Laboratory 02 - Interrupts

By: **Caleb Probst**

Spring 2023 ECE 381 Microcontrollers

Introduction

The main idea of this laboratory was to learn how interrupts work in software and how they can be an alternative to polling. The main use of interrupts is so that the processor is not constantly checking for something, but is instead interrupted by the software and taken into a different segment of code that can then be ran. The benefit to this, is that other sequences can be ran instead of just the polling loop. The downside to this, is that the software is harder to write and some interrupts can be missed or overwritten due to all interrupts on the PSoC being on the same priority level (at least for this laboratory).

Goal

The goal of this laboratory is to make a small game that will strobe a small bar on the bottom of the LCD with a movable target bar and tell you if you win or lose. The lab should use an encoder to move the target bar right or left and use asynchronous interrupts to know when the encoder is being twisted. The moving bar should be setup on a synchronous interrupt controlled by a timer that will go off every set amount of time and move the bar. When the button is pressed, which is also on an asynchronous interrupt, should stop the moving bar and if the moving bar matched the target bar, it should present the user with a “Congratulations” message and if the moving bar does not match, it should present the user with a “You lose!” message.

Materials

PsoC4 4200M Microcontroller	1
Hitachi LCD	1
Bourns ECW1J-C28-BC0024L Rotary Encoder	1
Push Button	1
1KΩ Resistor	2
100nF Capacitor	2

Procedure and Results

Part 1 – Setup in PSoC Creator

For this laboratory, we wanted 5 main modules that would be implemented in software later. These modules include a Character LCD module, 3 GPIO inputs, a Timer Counter (TCPWM), a 1KHz clock signal, and 3 ISR’s (Interrupt Service Routines). These modules are setup as shown by *Figure 1*.

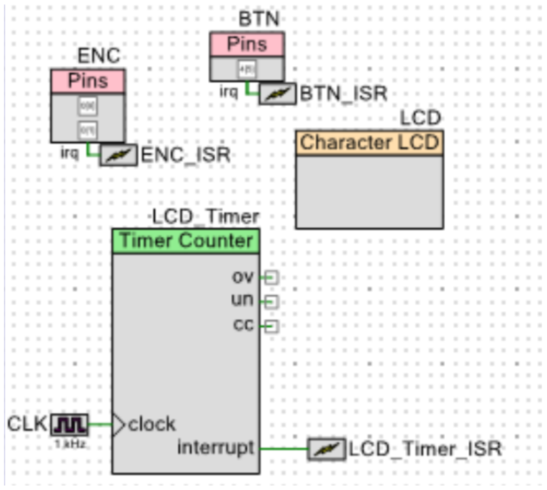

Figure 1: Top Design Layout

For the BTN input pin, we also set the interrupt to be on a rising edge, and set the drive mode to resistive pull down, so we did not use an external resistor on this design. For the ENC pins, we also set the interrupt to be on a rising edge, and set the drive mode to resistive pull down. The reason we wanted a rising edge is so that the interrupt is triggered as soon as the input is detected. We wanted the modes in resistive pull down so we did not burn up the input pins on the PSoC4. All of the interrupt types on the GPIO pins should also be on a “Dedicated Interrupt”. Shown below in *Table 1* is the pin numbers for the connections on the PSoC4. For the LCD module, ensure that the Horizontal Bargraph special character set is enabled.

ENC[1:0]	P0[1:0]
LCD[6:0]	P2[6:0]
BTN	P4[5]

Table 1: Pin Outs of Hardware on PsoC4

Part 2 – Hardware and Wiring

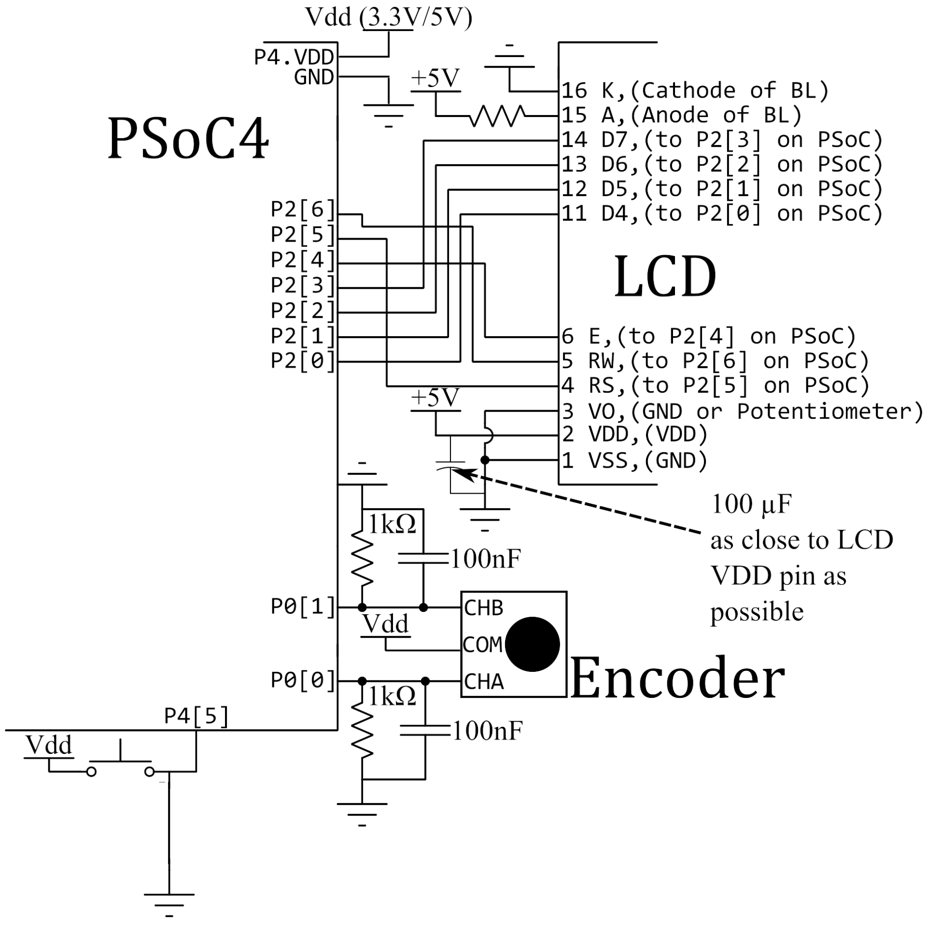

Figure 2: Wiring Diagram

Shown above by *Figure 2* is the wiring diagram for this laboratory. The resistors and capacitors on the encoder are meant to help with some of the noise from the bouncy mechanical encoder. The setup on the LCD is the same as the last laboratory, and simply just follows the pin diagram as shown. The final wiring harness for our project looks similar to *Figure 3* below.

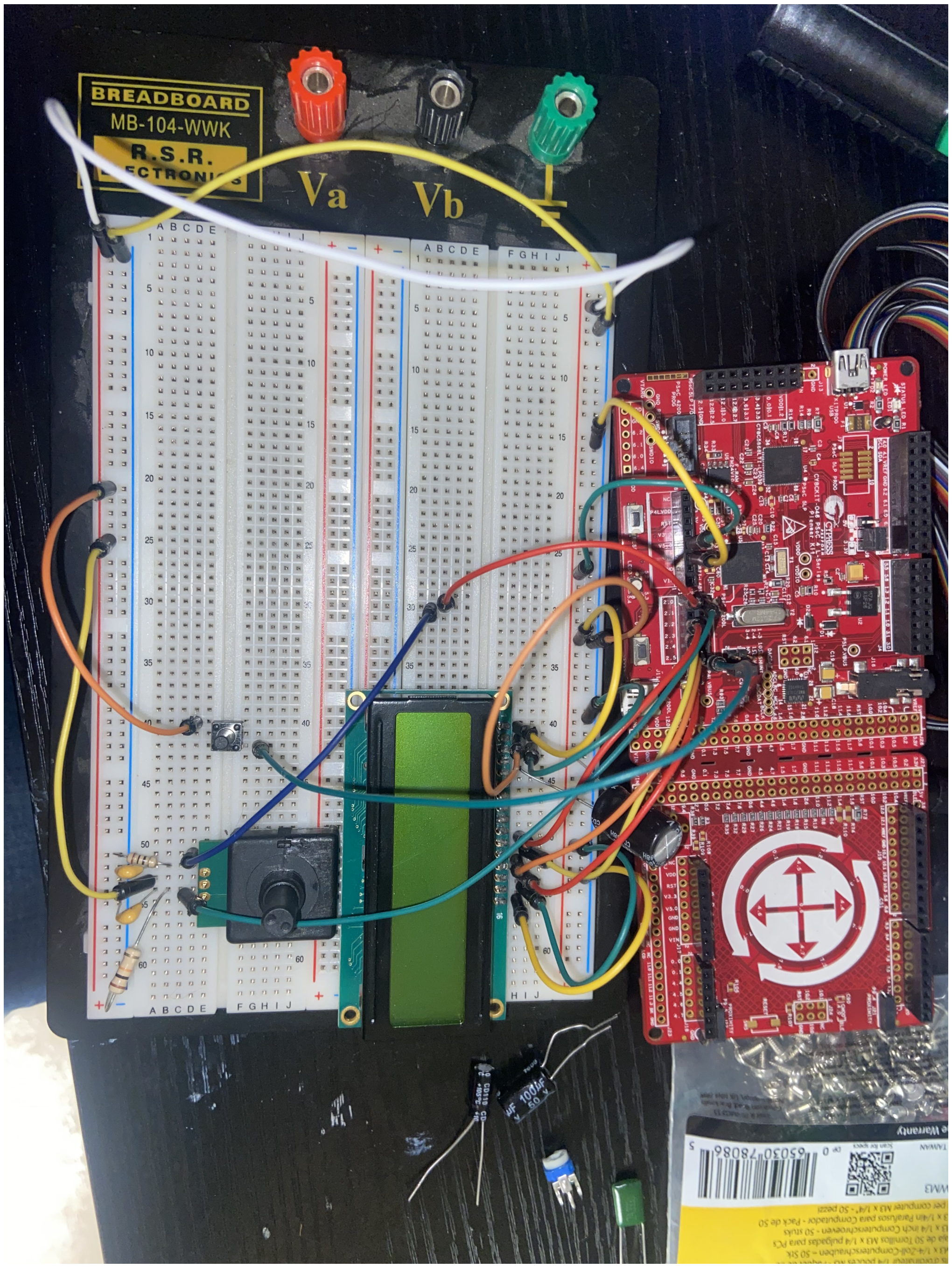

Figure 3: Physical Wiring Harness

Part 3 – Software

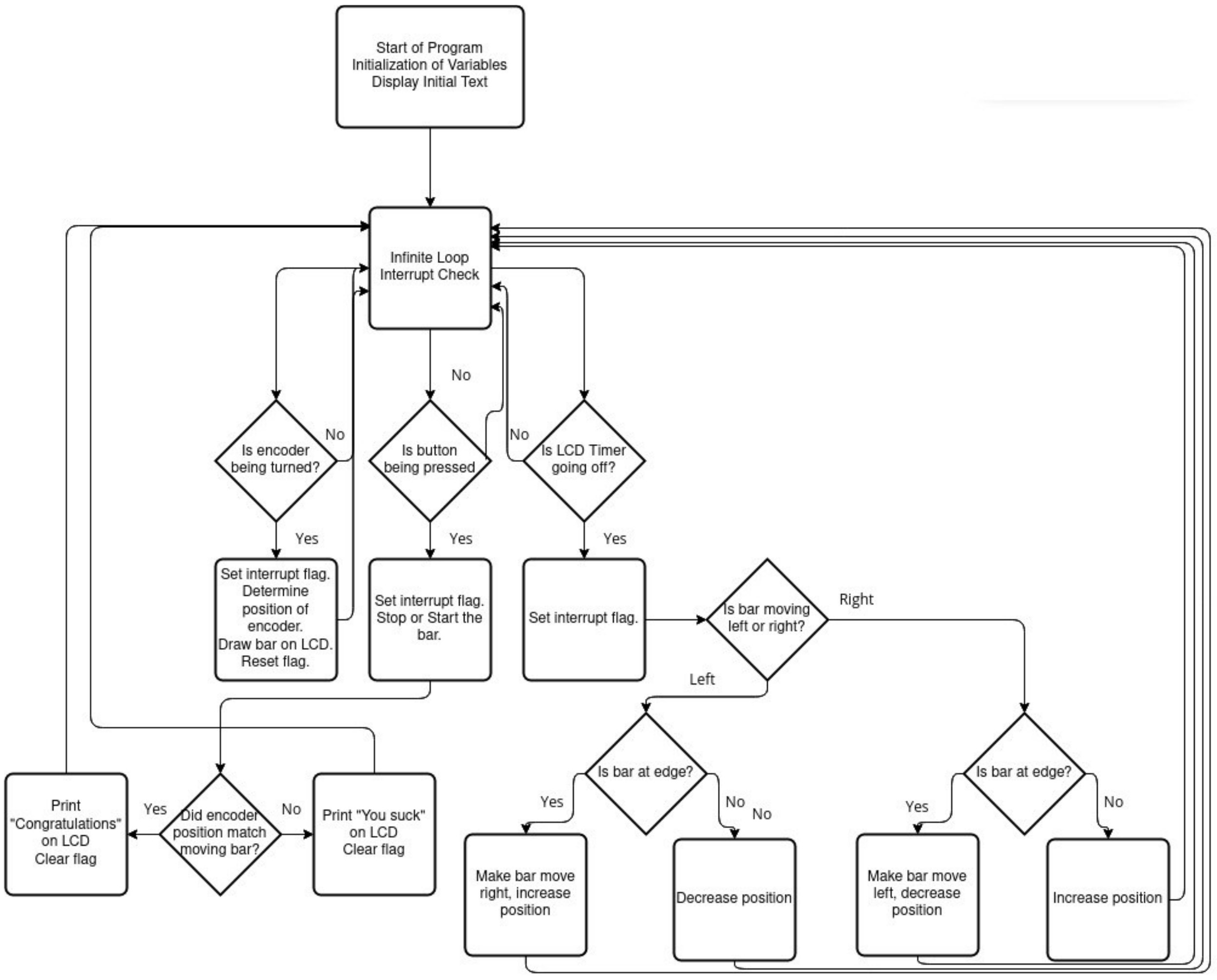
Start of Program
Initialization of Variables
Display Initial Text
Infinite Loop
Interrupt Check
Is encoder being turned?
Is button being pressed
Is LCD Timer going off?
No
No
No
Yes
Yes
Yes
Set interrupt flag. Determine position of encoder. Draw bar on LCD. Reset flag.
Set interrupt flag. Stop or Start the bar.
Set interrupt flag.
Is bar moving left or right?
Right
Left
Is bar at edge?
Is bar at edge?
Yes
No
No
Yes
No
Make bar move right, increase position
Decrease position
Make bar move left, decrease position
Increase position
Print "Congratulations" on LCD Clear flag
Yes
Did encoder position match moving bar?
No
Print "You suck" on LCD Clear flag

Figure 4: Flowchart of Program

The majority of this laboratory was done in software due to the nature of interrupts. All of the modified code (non-generated) can be found in the Appendix.

Main Code

The main code of this program was written in a few function calls and a conditional loop. The two functions that are introduced are the *drawMovingBar* and *drawMovableBar*. *drawMovingbar* is a function that simply draws the horizontal bar that is interrupted by the LCD_Timer_ISR and increments the position of the bar. Code can be found in the Appendix under *main.c*. Essentially, the function takes in the position of where it is currently and checks which direction it is supposed to be moving in. From here, it then draws the bar in its next position, and then checks whether or not it is at the end and needs to go in the opposite direction, given by the variable *movingDirection*. It then returns the position of the bar so that it can be stored in main. The other way to do this would be to use *position* as a global variable instead of as a local variable in main.

drawMovableBar is very similar, except it has no control whether on its position, which is determined by the encoder. Code can be found in the Appendix under *main.c.* This code is relatively simple, as it just draws the bar based on the variable *encoderPos* which is the position of the encoder, which is controlled by the encoder ISR.

In the actual main function of the program, is also very simple and only has some branch instructions and some other small features. The first part of the code is simply the initialization and starting of the modules. The global variables are also set to their initial values (all 0's) in this time as well. All global variables are defined as volatile integers. The keyword volatile is key here as that means the compiler will not make any optimizations to the code on that variable. This is used frequently in microcontrollers as the variables can be changed somewhere outside of our program, such as by the encoder. In the infinite loop, we then have 3 conditional statements, each of which corresponds to an interrupt flag. This code can be found in the Appendix under *main.c.* This conditional statement is very simple, as it simply looks for the flag to be set by the interrupt and then resets it and moves the bar if its supposed to be moving, by using the *drawMovingBar* function.

The next conditional looks at the asynchronous button flag, and then determines what happens. First, it resets the flag so that it can be interrupted again, and then decides if the bar is supposed to be moving using the *barMove* variable. If the bar is moving, then it stops the bar and prints the moving bar, has a small delay and then prints whether or not you won or not based on the position of both bars. If the bar is not moving, then it starts moving. Finally, it waits on the release of the button, which helps to debounce the button.

The final conditional looks at the asynchronous encoder flag, and then simply resets the flag and draws the movable bar.

BTN_ISR.c

For the button ISR, we first need to include the BTN header file so we have the functions that the GPIO pin uses. Along with that, we also need to tell the compiler to look for our variable *buttonAsync* by using the extern keyword. This links our two variables together and allows us to use them in both source files. All source code should go in between the START and END blocks. In the *CY_ISR* function, we can now write our service routine. Code can be found in the Appendix under *BTN_ISR.c.* First we have a small delay to ensure that the button is actually being pressed. We then clear the interrupt and check to see if the button is still being read as on, and then set the flag. This flag variable then can be read by our *main.c*. Note that all of the ISR code should go in between the START and END blocks of *CY_ISR.*

ENC_ISR.c

For the encoder ISR, we first need to include the ENC header file so we have the functions that the GPIO pins use. Along with that, we also need to tell the compiler to look for our variables *encoderAsync* and *encoderPos* by using the extern keyword. This links our variables together and allows us to use them in both source files. All source code should go in between the START and END blocks. In the *CY_ISR* function, we can now write our service routine. Code can be found in the Appendix under ENC_ISR.c. The first thing we do is store a variable called a and store the current state of the pins, immediately after the interrupt is sent. After that, it then waits for however long for that state to change. It then repeats that process 2 more times with b and c until we know the state of each turn. The reason that we need this is because the encoder goes through 4 states through each rotation. The states of the encoder are given by *Figure 5*.

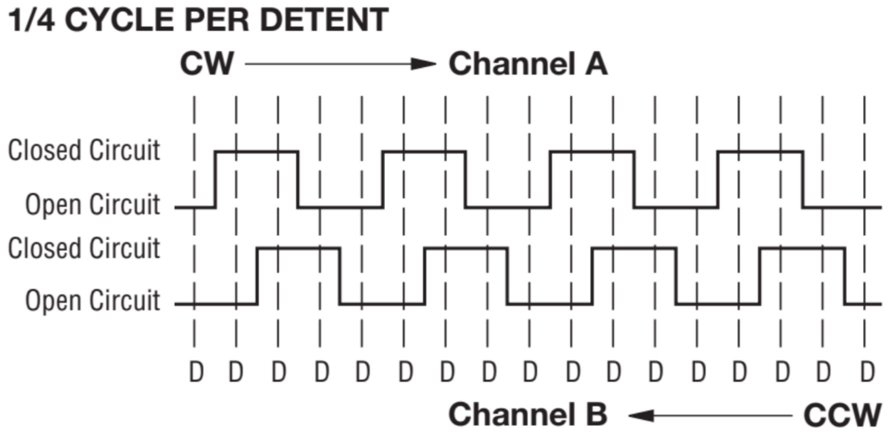

Figure 5: Encoder States

As shown, each cycle starts and ends at the same position, so we need to be able to read whether or not it is running clockwise or counterclockwise. Defining the following binary numbers as decimals (AB→ Decimal: 00 → 0, 01 →1, 10 → 2, 11 → 3), we can determine what is occurring in the encoder. If we start at the left, we start out at 0 and go through 2, 3, 1, back to 0, in that order, going clockwise. If we start at the right, we start out at 0 and go through 1, 3, 2, back to 0, in that order, going counter-clockwise. This is how we can determine the direction of the encoder. By storing each change in value, we can determine which way it is going. Since the middle value (3) is the same, we can look at the first and third value, which are stored as *a* and *c* in the program. The code can be found in the Appendix under *ENC_ISR.c*. The basic way to do this is to use a conditional to check whether *a* and *c* are 2 and 1 respectively, or 1 and 2 respectively. If it is 2 and 1, it must be moving clockwise. If it is 1 and 2, it must be moving counter-clockwise. So, using a conditional statement, we can check those conditions. If it is moving clockwise, then set the flag, check boundary and increase position. If it is moving counter-clockwise, then set the flag, check boundary and decrease position. Finally, clear the interrupt. Note that all of the ISR code should go in between the START and END blocks of *CY_ISR.*

LCD_Timer_ISR.c

For the LCD Timer ISR, we first need to include the LCD_Timer header file so we have the functions that the timer module uses. Along with that, we also need to tell the compiler to look for our variable *barSync* by using the extern keyword. This links our two variables together and allows us to use them in both source files. All source code should go in between the START and END blocks. In the *CY_ISR* function, we can now write our service routine. Code can be found in the Appendix under LCD_Timer*_ISR.c*. This code is very simple and simply clears the interrupt and sets the flag for *main.c*. Note that all of the ISR code should go in between the START and END blocks of *CY_ISR*.

Testing Methodology

For testing this device, it is very simple. First, turn the encoder and ensure that it rotates the correct direction per click. You should be able to do it quickly and slowly and it still work the same. Next, click the button and ensure that it starts and stops, and when it stops, it should have a small delay and display some text saying whether or not the user won or lost. Finally, if the user wins, it should display "Congratulations" and if the user lost, it should display "You suck!!!".

Conclusion

This laboratory was difficult to complete, and we had ran into multiple issues. First, the encoder we were using, we didn't know how it operated properly. We had to hook up an oscilloscope and found out that all 4 positions were being rotated through per detent. The other issue we had was in the software, we had a race condition as we were using an extended-if statement, instead of 3 separate if statements.

Appendix

main.c

```
#include "project.h"

// Interrupt flags and other variables needed
volatile int barSync;
volatile int buttonAsync;
volatile int encoderAsync;
volatile int encoderPos;
int movingDirection;


int drawMovingBar(int position){
    // This will draw the moving bar that is based on the synchronous
timer
    // Checks which direction it is moving in 0 = right, 1 = left
    if(movingDirection == 0){
        position++;
    }
    else{
        position--;
    }
    // Draw bar
    LCD_DrawHorizontalBG(1,0,16,position);
    // Checks position of bar, changes moving direction if needed
    if(position == 80 || position == 0){
        if(movingDirection == 0){
            movingDirection = 1;
        }
        else{
            movingDirection = 0;
        }
    }
    // Returns position to update in main
    return position;
}

void drawMovableBar(){
    // Draws bar based on encoder position, same as last
    LCD_DrawHorizontalBG(0,0,16,encoderPos);
}
int main(void)
{

    /* Place your initialization/startup code here (e.g.
MyInst_Start()) */
    // Start modules and timers
    LCD_Start();
    ENC_ISR_Start();
    BTN_ISR_Start();
    LCD_Timer_Start();
    LCD_Timer_ISR_Start();
    /*
    Write period of timer, we picked 12 because our timer frequency
was
    1 KHz, so this would make the bar move one position every 12ms.
    The reason we wanted 12 ms, was because it goes 80 positions
    which is approximately 1 second to get to one side of the LCD.
    */
    LCD_Timer_WritePeriod(12);
    // Setting global variables to initial positions
    barSync = 0;
    buttonAsync = 0;
    movingDirection = 0;
    encoderAsync = 0;
    encoderPos = 0;
    // Set local variables for main
    int barPos = 0;
    int barMove = 1;
    // Enable interrupts
    CyGlobalIntEnable;
    // Print initial message on LCD (40x2 LCD)
    LCD_PrintString("Turn the dial,                          press
button");
    for(;;)
    {
        /* Place your application code here. */
        // Synchronous timer for moving bar
        if(barSync == 1){
            // Reset flag
            barSync = 0;
            // Move bar forward if bar is supposed to be moving
            if(barMove != 1){
                barPos = drawMovingBar(barPos);
            }
        }
        // Asynchronous button interrupt for stopping bar
        if(buttonAsync == 1){
            // Reset flag
            buttonAsync = 0;
            // If bar is supposed to be moving, stop bar and decide
            if(barMove == 0){
                // Stop bar
                barMove = 1;
                // Clear display and print both bars
                LCD_ClearDisplay();
                barPos = drawMovingBar(barPos);
                drawMovableBar();
                // Half second delay in between bar print and string
print
                CyDelay(500);
                // Clear display and print top bar and string
                LCD_ClearDisplay();
                drawMovableBar();
                // Decide if win or not
                if(barPos == encoderPos){
                    // Win
                    LCD_PrintString("
Congratulations!");
                }
                else{
                    // Lose
                    LCD_PrintString("                        You
suck!!!!!!!!");
                }
            }
            else{
                // If bar is not moving, start bar moving
                barMove = 0;
            }
            // Debounce button
            while(BTN_Read() != 0);
        }
        // Asynchronous encoder interrupt
        if(encoderAsync == 1){
            // Reset flag
            encoderAsync = 0;
            // Draw bar
            drawMovableBar();
        }
    }
}
```

BTN_ISR.c

```
/* `#START BTN_ISR_intc` */
#include "BTN.h"
// Take global variable from main.c
extern volatile int buttonAsync;
/* `#END` */
...
...
CY_ISR(BTN_ISR_Interrupt)
{
    #ifdef BTN_ISR_INTERRUPT_INTERRUPT_CALLBACK
        BTN_ISR_Interrupt_InterruptCallback();
    #endif /* BTN_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START BTN_ISR_Interrupt` */
    // Small delay for debouncing
    CyDelay(20);
    // Clear interrupt
    BTN_ClearInterrupt();

    // If button is still 1 after delay, set flag, if not, don't
    if(BTN_Read() == 1){
        buttonAsync = 1;
    } else {
        buttonAsync = 0;
    }
    /* `#END` */
}
```

ENC_ISR.c

```
/* `#START ENC_ISR_intc` */
#include "ENC.h"
// Take variables from global space (main.c)
extern volatile int encoderAsync;
extern volatile int encoderPos;
/* `#END` */
...
...
CY_ISR(ENC_ISR_Interrupt)
{
    #ifdef ENC_ISR_INTERRUPT_INTERRUPT_CALLBACK
        ENC_ISR_Interrupt_InterruptCallback();
    #endif /* ENC_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START ENC_ISR_Interrupt` */
    // Read the first variable, then wait for it to change
    volatile int a  = ENC_Read();
    while(a == ENC_Read());
    // Read next variable, then wait for it to change
    volatile int b  = ENC_Read();
    while(b == ENC_Read());
    // Read next variable, then wait for it to change
    volatile int c  = ENC_Read();
    while(c == ENC_Read());
    // Based on the following variables, decide whether it is CW or
CCW
    if(a == 1 && c == 2){
        // Clockwise (increase bar)
        // Set flag
        encoderAsync = 1;
        // Set limit of 80 on the right side
        if(encoderPos == 80){}
        // If not at limit, increase position
        else{
            encoderPos++;
        }
    }
    else if(a == 2 && c==1){
        // Counter-Clockwise (decrease bar)
        // Set flag
        encoderAsync = 1;
        // Set limit of 0 on the left side
        if(encoderPos == 0){}
        // If not at limit, decrease position
        else{
            encoderPos--;
        }
    }
    // Clear interrupt
    ENC_ClearInterrupt();
    /* `#END` */
}
```

LCD_Timer_ISR.c

```
/* `#START LCD_Timer_ISR_intc` */
#include "LCD_Timer.h"
// Take global variables from main.c
extern volatile int barSync;
/* `#END` */
...
...
CY_ISR(LCD_Timer_ISR_Interrupt)
{
    #ifdef LCD_Timer_ISR_INTERRUPT_INTERRUPT_CALLBACK
        LCD_Timer_ISR_Interrupt_InterruptCallback();
    #endif /* LCD_Timer_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START LCD_Timer_ISR_Interrupt` */
    // Clear interrupt
    LCD_Timer_ClearInterrupt(LCD_Timer_TC_INTERRUPT_MASK);
    // Set flag
    barSync = 1;

    /* `#END` */
}
```