CPE 323
MODULE 09
MSP430 Clocks, Time & Timers
(Watchdog Timer, Timer_A, Timer_B)

Aleksandar Milenković

The LaCASA Laboratory, ECE Department, The University of Alabama in Huntsville

Email: milenka@uah.edu

Web: http://www.ece.uah.edu/~milenka

Overview

This module discusses time in embedded systems, starting from clock sources and clock subsystems through specialized peripherals such as watchdog timer, TimerA, and TimerB. You will learn about hardware and software aspects of these components and how to utilize them in embedded programs for time keeping, time stamping, and generating signals of desired shape.

Objectives

- *Learners will understand hardware and software aspects of clock sources and timer peripherals*
- *Learners will be able to configure and interact with clock sources*
- *Learners will be able to configure and interact with timer peripherals (Watchdog Timer, Timer_A, Timer_B)*

Contents

1 Introduction

In this section we will discuss time and timers in embedded systems. One of the key characteristics of embedded computer systems is that they maintain some notion of time. E.g., your microwave oven displays the current wall-clock time. You can use menu options to set the current time or to set the cook time. Next, assume you are tasked to design a smart road-side sensor that keeps track of traffic. This sensor should keep track of individual vehicles - for every passing vehicle it records and reports the moment the vehicle has been detected and perhaps its size. By collecting information from a network of widely deployed road-side sensors and analyzing them, transportation authorities will be in position to develop better plans for managing traffic or better plans for investing in infrastructure. Yet another example of an embedded system with a need for time keeping is when we want to trigger certain events to occur at specific time. E.g., we want a clock alarm to sound off at 4:30 AM (just kidding), or we want a robotic hand to rotate for 45 degrees. In all these examples, time and timers play a critical role.

Embedded computer systems usually have at least one timer peripheral device. You can think of timers as simple digital counters. In an active mode they increment or decrement their value at a specified clock frequency. Before using a timer in our application, we need to initialize it by setting its control registers. During initialization we need to specify timer's operating mode (whether they increment or decrement their value on each clock, pause, etc.), the clock frequency, and whether it will raise an interrupt request once the counter reaches zero or a value predetermined by software. Timers may have comparison logic to compare the value of the running counter to a specific value held in a separate register that is also set by software. When the values match, the timer may take certain actions, e.g., roll back to zero, toggle its output signal, request an interrupt, to name just a few possibilities. Timers are instrumental in generating *pulse width modulated* (PWM) signals used to control the speed of motors. Next, timers can be configured to capture the current value of the running counter when a certain event occurs (e.g., the input signal changes from a logic zero to a logic one). Timers can also be used to trigger execution of the corresponding interrupt service routines. The MSP430 family supports several types of timer peripheral devices, namely the Watchdog Timer, Basic Timer 1, Real Time Clock, Timer A, and Timer B.

2 Clocks, Counters

To measure time we rely on a system clock. The resolution of time measurements is determined by the clock frequency of the system clock. The MSP430 family includes a range of clock subsystems that vary in complexity, number and type of clock sources, and configurability. The clock subsystems often include on-chip digitally controlled oscillators that can generate higher clock frequencies while relying on relatively slow and inexpensive external clock sources. Common to all clock subsystems is that they provide three clocks to the rest of the MSP430:

- MCLK – main clock used by the processor core and select peripherals;

- SMCLK – sub-main clock used by a variety of peripherals; and
- ACLK – auxiliary clock used by a variety of peripherals.

Figure 1 illustrates a clock signal named MCLK that has period of 1 μs. It is 0.5 μs at logic '1' and 0.5 μs is at logic '0'. The duty cycle of a periodic signal is defined as the time the signal is at logic '1' divided by the signal period (0.5 μs/1 μs). Thus, the duty cycle of the MCLK clock in this example is 50%. Upon powering up the MSP430 we typically have MCLK and SMCLK set to 2^{20} Hz or 1,048,576 Hz. The clock cycle time is thus slightly below 1 μs. The ACLK is typically set to 2^{15} Hz or 32,768 Hz. An MSP430 clock subsystem is very sophisticated allowing developers to increase/decrease the clock frequency on-the-fly by setting certain bits in its control registers. This means that the frequency of individual clocks can be changed smoothly without stopping the system.

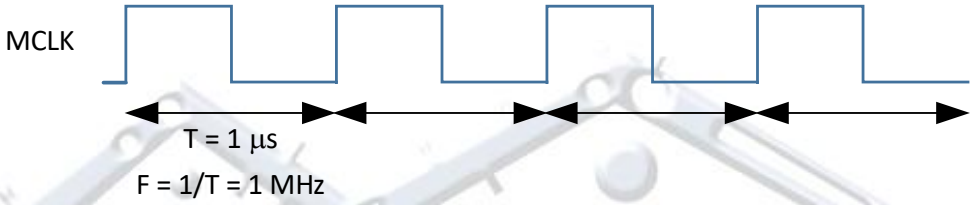


Figure 1. An example of a clock signal.
The clock period is T_{MCLK} = 1 μs and the clock frequency $F_{MCLK} = 1/T_{MCLK}$ = 1 MHz.

`Things to remember 2-1. Clocks and MSP430 Clocks.`

A clock signal is a square wave (oscillates between a low and a high state) that is used as a metronome to coordinate actions of digital circuits. An MSP430 Clock Subsystem provides three clock signals to other components: MCLK – main clock, SMCLK – submain clock, and ACLK – auxiliary clock. Clocks in MSP430 are configurable and can be changed on-the-fly without stopping the processor.

Figure 2 shows a 16-bit binary counter that counts up on every rising edge of the input clock signal, CLK. When the counter reaches its maximum 0xFFFF (65,535) it rolls back to 0x0000 and repeats the counting. The timer thus overflows every 65,536 (2^{16}) clock cycles. The time between any two count values is the elapsed time. The elapsed time in our example from Figure 2 is $\Delta t = N*T_{CLK}$, where N = 65,533.

When illustrating the counting up on every rising edge of the clock, we often use a ramp-like signal as shown on the bottom of Figure 2, rather than specifying the counter value in every clock cycle.

Please note that we can often configure the counter to count up to a certain constant value set by the software developer. For example, we can have 16-bit counter that counts from 0 up to 49,999 and then back to 0. This way the counter overflows every 50,000 clock cycles. Assuming

T_{CLK} = 1 μs, the counter overflows every 50 ms (50,000*1 μs = 50 ms). If we configure the counter in such a way, we can use it to create a periodic signal with the period T_S=50 ms (F_S = 20 Hz).

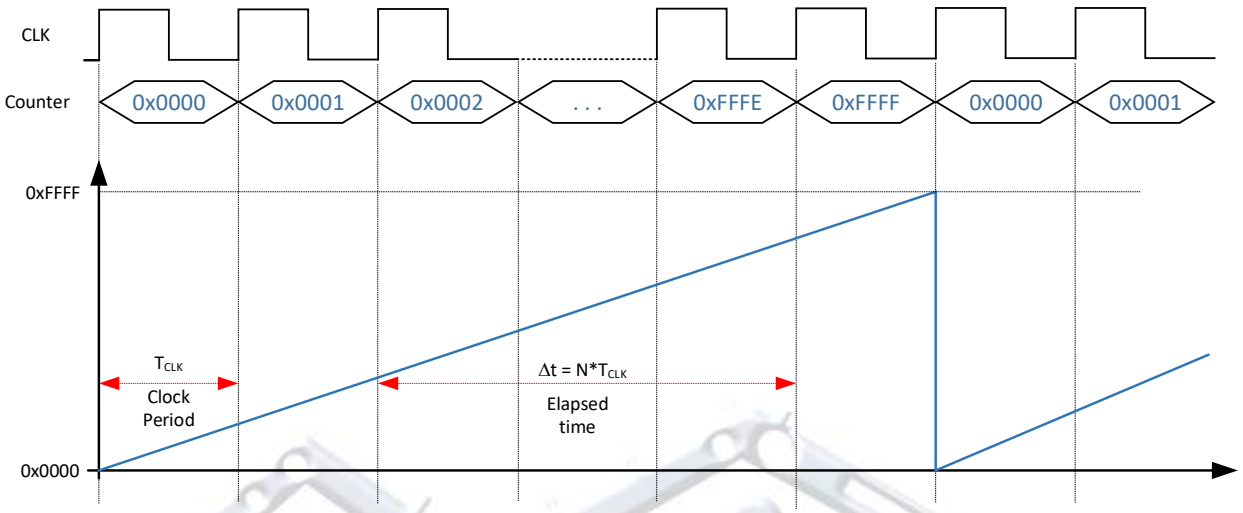


Figure 2. An example of a 16-bit counter counting up from 0x0000 to 0xFFFF. The counter counts up on every rising edge of the input clock, CLK, and rolls back to 0 after 0xFFFF - it overflows every 65,536 clock cycles. The time elapsed between any two count values is N*T_{CLK} (N=65,534-1=65,533 in this example).

`Things to remember 2-2. Counters.`

A counter is a component that counts the number of times a particular event has occurred. For example, an UP counter can increment its value on every rising edge of the clock signal, thus counting the number of clock cycles. Counters can be configured as modulus counters. Modulus counters count through a particular number of states that can be different from the maximum number of states determined by their size.

One typical use of timers is to precisely timestamp external events in the so-called *capture mode* illustrated in Figure 3. Events are translated into digital input changes (rising edge, falling edge, or both). For example, let us assume we want to precisely timestamp when a car enters a parking lot. A car entering the parking lot triggers a change of an input signal from a logic 0 to a logic 1. This input signal is brought to our timer device. Let us further assume that our timer's counter is configured as described above – it repeatedly counts up from 0 to 49,999. The internal logic detects a rising edge of the input signal and then triggers loading the value from the running counter into a dedicated register called Capture Register in Figure 3. The value captured in this register is a precise timestamp of the event and we can read the value of this register and further process it in software. Thus, a timer working in the capture mode is used to

precisely timestamp external or internal events. The resolution of this timestamp is determined by the resolution of the timer clock (1 μs in this example). By doing this in hardware we achieve the maximum accuracy and we do not have to use processor for monitoring external events. In addition, precise time measurements purely in software are a challenging proposition for various reasons (e.g., interrupts).

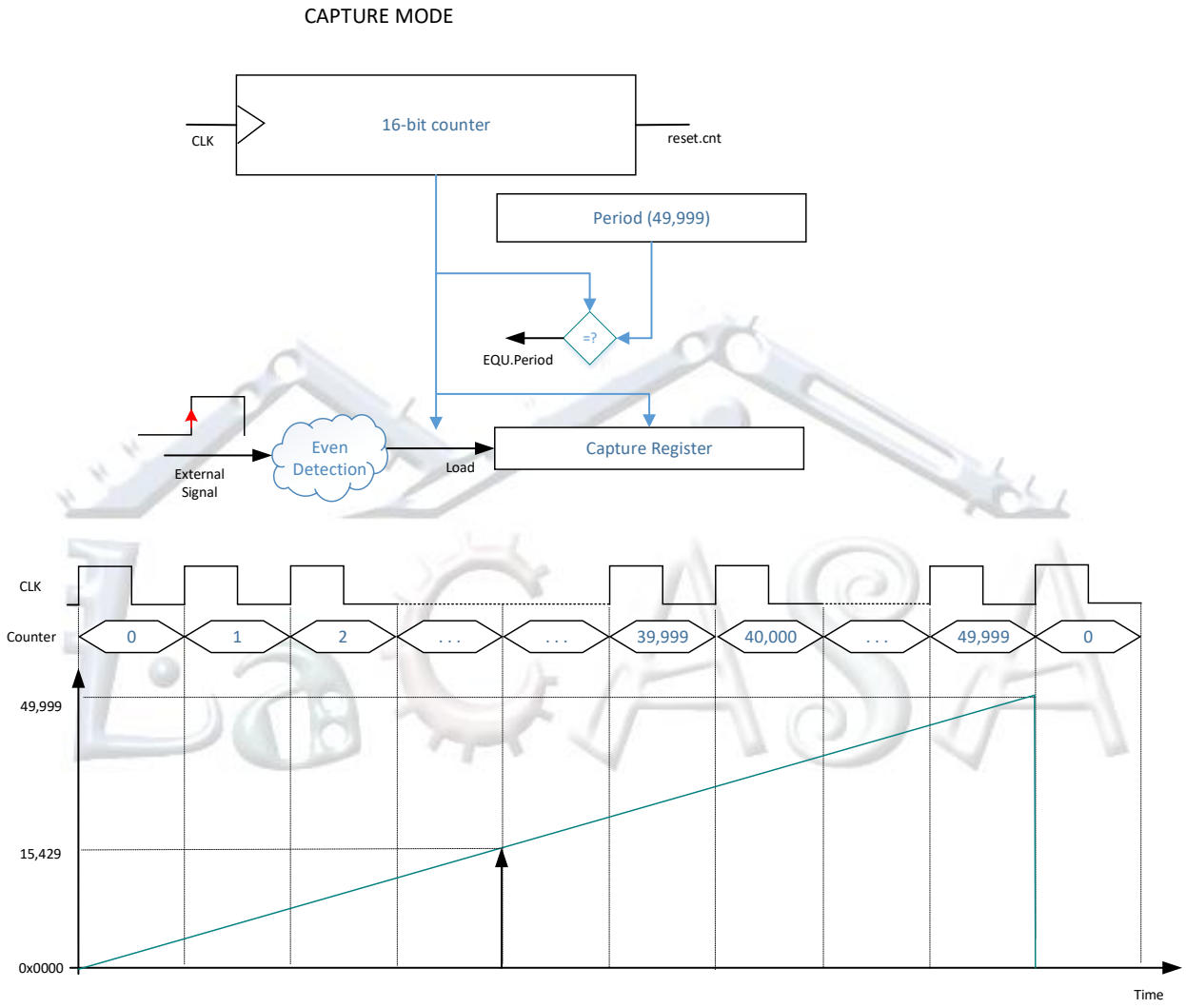


Figure 3. Timer capture mode. Description: Counter is set to count from 0 to 49,999 (50,000 clock cycles). The module is configured to capture the value of the running counter when a rising edge of an external signal is detected. The event detection logic triggers loading the current value of the running counter in the Capture Register. This way we can precisely timestamp events with resolution of a single clock cycle. In this example we know that an event occurred at clock cycle 15,429 – 15,429*1 μs or 15.429 ms from the beginning of the current 50 ms period.

Things to remember 2-3. Timer Capture Mode.

In the capture mode a value from the running counter is captured (stored) in a separate capture register when a particular event of interest occurs (e.g., an external input goes from a low state to a high state). This way the moment when this event occurs can be precisely timed (timestamped).

The timer modules operating in the so-called compare mode can be used to shape output signals up by controlling their period and duty cycle as illustrated in Figure 4. For example, let us assume we want to generate a periodic output signal with a period of 50 ms and the duty cycle of 80% (the signal is 40 ms on logic '1' and 10 ms on logic '0'). The period of 50 ms corresponds to 50,000 clock cycles. The 80% of that is 40,000 clock cycles. Consequently we can have another register called Duty Cycle initialized to 39,999 (40,000-1). The output signal OUT is set to a logic '1'. When the running counter reaches 39,999 the output of the comparator indicates that it matches the value stored in the Duty Cycle register. This triggers the output signal to go to a logic '0'. When the running counter reaches 49,999 it rolls back to 0 and the output signal OUT is set a logic '1'. The output signal is held at logic '1' for 40,000 clock cycles and at logic '0' for 10,000 clock cycles, thus producing the desired signal. Timers in the compare mode are used to generate pulse width modulated signal (PWM) that are widely used in robotic applications.

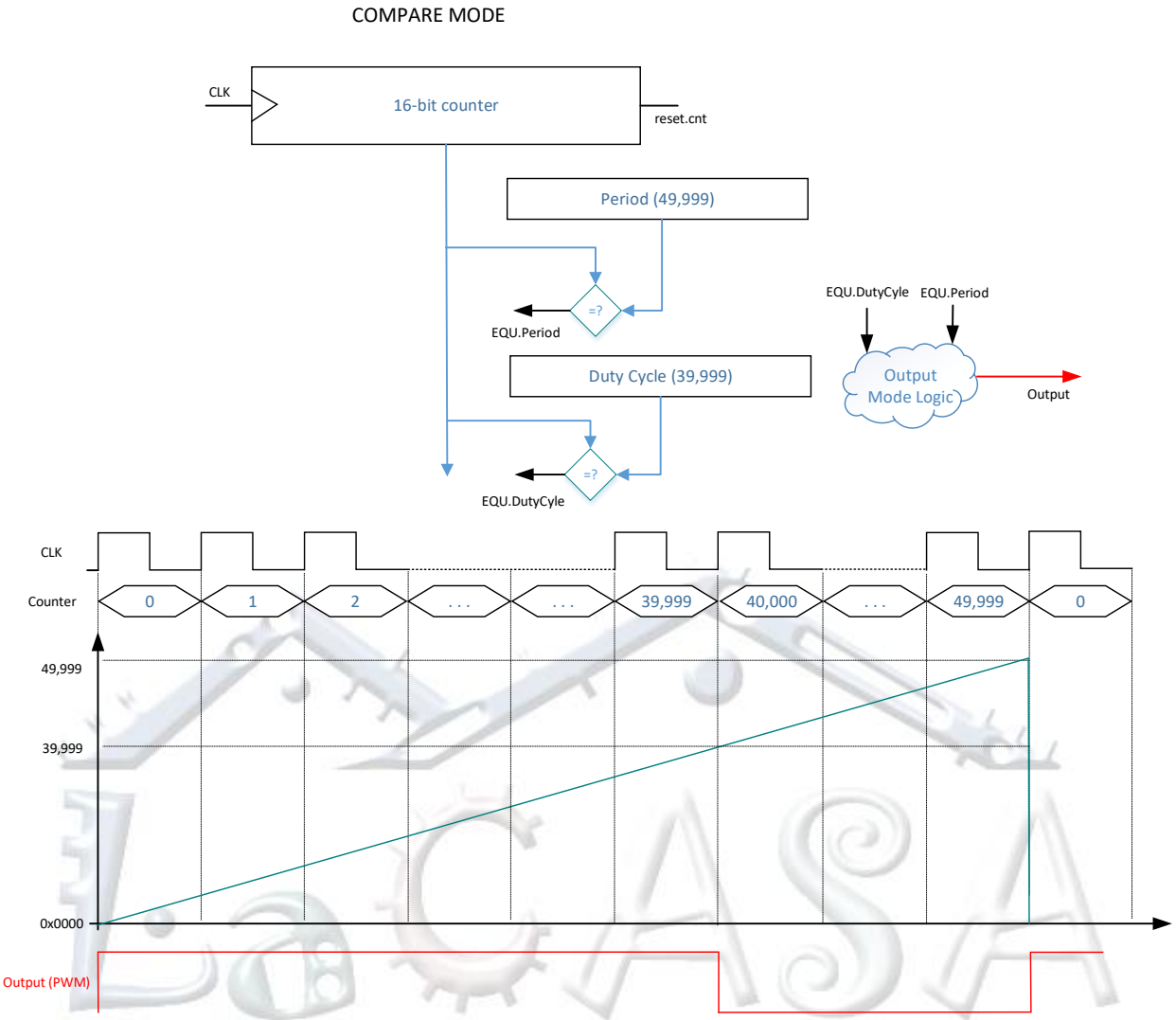


Figure 4. Timer in a compare mode configured to generate a signal with period of 50 ms and duty cycle of 80%. Counter is set to count from 0 to 49,999 (50,000 clock cycles) - the register Limit is initialized to 49,999. The register DutyCycle is set to 39,999. The control logic is configured to generate an output signal OUT that has period of 50 ms and the duty cycle of 80% (40 ms at logic 1 and 10 ms at logic 0).

`Things to remember 2-4. Timer Compare Mode.`

In the compare mode, a value from the running timer is compared to a predefined value stored in a compare register. When the two values match, a change on an output signal is triggered (set, reset, or toggle). Timers working in this mode are used to create Pulse-Width-Modulated output signals.

Figure 5 shows a block diagram of the MSP430F5529 device. The Unified Clock System provides three clock signals. The SYS module includes the Watchdog timer. In addition, it includes three Timer A devices, TA0, TA1, and TA2, one Timer B device, TB0, and a real-time clock, RTC_A. In the following sections we will discuss the Watchdog timer and Timer A.

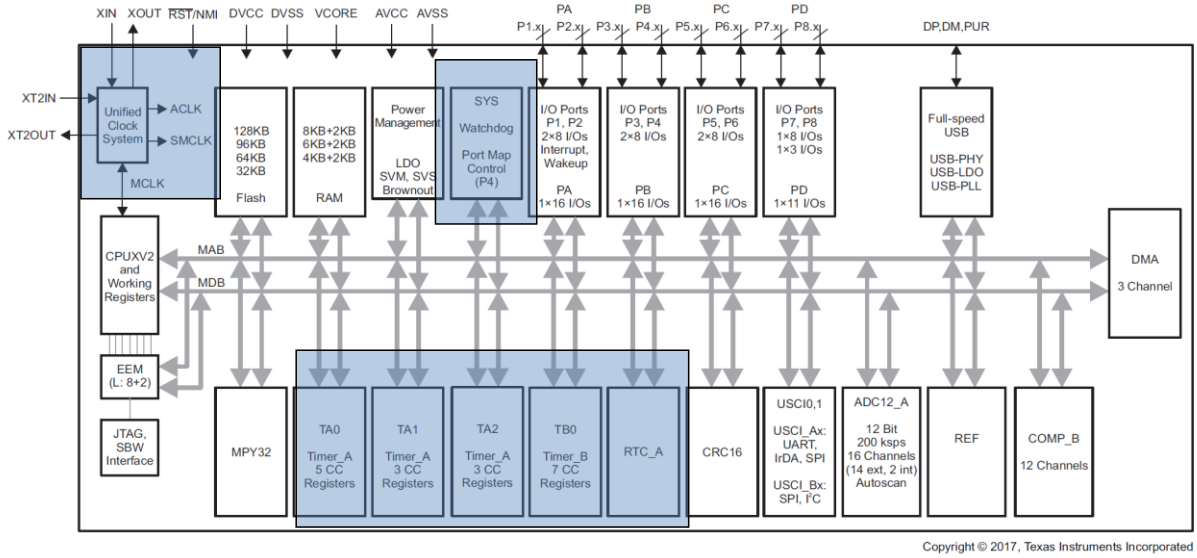


Figure 5. Block diagram of MSP430F5529 with clock and timer modules.

3 Watchdog Timer

The primary function of the watchdog-timer (WDT) is to perform a controlled-system restart after a software problem occurs. This mode of operation is called *watchdog mode*. If we use the watchdog timer in the watchdog mode, we organize our software in such a way that we always periodically reset the counter ("*petting the dog*"), so it never gets to the point to reach a predefined counter value. If the running program fails to reach the point where the counter reset is initiated due to unpredicted behavior or system fault, the timer is going to expire and hence cause a system reset (PUC – power up clear, "*dog is biting*"), thus signaling that something went wrong.

If the watchdog function is not needed in an application, the watchdog timer can work in the so-called *interval timer mode.* In this mode, when the timer reaches the predefined value it sets the WDTIFG flag without causing a system reset. If the corresponding WDTIE flag is enabled, this event will trigger a regular (non-resetting) interrupt request. Thus, the watchdog timer can generate periodic events.

Figure 6 illustrates a block diagram of the watchdog timer peripheral. It features a 16-bit control register, WDTCTL, and a 32-bit counter, WDTCNT. The watchdog timer counter (WDTCNT) is a 32-bit up-counter that is not directly accessible by software. The WDTCNT counter is controlled and time intervals are selected through the watchdog timer control register WDTCTL.

Things to remember 3-1. Watchdog Timer.

The MSP430 Watchdog Timer can operate in 3 modes: HALT (turned-off), watchdog mode, and interval-time mode.

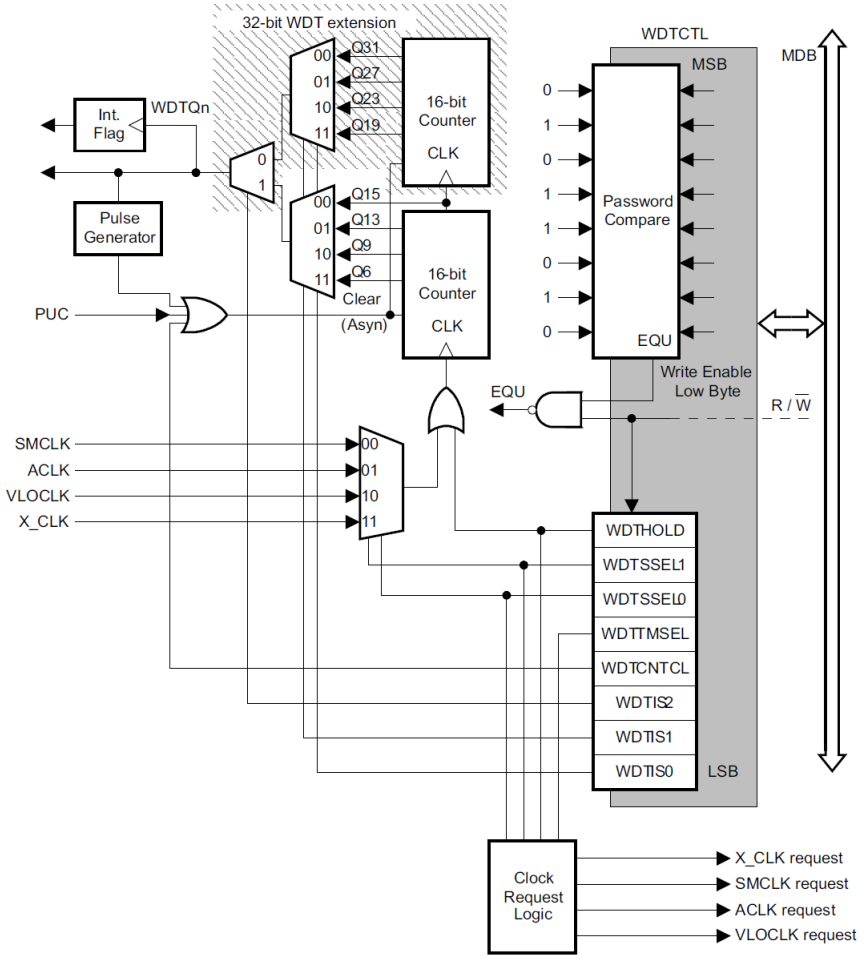


Figure 6. Block Diagram of the Watchdog Timer

Figure 7 shows the the format of the watchdog control register, WDTCTL. In includes control bits for stopping the watchdog timer (WDTHOLD), sourcing the WDCNT counter clock (WDTSSEL), selecting mode of operation (WDTTMSEL), clearing the WDTCNT counter (WDTCNTCL), and the interval selection (WDTIS). The WDTIFG bit resides in SFRIFG1, bit 0 and the corresponding WDTIE bit resides in SFRIE1, bit 0.

Setting the WDTTMSEL bit to a logic ‘1’ selects the interval timer mode. In the interval timer mode, the WDTIFG flag is set at the expiration of the selected time interval. A PUC is not generated in this mode at expiration of the selected timer interval and the WDTIFG enable bit WDTIE remains unchanged. If the WDTIE bit and the GIE bit are set, the set WDTIFG flag will request a WDT interrupt. The WDTIFG interrupt flag is automatically reset when its interrupt

request is accepted, or it needs to be reset by software if polling is used. The interrupt vector address associated with the interval timer mode is different from the one associated with the interrupt vector address used in the watchdog mode.

15	14	13	12	11	10	9	8
WDTPW							
7	6	5	4	3	2	1	0
WDTHOLD	WDTSSEL		WDTTMSEL	WDTCNTCL	WDTIS		
rw-0	rw-0	rw-0	rw-0	r0(w)	rw-1	rw-0	rw-0

Bit	Field	Type	Reset	Description
15-8	WDTPW	RW	69h	Watchdog timer password. Always read as 069h. Must be written as 5Ah; if any other value is written, a PUC is generated.
7	WDTHOLD	RW	0h	Watchdog timer hold. This bit stops the watchdog timer. Setting WDTHOLD = 1 when the WDT is not in use conserves power. 0b = Watchdog timer is not stopped. 1b = Watchdog timer is stopped.
6-5	WDTSSEL	RW	0h	Watchdog timer clock source select 00b = SMCLK 01b = ACLK 10b = VLOCLK 11b = X_CLK; VLOCLK in devices that do not support X_CLK
4	WDTTMSEL	RW	0h	Watchdog timer mode select 0b = Watchdog mode 1b = Interval timer mode
3	WDTCNTCL	RW	0h	Watchdog timer counter clear. Setting WDTCNTCL = 1 clears the count value to 0000h. WDTCNTCL is automatically reset. 0b = No action 1b = WDTCNT = 0000h
2-0	WDTIS	RW	4h	Watchdog timer interval select. These bits select the watchdog timer interval to set the WDTIFG flag and/or generate a PUC. 000b = Watchdog clock source /(2^{31}) (18h:12m:16s at 32.768 kHz) 001b = Watchdog clock source /(2^{27}) (01h:08m:16s at 32.768 kHz) 010b = Watchdog clock source /(2^{23}) (00h:04m:16s at 32.768 kHz) 011b = Watchdog clock source /(2^{19}) (00h:00m:16s at 32.768 kHz) 100b = Watchdog clock source /(2^{15}) (1 s at 32.768 kHz) 101b = Watchdog clock source /(2^{13}) (250 ms at 32.768 kHz) 110b = Watchdog clock source /(2^{9}) (15.625 ms at 32.768 kHz) 111b = Watchdog clock source /(2^{6}) (1.95 ms at 32.768 kHz)

Figure 7. Watchdog Timer Control Register. The WDT is interfaced through its 16-bit control register. We can select mode of operation (WDTMSEL and WDTHOLD), time interval (WDTIS), and clock source (WDTSSEL). To restart the counter we write a word into WTDCTL with the WDTCNTCL bit set to a logic '1'.

4 Watchdog Timer Example Programs

Let us consider an example of using WDT ISR to toggle a LED every every second (1 s off, 1 s on). The period is thus 2 s or the frequency of toggling is 0.5 Hz. Code 1 shows a program that toggles a LED in the watchdog timer interrupt service routine every second. To generate an interrupt request every second, we configure the WDT as follows: select the ACLK as the clock source, ACLK=32,768 Hz (or 2^{15} Hz), and select the tap to be 2^{15}; the WDT interval time will be exactly 1 s. The WDT control word will look like this: WDTMSEL selects interval mode, WDTSSEL selects ACLK, and WTTCNTCL clears the WDTCNT. Analyze the header file for msp430F5529.h to

locate pre-defined command words for the control register (e.g., WDT_ADLY_1000, WDT_ADLY_250, …).

```
/*-------------------------------------------------------------------------
 * File:        Lab7_D1.c (CPE 325 Lab7 Demo code)
 * Function:    Blinking LED1 using WDT ISR (MPS430F5529)
 *
 * Description: This C program configures the WDT in interval timer mode,
 *              clocked with the ACLK clock. The WDT is configured to give an
 *              interrupt for every 1s. LED1 is toggled in the WDT ISR
 *              by xoring P1.0. The blinking frequency of LED1 is 0.5Hz.
 *
 * Board:       MSP-EXP430F5529 (includes 32-KHZ crystal on XT1 and
 *                               4-MHz ceramic resonator on XT2)
 *
 * Clocks:      ACLK = XIN-XOUT = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                         MSP430F5529
 *                     -----------------
 *                 /|\|              XIN|-
 *                  | |                 | 32kHz crystal
 *                  --|RST           XOUT|-
 *                    |                 |
 *                    |            P1.0|-->LED1(RED)
 *                    |                 |
 * Input:       None
 * Output:      LED1 blinks at 0.5 Hz frequency
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *              Prawar Poudel
 * Date:        December 2008
 *-------------------------------------------------------------------------*/
#include <msp430.h>

void main(void) {
    WDTCTL = WDT_ADLY_1000;           // 1 s interval timer
    P1DIR |= BIT0;                    // Set P1.0 to output direction
    SFRIE1 |= WDTIE;                  // Enable WDT interrupt
    _BIS_SR(LPM0_bits + GIE);         // Enter LPM0 w/ interrupt
}

// Watchdog Timer Interrupt Service Routine
#pragma vector=WDT_VECTOR
__interrupt void watchdog_timer(void) {
    P1OUT ^= BIT0;                    // Toggle P1.0 using exclusive-OR
}
```

Code 1. Toggling a LED using WDT_ISR.

Code 2 shows the program that also toggles the LED1 every second in the WDT ISR. However, the watchdog timer uses the SMCLK as the clock source and the tap of 32,768 (2^{15}). The WDT generates an interrupt request every 32 ms. To toggle the LED1 every second we need to use a static local variable that is incremented every time we enter the ISR. When we collect 32

periods of 32 ms we have approximately 1 second of elapsed time, and we can then toggle LED1. Analyze the header file msp430f5529.h to locate pre-defined command word WDT_MDLY_32. What bits of the control register are set with WDT_MDLY_32? What is the purpose of using static variable in the ISR? What happens if we use a dynamically allocated variable?

```
1  /*-------------------------------------------------------------------------
2   * File:        Lab7_D2.c (CPE 325 Lab7 Demo code)
3   *
4   * Function:    Toggling LED1 using WDT ISR (MPS430F5529)
5   *
6   * Description: This C program configures the WDT in interval timer mode,
7   *              clocked with SMCLK. The WDT is configured to give an
8   *              interrupt for every 32ms. The WDT ISR is counted for 32 times
9   *              (32*32.5ms ~ 1sec) before toggling LED1 to get 1 s on/off.
10  *              The blinking frequency of LED1 is 0.5Hz.
11  *
12  * Clocks:      ACLK = XT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
13  *              An external watch crystal between XIN & XOUT is required for ACLK
14  *
15  *                      MSP430xF5529
16  *                    ------------------
17  *                /|\|              XIN|-
18  *                 | |                 | 32kHz
19  *                 --|RST          XOUT|-
20  *                   |                 |
21  *                   |           P1.0|-->LED1(RED)
22  *                   |                 |
23  * Input:       None
24  * Output:      LED1 blinks at 0.5Hz frequency
25  * Author:      Aleksandar Milenkovic, milenkovic@computer.org
26  *              Prawar Poudel
27  * Date:        December 2008
28  *-------------------------------------------------------------------------*/
29 #include <msp430.h>
30
31 void main(void)
32 {
33     WDTCTL = WDT_MDLY_32;           // 32ms interval (default)
34     P1DIR |= BIT0;                  // Set P1.0 to output direction
35     SFRIE1 |= WDTIE;                // Enable WDT interrupt
36
37     _BIS_SR(LPM0_bits + GIE);       // Enter LPM0 with interrupt
38 }
39
40 // Watchdog Timer interrupt service routine
41 #pragma vector=WDT_VECTOR
42 __interrupt void watchdog_timer(void) {
43     static int i = 0;
44     i++;
45     if (i == 32) {                  // 31.25 * 32 ms = 1s
46         P1OUT ^= BIT0;              // Toggle P1.0 using exclusive-OR
47                                     // 1s on, 1s off; period = 2s, f = 1/2s = 0.5Hz
48         i = 0;
```

```
    }
}
```

Code 2. Toggling the LED1 using WDT_ISR.

Code 3 shows the program that also toggles LED1 every second. The WDT is still configured in the interval mode and sets the WDTIFG every 1 s. The program however does not use the interrupt service routine (the interrupt from WDT remains disabled). Instead, the main program polls repeatedly the status of the WDTIFG. If it is set, LED1 is toggled and the WDTIFG is cleared. Otherwise, the program checks the WDTIFG status again. The program spends majority of time waiting for the flag to be set and this approach is known as software polling. It is inferior to using interrupt service routines, but sometimes can be used to interface various peripherals. What are the advantages of using interrupts over software polling?

```
/*-------------------------------------------------------------------------------
 * File:        Lab7_D3.c (CPE 325 Lab7 Demo code)
 * Function:    Blinking LED1 using software polling.
 * Description: This C program configures the WDT in interval timer mode and
 *              it is clocked with ACLK. The WDT sets the interrupt flag (WDTIFG)
 *              every 1 s. LED1 is toggled by verifying whether this flag
 *              is set or not. After it is detected as set, the WDTIFG is cleared.
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (2^20 Hz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                         MSP430F5529
 *                     -----------------
 *                 /|\|              XIN|-
 *                  | |                 | 32kHz
 *                  --|RST          XOUT|-
 *                    |                 |
 *                    |             P1.0|-->LED1(RED)
 *                    |                 |
 * Input:       None
 * Output:      LED1 blinks at 0.5Hz frequency
 *
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 * Revised by:  Prawar Poudel
 *-------------------------------------------------------------------------------*/
#include <msp430.h>

void main(void)
{
    WDTCTL = WDT_ADLY_1000;             // 1 s interval timer
    P1DIR |= BIT0;                      // Set P2.2 to output direction

    for (;;) {
        // Use software polling
        if ((SFRIFG1 & WDTIFG) == 1) {
            P1OUT ^= BIT0;
            SFRIFG1 &= ~WDTIFG;         // Clear bit WDTIFG in IFG1
        }
    }
}
```

Code 3. Toggling LED1 using WDT and Software Polling on WDTIFG.

5 Timer_A

In addition to the watchdog timer, the MSP430 family supports several types of other timer peripheral devices, such as, Real Time Clock, Timer A, Timer B and Timer D. In this part of the tutorial we will be studying Timer A and B.

We have seen that we rely on timer peripherals to measure time, timestamp external or internal events, generate periodic events, and generate digital outputs of desired period and duty cycle. A timer can use a different clock signal from the one used by the processor, so it is possible either to turn off the processor or to work on other computations while the timer is counting. This saves energy and reduces code complexity. Note that a MSP430 can have multiple timers and each timer can be utilized independently from the other timers. Furthermore, each timer has several modes of counting.

Figure 8 shows a block diagram of Timer A peripheral. It consists of a timer block and up to seven configurable capture and compare blocks (TAxCCR0 – TAxCCR6). The MSP430F5529 has three Timer A peripherals as follows: TA0 with 5 capture and compare blocks, TA1 with 3 capture and compare blocks, and TA2 with 3 capture and compare blocks. Thus, TAxCCR0 can mean any of TA0CCR0, TA1CCR0 or TA2CCR0. Timer B is similar to Timer A and has 7 capture and compare registers.

5.1 Timer Block

The timer block includes a 16-bit counter TAxR that increments or decrements on the rising edge of every clock cycle. This counter can be read or written with software. The timer block supports 4 counting modes: STOP (MCx=00), UP (MCx=01), Continuous (MCx=10), and UP/DOWN (MCx=11). The Up mode allows for setting timer periods through TAxCCR0 (see Figure 9). The timer repeatedly counts up to the value in TAxCCR0 for the period of TAxCCR0+1 clock cycles. The Continuous mode repeatedly counts from 0x0000 up to 0xFFFF as shown in Figure 10. The Up/Down mode is used when the period is different from 0xFFFF and symmetrical pulse generation is needed. The timer repeatedly counts up to the value in TAxCCR0 and back down to 0x0000. The period is twice the value in TAxCCR0 as shown in Figure 11. To illustrate this, let us assume that TAxCCR0=4. The counting states in one period are $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$ (2*4 = 8 states).

The source clock can be selected from multiple options (TAxSEL bits), and the selected clock can be further divided by 1 (IDx=00), 2 (IDx=01), 4 (IDx=10), and 8 (IDx=11). The timer block is configured using TAx control register, TAxCTL, which contains control bits TAxSEL, IDx, CNTLx, and others. Please examine the format of this register (Figure 12).

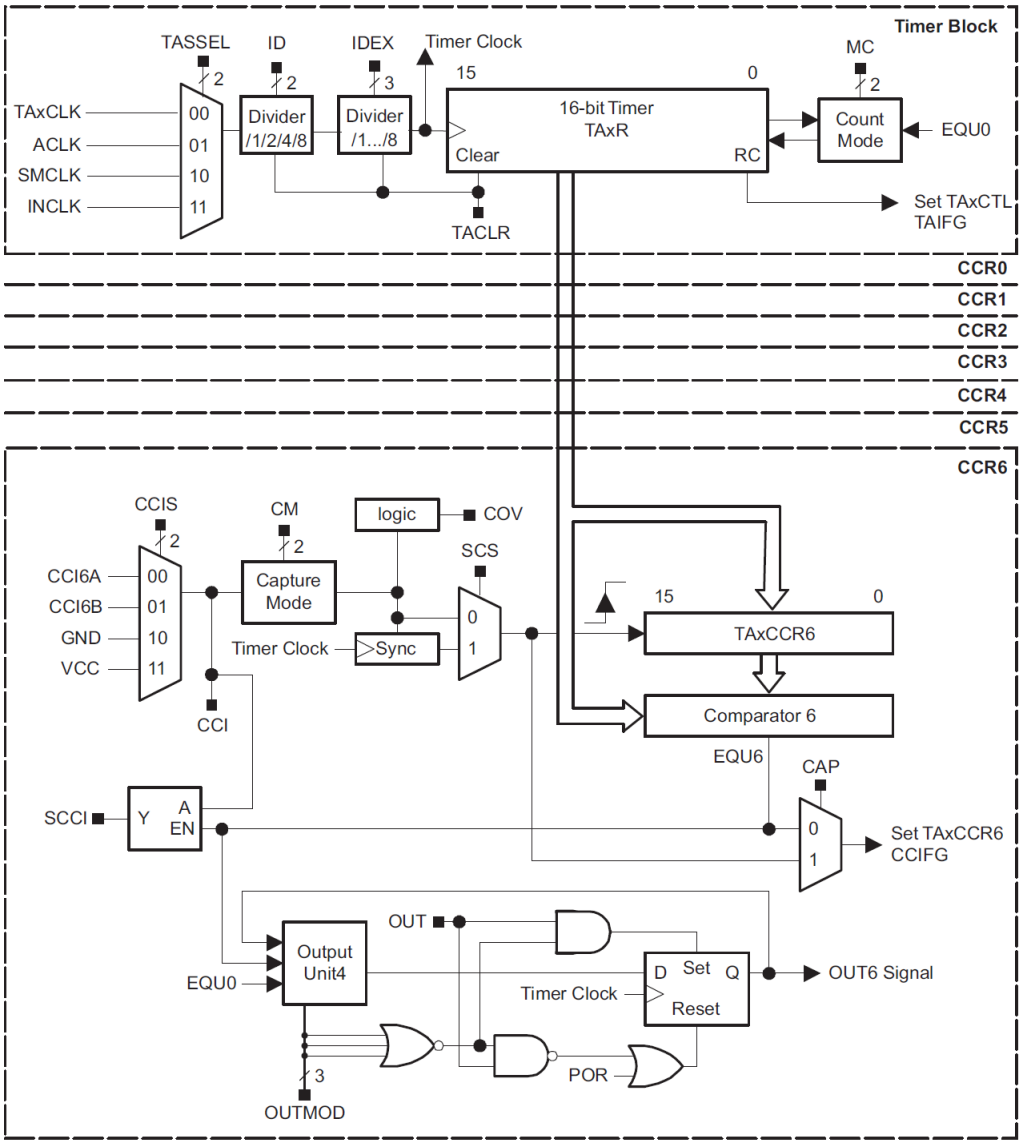


Figure 8. Timer_A Block Diagram

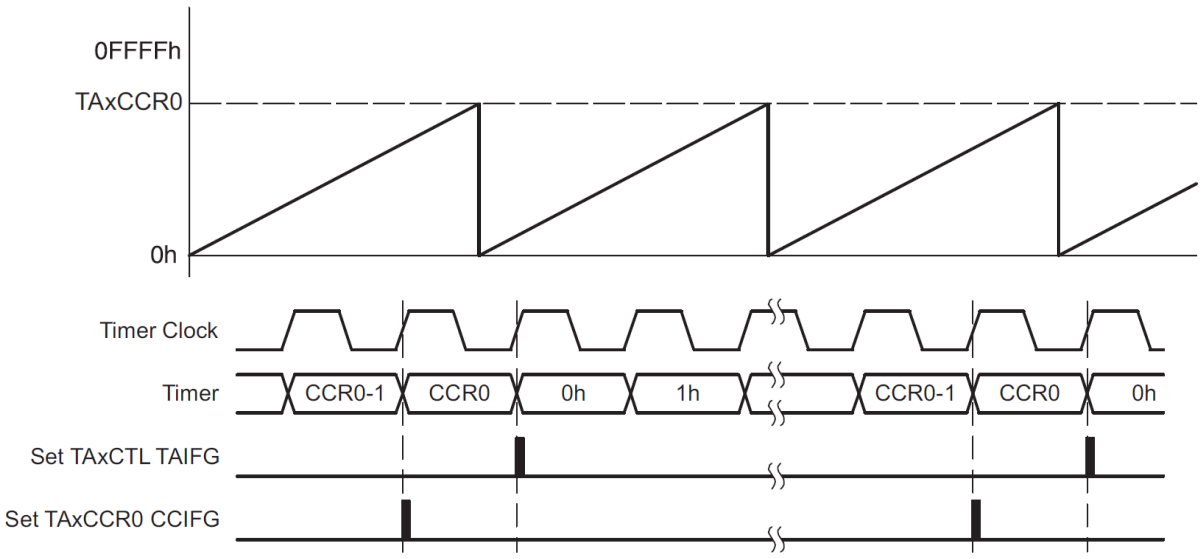


Figure 9. Up mode and flag settings. The TAxCCR0 CCIFG is set when counting from TAxCCR0-1 to TAxCCR0 and TAIF is set when counting from TAxCCR0 to 0.

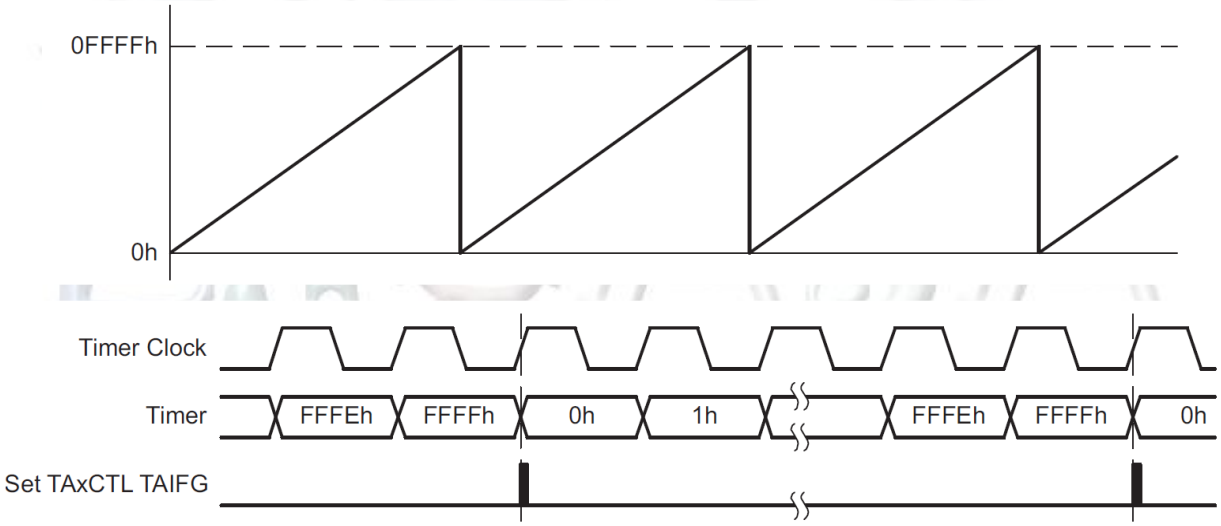


Figure 10. Continuous mode and flag settings. TAIFG is set when counting from 0xFFFF to 0x0000.

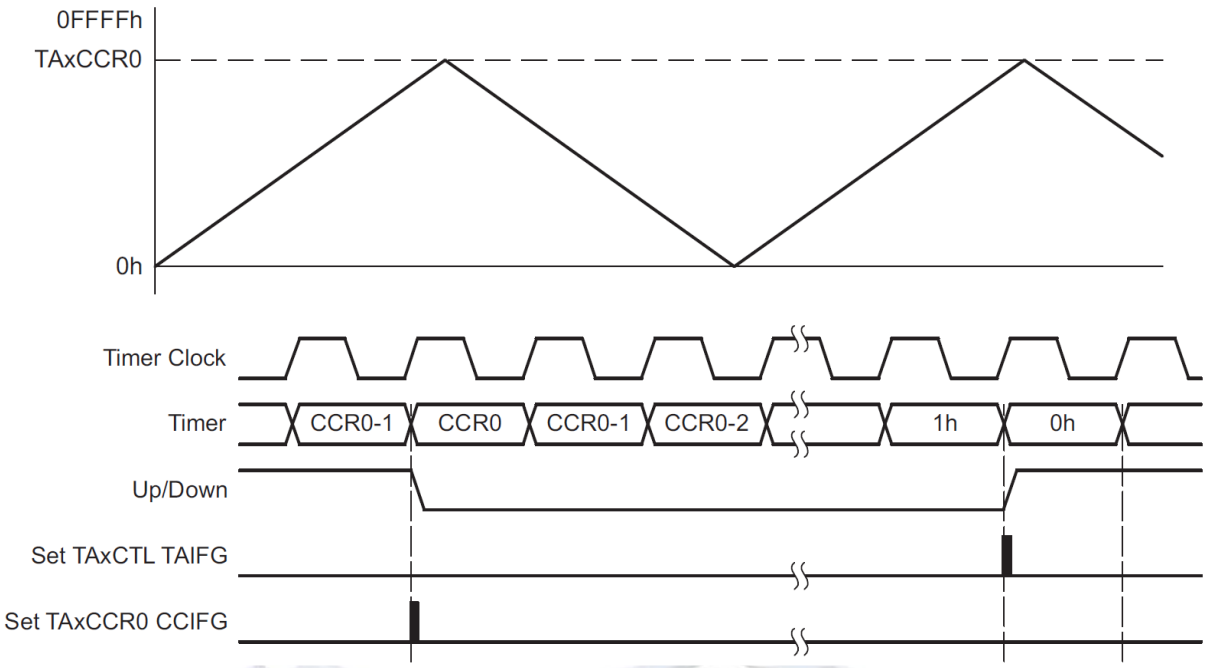


Figure 11. Up/Down mode and flag settings. TAIFG is set when counting from 0x0001 to 0x0000 and TAxCCR0 CCIFG is set when counting from CCR0-1 to CCR0.

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)

Bit	Field	Type	Reset	Description
15-10	Reserved	RW	0h	Reserved
9-8	TASSEL	RW	0h	Timer_A clock source select 00b = TAxCLK 01b = ACLK 10b = SMCLK 11b = INCLK
7-6	ID	RW	0h	Input divider. These bits along with the TAIDEX bits select the divider for the input clock. 00b = /1 01b = /2 10b = /4 11b = /8
5-4	MC	RW	0h	Mode control. Setting MC = 00h when Timer_A is not in use conserves power. 00b = Stop mode: Timer is halted 01b = Up mode: Timer counts up to TAxCCR0 10b = Continuous mode: Timer counts up to 0FFFFh 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. 0b = Interrupt disabled 1b = Interrupt enabled
0	TAIFG	RW	0h	Timer_A interrupt flag 0b = No interrupt pending 1b = Interrupt pending

Figure 12. TAxCTL. The TAxCTL includes bits for selecting input clock source (TASSEL), input clock divider (ID), and counting mode (MC). We can also force counter reset by writing a word with TACLR bit set. Bits 1 and 0 are reserved for TAxIE and TAxIFG.

5.2 Capture & Compare Block

Each *Capture and Compare block* (*CCn*, *n* from 0 to 6) contains a 16-bit latch register TAxCCRn and corresponding control logic enabling two type of operations CAPTURE and COMPARE. Each capture and compare block *n* is controlled by its own control register, TAxCCTLn.

Capture refers to an operation where the value of the running counter (TAxR) is captured in the TAxCCRn on a hardware event (e.g., external input changes its state from a logic 1 to a logic 0 or from a logic 0 to a logic 1) or on a software trigger (e.g., setting some control bits).

Compare refers to an operation where actions are triggered at specific moments in time. As discussed above, this operation is crucial for generating Pulse-Width-Modulated signals (PWMs) – periodic signals where the period and duty cycle is fully controlled. In the compare mode of operation, the corresponding latch register, TAxCCRn, is initialized to a certain value. When the value in the running counter (TAR) reaches the value in TAxCCRn, an output signal can change its state (set, reset, or toggle).

The shape of the output signal is controlled by the OUTMOD control bits, the values in TAxCCRn registers, and the counting mode. Let us consider an example of the timer running in UP mode and using TAxCCR1 to create a PWM signal at OUT1 as shown in Figure 13. The OUT1 signal is

changed when the timer counts up to the TAxCCR1 value and rolls from TAxCCR0 to zero, depending on the output mode. In *Output Mode 1: Set*, the output is asserted when the running counter reaches the value in TAxCCR1 (EQU1 is asserted). EQU1 is the output of the comparator in TAxCCR1 block that compares the value in the TAxCCR1 register and the current value of the running timer (TAxR). In *Output Mode 2: Toggle/Reset*, the output OUT1 signal is toggled when EQU1 is set, and then reset when the counter rolls over to zero (EQU0 is asserted). EQU0 is asserted when TAxCCR0 matches the value in the running counter. In *Output Mode 3: Set/Reset*, the output OUT1 signal is set when EQU1 is set and then reset when EQU0 is asserted. The *Output Mode 4: Toggle* changes the output signal OUT1 only when EQU1 is asserted. Similar rules apply to other output modes. If the counter is running in the continuous mode, the output mode logic acts in the same way, except that the period is now 65,536 clock cycles. What is the period of the output signal created in Output Mode 3? What is the period of of the output signal created in Output Mode 4?

Multiple capture and compare blocks can work concurrently, each generating its own output signal. Please note that they all share a single timer block, implying that the period of output signals will be the same, but their shape and duty cycle can be individually controlled.

Figure 14 illustrates the OUT2 signal for different OUTMOD settings when the timer block is running in UP/DOWN mode. Please note that events triggering changes on the OUT2 signal are when the running counter matches the value in TAxCCR0 (EQU0 is asserted) and when the running counter matches the value in TACCR2 (EQU2 is asserted). When interpreting the naming convention please note that first word describes what happens with the output when EQU2 is asserted and the second word describes what happens with the output when EQU0 is asserted.

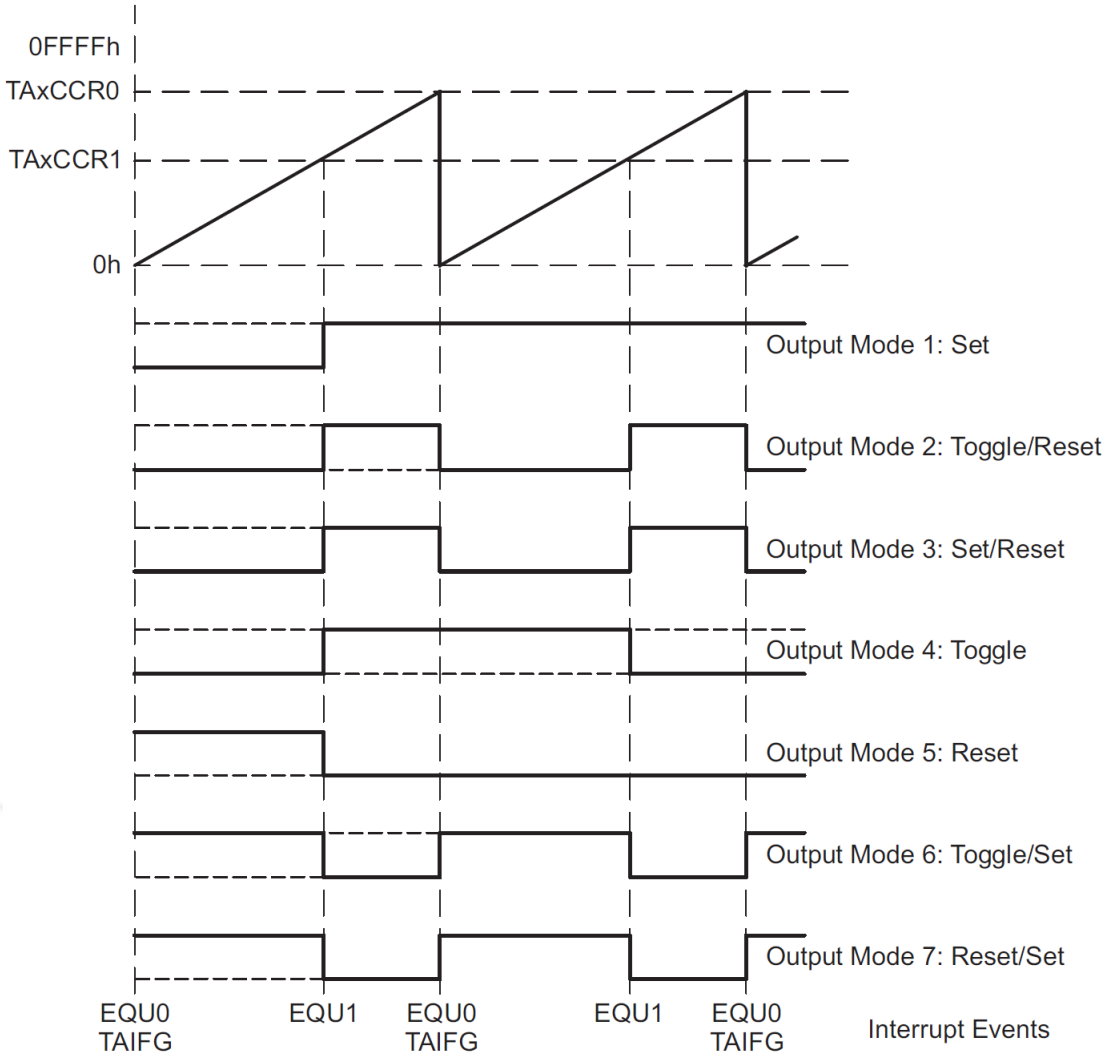


Figure 13. Output Example – timer in UP mode.

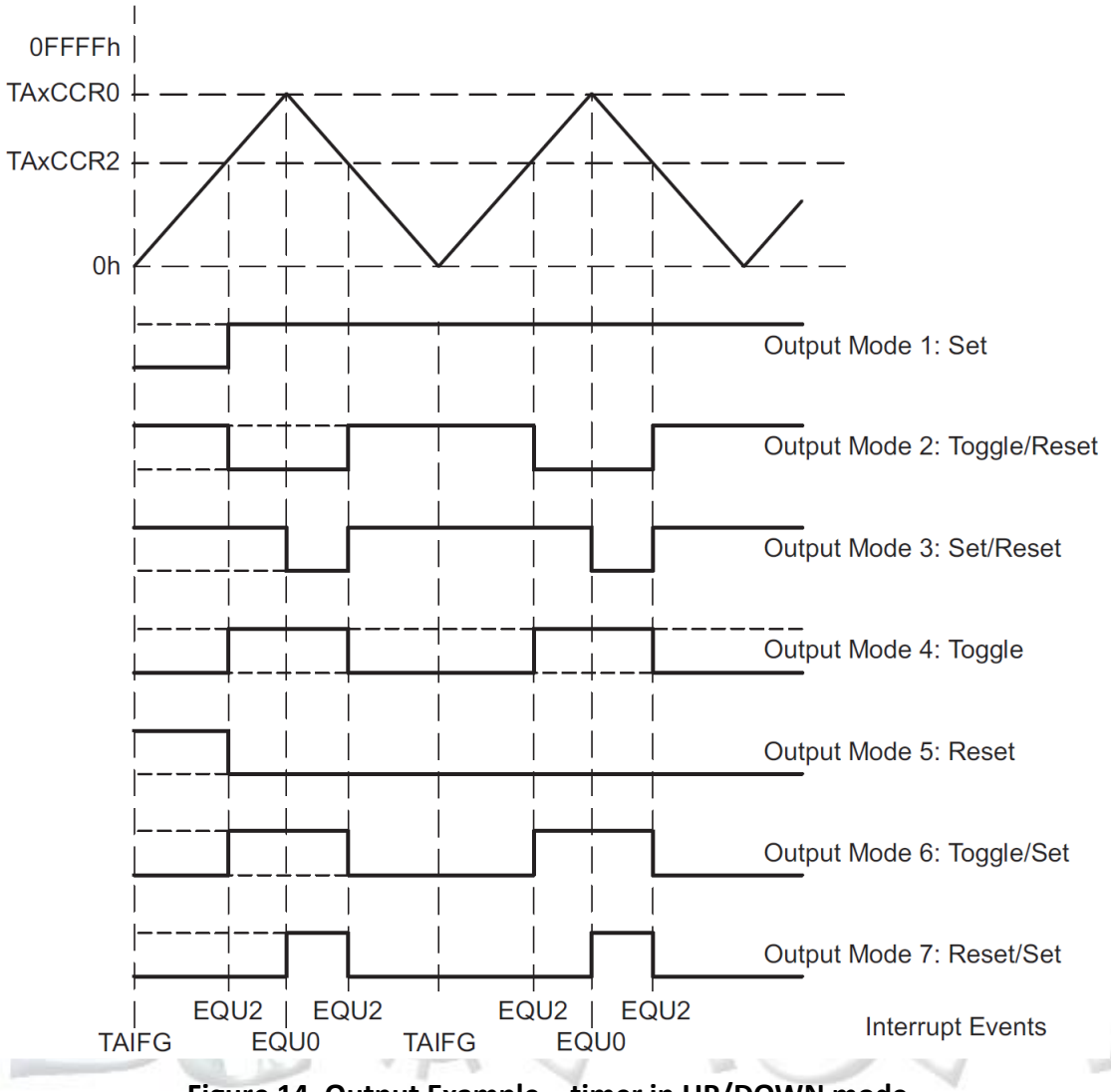


Figure 14. Output Example – timer in UP/DOWN mode.

Every capture and compare block of Timer_A has its own control register, TAxCCTn as described in Figure 15. The control register allows for selecting capture mode (CM bits), capture/compare input select (CCIS), synchronize capture source (SCS), mode of operation (CAP), output mode (OUTMOD), output state (OUT), capture and compare interrupt enable (CCIE) and interrupt flage (CCIFG).

15	14	13	12	11	10	9	8
CM		CCIS		SCS	SCCI	Reserved	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)	r-(0)	rw-(0)
7	6	5	4	3	2	1	0
OUTMOD			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

Bit	Field	Type	Reset	Description
15-14	CM	RW	0h	Capture mode 00b = No capture 01b = Capture on rising edge 10b = Capture on falling edge 11b = Capture on both rising and falling edges
13-12	CCIS	RW	0h	Capture/compare input select. These bits select the TAxCCR0 input signal. See the device-specific data sheet for specific signal connections. 00b = CCIxA 01b = CCIxB 10b = GND 11b = VCC
11	SCS	RW	0h	Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock. 0b = Asynchronous capture 1b = Synchronous capture
10	SCCI	RW	0h	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read from this bit.
9	Reserved	R	0h	Reserved. Reads as 0.
8	CAP	RW	0h	Capture mode 0b = Compare mode 1b = Capture mode
7-5	OUTMOD	RW	0h	Output mode. Modes 2, 3, 6, and 7 are not useful for TAxCCR0 because EQUx = EQU0. 000b = OUT bit value 001b = Set 010b = Toggle/reset 011b = Set/reset 100b = Toggle 101b = Reset 110b = Toggle/set 111b = Reset/set
4	CCIE	RW	0h	Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag. 0b = Interrupt disabled 1b = Interrupt enabled
3	CCI	R	0h	Capture/compare input. The selected input signal can be read by this bit.
2	OUT	RW	0h	Output. For output mode 0, this bit directly controls the state of the output. 0b = Output low 1b = Output high
1	COV	RW	0h	Capture overflow. This bit indicates a capture overflow occurred. COV must be reset with software. 0b = No capture overflow occurred 1b = Capture overflow occurred
0	CCIFG	RW	0h	Capture/compare interrupt flag 0b = No interrupt pending 1b = Interrupt pending

Figure 15. TAxCCTLn Register.

5.3 Timer_A Interrupts

Two interrupt vectors are associated with 16-bit Timer_A:

- TAxCCR0 for TAxCCR0 CCIFG (single-sourced interrupt);
- TAxIV for all other CCIFG flags and TAIFG (multi-sourced interrupt).

In capture mode, the corresponding CCIFG flag is set when a timer value is captured in the associated TAxCCRn register. In compare mode, the corresponding CCIFG flag is set if TAxR counts to the associated TAxCCRn value. Software may also set or clear any CCIFG flag. All CCIFG flags request an interrupt when their corresponding CCIE bit and the GIE bit are also set.

6 Timer_B

Timer_B is identical to Timer_A with the following exceptions:

- the length of TImer_B counter is programmable to be 8, 10, 12, or 16 bits;
- Timer_B TBxCCRn registers are double-buffered and can be grouped;
- All Timer_B outputs can be put into a high-impedance state;
- The SCCI bit function is not implemented in Timer_B.

For learning more about Timer_B, please consult the corresponding User's guide.

7 Timer_A Example Programs

7.1 Toggle an Output Using Timer_A

Let us first consider an example where we utilize the Timer A2 (TA2) device to toggle an output on the MSP-EXP430F5529LP board. For this example, we will be toggling the channel 1 (Capture&Compare Block 1) of Timer A2, i.e. TA2.1. TA2.1 is multiplexed with the digital I/O pin P2.4.

P2.4 should be periodically turned on for 0.065 seconds and then turned off for 0.065 seconds (one period is ~0.13 seconds, or toggling rate is ~7.6 Hz). We have learned how to execute the toggling using a software delay or by using the watchdog timer. Now, we would like to utilize the MSP430's TA2 peripheral device.

Code 4 shows a program that toggles P2.4 as specified. P2.4 is multiplexed with TA2.1 output signal, so its special special function should be selected. To configure P2.4 as special function, the P2 selection register, P2SEL, pin 4 is set to its special I/O function instead of its common digital I/O function (P2SEL |= BIT4;). This way we ensure that the P2.4 mirrors the behavior of the TA2.1 output signal.

The capture and compare block 1 can be configured to set/reset or toggle the output signal TA2.1 when the value in the running counter reaches the value in the capture and control register TA2CCR1. Thus, when a value in the Timer A2 counter is equal to the value in TA2CCR1, we can configure the Timer A2 to toggle its output, TA2.1, automatically. The default value in TA2CCR1 is 0, thus, the output will be toggled every time the counter rolls over to 0x0000.

The next step is to configure clock sources. The MSP430 clocks MCLK, SMCLK, and ACLK have default frequencies as follows: MCLK = SMCLK ~ 1 MHz and ACLK = 32 KHz. Timer A2 is

configured to use the SMCLK as its clock input and to operate in the continuous mode. Timer A2's counter will count from 0x0000 to 0xFFFF. When the counter value reaches 0x0000, the EQU1 will be asserted indicating that the counter has the same value as the TA2CCR1 (here it is not set because by default it is cleared). We can select the output mode 4 (toggle) that will toggle the output every time EQU1 is asserted. This way we can determine the time period when the TA2.1 is reset and set. The TA2.1 will be set for 65,536*1/2^20 = 0.0625 seconds and will be reset for 65,536*1/2^20 = 0.0625 seconds. Please note that we do not need to use an interrupt service routine to toggle the signal in this case. The Timer A2 will toggle the P2.4 independently, and we can go into a low power mode and remain there for the rest of the application lifetime.

To visualize the output, you can use the Grove Boosterpack. P2.4 is connected to header J14 on the digital section of the board. Using the connecter cable to hook the Buzzer in the pack, you can notice the output in the note played in Buzer.

```
/*-------------------------------------------------------------------------
 * File:        Lab7_D4.c (CPE 325 Lab7 Demo code)
 *
 * Function:    Toggling signal using Timer_A2 in continuous mode (MPS430F5529)
 *
 * Description: In this C program, Timer_A2 is configured for continuous mode. In
 *              this mode, the timer TA2 counts from 0 up to 0xFFFF (default 2^16).
 *              So, the counter period is 65,536*1/2^20 = 62.5ms when SMCLK is
 *              selected. The TA2.1 output signal is configured to toggle every
 *              time the counter reaches the maximum value, which corresponds to
 *              62.5ms. TA2.1 is multiplexed with the P2.4, and there is a extension
 *              header from this pin.
 *
 *              Thus the output frequency on P2.4 will be f = SMCLK/(2*65536) ~ 8 Hz.
 *              Please note that once configured, the Timer_A toggles the signal
 *              in pin P2.4 automatically even when the CPU is in sleep mode.
 *              Please use oscillator to see this.
 *
 *              Using the Grove Boosterpack, you can hook-up the Buzzer to the
 *              J14 header. This connects the Signal Pin of buzzer to P2.4.
 *              The buzzer produces sound when the signal value is high
 *              and vice versa.
 *
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (2^20 Hz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                           MSP430F5529
 *                       -----------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |       P2.4/TA2.1|-->Buzzer
 *                      |                 |
 * Input:       None
 * Output:      Toggle output at P2.4 at 8Hz frequency using hardware TA2
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
```

```
 *              Prawar Poudel
 *-------------------------------------------------------------------------------*/
#include <msp430F5529.h>

void main(void) {
    WDTCTL = WDTPW +WDTHOLD;  // Stop WDT

    P2DIR |= BIT4;            // P2.4 output (TA2.1)
    P2SEL |= BIT4;            // P2.4 special function (TA2.1 output)

    TA2CCTL1 = OUTMOD_4;      // TA2.1 output is in toggle mode
    TA2CTL = TASSEL_2 + MC_2; // SMCLK is clock source, Continuous mode

    _BIS_SR(LPM0_bits + GIE); // Enter Low Power Mode 0
}
```

Code 4. C Program for Toggling Pin2.4 Using TimerA, Continuous Mode

Try to modify the code from Code 4 by selecting a different source clock for Timer A. What happens if we use the following command: TA2CTL = TASSEL_1 + MC_2? What is the period of toggling the signal? Explain your answer. Try using divider for the clock source.

The given example may not be suitable if you want to control the period of toggling since the counter in the continuous mode always counts from 0x0000 to 0xFFFF. This problem can be solved by opting for the UP mode. The counter will count from 0x000 up to the value specified in the TA2CCR0. This way we can control the time period. Let us consider an example where we want the buzzer to be 1 second on and 1 second off (toggling rate is 0.5 Hz).

Code 5 shows the C code for this example. Note the changes. How do we specify UP mode? How do we select the ACLK clock as the TimerA source clock? What output mode do we use? Is it better to use ACLK instead of SMCLK in this example? Explain your answers.

```
/*-------------------------------------------------------------------------------
 * File:        Lab7_D5.c (CPE 325 Lab7 Demo code)
 *
 * Function:    Toggle signal using Timer_A2 in up mode (MPS430F5529)
 *
 * Description: In this C program, Timer_A2 is configured for UP mode. In this
 *              mode, the timer TA2 counts from 0 up to value stored in TA2CCR0.
 *              So, the counter period is CCR0*1us. The TA2.1 output signal is
 *              configured to toggle every time the counter reaches the value
 *              in TA2CCR1. TA2.1 is multiplexed with the P2.4. Thus, the output
 *              frequency on P2.4 will be f = ACLK/(2*CCR0) = 0.5Hz. Please note
 *              that once configured, the Timer_A2 toggles the signal automatically
 *              even when the CPU is in sleep mode.
 *
 *              Using the same connection as in Lab7_D4.c, you should be able to
 *              hear Buzzer ON for 1s and OFF for 1s continuously.
 *
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (2^20 Hz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
```

```
 *
 *                          MSP430xF5529
 *                      -----------------
 *                  /|\|              XIN|-
 *                   | |                 | 32kHz
 *                   --|RST          XOUT|-
 *                     |                 |
 *                     |       P2.4/TA2.1|--> Buzzer
 *                     |                 |
 * Input:       None
 * Output:      Toggle output at P2.4 at 0.5Hz frequency using hardware TA2
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *              Prawar Poudel
 *-------------------------------------------------------------------------------*/
#include <msp430F5529.h>

void main(void) {
    WDTCTL = WDTPW +WDTHOLD;    // Stop WDT

    P2DIR |= BIT4;              // P7.4 output
    P2SEL |= BIT4;              // P7.4 special function (TB0.2 output)

    TA2CCTL1 = OUTMOD_4;        // TB0.2 output is in toggle mode
    TA2CTL = TBSSEL_1 + MC_1;   // ACLK is clock source, UP mode
    TA2CCR0 = 32767;            // Value to count upto for Up mode

    _BIS_SR(LPM3_bits + GIE);   // Enter Low Power Mode 3
}
```

Code 5. C Program for Toggling Pin2.4 Using TIMER_A (UP MODE)

7.2 Additional Timer_A Functionality

As already seen, Timer A is quite powerful due to the selectable clocks, automated outputs, and adjustable maximum count value. The Timer A peripheral has additional features which greatly expand its functionality and versatility. These features include:

- Multiple capture/compare modules
- Multiple output control modes
- Ability to call multiple interrupts at different count values
- Ability to select from multiple counting modes

Often times, it will be necessary to perform multiple tasks with a single timer peripheral. Fortunately, the Timer A system has multiple channels that can be set up to perform their tasks at designated count values. The MSP430F5529 has 3 instantiations of Timer A, namely TA0, TA1 and TA2. Each of these Timer A devices has different number of channels. TA0 has 5 channels, TA1 has 3 channels and TA2 has 3 channels. Each channel normally has a shared output pin similar to what we saw with the TA2.1 pin in the examples above. In Figure 16 below, note that

some of the pins are shared with TA1 channels. This means that capture/compare channel can directly control output devices connected to these pins.

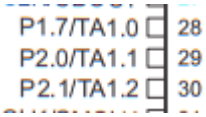


Figure 16. Port pins shared with TA1 channels

If you further examine the MSP-EXP430F5529LP experimenter board schematic, you can find where the other channel output pins are located. In order to use other channels in each instantiation of each of TA0, TA1 and TA2, channel 0 must still be set up in each of them. All of the channels have their own configuration register and their own count value register. Each channel can be configured to use its output pin directly (as in above example Lab7_D5), but they can also be used to call interrupt service routines. An interrupt vector is dedicated to channel 0, but other channels each of the timers can be configured to call a separate ISR.

It is important to think about how the counting methods affect the interrupt calls from the different capture/compare channels. The user's guide contains definitive information about the Timer B including examples that demonstrate how the various functions work. In general, the counting modes work as follows:

Counting mode 0 – Stop mode – The timer is inactive

Counting mode 1 – Up mode – The timer counts up to the value for channel 0. An interrupt for each channel set up is called at the corresponding count value on the way up. At the maximum value an interrupt for channel 0 is called, and at the next timer count a general interrupt is generated. Remember that these interrupts may correspond to output pin control or interrupt service routine vectoring depending on the channel configuration.

Counting mode 2 – Continuous mode – The timer counts up to its maximum value (65535 for 16 bit mode). Along the way, corresponding interrupts are called at each channel's count value register. At 0 a general Timer Ax interrupt is set.

Counting mode 3 – Up/Down mode – The timer counts up to the value in the channel 0 register, and then it counts back down to 0 again. The channel interrupts are called when the value is reach on the up count and the down count. The general Timer A interrupt is called when 0 is reached.

In Code 6 below, channels 0 and 1 are used to call two separate ISRs. Since the timer is in up/down mode, the channel 0 ISR is only called once per counting cycle (on the max value set by the CCR0 register) while the channel 1 ISR is called twice per counting cycle if its CCR1 value is less than CCR0. It is called on the up count and the down count. This mode is especially useful when creating PWM signals since the count register value determines the duty cycle of the output signal.

```
/*-------------------------------------------------------------------------
```

```
 * File:         Lab7_D6.c (CPE 325 Lab7 Demo code)
 *
 * Function:     Blinking LED1 & LED2 using Timer_A0 with interrupts (MPS430F5529)
 *
 * Description: In this C program, Timer_A0 is configured for up/down mode with
 *              ACLK source and interrupts for channel 0 and channel 1 are
 *              enabled. In up/down mode timer TA0 counts the value from 0 up to
 *              value stored in TA0CCR0 and then counts back to 0. The interrupt
 *              for TA0 is generated when the counter reaches value in TA0CCR0.
 *              The interrupt TA0.1 is generated whenever the counter reaches value
 *              in TA0CCR1. Thus, TA0.1 gets two interrupts while counting upwards
 *              and counting downwards. This simulates a PWM control - adjusting
 *              the TA0.1 and TA0.0 CCR register values adjusts the duty cycle of the
 *              PWM signal.
 *
 * Clocks:       ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (2^20 Hz)
 *               An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                           MSP430x5529x
 *                       -----------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |             P1.0|--> LED1(RED)
 *                      |             P4.7|--> LED2(GREEN)
 * Input:        None
 * Output:       LED1 blinks at 1.64Hz with 20-80% duty cycle and LED2 blinks at
 *               0.82Hz with 50-50% duty cycle.
 *
 * Author:       Aleksandar Milenkovic, milenkovic@computer.org
 *               Prawar Poudel
 *-------------------------------------------------------------------------------*/
#include <msp430F5529.h>

void main(void) {
    WDTCTL = WDTPW + WDTHOLD;     // Stop WDT
    _EINT();                      // Enable interrupts

    P1DIR |= BIT0;                //LED1 as output
    P4DIR |= BIT7;                //LED2 as output

    P1OUT &= ~BIT0;               // ensure LED1 and LED2 are off
    P4OUT &= ~BIT7;

    TA0CCTL0 = CCIE;              // TA0 count triggers interrupt
    TA0CCR0 = 10000;              // Set TA0 (and maximum) count value

    TA0CCTL1 = CCIE;              // TA0.1 count triggers interrupt
    TA0CCR1 = 2000;               // Set TA0.1 count value

    TA0CTL = TASSEL_1 | MC_3;     // ACLK is clock source, UP/DOWN mode

    _BIS_SR(LPM3);                // Enter Low Power Mode 3
}
```

```
#pragma vector = TIMER0_A0_VECTOR
__interrupt void timerISR(void) {
    P4OUT ^= BIT7;                  // Toggle LED2
}

#pragma vector = TIMER0_A1_VECTOR
__interrupt void timerISR2(void) {
    P1OUT ^= BIT0;                  // Toggle LED1
    TA0CCTL1 &= ~CCIFG;             // Clear interrupt flag
}
```

Code 6. Code Using Multiple Timer B CC Channels and ISRs to Toggle LEDs

8 Exercises

Problem 1.

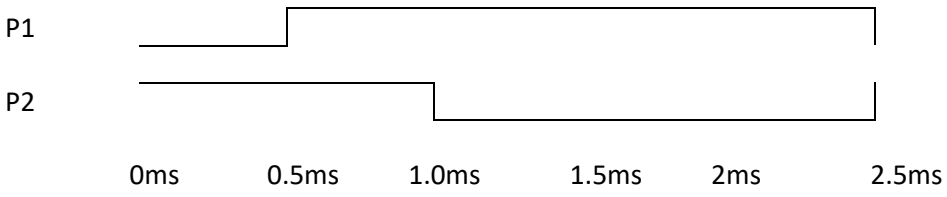
Consider the following code segment that utilizes the watchdog timer in the interval mode with a period set in line 4 of the code.

```
1.  #include <msp430xG46x.h>
2.  void main(void) {
3.    int p = 0;
4.    WDTCTL = WDT_ADLY_250; // check the meaning of this constant in the include file
5.    P2DIR |= BIT2; // Set P2.2 to output direction
6.    P2OUT &= ~BIT2; //
7.    for (;;) {
8.      if ((IFG1 & WDTIFG) == 1) {
9.        p++;
10.       IFG1 &= ~WDTIFG;
11.       if (p == 4) P2OUT ^= BIT2;
12.       if (p == 13) { P2OUT ^= BIT2; p=0;}
13.     }
14.    }
15. }
```

A. What does the code segment do? What does the code in line 10 do?

B. How would you implement the given functionality using an interrupt service routine.

C. You would like to generate two periodic pulse-width modulated (PWM) signals P1 and P2, with frequency of 400 Hz (one period is 2.5 ms). Assume that an 2^20 Hz clock on SMCLK is used by TimerB. Can you do this using TimerB? If yes, describe a TimerB configuration (content of control and data registers) that will carry out signal generation? Note: use English and waveforms to describe your solution.



Problem 2.

Consider the following code segment. Assume that processor clock in the active mode is set to 1,000,000 Hz. Assume that P3 is configured as output and initially P30UT, bit 5 is set to a logic '1'.

```
1.  while(1) {
2.    int i;
3.    for(i = 2000; i>0; i--);           // one loop iteration takes 5 clock cycles
4.    P3OUT ^= BIT5;                     //
5.    for (i = 1000; i>0; i--);          // Delay
6.    P3OUT |= BIT5;                    //
7.  }
```

A. What does the code segment do assuming that P3.5 is configured as a digital output. You may ignore delay needed to execute instructions in lines 4 and 6.

B. How would you implement functionality achieved by the code segment above using TimerB. Port P3.5 is multiplexed with the output signal from the capture and compare block 4 of TimerB. Give details. How would you initialize the system? What would you do in the main loop? Assume the SMCLK is used which is 2^20 =1,048,576 Hz.

LaCASA