Experiment #3 – Timer Module

# Youssef Samwel

# [yo800238@ucf.edu](mailto:yo800238@ucf.edu)

# EEL4742C Embedded Systems

Prof. Dr. Zakhia Abichar - Section 00419

# 2/8/2024



# **Project Description**

# In this lab, we will program the microcontroller’s timer module (Timer A) to time events. We will use the timer in two modes known as the continuous mode and the up mode. Our codes will use the polling technique, which is a simple approach based on reading a flag continuously.

# **2.0 Experiment Code**

## 2.1 Flashing the LED

void ContinousMode()

{

    ConfigureContinousTimerA();

    while (true)

    {

        WaitForTimerAInterrupt();

        ClearTimerAInterruptFlag();

        ToggleRedLED();

    }

}

This code defines a function called ContinousMode that creates a continuously blinking red LED. It first sets up a timer to generate interrupts. Then, it enters an infinite loop, waiting for the timer interrupt. When the interrupt arrives, it clears the flag and toggles the red LED, effectively turning it on/off. This cycle repeats indefinitely, resulting in a continuous blinking pattern synchronized with the timer's frequency.

Write an analysis showing what delays we expect to observe if the clock is divided by 2, 4 or 8. Test these cases and report if they match what you expected.

The analysis matched the implementation on the MSP430.

## 2.2 The Up Mode

void UpMode()

{

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_UP | TACLR;

    TA0CCR0 = 3278;

    while (true)

    {

        WaitForTimerAInterrupt();

        ClearTimerAInterruptFlag();

        ToggleGreenLED();

    }

}

This code, UpMode, manages a continuously blinking green LED with slightly more control compared to ContinousMode. It sets up timer A to count up from 0 to a specific value (~0.1 seconds) using the main clock. When the timer reaches the limit, it triggers an interrupt. The code then waits for and clears the interrupt before toggling a green LED, creating a blinking pattern. This cycle repeats indefinitely, resulting in a green LED blinking roughly every 0.1 seconds.

What value of TA0CCR0 achieves a delay of 0.1 seconds? Round up to the nearest integer and test this value.

What value of TA0CCR0 achieves a delay of 0.01 seconds? Round up to the nearest integer and test this value. What do you observe?

The LED is looks to be always on, but it has a dimmer light output.

## 2.3 Application: Signal Repeater

uint16\_t RecordSignal()

{

    SetContinousTimerState(true);

    ClearTimerAInterruptFlag();

    while (IsButton1Pressed())

    {

        if (IsTimerAInterrupted())

        {

            return UINT16\_MAX;

        }

    }

    return TA0R;

}

void SignalRepeaterErrorState()

{

    LightRedLED(false);

    SetContinousTimerState(false);

    while (!IsButton2Pressed())

    {

        \_\_delay\_cycles(1.25e5);

        ToggleGreenLED();

    }

    LightGreenLED(false);

}

void TransmitSignal(uint16\_t duration)

{

    LightRedLED(false);

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_UP | TACLR;

    TA0CCR0 = duration;

    LightGreenLED(true);

    WaitForTimerAInterrupt();

    LightGreenLED(false);

}

void ApplicationSignalRepeater\_3\_3()

{

    P1DIR &= ~BUTTON1;

    P1DIR &= ~BUTTON2;

    P1REN |= BUTTON1 | BUTTON2;

    P1OUT |= BUTTON1 | BUTTON2;

    ConfigureContinousTimerA();

    SetContinousTimerState(false);

    for (;;)

    {

        if (IsButton1Pressed())

        {

            LightRedLED(true);

            uint16\_t duration = RecordSignal();

            if (duration == UINT16\_MAX)

            {

                SignalRepeaterErrorState();

                continue;

            }

            TransmitSignal(duration);

        }

    }

}

1. **RecordSignal()**
   * This function records a signal duration.
   * It first sets a continuous timer state to true.
   * Enters a while loop that checks if button 1 is pressed.
   * If the Timer A interrupt occurs during this loop, the function returns **UINT16\_MAX**.
   * Otherwise, it returns the current value of Timer A register (**TA0R**), which represents the recorded signal duration.
2. **SignalRepeaterErrorState()**
   * This function handles the error state when the signal repeater encounters an issue.
   * Turns off the red LED.
   * Sets the continuous timer state to false.
   * Enters a loop that waits for button 2 to be pressed.
   * Inside the loop, it toggles the green LED and introduces a delay using **\_\_delay\_cycles**.
   * Once button 2 is pressed, it turns off the green LED.
3. **TransmitSignal(uint16\_t duration)**
   * This function is responsible for transmitting a signal with a specified duration.
   * Turns off the red LED.
   * Configures Timer A to use the ACLK (Auxiliary Clock) as the clock source, with a division factor of 1 (**ID\_\_1**), and in up mode (**MC\_\_UP**).
   * Sets the timer's compare register (**TA0CCR0**) to the specified duration.
   * Turns on the green LED.
   * Waits for a Timer A interrupt, and once it occurs, turns off the green LED.
4. **ApplicationSignalRepeater\_3\_3()**
   * This is the main application loop.
   * Configures the GPIO pins for BUTTON1 and BUTTON2 as inputs with pull-up resistors.
   * Configures and initializes Timer A in continuous mode.
   * Enters an infinite loop where it checks if button 1 is pressed.
   * If button 1 is pressed, it lights up the red LED, records the signal duration, and then transmits the signal. If an error occurs during recording, it enters the error state handled by **SignalRepeaterErrorState()**.

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.

Explain what the tradeoff is when we change between dividers.

When increase the clock division we gain a longer duration but timer resolution. A higher division means each increment in the counter register will account for a longer duration. For example, for a 32 kHz clock with unity division we have a timer resolution of 30.5 sec whereas with division multiple of 8, the timer resolution is reduced to 244.14 sec.

## 3.0 Complete Code

#include <msp430fr6989.h>

#include <stdint.h>

#include <stdbool.h>

#define BUTTON1 BIT1

#define BUTTON2 BIT2

#define redLED BIT0   // Red LED at P1.0

#define greenLED BIT7 // Green LED at P9.7

/\*\*

 \* main.c

 \*/

void ClearTimerAInterruptFlag()

{

    TA0CTL &= ~TAIFG;

}

bool IsTimerAInterrupted()

{

    return (TA0CTL & TAIFG) ? true : false;

}

void WaitForTimerAInterrupt()

{

    while (!IsTimerAInterrupted())

    {

    }

}

bool IsButton1Pressed()

{

    return (~P1IN & BUTTON1) ? true : false;

}

bool IsButton2Pressed()

{

    return (~P1IN & BUTTON2) ? true : false;

}

// Will clear Timer A CCR0 register and clear TAIFG interrupt flag

void SetContinousTimerState(bool state)

{

    TA0CTL &= ~MC\_\_UPDOWN;

    if (state)

    {

        TA0CTL |= MC\_\_CONTINUOUS | TACLR;

        TA0CTL &= ~TAIFG;

    }

}

void ConfigureContinousTimerA()

{

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_CONTINUOUS | TACLR;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 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;

}

void LightRedLED(bool state)

{

    if (state)

        P1OUT |= redLED;

    else

        P1OUT &= ~redLED;

}

void LightGreenLED(bool state)

{

    if (state)

        P9OUT |= greenLED;

    else

        P9OUT &= ~greenLED;

}

void ToggleRedLED() { P1OUT ^= redLED; }

void ToggleGreenLED() { P9OUT ^= greenLED; }

void ContinousMode()

{

    ConfigureContinousTimerA();

    while (true)

    {

        WaitForTimerAInterrupt();

        ClearTimerAInterruptFlag();

        ToggleRedLED();

    }

}

void UpMode()

{

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_UP | TACLR;

    TA0CCR0 = 6535; // 0.1 seconds

    while (true)

    {

        WaitForTimerAInterrupt();

        ClearTimerAInterruptFlag();

        ToggleGreenLED();

    }

}

uint16\_t RecordSignal()

{

    SetContinousTimerState(true);

    ClearTimerAInterruptFlag();

    while (IsButton1Pressed())

    {

        if (IsTimerAInterrupted())

        {

            return UINT16\_MAX;

        }

    }

    return TA0R;

}

void SignalRepeaterErrorState()

{

    LightRedLED(false);

    SetContinousTimerState(false);

    while (!IsButton2Pressed())

    {

        \_\_delay\_cycles(1.25e5);

        ToggleGreenLED();

    }

    LightGreenLED(false);

}

void TransmitSignal(uint16\_t duration)

{

    LightRedLED(false);

    // this should not happen

    if (duration >= UINT16\_MAX)

        return;

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_UP | TACLR;

    TA0CCR0 = duration;

    LightGreenLED(true);

    WaitForTimerAInterrupt();

    LightGreenLED(false);

}

void ApplicationSignalRepeater\_3\_3()

{

    P1DIR &= ~BUTTON1;

    P1DIR &= ~BUTTON2;

    P1REN |= BUTTON1 | BUTTON2;

    P1OUT |= BUTTON1 | BUTTON2;

    ConfigureContinousTimerA();

    SetContinousTimerState(false);

    for (;;)

    {

        if (IsButton1Pressed())

        {

            LightRedLED(true);

            uint16\_t duration = RecordSignal();

            if (duration == UINT16\_MAX)

            {

                SignalRepeaterErrorState();

                continue;

            }

            TransmitSignal(duration);

        }

    }

}

int main(void)

{

    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins

    P1DIR |= redLED;

    P9DIR |= greenLED;

    LightRedLED(false);

    LightGreenLED(false);

    config\_ACLK\_to\_32KHz\_crystal();

    UpMode();

    return 0;

}

# **4.0 Student Q&A**

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.  
The timer approach provides more control and accuracy because it uses a crystal, and we also control the oscillation frequency. The frequency of the timer is controlled and is constant, however, the MCU master clock could change depending on operation mode or if we use a different model. Also, since we’re using the crystal, the frequency deviation will be better compared to the RC clock of the MCU.

2. Explain the polling technique and how it’s used in this lab.

We configure the timer and constantly check if the interrupt flag is set to toggle the LED. Afterwards, we must clear the interrupt flag otherwise no more interrupts will occur.

3. Is the polling technique a suitable choice when we care about saving battery power? Explain.

No, polling method is inefficient use of power, because, multiple modules are turned on that are not used but consume power, such as the CPU master clock. Disabling the mastering clock would also disable many module’s inside the CPU that are not used, reducing power consumption.

4. If we write 0 to TAR using a line code, does TAIFG go to 1?

No, TAIFG is only set to 1 when the hardware overflows the register.

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, because we can shorten the duration of the timer, whereas the continuous mode interrupt duration is constant (only control is clock frequency and clock division).

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.

A close-up of a register

Description automatically generated

# **5.0 Conclusion**

In Experiment #3, the focus was on programming the microcontroller's Timer A module to handle timing events. Two modes, continuous and up mode, were explored using a polling technique based on flag checking. The codes demonstrated the creation of a continuously blinking LED with precise timing, showcasing the effectiveness of Timer A interrupts. Clock division effects on delay durations were analyzed, highlighting the tradeoff between longer durations and reduced timer resolution.

The experiments successfully aligned with expectations, demonstrating the ability to control LED blinking intervals precisely in Up Mode. The application section showcased a practical use case of a signal repeater, utilizing Timer A and GPIO pins for recording and transmitting signals. The Q&A section emphasized the advantages of the timer approach, providing accurate control over delays, especially when utilizing a crystal for consistent timing. Additionally, it highlighted the inefficiency of the polling technique in terms of power consumption, particularly when considering battery-saving concerns.

Overall, experiment #3 provided valuable hands-on experience in configuring and utilizing Timer A for precise timing applications, underscoring its versatility and control over timing events in embedded systems. The exploration of different timing modes and clock divisions contributed to a comprehensive understanding of the microcontroller's timer capabilities.