Laboratory 05 – I2C

By: **Caleb Probst**

Spring 2023 ECE 381 Microcontrollers

Introduction

In this laboratory, the main concept is the use of I2C (Inter-Integrated Circuits). I2C is another form of serial communication that is used as a communication method between chips. This is a two wire communication protocol that has an SDA (Serial Data) and SCL (Serial Clock). The data line is a bidirectional line that can both send and receive, and the clock line ensures the clock speed is the same for all devices that are reading the data line. The PSoC4 that we have has 2 built in I2C devices that we are going to use for this laboratory, which is the accelerometer and the FRAM (Ferroelectric RAM).

Goal

The goal of this laboratory is to create a battery powered jolt detector. The purpose of this jolt detector would be able to make it portable yet accurate. In order to make it portable, we need to be able to save power as portability requires battery operation. In order to make it accurate, we need to program the accelerometer properly to not interrupt the sleep of the processor when it does not need to, or when it shouldn’t. The accelerometer on the PSoC4 that we have has a built in interrupt pin that is internally connected to the processor for interrupting the processor. This is true even if the processor is in deep sleep mode. First, we need to be able to configure the accelerometer properly in order to interrupt at the requires threshold. But, what if we lose power? Well, this is where the FRAM comes into play. Since FRAM uses magnetic fields to store its bits, it is non-volatile memory which can be read from and written to. So, once we have the interrupt, we now need to store the data into the FRAM for safe keeping. Now, we need to find out which source it came from: X, Y or Z. This can be done through lighting up an LED to signify whether it was an X, Y, or Z interrupt.

Materials

PsoC4 4200M Microcontroller	1

Procedure and Results

Part 1 – Setup in PSoC Creator

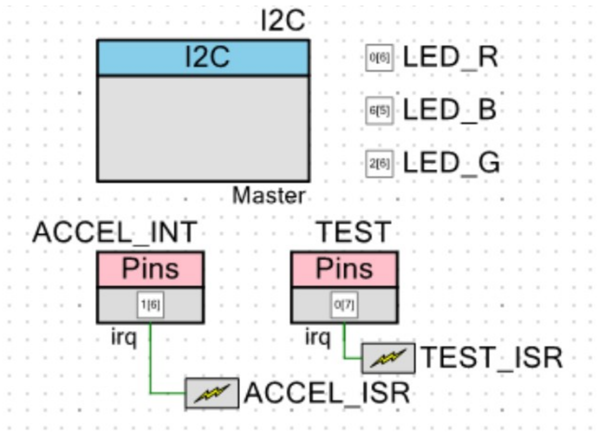


Figure 1: Top Design

Figure 1 shows the Top Design setup for the laboratory. The setup is very simple and only requires the I2C module, 2 input pins and 3 output pins. The input pins should be set to a dedicated interrupt (ACCEL_INT and TEST). ACCEL_INT should be set to a rising edge interrupt and TEST should be set to a falling edge interrupt, as well as resistive pull down mode. The output pins should be set to strong drive mode and be named LED_R, LED_G, and LED_B for the 3 LED colors. The pin setup for the PSoC are given by *Table 1*.

I2C:scl	P4[0]
I2C:sda	P4[1]
ACCEL_INT	P1[6]
TEST	P0[7]
LED_R	P0[6]
LED_G	P2[6]
LED_B	P6[5]

Table 1: Pin Outs of Hardware on PSoC4

Part 2 – Hardware and Wiring

For this laboratory, there is no hardware to be wired to the PSoC4 as everything is already built into the microcontroller.

Part 3 – Software

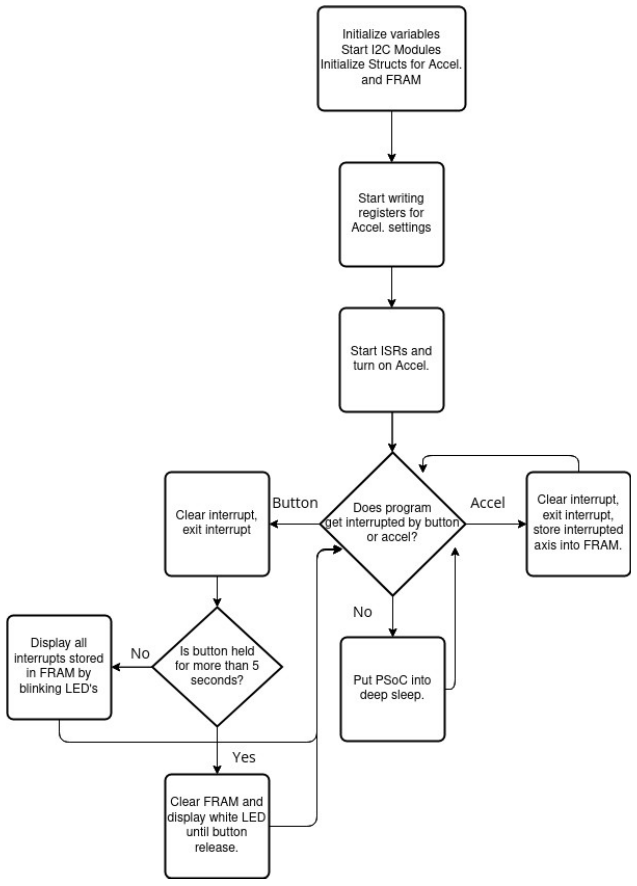

Initialize variables
Start I2C Modules
Initialize Structs for Accel. and FRAM
Start writing registers for Accel. settings
Start ISRs and turn on Accel.
Does program get interrupted by button or accel?
Button
Accel
Clear interrupt, exit interrupt
Clear interrupt, exit interrupt, store interrupted axis into FRAM.
No
Put PSoC into deep sleep.
Is button held for more than 5 seconds?
No
Display all interrupts stored in FRAM by blinking LED's
Yes
Clear FRAM and display white LED until button release.

Figure 2: Flowchart of Program

The software of this laboratory contributes almost entirely to the operation of the laboratory. All code can be found in the Appendix, under *Main Code*. The first thing we started with was the initialization of variables and starting the modules and interrupts for the laboratory. Instead of using constant values, we opted to using structs, as this would be able to "object-ify" our code. With that, we don't have to constantly use the register addresses shown by the data sheets. Next, we started setting the registers needed for the program to run properly, specifically the accelerometer. All addresses can be found in the datasheet as well as the struct initialization in *Main Code*. The first thing we changed was the resolution of the accelerometer from the default, to 12-bit. This helps with the accuracy of our acceleration values. Along with that, we set the range to go in between 8G and -8G. We also turned on the wakeup function, which will allow the motion to be detected. This is all through the CTRL_REG1 (Address 0x1B). The next register we changed was the wakeup timer, as this would allow the number of valid counts above the threshold are counted and then interrupt the program. We set this to 10 in WAKEUP_TIMER (Address 0x29). Next we set the threshold to 3G's using the following equation.

$$\text{WAKEUP\_THRESHOLD (counts)} = \text{Desired Threshold (g)} \times 16 \text{ (counts/g)}$$

Equation 1: WAKEUP_THRESHOLD Calculation

This was set in the WAKEUP_THRESHOLD register (Address 0x6A). Next, we set the wakeup data rate to 12.5Hz, which was in the CTRL_REG2 (Address 0x1D). The next thing we set was the interrupt axis, which we wanted all of them, so we set all of the axis to '1', which was set in INT_CTRL_REG2 (Address 0x1F). Next process is to enable the physical interrupt pin, as well as the polarity to active high. This is set through the INT_CTRL_REG1 (Address 0x1E). Then we set the LPF Roll-Off data rate to the same as the wakeup data rate, which is set in DATA_CTRL_REG (Address 0x21). Finally, we reset CTRL_REG1 to turn PC1 bit on, which turns on the accelerometer.

With this, we can now start the main program. Here we are only checking for interrupt flags. If we get an interrupt flag, we want to store the axis of interrupt into FRAM. We do this through the I2C bus, and also have to read INT_REL (Address 0x1A) to release the interrupt on the accelerometer. We simply stored the variable into FRAM. The way we implemented it, is it would only set the FRAM once and wouldn't set it again until cleared. Once we press the button, we then test to see if its held down for 5 seconds or more. If it is, we set the LED's to white, and wait for release, then clear FRAM with a sequence. If it is not, we play back which axis have been interrupted.

Testing Methodology

For the testing of this device, we had to beat up our PSoC. Shown by *Figure 3* is a reference to how we treated our PSoC during testing.

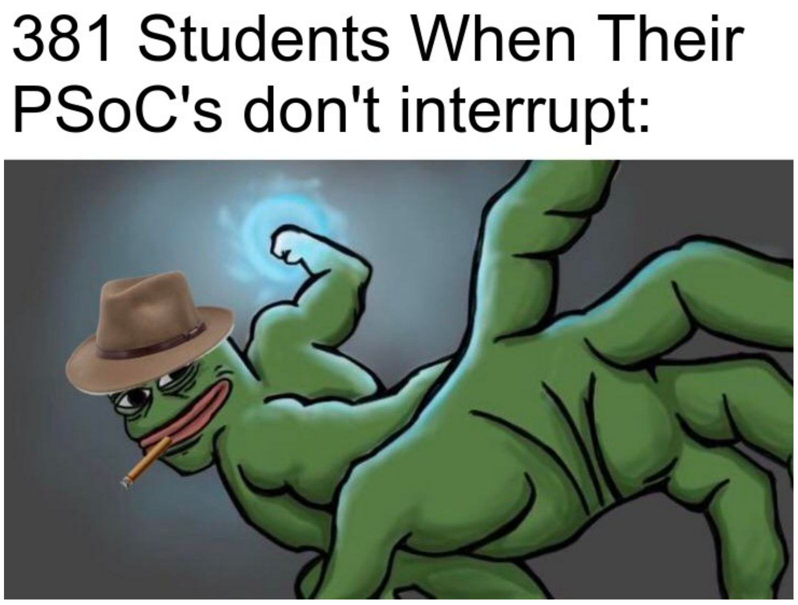
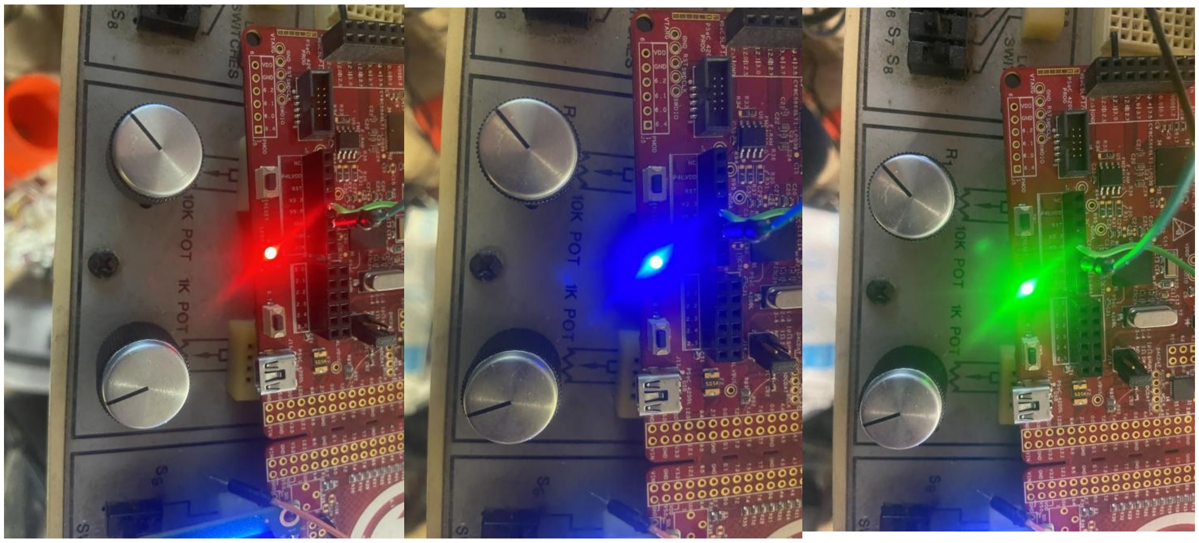


Figure 3: Student Testing Methodology

Here are some pictures of us testing our PSoC. This is shown by *Figures 4-9*.

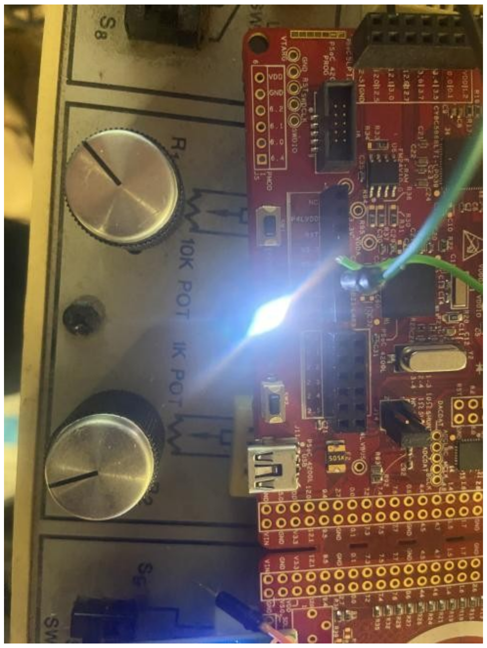
Figures 4-9: Testing Jolt Detector

To first test the system, we needed the accelerometer to interrupt the system and create some values in FRAM. We beat up our PSoC so this could happen. Once we were sure that we got some values, we simply pressed the button. The LED on the bottom of the PSoC would light up red for x, green for y, and blue for z. These tests are shown by *Figures 10-12*.



Figures 10-12: LED Tests

Then, when we held the button down, a small LED sequence would play and then if we clicked the FRAM again, nothing would show, which meant the FRAM was cleared. This is shown by *Figure 13*.



Figures 14: LED Test

Conclusion

This laboratory was relatively straightforward. We used many functions that allowed us to not only make our code simpler, but also made it more readable. With as many times as we read from and wrote to the I2C bus, making a function for it made everything much easier. While reading the lab writeup, it appears that we were supposed to continuously store the values that were interrupted into the FRAM, as opposed to only seeing which axis were interrupted. This could be simply fixed by looping through the FRAM and appending onto the last one the current interrupt.

Appendix

Main Code

```
#include "project.h"
// Create global variables for device addresses and flags
static const uint32 accelAddr = 0x0F;
static const uint32 ramLowerAddr = 0x50;
static const uint32 ramUpperAddr = 0x51;
volatile int accelFlag = 0;
volatile int buttonFlag = 0;

// Create struct for the accelerometer to store register addresses
struct accelerometer{
    uint32 XOUT_L;
    uint32 XOUT_H;
    uint32 YOUT_L;
    uint32 YOUT_H;
    uint32 ZOUT_L;
    uint32 ZOUT_H;
    uint32 CtrlReg1;
    uint32 CtrlReg2;
    uint32 IntSrc2;
    uint32 IntRel;
    uint32 DataCtrlReg;
    uint32 WakeupThreshold;
    uint32 WakeupTimer;
    uint32 IntCtrlReg1;
    uint32 IntCtrlReg2;
    uint32 Test;
};
// Create struct for FRAM to store memory addresses
struct FRAM{
    uint32 Xint;
    uint32 Yint;
    uint32 Zint;
};

/*
This function simply reads from any I2C device.
readBuf will be the size of readBytes and any read data will be stored
here..
readBytes is the number of bytes that the I2C is reading.
devAddr is the I2C address that the microcontroller is talking to.
memAddr is the memory or register address that is inside the I2C
device.
*/
void readFromAddr(uint8* readBuf, int readBytes, uint32 devAddr,
uint32 memAddr){
    // First write the memory address to the I2C bus.
    uint8 wrBuf[1];
    wrBuf[0] = memAddr;
    I2C_I2CMasterWriteBuf(devAddr, wrBuf, 1, I2C_I2C_MODE_NO_STOP);
    while(!(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_WR_CMPLT));
    I2C_I2CMasterClearStatus();
    // Then read the output from the I2C bus from that memory address.
    I2C_I2CMasterReadBuf(devAddr, readBuf, readBytes,
I2C_I2C_MODE_REPEAT_START);
    while(!(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_RD_CMPLT));
    I2C_I2CMasterClearStatus();
}
/*
This function simply writes to any I2C device.
writeBuf will be the size of writeBytes and any data inside here will
be written to the I2C bus.
writeBytes is the number of bytes written to the I2C bus.
    NOTE: The first byte of the writeBuf will be overwritten if it is
stored, as it is the memory address being written. Make sure to start
at 1 for the array and to add 1 to writeBytes to adjust for this.
devAddr is the I2C address that the microcontroller is talking to.
memAddr is the memory or register address that is inside the I2C
device.
*/
void writeToAddr(uint8* writeBuf, int writeBytes, uint32 devAddr,
uint32 memAddr){
    // Store the first byte as the memory address we are writing to.
    writeBuf[0] = memAddr;

    // Write the data to the device.
    I2C_I2CMasterWriteBuf(devAddr, writeBuf, writeBytes,
I2C_I2C_MODE_COMPLETE_XFER);
    while(!(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_WR_CMPLT));
    I2C_I2CMasterClearStatus();
}
/*
This function is similar to the readFromAddr, except requries two
memory addresses, as the address is 2 bytes instead of 1. This simply
reads data from RAM.
readBuf will be the size of readBytes and any data will be written
inside here.
readBytes is the number of bytes read from the I2C bus.
devAddr is the I2C address that the microcontroller is talking to.
memAddr1 is the top half of the memory address. Ex: the "12" in
"0x1234"
memAddr2 is the bottom half of the memory address. Ex: the "34" in
"0x1234"
*/
void readFromRam(uint8* readBuf, int readBytes, uint32 devAddr, uint32
memAddr1, uint32 memAddr2){
    // Write the memory address to the I2C bus.
    uint8 wrBuf[2];
    wrBuf[0] = memAddr1;
    wrBuf[1] = memAddr2;
    I2C_I2CMasterWriteBuf(devAddr, wrBuf, 2, I2C_I2C_MODE_NO_STOP);
    while(!(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_WR_CMPLT));
    I2C_I2CMasterClearStatus();

    // Read number of bytes from the I2C bus.
    I2C_I2CMasterReadBuf(devAddr, readBuf, readBytes,
I2C_I2C_MODE_REPEAT_START);
```

```
    while(!(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_RD_CMPLT));
    I2C_I2CMasterClearStatus();
}
/*
This function is similar to the writeToAddr, except requries two
memory addresses, as the address is 2 bytes instead of 1. This simply
writes data to RAM.
writeBuf will be the size of writeBytes and any data inside will be
written to the I2C bus.
writeBytes is the number of bytes written to the I2C bus.
    NOTE: The first two bytes of the writeBuf will be overwritten if
it is stored, as it is the memory address being written. Make sure to
start at 2 for the array and to add 2 to writeBytes to adjust for
this.
devAddr is the I2C address that the microcontroller is talking to.
memAddr1 is the top half of the memory address. Ex: the "12" in
"0x1234"
memAddr2 is the bottom half of the memory address. Ex: the "34" in
"0x1234"
*/
void writeToRam(uint8* writeBuf, int writeBytes, uint32 devAddr,
uint32 memAddr1, uint32 memAddr2){
    // Write the memory address to the array.
    writeBuf[0] = memAddr1;
    writeBuf[1] = memAddr2;

    // Write any data inside writeBuf to the bus.
    I2C_I2CMasterWriteBuf(devAddr, writeBuf, writeBytes,
I2C_I2C_MODE_COMPLETE_XFER);
    while(!(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_WR_CMPLT));
    I2C_I2CMasterClearStatus();
}
// This function simply clears the RAM, no matter what is in it.
void clearRam(){
    // Create a buffer that starts at address 0x0000 with value 0x00.
    uint8 wrBuf[3] = {0x00,0x00,0x00};

    // Write the address plus the first byte to the bus, no stop.
    I2C_I2CMasterWriteBuf(ramLowerAddr, wrBuf, 3,
I2C_I2C_MODE_NO_STOP);

    // Loop through the addresses  and write 0 to the address
    for (int i = 3; i < 65537; i++){

        // This simply to give a "clear sequence" to the user.
        CyDelayUs(100);
        if(i%10000 == 0){
            LED_B_Write(1);
            LED_G_Write(1);
            LED_R_Write(0);
        }
        else if(i%10000 == 1000){
            LED_R_Write(1);
            LED_B_Write(1);
            LED_G_Write(0);
        }
        else if(i%10000 == 100){
            LED_G_Write(1);
            LED_R_Write(1);
            LED_B_Write(0);
        }
        // End sequence

        // Create buffer of 0x00
        uint8 wrBuf[1] = {0x00};

        // Write 0x00 to I2C bus on the current address
        I2C_I2CMasterWriteBuf(ramLowerAddr, wrBuf, 1,
I2C_I2C_MODE_NO_STOP);
    }
    // Turn off LED's'
    LED_R_Write(1);
    LED_G_Write(1);
    LED_B_Write(1);

    // Create new buffer of 0x00
    uint8 wrBuf2[1] = {0x00};

    // Write last byte, repeat start so I2C ends.
    I2C_I2CMasterWriteBuf(ramLowerAddr, wrBuf2, 1,
I2C_I2C_MODE_REPEAT_START);
    while(!(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_WR_CMPLT));
    I2C_I2CMasterClearStatus();

    // LED Clear Sequence
    for(int i = 0; i < 6; i++){
        if(i%2 == 0){
            LED_R_Write(0);
            LED_G_Write(0);
            LED_B_Write(0);
        }
        else{
            LED_R_Write(1);
            LED_G_Write(1);
            LED_B_Write(1);
        }
        CyDelay(1000);
    }
    // End LED Clear Sequence
}

/*
This function was used in debugging, but was useful. This reads the
accelerometer and writes the accelValues array with the float values
for acceleration. Makes debugging and getting the values easy.
*/
//void readAccel(float* accelValues, struct accelerometer accel){
//        uint8 rdBuf[4];
//        readFromAddr(rdBuf, 1, accelAddr, accel.ZOUT_H);
//        int8 z_hi = (int8)rdBuf[0];
//        readFromAddr(rdBuf, 1, accelAddr, accel.ZOUT_L);
//        int8 z_lo = (int8)rdBuf[0];
//          accelValues[2] = (float)((float)8/((float)2048))*((z_hi*16)
+ (z_lo / 16));
//          readFromAddr(rdBuf, 1, accelAddr, accel.XOUT_H);
//          int8 x_hi = (int8)rdBuf[0];
//          readFromAddr(rdBuf, 1, accelAddr, accel.XOUT_L);
//          int8 x_lo = (int8)rdBuf[0];
//          accelValues[0] = (float)((float)8/((float)2048))*((x_hi*16)
+ (x_lo / 16));
//          readFromAddr(rdBuf, 1, accelAddr, accel.YOUT_H);
//          int8 y_hi = (int8)rdBuf[0];
//          readFromAddr(rdBuf, 1, accelAddr, accel.YOUT_L);
//          int8 y_lo = (int8)rdBuf[0];
//          accelValues[1] = (float)((float)8/((float)2048))*((y_hi*16)
+ (y_lo / 16));
//}
// Main function
int main(void)
{
    // Creates accelerometer struct for easy memory access
    struct accelerometer accel = {
        0x06, // XOUT_L
        0x07, // XOUT_H
        0x08, // YOUT_L
        0x09, // YOUT_H
        0x0A, // ZOUT_L
        0x0B, // ZOUT_H
        0x1B, // CTRL_REG1
        0x1D, // CTRL_REG2
        0x17, // INT_SOURCE2
        0x1A, // INT_REL
        0x21, // DATA_CTRL_REG
        0x6A, // WAKEUP_THRESHOLD
        0x29, // WAKEUP_TIMER
        0x1E, // INT_CTRL_REG1
        0x1F, // INT_CTRL_REG2
        0x0F  // WHO_AM_I
    };
    struct FRAM ram = {
        0x34, // X Interrupt Value Location
        0x36, // Y Interrupt Value Location
        0x38  // Z Interrupt Value Location
    };

    CyGlobalIntEnable; /* Enable global interrupts. */
    /* Place your initialization/startup code here (e.g.
MyInst_Start()) */
    // Start I2C
    I2C_Start();
    // Turn off LED's (was on at first)
    LED_R_Write(1);
    LED_B_Write(1);
    LED_G_Write(1);
    // Create write buffer for I2C (only writing 1 byte at a time)
    uint8 wrBuf[2];

    // Write 0101 0010 to CTRL_REG1
```

```
        // RES - High Current, 12-bit or 14-bit valid
        // DRDYE - Availability of new acceleration data is not
reflected as an interrupt
        // GSEL1 and GSEL0 - +/-8G range
        // WUFE - Wake up Function Enabled
    wrBuf[1] = 0x00 | (0x00 | 0x40 | 0x00 | 0x10 | 0x00 | 0x02);
    writeToAddr(wrBuf, 2, accelAddr, accel.CtrlReg1);

    // Set timer, write to WAKEUP_TIMER
    uint8 count = 10;
    wrBuf[1] = count;
    // WAKEUP_TIMER (counts) = Desired Delay Time (sec) x OWUF (Hz)
    writeToAddr(wrBuf, 2, accelAddr, accel.WakeupTimer);
    // Set threshold.
    uint8 threshold = 3;
    wrBuf[1] = threshold*16;
    // WAKEUP_THRESHOLD (counts) = Desired Threshold (g) x 16
(counts/g)
    writeToAddr(wrBuf, 2, accelAddr, accel.WakeupThreshold);

    // Write 0000 0100 to CtrlReg2
        // OWUFA, OWUFB, OWUFC = 12.5Hz, wakeup function output data
rate.
    wrBuf[1] = 0x00 | (0x04);
    writeToAddr(wrBuf, 2, accelAddr, accel.CtrlReg2);

    // Write 0011 1111 to INT_CTRL_REG2
        // XNWUE - X negative is enabled
        // XPWUE - X positive is enabled
        // YNWUE - Y negative is enabled
        // YPWUE - Y positive is enabled
        // ZNWUE - Z negative is enabled
        // ZPWUE - Z positive is enabled
    wrBuf[1] = 0x00 | (0x20 | 0x10 | 0x08 | 0x04 | 0x02 | 0x01);
    writeToAddr(wrBuf, 2, accelAddr, accel.IntCtrlReg2);

    // Write 0011 0000 to INT_CTRL_REG1
        // IEN - Physical interrupt pin (7) is enabled
        // IEA - Polarity of the physical interrupt pin (7) is active
high
    wrBuf[1] = 0x00 | (0x20 | 0x10);
    writeToAddr(wrBuf, 2, accelAddr, accel.IntCtrlReg1);

    // Write 0000 0001 to DATA_CTRL_REG
        // OSAA, OSAB, OSAC, OSAD = 25Hz Output Data Rate, 12.5 LPF
Roll-off
    wrBuf[1] = 0x00 | (0x01);
    writeToAddr(wrBuf, 2, accelAddr, accel.DataCtrlReg);

    // Write 1101 0010 to CTRL_REG1
        // PC1 - Operating Mode
        // RES - High Current, 12-bit or 14-bit valid
        // DRDYE - Availability of new acceleration data is not
reflected as an interrupt
        // GSEL1 and GSEL0 - +/-8G range
        // WUFE - Wake up Function Enabled
```

```c
wrBuf[1] = 0x00 | (0x80 | 0x40 | 0x00 | 0x10 | 0x00 | 0x02);
writeToAddr(wrBuf, 2, accelAddr, accel.CtrlReg1);

// Create read buffer
uint8 rdBuf[1];
// Delay for startup
CyDelay(12);
// Release interrupt if pin is active
readFromAddr(rdBuf, 1, accelAddr, accel.IntRel);
// Delay
CyDelay(12);
// Start ISR's
ACCEL_ISR_Start();
TEST_ISR_Start();
// Takeout?
TEST_ISR_Enable();

for(;;)
{
    /* Place your application code here. */
    // Put microcontroller into deep sleep, saves power
    CySysPmDeepSleep();

    // If out of sleep...
    // Check accel flag (from interrupt)
    if(accelFlag){
        // Reset flag
        accelFlag = 0;

        // Create read buffer
        uint8 rdBuf[1];

        // Read which source the interrupt came from
        readFromAddr(rdBuf, 1, accelAddr, accel.IntSrc2);
        uint8 reg = rdBuf[0];

        // Check for x
        if(reg&(0x10) || reg&(0x20)){

            // Store 0xFF into ram address ram.Xint
            uint8 wrBuf[3];
            wrBuf[2] = 0xFF;
            writeToRam(wrBuf,3, ramLowerAddr, 0x12, ram.Xint);
        }
        // Check for y
        if(reg&(0x08) || reg&(0x04)){

            // Store 0xFF into ram address ram.Yint
            uint8 wrBuf[3];
            wrBuf[2] = 0xFF;
            writeToRam(wrBuf,3, ramLowerAddr, 0x12, ram.Yint);
        }
        // Check for z
        if(reg&(0x02) || reg&(0x01)){

            // Store 0xFF into ram address ram.Zint
```

```
                uint8 wrBuf[3];
                wrBuf[2] = 0xFF;
                writeToRam(wrBuf,3, ramLowerAddr, 0x12, ram.Zint);
            }
            // Clear interrupt
            readFromAddr(rdBuf, 1, accelAddr, accel.IntRel);
        }
        // Check button interrupt for hold
        if(buttonFlag == 2){
            // Turn LED white
            LED_R_Write(0);
            LED_G_Write(0);
            LED_B_Write(0);

            // Wait for release
            while(TEST_Read() == 0);

            // Turn LED off
            LED_R_Write(1);
            LED_G_Write(1);
            LED_B_Write(1);

            // Reset flag
            buttonFlag = 0;

            // Clear memory
            clearRam();
        }
        // Check button interrupt for press
        if(buttonFlag == 1){

            // Clear flag
            buttonFlag = 0;

            // Create read buffer
            uint8 rdBuf[1];
            // Read xint, yint, and zint
            readFromRam(rdBuf, 1, ramLowerAddr, 0x12, ram.Xint);
            uint8 xint = rdBuf[0];
            readFromRam(rdBuf, 1, ramLowerAddr, 0x12, ram.Yint);
            uint8 yint = rdBuf[0];
            readFromRam(rdBuf, 1, ramLowerAddr, 0x12, ram.Zint);
            uint8 zint = rdBuf[0];

            // If x is stored
            if(xint == 0xFF){

                // Turn on red LED on for 1 second
                LED_R_Write(0);
                CyDelay(1000);
                LED_R_Write(1);
            }
            // If y is stored
            if(yint == 0xFF){

                // Turn on green LED on for 1 second
```

```
                    LED_G_Write(0);
                    CyDelay(1000);
                    LED_G_Write(1);
                }
                // If z is stored
                if(zint == 0xFF){
                    // Turn on blue LED on for 1 second
                    LED_B_Write(0);
                    CyDelay(1000);
                    LED_B_Write(1);
                }
                // Reset values
                xint = 0;
                yint = 0;
                zint = 0;
            }
        }
}
```

ACCEL_ISR.c

```
/* `#START ACCEL_ISR_intc` */
// Get all variables and necessary files
extern int accelFlag;
#include <ACCEL_INT.h>
/* `#END` */
...
...
...
CY_ISR(ACCEL_ISR_Interrupt)
{
    #ifdef ACCEL_ISR_INTERRUPT_INTERRUPT_CALLBACK
        ACCEL_ISR_Interrupt_InterruptCallback();
    #endif /* ACCEL_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START ACCEL_ISR_Interrupt` */

    // Set flag
    accelFlag = 1;

    // Clear interrupt
    ACCEL_INT_ClearInterrupt();

    /* `#END` */
}
```

TEST_ISR.c

```
/* `#START TEST_ISR_intc` */

// Get flag and file
extern volatile int buttonFlag;
#include "TEST.h"
```

```c
/* `#END` */
...
...
...
CY_ISR(TEST_ISR_Interrupt)
{
    #ifdef TEST_ISR_INTERRUPT_INTERRUPT_CALLBACK
        TEST_ISR_Interrupt_InterruptCallback();
    #endif /* TEST_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START TEST_ISR_Interrupt` */

    // Delay of 20ms for debounce
    CyDelay(20);
    // Clear interrupt
    TEST_ClearInterrupt();

    // If button is still 1 after delay, set flag, if not, don't
    if(TEST_Read() == 0){
        buttonFlag = 1;
    } else {
        buttonFlag = 0;
    }

    int counter = 0;
    // Debounce
    while(TEST_Read() != 1){
        CyDelayUs(1000);
        counter++;
        if(counter > 5000){
            // If held for more than 5 seconds, set flag as new value
and break
            buttonFlag = 2;
            break;
        }
    };
    /* `#END` */
}
```