Name: **Lab Section:**

CprE 288 Fall 2018 – Homework 7
Due Sunday. October 28 (on Canvas 11:59pm)

Notes:

- Homework must be typed and submitted as a PDF or Word Document (i.e. .doc or .docx) only.
- If collaborating with others, you must document who you collaborate with, and specify in what way you collaborated (see last page of homework assignment), review the homework policy section of the syllabus: http://class.ece.iastate.edu/cpre288/syllabus.asp for further details.
- Review University policy relating to the integrity of scholarship. See ("Academic Dishonesty"): http://catalog.iastate.edu/academic_conduct/#academicdishonestytext
- Late homework is accepted within two days from the due date. *Late penalty is 10% per day.* ***Except on Exam weeks****, homework only accepted 1 day late.*
- ***Note:*** *Code that will not compile is a typo. Answering a question as "will not compile"* ***will be marked incorrect****. Contact the Professor if you think you have found a typo.*
- **Note: You are not allowed to use any MACROs in your code, except for register names.**
  - Example: You will lose points for: GPIO_PORTA_DEN_R = GPIO_PORTA_DEN_R | PIN1
  - Must use: GPIO_PORTA_DEN_R = GPIO_PORTA_DEN_R | 0b0000_0010; // or 0x02

Note: Unless otherwise specified, all problems assume the TM4C123 is being used

Question 1: General Timer questions (10 pts)

a) Briefly describe each of the Timer modes given in Table 9.1 of the textbook. (6pts)

i) One shot mode: If in count down mode the counter starts from the loaded value and stops once 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 perscaler, in count up mode it extends a 16-bit counter to 24-bits

ii) Periodic mode: The same behavior as One shot mode, with the major difference being the timer does not stop. If in count up mode, then when it reaches its max value (i.e. loaded value) it goes back to 0 and continues counting. If in count down mode, then when it reaches 0 it goes back to its max value (i.e. loaded value) and continues counting

iii) RTC mode: The counter is configured with a specific clock rate, that makes each tick of the Timer 1 second.

iv) Edge Count mode: Allow counter each time an edge of an input signal occurs (i.e. count positive, negative, or both edges)

v) Edge Time mode: Allows the counter to capture the time at which the edge of an input signal occurs (i.e. capture the time of positive and/or negative edges of the input signal). Also called Input Capture.

vi) PWM mode: Allows one to generate a PWM wave efficiently.

Name: **Lab Section:**

b) For the GPTM Timer Mode Register (GPTMTnMR), under what Timer usage scenarios does it make sense to have the TAMRSU bit set to 0, how about set to 1? (2pts)

An example of where setting TAMRSU to 1 makes sense is when you are generating a PWM wave and you do not want a "glitch" to occur when changing the duty cycle of the PWM signal. Where glitch means having a pulse width that is unexpected. By not updating the Match Register until a timeout occurs, it guarantees the PWM waveform will not be modified in the middle of a PWM period.

An example of where setting TAMRSU to 0 makes sense is when you are generating a waveform using an interrupted-based Generic Waveform Generation approach. For this case, one wants to have the new match value immediately loaded. If instead the new value did not load until a timeout occurred, then you would not get the waveform you expect. You would get up to an extra timeout period added to the time you computed for when you wanted the next Match to occur.

c) Under what condition will the TBTORIS bit be set in the GPTM Raw Interrupt Status Register (GPTMRIS)? (2pts)

This bit indicates that a Time-out (or Time-up) event has occurred. This means the Timer has hit 0 (for counting down mode), or hit is maximum value (as defined by the load value and prescaler, for counting up mode).

Name: **Lab Section:**

Question 2: Don't Go into the Light! (25 pts)

Complete the program below to have a robot move away from a light source. The robot has two wheels, similar to the robot used in lab, and has a light sensor on each side. See figure.

Motor Control: Assume that you are programming Timer 1 module A (for left motor) and Timer1 module B (for right motor) to generate PWM waveforms to control the speed of each wheel's motor (**note, connect Timer 1 to Port B**). The speed of the motor is proportional to the percentage of time the PWM signal is high (i.e. PWM duty cycle).

Note: Your PWM wave must have a period of **1ms**. Note the system clock is 16 MHz.

Light Sensors: The light sensors are connected to **Channel 1** (left sensor) and **9** (right sensor) of the ADC as single channel inputs (i.e. not differential)

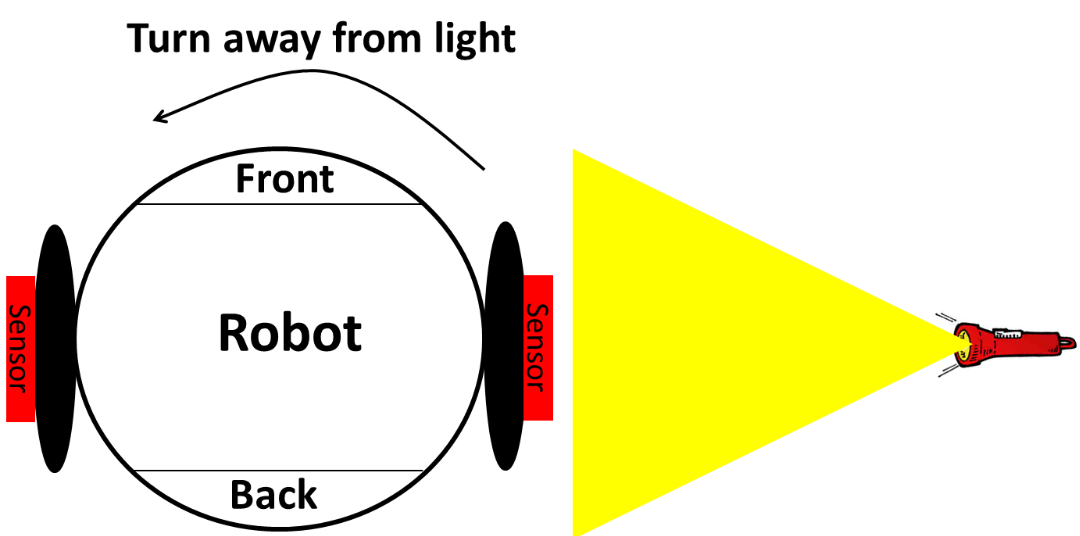
Robot behavior: The Robot should move away from the light in the following way. Where "Speed of motor" is the fraction of the motor's maximum speed.

- Speed of left motor = Intensity of left sensor / (Intensity of left sensor + Intensity of right sensor)
- Speed of right motor = Intensity of right sensor / (Intensity of left sensor + Intensity of right sensor)

Note: You are not allowed to use any MACROs in your code, except for register names.

- Example: You will lose points for: GPIO_PORTA_DEN_R = GPIO_PORTA_DEN_R | PIN1

- Must use: GPIO_PORTA_DEN_R = GPIO_PORTA_DEN_R | 0b0000_0010; // or 0x02



Name: **Lab Section:**

a. **Initialize TIMER 1 module A and B to meet the above requirements and so that both wheels initially move at 50% their maximum speed. (5 pts)**

```
 void init_TIMER1_A_B()
 {
// 1. Setup GPIO
// A) Configure GPIO module associated with Timer 1A and 1B

  // i. Turn on clock for GPIO Port B
  SYSCTL_RCGCGPIO_R = SYSCTL_RCGCGPIO_R | 0b000010; // 0x02

  // ii.Enable Alternate function and set Peripheral functionality
  GPIO_PORTB_AFSEL_R |= 0b0011_0000; // Timer 1A & 1B on bits 4 & 5
  GPIO_PORTB_PCTL_R |= 0x0077_0000; // use Timer for wire 4 & 5

  // iii. set digital or analog mode, and pin directions
  GPIO_PORTB_DEN_R |= 0b0011_0000; //enable pin 4 digital mode
  GPIO_PORTB_DIR_R &= 0b1100_1111; //set pin 4 to input

// 2. Setup Timer 1A and 1B
// A) Configure Timer 1 mode

  SYSCTL_RCGCTIMER_R |= 0b0000_0010; // Enable Timer 1's clock

  //Disable Timer 1A and 1B device while we set it up
  TIMER1_CTL_R &= ~(0x0101);

  // Set desired Timer 1 functionality
  TIMER1_CFG_R = 0x4; // Set to 16-bit mode
  TIMER1_TAMR_R = 0b0000_1010;//Timer 1A: PWM, Periodic Mode
  TIMER1_TBMR_R = 0b0000_1010;//Timer 1B: PWM, Periodic Mode

  TIMER1_CTL_R = 0b0000_0000; // Can assume defaults to all 0's

  TIMER1_TAPR_R = 0x00; //Timer 1A: prescaler not used
  TIMER1_TBPR_R = 0x00; //Timer 1B: prescaler not used

  TIMER1_TAILR_R = 16000; //Timer 1A: Set PWM period to 1ms
  TIMER1_TBILR_R = 16000; //Timer 1B: Set PWM period to 1ms
  TIMER1_TAMATCHR_R = 8000; // Timer 1A: 50% duty cycle
  TIMER1_TBMATCHR_R = 8000; // Timer 1B: 50% duty cycle
// B) Setup Timer 1 Interrupts
  NONE
// 3. NVIC setup
// A) Configure NVIC to allow Timer 1A and 1B interrupts
  NONE
// B)Bind Timer 1A/1B interrupt requests to User's Interrupt Handler
   NONE
//re-enable Timer 1A and 1B
TIMER1_CTL_R |= 0x0101;
}
```

Name: **Lab Section:**

b. **Initialize the ADC to meet the specification above. <u>No interrupts are to be used</u> (5 pts)**

```
   void init_ADC()
   {
// 1. Setup GPIO
  // A) Configure GPIO module associated with ADC

    // i. Turn on clock for GPIO Port E
    SYSCTL_RCGCGPIO_R = SYSCTL_RCGCGPIO_R | 0b01_0000; // 0x10

    // ii.Enable Alternate function and set Peripheral functionality
    GPIO_PORTE_AFSEL_R |= 0b0001_0100; //ADC Channel 1&9 & on pins 2&4
    // GPIO_PORTE_PCTL_R: Not Used for Analog mode

    // iii. set digital or analog mode, and pin directions
    GPIO_PORTE_DEN_R &= 0b1110_0111; //Disable pin 2 & 4 digital mode
    GPIO_PORTE_DIR_R &= 0b1110_0111;//Pin 2 & 4 set to input
    GPIO_PORTE_AMSEL_R |= 0b0001_0100;//Enable Analog Mode on pins 2&4
    GPIO_PORTE_ADCCTL_R &= 0b1110_0111; //Default fine, OK if omitted

  // 2. Setup ADC
  // A) Configure ADC

        SYSCTL_RCGCADC_R |= 0x1; // Enable ADC clock
    ADC0_ADCCC = 0x0; //Use SysClk as ADC clk. Default, OK if omitted

    //Disable ADC Sample sequencers, while be configuring
    ADC0_ACTSS_R &= ~0x1; //Using SS0, but using SS1 or SS2 fine

    // Set desired ADC SS0 functionality
    ADC0_EMUX_R &= ~0x000F;//Set SS0 to trigger based on ADCPSSI reg
    ADC0_SSMUX0_R = 0x0000_0000; // Set all channels to sequence to 0
    ADC0_SSMUX0_R |= 0x0000_0091; //Update to sample channel 1, then 9

    // When to stop, and enable setting the Raw Interrupt status flag
    // after channel 1 and 9 have been converted
    ADC0_SSCTL0_R = 0x0000_0000; // First clear register to 0
    ADC0_SSCTL0_R |= 0x0000_0060 // Configure register

  // B) Setup ADC Interrupts
    NONE
  // 3. NVIC setup
  // A) Configure NVIC to allow ADC interrupts
    NONE
  // B) Bind ADC interrupt requests to User's Interrupt Handler
     NONE

  //re-enable ADC SS0
  ADC0_ACTSS_R |= 0x1;
}
```

Name: **Lab Section:**

c. **Complete the following API function to read in the light sensor values. Use polling (i.e. no Interrupt Service Routines). (5 pts)**

```
void get_sensor_reading(int *left_sensor, int *right_sensor)
{
   //initiate a SS0 conversion sequence
   ADC0_PSSI_R = 0x1;

   //wait for SS0 ADC conversions to be complete
   while((ADC0_RIS_R & 0x1) == 0) // or (~ADC0_RIS_R & 0x1)
        //wait
   }

   //clear Raw interrupt status flag
   ADC0_ISC_R = 0x01;

   // Get converted results for SS0 FIFO
   *left_sensor  = ADC0_SSFIFO0_R; // Channel 1
   *right_sensor = ADC0_SSFIFO0_R; // Channel 9
}
```

d. **Complete the following API function to set the speed of each motor. The inputs should be specified on a 100-point scale (e.g. 50 means 50% speed). Assume the input parameters are no less than 1 and no greater than 99. Also rounding errors are acceptable (i.e. do NOT use floating-point calculations) (5 pts)**

```
void set_motor_speed(int left_motor, int right_motor)
{
  TIMER1_TAMATCHR_R = 16000 - (16000*left_motor)/100;
  TIMER1_TBMATCHR_R = 16000 - (16000*right_right)/100;
}
```

Name: **Lab Section:**

e. **Complete `main()` (5pts)**

```
// Don't go into the light program
main()
{
  int left_sensor;
  int right_sensor;
  int left_motor;
  int right_motor;

  init_TIMER1_A_B();
  init_ADC();

  while(1)
  {
    get_sensor_reading(&left_sensor, &right_sensor);

    // Computed motor speed commands
    left_motor = (100 * left_sensor) / (left_sensor + right_sensor);
    right motor = (100 * right_sensor) / (left_sensor + right_sensor);

    set_motor_speed(left_motor, right_motor);
  }
}
```

Name: **Lab Section:**

Question 3: Square Waves (15 pts)

a) For Timer 1 module B using PWM mode, generate a symmetric square wave (i.e. 50% duty cycle) with a **<u>10 ms</u>** period. Assume the associated GPIO module has already been configured (5pts)

<u>Note: You are not allowed to use any MACROs in your code, except for register names.</u>

- **Example: You will lose points for: GPIO_PORTA_DEN_R = GPIO_PORTA_DEN_R | PIN1**
- **Must use: GPIO_PORTA_DEN_R = GPIO_PORTA_DEN_R | 0b0000_0010; // or 0x02**

```
void init_TIMER1()
{
  // Assuming associated GPIO module has already been configured

 SYSCTL_RCGCTIMER_R |= 0b0000_0010; // Enable Timer 1's clock

 //Disable Timer 1B device while we set it up
 TIMER1_CTL_R &= ~0x0100;

 // Set desired Timer 1 functionality
 TIMER1_CFG_R = 0x4; // Set to 16-bit mode
 TIMER1_TBMR_R = 0b0000_1010; //Timer 1B: PWM, Periodic Mode

 TIMER1_CTL_R = 0b0000_0000; // Can assume defaults to all 0's

    // Configure a period of 10 ms (requires more than 16-bit)
 TIMER1_TBPR_R = 0x02; //Timer 1B: prescaler to set upper 8-bits
 TIMER1_TBILR_R = 0x7100; //Timer 1B: set lower 16-bit for period

    // Configure for 50% duty cycle for a 10ms period
 TIMER1_TBPMR_R = 0x01;//PrescaleMatch stores upper 8-bits to match
 TIMER1_TBMATCHR_R = 3880; // store lower 16-bits to match

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

Name: **Lab Section:**

b) Now assume there is no PWM mode and that you have to use a Generic Waveform Generation approach (i.e. using an Interrupt Service Routine) to generate a symmetric square wave. Also assume the time to setup and execute the code in your ISR takes 20 µs (i.e. the CPU overhead involved with the ISR). What CPU utilize (i.e. percent of the CPU time) would be spent handing interrupts for: (5 pts)

i) Generating a square wave with a 10ms period

Overhead = 2*20us / 10ms ; // Two interrupts per period.
= $40*10^{-6} / 10*10^{-3}$
= .004 = .4%

ii) Generating a square wave with a 100 µs period

Overhead = 2*20us / 100us ; // Two interrupts per period.
= $40*10^{-6} / 100*10^{-6}$
= .4 = 40%

iii) Generating a square wave with a 50 µs period.

Overhead = 2*20us / 100us ; // Two interrupts per period.
= $40*10^{-6} / 50*10^{-6}$
= .8 = 80%

c) What is the key trade-off between using a Generic Waveform Generation approach (i.e. using an Interrupt Service Routine) vs. a Timer in PWM Mode for generating a symmetric square wave? (5 pts)

The key tradeoff is that the Generic Waveform Generation approach incurs Interrupt overhead while, the Timer PWM mode does not for generating a square wave.

Collaboration Documentation

List the people (First and Last name) you collaborated with: __________________________.

For each collaborator, describe the manner in which you collaborated:

1)

2)