ECE381 – Lab 2 – Interrupts

Objectives:

- Learn how to program using interrupts to handle events
- Learn asynchronous interrupts using GPIO with buttons and a rotary encoder
- Learn synchronous interrupts using a Timer module
- Make a fun game in the process

Introduction:

In this lab, we are going to use interrupts to play a timing based target game. You will use the encoder to set the length of a bar on the top row of the LCD. You will then use a Timer and a button to make a second bar on the bottom row of the LCD expand and contract. The goal is to hit the button at the exact moment the bar in the bottom row is the same as the bar in the top row.

Background:

Interrupts

What is often desired for microcontrollers is for a particular function (or instruction sequence, like, say, toggling an LED) to execute when a particular event occurs (like, say, a button press). Polling achieves this using software loops, but is inefficient as it requires 100% of the CPU cycles to be spent executing this check, precluding power saving and potentially even missing other events. Further, for synchronous events at specified timing intervals, pre-programmed delay loops are prone to miscomputation, and are also inefficient in terms of power draw, as they too require the CPU to expend lots of energy in a loop that does nothing. What would be far better would be a *hardware* mechanism to detect such events without involving the CPU until the actual event occurred.

Congratulations, we have just developed an *interrupt*! Interrupts are a hardware mechanism for peripherals to signal to the CPU that an event of a type detected or generated by that peripheral has occurred. This *Interrupt Request (IRQ)* signal to the CPU has it stop its current execution path, save the current execution state (internal registers), usually by pushing them onto the stack, then jump to the *Interrupt Service Routine* (function/instruction sequence you want to execute for that event) by loading the Program Counter (PC) from the corresponding *Interrupt Vector Table* (known hardware location associated with an interrupt source) for that peripheral. Once the ISR is finished, the CPU may then return to normal execution by popping the previous internal registers back off the stack and restoring the PC to the instruction location at the time that IRQ was signaled.

Internally, most processors have a global interrupt enable bit. The ARM does, and has a corresponding opcode (`CPSIE i`) to set it or clear it (`CPSID i`) which acts like a global on/off switch for *any* possible interrupt source. Hardware peripherals typically have their own on/off bits, called *Interrupt Masks*, which must be set for that particular interrupt source to be able to signal an IRQ to the processor. A select few important interrupts are *unmaskable*, like Reset for example, that can't be turned off with either bit. Finally, most peripherals also have an associated *Interrupt Flag Bit* as well,

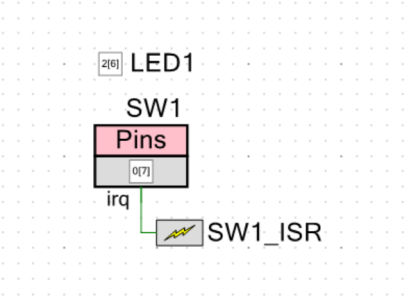
which goes high to signal that the peripheral's configured interrupt event has occurred. These flag bits typically must be cleared to allow another interrupt of that type to occur again. Peripherals do this differently, and HALs implement these differently too (sometimes automatically, sometimes manually, sometimes by reading a register, sometimes by writing a 1 to the flag bit to clear it), so consult your microcontroller's documentation to see how to do it for your use case.

Interrupts provide numerous benefits over polling. They provide a one-to-one matching between event source and code execution, allow for the processor to stop execution until an interrupt occurs to save power, and through pre-emption (interrupt priorities) and bounded execution times, can even allow for 100% provably reliable systems! No wonder we will learn about them for this lab!

Part 1 – Basic GPIO Interrupts

Configuration:

- Place a "Digital Input Pin" onto the schematic. Right-click and select "Configure". Rename the pin to SW1. Uncheck "HW connection". Change the Drive Mode to "Resistive pull up". Click on the Input tab and change the Interrupt to "Falling Edge". Click the box for "Dedicated Interrupt". Click OK.
- You should now see an external pin on the SW1 module titled irq. This is the interrupt request pin that goes high based on what you configured as the pin's interrupt condition (Falling Edge from above). Now, from the Cypress Component Catalog, expand System, grab the Interrupt and drag it into the schematic window. Connect the iqr pin on SW1 to the input of the Interrupt module. Right-click on the Interrupt module and select "Configure". Rename the module to SW1_ISR, make sure the InterruptType is DERIVED, and click OK.
- Place a "Digital Output Pin" onto the schematic. Right-click and select "Configure". Rename it to LED, uncheck "HW Connection", and click OK. Your schematic should look approximately like this:



- Open the Pins configuration window and assign SW1 to P0[7] and LED1 to P0[6], P2[6], or P6[5] for whichever color you want to toggle.
- Generate the Application (Build → Generate Application)

Software

- Open "main.c". Make sure CyGlobalIntEnble; is NOT commented.
- Under the line "/*Place your initialization...", add `SW1_ISR_Start();` to enable the ISR module. That's it, all other code will be handled in the ISR.
- On the left side, under "Generated Source/PSoC4/SW1_ISR" open "SW1_ISR.c". This contains the actual interrupt service routine (ie. the code that is run when the switch is pressed).

- SW1_ISR.c has lots of stuff in it. Find the function CY_ISR(SW1_ISR_Interrupt) (~line 160). In this function, there are two lines that are commented out:

```
/*    `#START SW1_ISR_Interrupt` */

/*    `#END`     */
```

  Do NOT mess with these lines!!! The compiler has scripts that look for them to figure out where your actual ISR is located. The code you want to run when the interrupt happens goes in between these two comments.
- We want to toggle the LED, but at this point, we can't really do that by name inside the ISR. Why? Because all the code that was generated for our port named LED resides under "Generated Source/PSoC4/LED1", which is separate from "Generated Source/PSoC4/SW1_ISR" (and also "Generated Source/PSoC4/SW1") since they are all separate modules. So, to include all of this code, we need to put #include directives for it in SW1_ISR.c. Go to the top of the file (~ line 27), and there should be some lines that look like this:

```
/*************************************************************
*
*  Place your includes, defines and code here
*************************************************************
/
/* `#START SW1_ISR_intc` */

/* `#END` */
```

  Like before, in the lines between the #START and #END comments, put:

```
#include "LED1.h"
#include "SW1.h"
```

  Save, and now all of the functions/defines/etc. for both the LED1 and SW1 modules should be accessible.
- Go back to the CY_ISR() function inside SW1_ISR.c. To toggle the LED, you can simply use 1 line of code, placed in the lines between the #START and #END comments.

```
LED1_Write(~LED1_Read());
```

  This is not all that is required, though. In order to receive another interrupt on Port3 (the whole port, not just pin 0 or 1!), we have to clear the current interrupt. Do so by placing this line of code, either right before or right after the LED1_Write():

```
SW1_ClearInterrupt();
```

- You can now build and program the device. Pressing SW1 should toggle the LED that is on the board. You don't have to demo this, but if you have issues let me or the TA know. I just want to get you used to the way PSoC Creator implements interrupts.

Part 2 – Asynchronous (Encoder) and Synchronous (Timer) Interrupts

a) Asynchronous Interrupts Using an Encoder

Rotary encoders are rotary encoders whose output is not a fixed, absolute position but rather a repeated binary pattern. Electro-mechanical ones based on making physical connections with a common wheel are very common. An optical encoder is another type of rotary encoder (also known as a shaft encoder) that converts the angular position of a shaft to a digital code. Rather than using electro-mechanical contacts to indicate the shaft position, it shines the light from an LED onto a slotted disk. On the opposite side of the disk is a photo sensor (usually a phototransistor) that turns on when the light from the LED is shining through one of the slots on the disk. The signal from the phototransistor is then conditioned into logic voltages suitable for the particular application. If only one optical sensor is used, the encoder is suitable as a tachometer or position sensor in applications where the shaft turns in only one direction. See Figure 1 for an illustration.

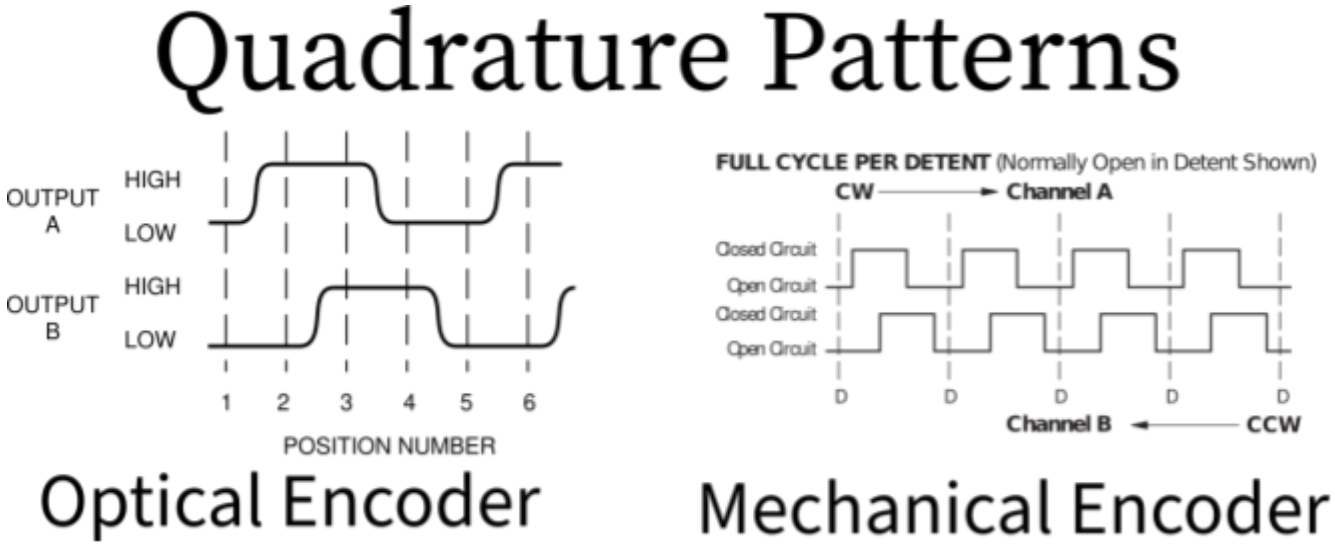


Figure 2: Output Quadrature Patterns

If an application requires sensing the rotation of the shaft in both the clockwise (CW) and counterclockwise (CCW) directions, at least two optical sensors are required. Encoders of this type are referred to as *quadrature encoders* since they give 2-bits ($2^2 = 4$ or *quad*) of information about the shaft position. These two channels produce digital signals that are phased 90-degrees apart. When looked at as bit pairs, they form a 2-bit Gray code, meaning that only one bit changes at a time (e.g. 00, 01, 11, 10, 00, 01, …). This is essential to prevent a race condition for the device interfaced to the encoder. For example, suppose the encoder changed from 11 to 00. Since both channels cannot simultaneously go low, the device interfaced to it could ambiguously read this as (11, 01, 00) or (11, 10, 00). Figure 2

shows the quadrature digital output signals of the Bourns encoder we will use in this lab. The signals are shown with the encoder shaft rotating in the clockwise direction.

Quadrature Encoder Applications

Because they provide information about the direction, position, and velocity of a rotating shaft, rotary encoders are used in many applications requiring feedback about motor driven systems, with robotics being a prominent area.

Another common application for rotary encoders is in user interfaces. When given a knob and some push button, they allow a user to select among a list of options in a menu. Since they only report relative position and not absolute position (like a potentiometer), they allow an application to start from a reasonable set of defaults.

We will use a rotary encoder in this lab to control a color wheel using the multi-color LEDs. You will detect the direction of the encoder in the ISR, however you will update the LED and LCD display in main() using a flag set in the ISR. This is due to the fact that updates to the LCD can take milliseconds of time, and can cause missed encoder edges if left in the ISR. The LCD should *never be continually updated* and should only update once per click.

Hardware:

Everyone should have a Bourns ECW1J-C28-BC0024L rotary encoder for the lab. **IF YOU DON'T HAVE ONE, LET ME OR THE TA KNOW!**. Also, note, while the Bourns ECW1J outputs a quadrature pattern, the

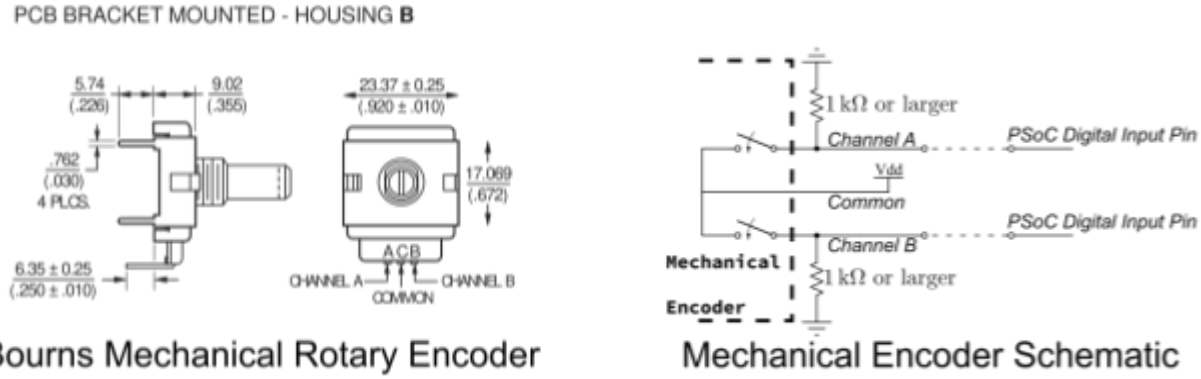
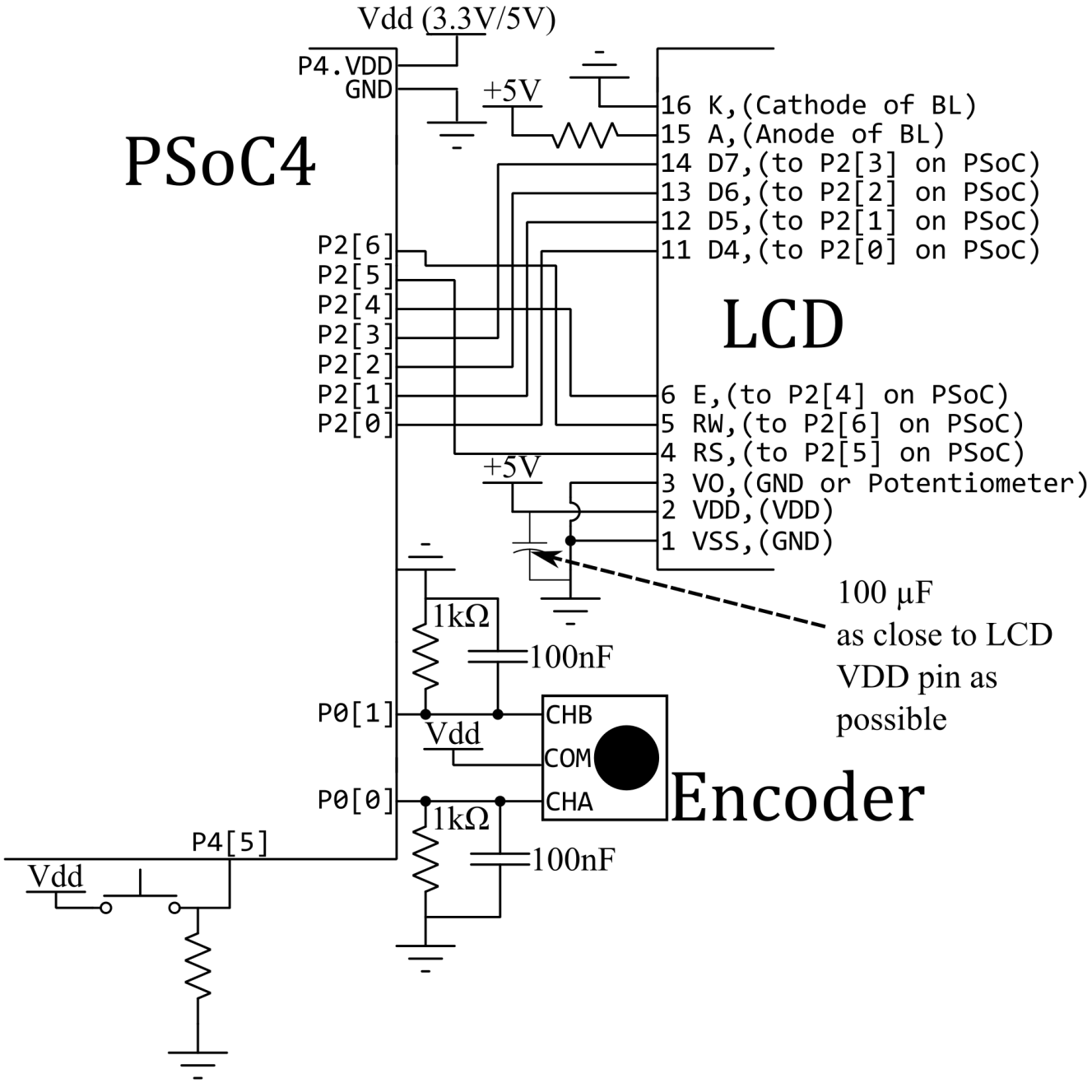


Figure 3: Encoder Schematic & Pinout

detents (clicks) are only every fourth state. As an example, after one click clockwise, the A and B values will still be 0 and 0, however, the actual voltage transitions will be 00 → 01 → 11 → 10 → 00. So, if you are testing, and your count always increments or decrements by 4 using the mechanical rotary encoder, this is the reason. You can fix this in software if that is the case (ie only count every 4 transitions or only detect one transition for every 4) or change the interrupt condition to only count up or down once per detent. The amount of time in between these edges is variable, as it's a mechanical device and depends on how hard/fast you torque it and the internal switching mechanism. Conservatively, the amount of time spent in each of the four states it goes through per detent is likely to be around 400-500 microseconds at the fastest, though it could easily end up being over 1 millisecond.

The key takeaway from this is to not make your ISR take too long (Shorter debounce delays if you have any in software, and no direct LCD updates). To make things go a little more smoothly, add an external debounce of a 1kΩ with 100nF (0.1µF) in parallel to help with the noisy encoder. Also, we are adding a decoupling capacitor (100µF) to the power pin of the LCD, as the encoder might cause a voltage sag when clicked. The final hardware diagram looks like this:



b) Synchronous Interrupts Using Timers

Often in microcontroller based programs, it is necessary to have the microcontroller do some repetitive, periodic task. While in some cases, a simple delay loop can suffice, delays are sub-optimal as then the

microcontroller is unable to process other events during the delay, nor is the timing truly guaranteed if there are other interrupts! Use of a hardware timer solves these issues, and hardware timers are found nearly universally on any microcontroller. Most operate by some combination of an input clock, programmable dividers (sometimes called prescalers) for this clock, a counting register, and some kind of period value. The counting register counts up/down on every clock edge, and when it hits 0/period, the timer triggers an interrupt. Most timers also include the ability to capture events by storing the current value of the count to a register when such a capture event occurs, for example on a rising pin edge. Input capture can be useful for measuring precise time between external events.

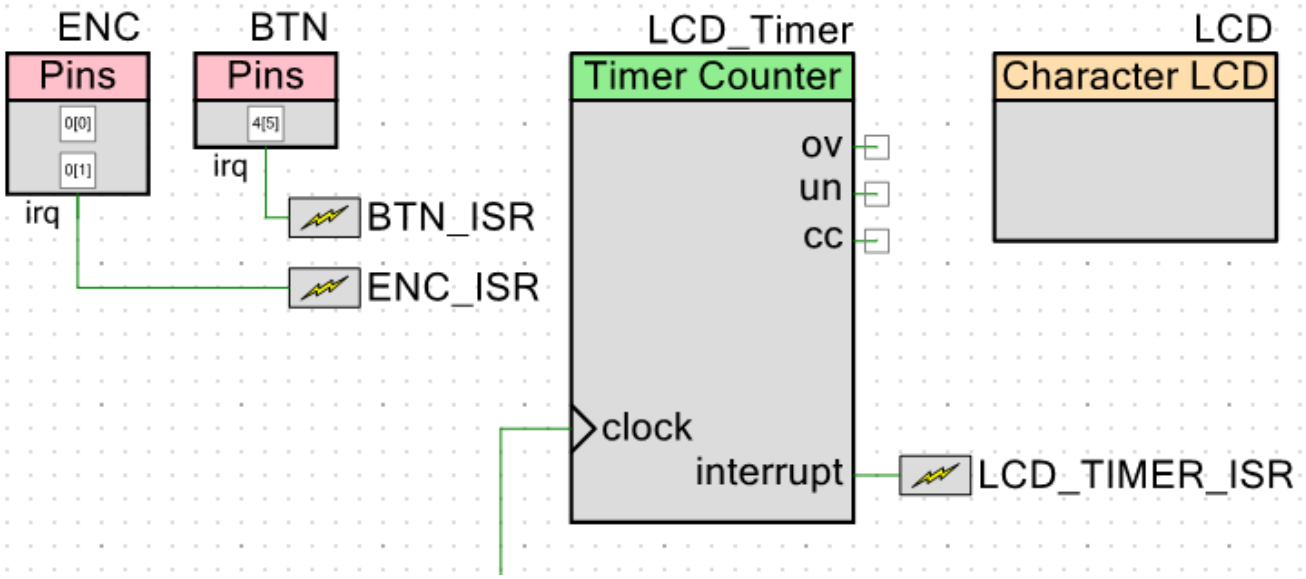
On the PSoC, the Timer module accomplishes this for us. We feed it with a clock whose target frequency we input (chosen by PSoC Creator from the PSoC clock system). On each rising edge of this clock, an internal *COUNTER* register increments/decrements by one. When this *COUNTER* register hits 0/*PERIOD* (terminal count), the module can trigger an interrupt. The *COUNTER* register then reloads with 0 or the value stored in the *PERIOD* register, and the whole process repeats. This process thus produces interrupts at regular intervals (or *synchronous* interrupts). For example, suppose the clock input to the Timer is set to 1 MHz, and the timer has a value of 99 stored in the *PERIOD* register. Then the interrupt would trigger every (0.000001 s)*(99+1) = 0.0001s or at 10 kHz. The `Timer_ReadPeriod()`, `Timer_WritePeriod()`, `Timer_ReadCounter()`, and `Timer_WriteCounter()` functions allow checking and modification of these in our code (NOTE: you almost never need to actually read the *COUNTER* register, you just wait for it to go to 0/*PERIOD* and trigger the ISR. In some circumstances you might need to write to it, especially to reset the *COUNTER* to 0/*PERIOD* manually to avoid triggering an ISR, but even that isn't very common).

In this assignment, we are going to use the Timer to sweep the bottom bar graph from blank, to full, to blank again. The speed of the timer will control how quickly the bar graph is drawn, and thus, the difficulty of the game.

Configuration:

- Connect Channel A & Channel B of your encoder to P0[0] and P0[1] respectively.
- Connect a push-button to P4[5]. (It can't be on Port0, since the encoder will be using that interrupt vector)
- Place a "Digital Input Pin" onto the schematic. Right-click and select "Configure". Rename the pin to ENC. Set the number of pins to be 2. Uncheck "HW connection". Click on the Input tab and change the Interrupt to "Either Edge", "Rising Edge", or "Falling Edge", depending on how you implement your Encoder detection logic. Make sure you check "Dedicated Interrupt". If you are not using an external pull-down resistor, change the drive mode to be "Resistive Pull-Down" for BOTH pins. Click OK.
- From System, grab the Interrupt and drag it into the schematic window. Connect the irq pin on ENC to the input of the Interrupt module. Right-click on the Interrupt module and select "Configure". Rename the module to ENC_ISR, make sure the InterruptType is DERIVED, and click OK.
- Place a "Digital Input Pin" onto the schematic. Right-click and select "Configure". Rename the pin to BTN. Set the number of pins to be 1. Uncheck "HW connection". Click on the Input tab and change the Interrupt to "Rising Edge", or "Falling Edge", depending on what kind of pull-up/pull-down resistor you use. Make sure you check "Dedicated Interrupt".

- From System, grab an Interrupt and drag it into the schematic window. Connect the irq pin on BTN to the input of the Interrupt module. Right-click on the Interrupt module and select "Configure". Rename the module to BTN_ISR, make sure the InterruptType is DERIVED, and click OK.
- Place an LCD module, right-click configure, make sure to check the "Horizontal Bar Graph" option for the custom character set, check "Include ASCII to Number Conversion", and rename it to "LCD". Configure the LCD module to use P2[6:0] as before.
- Place a Digital/Timer Counter (TCPWM) module. Right-click to configure, and name it LCD_Timer. Click on the Timer/Counter tab. You can configure the Period later using the HAL, just make sure that the "Interrupt, On terminal count" box is checked. Grab another "System/Interrupt" and connect it to the *interrupt* pin of the Timer. Rename it something like LCD_TIMER_ISR.
- Your schematic should look approximately like this:



Software:

- Don't forget to initialize your modules with LCD_Start() and ENC_ISR_Start() in main.c
- Place your code for detecting the direction and setting the update flag in the CY_ISR() function inside ENC_ISR.c. Don't forget to software debounce inside your ISR.
- You will also need to use the ENC_ClearInterrupt() inside your ISR, so you will also need to #include "ENC.h" as well
- You should only update the LCD in main with a flag variable set in the various ISRs. LCD functions take milliseconds, so you will potentially miss interrupts if you attempt to update the LCD directly in any ISR. ISRs block other ISRs by default unless you mess around with priorities, which should be unnecessary.
- You can use the LCD Bar Graph API for rendering the top row of the LCD. The LCD_DrawHorizontalBG() function takes the row, column position as the first two arguments, the max number of *whole* characters for the graph (16 for a whole row) as the 3rd, and then the

total number of *pixels* to draw. If you use the whole first row, there are 16 characters, each being 5 pixels wide, for a total of 80 pixels for the bar graph at maximum value. Your challenge game should match on the pixels, not whole squares.

- Don't forget to start the LCD_Timer and LCD_TIMER_ISR
- Don't forget to include an `LCD_Timer_ClearInterrupt(LCD_Timer_TC_INTERRUPT_MASK)` somewhere in your LCD_TIMER_ISR routine.

Requirements

- On reset, you should allow the user to set the target bar size by turning the encoder CW or CCW. A CW turn should increase the bar size by one pixel, to the maximum number (80), and a CCW turn should decrease the bar size by one pixel, to the minimum (0)
- The initial position of the target bar graph should be in the middle (40) for ease of use
- Once the button is pressed, the target bar size will remain fixed until system reset, so you can do this outside of the infinite for loop if needed
- The moving bar should increase by one pixel on the second row of the LCD, starting at 0, every LCD_Timer interrupt. When it reaches the maximum number of pixels in the second row (80), it should then decrease by one pixel every Timer interrupt, until reaching the minimum (0), then repeating.
- When the button is pressed to freeze the moving bar graph, if it hits the target, a congratulatory message should be displayed on the bottom row of the LCD. If it misses the target, an insulting message should be displayed. Pressing the button again should restart the bottom row moving again.
- There is no strict requirement on how fast/slow your Timer has to be. Too fast, and you will likely skip pixels during updates, as the LCD will be too slow to refresh, which isn't fair. Too slow, and it will be too easy and painful. You will be required to explain your exact interrupt rate.
- You should never have ANY LCD updates in ISRs!!! Move them outside to main and use flag variables.