Lab 10 Report
EEL4742C - 00446

Yousef Awad

November 2025

Contents

Introduction

This experiment investigates the advanced capabilities of the MSP430FR6989 Timer_A module, focusing on using multiple channels to handle concurrent hardware events. Objectives include implementing simultaneous timing intervals to control independent LED flashing rates in continuous mode, generating Pulse Width Modulation (PWM) signals to adjust LED brightness without CPU intervention , and utilizing Input Capture mode to timestamp external user input.

10.1 Timer's Multiple Channels

The visual comparison appears to be the same/similar to what I set the values to be at. Alongside this here is the following derivation for the number of cycles for each channel:

$$\frac{f_{clk}}{input\ divider} * desired\ delay\ in\ seconds$$

$$\frac{32768}{4} * 0.1 \approx 819 \rightarrow TA0CCR0 = 819$$
$$\frac{32768}{4} * 0.5 \approx 4096 \rightarrow TA0CCR1 = 4096$$

```c
#include "msp430fr6989.h"

#define redLED BIT0
#define greenLED BIT7
#define S1 BIT1
#define S2 BIT2

// 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;
}

```

```c
int main(void)
{
  // Configure WDT & GPIO
  WDTCTL = WDTPW | WDTHOLD;
  PM5CTL0 &= ~LOCKLPM5;

  // Configure LEDs
  P1DIR |= redLED;
  P9DIR |= greenLED;
  P1OUT &= ~redLED;
  P9OUT &= ~greenLED;

  // Configure buttons
  P1DIR &= ~(S1 | S2);
  P1REN |= (S1 | S2);
  P1OUT |= (S1 | S2);
  P1IFG &= ~(S1 | S2); // Flags are used for latched polling

  config_ACLK_to_32KHz_crystal();

  TA0CCR0 = 819; // 0.1s for red
  TA0CCTL0 |= CCIE; // Enabling channel 1 interrupt

  //        ACLK        /4      Continuous   Clear TAR
  TA0CTL = TASSEL_1 | ID_2 | MC_2       | TACLR;

  _low_power_mode_3(); // We only need ACLK

  return 0;
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR()
{
  P1OUT ^= redLED;
  TA0CCR0 += 819;
  // clearing the flag
  TA0CCTL0 &= ~CCIFG;
}

#pragma vector = TIMER0_A1_VECTOR
__interrupt void T0A1_ISR()
{
  if (TA0CCTL1 & CCIFG)
  {
    P9OUT ^= greenLED;
    TA0CCR1 += 4096; // 1/2 a second
    // clearing the flag
    TA0CCTL1 &= ~CCIFG;
  }
}
```

10.2 Using Three Channels

The durations, at least when tested by my stopwatch on my phone and on my watch, are close to the time's that I set (though, there was a slight error most likely due to human error).

```c
#include "msp430fr6989.h"

#define redLED BIT0
#define greenLED BIT7
#define S1 BIT1
#define S2 BIT2

// tracks system state between not flashing (0) and flashing (1)
volatile unsigned int state = 1;

// 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;
}

int main(void)
{
  // Configure WDT & GPIO
  WDTCTL = WDTPW | WDTHOLD;
  PM5CTL0 &= ~LOCKLPM5;

  // Configure LEDs
  P1DIR |= redLED;
  P9DIR |= greenLED;
  P1OUT &= ~redLED;
  P9OUT &= ~greenLED;

  // Configure buttons
  P1DIR &= ~(S1 | S2);
  P1REN |= (S1 | S2);
  P1OUT |= (S1 | S2);
  P1IFG &= ~(S1 | S2); // Flags are used for latched polling

  config_ACLK_to_32KHz_crystal();

  TA0CCR0  = 819;  // 0.1s
  TA0CCTL0 = CCIE; // Enabling channel 0 interrupt

  TA0CCR1  = 4096; // 0.5s
  TA0CCTL1 = CCIE; // enabling interrupt on channel 1

  TA0CCR2  = 32768; // 1s
  TA0CCTL2 = CCIE; // enabling interrupt on channel 2

  //       ACLK       /4     Continuous  Clear TAR
  TA0CTL = TASSEL_1 | ID_2 | MC_2      | TACLR;

  _low_power_mode_3(); // We only need ACLK

  return 0;
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR()
{
  P1OUT ^= redLED;
  TA0CCR0 += 819; // 0.1s
  // clearing the flag
  TA0CCTL0 &= ~CCIFG;
}

#pragma vector = TIMER0_A1_VECTOR
__interrupt void T0A1_ISR()
{
  // channel 1
  if (TA0CCTL1 & CCIFG)
  {
    P9OUT ^= greenLED;
    TA0CCR1 += 4096; // 0.5s
    // clearing the flag
    TA0CCTL1 &= ~CCIFG;
  }

  // channel 2
  if (TA0CCTL2 & CCIFG)
  {
    TA0CCR2 += 32768; // 4s

    // clearing the flag
    TA0CCTL2 &= ~ CCIFG;

    // checking state
    if (state)
    {
      state = 0;

      TA0CCTL0 &= ~CCIE;
      TA0CCTL1 &= ~CCIE;
      TA0CCR0  += 32768;
      TA0CCR1  += 32768;

```

```c
      P1OUT &= ~redLED;
      P9OUT &= ~greenLED;
    }
    else
    {
      // state was not flashing and must be turned on to flashing
      state = 1;
      TA0CCTL0 |= CCIE;
      TA0CCTL1 |= CCIE;

      TA0CCTL0 &= ~CCIFG;
      TA0CCTL1 &= ~CCIFG;
    }
  }
}
```

10.3 Driving a PWM Signal on the Pin

The brightness level, thankfully, is changed via the changing of TA0CCR1 in all the ranges.

```c
#include "msp430fr6989.h"

#define PWM BIT0 // P1.0
#define PWM_VAL 33
#define UP 3500
#define DOWN 600

// 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 Initialize_ADC(void)
{
  // Configure the pins to analog functionality
  // X-axis: A10/P9.2, for A10 (P9DIR=x, P9SEL1=1, P9SEL0=1)
  P9SEL1 |= BIT2;
  P9SEL0 |= BIT2;
  // Turn on the ADC module
  ADC12CTL0 |= ADC12ON;
  // Turn off ENC (Enable Conversion) bit while modifying the
    configuration
  ADC12CTL0 &= ~ADC12ENC;
  //*************** ADC12CTL0 ***************
  // ADC12SHT0x sets SHT cycles for results 0-7, 24-31
  // ADC12MSC sets multiple analog inputs
  // Sets SHT of 16 cycles (found in doc. slau367o table 34.4)
  ADC12CTL0 |= ADC12SHT0_2;
  //*************** ADC12CTL1 ***************
  // ADC12SHS sets read trigger
  // ADC12SHP sets SAMPCON use
  /// ADC12DIV sets clock divider
  // ADC12SSEL sets clock base
  // ADC12CONSEQx sets conversion sequence mode
  ADC12CTL1 |= ADC12SHS_0;  // 0 = ADC12SC bit
  ADC12CTL1 |= ADC12SHP;    // 1 = SAMPCON sourced from clock
  ADC12CTL1 |= ADC12DIV_0;  // 0 = /1
  ADC12CTL1 |= ADC12SSEL_0; // 0 = MODOSC
  // ADC12CTL1 |= ADC12CONSEQ_1;
  // values in doc. slau367o table 34.5
  //*************** ADC12CTL2 ***************
  // ADC12RES sets bit resolution
  // ADC12DF sets data format
  ADC12CTL2 |= ADC12RES_2; // 2 = 12-bit
  ADC12CTL2 &= ~ADC12DF;   // 0 = unsigned binary
  //*************** ADC12MCTL0 ***************
  // ADC12VRSELx sets VR+ and VR- sources as well as buffering
  // ADC12INCHx sets analog input
  ADC12MCTL0 |= ADC12VRSEL_0; // 0 -> VR+ = AVCC and VR- = AVSS
  ADC12MCTL0 |= ADC12INCH_10; // 10 = A10 input
  //*************** ADC12MCTL1 ***************
  // set ENC bit at end of config
  ADC12CTL0 |= ADC12ENC;
}

int main(void)
{
  // Configure WDT & GPIO
  WDTCTL = WDTPW | WDTHOLD;
  PM5CTL0 &= ~LOCKLPM5;

  // Configure PWM
  P1DIR  |= PWM;
  P1SEL1 &= ~PWM;
  P1SEL0 |= PWM;

  config_ACLK_to_32KHz_crystal();

  TA0CCR0  = 33;  // 33 cycles for 1000Hz
  TA0CCTL0 = CCIE; // Enabling channel 0 interrupt

  TA0CCR1  = 15; // 50% brightness
  TA0CCTL1 = OUTMOD_7; // Reset/Set Output Mode

  //       ACLK       /1     Up     Clear TAR
  TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;

  // Initializing ADC
  Initialize_ADC();

  _low_power_mode_3(); // We only need ACLK

  unsigned int x;
  unsigned int y;

  for (;;)
  {
    ADC12CTL0 |= ADC12SC;

    while (ADC12CTL0 & ADC12BUSY)
    {
      // Wait till not busy
    }

    x = ADC12MEM0;
    y = ADC12MEM1;

    if (y > UP)
    {
      // Brightness is maximum
      TA0CCR1 = 32;
    }
    if (y < DOWN)
    {
      // Brightness is off
      TA0CCR1 = 0;
    }

    if (x > UP)
    {
      if (TA0CCR1 > 0)
      {
        TA0CCR1 -= 1;
      }
    }

    if (x < DOWN)
    {
      if (TA0CCR1 < 32)
      {
        TA0CCR1 += 1;
      }
    }

    __delay_cycles(16000); // 1ms delay
  }
  return 0;
}
```

10.4 Timer Input Capture

The button push lasts [**answer here**]

```
#include "msp430fr6989.h"
#include <stdint.h>

#define redLED BIT0
#define greenLED BIT7
#define S1 BIT1
#define S2 BIT2

// We love UART
#define FLAGS UCA1IFG
#define RXFLAG UCRXIFG
#define TXFLAG UCTXIFG
#define TXBUFFER UCA1TXBUF

// 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 initialize_uart(void)
{
  // Configuring the pins to use backchannel uart (SAME)
  P3SEL1 &= ~(transmit | recieve);
  P3SEL0 |= (transmit | recieve);

  // Setting the clock to ACLK (SEL_1 and not SEL_2 [SMCLK])
  UCA1CTLW0 |= UCSSEL_1;

  // Setting the dividers and enabling oversampling
  UCA1BRW = 6; // SAME DIVIDER
  // setting the modulators and such
  UCA1MCTLW = UCBRS1 | UCBRS2 | UCBRS3 | UCBRS5 | UCBRS6 | UCBRS7;

  // Exiting the reset state
  UCA1CTLW0 &= ~UCSWRST;
  return;
}
```

```c

void uart_write_char(volatile unsigned char ch)
{
  while (!(FLAGS & TXFLAG))
  {
    // Wait for transmission that is ongoing to complete
  }

  TXBUFFER = ch;
  return;
}

void uart_write_string(char *string)
{
  int i; // counter
  for (i = 0; i < strlen(string); i++)
  {
    uart_write_char(string[i]);
  }
  return;
}

void uart_write_uint16(uint16_t number)
{
  // Converting the number via snprintf
  char buffer[6]; // 5 characters is the max amount of characters
    for 65,536
  custom_itoa(number, buffer);
  uart_write_string(buffer);
  return;
}

int main(void)
{
  // Configure WDT & GPIO
  WDTCTL = WDTPW | WDTHOLD;
  PM5CTL0 &= ~LOCKLPM5;

  // Configure buttons
  P1DIR  &= ~S1; // input
  P1SEL1 &= ~S1; // 0
  P1SEL0 |= S1;  // 1
  P1REN  |= S1; // enable pull-up
  P1OUT  |= S1; // selecting pull-up

  initialize_uart();
  config_ACLK_to_32KHz_crystal();

  /*
   * CM_3:   Capture on both edges
   * CCIS_0: Capture input A (P1.1)
   * CAP:    Enable Capture Mode
   * CCIE:   Enable Interrupts
   * SCS:    Synchronous Capture (sync w/ clock)
  */
  TA0CCTL2 = CM_3 | CCIS_0 | CAP | CCIE | SCS;

  //        ACLK        /1      Continuous  Clear TAR
  TA0CTL = TASSEL_1 | ID_0 | MC_2       | TACLR;

  __bis_SR_register(LPM3_bits | GIE);

  return 0;
}

#pragma vector = TIMER0_A1_VECTOR
__interrupt void T0A1_ISR()
{
  if (TA0CCTL2 & CCIFG)
  {
    uint16_t time = TA0CCR2;

    uart_print_string("Time: ");
    uart_print_uint16(time);
    uart_printf_string("\r\n");

    // delay of 20ms
    __delay_cycles(20000);

    TA0CCTL2 &= ~CCIFG;
  }
}
```

Student Q&A

1

Given: *Copy the description of P9.7 from the pinout diagram and determine whether this pin supports timer-based output.*
From the description of P9.7 is has the following functionality ESICI3/A15/C15. In that, none of these support CCR, therefore the pin does not support timer-based output.

2

Given: *In the code with three channels, why was it necessary to divide ACLK?*
It is necessary to divide ACLK due to the fact that we want a delay of 4 seconds. ACLK specifically has a frequency of 65535, of which in up mode (and/or continuous mode) will have a maximum period of two seconds between each interrupt. Therefore, if we divide it, we can extend the interrupts to be at most every four seconds.

3

Given: *In the first part, we configured two periodic interrupts using two channels of the timer. Is this approach scalable? For example, using a Timer A module with five channels, can we configure five periodic interrupts? Explain and mention in what mode the timer would run.*
This approach is scalable up to five channels, since Timer_A has support up to five channels. The other three modules of Timer only support two, three, and three channels respectively. If you wanted to have these five channels, the timer would then have to be configured with a mode that allows for the capture and compare registers to be utilized (eg: up mode). Continuous mode would, specifically, not allow for the unique interrupts to be raised.

4

Given: *As an example, Channel 1's interrupt occurs every 40K cycles. The first interrupt is scheduled for when TAR=40K cycles. Explain how the next interrupt is scheduled? Explain the overflow mechanism and show why it results in a correct value.*
With interrupts occuring every 40 thousand cycles, future interrupts are scheduled by adding the interrupt period to the TA0R value of the previous interrupt. In this case, it would be 40 thousand + 40 thousand, resulting in 80 thousand. This 80 thousand is greater than the maximum therefore making so that, when it overflows, you get 0x3380 in hex or 14464 in decimal, of which is now the new expected TA0R value for the next interrupt raise.

Conclusion

This laboratory successfully demonstrated the advanced, hardware-centric features of the Timer_A module, moving beyond simple software delays. We proved that multiple capture/compare channels can operate concurrently in continuous mode to manage independent, asynchronous timing intervals. Furthermore, this lab highlighted the efficiency of hardware offloading by successfully generating a stable, CPU-independent 1000Hz PWM signal to control LED brightness and by utilizing Input Capture mode to timestamp external button events with high precision. These exercises confirm that leveraging the timer's dedicated hardware for output modulation and input timing is essential for building efficient, responsive, and low-power embedded systems.