ME 461 Laboratory #4
SPI Serial Port and the MPU-9250 IMU

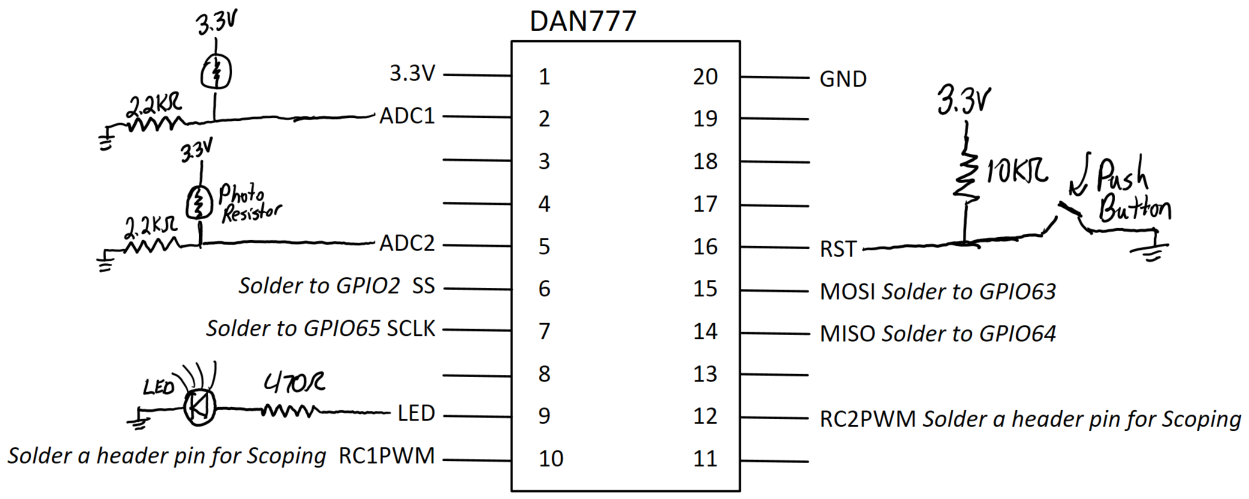
Goals:

1. Solder your break out board to connect the microphone, joystick and DAN777 (MSP430) chip to the Launchpad header pins. Also connect an LED and two photo resistors to the DAN777 chip.
2. Write code to setup SPIB. We will initially just scope the SPI Pin to see that transmission and the receive interrupt is working correctly.
3. Write code to communicate with the DAN777 chip. Your code should send two RC servo PWM values and receive two 10 bit ADC conversions
4. Write code to communicate with the MPU-9250, reading 3 accelerometer readings, 3 gyro readings and 3 compass readings.
5. Take home Exercise 1. Use the code (and debug if needed) you came up with in Lab 2, Question 2 that had you setup EPWM8A and EPWM8B to drive RC servos to drive and actual RC servo.
6. Take home Exercise 2. Setup GPIO16 as EPWM9A to drive the buzzer with different frequencies to create a short song. Also while testing this code, understand how to "Flash" your code so that each time you power on the board your short song will play.

Exercise 1:

Solder these items, after looking at the example already soldered in the lab.

1. Wire Microphone output to the ADCINB4 pin on your breakout board. So on the back side of the green board solder one of the Microphone Out pads to one of the pads connected to Launchpad pin 68 (ADCINB4).
2. Wire the two Joystick outputs to the ADCINA3 and ADCINA2 pins on your breakout board. So on the back side of the green board solder two wires. One from one of the pads connected to the Joystick's VRX pin to Launchpad pin 29 (ADCINA2). Then one from Joystick VRY pin to Launchpad pin 26 (ADCINA3).



3. Wire SS to GPIO2 on your breakout board. That is Launchpad pin 38 to DAN777 pin 6
4. Wire SCLK to GPIO65. That is Launchpad pin 47 to DAN777 pin 7.
5. Wire MOSI to GPIO63. That is Launchpad pin 55 to DAN777 pin 15.
6. Wire MISO to GPIO64. That is Launchpad pin 54 to DAN777 pin 14.
7. Wire ADC1 as shown in the figure to a photo-resistor and 2.2K ohm resistor.
8. Wire ADC2 as shown in the figure to another photo-resistor and 2.2K ohm resistor.
9. Solder a 2X1 header to the breakout pads of RC1PWM. You will just be scoping this output.
10. Solder another 2X1 header to the breakout pads of RC2PWM. You will just be scoping this output.
11. Solder a pushbutton to the RST (Reset) pin and then to Ground. Additionally solder a 10K ohm resistor to 3.3V to act as a pull-up resistor for RST. Make sure to solder the resistor such that the DAN777 chip can be removed from its socket.
12. Solder a 470 ohm resistor and an LED to the LED pin. See above figure.

Exercise 2:

For this exercise, I would like you to setup the SPI port for sending and receiving but you will not communicate with an actual chip. I would like you to setup the SPI and then every 10ms., in CPU timer 0's interrupt, transmit two bytes of data. Since the SPI pins will not be selecting any chip the transmitted data is not doing anything but it is allowing you to scope the four SPI pins and check that SPIB is setup correctly. Modify the following code after cutting and pasting it into the specified locations.

1. Copy and Paste this shell code in to your main() function somewhere below the "DINT;" line of code and before the while(1) serial_printf loop. Fill in the ??? with the correct value by reading the SPI Condensed TechRef and its register descriptions.

```
GPIO_SetupPinMux(2, GPIO_MUX_CPU1, 0);  // Set as GPIO2 and used as DAN777 SS
GPIO_SetupPinOptions(2, GPIO_OUTPUT, GPIO_PUSHPULL);  // Make GPIO2 an Output Pin
GpioDataRegs.GPASET.bit.GPIO2 = 1;  //Initially Set GPIO2/SS High so DAN777 is not selected

GPIO_SetupPinMux(66, GPIO_MUX_CPU1, 0);  // Set as GPIO66 and used as MPU-9250 SS
GPIO_SetupPinOptions(66, GPIO_OUTPUT, GPIO_PUSHPULL);  // Make GPIO66 an Output Pin
GpioDataRegs.GPCSET.bit.GPIO66 = 1;  //Initially Set GPIO66/SS High so MPU-9250 is not selected

GPIO_SetupPinMux(63, GPIO_MUX_CPU1, ???);  //Set GPIO63 pin to SPISIMOB
GPIO_SetupPinMux(64, GPIO_MUX_CPU1, ???);  //Set GPIO64 pin to SPISOMIB
GPIO_SetupPinMux(65, GPIO_MUX_CPU1, ???);  //Set GPIO65 pin to SPICLKB

EALLOW;
GpioCtrlRegs.GPBPUD.bit.GPIO63   = 0;  // Enable Pull-ups on SPI PINs Recommended by TI for SPI Pins
GpioCtrlRegs.GPCPUD.bit.GPIO64   = 0;
GpioCtrlRegs.GPCPUD.bit.GPIO65   = 0;
GpioCtrlRegs.GPBQSEL2.bit.GPIO63 = 3;  // Set prequalifier for SPI PINS
GpioCtrlRegs.GPCQSEL1.bit.GPIO64 = 3;  // The prequalifier eliminates short noise spikes
GpioCtrlRegs.GPCQSEL1.bit.GPIO65 = 3;  // by making sure the serial pin stays low for 3 clock periods.
EDIS;

// ---------------------------------------------------------------------------
SpibRegs.SPICCR.bit.SPISWRESET = ???;  // Put SPI in Reset

SpibRegs.SPICTL.bit.CLK_PHASE = 1;  //This happens to be the mode for both the DAN777 and
SpibRegs.SPICCR.bit.CLKPOLARITY = 0;  //The MPU-9250,  Mode 01.
SpibRegs.SPICTL.bit.MASTER_SLAVE = ???;  // Set to SPI Master
SpibRegs.SPICCR.bit.SPICHAR = ???;  // Set to transmit and receive 8 bits each write to SPITXBUF
SpibRegs.SPICTL.bit.TALK = ???;  // Enable transmission
SpibRegs.SPIPRI.bit.FREE = 1;  // Free run, continue SPI operation
SpibRegs.SPICTL.bit.SPIINTENA = ???;  // Disables the SPI interrupt

SpibRegs.SPIBRR.bit.SPI_BIT_RATE = ???; // Set SCLK bit rate to 1 MHz so 1us period.  SPI base clock is
                                        // 50MHZ.  And this setting divides that base clock to create SCLK's period
SpibRegs.SPISTS.all = 0x0000;  // Clear status flags just in case they are set for some reason
SpibRegs.SPIFFTX.bit.SPIRST = ???;// Pull SPI FIFO out of reset, SPI FIFO can resume transmit or receive.
SpibRegs.SPIFFTX.bit.SPIFFENA = ???;    // Enable SPI FIFO enhancements
SpibRegs.SPIFFTX.bit.TXFIFO =  0;    // Write 0 to reset the FIFO pointer to zero, and hold in reset
SpibRegs.SPIFFTX.bit.TXFFINTCLR = 1;    // Write 1 to clear SPIFFTX[TXFFINT] flag just in case it is set

SpibRegs.SPIFFRX.bit.RXFIFORESET = 0;    // Write 0 to reset the FIFO pointer to zero, and hold in reset
SpibRegs.SPIFFRX.bit.RXFFOVFCLR = 1;    // Write 1 to clear SPIFFRX[RXFFOVF] just in case it is set
SpibRegs.SPIFFRX.bit.RXFFINTCLR = ???;    // Write 1 to clear SPIFFRX[RXFFINT] flag just in case it is set
SpibRegs.SPIFFRX.bit.RXFFIENA = ???;   // Enable the RX FIFO Interrupt.  RXFFST >= RXFFIL

SpibRegs.SPIFFCT.bit.TXDLY = ???; //Set delay between transmits to 16 spi clocks. Needed by DAN777 chip

SpibRegs.SPICCR.bit.SPISWRESET = ???;    // Pull the SPI out of reset

SpibRegs.SPIFFTX.bit.TXFIFO = ???;    // Release transmit FIFO from reset.
SpibRegs.SPIFFRX.bit.RXFIFORESET = 1;    // Re-enable receive FIFO operation
SpibRegs.SPICTL.bit.SPIINTENA = 1;    // Enables SPI interrupt.  !! I don't think this is needed.  Need to Test

SpibRegs.SPIFFRX.bit.RXFFIL =???; //Interrupt Level to 2 words or more received into FIFO causes interrupt
```

2. Setup CPU Timer 0's interrupt function to be called every 10ms. This is the default so you may not need to change anything. Then inside CPU Timer 0's interrupt function call these three lines of code to tell the SPI to transmit two 8 bit values over the SPI, and because this is a SPI serial port, two 8 bit values will be received. Whenever you transmit data in a SPI serial port, you also receive. Once two 8 bit values are received into the FIFO the SPIB_RX_INT hardware interrupt function will be called. Explain to your TA why the << 8, left shift by 8, is needed?

```
Clear GPIO2 Low to act as a Slave Select.  Right now, just to scope.  Later to select DAN777 chip
GpioDataRegs.?????  = ???;
SpibRegs.SPIFFRX.bit.RXFFIL = 2;  // Issue the SPIB_RX_INT when two values are in the RX FIFO
SpibRegs.SPITXBUF = 0x4A << 8;  // 0x4A and 0xB5 have no special meaning.  Wanted to send
SpibRegs.SPITXBUF = 0xB5 << 8;  // something so you can see the pattern on the Oscilloscope
```

3.
   a. At the top of your C file add a predefinition of __interrupt void SPIB_isr(void).
   b. Then add this function to the PieVectTable like you did in Lab 3 for ADCD_ISR, ADCB_ISR, ADCA_ISR.
   c. Look up in the PIE Channel Mapping Table the interrupt number for SPIB_RX and enable this interrupt by both enabling the major interrupt in the IER: IER |= M_INT?; and enabling the correct PIEIER?.bit.???.
   d. Finally insert the SPIB_isr function and correct the ???

```
int16_t spivalue1 = 0;
int16_t spivalue2 = 0;
__interrupt void SPIB_isr(void){
```

```
spivalue1 = SpibRegs.???; // Read first 8 bit value off RX FIFO. Probably is zero since no chip
spivalue2 = SpibRegs.???; // Read second 8 bit value off RX FIFO.  Again probably zero

GpioDataRegs.???? = ???; // Set GPIO 2 high to end Slave Select.  Now to Scope. Later to deselect DAN777
// Later when actually communicating with the DAN777 do something with the data.  Now do nothing.

SpibRegs.SPIFFRX.bit.RXFFOVFCLR = 1;     // Clear Overflow flag just in case of an overflow
SpibRegs.SPIFFRX.bit.RXFFINTCLR = 1;     // Clear RX FIFO Interrupt flag so next interrupt will happen

PieCtrlRegs.PIEACK.all = PIEACK_GROUP6;  // Acknowledge INT6 PIE interrupt

}
```

4. Compile and run this code. Use the logic analyzer channel of the oscilloscope to scope SS, SCLK, MOSI and MISO. Trigger on SS. Verify that the correct clock rate and clock mode is being used. Verify that 0x4A and 0xB5 are being transmitted and probably zero is being received.

Exercise 3 Communicating with the DAN777 chip:

For this exercise, I would like you to use the DAN777 datasheet to figure out how to communicate with the DAN777 with the SPIB serial port. All the setups you performed in Exercise 2 are correct for the DAN777 chip so you have most of the code developed at this point. For example, the DAN777 communicates using SPI Mode 1 (01) and that is what you set in Exercise 2. Also, you need to communicate an odd number of 8 bit bytes to the DAN777 for one transmission so it is best to setup SPIB for 8 bit transfers just as you did in Exercise 2.

Modify your Exercise 2 code so that every 20 milliseconds it communicates two RC servo PWM values to the DAN777 chip and receives the two ADC values form the DAN777. Remember that when the SPI is transmitting it is also receiving just as the DAN777 datasheet specifies. For the values you send as the commands to the RC servos, note from the datasheet that the command value is from 1200 to 5200. So just as you did in Lab 2 when gradually increasing and then decreasing the LED brightness, every 20 milliseconds increment the RC servo commands by 10 until they reach 5200 and then start decrementing by 10 until they reach 1200 and then repeat the pattern. To see if these values are being sent correctly to the DAN777 chip, use the oscilloscope to scope the DAN777's RC1PWM and RC2PWM pins. You should see the PWM duty cycle gradually getting larger and then smaller.

For the two ADC readings that the DAN777 sends over SPI, print their value in units of volts to Tera Term every 100 milliseconds. Note that these ADC channels are only 10 bit and have a range of 0V to 3.3V. So 0 equates to 0V and 1023 equates to 3.3V. To test that ADC readings are correct, put your hand over the photoresistors and you should see the ADC voltage get small due to less light on the photoresistor. Cover just one photoresistor and see that only one ADC reading has a lower value.

Pointers to think about when developing this code:

- Figure out how many 8 bit values to write to the FIFO for one transmission to the DAN777. As long as it is less 16 you can write all the values one after the other to the FIFO and have the FIFO take care of sending each value one at a time across the SPI serial port.
- Don't forget to set the SpibRegs.SPIFFRX.bit.RXFFIL (Receive FIFO interrupt level) to the number of bytes you write to the TX FIFO each millisecond. Remember the number of words you write to the TX FIFO will be the number of words you receive in the RX FIFO and therefore cause an interrupt when all the values have been received.
- Think about how to prepare the data being sent to the DAN777. The two RC servo commands can be a value between 1200 and 5200. These numbers are larger then 255, which is the largest number stored in just 8 bits. That is the reason two 8 bit values need to be sent to the DAN777 for each RC servo command. Note that you send the RC1MSB byte first and then the RC1LSB byte second, etc. Example Code

```
uint16_t valuetosend;
uint16_t rc1msb;
uint16_t rc1lsb;
rc1msb = (valuetosend>>8) & 0xFF;
rc1lsb = valuetosend & 0xFF;
```

- The ADC values received during the SPI communication are also two 8 bit values per reading. This is because the ADC reading can be as large as 1023. These two 8 bit values will have to be put together as one 16 bit value for each ADC reading. Example Code

```
uint16_t ADCreading;
uint16_t ADC1msb;
uint16_t ADC1lsb;
ADCreading = (ADC1msb<<8) | ADC1lsb;
```

- Study the timing diagram of the DAN777 datasheet and ask questions.
- Remember that when `SpibRegs.SPICCR.bit.SPICHAR = ???` is set to send 8 bits at a time, the eight bits to be sent need to be in the upper 8 bits of the 16 bit value you write to the TX FIFO. i.e. `SpibRegs.SPITXBUF = rc1msb << 8;` The 8 bits received, though, are in the bottom 8 bits of the RX FIFO's 16 bit values so no shifting is needed for the receive.
  i.e. `ADC1msb = SpibRegs.SPIRXBUF;`
- Don't forget that some of the data you receive may not be a part of a ADC reading and should not be used.

Exercise 4 Communicating with the MPU-9250:

For this exercise you are going in initialize the MPU-9250 and then every 1 ms. read its three accelerometer readings and its three gyro readings. Use the MPU-9250 Datasheet, MPU-9250 Register Reference and especially my MPU-9250 Addendum for the explanation on how to fill in the needed code below.

First finish the setupSpib() function given below. Much of the code is given to you but you will need to add to this function the parts described. Copy this function to the bottom of your project's C file create a predefinition of the function at the top of your C file. Then make sure to call setupSpib() somewhere in main() after the DINT; statement and before interrupts are enabled.

```
void setupSpib(void) //Call this function in main() somewhere after the DINT; line of code.
{
   int16_t temp = 0;
Step 1.
   // cut and paste here all the SpibRegs initializations you found for part 3.  Change so that 16 bits are
transmitted each TX FIFO write and change the delay in between each transfer to 0.   Also don't forget to
cut and paste the GPIO settings for GPIO2, 63, 64, 65, 66 which are also a part of the SPIB setup.
   SpibRegs.SPICCR.bit.SPICHAR      = 0xF;
   SpibRegs.SPIFFCT.bit.TXDLY       = 0x00;
   //-----------------------------------------------------------------------------------------------------------------

Step 2.
   // perform a multiple 16 bit transfer to initialize MPU-9250 registers 0x13,0x14,0x15,0x16
   // 0x17, 0x18, 0x19, 0x1A, 0x1B, 0x1C 0x1D, 0x1E, 0x1F.  Use only one SS low to high for all these writes
   // some code is given, most you have to fill you yourself.
   GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;  // Slave Select Low

   // Perform the number of needed writes to SPITXBUF to write to all 13 registers
   // To address 00x13 write 0x00
   // To address 00x14 write 0x00
   // To address 00x15 write 0x00
   // To address 00x16 write 0x00
   // To address 00x17 write 0x00
   // To address 00x18 write 0x00
   // To address 00x19 write 0x13
   // To address 00x1A write 0x02
   // To address 00x1B write 0x00
   // To address 00x1C write 0x08
   // To address 00x1D write 0x06
   // To address 00x1E write 0x00
   // To address 00x1F write 0x00

   // wait for the correct number of 16 bit values to be received into the RX FIFO
   while(SpibRegs.SPIFFRX.bit.RXFFST !=???);
   GpioDataRegs.GPCSET.bit.GPIO66 = 1; // Slave Select High
```

```
  temp = SpibRegs.SPIRXBUF;
  // ???? read the additional number of garbage receive values off the RX FIFO to clear out the RX FIFO
  DELAY_US(10);  // Delay 10us to allow time for the MPU-2950 to get ready for next transfer.
```

Step 3.

```
   // perform a multiple 16 bit transfer to initialize MPU-9250 registers 0x23,0x24,0x25,0x26
  // 0x27, 0x28, 0x29.  Use only one SS low to high for all these writes
  // some code is given, most you have to fill you yourself.
  GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;  // Slave Select Low

  // Perform the number of needed writes to SPITXBUF to write to all 13 registers
  // To address 00x23 write 0x00
  // To address 00x24 write 0x40
  // To address 00x25 write 0x8C
  // To address 00x26 write 0x02
  // To address 00x27 write 0x88
  // To address 00x28 write 0x0C
  // To address 00x29 write 0x0A

  // wait for the correct number of 16 bit values to be received into the RX FIFO
  while(SpibRegs.SPIFFRX.bit.RXFFST !=???);
  GpioDataRegs.GPCSET.bit.GPIO66 = 1; // Slave Select High
  temp = SpibRegs.SPIRXBUF;
  // ???? read the additional number of garbage receive values off the RX FIFO to clear out the RX FIFO
  DELAY_US(10);  // Delay 10us to allow time for the MPU-2950 to get ready for next transfer.
```

Step 4.

```
  // perform a single 16 bit transfer to initialize MPU-9250 register 0x2A
  GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
  // Write to address 0x2A the value 0x81

  // wait for one byte to be received
  while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
  GpioDataRegs.GPCSET.bit.GPIO66 = 1;
  temp = SpibRegs.SPIRXBUF;
  DELAY_US(10);

  // The Remainder of this code is given to you and you do not need to make any changes.
  GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
  SpibRegs.SPITXBUF = (0x3800 | 0x0001);  // 0x3800
  while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
  GpioDataRegs.GPCSET.bit.GPIO66 = 1;
  temp = SpibRegs.SPIRXBUF;
  DELAY_US(10);
  GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
  SpibRegs.SPITXBUF = (0x3A00 | 0x0001);  // 0x3A00
  while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
  GpioDataRegs.GPCSET.bit.GPIO66 = 1;
```

```
temp = SpibRegs.SPIRXBUF;
DELAY_US(10);
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x6400 | 0x0001);  // 0x6400
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;
DELAY_US(10);
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x6700 | 0x0003); // 0x6700
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;
DELAY_US(10);
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x6A00 | 0x0020);  // 0x6A00
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;
DELAY_US(10);
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x6B00 | 0x0001); // 0x6B00
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;
DELAY_US(10);
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x7500 | 0x0071);  // 0x7500
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;
DELAY_US(10);
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x7700 | 0x00EB); // 0x7700
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;
DELAY_US(10);
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x7800 | 0x0012); // 0x7800
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;
DELAY_US(10);
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x7A00 | 0x0010); // 0x7A00
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
```

```
    temp = SpibRegs.SPIRXBUF;
    DELAY_US(10);
    GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
    SpibRegs.SPITXBUF = (0x7B00 | 0x00FA); // 0x7B00
    while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
    GpioDataRegs.GPCSET.bit.GPIO66 = 1;
    temp = SpibRegs.SPIRXBUF;
    DELAY_US(10);
    GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
    SpibRegs.SPITXBUF = (0x7D00 | 0x0021); // 0x7D00
    while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
    GpioDataRegs.GPCSET.bit.GPIO66 = 1;
    temp = SpibRegs.SPIRXBUF;
    DELAY_US(10);
    GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
    SpibRegs.SPITXBUF = (0x7E00 | 0x0050); // 0x7E00
    while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
    GpioDataRegs.GPCSET.bit.GPIO66 = 1;
    temp = SpibRegs.SPIRXBUF;
    DELAY_US(50);

    // Clear SPIB interrupt source just in case it was issued due to any of the above initializations.
    SpibRegs.SPIFFRX.bit.RXFFOVFCLR=1;  // Clear Overflow flag
    SpibRegs.SPIFFRX.bit.RXFFINTCLR=1;  // Clear Interrupt flag
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP6;
}
```

Since we are now communicating to the MPU-9250 in 16 bit words instead of 8 bit words we need to comment out all code that communicated with the DAN777 chip. You should, though, use your DAN777 code as a guide to complete these next steps. Also my MPU-9250 Addendum will help you with these steps.

Now every 1ms inside your CPU Timer 0 interrupt function transmit the correct 16 bit values and correct number of 16 bit values to the MPU-9250 so that it will transmit back the three accelerometer readings and the three gyro readings. *I am going to leave reading and processing the magnetometer readings as a possible final project for the class, so we will not worry about them for this lab.* Make sure to set SpibRegs.SPIFFRX.bit.RXFFIL to the correct value so that the SPIB interrupt function will be called when the SPI transmission from the master to the slave and also from the slave to the master is complete. Remember these transmissions happen at the same time.

Inside the SPIB interrupt function, make sure to pull Slave Select high. Then read the three accelerometer integer readings and the three gyro integer readings. Scale the accelerometer readings to units of g. Remember the initialization chose the range of -4g to 4g for the accelerometers. Also scale the gyro readings to units of degrees/second. The initialization chose the range of -250 to 250 degrees per second. Print these six sensor readings to Tera Term every 200ms.

<u>Take Home Exercise RCServo:</u>

In Lab 2 Question number 2 at the end of the lab, you developed code to initialize EPWM8A and EPWM8B to drive RC Servo (Hobby) motors. RC servos are driven to a position (angle) by changing the duty cycle between about 3% to 13%. The carrier frequency of the PWM signal needs to be 50 Hz. In Lab 2's question your code setup EPWM8 and command the duty cycle to 8% which should hold the RC servo at its zero angle. With this take home exercise, I will give you one RC servo to keep. Your job will be to try out / debug your code from Lab 2's question and drive your RC servo. Once you get the PWM signal working correctly, demo your RCservo moving to a number of different angles repeatedly. For example, every 500ms. you could move the RCservo to one angle and then the next 500ms. to another angle, etc. For the video demo, make sure to move the RCservo to at least 3 angles repeatedly. You can of course have your demo move the RCservo to more angles. Also plug the single RCservo that I am giving you into both three pin header connectors to check out that you setup both EPWM8A and EPWM8B correctly. Note that you should already have everything solder for driving the two RC servos.

<u>Take Home Exercise Buzzer Song:</u>

- Create at least a 20 note song or beep sequence your system will play when your program first runs.
- GPIO16, which can be set to be EPWM9A, is connected to the buzzer. So initially set GPIO16 as EPWM9A.
- Setup EPWM9A so that it generates a square wave. Note that you can set AQCTLA.bit.ZRO to "toggle" and AQCTLA.bit.CAU to "no action". With this setting, CMPA is not used and you just change TBPRD to be the period of the note you would like to play.
- Dedicate CPU Timer 1 to play your notes at a timer period you choose. For example if you want to play 4 notes in one second you would set CPU Timer 1's period to 250 milliseconds.
- To play a rest note you could set the EPWM9A period to a low note that we cannot hear.
- When CPU Timer 1's interrupt service routine has played the last note of your song, instead of repeating the song and annoying all of us, turn EPWM9A off. Do this inside CPU Timer1's interrupt service routine by turning EPWM9A back to GPIO16 and set GPIO16 to low.
- Look online for the frequency of different notes.

<u>Additional Questions:</u>

1. In the initialization of the MPU-9250 you were told what values to write to certain registers. I would like you to read the Register Map document and explain how these following register assignments setup the MPU-9250. Setting CONFIG (address 0x1A) to 0x2, GYRO_CONFIG (0x1B) to 0x0, ACCEL_CONFIG (0x1C) to 0x8 and ACCEL_CONFIG2 (0x1D) to 0x6.

2. I wanted to get to working with the I2C serial port and talking to the BQ32000 real-time clock chip on your break out board but there was no time. So I am hoping to give you an exercise with this chip later in the semester. For this question, I want you to start becoming familiar with the chip. Skim through its data sheet and answer the following questions. BQ32000 Data Sheet
    a. Briefly what does this chip do? What is its purpose?
    b. Find all the BQ32000 registers and their size in bits. What are the assigned addresses of each of these registers?