MSP430 Timers: PWM (Compare Mode), Capture Mode, Frequency Measurement, and Applications

Embedded Systems Lab Report

September 22, 2025

Contents

Chapter 1

Overview

This document presents solutions and code for the lab tasks involving MSP430 timers:

1. Compare mode (PWM) and Capture mode tests and understanding.
2. Generating a DC average voltage using PWM (technique, components, calculations).
3. Frequency generation and measurement (reciprocal capture).
4. Generating an aperiodic burst of x square cycles.
5. Stopwatch using pushbutton + capture (P2.7 wired to P1.2 / TA0.CCI2A).

Assumptions made consistently in the examples:

- Default **SMCLK = 1 MHz** (as provided in your course material).
- Timer_A is 16-bit: maximum count 65535.
- Timer in **Up mode**: counts from 0 to TAxCCR0 inclusive; period ticks = TAxCCR0+ 1.
- Digital output high voltage during PWM is $V_{HIGH} = V_0$ (so average $V_{avg} = \text{duty} \times V_0$).

Chapter 2

1. Compare mode (PWM) vs Capture mode — concept and quick checks

2.1 Compare (PWM) mode

- Timer run in up mode with TAxCCR0 = period (counts).
- TAxCCRn (n¿=1) configured with an output mode (e.g. OUTMOD_7 = Reset/Set) produces PWM on the associated TAx.n pin.
- Duty cycle $= \dfrac{\text{TAxCCRn}}{\text{TAxCCR0} + 1}$ (if TAxCCRn is the compare count at which output changes).
- Use a DMM in DC mode (measuring voltage average) or an oscilloscope to measure waveform and duty. DMM will read average voltage roughly $V_{avg} = duty \times V_{HIGH}$.

2.2 Capture mode

- Timer capture input (TAx.CCIyA) latches current timer value into TAxCCRy on an edge (rising/falling/both).
- Useful to measure time differences: differences of consecutive captures give period (or high/low durations).
- To measure an unknown frequency, route the signal to a capture pin; on two rising-edge captures:

  $$\Delta = \text{capture2} - \text{capture1} \quad \text{(with overflow handling)}$$

  frequency $f = \dfrac{f_{timer}}{\Delta}$ where f_{timer} is the timer tick rate (e.g. SMCLK).

2.3 How to check on your board

- Run PWM code (compare): measure average voltage using DMM (DC mode). Compare to expected V_{avg}.

- Run capture code: feed the PWM signal into the capture pin and compute frequency from captured deltas. Compare to expected frequency.

Chapter 3

2. DC motor speed control by varying DC voltage using timer-generated PWM

3.1 (a) Name of technique

The technique is called **Pulse Width Modulation (PWM)** followed by low-pass filtering (or the motor inductance acting as a low-pass) to obtain an effective DC voltage (average voltage). For motors specifically, PWM is frequently used (the motor driver behave as an electrical low-pass, producing an effective torque proportional to the average voltage/current).

3.2 (b) Additional components required to obtain DC voltage

To convert the PWM waveform to a smoother DC voltage you typically need:

- A low-pass filter (simple RC or better LC filter) to average the PWM into voltage: resistor + capacitor (or in practice an inductor + capacitor for motor drive).
- A **power stage / driver** (e.g. MOSFET or transistor driver or H-bridge) capable of supplying motor current. The MSP430 PWM pin cannot drive motor current directly.
- Flyback diode or proper motor driver (for DC motor inductive load) if not using an H-bridge that handles it.
- Optional: LC filter, smoothing capacitor, buffer (op amp) — depending on quality of DC required.

3.3 (c) Can CCR0 alone generate the required signal?

- **Short answer:** No, not for arbitrary duty ratios. TAxCCR0 determines the PWM period in up mode. To produce arbitrary duty you normally need another compare register (TAxCCR1 or TAxCCRn) to set the compare time when the pin toggles (or set/reset).

- You can produce 50% duty with clever toggling using CCR0 only (toggle output on CCR0, use interrupts or toggle mode) but arbitrary duty (25%, 75%) requires a second compare (CCR1) or software interrupts to toggle at the desired moment — software toggling increases jitter and CPU load.

3.4 Counts required to get V0/4 and 3V0/4

We choose an example PWM frequency for clarity. You can change frequency by adjusting TA0CCR0.

Example: choose PWM frequency $f_{PWM} = 1\,\text{kHz}$. With SMCLK = 1 MHz:

$$\text{Period counts} = \frac{f_{timer}}{f_{PWM}} = \frac{1,000,000}{1,000} = 1000 \text{ ticks.}$$

In up mode TA0CCR0 = period - 1 = 999 (since counts are 0..999 inclusive).

- For $V_{avg} = V_0/4$: required duty = 0.25. Compare count for CCR1:

$$\text{TA0CCR1} \approx 0.25 \times 1000 = 250.$$

  So set TA0CCR1 = 250 (or 249 depending on +1 indexing). Using formula:

$$\text{TA0CCR1} = \text{round}\big(duty \cdot (\text{TA0CCR0} + 1)\big).$$

- For $V_{avg} = 3V_0/4$: duty = 0.75:

$$\text{TA0CCR1} \approx 0.75 \times 1000 = 750.$$

If you choose a different PWM frequency simply compute:

$$\text{TA0CCR0} = \frac{f_{timer}}{f_{PWM}} - 1, \qquad \text{TA0CCR1} = duty \cdot (\text{TA0CCR0} + 1).$$

If SMCLK is divided

If you enable timer input divider, use effective timer tick frequency $f_{timer} = \dfrac{SMCLK}{\text{divider}}$.

3.5 (c) Code to generate V0/4 and 3V0/4 using TA0CCR0 + TA0CCR1 (PWM hardware)

Below are two example codes. Both assume SMCLK=1 MHz, Up mode, and PWM on TA0.1 pin (output mapped to the appropriate MSP430 pin). Adjust pin selections to your board's TA0.1 physical pin.

(i) PWM code: configure CCR0 for 1 kHz and CCR1 for duty

```c
/* PWM using Timer_A0: TA0CCR0 = period, TA0CCR1 = duty
   Example: SMCLK = 1 MHz, PWM = 1 kHz
   TA0.1 outputs PWM (use the proper port mapping for TA0.1 on
      your board)
*/
#include <msp430.h>

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

    // Example: assume TA0.1 is mapped to P1.2 (check device-
       specific pinout)
    // Set peripheral function for that pin (adjust to actual pin
       )
    P1SEL0 |= BIT2;  // example; change depending on board
    P1SEL1 &= ~BIT2;

    // TimerA0: SMCLK, Up mode, clear TAR
    TA0CTL = TASSEL__SMCLK | MC__UP | TACLR;

    // Choose PWM frequency 1 kHz
    TA0CCR0 = 999;            // Period = 1000 ticks (0..999)
    TA0CCTL1 = OUTMOD_7;      // Reset/Set output mode for CCR1

    // For V0/4 (25%):
    TA0CCR1 = 250;  // 250/1000 -> 25%

    // To get 3V0/4 (75%) change TA0CCR1=750

    __bis_SR_register(GIE);
    while(1) {
        __no_operation();
    }
}
```

(ii) Using CCR0-only + software toggling (not recommended for precise duty)

You could use CCR0 with interrupts to toggle the output to create arbitrary duty but jitter and CPU load increase. Example outline:

- Set timer in continuous mode.
- Use TA0CCR0 interrupt: on compare 1 toggle pin, reprogram TA0CCR0 to next toggle time.

(This approach is shown below in another example where both CCR0 and CCR1 are used for manual set/reset.)

3.6 (d) Use both TA0CCR0 and TA0CCR1 to create V0/4 — alternative hardware technique

One method: configure Timer in continuous mode and use CCR0 and CCR1 compares to **set and reset** output using software (or output modes). But best is using OUTMOD on CCR1 with CCR0 as period. If the requirement specifically wants both registers used to define edges (e.g. CCR0 used to reset and CCR1 used to set), an approach is:

- Set TA0 in continuous mode (counts up to 0xFFFF) and use CCR0 and CCR1 compare interrupts to set/reset a GPIO (or to control TA0.1 OUT with appropriate OUTMOD and toggles).
- For V0/4 with period counts = 1000 ticks, arrange toggles at times: t=0 set high (use CCR1), t=250 reset (use CCR0 at absolute time 250) then next cycle wrap at 1000 etc.

A simpler hardware solution: Up mode with TA0CCR0 = 999 (period), TA0CCR1 = 250 (duty) and TA0CCTL1 = OUTMOD_7 is sufficient. But requested example using both CCR0 and CCR1 (explicit) is shown in code below: CCR0 is used as period (TA0CCR0) and CCR1 for duty (TA0CCR1) and using OUTMODs — this is functionally identical to (c) but explicitly references both registers.

```c
/* Alternative: use CCR0 as period and CCR1 as compare; CCR0 and
   CCR1 both active.
   Set CCR0=999, CCR1=250 for 25% duty. Uses hardware OUTMOD.
*/
#include <msp430.h>
void main(void) {
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;
    // Map TA0.1 to port pin (adjust to your board)
    P1SEL0 |= BIT2; P1SEL1 &= ~BIT2;

    TA0CTL = TASSEL__SMCLK | MC__UP | TACLR;
    TA0CCR0 = 999;       // period
    TA0CCTL1 = OUTMOD_7; // reset/set on CCR1
    TA0CCR1 = 250;       // duty 25%
    while(1) __no_operation();
}
```

Summary: Use CCR0 for period and CCR1 for duty is standard and recommended. Setting CCR1 to 250 gives V0/4 (25%) if VHIGH = V0.

Chapter 4

3. Frequency generation and measurement

4.1 Batch tasks: generate and measure

- **Batch-1** generate signal at frequency x kHz using Timer compare/PWM.
- **Batch-2** measure the frequency generated by Batch-1 using Timer capture mode.
- Then switch tasks and repeat.

4.1.1 Measurement approach (reciprocal capture)

Assume input signal is connected to capture pin TA0.CCI2A (mapped to P1.2 in your setup). Configure TA0 in continuous mode with SMCLK ticks. Capture on rising edge twice, compute difference (handle wraps), and compute frequency as:

$$f = \frac{f_{timer}}{\Delta}$$

where $f_{timer} = 1\,\text{MHz}$.

4.1.2 Code: generate a test PWM and measure it via capture (stores frequency in variable)

This single program toggles between generator and measurement modes (comment/uncomment as needed). Below is the measurement code that stores measured frequency in the variable `measured_freq`.

```c
/* Frequency measurement using Timer_A capture.
   Assumes input signal connected to TA0.CCI2A (P1.2) or adjust
      pin mapping.
   SMCLK = 1MHz used as timer clock.
*/

#include <msp430.h>
volatile unsigned int cap1=0, cap2=0;
volatile unsigned char captured=0;
volatile float measured_freq=0.0;
volatile unsigned int overflow_count = 0;

// Use TIMER0_A1_VECTOR for CCR1/CCR2 and overflow (TA0IV)
#pragma vector = TIMER0_A1_VECTOR
__interrupt void Timer0_A1 (void)
{
    switch (TA0IV) {
        case TA0IV_TACCR1: break;
        case TA0IV_TACCR2:
            if (captured==0) {
                cap1 = TA0CCR2;
                captured = 1;
            } else {
                cap2 = TA0CCR2;
                // compute delta with timer wrap handling (timer in continuous)
                unsigned long delta;
                if (cap2 >= cap1) delta = (unsigned long)(cap2 - cap1);
                else delta = (unsigned long)(0x10000u + cap2 - cap1); // wrap (16-bit)
                // If using divider or overflow counting, include that.
                // Timer tick rate:
                const float f_timer = 1000000.0f; // 1 MHz
                if (delta>0) measured_freq = f_timer / (float)delta;
                else measured_freq = 0.0f;
                // prepare for next measurement
                cap1 = cap2;
                captured = 1; // let next capture overwrite cap2
            }
            break;
        case TA0IV_TACCR0: break;
        default: break;
    }
}

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

    // Configure P1.2 as TA0.CCI2A input (if device pin mapping requires P1SEL)
    P1SEL0 |= BIT2; P1SEL1 &= ~BIT2;

    // TimerA in continuous mode so capture times increase monotonically
    TA0CTL = TASSEL__SMCLK | MC__CONTINUOUS | TACLR;

    // Capture on rising edge, CCI2A, enable interrupt
    TA0CCTL2 = CM_1 | CCIS_0 | CAP | CCIE; // Capture rising
       edges only

    __bis_SR_register(GIE);

    while(1) {
        // measured_freq contains the latest frequency in Hz
        __no_operation();
        // you can use measured_freq in app code or send via UART
    }
}
```

4.2 Accuracy and how to improve it

- Sources of error:
  - Timer clock stability (if DCO is 1 MHz but not calibrated it may vary).
  - Resolution limited by timer tick period (1 μs for 1 MHz).
  - Jitter triggers (noise on input edges).
  - Quantization: when measuring high frequencies the delta can be small (few ticks) so percentage error increases.
- Improvements:
  - Use a higher-stability clock source (external crystal) or increase SMCLK frequency.
  - Average multiple period measurements (take N measurements and average).
  - Use capture of multiple cycles (count multiple periods N and measure combined delta; frequency=f_timer/(delta/N) reduces quantization).
  - Use input conditioning (Schmitt trigger, buffering) to reduce jitter and false triggers.
  - Use prescalers carefully so delta fits range and resolution is adequate.
- Programmatic improvement example: measure time for 100 cycles and divide:

$$f \approx \frac{f_{timer} \times 100}{\Delta_{100cycles}}$$

  That reduces relative quantization error by factor 100.

Chapter 5

4. Generate an aperiodic waveform of x cycles then zero

5.1 Approach

Generate x cycles of square wave and then stop the timer output. Implementation idea:

- Use PWM in hardware and count the number of period interrupts (TAx CCR0 interrupts) in an ISR.
- After x periods, disable the timer output or set output pin to 0.

5.1.1 Code: generate x cycles

```c
/* Generate x cycles of square PWM and then stop output */
#include <msp430.h>
volatile unsigned int cycle_count = 0;
const unsigned int X = 50; // choose x cycles here

#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer0_A0_ISR(void)
{
    cycle_count++;
    if (cycle_count >= X) {
        // turn off PWM output: clear OUTMOD or set output low
        TA0CCTL1 &= ~OUTMOD_7; // disable reset/set
        // Optional: set pin low as GPIO
        // P1OUT &= ~BIT2;
        // P1DIR |= BIT2; // make it GPIO
        TA0CTL &= ~MC__UP; // stop timer
    }
}

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;
    // Setup PWM similar to earlier example
    P1SEL0 |= BIT2; P1SEL1 &= ~BIT2; // map TA0.1
    TA0CTL = TASSEL__SMCLK | MC__UP | TACLR;
    TA0CCR0 = 999; TA0CCTL1 = OUTMOD_7; TA0CCR1 = 500; // 50%
       example
    // TA0CCR0 interrupt enable for counting cycles
    TA0CCTL0 |= CCIE;
    __bis_SR_register(GIE);
    while(1) __no_operation();
}
```

5.2 Minimum number of cycles x for DMM to recognize frequency

- There is no single universal number — it depends on the DMM's measurement integration time and algorithm.
- Practical guidance:
  - Many DMMs sample over a time window (e.g., 100 ms to 1 s). If your burst duration is much shorter than the DMM's integration window, the meter may show zero or an inaccurate reading.
  - To be safe, ensure the burst duration is at least ~100–500 ms so DMM can stabilize and display frequency. For high frequencies this means many cycles; for low frequencies, fewer cycles might suffice.
  - As a heuristic: generate at least **50–100** cycles or make sure the burst lasts at least 0.1 s. This usually lets the DMM produce a stable reading.
- Conclusion: pick X so that $X/f \geq 0.1$ s where f is waveform frequency.

Chapter 6

5. Stopwatch (pushbutton start/stop using capture) — full code

This uses the hardware configuration you specified:

- Pushbutton: P2.7 (user switch).
- Jumper P2.7 → P1.2 (TA0.CCI2A) so pushing the button provides capture input.
- LED1 (red): P1.0 ON when timer running (start).
- LED2 (green): P6.6 ON when timer stops (stop).

```
/* Stopwatch using pushbutton P2.7 connected to P1.2 (TA0.CCI2A)
   Light P1.0 (red) when running, P6.6 (green) when stopped
*/
#include <msp430.h>
volatile unsigned int start_time = 0;
volatile unsigned int stop_time = 0;
volatile unsigned long elapsed_us = 0;
volatile unsigned char running = 0;

#pragma vector = TIMER0_A1_VECTOR
__interrupt void Timer0_A1_ISR(void)
{
    if (TA0IV == TA0IV_TACCR2) {
        // Capture occurred on TA0CCR2
        // Read button state via P2IN to distinguish press vs
           release
        if (P2IN & BIT7) {
            // Button is HIGH: treat as not pressed (depends on
               switch wiring)
            // If your hardware has active-low button, invert
               logic accordingly.
            // Here we treat rising edge as button release.
            stop_time = TA0CCR2;
            // compute elapsed (handle wrap)
            if (stop_time >= start_time) elapsed_us = stop_time -
                start_time;
```

```c
            else elapsed_us = (unsigned long)(0x10000u +
               stop_time - start_time);
            running = 0;
            P6OUT |= BIT6;  // green ON (stopped)
            P1OUT &= ~BIT0; // red OFF
        } else {
            // Button pressed (falling edge)
            start_time = TA0CCR2;
            running = 1;
            P1OUT |= BIT0;  // red ON (running)
            P6OUT &= ~BIT6; // green OFF
        }
    }
}

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

    // Configure LEDs
    P1DIR |= BIT0; P1OUT &= ~BIT0; // Red LED (P1.0)
    P6DIR |= BIT6; P6OUT &= ~BIT6; // Green LED (P6.6)

    // Configure P1.2 as TA0.CCI2A input
    P1SEL0 |= BIT2; P1SEL1 &= ~BIT2;

    // Configure TimerA0 continuous mode
    TA0CTL = TASSEL__SMCLK | MC__CONTINUOUS | TACLR;

    // Capture on both edges, CCI2A, enable interrupt
    TA0CCTL2 = CM_3 | CCIS_0 | CAP | CCIE;

    // Configure P2.7 as button input (with pull-up or as per
       board)
    P2DIR &= ~BIT7;
    P2REN |= BIT7;
    P2OUT |= BIT7; // pull-up

    __bis_SR_register(GIE);
    while(1) {
        // elapsed_us holds time in timer ticks (1 tick = 1 us if
            SMCLK=1MHz)
        // convert to microseconds if needed and use elsewhere
        __no_operation();
    }
}
```

Notes:

- If your pushbutton is active-low (most on LaunchPad are), the code expects that pressing pulls the line low. Adjust logic if your board uses opposite wiring.

- `elapsed_us` is in ticks; with SMCLK=1 MHz each tick = 1 μs, so elapsed time in microseconds = `elapsed_us`. For milliseconds: `elapsed_us/1000`.

Chapter 7

Appendix: Practical tips and checklist

- Always verify actual SMCLK with a scope or by toggling a pin and measuring — the DCO may not be exactly 1 MHz.
- Inspect device datasheet for exact pin mapping of TA0 channels and configure port function (PSEL) accordingly.
- Use hardware PWM (OUTMOD) when possible; it is far more accurate and has less jitter than software toggling.
- Use input debouncing for mechanical buttons if you rely on button edges (or use capture on both edges but filter).
- For DC motor drive use a proper driver (MOSFET) and consider current limiting and thermal aspects.

Chapter 8

References

- MSP430FR2433 Family Datasheet and User Guide (use your course links).
- Lecture materials and code samples you provided.
- Electronics tutorials for 7-segment and LCD (as previously referenced).