ENCM 511 Assignment 4

Preamble

```
#define BAR_POSITIONS 64                    // number of positions in the bar graph

#define BAR_DIVISOR (1024 / BAR_POSITIONS)  // divide 10bit adc read into bar postions

#define AVERAGING_N 11                      // number of elements used in moving average filter

char bar_char = '='; // = or - or X            char which makes up the bar graph
```

The above definitions have been added at the start of the code to improve readability. BAR_POSITIONS is the maximum number of positions in the bar graph. BAR_DIVISOR is used to divide the count of the ADC (1024 for the 10-bit ADC) to determine the correct number of bar graph characters. AVERAGING_N is the size of the moving average filter used to smooth out the ADC values. The bar_char is updated during runtime and dictates which characters compose the bar graph.

```
uint8_t app_flags = 0;
#define BAR_UPDATE_FLAG     0   // Flag that ADC bar has to be updated
#define NUM_UPDATE_FLAG     1   // Flag that ADC value display has to be updated
#define PB_UPDATE_FLAG      2   // Flag that an IOC interrupt occured (for buttons)
#define PB0_LAST            3   // Last value of PB0
#define PB1_LAST            4   // Last value of PB1
#define PB2_LAST            5   // Last value of PB2
#define NUM_MODE            6   // 0 = Hex display, 1 = decimal display
```

The app_flags bit field is used to keep track of flags to control program flow.

```
#define SET_BIT(flags, n)       ((flags) |= (1 << (n)))
#define CHECK_BIT(flags, n)     (((flags) >> (n)) & 1)
#define CLEAR_BIT(flags, n)     ((flags) &= ~(1 << (n)))
#define TOGGLE_BIT(flags, n)    ((flags) ^= (1 << (n)))
```

It is used in conjunctions with these helper macros to allow reading/writing of the bitfield.

```
SET_BIT(app_flags, BAR_UPDATE_FLAG);
```

Example above.

Peripherals

```
void adc_init(void)
{
    AD1CON2bits.PVCFG = 0;      // Selects Positive voltage reference
    AD1CON2bits.NVCFG0 = 0;     // Selects Negative voltage reference

    //Setup Channel 0 sample A
    AD1CHSbits.CH0NA = 0;       // Negative input is selected as gnd
    AD1CHSbits.CH0SA = 5;       // Positive input selected as AN5 (PIN7)

    AD1CON3bits.ADCS = 255;     // AD Conversion clock is set to 256 * Tcy
    AD1CON1bits.SSRC = 7;       // ADC occurs based off SAMP
    AD1CON1bits.FORM = 0;       // Data output is absolute decimal unsigned right justified
    AD1CON5bits.ASEN = 0;       // Disable auto scan
    AD1CON1bits.DMAEN = 0;      // Disable DMA
    AD1CON2bits.SMPI = 0;       // Interrupts at the completion of each sample
    AD1CON1bits.MODE12 = 0;     // 10-bit ADC mode
    AD1CON1bits.ASAM = 0;       // Sampling starts when SAMP is set manually

    AD1CON1bits.ADON = 1;       // Enable ADC
    AD1CON1bits.SAMP = 0;
}
```

```
void IO_init(void)
{
    ANSELA = 0x0000; /* keep this line as it sets I/O pins that can also be analog to be digital */
    ANSELB = 0x0008; /* keep this line as it sets I/O pins that can also be analog to be digital */

    TRISBbits.TRISB3 = 1;   // Set to input (ADC_input)

    TRISAbits.TRISA4 = 1;   // Set to input (PB0)
    TRISBbits.TRISB8 = 1;   // Set to input (PB1)
    TRISBbits.TRISB9 = 1;   // Set to input (PB2)

    IOCPUAbits.IOCPA4 = 1;  // Enable pull-up (PB0)
    IOCPUBbits.IOCPB8 = 1;  // Enable pull-up (PB1)
    IOCPUBbits.IOCPB9 = 1;  // Enable pull-up (PB2)

    PADCONbits.IOCON = 1;   // Enable interrupt-on-change (IOC)

    IOCNAbits.IOCNA4 = 1;   // Enable high-to-low IOC (PB0)
    IOCPAbits.IOCPA4 = 1;   // Enable low-to-high IOC (PB0)
    IOCNBbits.IOCNB8 = 1;   // Enable high-to-low IOC (PB1)
    IOCPBbits.IOCPB8 = 1;   // Enable low-to-high IOC (PB1)
    IOCNBbits.IOCNB9 = 1;   // Enable high-to-low IOC (PB2)
    IOCPBbits.IOCPB9 = 1;   // Enable low-to-high IOC (PB2)

    IFS1bits.IOCIF = 0;             // Clear system-wide IOC flag
    IPC4bits.IOCIP = ISR_PRIORITY;  // Set IOC priority)
    IEC1bits.IOCIE = 1;             // Enable IOC
    return;
}
```

```c
void timer_init(void)
{
    T2CONbits.T32 = 0;              // Operate timers 2 & 3 as separate 16-bit timers

    // Timer 3 (for delay function)
    T3CONbits.TCKPS = 2;            // set prescaler to 1:64
    IFS0bits.T3IF = 0;              // clear interrupt flag
    IEC0bits.T3IE = 0;              // disable interrupt
    PR3 = 0xFFFF;                   // max timer period (use as counter)

    return;
}
```

```c
void InitUART2(void)
{

    RPINR19bits.U2RXR = 11; // Assign U2RX to RP11 (pin 22)
    RPOR5bits.RP10R = 5;    // Assign RP10 (pin 21) to U2TX

    //U2MODE = 0b0000000010001000;
    U2MODEbits.BRGH = 1;
    U2MODEbits.WAKE = 1;

    //U2BRG = 25;               // Baud rate = 38400
    U2BRG = 8;                  // Baud rate = 115200

    U2STAbits.UTXISEL0 = 0;     // Interrupt when a character is transferred to the Transmit Shift Register
    U2STAbits.UTXISEL1 = 0;     // (this implies there is at least one character open in the transmit buffer)
    U2STAbits.URXEN = 1;        // Receive is enabled, U2RX pin is controlled by UART2
    U2STAbits.UTXEN = 1;        // Transmit is enabled, U2TX pin is controlled by UART2
    U2STAbits.URXISEL = 0b00;   // Interrupt is set when any character is received and transferred from the RSR to the receive buffer

    IFS1bits.U2TXIF = 0;        // Set flag to 0
    IPC7bits.U2TXIP = 3;        // Set priority
    IEC1bits.U2TXIE = 1;        // Enable interrupt

    IFS1bits.U2RXIF = 0;        // Set flag to 0
    IPC7bits.U2RXIP = 4;        // Set priority
    IEC1bits.U2RXIE = 1;        // Enable interrupt

    U2MODEbits.UARTEN = 1;      // Enable UART Rx

    U2STAbits.UTXEN = 1;        // Enable UART Tx
    return;
}
```

Above are the initialization functions to be called at the beginning of main. Line functions are elucidated by their comments.

Datatypes and Initialization

```
uint16_t prev_reading = 0;
uint8_t prev_bar_val = 0;
```

These two variables store the previous reading of the ADC and the previous value of the bar graph. The ADC reading is selected as uint16_t as we are using the ADC in 10-bit mode, and the previous bar value is selected to be uint8_t as we have 64 graphical elements for the graph. These are both used to compare the previous value with the new value to determine if the UART display must be updated (as opposed to constantly updating the UART display with the current value).

```
uint16_t adc_reading = 0;
uint8_t bar_val = 0;

uint16_t samples[AVERAGING_N] = {0};
```

These are used to hold the new value from the ADC and the current bar graph value. The samples variable is part of the aforementioned moving average filter.

Operation

```
    IO_init();
    timer_init();
    InitUART2();
    adc_init();

    Disp2String("\033[?25l");   // Hide cursor
    Disp2String("\033[2J");     // Clear screen
    Disp2String("\r");          // Return cursor

    XmitUART2(' ', BAR_POSITIONS + 1);    // move cursor to 'home'

    // Ensure these are printed once right at startup
    SET_BIT(app_flags, BAR_UPDATE_FLAG);
    SET_BIT(app_flags, NUM_UPDATE_FLAG);
```

The code begins by calling all four initialization functions and then sending some ASCII escape codes to clean up the terminal and set the cursor to the ‘home’ position. It then sets the BAR_UPDATE_FLAG and NUM_UPDATE_FLAG to ensure that the graph is printed on startup.

```
if(CHECK_BIT(app_flags, PB_UPDATE_FLAG))
{
    CLEAR_BIT(app_flags, PB_UPDATE_FLAG);

    if(!PB0 && CHECK_BIT(app_flags, PB0_LAST)) {    // Transition to pressed
        bar_char = '=';
        SET_BIT(app_flags, BAR_UPDATE_FLAG);
    }
    if(!PB1 && CHECK_BIT(app_flags, PB1_LAST)) {    // Transition to pressed
        bar_char = '-';
        SET_BIT(app_flags, BAR_UPDATE_FLAG);
    }
    if(!PB2 && CHECK_BIT(app_flags, PB2_LAST)) {    // Transition to pressed
        bar_char = 'X';
        SET_BIT(app_flags, BAR_UPDATE_FLAG);
    }

    // Update last button values
    if(PB0)
        SET_BIT(app_flags, PB0_LAST);
    else
        CLEAR_BIT(app_flags, PB0_LAST);
    if(PB1)
        SET_BIT(app_flags, PB1_LAST);
    else
        CLEAR_BIT(app_flags, PB1_LAST);
    if(PB2)
        SET_BIT(app_flags, PB2_LAST);
    else
        CLEAR_BIT(app_flags, PB2_LAST);
}
```

Inside the while(1) loop, the code checks to see if a button event has occurred, if it has it will compare each button position with its last state to determine if a button becomes pressed, after which it will update the bar_char value accordingly and raise the BAR_UPDATE_FLAG. Afterwards it will update the last state of each flag for each of the buttons to be able to determine if a button press has occurred next time.A

```
// Interrupt-on-change ISR
void __attribute__ ((interrupt, no_auto_psv)) _IOCInterrupt(void) {
    SET_BIT(app_flags, PB_UPDATE_FLAG);
    IFS1bits.IOCIF = 0; // Clear system-wide IOC flag
}
```

This is the where the PB_UPDATE_FLAG gets raised upon an IOC interrupt, after which the button logic will run next time the while(1) loop repeats.

```
switch(RecvUartChar()) {
    case 'x':
        CLEAR_BIT(app_flags, NUM_MODE);
        SET_BIT(app_flags, NUM_UPDATE_FLAG);
        break;
    case 'd':
        SET_BIT(app_flags, NUM_MODE);
        SET_BIT(app_flags, NUM_UPDATE_FLAG);
        break;
    default:
        break;
}
```

Next, the program reads the input from the UART and determines if there was a key press and if the key was either ‘x’ or ‘d’, if it is, it will update the NUM_MODE flag accordingly and also raise the NUM_UPDATE_FLAG. This informs the code that it needs to update that part of the terminal. If any other key is pressed, nothing happens.

```
// MOVING AVERAGE FILTER
// shift old values back in averaging array
for(int i = 0; i < AVERAGING_N - 1; i++) {
    samples[i] = samples[i + 1];
}
// Sample ADC to latest value in averaging array
samples[AVERAGING_N - 1] = do_adc();
// calculate average
adc_reading = 0;
for(int i = 0; i < AVERAGING_N; i++) {
    adc_reading += samples[i];
}
adc_reading = adc_reading / AVERAGING_N;
```

This section is where the moving average filter and ADC read occur. First, we shift the contents of the samples array down one index and update the highest index with the new ADC reading. Then we reset adc_reading (holding the value from last time) and sum the contents of the array before and then divide by the length of the samples array. This gives us a moving average filter to eliminate noise from the potentiometer and ADC.

```
if(adc_reading != prev_reading) {

    SET_BIT(app_flags, NUM_UPDATE_FLAG);
    prev_reading = adc_reading;

    bar_val = adc_reading / BAR_DIVISOR;

    if(bar_val != prev_bar_val) {
        SET_BIT(app_flags, BAR_UPDATE_FLAG);
        prev_bar_val = bar_val;
    }
}
```

Next, we check if the new value of the ADC differs from the previous, if so, we must update the displayed number next to our graph. Then we divide the reading by the amount of bar positions which is 1024/64 = 16, this means that only when the ADC value changes 16 times a character changes on the screen. If the new bar_val differs from prev_bar_val, the BAR_UPDATE_FLAG is asserted.

```
if(CHECK_BIT(app_flags, BAR_UPDATE_FLAG)) {
    update_bar();
    CLEAR_BIT(app_flags, BAR_UPDATE_FLAG);
}
if(CHECK_BIT(app_flags, NUM_UPDATE_FLAG)) {
    update_num();
    CLEAR_BIT(app_flags, NUM_UPDATE_FLAG);
}
```

Finally, we check to see if the BAR_UPDATE_FLAG or NUM_UPDATE_FLAG has been asserted and call the corresponding function.

```
void update_bar(void)
{
    Disp2String("\033[s");  // save cursor position
    XmitUART2('\r', 1);     // reset cursor to beginning of line

    // build bar graph
    XmitUART2('[', 1);
    XmitUART2(bar_char, bar_val);
    XmitUART2(' ', BAR_POSITIONS - bar_val - 1);    // minus one for max of 63rd bar char (64 positions 0-indexed)
    XmitUART2(']', 1);

    Disp2String("\033[u");  // restore cursor position
}
```

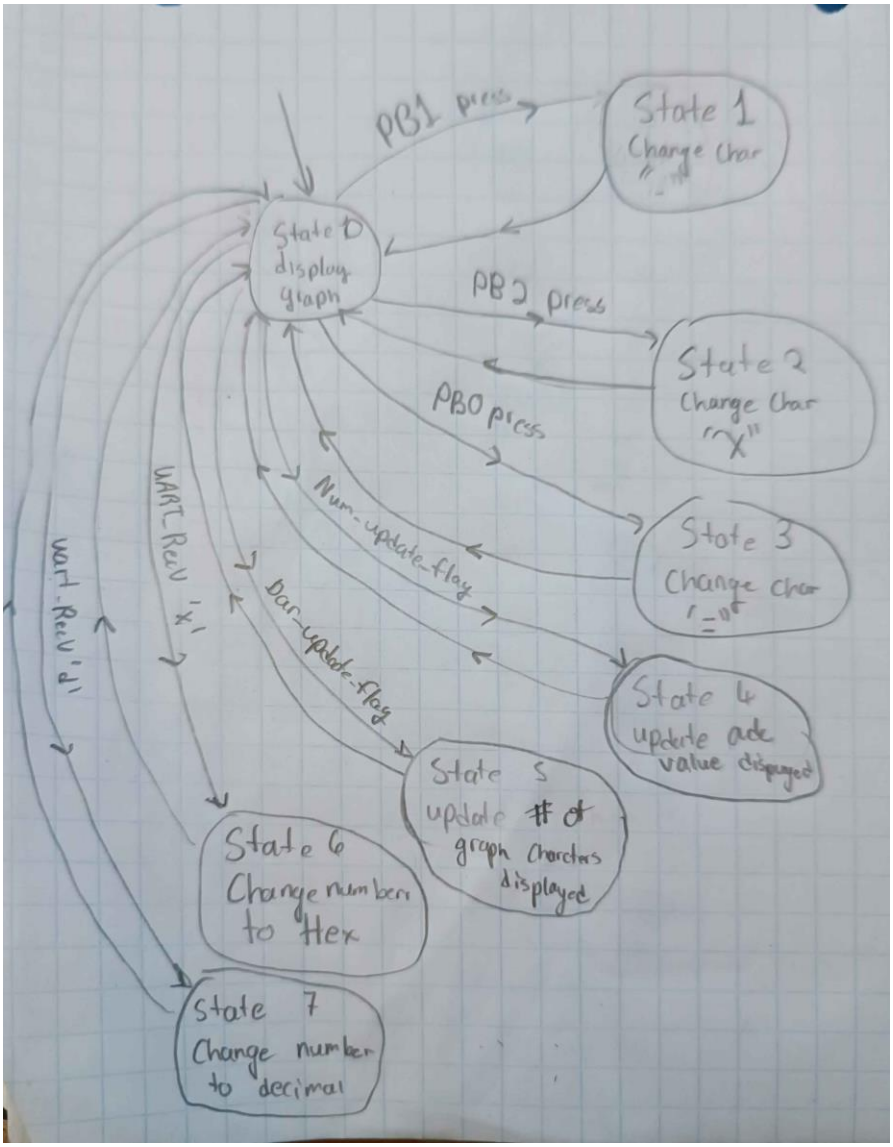
The update_bar function saves the ‘home’ position (between the graphical bar and the numerical output of the ADC value), goes to the beginning of the line and prints the selected bar_char, bar_val times the rest of the space is filled with a blank space. The cursor is then returned to the ‘home’ position.

```
void update_num(void)
{
    Disp2String("\033[s");  // save cursor position
    // transmit ADC value
    CHECK_BIT(app_flags, NUM_MODE) ? Disp2Dec(adc_reading) : Disp2Hex(adc_reading);
    Disp2String("\033[u");  // restore cursor position
}
```

The update_num function also saves the ‘home’ position, checks to see if NUM_MODE is high and displays the adc reading in decimal or if its low displays the adc reading in hex and then returns the cursor.

```
delay_ms(10);
```

At the end of all this we delay for 10ms so slow down this loop.



This is the state diagram based off the code. Each state simply changes a part of the ASCII terminal display and then it immediately goes to state 0, where it waits for an input to change.

Answers to Lab Questions

In your assignment part, how do you accept the keyboard press at any time?

Whenever a UART character is received it is immediately loaded into the U2RXREG register by hardware and an interrupt flag is set. When the interrupt occurs, the value is then loaded into the received_char variable. Then when the RecvUartChar function is called from the main loop, the program reads the value in the received_char variable.

How did you setup the ADC?

```
void adc_init(void)
{
    AD1CON2bits.PVCFG = 0;      //Selects Positive voltage reference
    AD1CON2bits.NVCFG0 = 0;     //Selects Negative voltage reference

    //Setup Channel 0 sample A
    AD1CHSbits.CH0NA = 0;       //Negative input is selected as gnd
    AD1CHSbits.CH0SA = 5;       //Positive input selected as AN5 (PIN7)

    AD1CON3bits.ADCS = 255;     //AD Conversion clock is set to 256 * Tcy
    AD1CON1bits.SSRC = 7;       //ADC occurs based off SAMP
    AD1CON1bits.FORM = 0;       //Data output is absolute decimal unsigned right justified
    AD1CON5bits.ASEN = 0;       //Disable auto scan
    AD1CON1bits.DMAEN = 0;      //Disable DMA
    AD1CON2bits.SMPI = 0;       //Interrupts at the completion of each sample
    AD1CON1bits.MODE12 = 0;     //10-bit ADC mode
    AD1CON1bits.ASAM = 0;       //Sampling starts when SAMP is set manually

    AD1CON1bits.ADON = 1;       //Enable ADC
    AD1CON1bits.SAMP = 0;
}
```

Some highlights of the init function are: we set MODE12 to 0 to select 10 bit ADC mode, set ASAM to 0 to enable sampling when the SAMP bit is manually set, and we set FORM to 0 to have the contents of the ADCBUFF to be absolute decimal values of the ADC steps and they are unsigned right justified so that the content are in the lower part of the 16bits of ADCBUFF0.