

# Lab 10 Report

EEL4742C - 00446

Yousef Awad

November 2025

## Contents

<b>Introduction</b>	<b>2</b>
<b>10.1 Timer's Multiple Channels</b>	<b>2</b>
<b>10.2 Using Three Channels</b>	<b>4</b>
<b>10.3 Driving a PWM Signal on the Pin</b>	<b>7</b>
<b>10.4 Timer Input Capture</b>	<b>10</b>
<b>Student Q&amp;A</b>	<b>13</b>
1 . . . . .	13
2 . . . . .	13
3 . . . . .	13
4 . . . . .	13
<b>Conclusion</b>	<b>14</b>

## 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:

$$\frac{f_{clk}}{\text{input divider}} * \text{desired delay in seconds}$$
$$\frac{32768}{4} * 0.1 \approx 819 \rightarrow TA0CCR0 = 819$$
$$\frac{32768}{4} * 0.5 \approx 4096 \rightarrow TA0CCR1 = 4096$$

```
1 #include "msp430fr6989.h"
2
3 #define redLED BIT0
4 #define greenLED BIT7
5 #define S1 BIT1
6 #define S2 BIT2
7
8 // Configures ACLK to 32 KHz crystal
9 void config_ACLK_to_32KHz_crystal()
10 {
11     // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz
12
13     // Reroute pins to LFXIN/LFXOUT functionality
14     PJSEL1 &= ~BIT4;
15     PJSEL0 |= BIT4;
16
17     // Wait until the oscillator fault flags remain cleared
18     CSCTL0 = CSKEY; // Unlock CS registers
19     do
20     {
21         CSCTL5 &= ~LFXTOFFG; // Local fault flag
22         SFRIFG1 &= ~OFIFG; // Global fault flag
23     }
24     while((CSCTL5 & LFXTOFFG) != 0);
25
26     CSCTL0_H = 0; // Lock CS registers
27     return;
28 }
```

```

30 int main(void)
31 {
32     // Configure WDT & GPIO
33     WDTCTL = WDTPW | WDTHOLD;
34     PM5CTL0 &= ~LOCKLPM5;
35
36     // Configure LEDs
37     P1DIR |= redLED;
38     P9DIR |= greenLED;
39     P1OUT &= ~redLED;
40     P9OUT &= ~greenLED;
41
42     // Configure buttons
43     P1DIR &= ~(S1 | S2);
44     P1REN |= (S1 | S2);
45     P1OUT |= (S1 | S2);
46     P1IFG &= ~(S1 | S2); // Flags are used for latched polling
47
48     config_ACLK_to_32KHz_crystal();
49
50     TAOCCR0 = 819; // 0.1s for red
51     TAOCTLO |= CCIE; // Enabling channel 1 interrupt
52
53     //      ACLK      /4      Continuous Clear TAR
54     TAOCTL = TASSEL_1 | ID_2 | MC_2          | TACLR;
55
56     _low_power_mode_3(); // We only need ACLK
57
58     return 0;
59 }
60
61 #pragma vector = TIMERO_A0_VECTOR
62 __interrupt void TOA0_ISR()
63 {
64     P1OUT ^= redLED;
65     TAOCCR0 += 819;
66     // clearing the flag
67     TAOCTLO &= ~CCIFG;
68 }
69
70 #pragma vector = TIMERO_A1_VECTOR
71 __interrupt void TOA1_ISR()
72 {
73     if (TAOCTL1 & CCIFG)
74     {
75         P9OUT ^= greenLED;
76         TAOCCR1 += 4096; // 1/2 a second
77         // clearing the flag
78         TAOCTL1 &= ~CCIFG;
79     }
80 }

```

## 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).

```
1 #include "msp430fr6989.h"
2
3 #define redLED BIT0
4 #define greenLED BIT7
5 #define S1 BIT1
6 #define S2 BIT2
7
8 // tracks system state between not flashing (0) and flashing (1)
9 volatile unsigned int state = 1;
10
11 // Configures ACLK to 32 KHz crystal
12 void config_ACLK_to_32KHz_crystal()
13 {
14     // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz
15
16     // Reroute pins to LFXIN/LFXOUT functionality
17     PJSEL1 &= ~BIT4;
18     PJSELO |= BIT4;
19
20     // Wait until the oscillator fault flags remain cleared
21     CSCTL0 = CSKEY; // Unlock CS registers
22     do
23     {
24         CSCTL5 &= ~LFXTOFFG; // Local fault flag
25         SFRIFG1 &= ~OFIFG; // Global fault flag
26     }
27     while((CSCTL5 & LFXTOFFG) != 0);
28
29     CSCTL0_H = 0; // Lock CS registers
30     return;
31 }
32
33 int main(void)
34 {
35     // Configure WDT & GPIO
36     WDTCTL = WDTPW | WDTHOLD;
37     PM5CTL0 &= ~LOCKLPM5;
38
39     // Configure LEDs
40     P1DIR |= redLED;
41     P9DIR |= greenLED;
42     P1OUT &= ~redLED;
43     P9OUT &= ~greenLED;
44
45     // Configure buttons
46     P1DIR &= ~(S1 | S2);
47     P1REN |= (S1 | S2);
48     P1OUT |= (S1 | S2);
49     P1IFG &= ~(S1 | S2); // Flags are used for latched polling
50
51     config_ACLK_to_32KHz_crystal();
```

```

52     TAOCCR0  = 819; // 0.1s
53     TAOCTL0 = CCIE; // Enabling channel 0 interrupt
54
55     TAOCCR1  = 4096; // 0.5s
56     TAOCTL1 = CCIE; // enabling interrupt on channel 1
57
58     TAOCCR2  = 32768; // 1s
59     TAOCTL2 = CCIE; // enabling interrupt on channel 2
60
61     //      ACLK      /4      Continuous Clear TAR
62     TAOCTL = TASSEL_1 | ID_2 | MC_2           | TACLR;
63
64     _low_power_mode_3(); // We only need ACLK
65
66     return 0;
67 }
68
69 #pragma vector = TIMERO_A0_VECTOR
70 __interrupt void TOAO_ISR()
71 {
72     P1OUT ^= redLED;
73     TAOCCR0 += 819; // 0.1s
74     // clearing the flag
75     TAOCTL0 &= ~CCIFG;
76 }
77
78 #pragma vector = TIMERO_A1_VECTOR
79 __interrupt void TOA1_ISR()
80 {
81     // channel 1
82     if (TAOCTL1 & CCIFG)
83     {
84         P9OUT ^= greenLED;
85         TAOCCR1 += 4096; // 0.5s
86         // clearing the flag
87         TAOCTL1 &= ~CCIFG;
88     }
89
90     // channel 2
91     if (TAOCTL2 & CCIFG)
92     {
93         TAOCCR2 += 32768; // 4s
94
95         // clearing the flag
96         TAOCTL2 &= ~CCIFG;
97
98         // checking state
99         if (state)
100         {
101             state = 0;
102
103             TAOCTL0 &= ~CCIE;
104             TAOCTL1 &= ~CCIE;
105             TAOCCR0 += 32768;
106             TAOCCR1 += 32768;
107
108

```

```
109     P1OUT &= ~redLED;
110     P9OUT &= ~greenLED;
111 }
112 else
113 {
114     // state was not flashing and must be turned on to flashing
115     state = 1;
116     TAOCCCTL0 |= CCIE;
117     TAOCCCTL1 |= CCIE;
118
119     TAOCCCTL0 &= ~CCIFG;
120     TAOCCCTL1 &= ~CCIFG;
121 }
122 }
123 }
```

### 10.3 Driving a PWM Signal on the Pin

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

```
1 #include "msp430fr6989.h"
2
3 #define PWM BIT0 // P1.0
4 #define PWM_VAL 33
5 #define UP 3500
6 #define DOWN 600
7
8 // Configures ACLK to 32 KHz crystal
9 void config_ACLK_to_32KHz_crystal()
10 {
11     // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz
12
13     // Reroute pins to LFXIN/LFXOUT functionality
14     PJSEL1 &= ~BIT4;
15     PJSELO |= BIT4;
16
17     // Wait until the oscillator fault flags remain cleared
18     CSCTL0 = CSKEY; // Unlock CS registers
19     do
20     {
21         CSCTL5 &= ~LFXTOFFG; // Local fault flag
22         SFRIFG1 &= ~OFIFG; // Global fault flag
23     }
24     while((CSCTL5 & LFXTOFFG) != 0);
25
26     CSCTL0_H = 0; // Lock CS registers
27     return;
28 }
29
30 void Initialize_ADC(void)
31 {
32     // Configure the pins to analog functionality
33     // X-axis: A10/P9.2, for A10 (P9DIR=x, P9SEL1=1, P9SELO=1)
34     P9SEL1 |= BIT2;
35     P9SELO |= BIT2;
36     // Turn on the ADC module
37     ADC12CTL0 |= ADC12ON;
38     // Turn off ENC (Enable Conversion) bit while modifying the
39     // configuration
40     ADC12CTL0 &= ~ADC12ENC;
41     //***** ADC12CTL0 ***** *****
42     // ADC12SHT0x sets SHT cycles for results 0-7, 24-31
43     // ADC12MSC sets multiple analog inputs
44     // Sets SHT of 16 cycles (found in doc. slau367o table 34.4)
45     ADC12CTL0 |= ADC12SHTO_2;
46     //***** ADC12CTL1 *****
47     // ADC12SHS sets read trigger
48     // ADC12SHP sets SAMPCON use
49     // ADC12DIV sets clock divider
50     // ADC12SSEL sets clock base
51     // ADC12CONSEQx sets conversion sequence mode
52     ADC12CTL1 |= ADC12SHS_0; // 0 = ADC12SC bit
```

```

52     ADC12CTL1 |= ADC12SHP;      // 1 = SAMPCON sourced from clock
53     ADC12CTL1 |= ADC12DIV_0;    // 0 = /1
54     ADC12CTL1 |= ADC12SSEL_0;   // 0 = MODOSC
55     // ADC12CTL1 |= ADC12CONSEQ_1;
56     // values in doc. slau367o table 34.5
57     //***** ADC12CTL2 *****
58     // ADC12RES sets bit resolution
59     // ADC12DF sets data format
60     ADC12CTL2 |= ADC12RES_2;   // 2 = 12-bit
61     ADC12CTL2 &= ~ADC12DF;     // 0 = unsigned binary
62     //***** ADC12MCTL0 *****
63     // ADC12VRSELx sets VR+ and VR- sources as well as buffering
64     // ADC12INCHx sets analog input
65     ADC12MCTL0 |= ADC12VRSEL_0; // 0 -> VR+ = AVCC and VR- = AVSS
66     ADC12MCTL0 |= ADC12INCH_10; // 10 = A10 input
67     //***** ADC12MCTL1 *****
68     // set ENC bit at end of config
69     ADC12CTL0 |= ADC12ENC;
70 }
71
72 int main(void)
73 {
74     // Configure WDT & GPIO
75     WDTCTL = WDTPW | WDTHOLD;
76     PM5CTL0 &= ~LOCKLPM5;
77
78     // Configure PWM
79     P1DIR |= PWM;
80     P1SEL1 &= ~PWM;
81     P1SEL0 |= PWM;
82
83     config_ACLK_to_32KHz_crystal();
84
85     TAOCCR0 = 33; // 33 cycles for 1000Hz
86     TAOCCTL0 = CCIE; // Enabling channel 0 interrupt
87
88     TAOCCR1 = 15; // 50% brightness
89     TAOCCTL1 = OUTMOD_7; // Reset/Set Output Mode
90
91     //      ACLK      /1      Up      Clear TAR
92     TAOCTL = TASSEL_1 | ID_0 | MC_1 | TACLR;
93
94     // Initializing ADC
95     Initialize_ADC();
96
97     _low_power_mode_3(); // We only need ACLK
98
99     unsigned int x;
100    unsigned int y;
101
102    for (;;)
103    {
104        ADC12CTL0 |= ADC12SC;
105
106        while (ADC12CTL0 & ADC12BUSY)
107        {
108            // Wait till not busy

```

```

109
110
111     x = ADC12MEM0;
112     y = ADC12MEM1;
113
114     if (y > UP)
115     {
116         // Brightness is maximum
117         TAOCCR1 = 32;
118     }
119     if (y < DOWN)
120     {
121         // Brightness is off
122         TAOCCR1 = 0;
123     }
124
125     if (x > UP)
126     {
127         if (TAOCCR1 > 0)
128         {
129             TAOCCR1 -= 1;
130         }
131     }
132
133     if (x < DOWN)
134     {
135         if (TAOCCR1 < 32)
136         {
137             TAOCCR1 += 1;
138         }
139     }
140
141     __delay_cycles(16000); // 1ms delay
142 }
143 return 0;
144 }
```

## 10.4 Timer Input Capture

The button push lasts [answer here]

```
1 #include "msp430fr6989.h"
2 #include <stdint.h>
3
4 #define redLED BIT0
5 #define greenLED BIT7
6 #define S1 BIT1
7 #define S2 BIT2
8
9 // We love UART
10 #define FLAGS UCA1IFG
11 #define RXFLAG UCRXIFG
12 #define TXFLAG UCTXIFG
13 #define TXBUFFER UCA1TXBUF
14
15 // Configures ACLK to 32 KHz crystal
16 void config_ACLK_to_32KHz_crystal()
17 {
18     // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz
19
20     // Reroute pins to LFXIN/LFXOUT functionality
21     PJSEL1 &= ~BIT4;
22     PJSEL0 |= BIT4;
23
24     // Wait until the oscillator fault flags remain cleared
25     CSCTL0 = CSKEY; // Unlock CS registers
26     do
27     {
28         CSCTL5 &= ~LFXTOFFG; // Local fault flag
29         SFRIFG1 &= ~OFIFG; // Global fault flag
30     }
31     while((CSCTL5 & LFXTOFFG) != 0);
32
33     CSCTL0_H = 0; // Lock CS registers
34     return;
35 }
36
37 void initialize_uart(void)
38 {
39     // Configuring the pins to use backchannel uart (SAME)
40     P3SEL1 &= ~(transmit | recieve);
41     P3SEL0 |= (transmit | recieve);
42
43     // Setting the clock to ACLK (SEL_1 and not SEL_2 [SMCLK])
44     UCA1CTLW0 |= UCSSEL_1;
45
46     // Setting the dividers and enabling oversampling
47     UCA1BRW = 6; // SAME DIVIDER
48     // setting the modulators and such
49     UCA1MCTLW = UCBRS1 | UCBRS2 | UCBRS3 | UCBRS5 | UCBRS6 | UCBRS7;
50
51     // Exiting the reset state
52     UCA1CTLW0 &= ~UCSWRST;
53     return;
54 }
```

```

55
56 void uart_write_char(volatile unsigned char ch)
57 {
58     while (!(FLAGS & TXFLAG))
59     {
60         // Wait for transmission that is ongoing to complete
61     }
62
63     TXBUFFER = ch;
64     return;
65 }
66
67 void uart_write_string(char *string)
68 {
69     int i; // counter
70     for (i = 0; i < strlen(string); i++)
71     {
72         uart_write_char(string[i]);
73     }
74     return;
75 }
76
77 void uart_write_uint16(uint16_t number)
78 {
79     // Converting the number via sprintf
80     char buffer[6]; // 5 characters is the max amount of characters
81     for 65,536
82     custom_itoa(number, buffer);
83     uart_write_string(buffer);
84     return;
85 }
86
87 int main(void)
88 {
89     // Configure WDT & GPIO
90     WDTCTL = WDTPW | WDTHOLD;
91     PM5CTL0 &= ~LOCKLPM5;
92
93     // Configure buttons
94     P1DIR &= ~S1; // input
95     P1SEL1 &= ~S1; // 0
96     P1SEL0 |= S1; // 1
97     P1REN |= S1; // enable pull-up
98     P1OUT |= S1; // selecting pull-up
99
100    initialize_uart();
101    config_ACLK_to_32KHz_crystal();
102
103    /*
104     * CM_3: Capture on both edges
105     * CCIS_0: Capture input A (P1.1)
106     * CAP: Enable Capture Mode
107     * CCIE: Enable Interrupts
108     * SCS: Synchronous Capture (sync w/ clock)
109     */
110    TAOCCTL2 = CM_3 | CCIS_0 | CAP | CCIE | SCS;

```

```

111 //      ACLK      /1      Continuous  Clear TAR
112 TAOCTL = TASSEL_1 | ID_0 | MC_2          | TACLR;
113
114 __bis_SR_register(LPM3_bits | GIE);
115
116 return 0;
117 }
118
119 #pragma vector = TIMERO_A1_VECTOR
120 __interrupt void TOA1_ISR()
121 {
122     if (TAOCCTL2 & CCIFG)
123     {
124         uint16_t time = TA0CCR2;
125
126         uart_print_string("Time: ");
127         uart_print_uint16(time);
128         uart_printf_string("\r\n");
129
130         // delay of 20ms
131         __delay_cycles(20000);
132
133         TAOCCTL2 &= ~CCIFG;
134     }
135 }
```

## 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 it 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 occurring 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.