EEL 4742C: Embedded Systems
Name: Hamzah Ullah
Lab 7: Concurrency via Interrupts

Introduction:
This lab focuses on multiple interrupts that are dependent on one another. Their dependency comes from enabling and disabling the interrupts of each other within their interrupt. This is called Concurrent Interrupts. Using everything we have learned thus far, there are 3 lab examples we are to demonstrate.
Part 1: Long Pulse on the LED(non-renewing interval)
This part of the lab we are to have an LED triggered by a push button initialize an interrupt(by the push button). The push button than toggles on the redLED, where we enable the next interrupt; the timer. The timer is timed for 3 seconds before that interrupt occurs. When it does occur, it turns off the LED, and disables itself right after. The concurrency comes from keeping the interrupts enabled and disabled for both the timer and buttons. Another parameter is that the button cannot be pressed again to trigger any other action, just the 3 second timer till the redLED toggles off. Below is a table of what each interrupt needs to do when their interrupt occurs.

Button	**Timer**
Turn on redLED	Turn off redLED
Enable the timer interrupt	Enable button interrupt
Clear timer interrupt flag	Clear button flag
Disable button interrupt	Disable timer interrupt
Clear button interrupt	Clear timer Flag
Set status	Reset status

To achieve a 3 second interval I divided the 32khz clock by 4, and set the TA0CCR0 to 3 times that value, so its still well below the max for the clock. Below is the code of how I implemented the code.
ACLK:32khz, Div/4 -> 8192hz/s , In up mode where TACCR0 = 24576-1 <- 8192, Timer A channel 0. LPM3

```
#include <msp430fr6989.h>
#define redLED BIT0 //Red LED at P1.0
#define BUT1 BIT1 //Button S1 at Port 1.1
int status = 0;
/**
 * main.c
 */
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;   //enable GPIO pins

    P1DIR |= redLED;    //pins as output
    P1OUT &= ~redLED;    //red off

    P1DIR &= ~BUT1; //Button 1 enable for interrupt 16-21
    P1REN |= BUT1;
    P1OUT |= BUT1;
    P1IE  |= BUT1;
    P1IES |= BUT1;
    P1IFG &= ~BUT1;

    //ACLK @ 32khz
    config_ACLK_to_32KHz_crystal();

    //8192 per second * 3 for 3 second LED timer
    TA0CCR0 = (24576-1);
    TA0CCTL0 |= CCIE;
    TA0CCTL0 &= ~CCIFG;
    // up Mode ACLK /4 div
    TA0CTL = TASSEL_1 | ID_2 | MC_1 | TACLR;

    //enable LPM3
    _low_power_mode_3();

}
//********ISR PORT1 for button**********
#pragma vector = PORT1_VECTOR //Link the ISR to the vector
__interrupt void PORT1_ISR()
{
    //Button interrupt
    //Check flag
  if((P1IFG & BUT1) != 0)
  {
      P1OUT |=redLED; //turn on LED
      TA0CCTL0 |= CCIE; //Turn on timer interrupt
      TA0CCTL0 &= ~CCIFG; //Turn on timer interrupt flag
      P1IFG &= ~BUT1; // reset button flag
      status = 1; // set status to 1
  }
  if(status == 1)
  {
      P1IE &= ~BUT1; //turn off button interrupt/flag
      P1IFG &= ~BUT1;
  }

}
//**********ISR Timer0 for timer***********
#pragma vector = TIMER0_A0_VECTOR //Link the ISR to the Vector
__interrupt void T0A0_ISR()
{

    if(status == 1)
     P1OUT &= ~redLED;//turn led off
     TA0CCTL0 &= ~CCIE;//disable timer and flag
     TA0CCTL0 &= ~CCIFG;
     P1IFG &= ~BUT1; //renable button interrupt
     P1IE |= BUT1;
     status = 0;

}
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;
```

Part 2: Long Pulse on the LED(renewing Interval)

Very similar to the first set of code, we are to make the LED respond to multiple button presses by added 3 more seconds on top of each press. To do this, we simply get rid of the disable for the button, and allow the code to keep triggering the interrupt caused by the button press, instead of turning off the button interrupt right away. I used continuous mode for this because during stress test, the light would flicker for a second, showing that the interrupt didn't truly implement, allowing a value of TACCR0 to overlap and trigger the timer interrupt to turn the LED off.

Button	**Timer**
Turn on redLED	Turn off redLED
Enable the timer interrupt	Enable button interrupt
Clear timer interrupt flag	Clear button flag
Set TA0CCR0 to TAR+3 second interval timer	Disable timer interrupt
Clear button interrupt	Clear timer Flag
Set status	Reset status

Setting TA0CCR0 to TAR+3second interval (24576-1) resets the clock time to zero, allowing it to restart the count without re-enabling and disabling the clock which is one method I tried. The status values are there for some redundancy but couldn't be used as a solo implementation.

ACLK:32khz, Div/4 -> 8192hz/s , In continuous mode where TACCR0 = 24576-1 <- 8192, Timer A channel 0.

```
#include <msp430fr6989.h>
#define redLED BIT0 //Red LED at P1.0
#define BUT1 BIT1 //Button S1 at Port 1.1
int status = 0;
/**
 * main.c
 */
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;   //enable GPIO pins

    P1DIR |= redLED;    //pins as output
    P1OUT &= ~redLED;    //red off

    P1DIR &= ~BUT1; //Button 1 enable for interrupt 16-21
    P1REN |= BUT1;
    P1OUT |= BUT1;
    P1IE  |= BUT1;
    P1IES |= BUT1;
    P1IFG &= ~BUT1;

    //ACLK @ 32khz
    config_ACLK_to_32KHz_crystal();

    //8192 per second * 3 for 3 second LED timer
    //TA0CCR0 = (24576-1);
    TA0CCTL0 |= CCIE;
    TA0CCTL0 &= ~CCIFG;
    // up Mode ACLK /4 div
    TA0CTL = TASSEL_1 | ID_2 | MC_2 | TACLR;

    //enable LPM3
    _low_power_mode_3();

}
//********ISR PORT1 for button**********
#pragma vector = PORT1_VECTOR //Link the ISR to the vector
__interrupt void PORT1_ISR()
{
    //Button interrupt
    //Check flag
//  if((P1IFG & BUT1) != 0)
//  {
      P1OUT |=redLED; //turn on LED
      TA0CCTL0 |= CCIE; //Turn on timer interrupt
      TA0CCTL0 &= ~CCIFG; //Turn on timer interrupt flag
      P1IFG &= ~BUT1; // reset button flag
      status = 1; // set status to 1
      TA0CCR0 = TA0R + (24576-1);
  if(status == 1)
  {
      //P1IE &= ~BUT1; //turn off button interrupt/flag
      P1IFG &= ~BUT1;
  }

}
//**********ISR Timer0 for timer***********
#pragma vector = TIMER0_A0_VECTOR //Link the ISR to the Vector
__interrupt void T0A0_ISR()
{

    if(status == 1)
     P1OUT &= ~redLED;//turn led off
     TA0CCTL0 &= ~CCIE;//disable timer and flag
     TA0CCTL0 &= ~CCIFG;
     P1IFG &= ~BUT1; //renable button interrupt
     P1IE |= BUT1;
     status = 0;

}
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;
    }
```

Part 3: Button Debouncing

Button debouncing occurs when the button press is read at different intervals of the clock signaling, triggering a debounce, where it flickers and/or turns off the LED. The code I used was the same from what was tested before, and it was more reliable than the earlier lab.

Button	**Timer**
Set Status	Toggle LED
Enable the timer interrupt	Enable button interrupt
Clear timer interrupt flag	Clear button flag
Disable button interrupt	Disable timer interrupt
Clear button interrupt	Clear timer Flag
	Reset status

The maximum value I was able to achieve was only 4500hz, which is a lot higher than a lot of my classmates. I don't know why I wasn't able to get a better debounce, but I tried several ways of implementing it. I kept getting a debounce when you lift up from the button than pressing down but it was difficult to tell with quick presses, so I kept raising it till it was completely stable. Using the 32khz ACLK standard with up mode, no divisor gives us plenty of time to have the bounce pass very quickly.

Button Debouncing @ 4500hz (.13 seconds).

```
#include <msp430fr6989.h>
#define redLED BIT0 //Red LED at P1.0
#define greenLED BIT7
#define BUT1 BIT1 //Button S1 at Port 1.1
#define BUT2 BIT2
/**
 * main.c
 */
int status = 0;
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;   //enable GPIO pins

    P1DIR |= redLED;    //pins as output
    P1OUT &= ~redLED;    //red off
    P9DIR |= greenLED;
    P9OUT &= ~greenLED;

    P1DIR &= ~(BUT1|BUT2); //Button 1 enable for interrupt 16-21
    P1REN |= (BUT1|BUT2);
    P1OUT |= (BUT1|BUT2);
    P1IE  |= (BUT1|BUT2);
    P1IES |= (BUT1|BUT2);
    P1IFG &= ~(BUT1|BUT2);

    //ACLK @ 32khz
    config_ACLK_to_32KHz_crystal();


    TA0CCR0 = (6000-1);
    TA0CCTL0 |= CCIE;
    TA0CCTL0 &= ~CCIFG;

    // up Mode ACLK /1 div
    TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;

    //enable LPM3
    _low_power_mode_3();

}
//********ISR PORT1 for button**********
#pragma vector = PORT1_VECTOR //Link the ISR to the vector
__interrupt void PORT1_ISR()
{
    while(status==0){
    if((P1IFG & BUT1)!= 0)
        {
            P1IFG &= ~BUT1;
            P1IE &= ~BUT1;
            TA0CCTL0 |= CCIE;
            TA0CCTL0 &= ~CCIFG;
            status = 1;
        }
    if((P1IFG & BUT2)!= 0)
    {
        P1IFG &= ~BUT2;
        P1IE &= ~BUT2;
        TA0CCTL0 |= CCIE;
        TA0CCTL0 &= ~CCIFG;
        status = 2;
    }
    }
}
//**********ISR Timer0 for timer***********
#pragma vector = TIMER0_A0_VECTOR //Link the ISR to the Vector
__interrupt void T0A0_ISR()
{
    while(status == 2)
    {
    P1IFG &= ~BUT2;
    P1IE |= BUT2;
    if((BUT2)!= 0)
    {
        P9OUT ^= greenLED;
        P1IFG &= ~BUT2;
    }
    status = 0;
    }
    while(status == 1)
    {
    P1IFG &= ~BUT1;
    P1IE |= BUT1;
    if((BUT1)!= 0)
    {
        P1OUT ^= redLED;
        P1IFG &= ~BUT1;
    }

    status = 0;
    }
    TA0CCTL0 &= ~CCIFG;
    TA0CCTL0 &= ~CCIE;
}
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;
    }
```

QUESTIONS:

1. **For the debouncing algorithm we implemented, is it possible the LED will be toggled when the button is released? Explain**
   No because it only checks for first time interval and not the last portion of the time interval. For instance, we checked .02seconds or 656hz. The debounce occurs even when we lift off the button. The only way its possible is to delay the whole time period to the point of forcing the person to hit the button once the timer has completely passed. (The button interrupt is completely turned off for a very long period within the interval).

2. **If two random pulses occur on the push button due to noise and these pulses are separated by the maximum bounce duration, will our algorithm fail? Explain**
   Yes because the max duration is only measuring from the button push , but not on every possible debounce. Essentially what happened is the first debounce is accounted for at .02seconds but the second debounce also occurred within a span of the max bounce duration. So the entire length of all debounces is TAR+.02+.02 which is TAR+.04 seconds. This means it will fail because we are only accounting for a single max debounce of .02seconds and not multiple.