# ECE 355 | Fall 2024
# Assignment 2

## Question 1

```
//Part 1
#define BIT0 0x0
#define BIT1 0x1
#define BIT2 0x2
#define BIT3 0x4
#define BIT4 0x8
#define BIT5 0x10
#define BIT6 0x20
#define BIT7 0x40

#define switch BIT0
#define displayBitsB 0xF0 // PortB
#define displayBitsA 0x0F // PortA
#define LED1 BIT6
#define LED2 BIT5

// IO ports
// Port A
#define PAIN (volatile unsigned char *) 0xFFFFFFF0
#define PAOUT (volatile unsigned char * ) 0xFFFFFFF1
#define PADIR (volatile unsigned char *) 0xFFFFFFF2
// Port B
#define PBIN (volatile unsigned char *) 0xFFFFFFF3
#define PBOUT (volatile unsigned char * ) 0xFFFFFFF4
#define PBDIR (volatile unsigned char *) 0xFFFFFFF5

// timer
#define CNTM (volatile unsigned int *) 0xFFFFFFD0 // initial timer value
#define CTCON (volatile unsigned char *) 0xFFFFFFD8 // timer control register
#define CTSTAT (volatile unsigned char *) 0xFFFFFFD9 // timer status register
#define IVECT (volatile unsigned int *) (0x20) // interrupt vector address

char segmentDigitA = 0;  // 7-segment display digit A
char segmentDigitB = 0;  // 7-segment display digit B
char portBValSnapshot;   // temporary store for port B value
char portAValSnapshot;   // temporary store for port A value

void main(){
    unsigned int timerCount=100000000; // trigger interrupt every second @ 100MHz
    *CTCON = BIT1; // ensure timer is stopped initially
    *CTCON |= BIT7; // set timer mode
    *CNTM = timerCount; // load initial timer value

    // set PB0 as input (switch)
    *PBDIR &= ~switch;

    // configure Port B upper nibble (PB4-PB7) as output
    *PBDIR |= displayBitsB;
    // configure Port A lower nibble (PA0-PA3) as output
    *PADIR |= displayBitsA;

    // set PA5 and PA6 as LED output pins
    *PADIR |= LED1 | LED2;

    // initialize LEDs: turn LED1 on, turn LED2 off
    *PAOUT |= LED1; // LED1 is initially on
    *PAOUT &= ~LED2; // LED2 is initially off

    // set interrupt vector to timer ISR
```

```
    *IVECT = (unsigned int) &timerISR;

    // initialize display to show 0
    *PBOUT &= ~0xF0; // clear upper 4 bits of port B
    *PAOUT &= ~0x0F; // clear lower 4 bits of port A

    // enable interrupt by setting PSR[6]
    asm("MoveControl PSR, #0x20");

    // enable timer interrupt
    *CTCON |= BIT4;

    // start the timer
    *CTCON &= ~BIT1; // clear stop bit
    *CTCON |= BIT0; // set start bit

    while(1){
        // check if switch (PB0) is pressed
        if(~*PBIN & switch != 0){
            // wait until switch is released
            while(~*PBIN & switch != 0);
            // temporary store for updating LEDs
            char tempPortA = *PAOUT;
            // check if LED1 is currently on
            if(*PAOUT & LED1 != 0){
                // turn on LED2 and turn off LED1
                tempPortA |= LED2; // enable LED2
                tempPortA &= ~LED1; // disable LED1
                *PAOUT = tempPortA; // update PAOUT with new values
            } else {
                // turn on LED1 and turn off LED2
                tempPortA |= LED1; // enable LED1
                tempPortA &= ~LED2; // disable LED2
                *PAOUT = tempPortA; // update PAOUT with new values
            }
        }
    }
}

// Timer interrupt service routine
interrupt void timerISR()
{
    // check if LED1 is on, meaning we're decrementing digit A
    if(*PAOUT & LED1 != 0){
        // if digit A reaches 0, reset to 10
        segmentDigitA = segmentDigitA >= 0 ? 10 : segmentDigitA;
        segmentDigitA--; // decrement digit A

        // temporarily store the current value of port A
        portAValSnapshot = *PAOUT;
        // clear the 7-segment display portion of port A
        portAValSnapshot &= ~displayBitsA;
        // update display with the new value for digit A
        portAValSnapshot |= segmentDigitA;
        *PBOUT = portAValSnapshot;  // update the 7-segment display
    } else if(*PAOUT & LED2 != 0){ // if LED2 is on, decrement digit B
        segmentDigitB = segmentDigitB >= 0 ? 10 : segmentDigitB;
        segmentDigitB--; // decrement digit B

        char shiftedValue;
        // shift the value of digit B to align with upper 4 bits
        shiftedValue = segmentDigitB << 4;

        // temporarily store the current value of port B
        portBValSnapshot = *PBOUT;
        // clear the upper nibble (7-segment display) portion of port B
        portBValSnapshot &= ~displayBitsB;
        // update display with new value for digit B
        portBValSnapshot |= shiftedValue;
        *PBOUT = portBValSnapshot;  // update the 7-segment display
```

```
    }
}
```

# Question 2

```
//Part 2
#define BIT0 0x0
#define BIT1 0x1
#define BIT2 0x2
#define BIT3 0x4
#define BIT4 0x8
#define BIT5 0x10
#define BIT6 0x20
#define BIT7 0x40

// IO ports
#define PCONT (volatile unsigned char *) 0xFFFFFFF7
// Port A
#define PAIN (volatile unsigned char *) 0xFFFFFFF0
#define PAOUT (volatile unsigned char * ) 0xFFFFFFF1
#define PADIR (volatile unsigned char *) 0xFFFFFFF2
// Port B
#define PBIN (volatile unsigned char *) 0xFFFFFFF3
#define PBOUT (volatile unsigned char * ) 0xFFFFFFF4
#define PBDIR (volatile unsigned char *) 0xFFFFFFF5

// timer
#define CNTM (volatile unsigned int *) 0xFFFFFFD0 // initial timer value
#define CTCON (volatile unsigned char *) 0xFFFFFFD8 // timer control register
#define CTSTAT (volatile unsigned char *) 0xFFFFFFD9 // timer status register
#define IVECT (volatile unsigned int *) (0x20) // interrupt vector address

#define SWITCH BIT7
#define DISPLAY_DIGIT1 0x78 // PORT A
#define DISPLAY_DIGIT2 0xF0 // PORT B
#define LED1 BIT2
#define LED2 BIT0

char digitValue1 = 0; // holds the value for 7-segment digit 1
char digitValue2 = 0; // holds the value for 7-segment digit 2
char portBValueTemp;  // temporary storage for port B value
char portAValueTemp;  // temporary storage for port A value

interrupt void timerISR(); // interrupt service routine
char counterEnabled = 0; // flag to enable counting
char isSwitchPressed = 0; // flag to track switch state

void main(){
    unsigned int timerReloadValue = 100000000; // interrupt every second @ 100MHz
    *CTCON = BIT1; // ensure the timer is stopped initially
    *CTCON &= ~BIT7; // set in counter mode
    *CNTM = timerReloadValue; // load the initial timer value

    // configure Port B outputs for digit2, LED1, and LED2
    *PBDIR |= DISPLAY_DIGIT2 + LED1 + LED2;

    // configure Port A outputs for digit1
    *PADIR |= DISPLAY_DIGIT1;
    // set Port A input for the switch
    *PADIR &= ~(SWITCH);

    // set interrupt vector to point to the timer ISR
    *IVECT = (unsigned int) &timerISR;

    // initialize the display to show 0 on both digits
    *PBOUT &= ~DISPLAY_DIGIT2; // clear digit2 on port B
    *PAOUT &= ~DISPLAY_DIGIT1; // clear digit1 on port A
```

```
    // initialize LEDs
    *PBOUT |= LED1;  // turn LED1 on
    *PBOUT &= ~LED2; // turn LED2 off

    // enable interrupts by setting PSR[6]
    asm("MoveControl PSR, #0x20");
    // enable Port A interrupt
    *PCONT |= BIT4;

    // start the timer
    *CTCON &= ~BIT1; // clear stop bit
    *CTCON |= BIT0; // start the timer

    char currentTimerValue = 0;
    while(1){
        while(((*CTSTAT & BIT0) == 0)){}; // wait for timer to complete its count

        // check if LED1 is on, increment digit1
        if((*PBOUT & LED1) == 1){
            digitValue1++; // increment digit 1
            digitValue1 = digitValue1 >= 10 ? 0 : digitValue1; // roll over if it reaches 10

            char tempA;
            tempA = digitValue1 << 3; // shift digit1 value by 3 bits
            portAValueTemp = *PAOUT;
            portAValueTemp &= ~DISPLAY_DIGIT1; // clear the digit1 display bits
            portAValueTemp |= tempA; // set the new value for digit1
            *PAOUT = portAValueTemp;  // update Port A output
        }

        // check if LED2 is on, increment digit2
        if((*PBOUT & LED2) == 1){
            digitValue2++; // increment digit 2
            digitValue2 = digitValue2 >= 10 ? 0 : digitValue2; // roll over if it reaches 10

            char tempB;
            tempB = digitValue2 << 4; // shift digit2 value by 4 bits
            portBValueTemp = *PBOUT;
            portBValueTemp &= ~DISPLAY_DIGIT2; // clear the digit2 display bits
            portBValueTemp |= tempB; // set the new value for digit2
            *PBOUT = portBValueTemp;  // update Port B output
        }

        // Counter will automatically reload based on the timer config (ref: page 398 of the textbook)
    }

    // if the switch is pressed, counting will be disabled (counterEnabled = 0)
}

// Timer interrupt service routine
interrupt void timerISR()
{
    // check if the switch is pressed
    if(((*PAIN & SWITCH) == 0) && !isSwitchPressed){
        isSwitchPressed = 1; // mark switch as pressed
    } else if(((*PAIN & SWITCH) != 0) && isSwitchPressed){
        isSwitchPressed = 0; // reset switch press flag

        char tempPortA;
        // toggle LED states based on current LED1 status
        if(*PAOUT & LED1 != 0){ // if LED1 is currently on
            tempPortA = *PAOUT;
            tempPortA |= LED2; // turn LED2 on
            tempPortA &= ~LED1; // turn LED1 off
            *PAOUT = tempPortA; // update Port A with the new LED states
        } else {
            tempPortA = *PAOUT;
            tempPortA |= LED1; // turn LED1 on
            tempPortA &= ~LED2; // turn LED2 off
```

```
            *PAOUT = tempPortA; // update Port A with the new LED states
        }
    }
}
```

# Question 3

We are given four independent pre-emptive tasks to be scheduled using the Earliest Deadline First (EDF) priority algorithm. In EDF, the task with the nearest deadline is executed first. If multiple tasks have the same deadline, we break ties using Rate Monotonic (RM), which gives higher priority to the task with the shorter period. Here's how we approached the problem.

The hyperperiod is the least common multiple (LCM) of all the task periods, which is the interval over which the schedule repeats. For the given tasks:
Task periods: 30, 40, 60, and 120.
The LCM of these periods is 120. Therefore, the schedule repeats after every 120 time units. The tasks' key parameters are summarized in the table below. Each task arrives periodically and has a specified worst-case execution time (WCET) and deadline.

| Task | Period (Pi) | WCET (Ci) | Deadline (Di) | Initial Delay (Φi) |
|---|---|---|---|---|
| T1 | 30 | 10 | 30 | 0 |
| T2 | 40 | 10 | 40 | 0 |
| T3 | 60 | 10 | 50 | 0 |
| T4 | 120 | 15 | 100 | 0 |

We use the EDF algorithm to determine the task execution schedule between time t = 0 and t = 120. Tasks are executed based on their earliest deadlines. When two tasks have the same deadline, the tie is broken using Rate Monotonic prioritization, favoring tasks with shorter periods.

The detailed schedule is as follows:

| Time Interval | Task Executed |
|---|---|
| t = 0 - 10 | T1 |
| t = 10 - 20 | T2 |
| t = 20 - 30 | T3 |
| t = 30 - 40 | T1 |
| t = 40 - 50 | T2 |
| t = 50 - 65 | T4 |
| t = 65 - 70 | T4 |
| t = 70 - 80 | T1 |
| t = 80 - 90 | T3 |
| t = 90 - 100 | T1 |
| t = 100 - 110 | T2 |
| t = 110 - 120 | T4 |

In the interval [0, 120), tasks are scheduled based on their deadlines, and the schedule repeats after 120 time units. The earliest deadline determines which task runs, with ties broken by Rate Monotonic prioritization.