16-bit value
  destination[1] = (int16_t)((rawData[2] << 8) | rawData[3]) ;
  destination[2] = (int16_t)((rawData[4] << 8) | rawData[5]) ;
}
```

```
int16_t readTempData()
{
  uint8_t rawData[2];  // x/y/z gyro register data stored here
  readBytes(MPU6050_ADDRESS, TEMP_OUT_H, 2, &rawData[0]);  // Read the two raw data registers
sequentially into data array
  return ((int16_t)rawData[0]) << 8 | rawData[1] ;  // Turn the MSB and LSB into a 16-bit value
}



// Configure the motion detection control for low power accelerometer mode
void LowPowerAccelOnlyMPU6050()
{

// The sensor has a high-pass filter necessary to invoke to allow the sensor motion detection
algorithms work properly
// Motion detection occurs on free-fall (acceleration below a threshold for some time for all axes),
motion (acceleration
// above a threshold for some time on at least one axis), and zero-motion toggle (acceleration on each
axis less than a
// threshold for some time sets this flag, motion above the threshold turns it off). The high-pass
filter takes gravity out
// consideration for these threshold evaluations; otherwise, the flags would be set all the time!

  uint8_t c = readByte(MPU6050_ADDRESS, PWR_MGMT_1);
  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, c & ~0x30); // Clear sleep and cycle bits [5:6]
  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, c |  0x30); // Set sleep and cycle bits [5:6] to zero to make
sure accelerometer is running

  c = readByte(MPU6050_ADDRESS, PWR_MGMT_2);
  writeByte(MPU6050_ADDRESS, PWR_MGMT_2, c & ~0x38); // Clear standby XA, YA, and ZA bits [3:5]
  writeByte(MPU6050_ADDRESS, PWR_MGMT_2, c |  0x00); // Set XA, YA, and ZA bits [3:5] to zero to make
sure accelerometer is running

  c = readByte(MPU6050_ADDRESS, ACCEL_CONFIG);
  writeByte(MPU6050_ADDRESS, ACCEL_CONFIG, c & ~0x07); // Clear high-pass filter bits [2:0]
// Set high-pass filter to 0) reset (disable), 1) 5 Hz, 2) 2.5 Hz, 3) 1.25 Hz, 4) 0.63 Hz, or 7) Hold
  writeByte(MPU6050_ADDRESS, ACCEL_CONFIG,  c | 0x00);  // Set ACCEL_HPF to 0; reset mode disbaling
high-pass filter

  c = readByte(MPU6050_ADDRESS, CONFIG);
  writeByte(MPU6050_ADDRESS, CONFIG, c & ~0x07); // Clear low-pass filter bits [2:0]
  writeByte(MPU6050_ADDRESS, CONFIG, c |  0x00);  // Set DLPD_CFG to 0; 260 Hz bandwidth, 1 kHz rate

  c = readByte(MPU6050_ADDRESS, INT_ENABLE);
  writeByte(MPU6050_ADDRESS, INT_ENABLE, c & ~0xFF);  // Clear all interrupts
  writeByte(MPU6050_ADDRESS, INT_ENABLE, 0x40);  // Enable motion threshold (bits 5) interrupt only

// Motion detection interrupt requires the absolute value of any axis to lie above the detection
threshold
// for at least the counter duration
  writeByte(MPU6050_ADDRESS, MOT_THR, 0x80); // Set motion detection to 0.256 g; LSB = 2 mg
  writeByte(MPU6050_ADDRESS, MOT_DUR, 0x01); // Set motion detect duration to 1  ms; LSB is 1 ms @ 1
kHz rate

  delay (100);  // Add delay for accumulation of samples

  c = readByte(MPU6050_ADDRESS, ACCEL_CONFIG);
  writeByte(MPU6050_ADDRESS, ACCEL_CONFIG, c & ~0x07); // Clear high-pass filter bits [2:0]
  writeByte(MPU6050_ADDRESS, ACCEL_CONFIG, c |  0x07);  // Set ACCEL_HPF to 7; hold the initial
accleration value as a referance

  c = readByte(MPU6050_ADDRESS, PWR_MGMT_2);
  writeByte(MPU6050_ADDRESS, PWR_MGMT_2, c & ~0xC7); // Clear standby XA, YA, and ZA bits [3:5] and
LP_WAKE_CTRL bits [6:7]
  writeByte(MPU6050_ADDRESS, PWR_MGMT_2, c |  0x47); // Set wakeup frequency to 5 Hz, and disable XG,
YG, and ZG gyros (bits [0:2])

  c = readByte(MPU6050_ADDRESS, PWR_MGMT_1);
  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, c & ~0x20); // Clear sleep and cycle bit 5
  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, c |  0x20); // Set cycle bit 5 to begin low power
accelerometer motion interrupts
```

```
}


void initMPU6050()
{
// wake up device-don't need this here if using calibration function below
//  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, 0x00); // Clear sleep mode bit (6), enable all sensors
//  delay(100); // Delay 100 ms for PLL to get established on x-axis gyro; should check for PLL ready
interrupt

 // get stable time source
  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, 0x01);  // Set clock source to be PLL with x-axis gyroscope
reference, bits 2:0 = 001

 // Configure Gyro and Accelerometer
 // Disable FSYNC and set accelerometer and gyro bandwidth to 44 and 42 Hz, respectively;
 // DLPF_CFG = bits 2:0 = 010; this sets the sample rate at 1 kHz for both
 // Maximum delay time is 4.9 ms corresponding to just over 200 Hz sample rate
  writeByte(MPU6050_ADDRESS, CONFIG, 0x03);

 // Set sample rate = gyroscope output rate/(1 + SMPLRT_DIV)
  writeByte(MPU6050_ADDRESS, SMPLRT_DIV, 0x04);  // Use a 200 Hz rate; the same rate set in CONFIG
above

 // Set gyroscope full scale range
 // Range selects FS_SEL and AFS_SEL are 0 - 3, so 2-bit values are left-shifted into positions 4:3
  uint8_t c =  readByte(MPU6050_ADDRESS, GYRO_CONFIG);
  writeByte(MPU6050_ADDRESS, GYRO_CONFIG, c & ~0xE0); // Clear self-test bits [7:5]
  writeByte(MPU6050_ADDRESS, GYRO_CONFIG, c & ~0x18); // Clear AFS bits [4:3]
  writeByte(MPU6050_ADDRESS, GYRO_CONFIG, c | Gscale << 3); // Set full scale range for the gyro

 // Set accelerometer configuration
  c =  readByte(MPU6050_ADDRESS, ACCEL_CONFIG);
  writeByte(MPU6050_ADDRESS, ACCEL_CONFIG, c & ~0xE0); // Clear self-test bits [7:5]
  writeByte(MPU6050_ADDRESS, ACCEL_CONFIG, c & ~0x18); // Clear AFS bits [4:3]
  writeByte(MPU6050_ADDRESS, ACCEL_CONFIG, c | Ascale << 3); // Set full scale range for the
accelerometer

  // Configure Interrupts and Bypass Enable
  // Set interrupt pin active high, push-pull, and clear on read of INT_STATUS, enable I2C_BYPASS_EN so
additional chips
  // can join the I2C bus and all can be controlled by the Arduino as master
   writeByte(MPU6050_ADDRESS, INT_PIN_CFG, 0x22);
   writeByte(MPU6050_ADDRESS, INT_ENABLE, 0x01);  // Enable data ready (bit 0) interrupt
}

// Function which accumulates gyro and accelerometer data after device initialization. It calculates
the average
// of the at-rest readings and then loads the resulting offsets into accelerometer and gyro bias
registers.
void calibrateMPU6050(float * dest1, float * dest2)
{
  uint8_t data[12]; // data array to hold accelerometer and gyro x, y, z, data
  uint16_t ii, packet_count, fifo_count;
  int32_t gyro_bias[3] = {0, 0, 0}, accel_bias[3] = {0, 0, 0};

// reset device, reset all registers, clear gyro and accelerometer bias registers
  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, 0x80); // Write a one to bit 7 reset bit; toggle reset device
  delay(100);

// get stable time source
// Set clock source to be PLL with x-axis gyroscope reference, bits 2:0 = 001
  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, 0x01);
  writeByte(MPU6050_ADDRESS, PWR_MGMT_2, 0x00);
  delay(200);

// Configure device for bias calculation
  writeByte(MPU6050_ADDRESS, INT_ENABLE, 0x00);   // Disable all interrupts
  writeByte(MPU6050_ADDRESS, FIFO_EN, 0x00);      // Disable FIFO
  writeByte(MPU6050_ADDRESS, PWR_MGMT_1, 0x00);   // Turn on internal clock source
  writeByte(MPU6050_ADDRESS, I2C_MST_CTRL, 0x00); // Disable I2C master
  writeByte(MPU6050_ADDRESS, USER_CTRL, 0x00);    // Disable FIFO and I2C master modes
  writeByte(MPU6050_ADDRESS, USER_CTRL, 0x0C);    // Reset FIFO and DMP
```

```
    delay(15);

  // Configure MPU6050 gyro and accelerometer for bias calculation
    writeByte(MPU6050_ADDRESS, CONFIG, 0x01);      // Set low-pass filter to 188 Hz
    writeByte(MPU6050_ADDRESS, SMPLRT_DIV, 0x00);  // Set sample rate to 1 kHz
    writeByte(MPU6050_ADDRESS, GYRO_CONFIG, 0x00);  // Set gyro full-scale to 250 degrees per second,
  maximum sensitivity
    writeByte(MPU6050_ADDRESS, ACCEL_CONFIG, 0x00); // Set accelerometer full-scale to 2 g, maximum
  sensitivity

    uint16_t  gyrosensitivity  = 131;   // = 131 LSB/degrees/sec
    uint16_t  accelsensitivity = 16384;  // = 16384 LSB/g

  // Configure FIFO to capture accelerometer and gyro data for bias calculation
    writeByte(MPU6050_ADDRESS, USER_CTRL, 0x40);   // Enable FIFO
    writeByte(MPU6050_ADDRESS, FIFO_EN, 0x78);     // Enable gyro and accelerometer sensors for FIFO
  (max size 1024 bytes in MPU-6050)
    delay(80); // accumulate 80 samples in 80 milliseconds = 960 bytes

  // At end of sample accumulation, turn off FIFO sensor read
    writeByte(MPU6050_ADDRESS, FIFO_EN, 0x00);        // Disable gyro and accelerometer sensors for FIFO
    readBytes(MPU6050_ADDRESS, FIFO_COUNTH, 2, &data[0]); // read FIFO sample count
    fifo_count = ((uint16_t)data[0] << 8) | data[1];
    packet_count = fifo_count/12;// How many sets of full gyro and accelerometer data for averaging

    for (ii = 0; ii < packet_count; ii++) {
      int16_t accel_temp[3] = {0, 0, 0}, gyro_temp[3] = {0, 0, 0};
      readBytes(MPU6050_ADDRESS, FIFO_R_W, 12, &data[0]); // read data for averaging
      accel_temp[0] = (int16_t) (((int16_t)data[0] << 8) | data[1]  ) ;  // Form signed 16-bit integer
  for each sample in FIFO
      accel_temp[1] = (int16_t) (((int16_t)data[2] << 8) | data[3]  ) ;
      accel_temp[2] = (int16_t) (((int16_t)data[4] << 8) | data[5]  ) ;
      gyro_temp[0]  = (int16_t) (((int16_t)data[6] << 8) | data[7]  ) ;
      gyro_temp[1]  = (int16_t) (((int16_t)data[8] << 8) | data[9]  ) ;
      gyro_temp[2]  = (int16_t) (((int16_t)data[10] << 8) | data[11]) ;

      accel_bias[0] += (int32_t) accel_temp[0]; // Sum individual signed 16-bit biases to get accumulated
  signed 32-bit biases
      accel_bias[1] += (int32_t) accel_temp[1];
      accel_bias[2] += (int32_t) accel_temp[2];
      gyro_bias[0]  += (int32_t) gyro_temp[0];
      gyro_bias[1]  += (int32_t) gyro_temp[1];
      gyro_bias[2]  += (int32_t) gyro_temp[2];

  }
      accel_bias[0] /= (int32_t) packet_count; // Normalize sums to get average count biases
      accel_bias[1] /= (int32_t) packet_count;
      accel_bias[2] /= (int32_t) packet_count;
      gyro_bias[0]  /= (int32_t) packet_count;
      gyro_bias[1]  /= (int32_t) packet_count;
      gyro_bias[2]  /= (int32_t) packet_count;

    if(accel_bias[2] > 0L) {accel_bias[2] -= (int32_t) accelsensitivity;}  // Remove gravity from the z-
  axis accelerometer bias calculation
    else {accel_bias[2] += (int32_t) accelsensitivity;}

  // Construct the gyro biases for push to the hardware gyro bias registers, which are reset to zero upon
  device startup
    data[0] = (-gyro_bias[0]/4  >> 8) & 0xFF; // Divide by 4 to get 32.9 LSB per deg/s to conform to
  expected bias input format
    data[1] = (-gyro_bias[0]/4)       & 0xFF; // Biases are additive, so change sign on calculated
  average gyro biases
    data[2] = (-gyro_bias[1]/4  >> 8) & 0xFF;
    data[3] = (-gyro_bias[1]/4)       & 0xFF;
    data[4] = (-gyro_bias[2]/4  >> 8) & 0xFF;
    data[5] = (-gyro_bias[2]/4)       & 0xFF;

  // Push gyro biases to hardware registers
    writeByte(MPU6050_ADDRESS, XG_OFFS_USRH, data[0]);// might not be supported in MPU6050
    writeByte(MPU6050_ADDRESS, XG_OFFS_USRL, data[1]);
    writeByte(MPU6050_ADDRESS, YG_OFFS_USRH, data[2]);
    writeByte(MPU6050_ADDRESS, YG_OFFS_USRL, data[3]);
    writeByte(MPU6050_ADDRESS, ZG_OFFS_USRH, data[4]);
```

```
  writeByte(MPU6050_ADDRESS, ZG_OFFS_USRL, data[5]);

  dest1[0] = (float) gyro_bias[0]/(float) gyrosensitivity; // construct gyro bias in deg/s for later
manual subtraction
  dest1[1] = (float) gyro_bias[1]/(float) gyrosensitivity;
  dest1[2] = (float) gyro_bias[2]/(float) gyrosensitivity;

// Construct the accelerometer biases for push to the hardware accelerometer bias registers. These
registers contain
// factory trim values which must be added to the calculated accelerometer biases; on boot up these
registers will hold
// non-zero values. In addition, bit 0 of the lower byte must be preserved since it is used for
temperature
// compensation calculations. Accelerometer bias registers expect bias input as 2048 LSB per g, so that
// the accelerometer biases calculated above must be divided by 8.

  int32_t accel_bias_reg[3] = {0, 0, 0}; // A place to hold the factory accelerometer trim biases
  readBytes(MPU6050_ADDRESS, XA_OFFSET_H, 2, &data[0]); // Read factory accelerometer trim values
  accel_bias_reg[0] = (int16_t) ((int16_t)data[0] << 8) | data[1];
  readBytes(MPU6050_ADDRESS, YA_OFFSET_H, 2, &data[0]);
  accel_bias_reg[1] = (int16_t) ((int16_t)data[0] << 8) | data[1];
  readBytes(MPU6050_ADDRESS, ZA_OFFSET_H, 2, &data[0]);
  accel_bias_reg[2] = (int16_t) ((int16_t)data[0] << 8) | data[1];

  uint32_t mask = 1uL; // Define mask for temperature compensation bit 0 of lower byte of accelerometer
bias registers
  uint8_t mask_bit[3] = {0, 0, 0}; // Define array to hold mask bit for each accelerometer bias axis

  for(ii = 0; ii < 3; ii++) {
    if(accel_bias_reg[ii] & mask) mask_bit[ii] = 0x01; // If temperature compensation bit is set,
record that fact in mask_bit
  }

  // Construct total accelerometer bias, including calculated average accelerometer bias from above
  accel_bias_reg[0] -= (accel_bias[0]/8); // Subtract calculated averaged accelerometer bias scaled to
2048 LSB/g (16 g full scale)
  accel_bias_reg[1] -= (accel_bias[1]/8);
  accel_bias_reg[2] -= (accel_bias[2]/8);

  data[0] = (accel_bias_reg[0] >> 8) & 0xFF;
  data[1] = (accel_bias_reg[0])      & 0xFF;
  data[1] = data[1] | mask_bit[0]; // preserve temperature compensation bit when writing back to
accelerometer bias registers
  data[2] = (accel_bias_reg[1] >> 8) & 0xFF;
  data[3] = (accel_bias_reg[1])      & 0xFF;
  data[3] = data[3] | mask_bit[1]; // preserve temperature compensation bit when writing back to
accelerometer bias registers
  data[4] = (accel_bias_reg[2] >> 8) & 0xFF;
  data[5] = (accel_bias_reg[2])      & 0xFF;
  data[5] = data[5] | mask_bit[2]; // preserve temperature compensation bit when writing back to
accelerometer bias registers

  // Push accelerometer biases to hardware registers
  writeByte(MPU6050_ADDRESS, XA_OFFSET_H, data[0]); // might not be supported in MPU6050
  writeByte(MPU6050_ADDRESS, XA_OFFSET_L_TC, data[1]);
  writeByte(MPU6050_ADDRESS, YA_OFFSET_H, data[2]);
  writeByte(MPU6050_ADDRESS, YA_OFFSET_L_TC, data[3]);
  writeByte(MPU6050_ADDRESS, ZA_OFFSET_H, data[4]);
  writeByte(MPU6050_ADDRESS, ZA_OFFSET_L_TC, data[5]);

// Output scaled accelerometer biases for manual subtraction in the main program
   dest2[0] = (float)accel_bias[0]/(float)accelsensitivity;
   dest2[1] = (float)accel_bias[1]/(float)accelsensitivity;
   dest2[2] = (float)accel_bias[2]/(float)accelsensitivity;
}


// Accelerometer and gyroscope self test; check calibration wrt factory settings
void MPU6050SelfTest(float * destination) // Should return percent deviation from factory trim values,
+/- 14 or less deviation is a pass
{
   uint8_t rawData[4];
   uint8_t selfTest[6];
```

```
   float factoryTrim[6];

   // Configure the accelerometer for self-test
   writeByte(MPU6050_ADDRESS, ACCEL_CONFIG, 0xF0); // Enable self test on all three axes and set
accelerometer range to +/- 8 g
   writeByte(MPU6050_ADDRESS, GYRO_CONFIG,  0xE0); // Enable self test on all three axes and set gyro
range to +/- 250 degrees/s
   delay(250);  // Delay a while to let the device execute the self-test
   rawData[0] = readByte(MPU6050_ADDRESS, SELF_TEST_X); // X-axis self-test results
   rawData[1] = readByte(MPU6050_ADDRESS, SELF_TEST_Y); // Y-axis self-test results
   rawData[2] = readByte(MPU6050_ADDRESS, SELF_TEST_Z); // Z-axis self-test results
   rawData[3] = readByte(MPU6050_ADDRESS, SELF_TEST_A); // Mixed-axis self-test results
   // Extract the acceleration test results first
   selfTest[0] = (rawData[0] >> 3) | (rawData[3] & 0x30) >> 4 ; // XA_TEST result is a five-bit
unsigned integer
   selfTest[1] = (rawData[1] >> 3) | (rawData[3] & 0x0C) >> 4 ; // YA_TEST result is a five-bit
unsigned integer
   selfTest[2] = (rawData[2] >> 3) | (rawData[3] & 0x03) >> 4 ; // ZA_TEST result is a five-bit
unsigned integer
   // Extract the gyration test results first
   selfTest[3] = rawData[0]  & 0x1F ; // XG_TEST result is a five-bit unsigned integer
   selfTest[4] = rawData[1]  & 0x1F ; // YG_TEST result is a five-bit unsigned integer
   selfTest[5] = rawData[2]  & 0x1F ; // ZG_TEST result is a five-bit unsigned integer
   // Process results to allow final comparison with factory set values
   factoryTrim[0] = (4096.0*0.34)*(pow( (0.92/0.34) , (((float)selfTest[0] - 1.0)/30.0))); // FT[Xa]
factory trim calculation
   factoryTrim[1] = (4096.0*0.34)*(pow( (0.92/0.34) , (((float)selfTest[1] - 1.0)/30.0))); // FT[Ya]
factory trim calculation
   factoryTrim[2] = (4096.0*0.34)*(pow( (0.92/0.34) , (((float)selfTest[2] - 1.0)/30.0))); // FT[Za]
factory trim calculation
   factoryTrim[3] =  ( 25.0*131.0)*(pow( 1.046 , ((float)selfTest[3] - 1.0) ));             // FT[Xg]
factory trim calculation
   factoryTrim[4] =  (-25.0*131.0)*(pow( 1.046 , ((float)selfTest[4] - 1.0) ));             // FT[Yg]
factory trim calculation
   factoryTrim[5] =  ( 25.0*131.0)*(pow( 1.046 , ((float)selfTest[5] - 1.0) ));             // FT[Zg]
factory trim calculation

 //  Output self-test results and factory trim calculation if desired
 //  Serial.println(selfTest[0]); Serial.println(selfTest[1]); Serial.println(selfTest[2]);
 //  Serial.println(selfTest[3]); Serial.println(selfTest[4]); Serial.println(selfTest[5]);
 //  Serial.println(factoryTrim[0]); Serial.println(factoryTrim[1]); Serial.println(factoryTrim[2]);
 //  Serial.println(factoryTrim[3]); Serial.println(factoryTrim[4]); Serial.println(factoryTrim[5]);

 // Report results as a ratio of (STR - FT)/FT; the change from Factory Trim of the Self-Test Response
 // To get to percent, must multiply by 100 and subtract result from 100
   for (int i = 0; i < 6; i++) {
     destination[i] = 100.0 + 100.0*((float)selfTest[i] - factoryTrim[i])/factoryTrim[i]; // Report
percent differences
   }

}

        void writeByte(uint8_t address, uint8_t subAddress, uint8_t data)
{
        Wire.beginTransmission(address);  // Initialize the Tx buffer
        Wire.write(subAddress);           // Put slave register address in Tx buffer
        Wire.write(data);                 // Put data in Tx buffer
        Wire.endTransmission();           // Send the Tx buffer
}

        uint8_t readByte(uint8_t address, uint8_t subAddress)
{
        uint8_t data; // `data` will store the register data
        Wire.beginTransmission(address);         // Initialize the Tx buffer
        Wire.write(subAddress);                  // Put slave register address in Tx buffer
        Wire.endTransmission(false);             // Send the Tx buffer, but send a restart to keep
connection alive
        Wire.requestFrom(address, (uint8_t) 1);  // Read one byte from slave register address
        data = Wire.read();                      // Fill Rx buffer with result
        return data;                             // Return data read from slave register
}

        void readBytes(uint8_t address, uint8_t subAddress, uint8_t count, uint8_t * dest)
```

```
{
        Wire.beginTransmission(address);   // Initialize the Tx buffer
        Wire.write(subAddress);            // Put slave register address in Tx buffer
        Wire.endTransmission(false);       // Send the Tx buffer, but send a restart to keep connection
alive
        uint8_t i = 0;
        Wire.requestFrom(address, count);  // Read bytes from slave register address
        while (Wire.available()) {
        dest[i++] = Wire.read(); }         // Put read results in the Rx buffer
}
// Implementation of Sebastian Madgwick's "...efficient orientation filter for... inertial/magnetic
sensor arrays"
// (see http://www.x-io.co.uk/category/open-source/ for examples and more details)
// which fuses acceleration and rotation rate to produce a quaternion-based estimate of relative
// device orientation -- which can be converted to yaw, pitch, and roll. Useful for stabilizing
quadcopters, etc.
// The performance of the orientation filter is at least as good as conventional Kalman-based filtering
algorithms
// but is much less computationally intensive---it can be performed on a 3.3 V Pro Mini operating at 8
MHz!
        void MadgwickQuaternionUpdate(float ax, float ay, float az, float gx, float gy, float gz)
        {
            float q1 = q[0], q2 = q[1], q3 = q[2], q4 = q[3];         // short name local variable for
readability
            float norm;                                               // vector norm
            float f1, f2, f3;                                         // objetive funcyion elements
            float J_11or24, J_12or23, J_13or22, J_14or21, J_32, J_33; // objective function Jacobian
elements
            float qDot1, qDot2, qDot3, qDot4;
            float hatDot1, hatDot2, hatDot3, hatDot4;
            float gerrx, gerry, gerrz, gbiasx, gbiasy, gbiasz;        // gyro bias error

            // Auxiliary variables to avoid repeated arithmetic
            float _halfq1 = 0.5f * q1;
            float _halfq2 = 0.5f * q2;
            float _halfq3 = 0.5f * q3;
            float _halfq4 = 0.5f * q4;
            float _2q1 = 2.0f * q1;
            float _2q2 = 2.0f * q2;
            float _2q3 = 2.0f * q3;
            float _2q4 = 2.0f * q4;
            float _2q1q3 = 2.0f * q1 * q3;
            float _2q3q4 = 2.0f * q3 * q4;

            // Normalise accelerometer measurement
            norm = sqrt(ax * ax + ay * ay + az * az);
            if (norm == 0.0f) return; // handle NaN
            norm = 1.0f/norm;
            ax *= norm;
            ay *= norm;
            az *= norm;

            // Compute the objective function and Jacobian
            f1 = _2q2 * q4 - _2q1 * q3 - ax;
            f2 = _2q1 * q2 + _2q3 * q4 - ay;
            f3 = 1.0f - _2q2 * q2 - _2q3 * q3 - az;
            J_11or24 = _2q3;
            J_12or23 = _2q4;
            J_13or22 = _2q1;
            J_14or21 = _2q2;
            J_32 = 2.0f * J_14or21;
            J_33 = 2.0f * J_11or24;

            // Compute the gradient (matrix multiplication)
            hatDot1 = J_14or21 * f2 - J_11or24 * f1;
            hatDot2 = J_12or23 * f1 + J_13or22 * f2 - J_32 * f3;
            hatDot3 = J_12or23 * f2 - J_33 *f3 - J_13or22 * f1;
            hatDot4 = J_14or21 * f1 + J_11or24 * f2;

            // Normalize the gradient
            norm = sqrt(hatDot1 * hatDot1 + hatDot2 * hatDot2 + hatDot3 * hatDot3 + hatDot4 * hatDot4);
            hatDot1 /= norm;
            hatDot2 /= norm;
```

```
    hatDot3 /= norm;
    hatDot4 /= norm;

    // Compute estimated gyroscope biases
    gerrx = _2q1 * hatDot2 - _2q2 * hatDot1 - _2q3 * hatDot4 + _2q4 * hatDot3;
    gerry = _2q1 * hatDot3 + _2q2 * hatDot4 - _2q3 * hatDot1 - _2q4 * hatDot2;
    gerrz = _2q1 * hatDot4 - _2q2 * hatDot3 + _2q3 * hatDot2 - _2q4 * hatDot1;

    // Compute and remove gyroscope biases
    gbiasx += gerrx * deltat * zeta;
    gbiasy += gerry * deltat * zeta;
    gbiasz += gerrz * deltat * zeta;
    gx -= gbiasx;
    gy -= gbiasy;
    gz -= gbiasz;

    // Compute the quaternion derivative
    qDot1 = -_halfq2 * gx - _halfq3 * gy - _halfq4 * gz;
    qDot2 =  _halfq1 * gx + _halfq3 * gz - _halfq4 * gy;
    qDot3 =  _halfq1 * gy - _halfq2 * gz + _halfq4 * gx;
    qDot4 =  _halfq1 * gz + _halfq2 * gy - _halfq3 * gx;

    // Compute then integrate estimated quaternion derivative
    q1 += (qDot1 -(beta * hatDot1)) * deltat;
    q2 += (qDot2 -(beta * hatDot2)) * deltat;
    q3 += (qDot3 -(beta * hatDot3)) * deltat;
    q4 += (qDot4 -(beta * hatDot4)) * deltat;

    // Normalize the quaternion
    norm = sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4);    // normalise quaternion
    norm = 1.0f/norm;
    q[0] = q1 * norm;
    q[1] = q2 * norm;
    q[2] = q3 * norm;
    q[3] = q4 * norm;
}
```