EEL 4742C: Embedded Systems
Name: Daniel Vu
Lab 3 Timer Module

## Introduction

Lab 3 introduced the design and operation of a timer module, a fundamental building block in embedded systems used for generating precise delays, periodic events, and time-based control signals. The objective of this experiment was to explore how hardware timers function, how they are configured through registers, and how they interact with the microcontroller's clock system. By implementing timed delays and observing the resulting output behavior, we gained practical experience translating theoretical timing calculations into working code. This lab reinforced the importance of accurate timing in digital systems and prepared us for more advanced applications such as pulse-width modulation, event scheduling, and real-time control.

Part one
Previously we used an empty for loop which can be unreliable due to processes under the hood, By using the timer, we can configure the resolution and know precisely how long something will take. We can check if a flag is raised to determine whether the timer has filled its register and restarted.

```
// Flashing the LED with Timer_A, continuous mode, via polling
#include <msp430fr6989.h>
#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7


//*********************************
// Configures ACLK to 32 KHz crystal
void config_ACLK_to_32KHz_crystal() {
   // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz

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

   // Wait until the oscillator fault flags remain cleared
   CSCTL0 = CSKEY; // Unlock CS registers

   do {
       CSCTL5 &= ~LFXTOFFG; // Local fault flag
       SFRIFG1 &= ~OFIFG; // Global fault flag
   } while((CSCTL5 & LFXTOFFG) != 0);


   CSCTL0_H = 0; // Lock CS registers
   return;
}

/**
* main.c
*/
void main(void) {

   WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
```

```
    PM5CTL0 &= ~LOCKLPM5; // Disable GPIO power-on default high-impedance mode


    // Configure the LEDs as output
    P1DIR |= redLED;
    P9DIR |= greenLED;
    // Configure ACLK to the 32 KHz crystal (function call)
    config_ACLK_to_32KHz_crystal();
    // Configure Timer_A
    // Use ACLK, divide by 1, continuous mode, clear TAR
    TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR;

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

    // Infinite loop
    for(;;) {
    // Wait in this empty loop for the flag to raise
        while( (TA0CTL & TAIFG) == 0 ) {}
            // Do the action here
            P1OUT ^= redLED;
            P1OUT ^= greenLED;

            // Reset Flag
            TA0CTL &= ~TAIFG;
    }
}
```

Since the clock runs at 32khz, it will fill its 64k register in exactly two seconds meaning the LED will toggle every 2 seconds.

Measuring with my phone, we find that during this time the LED toggles 10 times which is expected.

If the clock is divided by 2, it will take twice as long because we are counting twice as slow. Same thing goes for 4 and 8.

Testing clock divider 2, the LED toggles every 4 seconds instead of 2.

Clock divider 4 gives 8, and 8 gives 16.

part 2.

Up mode is when the timer will count *up* to a certain number rather than just waiting for the register to fill. The number we count up to will be stored in TA0CCR0

```
// Flashing the LED with Timer_A, continuous mode, via polling
#include <msp430fr6989.h>
#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7


//*********************************
// Configures ACLK to 32 KHz crystal
void config_ACLK_to_32KHz_crystal() {
   // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz

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

   // Wait until the oscillator fault flags remain cleared
   CSCTL0 = CSKEY; // Unlock CS registers

   do {
       CSCTL5 &= ~LFXTOFFG; // Local fault flag
       SFRIFG1 &= ~OFIFG; // Global fault flag
   } while((CSCTL5 & LFXTOFFG) != 0);


   CSCTL0_H = 0; // Lock CS registers
   return;
}

/**
* main.c
*/
void main(void) {

   WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
   PM5CTL0 &= ~LOCKLPM5; // Disable GPIO power-on default high-impedance mode
```

```
    // Configure the LEDs as output
    P1DIR |= redLED;
    P9DIR |= greenLED;
    // Configure ACLK to the 32 KHz crystal (function call)
    config_ACLK_to_32KHz_crystal();
    // Configure Timer_A
    // Use ACLK, divide by 1, continuous mode, clear TAR
    TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;

    // Count up to the value stored in TA0CCR0
    // @32khz, cycles = seconds * 32,768
    float seconds = 0.1;
    TA0CCR0 = seconds * 32768;



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

    // Infinite loop
    for(;;) {
    // Wait in this empty loop for the flag to raise
        while( (TA0CTL & TAIFG) == 0 ) {}
            // Do the action here
            P1OUT ^= redLED;
            P1OUT ^= greenLED;

            // Reset Flag
            TA0CTL &= ~TAIFG;
    }
}
```

Since the clock runs at 32KHz, the amount of clock cycles we need to wait is seconds*32k. We can attach this to a variable to show that. Since we want one second, we put 32k on this and timing this for 20 seconds, we find that the LED toggles 20 times. We can adjust the variable to

be 0.1 or 0.01. At 0.001, the LED does not have enough time to fully turn on and off so it looks dim, and we begin operating at the PWM speed.

Part 3

For part 3, we will make a signal repeated. We can do this starting the timer in continuous mode when the button is pressed. When the button is released, switch to up mode, store the value of timer in TACCR0, reset the timer, and turn the LED on until the timer is complete. If the button is held for too long, enter an error state that can be cleared by pressing the other button.

```
// Flashing the LED with Timer_A, continuous mode, via polling
#include <msp430fr6989.h>
#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7
#define BUT1 BIT1 // Button S1 at P1.1
#define BUT2 BIT2 // Button S2 at P1.2



//*********************************
// Configures ACLK to 32 KHz crystal
void config_ACLK_to_32KHz_crystal() {
   // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz

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

   // Wait until the oscillator fault flags remain cleared
   CSCTL0 = CSKEY; // Unlock CS registers

   do {
       CSCTL5 &= ~LFXTOFFG; // Local fault flag
       SFRIFG1 &= ~OFIFG; // Global fault flag
   } while((CSCTL5 & LFXTOFFG) != 0);


   CSCTL0_H = 0; // Lock CS registers
   return;
}


// Helper functions
```

```
int button_is_pressed(int button) {
   return (P1IN & button) == 0;
}

void red_led(int value) {
   if (value == 1)
       P1OUT |= redLED;
   else
       P1OUT &= ~redLED;
}

void green_led(int value) {
   if (value == 1)
       P9OUT |= greenLED;
   else
       P9OUT &= ~greenLED;
}

/**
* main.c
*/
void main(void) {

   WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
   PM5CTL0 &= ~LOCKLPM5; // Disable GPIO power-on default high-impedance mode


   // Configure the LEDs as output
   P1DIR |= redLED;
   P9DIR |= greenLED;

   // Turn off the LEDs
   P1OUT &= ~redLED; // Turn LED Off
   P9OUT &= ~greenLED; // Turn LED Off


   // Configure buttons
   P1DIR &= ~BUT1; // Direct pin as input
```

```
P1REN |= BUT1;  // Enable built-in resistor
P1OUT |= BUT1; // Set resistor as pull-up

P1DIR &= ~BUT2; // Direct pin as input
P1REN |= BUT2;  // Enable built-in resistor
P1OUT |= BUT2; // Set resistor as pull-up


// Configure ACLK to the 32 KHz crystal (function call)
config_ACLK_to_32KHz_crystal();
// Configure Timer_A
// Use ACLK, divide by 1, continuous mode, clear TAR
TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR;

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

int error = 0;

// Infinite loop
for(;;) {
    // Check if in error
    if (error == 1) {
        if (!button_is_pressed(BUT2))
            continue;

        TA0CTL &= ~TAIFG;
        TA0CTL |= TACLR;
        green_led(0);
        error = 0;
    }


    // Detect when the button is pressed.
    if (button_is_pressed(BUT1)) {
        // Turn on the timer and clear the timer
        TA0CTL = (TA0CTL & ~MC_3) | TACLR | MC_2;
```

```
            // Begin while to determine when the button is released
            while (button_is_pressed(BUT1)) {
                // if the timer overflows, enter error state
                if ((TA0CTL & TAIFG) == TAIFG) {
                    error = 1;
                    break;
                }
            }

            // Check if errored
            if (error == 1) {
                green_led(1);
                continue;
            }

            // After button is released, put the timer in up, store the current
value in the upto value and reset
            TA0CTL = (TA0CTL & ~MC_3) | MC_1;
            TA0CCR0 = TA0R;
            TA0CTL |= TACLR;
            TA0CTL &= ~TAIFG;

            // Turn on the LED until the flag is turned on
            red_led(1);
            while( (TA0CTL & TAIFG) == 0 ) {}
            red_led(0);

            // turn off the timer and clear the flag
            TA0CTL = (TA0CTL & ~MC_3) | MC_0;
            TA0CTL &= ~TAIFG;

        }
    }
}
```

compute the maximum pulse delay that our code supports for all cases of the divider (1, 2, 4, 8) based on the 32 KHz crystal.

At id1, we can support two seconds (32khz fills 64k in 2 seconds). at 2, the max is 4, at 4 the max is 8, and at 8 the max is 16.

The trade off is that the resolution of accuracy will be halved with each clock division. At clock divider 1, we are accurate up to 1/32k of second. At id2, we are accurate up to 2/32k of a second.

This can be modified to measure any length delay by counting the number of times the register fills. and replaying the signal first counting to the number we got previously, the on the excess, play the number of clock cycles needed.

1. So far, we have seen two ways of timing delays: using a delay loop and using Timer A. Which approach provides more control and accuracy over the delays? Explain.
    Using timer a is more accurate because it is not attached to the subprocesses of a for loop. Different compilers may treat for loops differently. In addition, with timer a we can control which clock the timer uses and configure the speed with more versatility.
2. Explain the polling technique and how it's used in this lab with the timer.
    The polling technique is to continuously check something. In this lab, we use to always check if the flag is raised.
3. Is the polling technique a suitable choice when we care about saving battery power? Explain.
    No, each check requires power to be spent. Interrupts are a better option as power is only spent when the signal we detect triggers
4. If we write 0 to TAR using a line code, does TAIFG go to 1?
    no. This only occurs on overflow.
5. From what we have seen in this lab, which mode gives us more control over the timing duration:
the up mode or the continuous mode?
    Up Mode. in up mode we can choose which number causes TAIFG to go high instead of just register overflow at 2^16.

6. In this lab, you were given a summary of the timer's main control register, TACTL, and the fields
within. To practice reading the documentation, find this information in the Family User's Guide (slau367o) document and include a screenshot of TACTL's layout and the table describing the fields within it. This information can be found at the end of the Timer A chapter in the Family User's Guide.
pg 658



### *25.3.1 TAxCTL Register*

Timer_Ax Control Register

**Figure 25-16. TAxCTL Register**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|
| Reserved | | | | | | TASSEL | |
| rw-(0) | rw-(0) | rw-(0) | rw-(0) | rw-(0) | rw-(0) | rw-(0) | rw-(0) |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ID | | MC | | Reserved | TACLR | TAIE | TAIFG |
| rw-(0) | rw-(0) | rw-(0) | rw-(0) | rw-(0) | w-(0) | rw-(0) | rw-(0) |

**Table 25-4. TAxCTL Register Description**

| Bit | Field | Type | Reset | Description |
|---|---|---|---|---|
| 15-10 | Reserved | RW | 0h | Reserved |
| 9-8 | TASSEL | RW | 0h | Timer_A clock source select<br>00b = TAxCLK<br>01b = ACLK<br>10b = SMCLK<br>11b = INCLK |
| 7-6 | ID | RW | 0h | Input divider. These bits along with the TAIDEX bits select the divider for the input clock.<br>00b = /1<br>01b = /2<br>10b = /4<br>11b = /8 |
| 5-4 | MC | RW | 0h | Mode control. Setting MC = 00h when Timer_A is not in use conserves power.<br>00b = Stop mode: Timer is halted<br>01b = Up mode: Timer counts up to TAxCCR0<br>10b = Continuous mode: Timer counts up to 0FFFFh<br>11b = Up/down mode: Timer counts up to TAxCCR0 then down to 0000h |
| 3 | Reserved | RW | 0h | Reserved |
| 2 | TACLR | RW | 0h | Timer_A clear. Setting this bit clears TAR, the clock divider logic (the divider setting remains unchanged), and the count direction. The TACLR bit is automatically reset and is always read as zero. |
| 1 | TAIE | RW | 0h | Timer_A interrupt enable. This bit enables the TAIFG interrupt request.<br>0b = Interrupt disabled<br>1b = Interrupt enabled |
| 0 | TAIFG | RW | 0h | Timer_A interrupt flag<br>0b = No interrupt pending<br>1b = Interrupt pending |

Conclusion:
In this lab, we developed a deeper understanding of the MSP430 Timer_A module by configuring its control registers, generating precise timing intervals, and observing how different operating modes influence system behavior. Through hands-on experimentation with the TACTL register, capture/compare blocks, and interrupt-driven timing, we saw how hardware timers provide reliable, low-overhead timing compared to software-based delays. Overall, the lab reinforced how essential timers are for embedded systems—enabling accurate event scheduling, PWM generation, and time measurement—while giving us practical experience interpreting datasheet documentation and applying it to real hardware.