EEL 4742C: Embedded Systems
Name: Hamzah Ullah
Lab 4: Interrupts & Low-Power Modes

Introduction:

This lab covers an introduction to the interrupts of the clock and how to use low power mode to efficiently code the board.

Part 1: Timer's Continuous Mode with Interrupt

This part of the lab introduces us to the different concepts of what the makes up parts of the interrupt which includes the Global Interrupt enable, Interrupt enable bit, Interrupt flag bit, interrupt service routine and vector table. The global interrupt enable is the on off switch for all interrupts which is located in the status register. It is controlled by intrinsic functions. The interrupt enable bit is allows us to control individual functions such as the timer port and global interrupts. An example is the rollback to zero event which we use in this part of the lab for continuous mode. The Interrupt flag bit tells us when the interrupt occurs. The hardware raises it. Next, we have the interrupt service routine which is a special function. Its purpose is to go through the functions of the interrupt launched by the vector table. The vector table is a list of ISRs that points to its location. It is located at the top of the memory.
This lab we use continuous mode and the A1 vector to cause an interrupt of 2 seconds, and flash the LEDs. Below is the corresponding code.

Questions

Delay of 2 second

~7 flashes

If we don't clear the flag in the ISR, the LED does not turn off and the timer doesn't continue.

The CPU is staying in the for loop, iterating through the timer waiting for the flag to clear when raised.

The hardware is calling the ISR. The hardware calls and finds it by the vector table. The vector table points to where it is which is called in the software. When the vector table is called, the ISR is triggered by the hardware. We are simply pointing to the appropriate vector so the computer can find the ISR.

Flashing the LEDs w/ a 1 second delay

```c
#include <msp430fr6989.h>
#define redLED BIT0
#define greenLED BIT7

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;        //Enable the GPIO pins

    P1DIR |= redLED;            //Direct pin as output
    P9DIR |= greenLED;          //Direct pin as output
    P1OUT &= ~redLED;           //Turn LED off
    P9OUT &= ~greenLED;         //Turn LED off

    //Configure ACLK to the 32khz crystal
    config_ACLK_to_32KHz_crystal();

    //Timer_A configuration (fill the line below)
    //Use ACLK, divide by 1, continuous mode, TAR cleared,
    //enable interrupt for rollback to zero event
    TA0CTL =TASSEL_1 | ID_0 | MC_2 | TACLR | TAIE;

    //ensure the flag is cleared at the start
    TA0CTL &= ~TAIFG;

    //Enable global interrupt bit (call an intrinsic function)
    _enable_interrupt();

    //Infinite loop... the code waits here between interrupts
    for(;;){}
}
```

```c
void config_ACLK_to_32KHz_crystal()
    {
        //By default, ACLK runs on LFMODCLK at 5MHz/128 = 39kHz

        //Reroute pins to LFXIN/LFXOUT functionality
        PJSEL1 &= ~BIT4;
        PJSEL0 |= BIT4;

        //Wait until the oscillator fault flags remain cleared
        CSCTL0 = CSKEY;
        do
        {
            CSCTL5 &= ~LFXTOFFG;    //local fault flag
            SFRIFG1 &= ~OFIFG;      //Global fault flag
        }
        while((CSCTL5 & LFXTOFFG) != 0);

        CSCTL0_H = 0; //lock CS registers
        return;
    }
//*******Writing the ISR*********
#pragma vector = TIMER0_A1_VECTOR //Link the ISR to the Vector
__interrupt void T0A1_ISR()
{
    //Toggle both LEDs
    P1OUT ^= redLED;
    P9OUT ^= greenLED;
    //clear the TAIFG flag
    TA0CTL &= ~TAIFG;
}
```

Part 2: Timer's Up Mode with Interrupt

Because we are using a different type of clock mode, we have to call a different vector to launch the appropriate ISR. In this part of the lab, we use up mode to create a delay and trigger the interrupt to alternate between red and green lights. One important distinction between up mode and continuous, is the the vector A0s flag in up mode is cleared by the hardware, so we do not have to code a way to clear the flag at the end of each interval. Below is the code to flash the LEDs at different intervals.

Flash the LEDs with up mode, 1 second delay between green and red, alternating

```
#include <msp430fr6989.h>
#define redLED BIT0
#define greenLED BIT7

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;        //Enable the GPIO pins

    P1DIR |= redLED;             //Direct pin as output
    P9DIR |= greenLED;           //Direct pin as output
    P1OUT &= ~redLED;            //Turn LED off
    P9OUT |= greenLED;          //Turn LED off

    //Configure ACLK to the 32khz crystal
    config_ACLK_to_32KHz_crystal();

    //configure channel 0 for up mode with interrupt
    TA0CCR0 = (32768-1); //fill to get 1 second at 32khz
    TA0CCTL0 |= CCIE;//enable channel 0 CCIE bit
    TA0CCTL0 &= ~CCIFG; //clear channel 0 CCIFG bit
    //Use ACLK, divide by 1, up, TAR cleared,(leaves TAIE = 0)
    TA0CTL =TASSEL_1 | ID_0 | MC_1 | TACLR;


    //Enable global interrupt bit (call an intrinsic function)
    _enable_interrupt();

    //Infinite loop... the code waits here between interrupts
    for(;;){}
}
```

```
void config_ACLK_to_32KHz_crystal()
    {
        //By default, ACLK runs on LFMODCLK at 5MHz/128 = 39kHz
        //Reroute pins to LFXIN/LFXOUT functionality
        PJSEL1 &= ~BIT4;
        PJSEL0 |= BIT4;

        //Wait until the oscillator fault flags remain cleared
        CSCTL0 = CSKEY;
        do
        {
            CSCTL5 &= ~LFXTOFFG;    //local fault flag
            SFRIFG1 &= ~OFIFG;      //Global fault flag
        }
        while((CSCTL5 & LFXTOFFG) != 0);

        CSCTL0_H = 0; //lock CS registers
        return;
    }
//*******Writing the ISR*********
#pragma vector = TIMER0_A0_VECTOR //Link the ISR to the Vector
__interrupt void T0A0_ISR()
{
    //Toggle both LEDs
    P1OUT ^= redLED;
    P9OUT ^= greenLED;
    //hardware clears the flag(CCIFG in TA0CCTL0
}
```

Part 3: Push Button with Interrupt

This part of the lab requires us to use the push buttons as interrupts where they are mapped to the LEDs. To do this we have to create interrupts on the falling edge as opposed to the rising edge. To do this we code the P1IES to BUT1 | BUT2 indicating that at those bits, the interrupt is set to occur. Furthermore to clear the interrupt flag is just like clearing a field in data by inverse ANDing it to the buttons bit (P1IFG & ~(BUT1|BUT2)). Using the push buttons requires us to use the Port1 vector, which interrupts when a button is pushed. Using the same code we use to toggle the LEDs, we check if the flag and the buttons bits are raised before enabling the interrupt, than clearing the flag by the inverse of the button press. Below is the code and questions asked in this portion of the lab demonstrating how the buttons toggle the LED on and off.

Questions

The code does not work flawlessly, after 30 buttons with both the green and red LED, I got somewhere between 2 to 5 failures each.

The success rate is between 83.33% and 93.33%(25/30 – 28/30 range with each button)

Button toggles LED ON and OFF

```
#include <msp430fr6989.h>
#define redLED BIT0
#define greenLED BIT7
#define BUT1 BIT1 //Button S1 at Port 1.1
#define BUT2 BIT2 //Button S2 at Port 1.2

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;        //Enable the GPIO pins

    P1DIR |= redLED;            //Direct pin as output
    P9DIR |= greenLED;          //Direct pin as output
    P1OUT &= ~redLED;            //Turn LED off
    P9OUT &= ~greenLED;          //Turn LED off

    P1DIR &= ~(BUT1|BUT2);      //0: input
    P1REN |= (BUT1|BUT2);       //1: enable built-in resistors
    P1OUT |= (BUT1|BUT2);       //1:Built in resistors is pulled up to VCC
    P1IE  |= (BUT1|BUT2);       //1:Enable Interrupts
    P1IES |= (BUT1|BUT2);       //1:Interrupt on falling edge
    P1IFG &= ~(BUT1|BUT2);      //0:Clear the interrupt flags

    //Enable global interrupt bit (call an intrinsic function)
    _enable_interrupt();

    //Infinite loop... the code waits here between interrupts
    for(;;){}
}
//*******Writing the ISR*********
#pragma vector = PORT1_VECTOR //Link the ISR to the Vector
__interrupt void PORT1_ISR()
{   //Detect button 1 (BUT1 in P1IFG is 1
    if((P1IFG & BUT1)!=0)
    {
    //Toggle redLED
    P1OUT ^= redLED;
    //Clear BUT1 in P1IFG
    P1IFG &= ~BUT1;
    }
    //Detect button 2 (BUT2 in P1IFG is 1)
    if((P1IFG & BUT2)!=0)
    {
    //Toggle the green LED
    P9OUT ^= greenLED;
    //Clear BUT2 in P1IFG
    P1IFG &= ~BUT2;
    }
}
```

Part 4: Low-Power modes

This part of the lab simply requires us to check the graphs and look for which LPM mode is required for which clock types. It was very simple and requires less code than an enable/disable interrupt code. LPM allows us to turn off all the clocks which is used for the push. To save yourself time to look through the code, the LPM for continuous and up mode is 3 and push buttons is 4.

LPM for continuous: LPM3(Line 28)

```
1 #include <msp430fr6989.h>
2 #define redLED BIT0
3 #define greenLED BIT7
4 #define BUT1 BIT1 //Button S1 at Port 1.1
5 #define BUT2 BIT2 //Button S2 at Port 1.2
6
7 int main(void)
8 {
9     WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
10    PM5CTL0 &= ~LOCKLPM5;        //Enable the GPIO pins
11
12    P1DIR |= redLED;            //Direct pin as output
13    P9DIR |= greenLED;          //Direct pin as output
14    P1OUT &= ~redLED;            //Turn LED off
15    P9OUT &= ~greenLED;          //Turn LED off
16
17    //Configure ACLK to 32KHz crystal
18    config_ACLK_to_32KHz_crystal();
19
20    //use ACLK divide by 1, continuous mode, TAR cleared
21    //enable interrupt for rollback to zero event
22    TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR | TAIE;
23
24    //Ensure the flag is cleared at the start
25    TA0CTL &= ~TAIFG;
26
27    //Enable LPM
28    _low_power_mode_3();
29 }
```

```
30 void config_ACLK_to_32KHz_crystal()
31     {
32     //By default, ACLK runs on LFMODCLK at 5MHz/128 = 39kHz
33     //Reroute pins to LFXIN/LFXOUT functionality
34     PJSEL1 &= ~BIT4;
35     PJSEL0 |= BIT4;
36
37     //Wait until the oscillator fault flags remain cleared
38     CSCTL0 = CSKEY;
39     do
40     {
41         CSCTL5 &= ~LFXTOFFG;    //local fault flag
42         SFRIFG1 &= ~OFIFG;      //Global fault flag
43     }
44     while((CSCTL5 & LFXTOFFG) != 0);
45
46     CSCTL0_H = 0;   //lock CS registers
47     return;
48     }
49 //*******Writing the ISR*********
50 #pragma vector = TIMER0_A1_VECTOR //Link the ISR to the Vector
51 __interrupt void T0A1_ISR()
52 {
53     //Toggle both LEDs
54     P1OUT ^= redLED;
55     P9OUT ^= greenLED;
56     //clear the TAIFG flag
57     TA0CTL &= ~TAIFG;
58 }
```

LPM for up mode: LPM3(Line 29)

```
1 #include <msp430fr6989.h>
2 #define redLED BIT0
3 #define greenLED BIT7
4 #define BUT1 BIT1 //Button S1 at Port 1.1
5 #define BUT2 BIT2 //Button S2 at Port 1.2
6
7 int main(void)
8 {
9     WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
10    PM5CTL0 &= ~LOCKLPM5;         //Enable the GPIO pins
11
12    P1DIR |= redLED;             //Direct pin as output
13    P9DIR |= greenLED;           //Direct pin as output
14    P1OUT &= ~redLED;             //Turn LED off
15    P9OUT |= greenLED;           //Turn LED off
16
17    //Configure ACLK to 32KHz crystal
18    config_ACLK_to_32KHz_crystal();
19
20    //configure channel 0 for up mode with interrupt
21    TA0CCR0 = (32768-1); //fill to get 1 second at 32khz
22    TA0CCTL0 |= CCIE; //Enable channel 0 CCIE bit
23    TA0CCTL0 &= ~CCIFG; //clear channel 0 CCIFG bit
24
25    //use ACLK divide by 1, up mode, TAR cleared, (leaves TAIE=0)
26    TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;
27
28    //Enable LPM
29    _low_power_mode_3();
30 }
```

```
31 void config_ACLK_to_32KHz_crystal()
32     {
33     //By default, ACLK runs on LFMODCLK at 5MHz/128 = 39kHz
34     //Reroute pins to LFXIN/LFXOUT functionality
35     PJSEL1 &= ~BIT4;
36     PJSEL0 |= BIT4;
37
38     //Wait until the oscillator fault flags remain cleared
39     CSCTL0 = CSKEY;
40     do
41     {
42         CSCTL5 &= ~LFXTOFFG;    //local fault flag
43         SFRIFG1 &= ~OFIFG;      //Global fault flag
44     }
45     while((CSCTL5 & LFXTOFFG) != 0);
46
47     CSCTL0_H = 0;   //lock CS registers
48     return;
49     }
50
51 //*******Writing the ISR*********
52 #pragma vector = TIMER0_A0_VECTOR //Link the ISR to the Vector
53 __interrupt void T0A0_ISR()
54 {
55     //Toggle both LEDs
56     P1OUT ^= redLED;
57     P9OUT ^= greenLED;
58     //Hardware clears the flag(CCIFG in TA0CCTL0)
59 }
```

LPM for push button case: LPM4(Line 25)

```
1  #include <msp430fr6989.h>
2  #define redLED BIT0
3  #define greenLED BIT7
4  #define BUT1 BIT1 //Button S1 at Port 1.1
5  #define BUT2 BIT2 //Button S2 at Port 1.2
6
7  int main(void)
8  {
9      WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
10     PM5CTL0 &= ~LOCKLPM5;        //Enable the GPIO pins
11
12     P1DIR |= redLED;            //Direct pin as output
13     P9DIR |= greenLED;          //Direct pin as output
14     P1OUT &= ~redLED;            //Turn LED off
15     P9OUT &= ~greenLED;          //Turn LED off
16
17     P1DIR &= ~(BUT1|BUT2);      //0: input
18     P1REN |= (BUT1|BUT2);       //1: enable built-in resistors
19     P1OUT |= (BUT1|BUT2);       //1:Built in resistors is pulled up to VCC
20     P1IE  |= (BUT1|BUT2);       //1:Enable Interrupts
21     P1IES |= (BUT1|BUT2);       //1:Interrupt on falling edge
22     P1IFG &= ~(BUT1|BUT2);      //0:Clear the interrupt flags
23
24     //Enable LPM
25     _low_power_mode_4();
26 }
27 //*******Writing the ISR*********
28 #pragma vector = PORT1_VECTOR //Link the ISR to the Vector
29 __interrupt void PORT1_ISR()
30 {   //Detect button 1 (BUT1 in P1IFG is 1
31     if((P1IFG & BUT1)!=0)
32     {
33     //Toggle redLED
34     P1OUT ^= redLED;
35     //Clear BUT1 in P1IFG
36     P1IFG &= ~BUT1;
37     }
38     //Detect button 2 (BUT2 in P1IFG is 1)
39     if((P1IFG & BUT2)!=0)
40     {
41     //Toggle the green LED
42     P9OUT ^= greenLED;
43     //Clear BUT2 in P1IFG
44     P1IFG &= ~BUT2;
45     }
46 }
```

QUESTIONS:

1. **Explain the difference between using a low-power mode and not. What would be the CPU doing between interrupts for each case?**
   Low power mode turns off some or all of the clocks depending on the mode. Between interrupts, the CPU is pushed onto the stack when the interrupt occurs and is popped off after the ISR completes. In low power mode the SR or service register is pushed onto the stack before the ISR and popped off after the interrupt.

2. **We're using a module, ex. The ADC converter, and we're not sure about the vector name. We expect it should be something like ADC_VECTOR. Where do we find the exact vector name?**
   By checking the header file, we can check the top of the address reserved for vectors exclusively. We should be able to go through and find which vector the ADC corresponds to.

3. **A vector, therefore, the ISR, is shared between multiple interrupt events. Who is responsible for clearing the interrupt flags?**
   That is vector A1, which is cleared by the software.

4. **A vector, and its corresponding ISR, is used by one interrupt event exclusively. Who is responsible for clearing the interrupt flag?**
   That is the A0 Vector which is cleared by the hardware.

5. **In the first code, the ISR's name is T0A1_ISR. Is it allowed we rename the function to any other name?**
   Yes because we are simply renaming a function, we cannot rename the vector which searches the function. The ISR is simply a designation of such function being an ISR but we can rename it to what we need it to be.

6. **What happens if the ISR is supposed to clear the interrupt flag and it didn't?**
   If it doesn't clear the flag, the function within the vector for the ISR continues running till the flag is cleared, or the program finishes. For instance, if a button is supposed to register that a light turns on, that light will remain on and there will be no way to return the program to normal state without clearing the interrupt. Essentially, the clock is off till it clears. But sometimes, there is a protocol to automatically clear the flag which the MSP430 does implement to prevent bricking your device which we did not use in this instance.