EGEC 451 Lab Report 3

Dominic Jacobo

1. **Os.h:**

```
#ifndef __OS_H__
#define __OS_H__

//External Variables
volatile uint32_t Time_Slice_Counter;
volatile uint32_t Global_Thread_Id;

// ******** OS_Init ************
// initialize operating system, disable interrupts until OS_Launch
// initialize OS controlled I/O: SysTick, 16 MHz PLL
// Inputs: none
// Outputs: none
void OS_Init(void);

// ******** OS_AddThread ***************
// add three foreground threads to the scheduler
// Inputs: three pointers to a void/void foreground tasks
// Outputs: 1 if successful, 0 if this thread can not be added
int OS_AddThreads(void(*task0)(void),
		void(*task1)(void),
		void(*task2)(void));

// ******** OS_Launch ***************
// start the scheduler, enable interrupts
```

```
// Inputs: number of bus clock cycles for each time slice
//         (maximum of 24 bits)
// Outputs: none (does not return)
void OS_Launch(uint32_t theTimeSlice);

#endif
```

Os.c:

```
#include <stdint.h>
#include "os.h"
#include "PLL.h"
#include "tm4c123gh6pm.h"

// function definitions in OSasm.s
void OS_DisableInterrupts(void); // Disable interrupts
void OS_EnableInterrupts(void);  // Enable interrupts
int32_t StartCritical(void);
void EndCritical(int32_t primask);
void StartOS(void);

#define NUMTHREADS  3        // maximum number of threads
#define STACKSIZE   100      // number of 32-bit words in stack
struct tcb{ // This is the thread control block
  int32_t *sp;       // pointer to stack (valid for threads not running
  uint32_t id; // thread identifier number
  struct tcb *next;  // linked-list pointer
};
typedef struct tcb tcbType;
```

```
tcbType tcbs[NUMTHREADS];
tcbType *RunPt; // Used to point to the running thread
int32_t Stacks[NUMTHREADS][STACKSIZE];


// ******** OS_Init ************
// initialize operating system, disable interrupts until OS_Launch
// initialize OS controlled I/O: systick, 16 MHz PLL
// Inputs: none
// Outputs: none
void OS_Init(void){
  OS_DisableInterrupts();
  PLL_Init(Bus8MHz);        // set processor clock to 8 MHz
  NVIC_ST_CTRL_R = 0;        // disable SysTick during setup
  NVIC_ST_CURRENT_R = 0;     // any write to current clears it
  NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0x00FFFFFF)|0xE0000000; // priority 7
(Highest Prio for thread handler)
}

void SetInitialStack(int i){
  tcbs[i].sp = &Stacks[i][STACKSIZE-16]; // thread stack pointer
  Stacks[i][STACKSIZE-1] = 0x01000000;   // thumb bit
  Stacks[i][STACKSIZE-3] = 0x14141414;   // R14
  Stacks[i][STACKSIZE-4] = 0x12121212;   // R12
  Stacks[i][STACKSIZE-5] = 0x03030303;   // R3
  Stacks[i][STACKSIZE-6] = 0x02020202;   // R2
  Stacks[i][STACKSIZE-7] = 0x01010101;   // R1
  Stacks[i][STACKSIZE-8] = 0x00000000;   // R0
  Stacks[i][STACKSIZE-9] = 0x11111111;   // R11
```

```
  Stacks[i][STACKSIZE-10] = 0x10101010;  // R10
  Stacks[i][STACKSIZE-11] = 0x09090909;  // R9
  Stacks[i][STACKSIZE-12] = 0x08080808;  // R8
  Stacks[i][STACKSIZE-13] = 0x07070707;  // R7
  Stacks[i][STACKSIZE-14] = 0x06060606;  // R6
  Stacks[i][STACKSIZE-15] = 0x05050505;  // R5
  Stacks[i][STACKSIZE-16] = 0x04040404;  // R4
}

// ******** OS_AddThread ***************
// add three foreground threads to the scheduler
// Inputs: three pointers to a void/void foreground tasks
// Outputs: 1 if successful, 0 if this thread can not be added
int OS_AddThreads(void(*task0)(void),
                 void(*task1)(void),
                 void(*task2)(void)){ int32_t status;
  status = StartCritical();
  tcbs[0].next = &tcbs[1]; // 0 points to 1
  tcbs[1].next = &tcbs[2]; // 1 points to 2
  tcbs[2].next = &tcbs[0]; // 2 points to 0
  tcbs[0].id = 0; // id number for thread 0
  tcbs[1].id = 1; // id number for thread 1
  tcbs[2].id = 2; // id number for thread 2
  SetInitialStack(0); Stacks[0][STACKSIZE-2] = (int32_t)(task0); // PC
  SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32_t)(task1); // PC
  SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32_t)(task2); // PC
  RunPt = &tcbs[0];       // thread 0 will run first
  EndCritical(status);
```

```
  return 1;               // successful
}

/// ******** OS_Launch ***************
// start the scheduler, enable interrupts
// Inputs: number of bus clock cycles for each time slice
//         (maximum of 24 bits)
// Outputs: none (does not return)
void OS_Launch(uint32_t theTimeSlice){
  NVIC_ST_RELOAD_R = theTimeSlice - 1; // reload value
  NVIC_ST_CTRL_R = 0x00000007; // enable, core clock and interrupt arm
  StartOS();                   // start on the first task
}
```

OSasm.asm:

```
.thumb
        .text
        .align 2



        .global  RunPt            ; currently running thread
        .global  OS_DisableInterrupts
        .global  OS_EnableInterrupts
        .global  StartOS
        .global  SysTick_Handler
        .global Time_Slice_Counter ; global time slice varible here
        .global Global_Thread_Id ; global thread identifier number
```

```
OS_DisableInterrupts:  .asmfunc
      CPSID   I
      BX      LR
     .endasmfunc

OS_EnableInterrupts:  .asmfunc
      CPSIE   I
      BX      LR
     .endasmfunc

SysTick_Handler:  .asmfunc     ; 1) Saves R0-R3,R12,LR,PC,PSR

   CPSID   I                  ; 2) Prevent interrupt during switch

   LDR                        R0, Time_Slice_CounterAddr ; May need to put this whole part after
CPSID I (because increamenting counter than being interrupted sounds wrong)
         LDR R1, [R0]
         ADD R1, #1
         STR R1, [R0]

   PUSH    {R4-R11}           ; 3) Save remaining regs r4-11
   LDR     R0, RunPtAddr      ; 4) R0=pointer to RunPt, old thread
   LDR     R1, [R0]           ;    R1 = RunPt
   STR     SP, [R1]           ; 5) Save SP into TCB
   LDR     R1, [R1,#8]        ; 6) R1 = RunPt->next
   STR     R1, [R0]           ;    RunPt = R1

   LDR                        R2, [R0]               ; R2 = new RunPt
```

```
   LDR          R4, [R2,#4]       ; R4 = value of new RunPt.id
   LDR     R3, Global_Thread_IdAddr ; R3 points to Global_Thread_IdAddr
   STR          R4, [R3]                     ; R3 (Global_Thread_Id) = R2 (Value of new RunPt.id)

   LDR     SP, [R1]          ; 7) new thread SP; SP = RunPt->sp;
   POP     {R4-R11}          ; 8) restore regs r4-11
   CPSIE   I                 ; 9) tasks run with interrupts enabled
   BX      LR                ; 10) restore R0-R3,R12,LR,PC,PSR
  .endasmfunc
RunPtAddr .field RunPt,32
Time_Slice_CounterAddr .field Time_Slice_Counter,32
Global_Thread_IdAddr .field Global_Thread_Id,32

StartOS:  .asmfunc
         LDR    R0, Time_Slice_CounterAddr
         LDR             R1, [R0]                    ; Load in value of Time_Slice_Counter to R1
         MOV   R1, #0                          ; May not be needed because Time_Slice_Counter is 0 by
default
         ADD             R1, #1
         STR             R1, [R0]

   LDR     R0, RunPtAddr      ; currently running thread
   LDR     R2, [R0]          ; R2 = value of RunPt

   LDR          R1, [R2,#4]       ; R1 = value at RunPt.id
   LDR          R3, Global_Thread_IdAddr ; R3 points to Global_Thread_IdAddr
   STR     R1, [R3]                  ; R3 (Global_Thread_Id) = R1 (Value of RunPt.id)

   LDR     SP, [R2]          ; new thread SP; SP = RunPt->stackPointer;
```

```
    POP     {R4-R11}          ; restore regs r4-11
    POP     {R0-R3}           ; restore regs r0-3
    POP     {R12}
    POP     {LR}              ; discard LR from initial stack
    POP     {LR}              ; start location
    POP     {R1}              ; discard PSR
    CPSIE   I                 ; Enable interrupts at processor level
    BX      LR                ; start first thread
   .endasmfunc
   .end
```

User.c:

```
#include <stdint.h>
#include "os.h"
#include "tm4c123gh6pm.h"
#include "SSEG.h"
#include "Timer0A.h"
#include "Timer1A.h"
#include "Timer2A.h"
#include "Timer3A.h"
#include "LCD.h"

#define TIMESLICE 8000000  // The thread switch time in number of SysTick counts (bus clock cycles at 8 MHz)

#define FREQUENCY 8000000.0f
#define SEC_PER_MIN 60
#define GEARBOX_RATIO 120
#define PULSE_PER_ROTATION 8
```

```
uint32_t Count1;   // number of times thread1 loops
uint32_t Count2;   // number of times thread2 loops
uint32_t Count3;   // number of times thread3 loops

int pw = 0;
uint32_t RPM = 0;

void Task1(void){
  Count1 = 0;
  for(;;){
    Count1++;
    //GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | (0x01<<1);  // Show red
(Save this line for task 3)
    //Below is my attempt at the PWM DC Motor Code

    // set direction
    GPIO_PORTB_DATA_R &= ~0x04;
    GPIO_PORTB_DATA_R |= 0x08;

    // ramp up speed
    for (pw = 100; pw < 3999; pw += 20) {//Max PW 3999 (Which is MAX speed) Start from 100
         PWM1_3_CMPB_R = pw;
         Timer3A_Wait1ms(50);
         //LCD_Clear();
         RPM =
(SEC_PER_MIN*FREQUENCY)/(PULSE_PER_ROTATION*GEARBOX_RATIO*CC_Differ
ence)*10;
         //LCD_OutUFix(RPM);
```

```
    }

    // ramp down speed
    for (pw = 3940; pw >100; pw -= 20) {//Min PW 100(Which is pretty much min speed) Start from 3900
        PWM1_3_CMPB_R = pw;
        Timer3A_Wait1ms(50);
        //LCD_Clear();
        RPM = (SEC_PER_MIN*FREQUENCY)/(PULSE_PER_ROTATION*GEARBOX_RATIO*CC_Difference)*10;
        //LCD_OutUFix(RPM);
    }

    // reverse direction
    GPIO_PORTB_DATA_R &= ~0x08;
    GPIO_PORTB_DATA_R |= 0x04;

    // ramp up speed
    for (pw = 100; pw < 3999; pw += 20) {//Max PW 3999 (Which is MAX speed) Start from 100
        PWM1_3_CMPB_R = pw;
        Timer3A_Wait1ms(50);
        //LCD_Clear();
        RPM = (SEC_PER_MIN*FREQUENCY)/(PULSE_PER_ROTATION*GEARBOX_RATIO*CC_Difference)*10;
        //LCD_OutUFix(RPM);
    }
    // ramp down speed
```

```
  for (pw = 3940; pw >100; pw -= 20) {//Min PW 100(Which is pretty much min speed) Start from 3900
       PWM1_3_CMPB_R = pw;
       Timer3A_Wait1ms(50);
       //LCD_Clear();
       RPM = (SEC_PER_MIN*FREQUENCY)/(PULSE_PER_ROTATION*GEARBOX_RATIO*CC_Difference)*10;
       //LCD_OutUFix(RPM);
  }
 }
}

void Task2(void){
 Count2 = 0;
 for(;;){
  Count2++;
 //GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | (0x02<<1);  // Show blue (Save this line for task 3)
  if(Mailbox_Flag == 1){
      Timer0A_Wait1ms(40);
      LCD_Clear();
      LCD_OutUFix(RPM);
      Mailbox_Flag = 0;
  }
 }
}

void Task3(void){
 Count3 = 0;
```

```
  for(;;){
    Count3++;
    GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | (0x03<<1);  // Show red +
blue = purple/magenta
  }
}

int main(void){
  OS_Init();           // initialize, disable interrupts, set PLL to 8 MHz (equivalent to PLL_Init())
checked(this means should be ready)
  SYSCTL_RCGCGPIO_R |= 0x20;            // activate clock for Port F
  while((SYSCTL_PRGPIO_R&0x20) == 0){}; // allow time for clock to stabilize
  GPIO_PORTF_DIR_R |= 0x0E;             // make PF3-1 out
  GPIO_PORTF_AFSEL_R &= ~0x0E;          // disable alt funct on PF3-1
  GPIO_PORTF_DEN_R |= 0x0E;             // enable digital I/O on PF3-1
  GPIO_PORTF_PCTL_R &= ~0x0000FFF0;     // configure PF3-1 as GPIO
  GPIO_PORTF_AMSEL_R &= ~0x0E;          // disable analog functionality on PF3-1
  OS_AddThreads(&Task1, &Task2, &Task3);

  Timer0A_Init(8000000);
  Timer1A_Init(8000000);
  Timer2A_Init();
  Timer3A_Init(8000000);
  sevenseg_init(); // This is equivalent to SSI2_init()
  LCD_init();

  //Code for PWM init is below

   SYSCTL_RCGCPWM_R |= 0x02;        // enable clock to PWM1
```

```
SYSCTL_RCGCGPIO_R |= 0x20;       // enable clock to GPIOF
SYSCTL_RCGCGPIO_R |= 0x02;       // enable clock to GPIOB

//delayMs(1);                    // PWM1 seems to take a while to start
Timer0A_Wait1ms(1);

SYSCTL_RCC_R &= ~0x00100000;     // use system clock for PWM
PWM1_INVERT_R  |= 0x80;          // positive pulse
PWM1_3_CTL_R = 0;                // disable PWM1_3 during configuration
PWM1_3_GENB_R = 0x0000080C;      // output high when load and low when match
PWM1_3_LOAD_R = 3999;            // 4 kHz
PWM1_3_CTL_R = 1;                // enable PWM1_3
PWM1_ENABLE_R |= 0x80;           // enable PWM1

GPIO_PORTF_DIR_R |= 0x08;            // set PORTF 3 pins as output (LED) pin
GPIO_PORTF_DEN_R |= 0x08;            // set PORTF 3 pins as digital pins
GPIO_PORTF_AFSEL_R |= 0x08;          // enable alternate function
GPIO_PORTF_PCTL_R &= ~0x0000F000;    // clear PORTF 3 alternate function
GPIO_PORTF_PCTL_R |= 0x00005000;     // set PORTF 3 alternate funtion to PWM

GPIO_PORTB_DEN_R |= 0x0C;            // PORTB 3 as digital pins
GPIO_PORTB_DIR_R |= 0x0C;            // set PORTB 3 as output
GPIO_PORTB_DATA_R |= 0x08;           // enable PORTB 3

//Code for PWM init ends on the line above

OS_Launch(TIMESLICE); // doesn't return, interrupts enabled in here
return 0;            // this never executes
```

```
}
```

SSEG.h:

```
#ifndef SSEG_H_
#define SSEG_H_

//Global Variables
const static uint8_t digitPattern[] = {0xC0, 0xF9, 0xA4, 0xB0, 0x99,
                                       0x92, 0x82, 0xF8, 0x80, 0x90};
volatile uint8_t digitPattern_count; //Used to be just int i;

void SSEG1(uint32_t Thread_Id);
void SSEG2(uint32_t Time_Slice);
void sevenseg_init(void);
void SSI2_write(uint8_t data, uint8_t csMask);



#endif /* SSEG_H_ */
```

SSEG.C:

```
#include <stdint.h>
#include <math.h>
#include "tm4c123gh6pm.h"
#include "SSEG.h"
#include "Timer1A.h"

//void sevenseg_init(void);
//void SSI2_write(unsigned char data);
```

```
void SSEG1(uint32_t Thread_Id) {

   //(Set this function in user.c main somewhere I think) sevenseg_init();    // initialize SSI2 that connects to the shift registers

   //The seven segment digits start from the right and go left. So writing a 1 would be right most and writing a 8 would be left most. Look at binary representation

      //SSI2_write(digitPattern[digitPattern_count]);    // write digit pattern to the seven segments (First HCT595, which drives the cathodes)

      //SSI2_write((1 << digitPattern_count));           // select digit (Second HCT595, which drives the common anodes)

      //if (++digitPattern_count > 3)

         //digitPattern_count = 0;

   switch(Thread_Id){

      case 0: SSI2_write(digitPattern[0], 0x80);

            SSI2_write(0x08, 0x80);

            break;

      case 1: SSI2_write(digitPattern[1], 0x80);

            SSI2_write(0x08, 0x80);

            break;

      case 2: SSI2_write(digitPattern[2], 0x80);

            SSI2_write(0x08, 0x80);

            break;

      default: SSI2_write(0x8E, 0x80);

             SSI2_write(0x08, 0x80);

   }

}


void SSEG2(uint32_t Time_Slice){

   uint8_t i;
```

```
uint8_t digits[4];
uint8_t n_digits = 0;
uint32_t temp = Time_Slice;

while(temp != 0){
    temp /= 10;
    n_digits++;
}

for(i = 0; i < n_digits; ++i){
    digits[i] = Time_Slice % 10;
    Time_Slice /= 10;
}

for(i = 0; i < n_digits; ++i){
    switch(digits[i]){
    case 0: SSI2_write(digitPattern[0], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
    case 1: SSI2_write(digitPattern[1], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
    case 2: SSI2_write(digitPattern[2], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
    case 3: SSI2_write(digitPattern[3], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
```

```
        case 4: SSI2_write(digitPattern[4], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
        case 5: SSI2_write(digitPattern[5], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
        case 6: SSI2_write(digitPattern[6], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
        case 7: SSI2_write(digitPattern[7], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
        case 8: SSI2_write(digitPattern[8], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
        case 9: SSI2_write(digitPattern[9], 0x80);
            SSI2_write(1 << i, 0x80);
            break;
        }
    }
}

// enable SSI2 and associated GPIO pins
void sevenseg_init(void) {
    SYSCTL_RCGCGPIO_R |= 0x02;   // enable clock to GPIOB
    SYSCTL_RCGCGPIO_R |= 0x04;   // enable clock to GPIOC
    SYSCTL_RCGCSSI_R |= 0x04;    // enable clock to SSI2
```

```
    // PORTB 7, 4 for SSI2 TX and SCLK
    GPIO_PORTB_AMSEL_R &= ~0x90;      // turn off analog of PORTB 7, 4
    GPIO_PORTB_AFSEL_R |= 0x90;       // PORTB 7, 4 for alternate function
    GPIO_PORTB_PCTL_R &= ~0xF00F0000; // clear functions for PORTB 7, 4
    GPIO_PORTB_PCTL_R |= 0x20020000;  // PORTB 7, 4 for SSI2 function
    GPIO_PORTB_DEN_R |= 0x90;         // PORTB 7, 4 as digital pins

    // PORTC 7 for SSI2 slave select
    GPIO_PORTC_AMSEL_R &= ~0x80;      // disable analog of PORTC 7
    GPIO_PORTC_DATA_R |= 0x80;        // set PORTC 7 idle high
    GPIO_PORTC_DIR_R |= 0x80;         // set PORTC 7 as output for SS
    GPIO_PORTC_DEN_R |= 0x80;         // set PORTC 7 as digital pin

    SSI2_CR1_R = 0;              // turn off SSI2 during configuration
    SSI2_CC_R = 0;               // use system clock
    SSI2_CPSR_R = 16;            // clock prescaler divide by 16 gets 1 MHz clock
    SSI2_CR0_R = 0x0007;         // clock rate div by 1, phase/polarity 0 0, mode freescale, data
size 8
    SSI2_CR1_R = 2;              // enable SSI2 as master
}

// This function enables slave select, writes one byte to SSI2,
// wait for transmit complete and deassert slave select.
void SSI2_write(uint8_t data, uint8_t csMask) {
    GPIO_PORTC_DATA_R &= ~csMask;       // assert slave select
    SSI2_DR_R = data;            // write data
    while (SSI2_SR_R & 0x10) {}  // wait for transmit done
    GPIO_PORTC_DATA_R |= csMask;        // deassert slave select
}
```

Timer0A.h:

```
#ifndef __TIMER0A_H__
#define __TIMER0A_H__

// Set clock freq. so Timer0A_Wait10ms delays for exactly 10 ms if clock is not 80 MHz
void Timer0A_Init( uint32_t clkFreq );

// Time delay using busy wait
// The delay parameter is in units of the core clock (units of 12.5 nsec for 80 MHz clock)
void Timer0A_Wait( uint32_t delay );

// Time delay using busy wait
// This assumes 80 MHz system clock
void Timer0A_Wait1ms( uint32_t delay );

#endif
```

Timer0A.c:

```
#include <stdint.h>
#include "tm4c123gh6pm.h"
#include "Timer0A.h"

static uint32_t sysClkFreq = 8000000; // Assume 8 MHz clock by default

// Set clock freq. so Timer0A_Wait10ms delays for exactly 10 ms if clock is not 80 MHz
void Timer0A_Init( uint32_t clkFreq ){
  sysClkFreq = clkFreq;
  return;
```

```
}

// Time delay using busy wait
// The delay parameter is in units of the core clock (units of 12.5 nsec for 80 MHz clock)
//   Adapted from Program 9.8 from the book:
/*   "Embedded Systems: Introduction to ARM Cortex-M Microcontrollers",
   ISBN: 978-1477508992, Jonathan Valvano, copyright (c) 2013
   Volume 1, Program 9.8
*/
void Timer0A_Wait( uint32_t delay ){

  if(delay <= 1){ return; } // Immediately return if requested delay less than one clock

  SYSCTL_RCGCTIMER_R |= 0x00000001;  // 0) Activate Timer0
  TIMER0_CTL_R &= ~0x00000001;       // 1) Disable Timer0A during setup
  TIMER0_CFG_R = 0;                  // 2) Configure for 32-bit timer mode
  TIMER0_TAMR_R = 1;                 // 3) Configure for one-shot mode
  TIMER0_TAILR_R = delay - 1;        // 4) Specify reload value
  TIMER0_TAPR_R = 0;                 // 5) No prescale
  TIMER0_IMR_R = 0;                  // 6-9) No interrupts
  TIMER0_CTL_R |= 0x00000001;        // 10) Enable Timer0A

  //while( TIMER0_TAR_R ){} // Doesn't work; Wait until timer expires (value equals 0)
  // Or, clear interrupt and wait for raw interrupt flag to be set
  TIMER0_ICR_R = 1;
  while( !(TIMER0_RIS_R & 0x1) ){}
  return;
}
```

```
// Time delay using busy wait
// This assumes 80 MHz system clock
void Timer0A_Wait1ms( uint32_t delay ){
  uint32_t i;
  for( i = 0; i < delay; i++ ){
    Timer0A_Wait(sysClkFreq/1000);  // wait 1ms
  }
  return;
}
```

Timer1A.h:

```
#ifndef __TIMER1A_H__
#define __TIMER1A_H__

// Set clock freq. so Timer1A_Wait10ms delays for exactly 10 ms if clock is not 80 MHz
void Timer1A_Init( uint32_t clkFreq );

#endif
```

Timer1A.c:

```
#include <stdint.h>
#include <Timer1A.h>
#include "tm4c123gh6pm.h"
#include "SSEG.h"
#include "os.h"

static uint32_t sysClkFreq = 8000000; // Assume 8 MHz clock by default
```

```
// Set clock freq. so Timer1A_Wait10ms delays for exactly 10 ms if clock is not 80 MHz
void Timer1A_Init( uint32_t clkFreq ){
  sysClkFreq = clkFreq;

  //Sets up Timer1A for periodic interrupts

  SYSCTL_RCGCTIMER_R |= 0x00000002; // 0) Activate Timer1
  TIMER1_CTL_R &= ~0x00000001;      // 1) Disable Timer1A during setup
  TIMER1_CFG_R = 0x0;               // 2) Configure for 32-bit timer mode
  TIMER1_TAMR_R = 0x2;              // 3) Configure for Periodic mode
  TIMER1_TAILR_R = 0x3E80;          // 5) Specify reload value (Using 8000 because I want a
count down of 1ms)
  TIMER1_TAPR_R = 0;                // N/A) No prescalez
  TIMER1_IMR_R |= TIMER_IMR_TATOIM; // 6) Enable Time-Out interrupt
  TIMER1_ICR_R = TIMER_ICR_TATOCINT;// N/A) Clear Timer1A Time_Out RAW Interrupt
  TIMER1_CTL_R |= TIMER_CTL_TAEN;   // 7) Enable Timer1A
  NVIC_PRI5_R = (NVIC_PRI5_R & 0xFFFF0FFF) | 0xFFFF2FFF; //Timer2A = Priority of 1
  NVIC_EN0_R = 0x00200000;          // Enable interrupt 21 in NVIC

  return;
}


//Set up the Interrupt handler/systick handler for periodic interrupts
void Timer1A_Handler(){
   TIMER1_ICR_R = TIMER_ICR_TATOCINT; // Acknowledge Timer1A Timeout
   SSEG1(Global_Thread_Id);
   SSEG2(Time_Slice_Counter);
```

```
  return;
}
```

Timer2A.h:

```
#ifndef __TIMER2A_H__
#define __TIMER2A_H__

//Global Variables
volatile uint32_t Last_CC_Count;
volatile uint32_t Current_CC_Count;
volatile uint32_t CC_Difference;
volatile uint8_t Mailbox_Flag;

void EnableInterrupts();
void Timer2A_Init(); // Using PB0 for input capture (T2CCP0)

#endif
```

Timer2A.c:

```
#include <stdint.h>
#include <stdlib.h>
#include "tm4c123gh6pm.h"
#include "Timer2A.h"

// Using PB0 for input capture (T2CCP0)
void Timer2A_Init(){
  SYSCTL_RCGCTIMER_R |= 0x00000004; // Activate Timer2
  SYSCTL_RCGCGPIO_R |= 0x00000002;  // Activate Port B
```

```
  GPIO_PORTB_DEN_R |= 0x01;   // Enable digital I/O on PB0
  GPIO_PORTB_AFSEL_R |= 0x01; // Enable alternate function on PB0
  GPIO_PORTB_PCTL_R = (GPIO_PORTB_PCTL_R & 0xFFFFFFF0) | 0x00000007; // Enable
T2CCP0
  TIMER2_CTL_R &= ~TIMER_CTL_TAEN; // Disable Timer2A during setup
  TIMER2_CFG_R = TIMER_CFG_16_BIT; // Configure for 16-bit timer mode
  TIMER2_TAMR_R = TIMER_TAMR_TACMR | TIMER_TAMR_TAMR_CAP; // Configure
for capture mode
  TIMER2_CTL_R &= ~(TIMER_CTL_TAEVENT_POS | 0xC ); // Configure for rising-edge
event
  TIMER2_TAILR_R = 0x0000FFFF;     // Start value
  TIMER2_IMR_R |= TIMER_IMR_CAEIM; // Enable capture match interrupt
  TIMER2_ICR_R = TIMER_ICR_CAECINT; // Clear Timer2A capture match flag
  TIMER2_CTL_R |= TIMER_CTL_TAEN;  // Enable Timer2A
  NVIC_PRI5_R = (NVIC_PRI5_R & 0x00FFFFFF) | 0x20000000; // Timer2A = Priority 1
(Before Timer2A = Priority 2 0x40000000)
  NVIC_EN0_R = 0x00800000; // Enable interrupt 23 in NVIC
  EnableInterrupts();
  return;
}

void Timer2A_Handler(){
  TIMER2_ICR_R = TIMER_ICR_CAECINT; // Acknowledge Timer2A capture
  //Calculates the period or pulse length of the DC motor's encoder here
  Current_CC_Count = TIMER2_TAR_R;
  CC_Difference = abs(Last_CC_Count - Current_CC_Count);
  Last_CC_Count = Current_CC_Count;
  Mailbox_Flag = 1;
  return;
}
```

Timer3A.h:

```
#ifndef __TIMER3A_H__
#define __TIMER3A_H__

// Set clock freq. so Timer0A_Wait10ms delays for exactly 10 ms if clock is not 8 MHz
void Timer3A_Init( uint32_t clkFreq );

// Time delay using busy wait
// The delay parameter is in units of the core clock (units of 12.5 nsec for 8 MHz clock)
void Timer3A_Wait( uint32_t delay );

// Time delay using busy wait
// This assumes 8 MHz system clock
void Timer3A_Wait1ms( uint32_t delay );

#endif
```

Timer3A.c:

```
#include <stdint.h>
#include "tm4c123gh6pm.h"
#include "Timer3A.h"

static uint32_t sysClkFreq = 8000000; // Assume 8 MHz clock by default

// Set clock freq. so Timer0A_Wait10ms delays for exactly 10 ms if clock is not 8 MHz
void Timer3A_Init( uint32_t clkFreq ){
  sysClkFreq = clkFreq;
  return;
```

```
}

// Time delay using busy wait
// The delay parameter is in units of the core clock (units of 12.5 nsec for 80 MHz clock)
//   Adapted from Program 9.8 from the book:
/*   "Embedded Systems: Introduction to ARM Cortex-M Microcontrollers",
   ISBN: 978-1477508992, Jonathan Valvano, copyright (c) 2013
   Volume 1, Program 9.8
*/
void Timer3A_Wait( uint32_t delay ){

  if(delay <= 1){ return; } // Immediately return if requested delay less than one clock

  SYSCTL_RCGCTIMER_R |= 0x00000008;  // 0) Activate Timer3
  TIMER3_CTL_R &= ~0x00000001;       // 1) Disable Timer3A during setup
  TIMER3_CFG_R = 0;                  // 2) Configure for 32-bit timer mode
  TIMER3_TAMR_R = 1;                 // 3) Configure for one-shot mode
  TIMER3_TAILR_R = delay - 1;        // 4) Specify reload value
  TIMER3_TAPR_R = 0;                 // 5) No prescale
  TIMER3_IMR_R = 0;                  // 6-9) No interrupts
  TIMER3_CTL_R |= 0x00000001;        // 10) Enable Timer3A

  //while( TIMER0_TAR_R ){} // Doesn't work; Wait until timer expires (value equals 0)
  // Or, clear interrupt and wait for raw interrupt flag to be set
  TIMER3_ICR_R = 1;
  while( !(TIMER3_RIS_R & 0x1) ){}
  return;
}
```

```
// Time delay using busy wait
// This assumes 8 MHz system clock
void Timer3A_Wait1ms( uint32_t delay ){
  uint32_t i;
  for( i = 0; i < delay; i++ ){
    Timer3A_Wait(sysClkFreq/1000);  // wait 1ms
  }
  return;
}
```

LCD.c:

```
#include <stdint.h>
#include "Timer0A.h"
#include "SSEG.h"
#include "LCD.h"
#include "tm4c123gh6pm.h"
#include <stdio.h>

void DisableInterrupts(void);    // Disable interrupts
void EnableInterrupts(void);     // Enable interrupts
uint32_t StartCritical (void);   // previous I bit, disable interrupts
void EndCritical( uint32_t sr ); // restore I bit to previous value
void WaitForInterrupt(void);     // low power mode

// Macros
#define RS 1    // BIT0 mask for reg select
#define EN 2    // BIT1 mask for E
```

```
/*************** Private Functions ***************/

// LCD's SPI chip select is at PC6 (mask of 0x40 for SSI2_Write)
void LCD_nibble_write( uint8_t data, uint8_t control) {
  data &= 0xF0;       // clear lower nibble for control
  control &= 0x0F;    // clear upper nibble for data
  SSI2_write( (data | control), 0x40 );       // RS = 0, R/W = 0
  SSI2_write( (data | control | EN), 0x40 );  // pulse E
  //delayMs(0);
  SSI2_write( data, 0x40 );
  return;
}

/*************** Public Functions ***************/

// Clear the LCD
// Inputs: none
// Outputs: none
void LCD_Clear(void) {
  LCD_command(0x01);  // Clear Display
  // not necessary //LCD_command(0x80);  // Move cursor back to 1st position
}

// initialize SSI2 CS for LCD, then initialize LCD controller
// assumes Timer0A and SSI2 have already been initialized
void LCD_init(void) {
  SYSCTL_RCGCGPIO_R |= 0x04;   // enable clock to GPIOC
```

```
  // PORTC 6 for SSI2 chip select
  GPIO_PORTC_AMSEL_R &= ~0x40;      // disable analog
  GPIO_PORTC_DATA_R |= 0x40;        // set PORTC6 idle high
  GPIO_PORTC_DIR_R |= 0x40;         // set PORTC6 as output for CS
  GPIO_PORTC_DEN_R |= 0x40;         // set PORTC6 as digital pins

  Timer0A_Wait1ms(20);        // LCD controller reset sequence
  LCD_nibble_write(0x30, 0);
  Timer0A_Wait1ms(5);
  LCD_nibble_write(0x30, 0);
  Timer0A_Wait1ms(1);
  LCD_nibble_write(0x30, 0);
  Timer0A_Wait1ms(1);

  LCD_nibble_write(0x20, 0);  // use 4-bit data mode
  Timer0A_Wait1ms(1);
  LCD_command(0x28);          // set 4-bit data, 2-line, 5x7 font
  LCD_command(0x06);          // move cursor right
  LCD_command(0x01);          // clear screen, move cursor to home
  LCD_command(0x0F);          // turn on display, cursor blinking

  return;
}

// send a command to the LCD
void LCD_command( uint8_t command ) {
  uint32_t intStatus = StartCritical();
```

```
  LCD_nibble_write(command & 0xF0, 0);    // upper nibble first
  LCD_nibble_write(command << 4, 0);      // then lower nibble
  EndCritical( intStatus );

  if (command < 4)
    Timer0A_Wait1ms(2);         // command 1 and 2 needs up to 1.64ms
  else
    Timer0A_Wait1ms(1);         // all others 40 us

  return;
}

// send data (a character) to the LCD
void LCD_data( uint8_t data ) {
  uint32_t intStatus = StartCritical();
  LCD_nibble_write(data & 0xF0, RS);      // upper nibble first
  LCD_nibble_write(data << 4, RS);        // then lower nibble
  EndCritical( intStatus );

  Timer0A_Wait1ms(1);

  return;
}

//------------LCD_OutString------------
// Output String (NULL termination)
// Input: pointer to a NULL-terminated string to be transferred
// Output: none
```

```
void LCD_OutString( uint8_t *ptr ) {
  return;
}

//----------------------LCD_OutUDec----------------------
// Output a 32-bit number in unsigned decimal format
// Input: 32-bit number to be transferred
// Output: none
// Variable format 1-10 digits with no space before or after
void LCD_OutUDec( uint32_t n ) {
  // This function uses recursion to convert decimal number
  //   of unspecified length as an ASCII string
  if(n < 10){
     uint8_t decimal_to_char = (n % 10) + '0';
     LCD_data(decimal_to_char);
     return;
  }
  LCD_OutUDec(n / 10);
  LCD_OutUDec(n % 10);
}

//----------------------LCD_OutUHex----------------------
// Output a 32-bit number in unsigned hexadecimal format
// Input: 32-bit number to be transferred
// Output: none
// Variable format 1 to 8 digits with no space before or after
void LCD_OutUHex( uint32_t number ) {
// This function uses recursion to convert the number of
```

```
// unspecified length as an ASCII string
  return;
}

// -----------------------LCD_OutUFix----------------------
// Output characters to LCD display in fixed-point format
// unsigned decimal, resolution 0.1, range 000.0 to 999.9
// Inputs:  an unsigned 32-bit number
// Outputs: none
// E.g., 0,    then output "0.0"
//       3,    then output "0.3"
//       89,   then output "8.9"
//       123,  then output "12.3"
//       9999, then output "999.9"
//     > 9999, then output "*.***"
void LCD_OutUFix( uint32_t number ) {
  if(number < 10){// maybe condition should be if number < 10
     //send decimal point to LCD_data
     LCD_data('.');
     //send last number to LCD_data using mod 10 I think would work
     uint8_t decimal_to_char = (number % 10) + '0';
     LCD_data(decimal_to_char);
     return;
  }
  else if(number > 9999){//This should never happen but just incase
     //Just send -.--- or *.*** to the LCD. I think this is what he said to do
     //To implement this I should be able to  just hard code this and return
     LCD_data('*');
```

```
    LCD_data('.');
    LCD_data('*');
    LCD_data('*');
    LCD_data('*');
    return;
  }
  else{
  LCD_OutUDec(number / 10);
  LCD_OutUFix(number % 10);
  }
}
```

2. **Explain how the thread switcher works; this is the SysTick_Handler code in OSasm.asm. What must occur in order to switch from one thread to another?**
   -When the thread switcher is entered, interrupts are disabled to prevent possible error. Next the Time_Slice_Counter is incremented to show the start of a new thread. Next registers 4-11 and the SP for this old thread are saved for future use for this TCB. The TCB is then switched to the next TCB using the pointer to the next TCB, RunPt is set to this address and the thread switching is complete. Lastly the threads ID number is giving to the Global thread Id variable and SP and registers 4-11 are popped from the stack to be used. Interrupts are reenabled. In order to switch from one thread to another this handler must be called and this only happens when the timeslice value reaches zero for the timer. We set this timeslice to be one second for each thread.

3. **Explain what specific changes you made to the kernel/RTOS; i.e., the os.c and OSasm.asm. How exactly did you add support for the time slice counter and thread identifier number? Please be specific, especially when describing changes to the assembly code.**
   -The changes I made to the os involve changes to the TCB. Specifically, adding a uint32_t id variable for the thread identifier number. By putting this variable right after the the sp pointer I needed to make changes to the assembly code because if we wanted to address the tcb next pointer we would need to use a offset of 8 bytes rather than 4 when working with the RunPt pointer. I assigned the id numbers for the specific thread in the OS_AddThreads function. The last change I did to the kernel/RTOS was add global variable for the Thread_Slice_Counter and Global_Thread_Id which are set by the assembly code. In order to support this new functionality in the assembly code I had to

create global variables in the assembly code to connect to the global variables of the C code. For the time_slice_counter I would load its address into a register then load the value into another register and lastly increment its value by 1 and store this value back to its memory address. For the Global_Thread_Id I would use the RunPt address with an offset of 4 bytes to get the value of the running TCB's value for id, so the value from RunPt.id. I would put this value into a register then store this value to the memory address of the Global_Thread_Id variable.

4. **Briefly explain your user code for interfacing with the seven-segment display. How does your code time-multiplex the display?**
   -For interfacing with the seven-segment display I pushed this code to its own files called SSEG.h and SSEG.c. The functions in this file are only called when Timer1A periodically times out, which in this case is every 2ms (with a priority of 1). Timer1A first Acknowledges the timeout then calls the SSEG1 function. This function is responsible for time-multiplexing the Global_Thread_Id number onto the left most segment. The function uses a switch statement to takes the Global_Thread_Id variable as input. Case 0-2 will first send the digit pattern to the SSI2_write function and the correct cs for the device. Lastly, the case sends a hex value for the digit to display on and the cs hex value and breaks the switch statement. SSEG2 is called next in the handler function. This function asks similarly but for outputting the Time_Slice_Counter to the 3 remaining segments. First the function finds the number of digits that makes up the uint32_t variable. Next it stores digit by digit into an array. Each digit is then sent one by one as inputs input a switch statement that act similar to the other switch statement.

5. **Compare your program for Lab 2 versus your program for Lab 3. Consider the speed (in rpm) that is being displayed on the LCD; which program is more responsive , i.e., shows the true speed of the DC motor on the LCD? In addition, comment on which program is more responsive if the time slice length for Lab 3 is reduced from 1 second to 1ms.**
   -Do to the tasks only being able to run for 1 second at a time this causes a a lot more delay for outputting to the LCD and ramping up the DC motor. Due to this Lab 2 is more responsive for displaying the rpm, so you get a much more accurate reading. I believe that if the time slice length for Lab 3 was reduced from 1 seond to 1 ms the responsive to our human eye would be roughly the same. It would look pretty much the same at those speeds.

6. **List any sources/websites used or students with whom you discussed this assignment. Use IEEE-syle citations in your question responses to show where you used the information from each source/website.**
   -N/A