```c
#include "ES_Configure.h"
#include "ES_Framework.h"
#include "ES_DeferRecall.h"
#include "TestHarnessService0.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "inc/hw_sysctl.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"      // Define PART_TM4C123GH6PM in project
#include "driverlib/gpio.h"
#include "ES_ShortTimer.h"
#include "driverlib/i2c.h"
#include "inc/hw_i2c.h"
#include <math.h>

#include "Hardware.h"
#include "Constants.h"
#include "I2C_Service.h"

#define IMPACT_THRESHOLD 1500
#define IMU_DEBOUNCE 90
#define PERIOD_THRESHOLD 1000

static uint8_t MyPriority;
static I2C_State CurrentState = I2C_Init;
static int16_t Accel_X = 0;
static int16_t Accel_Y = 0;
static int16_t Accel_Z = 0;
static int16_t Gyro_X = 0;
static int16_t Gyro_Y = 0;
static int16_t Gyro_Z = 0;
static int16_t Accel_X_OFF = 0;
static int16_t Accel_Y_OFF = 0;
static int16_t Accel_Z_OFF = 0;
static int16_t Gyro_X_OFF = 0;
static int16_t Gyro_Y_OFF = 0;
static int16_t Gyro_Z_OFF = 0;
//static int16_t thX = 0;
//static int16_t thY = 0;
//static int16_t thZ = 0;

static int16_t Rate_of_Change = 0;
static int16_t Last_Rate_of_Change = 0;
static uint8_t Num_Vals = 4;
static int16_t Rate_History[10] = {0};
static int16_t Last_Accel[3] = {0, 0, 0};
static uint8_t Debounce_Counter =0;
static uint16_t Last_Time = 0;
static uint16_t Av_Period = 1000;
static bool stopped = true;

static bool read = 0;
static uint8_t Send_Registers[1] = {POWER_REGISTER};
static uint8_t Send_Data[1] = {POWER_SETTING};
static uint8_t Receive_Registers[12] = {GYROSCOPE_X_REGISTER_BASE,
GYROSCOPE_X_REGISTER_BASE - 1, GYROSCOPE_Y_REGISTER_BASE,
GYROSCOPE_Y_REGISTER_BASE - 1,
GYROSCOPE_Z_REGISTER_BASE,GYROSCOPE_Z_REGISTER_BASE - 1,
ACCELEROMETER_X_REGISTER_BASE, ACCELEROMETER_X_REGISTER_BASE - 1,
ACCELEROMETER_Y_REGISTER_BASE, ACCELEROMETER_Y_REGISTER_BASE - 1,
ACCELEROMETER_Z_REGISTER_BASE, ACCELEROMETER_Z_REGISTER_BASE - 1};
static uint16_t Receive_Data[12] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```
bool Init_I2C(uint8_t Priority)
{
      // set local priority
  MyPriority = Priority;
  // set timer to allow I2C to hookup
      ES_Timer_InitTimer(IMU_TIMER, I2C_DELAY_TIME);
      // state is init

      return true;
}

bool Post_I2C(ES_Event ThisEvent)
{
  return ES_PostToService( MyPriority, ThisEvent);
}

ES_Event Run_I2C( ES_Event ThisEvent )
{
      I2C_State NextState = CurrentState;
  ES_Event ReturnEvent;
  ReturnEvent.EventType = ES_NO_EVENT; // assume no errors

      // loop through states
      switch (CurrentState)
      {
            // if state is init
            case (I2C_Init):
            {
                  // if event is IMU_Timeout
                  if ((ThisEvent.EventType == ES_TIMEOUT) && (ThisEvent.EventParam
== IMU_TIMER))
                  {
                        //printf("\r\nGyro X\tGyro Y\tGyro Z\tAccel X\tAccel
Y\tAccel Z\r\n");
                        // initialize Gyro/accelerometer power settings
                        HWREG(I2C2_BASE + I2C_O_MDR) = Send_Registers[0];
                        HWREG(I2C2_BASE + I2C_O_MCS) = I2C_MCS_START_TX;
                        // set IMU Timer
                        ES_Timer_InitTimer(IMU_TIMER, IMU_POLL_TIME);
                        // next state is calibrate
                        NextState = I2C_Poll_IMU;
                  }
                  break;
            }
            // else if state is poll
            case (I2C_Poll_IMU):
            {
                  // if event is timeout
                  if ((ThisEvent.EventType == ES_TIMEOUT) && (ThisEvent.EventParam
== IMU_TIMER))
                  {
                        // reset timer
                        ES_Timer_InitTimer(IMU_TIMER, IMU_POLL_TIME);
                        // start next read
                        // set addr to send
                        HWREG(I2C2_BASE + I2C_O_MSA) = IMU_SLAVE_ADDRESS;
                        HWREG(I2C2_BASE + I2C_O_MSA) &= ~I2C_MSA_RS;
                        // load register to read
                        HWREG(I2C2_BASE + I2C_O_MDR) = Receive_Registers[11];
                        // load START TX
                        HWREG(I2C2_BASE + I2C_O_MCS) = I2C_MCS_START_TX;

                  }
                  else if (ThisEvent.EventType == ES_IMPACT)
                  {
                        //printf("\r\nImpact: %d\r\n", Rate_of_Change);
                        printf("\r\nImpact: %d\r\n", Av_Period);
                  }
                  break;
            }
      }

      CurrentState = NextState;
  return ReturnEvent;
}

uint16_t getPeriod(void)
{
      return Av_Period;
}


void I2C_ISR(void)
{
      static uint8_t Read_Index = 0;
      static uint8_t Send_Index = 0;
      static uint8_t Sends_Left = 0;
      static uint8_t Reads_Left = 11;
      //clear the source of the interrupt
      HWREG(I2C2_BASE + I2C_O_MICR) = I2C_MICR_IC;
      //if read is set
      if (read == 1)
      {
            // if index is 0
            if (Read_Index == 0)
            {
                  for (int i = 0; i<400; i++);
                  // set addr to read
                  HWREG(I2C2_BASE + I2C_O_MSA) = IMU_SLAVE_ADDRESS;
                  HWREG(I2C2_BASE + I2C_O_MSA) |= I2C_MSA_RS;
                  // load START RX
                  HWREG(I2C2_BASE + I2C_O_MCS) = I2C_MCS_SINGLE_RX;
                  // increment index
                  Read_Index ++;
            }
            // else if index is 1
            else if (Read_Index == 1)
            {
                  // read data from buffer
                  Receive_Data[Reads_Left] = (HWREG(I2C2_BASE + I2C_O_MDR) & 0xff);
                  // if reads left is 0
                  if (Reads_Left == 0)
                  {
                        // update Accel/Gyro vals
                        Gyro_X = ((Receive_Data[0]) | (Receive_Data[1] << 8)) -
Gyro_X_OFF;
                        Gyro_Y = ((Receive_Data[2]) | (Receive_Data[3] << 8)) -
Gyro_Y_OFF;
                        Gyro_Z = ((Receive_Data[4]) | (Receive_Data[5] << 8)) -
Gyro_Z_OFF;
                        Accel_X = ((Receive_Data[6]) | (Receive_Data[7] << 8)) -
Accel_X_OFF;
                        Accel_Y = ((Receive_Data[8]) | (Receive_Data[9] << 8)) -
Accel_Y_OFF;
                        Accel_Z = ((Receive_Data[10]) | (Receive_Data[11] << 8)) -
Accel_Z_OFF;
```

```
                        //int16_t New_Rate = sqrt((Accel_X - Last_Accel[0])^2 +
(Accel_Y - Last_Accel[1])^2 + (Accel_Z - Last_Accel[2])^2)/Num_Vals;
                        Last_Rate_of_Change = Rate_of_Change;
//                      int16_t New_Rate = (abs((Accel_X>>4) - Last_Accel[0]) +
abs((Accel_Y>>4) - Last_Accel[1]) + abs((Accel_Z>>4) - Last_Accel[2]));
//                      Last_Accel[0] = Accel_X>>4;
//                      Last_Accel[1] = Accel_Y>>4;
//                      Last_Accel[2] = Accel_Z>>4;
//                      int16_t sum = 0;
//                      for (int j = 0; j < (Num_Vals - 1); j++)
//                      {
//                            Rate_History[j] = Rate_History[j + 1];
//                            sum += Rate_History[j]/(Num_Vals*2);
//                      }
//                      Rate_of_Change = sum + New_Rate/(Num_Vals);
//                      Rate_History[Num_Vals - 1] = New_Rate;
                        Rate_of_Change = (abs((Accel_X>>4) - Last_Accel[0]) +
abs((Accel_Y>>4) - Last_Accel[1]) + abs((Accel_Z>>4) - Last_Accel[2]));
                        Last_Accel[0] = Accel_X>>4;
                        Last_Accel[1] = Accel_Y>>4;
                        Last_Accel[2] = Accel_Z>>4;
//                      printf("\r\nRate: %d\r", Rate_of_Change);
//                      if (Debounce_Counter != 0)
//                      {
//                            Debounce_Counter --;
//                      }

                        uint16_t Time = ES_Timer_GetTime();
                        uint16_t Period = Time - Last_Time;
                        if ((Period > PERIOD_THRESHOLD) && (stopped == false))
                              {
                                    stopped = true;
                                    Last_Time = Time;
                                    Av_Period = PERIOD_THRESHOLD;
                                    ES_Event Event2Post;
                                    Event2Post.EventType = ES_IMPACT;
                                    Post_I2C(Event2Post);
//                                  Debounce_Counter = IMU_DEBOUNCE;
                              }

                        if ((Rate_of_Change > IMPACT_THRESHOLD) &&
(Last_Rate_of_Change <= IMPACT_THRESHOLD)) // && (Debounce_Counter == 0))
                        {
                              Last_Time = Time;
                              if (stopped == true)
                              {
                                    Av_Period = PERIOD_THRESHOLD/10*9;
                              }
                              else if (stopped == false)
                              {
                                    Av_Period = (Av_Period + Period)/2;
                              }
                              ES_Event Event2Post;
                              Event2Post.EventType = ES_IMPACT;
                              Post_I2C(Event2Post);
//                            Debounce_Counter = IMU_DEBOUNCE;
                              stopped = false;
                        }

                        // reset reads left
                        Reads_Left = 11;
                        Read_Index = 0;
                  }
                  else
```

```
                {
                    // decrement Reads left
                    Reads_Left --;
                    // reset index to 0
                    Read_Index = 0;
                    // start next read
                    // set addr to send
                    HWREG(I2C2_BASE + I2C_O_MSA) = IMU_SLAVE_ADDRESS;
                    HWREG(I2C2_BASE + I2C_O_MSA) &= ~I2C_MSA_RS;
                    // load register to read
                    HWREG(I2C2_BASE + I2C_O_MDR) =
Receive_Registers[Reads_Left];
                    // load START TX
                    HWREG(I2C2_BASE + I2C_O_MCS) = I2C_MCS_START_TX;
                }
            }
    }
    // else if not read (send)
    else if (read == 0)
    {
        // if send index is 0
        if (Send_Index == 0)
        {
            // load Data
            HWREG(I2C2_BASE + I2C_O_MDR) = Send_Data[Sends_Left];
            // load LAST TX
            HWREG(I2C2_BASE + I2C_O_MCS) = I2C_MCS_LAST_TX;
            // increment send index
            Send_Index ++;
        }
        // else if send index is 2
        else if (Send_Index == 1)
        {
            // if sends left is 1
            if (Sends_Left != 0)
            {
                // decrement sends left
                Sends_Left --;
                // load register to write
                HWREG(I2C2_BASE + I2C_O_MDR) = Send_Registers[Sends_Left];
            }
            // else if sends left is 0
            else if (Sends_Left == 0)
            {
                // set read
                read = 1;
                // load register to read
                HWREG(I2C2_BASE + I2C_O_MDR) =
Receive_Registers[Reads_Left];
            }
            // set send index to 0
            Send_Index = 0;
            // load START TX
            HWREG(I2C2_BASE + I2C_O_MCS) = I2C_MCS_START_TX;
        }
    }
}
```