CprE 288 – Homework Question Set 4 - SOLUTIONS

Question 1: ADC Successive Approximation

A 4-bit ADC (with M = 16 steps) uses the Successive Approximation implementation. The input voltage range of the ADC is 0 V – 20 V. If the input is 14 V, how does the ADC compute each bit of the digital encoding? Fill in the following table to show the steps (refer to examples in slides and the VYES book chapter on ADC). The first step is given.

Note: DN_Mid is a 4-bit digital value in binary. AV-Mid is a real value in decimal.

Step	Output	DN_Mid	AV_Mid(V)	Input >= AV_Mid?
0	xxxx	1000	10	1 (yes, 14 > 10)
1	1xxx	1100	[10..20] 15 V	0 (no, 14 < 15)
2	10xx	1010	[10..15] 12.5 V	1 (yes, 14 > 12.5)
3	101x	1011	[12.5..15] 13.75	1 (yes, 14 > 13.75)
4	1011			

The final digital value is (in binary and decimal): 0b1011 = 11 (decimal)

Note that the result can be double-checked in several ways using resolution, bit weights and/or proportional ratios. For example, the resolution is 20 V / 16 steps = 1.25 V per step. Thus the bit weights are, starting at bit 0: 1.25 V, 2.5 V, 5 V, 10 V (bit 3). Thus 0b1011 = 10 V + 2.5 V + 1.25 V = 13.75 V. The next digital output, 0b1100 = 10 V + 5 V = 15 V, which is greater than the input voltage. Successive approximation uses the lower digital output.

Alternatively, for an input of 14 V, this ratio holds: 14 V / 20 V = x / 16 → x = 11.2, and this digital result would truncate to 11.

Question 2: GPTM Timers and Input Capture

Timer 1B is being used to measure the time between events on an input. It is configured to count up and detect falling edges. It uses the system clock of 16 MHz.

a. Consider the following input signal connected to a Timer 1B CCP input pin. Event E1 is the first falling edge. Event E2 is the second falling edge.

Suppose the 16-bit timer count values are: E1 = 15,080 and E2 = 56,250. What is the elapsed time in seconds between events E1 and E2?

Answer: **2.57 ms = (56250-15080)*62.5 ns**

b. Write a line of code that configures the timer (Timer 1B) to detect falling edges.

TIMER1_CTL_R = 0x400

c. Assume that capture mode event interrupts have been enabled using GPTM timer interrupt registers for Timer 1B. Event interrupts also need to be enabled in the interrupt controller. Determine **j** and **m** (see the register names) for the names of the NVIC enable and priority registers to be configured.

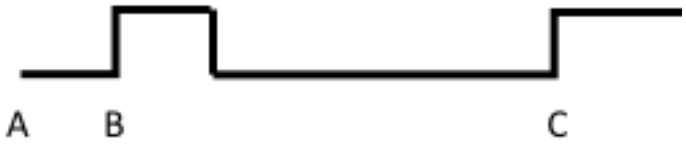
```
NVIC_EN j_R ... //assume appropriate assignment
NVIC_PRI m_R ... //assume appropriate assignment
```

What number is j? **0**

What number is m? **5**

Question 3: GPTM Timers and Input Capture

a. Consider the following input signal connected to the TM4C123 microcontroller Timer 3 input capture hardware via a GPIO pin. The TM4C123 system clock is 16 MHz.



At point A, let Timer 3 = 0. At event B, let Timer 3 = 500. Note that these are the count values in the timer register in decimal. It is also given that the time period between events B and C rising edges is 2.5 ms.

Based on known information and the figure, what will the value of Timer 3 be when event C occurs, i.e., the second rising-edge event? Write your answer in hex. Show your work.

The timer increments every system clock cycle, or every 62.5 ns (for 16 MHz system clock).

We know the count value for B. We can calculate the count value from B to C (let it be K). Then the count value for C is B+K.

K = number of count values from B to C = 2.5 ms / 62.5 ns (per count value) = 40,000 count values

Timer value at event C = count value for C = count value for B + K = 500 + 40,000 = 40,500 = 0x9E34

b. Suppose Timer 2B is being used in edge-time mode with an interrupt priority of 3. It uses IRQ (Interrupt ReQuest) number 24. Complete the code to initialize interrupts for the timer.

```
Config_Timer2()
{
  // Assume the associated GPIO module has been initialized.
  // Assume Timer 2B has been configured for 16-bit, count up,
  // edge-time, capture mode to detect positive edges.

  // Set up Timer 2 interrupts
  // Clear event capture interrupt status
  Timer B capture mode event is bit 10 in these registers. It's helpful
to notice when registers have similar bits/bit fields.

    TIMER2_ICR_R |= 0b10000000000; // 0x400

  // Enable event capture interrupts

    TIMER2_IMR_R |= 0b10000000000); // 0x400

  // Set up NVIC for Timer 2B interrupts
  // Put the correct PRI and EN register numbers in the blank.

    NVIC_PRI_6__R = (NVIC_PRI6_R & ~0xF0) | 0x60;
      //clear first field then assign priority 3 = 0b011 (0b0110=0x6)
     //other & masks include 0xFFFFFF0F, 0xFFFFFF1F

    NVIC_EN_0__R |= 0x1000000;
```

```
 // Assume initialization finishes by setting up the interrupt
 // handler, re-enabling timer, and globally enabling interrupts
}
```

For the NVIC registers, in addition to the datasheet, the Bai book tables provided in lecture slides, lab, etc. are helpful. These tables show the layout of the NVIC enable and priority registers.

Timer2B is IRQ 24 and this determines the position of its priority field in the PRI register set and its enable bit in the EN register set.

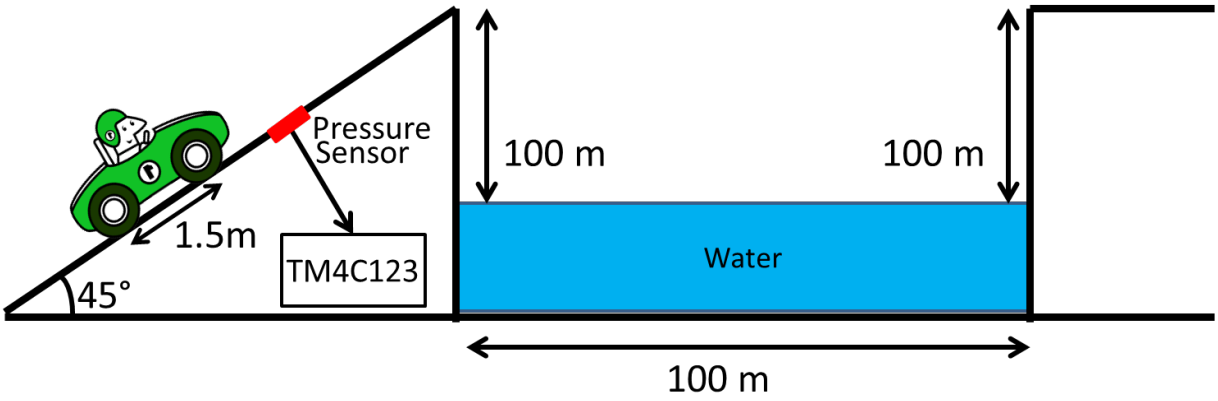
The enable bit is in bit position 24, where EN registers have 32 bits, thus in EN0 register, it is bit 24.

The 3-bit priority field is the 24th field, where PRI registers have 4 fields per register (simply how they designed them), thus in PRI6 register, it is the first field (which is the 3-bit priority value in bits 5..7).

The priority value is given in the problem statement, i.e., 3, which as 3 bits is 0b011.

Question 4: Timer Input Capture Programming

A pressure sensor is embedded in the ramp below.



When pressure is applied to the sensor, a logic 1 is driven on the Timer 1A Input Capture input pin of the TM4C123, else a logic 0 is driven. You are to write a C program that will control the car's brakes to stop the car if its velocity is not fast enough to jump the gap shown. This program uses the Timer 1A Input Capture interrupt.

Assumptions:
1) The car is moving at a constant velocity while on the ramp.

2) The car is treated as a “point mass” for computing the physics of the problem. This results in a minimum velocity of about 31.2 m/s needed to jump the gap. (You may want to try to calculate this on your own just for fun. These resources might help: https://www.khanacademy.org/science/physics/two-dimensional-motion/two-dimensional-projectile-mot/a/what-are-velocity-components , http://www.physicsclassroom.com/class/vectors/Lesson-2/Initial-Velocity-Components .)

a. Initialize Timer 1A as a 24-bit timer, to detect positive (rising) edge events, and with the input capture interrupt enabled.

i. Select an appropriate TM4C GPIO pin to use for the input pulse from the pressure sensor. What GPIO port and pin did you select? Hint: Input capture pins are denoted by TnCCPm labels in the Alternate Function list in Table 23-5 in the datasheet, where n=1 for Timer 1 and m=0 for timer A.

To use Timer 1A, we need to find T1CCP0 pins. There are two options, PB4 and PF2. PB4 will be used in this solution.

PB4	58	AIN10	-	SSI2Clk	-	M0PWM2	-	-	T1CCP0	CAN0Rx	-	-	-
PF2	30	-	-	SSI1Clk	-	M0FAULT0	M1PWM6	-	T1CCP0	-	-	TRD0	-

ii. Briefly describe the initialization tasks at a high level (at a higher level than C code or comments). What features need to be initialized and for what purpose? Do not provide specific register macros, bitwise operations, or code.

Enable and initialize a GPIO port to use an input pin for input capture with a timer.
Enable a general-purpose timer module and configure a timer for input capture mode to detect the positive edges from the pressure sensor.
Enable interrupts at all levels for input capture.

b. Complete the initialization code below.

```
Config_Timer1A()
{
  // 1. Set up GPIO
  // A) Configure GPIO module associated with Timer 1A
    // i. Turn on clocks for GPIO Port B and Timer 1
    // Note: Timer 1A can use Port B or F, this code uses Port B
    // Note: Port F would use different pins of its port.
    SYSCTL_RCGCGPIO_R |= 0b000010; // 0x02
    while ((SYSCTL_PRGPIO_R & 0x2) {};
    SYSCTL_RCGCTIMER_R |= 0b000010; // 0x02
    while ((SYSCTL_PRTIMER_R & 0x2) {};

    // ii. Enable alternate function and set peripheral functionality
    GPIO_PORTB_AFSEL_R |= 0b00010000; // Timer1A input is PB4 (pin 4)
    GPIO_PORTB_PCTL_R = (GPIO_PORTB_PCTL_R & ~0xF0000) | 0x00070000;
      // Use function T1CCP0 from column 7 of alternate function table
```

```
    // iii. Set digital mode
    GPIO_PORTB_DEN_R |= 0b00010000; // Set pin 4 digital mode

  // 2. Set up Timer 1A
  // A) Configure Timer 1 mode
    //Disable Timer 1A device while we set it up
    TIMER1_CTL_R &= ~0x1;

    // Configure Timer 1A functionality
    TIMER1_CFG_R = 0x4; // Set to 16-bit mode
    TIMER1_TAMR_R = 0b10111; //Count up, edge-time, capture mode
    TIMER1_CTL_R = 0b0000; // Detect positive edges (3:2=00)
    // these TAMR and CTL register assignments are not preserving bits

    TIMER1_TAPR_R = 0xFF; // Use prescaler extension to 24 bits
    TIMER1_TAILR_R = 0xFFFF // Load max 24-bit value

  // B) Set up Timer 1A interrupts
    TIMER1_ICR_R |= 0b00000100; // Clear capture event interrupt status
    TIMER1_IM_R  |= 0b00000100; // Enable capture event interrupts

  // 3. NVIC setup
  // A) Configure NVIC to allow Timer 1A interrupts (use priority=1)
    NVIC_EN0_R |= 0x00200000; //Enable Timer1A IRQ #21, bit 21
    NVIC_PRI5_R = (NVIC_PRI5_R & ~0xF000) | 0x00002000;
      //Timer1A IRQ priority=1, bits 15-13 = 0b001 (0b0010 = 0x2)

 //  B) Bind Timer 1A interrupt requests to user's interrupt handler
    IntRegister( 37, TIMER1A_Handler); // use interrupt vector number

 // Re-enable Timer 1A
    TIMER1_CTL_R |= 0x1;

 // Globally enable CPU to service interrupts
    IntMasterEnable();
}
```

c. Suppose you are given the following main program and Stop_car function. Answer questions i. and ii.

```
// Global variables (additional variables may be used)
volatile unsigned int first_wheel_hit;  // 1st wheel hits sensor
volatile unsigned int second_wheel_hit; // 2nd wheel hits sensor
volatile int done_flag=0; // 1 after both first and
                          // second_wheel_hit have been stored
// Program to stop car if it is not fast enough to jump the gap.
main()
{
  int stop = 0;
  Config_Timer1A();
  while(1)
  {
```

```
    // Wait for sensor input events to be captured
    while(!done_flag)
    {
    }
    done_flag = 0;  // Clear flag
    // Check if car needs to stop
    stop = Stop_car();
    if(stop)
    {
      lcd_printf("Stopping car!");
    }
    else
    {
      lcd_printf("Car going to jump!!");
    }
  } // end while
}

// Return 0 if the car is fast enough to jump the gap
int Stop_car(void)
{
  float velocity_car;
  float time;

  // Compute car velocity based on captured times and distance
  // 16 MHz clock and counter tick rate => .0625us tick period
  time= (second_wheel_hit - first_wheel_hit) * .0625 // in usec
  time= time/1000000; // in seconds
  velocity_car = 1.5/time;  // in meter/s
  if(velocity_car < 31.2)
  {
    return 1;  // Not fast enough, stop
  }
  else
  {
    return 0;  // Fast enough, make jump
  }
}
```

i. Finish the Interrupt Service Routine code.

```
// Store timer values in first_wheel_hit and second_wheel_hit
void TIMER1A_Handler(void)
{
  static int state = 0; // 0/1: before/after first wheel hit

  // Check if an input edge-time capture interrupt occurred
  // Test the interrupt status bit
  if ((TIMER1_MIS_R & 0b00000100)
  {
    // Clear the interrupt status bit by writing 1 to ICR bit
    TIMER1_ICR_R |= 0b00000100;
```

```
    // Capture rising edge time for when 1st wheel hits sensor
    if(state == 0)
    {
      first_wheel_hit = TIMER1_TAR_R;
      state = 1;
    }
    else // Capture rising edge time for when 2nd wheel hits sensor
    {
      second_wheel_hit = TIMER1_TAR_R;
      done_flag = 1; //tell main done capturing 1st & 2nd wheel times
      state = 0;
    }
  }
}
```

ii. Is it necessary for the `Stop_car()` function to check for timer overflow? Briefly explain why or why not?

Currently, the function does not check for timer overflow. It assumes that the timer value for second_wheel_hit is greater than the value for first_wheel_hit, resulting in positive values for time and velocity. If there is overflow, the timer resets to 0 and starts over between the first and second wheels hitting the sensor. Then the second_wheel_hit has a smaller value than the first, and the computed values will be negative. The function will return 1, stopping the car. It would be better to check and correct for overflow rather than stop the car.

Question 5: GPTM Timers and PWM

Suppose GPTM Timer 5A is configured in PWM mode. The system clock is 16 MHz. A variable for the match value is initialized as follows.

```
unsigned long match_value = 600,000;
```

a. Counting down to 0 from the match value, what time in seconds would elapse for a `match_value` of 600,000?

600000* 62.5 ns = 37.5 ms

b. What hex values would be put in these match registers during initialization?

600000 = 0x927C0 →

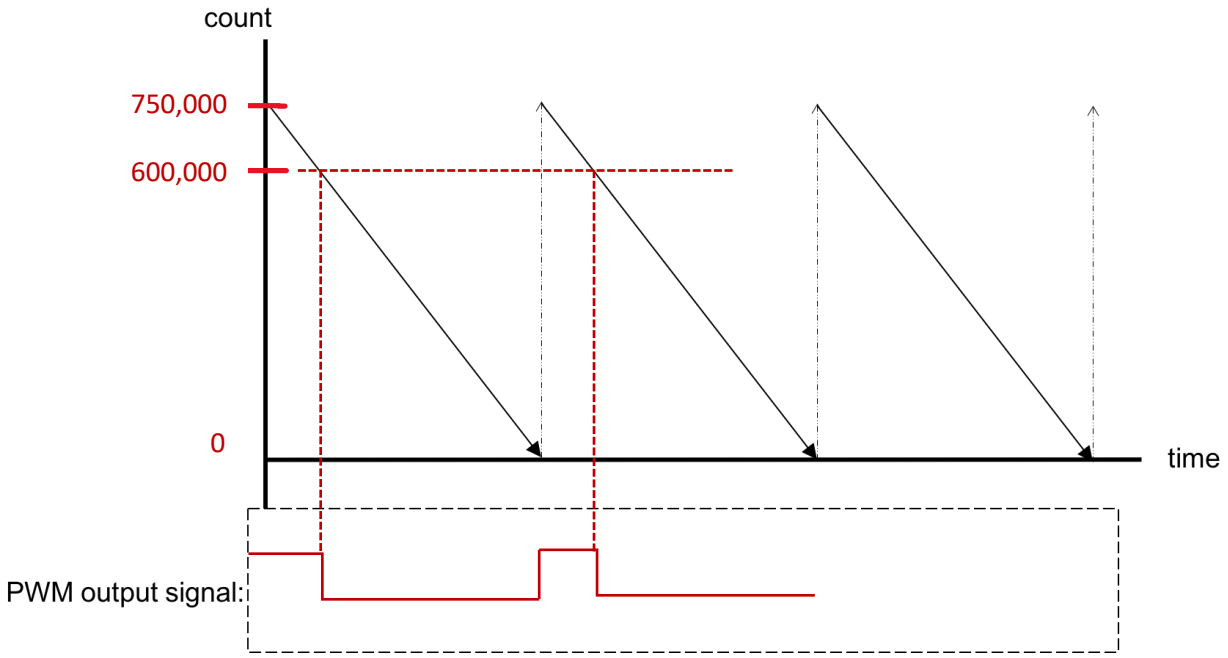
GPTMTAPMR	GPTMTAMATCHR
0x **9**	0x **27C0**

c. For a start value of 750,000 and the match value of 600,000, what is the duty cycle of the PWM output signal? Note: the PWM output is high at the start (not inverted).

high = period – low = 750000 – 600000 = 150000

duty cycle = high/period = 150000/750000= 0.2 or 20%

d. Similar to Figure 11-5 in the datasheet, show the start and match values in the diagram below on the y (count) axis, and sketch several cycles of the PWM output waveform in the dashed rectangle below the diagram. Use the values given in part c.



Question 6: GPTM Timers and PWM

a. Suppose Timer 2 is configured as follows. Assume that the GPIO module has been initialized appropriately.

```
   //Timer 2 configuration

   void init_TIMER2()
   {
L1:    SYSCTL_RCGCTIMER_R |= 0b00000100;
L2:    while ((SYSCTL_PRTIMER_R & 0b00000100) {};
L3:    TIMER2_CTL_R &= ~0x1; // Timer 2 Control
L4:    TIMER2_CFG_R = 0x4;  // Timer 2 Configuration
L5:    TIMER2_TAMR_R = 0xA; // Timer 2A Mode
L6:    TIMER2_TAPR_R = 0;   // Timer 2A Prescaler
L7:    TIMER2_TAILR_R = 32; // Timer 2A Interval Load
L8:    TIMER2_TAPMR_R = 0; // Timer 2A Prescaler Match
L9:    TIMER2_TAMATCHR_R = 8; // Timer 2A Match
L10:   TIMER2_CTL_R |= 0x1; // Timer 2 Control
   }
```

i) Describe the modes and configuration of the timer. Be as specific as possible.

These are specified using registers in lines L4 and L5. Look up registers and bit assignments.

The timer is in 16-bit, periodic, PWM, and count down modes.

ii) Which GPIO port and pin would be used with this configuration of the timer?

Timer 2A uses T2CCP0 pin which is associated with pins PB0 and PF4, and either pin could be used.

iii) What is the duty cycle of the output waveform?

duty cycle = high pulse time / PWM period, where high pulse time = PWM period – low pulse time
PWM period = start value = 24-bit PR:ILR register combination
low pulse time = match value = 24-bit PM:MATCH register combination
duty cycle = (PR:ILR – PM:MATCH) / PR:ILR = (32-8) / 32 = 24/32 = 75%

b. Suppose you want to generate a waveform having a period of 5 ms and high pulse width of 1 ms. What values should be assigned in lines L6 through L9? Calculate numbers for the values, and write the numbers in hex in the blanks.

period = 5 ms / 62.5 ns = 80,000 = 0x13880 (start value)
low pulse = period – high pulse = 4 ms / 62.5 ns = 64,000 = 0xFA00 (match value) (or 4/5 x 80,000)

```
L6:    TIMER2_TAPR_R =      0x01;
```

```
L7:    TIMER2_TAILR_R =     0x3880;

L8:    TIMER2_TAPMR_R =     0x0;

L9:    TIMER2_TAMATCHR_R =  0xFA00;
```

Question 7: PWM Programming

Generate a square wave (50% duty cycle) with a 12 ms period. Use GPTM Timer 1B in PWM mode. Assume the associated GPIO module has already been configured using another function. Your function should initialize and enable the timer.

```
void init_TIMER1()
{
    // Assume associated GPIO module already configured

    SYSCTL_RCGCTIMER_R |= 0b00000010; // Enable Timer 1's clock
    while ((SYSCTL_PRTIMER_R & 0b00000010) {};

    //Disable Timer 1B device while being configured
    TIMER1_CTL_R &= ~0x0100; //or &= 0xFEFF, clear TBEN

    // Set Timer 1 functionality
    TIMER1_CFG_R = 0x4; // Set GPTMCFG field to 16-bit mode
    TIMER1_TBMR_R = 0b00001010; // PWM, Periodic Mode

    // PWM period = 12 ms, 16MHz clock, 62.5 ns clock period
    // PWM period = 12 ms / 62.5 ns = 192,000 cycles = 0x2EE00
    // 24-bit timer needed
    TIMER1_TBPR_R = 0x02; // Upper 8 bits
    TIMER1_TBILR_R = 0xEE00; // Lower 16 bits

    // Configure 50% duty cycle using match registers
    // Both high and low pulse = 0.5 * 0x2EE00 = 0x17700
    TIMER1_TBPMR_R = 0x01; // Upper 8 bits
    TIMER1_TBMATCHR_R = 0x7700; // Lower 16 bits

    //Re-enable Timer 1B device
    TIMER1_CTL_R |= 0x0100;  // Set TBEN
}
```

Question 8: Timer Initialization

Suppose it's your job to design a microcontroller-based system that detects edges on an input signal waveform, calculates the period of the input signal, keeps a running average of the period, and generates an output signal having the same (average) period. The output signal should be a pulse train with a constant high pulse of 500 microseconds. The output signal period should vary with the average period of the input signal.

You have started your design and selected GPTM Timer 2 in 16-bit mode. You will use both the A and B timers from Timer 2. Timer 2A will detect rising edges of the input signal waveform (input CCP pin), and Timer 2B will generate the PWM output signal (output CCP pin). The CCP pins are alternate functions of GPIO pins.

a. Find the GPIO port(s) for the CCP pins for Timer 2. Write code that configures the GPIOAFSEL register so that CCP pins are set up as alternate functions. Don't worry about other GPIO registers; just initialize the alternate function of the pins for CCP. Preserve other bits.

Timer 2A is alternate function T2CCP0 → PB0
Timer 2B is alternate function T2CCP1 → PB1

```
GPIO_PORTB_AFSEL_R |= 0b11; // 0x3
```

b. Next, configure Timer 2 (and A and B) as needed using the following registers: RCGCTIMER, PRTIMER, GPTMCFG, GPTMTnMR, GPTMCTL. Don't worry about other registers that may need to be initialized. Write initialization code for this subset of registers only. Make assumptions as needed.

```
SYSCTL_RCGCTIMER_R |= 0b00000100; // enable Timer 2 clock
while ((SYSCTL_PRTIMER_R & 0b00000100) {};
// disable both Timer A and Timer B (bits 0 and 8)
TIMER2_CTL_R &= ~0x101; // mask variations okay ~0b100000001 or 0xFEFE
TIMER2_CFG_R = 0x4; // 16-bit configuration
// Timer A: input capture, up or down okay (0x17 or 0x7)
TIMER2_TAMR_R = 0b10111; // capture mode, edge-time mode, count up
// Timer B: PWM mode
TIMER2_TBMR_R = 0b1010; // PWM mode, periodic mode (or 0xA)
TIMER2_CTL_R &= ~0b1100; // positive edges on Timer A, [3:2]=00
// enable both Timer A and Timer B (bits 0 and 8)
TIMER2_CTL_R |= 0x101;
```

Question 9: General-Purpose Timers (GPTM)

Note: Student answers can be shorter. More information is given here for clarity.

a. Briefly describe each of the timer modes given in Table 9.1 of the Bai textbook.

One-shot mode: The timer runs for some number of clock ticks and stops. If in count down mode, the counter starts from the loaded value and stops when it reaches 0. In count up mode, the counter starts from 0 and stops when it reaches the loaded value. In count down mode, the prescaler is used as a proper prescaler, whereas in count up mode it extends a 16-bit counter to 24 bits.

Periodic mode: Same behavior as in one-shot mode, except the timer does not stop. It continues to repeat a counting cycle. If in count up mode, when it reaches its max value (loaded value), it goes back to 0 and continues counting up. If in count down mode, when it reaches 0 it goes back to its max value (loaded value) and continues counting down.

RTC mode: This is Real-Time Clock mode. The counter is configured with a specific clock rate so that each tick of the timer equals 1 second. It requires a 32.768 KHz clock as input (1 second is 32768 clock cycles). Section 9.2.3.4 of the textbook explains that the clock is divided down to a 1 Hz rate and passed to the input of the counter (timer in RTC mode).

Edge Count mode: This is a type of input capture mode in which the timer recognizes and counts edge events on its CCP input. It compares the current count with a match value and sets a status flag when they are equal. In this mode, the GPTMTAR register holds the count of the input events, and the GPTMTAV register holds the free-running timer value. (see Fig. 9.3)

Edge Time mode: This is a type of input capture mode in which the timer recognizes edge events on its CCP input and the time of an event is captured. When an input event is detected, the current timer counting value (GPTMTAV) is captured in the GPTMTAR register and can be read by the CPU. In this mode, the GPTMTAR register holds the time at which the input event occurred. The timer does not stop counting.

PWM mode: The timer is configured as a 24-bit count-down counter and generates a PWM output signal on its CCP output. The load value determines the period of the PWM waveform and the match value determines the high and low pulse widths (duty cycle).

b. For the GPTM Raw Interrupt Status Register (GPTMRIS), refer to information about bit 8, TBTORIS (Timer B Time-Out Raw Interrupt) time-out status flag. Under what condition(s) will the TBTORIS bit be set in this register? Hint: What does timeout mean?

Note: One purpose of this question is to have you look up something specific in the datasheet and/or other materials, read about it, look up terminology you may not be familiar with, etc.

Look up GPTMRIS in the datasheet (e.g., p. 748). You can also find information about the register on other datasheet pages and in the textbook. Read about the TBTORIS time-out bit (bit 8). It states that a value of 1 means that "Timer B has timed out. This interrupt is asserted when a one-shot or periodic mode timer reaches its count limit (0 or the valued loaded into GPTMTBILR, depending on the count direction)." You can also search for the term "timeout" to get more information. For example, see Table 11-11 in the datasheet.

Thus, this bit indicates that a timeout event has occurred for Timer B. This means the timer reached 0 (in count down mode), or its maximum value, as defined by the load value (in count up mode).

c. For the GPTM Timer Mode Register (GPTMTnMR), refer to information about bit 10, TnMRSU (Match Register Update). This bit defaults to 0, and could be set to 1 by a program. Describe and/or sketch how the value of the MRSU bit affects when the assignment to the match register takes effect and hence the timing of the PWM output pulse in relation to the free-running counter.

You used the GPTMTnMATCHR register in Lab 8 to generate an output pulse in PWM mode. Your program may have assigned a new match value when you wanted to change the pulse width and hence the position of the servo motor.

Hint: Use Figure 11-5 (datasheet) as an example. Suppose a new value is assigned to the match register shortly after the count-down cycle begins. This new value could be any value in the count range. If MRSU = 0, when does the new match value take effect? If MRSU = 1, when does the new match value take effect? How does this affect when the PWM output signal goes low?

Note: One purpose of this question is to have you look up something specific in the datasheet and/or other materials, read about it, consider concepts in new contexts, use something we understand (or should understand given our repeated use of it) like Figure 11-5 as a tool to interpret new information, etc.

The final answer:

When does the new match value take effect?
A) Immediately (MRSU=0)
B) Next timeout (MRSU=1)

For the PWM waveform:

A) PWM goes low when the match value is reached
B) PWM waits until timeout (next PWM period) to change the match value

The result for A) would vary depending on the old and new values of the match value and the current timer value. The result for B) is guaranteed to take effect during the next PWM period.

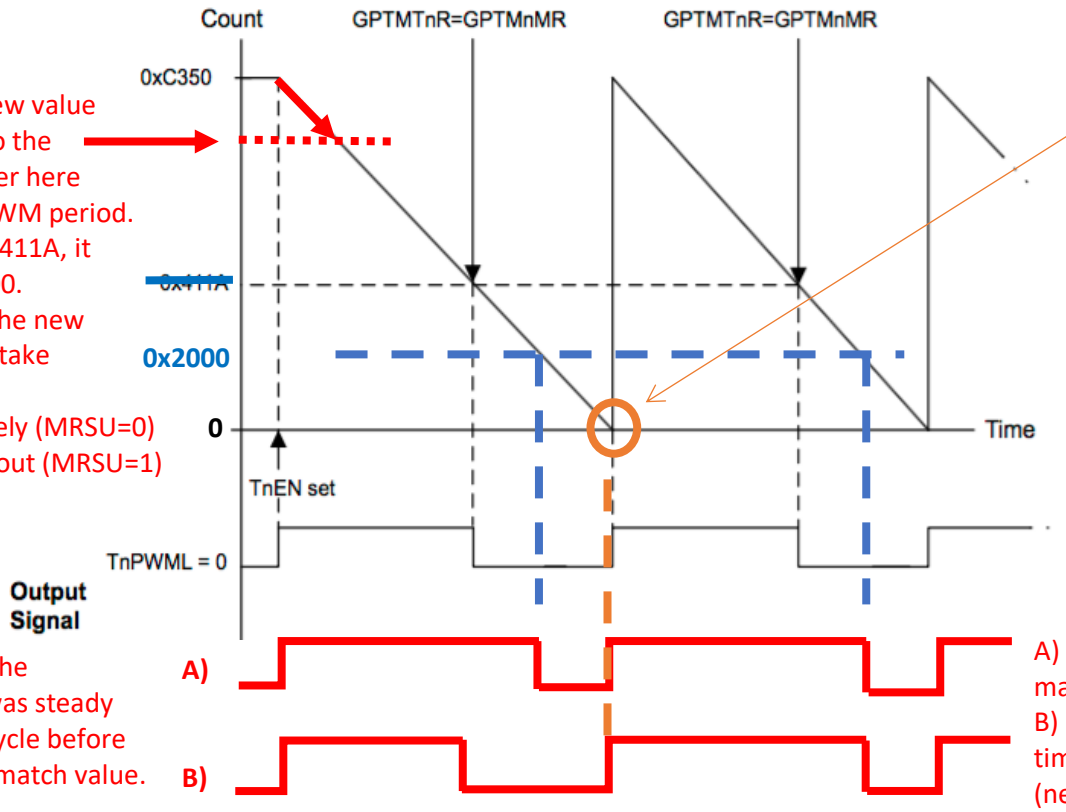
The work to get to that final answer:

Look up GPTMTAMR in the datasheet (e.g., p. 729). You can also find information about the register on other datasheet pages and in the textbook. Read about the TAMRSU match register update bit (bit 10). It states that a value of 0 means "Update the match register on the next cycle," and 1 means "Update the match register on the next timeout."

Thus, a new value is written to the match register either immediately on the next clock cycle, or later, when there is a timeout. A timeout happens in PWM mode when the timer counts down and reaches 0, thus at the end of the PWM period (before another PWM period begins). In the first case (MRSU=0), the PWM waveform could be modified in the middle of the current period. In the second case (MRSU=1), the current PWM period finishes before the duty cycle changes. So when MRSU=1, the PWM waveform is not modified in the middle of a PWM period.

Using an example from the datasheet (Figure 11-5), the effect could be sketched as follows.

Figure 11-5. 16-Bit PWM Mode Example



Suppose a new value is assigned to the match register here during the PWM period. Instead of 0x411A, it is now 0x2000. When does the new match value take effect?
A) Immediately (MRSU=0)
B) Next timeout (MRSU=1)

Assume that the PWM signal was steady at 66% duty cycle before changing the match value.

A) PWM goes low when match value is reached
B) PWM waits until timeout (next PWM period) to change the match value

The original figure is shown here for convenience:

Figure 11-5. 16-Bit PWM Mode Example

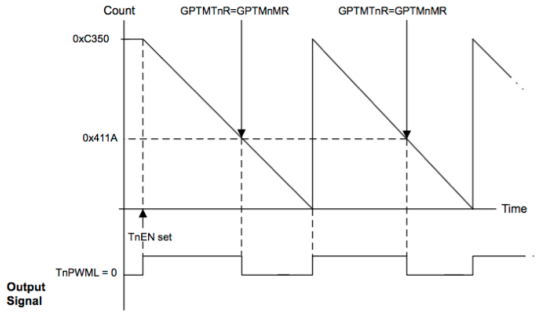


Figure 11-5 on page 717 shows how to generate an output PWM with a 1-ms period and a 66% duty cycle assuming a 50-MHz input clock and **TnPWML** =0 (duty cycle would be 33% for the **TnPWML** =1 configuration). For this example, the start value is **GPTMTnILR**=0xC350 and the match value is **GPTMTnMATCHR**=0x411A.

This example: 50 MHz clock
20 ns clock period
1 ms PWM period
1 ms / 20 ns = 50,000 cycles = 0xC350
66% duty cycle = 2/3 high, 1/3 low

Question 10: Software-Implemented Input Capture

There are various possible solutions. The key is to detect the rising edge, which means going from 0 to 1 on the GPIO input pin. So first you need to know the state of the pin is 0, and then you want to wait until it becomes 1, at which point you capture the timer value.

You have been using the input capture (edge time) mode of a GPTM timer. This means that input capture is implemented in the hardware of the timer module. You use the GPTM registers to set up and control the input capture hardware. What if the TM4C microcontroller did not have special input capture hardware? You would have to implement similar functionality in software.

a. Write a C program to save Timer 1A's count value (e.g., GPTMTAV) when a positive edge event occurs on GPIO Port D, pin 4 (PD4). It should store the timer value into variable `rising_edge_time`. Assume that the GPIO and GPTM modules have been properly configured in other code. Assume Timer 1A is configured and running in periodic mode.

```
int main(void)
{
  unsigned int rising_edge_time;

  // Assume GPIO Port D already configured properly
  // Assume GPTM Timer 1A already running in periodic mode
  while(1)
  {
// Wait until PD4 is at logic level 0
// Then wait for logic level 1, indicating positive edge event
    if( (GPIO_PORTD_DATA_R & 0b00010000) == 0)
    {
      while( !( GPIO_PORTD_DATA_R & 0b00010000)) // PD4 not 1
      {}
      rising_edge_time = TIMER1_TAV_R; // Store current time
    }
  }
}
```

b. Describe two disadvantages of the software-implemented input capture as compared to input capture mode of the timer hardware.

The CPU is busy-waiting on an input signal. During this time, the CPU is not executing other software tasks. In contrast, input capture mode of the timer monitors the input signal in hardware, independent of the CPU. The CPU is free to perform other tasks. This is less of an issue if your solution uses a GPIO interrupt to detect the rising edge, but it still takes more time and is less accurate.

It takes time to execute the polling and assignment code, and thus the value of rising_edge_time will have software-related error compared to the actual time of the event on the Port D pin. However, the input capture hardware detects an event and immediately captures the timer value.