Pre-lab

1. TMP36 is an external temperature sensor. Briefly explain how this sensor works.
TMP36 use the fact as temperature increases, the voltage across a diode increases at a known rate. (Technically, this is actually the voltage drop between the base and emitter - the Vbe - of a transistor.) By precisely amplifying the voltage change, it is easy to generate an analog signal that is directly proportional to temperature.

2. How do accelerometer sensors work?
Accelerometers measure proper acceleration, meaning acceleration due to any force, including the force of gravity, unlike other coordinate acceleration, being the acceleration in a fixed coordinate system.

3. Is top of the stack different for push and pull operations?
No, the top of the stack is the same for push and pull operations since it applies the LIFO method (last in, first out)

4. How is the memory cell notation different for stack compared to a regular memory bank?
The SP usually starts pushing the top address of RAM, ie the stack grows down.

5. Explain what case A and case B, are when it comes to stack operations.
- Pushing when SP points at TOS
- Popping when SP Point at TOS

6. What are the RTN notation for push and pop?
Push:
(SP) <- src
SP <- SP - N

Pull:
SP <- SP + N
Dst <- (SP)
(Depending on if it is CASE A or B

7. How is a subroutine executed? Explain the roles of stack and program counter.
When reaching a subroutine, the contents of the PC is pushed onto the stack, while maintaining the address for the next instruction. After the subroutine is executed, the next instruction is called and execution is resumed.

8. What is an orthogonal CPU?
A CPU is said to orthogonal if all its registers and addressing modes can be used as operands, except for the immediate mode as a destination

9. What are the advantages and disadvantages of using an interrupt?
- Advantages:
    - Preferred when working with low power mode
    - Using interrupts leads to less CPU cycles and less power consumption
- Disadvantages:
    - susceptible to false triggering of interrupts so in noisy environments it might be better to use polling to get a more accurate functionality
    - If not handled properly, interrupt driven systems may have delayed responses to different IRQs

10. What are maskable and non-maskable interrupts?
Maskable interrupts are interrupts that can be turned off (modified), non-maskable interrupts cannot be turned off (modified)

11. Explain the 6 steps for servicing interrupts
- Finish the instruction being executed
- Save the current PC value and the SR onto the stack
- Clear the global interrupt enable flag
- Load the PC with the address of the ISR to be executed.
- Execute the corresponding ISR.
- Restore the PC and any other register that was saved onto the stack in Step

12. What is a brown-out in electrical circuit?

Lab

Q1: Parts of code responsible for capturing temperature:

```
void Mode4(void)
{
  // One time initialization of header and footer transmit package
  TX_Buffer[0] = 0xFA;
  TX_Buffer[6] = 0xFE;

  // variable initialization
  ADCTemp = 0;
  temp = 0;
  WriteCounter = 0;
  active = 1;
  ULPBreakSync = 0;
  counter = 0;

  // One time setup and calibration
  SetupThermistor();
  CalValue = CalibrateADC();
  while((mode == TEMP_MEAS) && (UserInput == 0))
          {
          // Take 1 ADC Sample
          TakeADCMeas();
          if (ADCResult >= CalValue)
          {
          temp = DOWN;
          ADCTemp = ADCResult - CalValue;
          }
          else
          {
          temp = UP;
          ADCTemp = CalValue - ADCResult;
          }

          if((ULP==1) && (UserInput == 0))
          {
          // P3.4- P3.7 are set as output, low
          P3OUT &= ~(BIT4 + BIT5 + BIT6 + BIT7);
    P3DIR |= BIT4 + BIT5 + BIT6 + BIT7;
          // PJ.0,1,2,3 are set as output, low
          PJOUT &= ~(BIT0 + BIT1 + BIT2 + BIT3);
    PJDIR |= BIT0 + BIT1 + BIT2 + BIT3;
```

```
        // Transmit break packet for GUI freeze
        if(!(ULPBreakSync))
        {
        TXBreak(mode);
        ULPBreakSync++;
        }

        }
        if((ULP==0) && (UserInput == 0))
        {
        ULPBreakSync = 0;
        WriteCounter++;
        if(WriteCounter > 300)
        {
        LEDSequence(ADCTemp,temp);
        // Every 300 samples
        // Transmit 7 Bytes
        // Prepare mode-specific data
        // Standard header and footer
        WriteCounter = 0;
        TX_Buffer[1] = 0x04;
        TX_Buffer[2] = counter;
        TX_Buffer[3] = 0x00;
        TX_Buffer[4] = 0x00;
        TX_Buffer[5] = 0x00;
        TXData();
        }
        }
        }
        // turn off Thermistor bridge for low power
        ShutDownTherm();

}
```

Q2: Parts of code responsible for Accelerometer use:

```
void Mode3(void)
{

  // One time initialization of header and footer transmit package
  TX_Buffer[0] = 0xFA;
  TX_Buffer[6] = 0xFE;

  // variable initialization
  active = 1;
  ADCTemp = 0;
  temp = 0;
  WriteCounter = 0;
  ULPBreakSync = 0;
  counter = 0;

  // One time setup and calibration
  SetupAccel();
  CalValue = CalibrateADC();
  while ((mode == ACCEL_MEAS) && (UserInput == 0))
        {
        // Take 1 ADC Sample
        TakeADCMeas();
        if (ADCResult >= CalValue)
        {
        temp = DOWN;
        ADCTemp = ADCResult - CalValue;
        }
        else
        {
        temp = UP;
        ADCTemp = CalValue - ADCResult;
        }
        if((ULP==1) && (UserInput == 0))
        {
        // P3.4- P3.7 are set as output, low
        P3OUT &= ~(BIT4 + BIT5 + BIT6 + BIT7);
        P3DIR |= BIT4 + BIT5 + BIT6 + BIT7;
        // PJ.0,1,2,3 are set as output, low
        PJOUT &= ~(BIT0 + BIT1 + BIT2 + BIT3);
        PJDIR |= BIT0 + BIT1 + BIT2 + BIT3;
        // Transmit break packet for GUI freeze
        if(!(ULPBreakSync))
        {
```

```
        TXBreak(mode);
        ULPBreakSync++;
        }
        }
        if((ULP==0) && (UserInput == 0))
        {
        ULPBreakSync = 0;
        WriteCounter++;
        if(WriteCounter > 300)
        {
        LEDSequence(ADCTemp,temp);
        // Every 300 samples
        // Transmit 7 Bytes
        // Prepare mode-specific data
        // Standard header and footer
        WriteCounter = 0;
        TX_Buffer[1] = 0x03;
        TX_Buffer[2] = counter;
        TX_Buffer[3] = 0x00;
        TX_Buffer[4] = 0x00;
        TX_Buffer[5] = 0x00;
        TXData();
        }
        }
        }
        // end while() loop
        // turn off Accelerometer for low power
        ShutDownAccel();
}
```

Q3:

```
/*******************************************************************************
 *
 * main.c
 * User Experience Code for the MSP-EXP430FR5739
 *
 *
 * Copyright (C) 2010 Texas Instruments Incorporated - http://www.ti.com/
 *
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *        Redistributions of source code must retain the above copyright
 *        notice, this list of conditions and the following disclaimer.
 *
 *        Redistributions in binary form must reproduce the above copyright
 *        notice, this list of conditions and the following disclaimer in the
 *        documentation and/or other materials provided with the
 *        distribution.
 *
 *        Neither the name of Texas Instruments Incorporated nor the names of
 *        its contributors may be used to endorse or promote products derived
 *        from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 *  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 *  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 *  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 *  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 *  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 *  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 *  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 *  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * Created: Version 1.0 04/13/2011
 *        Version 1.1 05/11/2011
 *        Version 1.2 08/31/2011
 *
 *******************************************************************************/
```

```
#include "msp430fr5739.h"
#include "FR_EXP.h"
const unsigned char LED_Menu[] = {0x80,0xC0,0xE0,0xF0, 0xF8, 0xFC, 0xFE, 0xFF};
// These golabal variables are used in the ISRs and in FR_EXP.c
volatile unsigned char mode = 0;
volatile unsigned char UserInput = 0;
volatile unsigned char ULP =0;
volatile unsigned int *FRAMPtr = 0;
volatile unsigned char active = 0;
volatile unsigned char SwitchCounter=0;
volatile unsigned char Switch1Pressed=0;
volatile unsigned char Switch2Pressed=0;
volatile unsigned int ADCResult = 0;
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                     // Stop WDT
  SystemInit();                             // Init the Board
  StartUpSequence();                        // Light up LEDs

  // Setup ADC data storage pointer for Modes 3&4
  FRAMPtr = (unsigned int *) ADC_START_ADD;

  while(1)
  {
        // Variable initialization
        active = 0;
        Switch2Pressed = 0;
        ULP = 0;
        // Wait in LPM4 for user input
        __bis_SR_register(LPM4_bits + GIE);         // Enter LPM4 w/interrupt
        __no_operation();                   // For debugger

        // Wake up from LPM because user has entered a mode
//      switch(mode)
//      {
//      case MAX_FRAM_WRITE:
//      Mode1();
//      break;
//
//      case SLOW_FRAM_WRITE:
//      Mode2();
//      break;
//
//      case ACCEL_MEAS:
```

```
//          Mode3();
//          break;
//
//          case TEMP_MEAS:
//          Mode4();
//          break;
//
//          default:
//          // This is not a valid mode
//          // Blink LED1 to indicate invalid entry
//          // Switch S2 was pressed w/o mode select
//          while((mode > 0x08)&& (UserInput == 0))
//          {
//          P3OUT ^= BIT7;
//          LongDelay();
//          }
//          break;
//          }
  }
}

// Interrupt Service Routines
/*************************************************************************//**
 * @brief  Port 4 ISR for Switch Press Detect
 *
 * @param  none
 *
 * @return none
 ****************************************************************************/
#pragma vector=PORT4_VECTOR
__interrupt void Port_4(void)
{
  // Clear all LEDs
  PJOUT &= ~(BIT0 +BIT1+BIT2+BIT3);
  P3OUT &= ~(BIT4 +BIT5+BIT6+BIT7);

  switch(__even_in_range(P4IV,P4IV_P4IFG1))
          {
          case P4IV_P4IFG0:              // Button 1
          DisableSwitches();
          Switch2Pressed = 0;
          UserInput = 1;
          P4IFG &= ~BIT0;                          // Clear P4.0 IFG
          PJOUT = LED_Menu[SwitchCounter];
```

```
        P3OUT = LED_Menu[SwitchCounter];
//      P3OUT = LED_Menu[SwitchCounter];
        SwitchCounter++;
        if (SwitchCounter>7)
        {
        SwitchCounter =0;
//      Switch1Pressed++;
        }
        StartDebounceTimer(0);              // Reenable switches after debounce
        break;

        case P4IV_P4IFG1:           // Button 2
        DisableSwitches();
        Switch2Pressed = 1;
        UserInput = 1;
        P4IFG &= ~BIT0;                     // Clear P4.0 IFG
//      P3OUT = LED_Menu[SwitchCounter];
        SwitchCounter--;
        if (SwitchCounter<0)
        {
        SwitchCounter =7;
//      Switch1Pressed++;
        }
        PJOUT = LED_Menu[SwitchCounter];
        P3OUT = LED_Menu[SwitchCounter];
        StartDebounceTimer(0);              // Reenable switches after debounce
        break;

        default:
        break;
        }
}
/*************************************************************************//**
 * @brief  Timer A0 ISR for MODE2, Slow FRAM writes, 40ms timer
 *
 * @param  none
 *
 * @return none
 ****************************************************************************/
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
  __bic_SR_register_on_exit(LPM4_bits);
}
```

```
/*************************************************************************//**
 * @brief  ADC10 ISR for MODE3 and MODE4
 *
 * @param  none
 *
 * @return none
 ****************************************************************************/
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
  switch(__even_in_range(ADC10IV,ADC10IV_ADC10IFG))
  {
        case ADC10IV_NONE: break;                 // No interrupt
        case ADC10IV_ADC10OVIFG: break;           // conversion result overflow
        case ADC10IV_ADC10TOVIFG: break;          // conversion time overflow
        case ADC10IV_ADC10HIIFG: break;           // ADC10HI
        case ADC10IV_ADC10LOIFG: break;           // ADC10LO
        case ADC10IV_ADC10INIFG: break;           // ADC10IN
        case ADC10IV_ADC10IFG:
        ADCResult = ADC10MEM0;
        *FRAMPtr = ADCResult;
        FRAMPtr++;
        // Pointer round off, once 0x200 locations are written, the pointer
        // rolls over
        if (FRAMPtr > (unsigned int *)ADC_END_ADD)
                FRAMPtr = (unsigned int *) ADC_START_ADD;
        __bic_SR_register_on_exit(CPUOFF);
        break;                    // Clear CPUOFF bit from 0(SR)
        default: break;
  }
}
/*************************************************************************//**
 * @brief Timer A1 ISR for debounce Timer
 *
 * @param  none
 *
 * @return none
 ****************************************************************************/
#pragma vector = TIMER1_A0_VECTOR
__interrupt void Timer1_A0_ISR(void)
{
  TA1CCTL0 = 0;
  TA1CTL = 0;
  EnableSwitches();
```

```
}
```