Lab 4 – Bouncing LEDs

Introduction

In this lab, you will start to work with microcontrollers in C. You will learn about timers and interrupt service routines (ISRs or interrupts for short), which are used heavily in embedded systems. This lab highlights two typical applications: reading sensors and triggering output.

Reading

- K&R chapters 1.10
- [optional] Fly24 chapter 5 – Interrupts

Provided files

- bounce.c – contains main() and other setup code. There is no support for user input or output in this lab. Debugging will require the use of breakpoints.

Assignment requirements

- Your program should light a single LED (D3-D10) at a time on the development board, starting in one direction and reversing upon reaching the end, in essence bouncing off the ends of the row of LEDs.
  - Define a struct called "bounceState" to hold the counter, speed, and direction (as char, int, char respectively). This struct will be used in both main() and the ISRs.
  - Declare a single global variable that is an instance of the bounceState struct. The interrupts will be modifying that struct to store the necessary values.
  - Define the constants TRUE, FALSE, LEFT, and RIGHT and use them with your while-loop and for the direction member of the bounceState struct.
  - Only update the LEDs every 10 counter increments by checking it in main().
  - Map the 10-bit ADC input into 3 bits.
  - Use this 3-bit value generated from the ADC input to alter the timer preloader register upper-order bits, speeding up the light as the input gets higher.
• Implement the Timer1 ISR
  o Update a bounceState struct’s counter.
• Implement the ADC1 ISR
  o Read the current ADC value and update the speed member of the bounceState struct.
• Add inline comments to explain your code.
• Fill in the block comments above each of the ISRs with descriptions of when they’re triggered and (briefly!) what they do.
• Add the following to the top of your bounce.c file as comments:
  o Your name
  o The names of colleagues who you have collaborated with
• Format your code to match the style guidelines that have been provided.
• Make sure that your code triggers no errors or warnings when compiling. Compilation errors will result in NO CREDIT. Any compilation warnings will result in an additional lost point.
• Submit bounce.c via eCommons before the due date.

Grading

This assignment again consists of 10 points:
• 4 points – Properly implementing the interrupts
• 4 points – Making the LEDs bounce
• 2 points – Adhering to the style and variable naming guidelines (this means both variable capitalization and naming)

You will lose points for the following:
• -10 points: code doesn’t compile (seriously)
• -1 point: any compiler warnings
• -1 point: not using a struct as described in the requirements
• -1 point: the files you submit aren’t named as described in this document or you submit more than just the required documents
• -1 point: if gotos were used
• -2 points: more than a single bounceState global was declared

Program flow

Your ISRs will trigger asynchronously as your main() runs, updating state variables. Check these variables from the loop in main() and light the LEDs appropriately.

While (TRUE)
  if counter >= 10
    If next LED should be triggered
      Trigger next LED
If we're at the last LED
Reverse direction

Program input/output

This program uses some of the hardware features of the Proteus simulator, specifically a potentiometer and a row of 8 LEDs. You have used the LEDs in a previous lab, but the potentiometer is a new input. Their locations are highlighted below.

On the left of the potentiometer are two arrows: up and down. These change its value and increase and decrease respectively the output of the ADC (described later).

Interrupts

CPUs and microcontrollers handle interrupts continuously, as they are generally a part of normal operation. Many involve the internal operation of the CPU while others are triggered externally.

These external interrupts are used for processing data arriving to the processor. This means declaring an interrupt service routine (ISR) to the processor and it will be called by the processor itself when its corresponding interrupt triggers. For the PIC24F ISRs are declared similar to regular C functions:

```c
void __attribute__((__interrupt__, __auto_psv__)) T1Interrupt();
```

The void return type and the function name and parenthesis on the far left and right should look familiar, but what's that gibberish in the middle? That's just how the PIC24F declares its interrupts and you may safely ignore it for the purposes of this lab. Note that it returns nothing and takes no arguments. Since your code doesn't call these functions directly, there is no way to pass arguments or process the return value. So
how does the ISR do anything? The most common method for getting data into and out of an ISR is to declare variables globally: outside of any functions or curly-braces (look for the comment above main()).

Asynchronous execution

One thing that you cannot safely ignore in this lab is that ISRs are called *asynchronously*. This is the term for what was described above, where the processor could call your code whenever it deemed it appropriate. Since your code within an ISR can be called at any time it is important to keep that in mind.

So what happens if the processor is already within an ISR? The PIC24F processors use an interrupt flag to prevent this from happening. This flag is set (to 1) when the ISR is called to prevent other interrupts from being called. It is therefore important that this flag be cleared (to 0) before the interrupt finishes or it will never be called again!

For this reason interrupts that take a while to execute will cause problems; with the interrupt flag set, neither other interrupts nor the rest of your code can run. Perform slow operations (like complicated function calls such as printf()) only within the synchronous portions of your code.

LATA

The LEDs D3-D10 are controlled by configuring pins on the processor: high for turning them on and low for turning them off. These LEDs are all connected to the A-pins and so are controlled by the LATA register and the LATAbits struct. The LATA register is used to configure all the LEDs simultaneously; the LATAbits struct is used to configure the LEDs individually.

Configuring all-at-once:

The LATA control register is defined as an int and so can be used as such within your code. Just be sure to only set the lowest 8 bits as those are controlling the LEDs. This means that you shouldn’t be settings LATA to anything above 255.

// Turn on the LEDs connected to RA0 and RA1 (remember your binary!) LATA = 3;

Configuring individually:

A global LATAbits variable has already been declared for you for configuring the A-pins individually. It is a struct with members that look like LATA0, LATA1, etc. So you can refer to its specific members like so:

// Turn on the LEDs connected to RA0 and RA1 LATAbits.LATA0 = 1; LATAbits.LATA1 = 1;

Timers
Timers trigger an interrupt repeatedly at a frequency you can specify. Commonly used for triggering output repeatedly like in this lab. In this lab you will use timer1 (there are several) and the provided code configures timer1 for you already. Whenever Timer1 triggers the T1Interrupt() interrupt is called. In this ISR you should increment your bounceState struct's counter member. Timer1's interrupt flag is IFS0bits.T1IF, so clear that bit before returning (assign it to 0).

The timer is configured to apply a prescalar to the processor frequency to slow it down. Timer1's prescalar is set by the register PR1, with a valid range from 1 to 65536 (or 0xFFFF) or invisibly fast to fast. For the PR1 register, 0 is a special control value and if it is set to that no timer interrupts will be triggered!

While the prescalar can adjust the frequency of the timer over a large range, even at its slowest it is 10x faster than we would like. You should account for the counter variable incrementing faster than you want within main().

**Analog to digital converters (ADC)**

Think of a volume dial; the user turns it to a value between some minimum and maximum. This is what's called an analog control: it has a large range of possible values. Hardware called an analog-to-digital converter is used to translate these analog signals into digital values that a processor can understand, namely an integer.

For this lab you will use just one of the ADCs available on the processor: ADC1. Like timer1, it has already been configured for you. It has been set to sample the voltage from the potentiometer on the development board and then store the result as an unsigned 10-bit value in the integer register ADC1BUF0 before calling the ISR ADC1Interrupt(). It repeats this process continually and as fast as possible.

The interrupt flag for ADC1Interrupt() is IFS0bits.AD1IF so be sure to clear this before returning from it.

**Integer range mapping**

Imagine writing a program which monitors the conditions of a drill bit used to dig for oil, stopping the drill string automatically if it gets over 2500 degrees Fahrenheit. Unfortunately, the temperature gauge was manufactured in Britain and gives a reading in Celsius. Your first job, then, would be to map the temperature reading to Fahrenheit.

This is quite a common mathematical operation when working with embedded systems. Luckily C's arithmetic and bitwise operators make mapping easy. In embedded systems, however, multiplications and division are very slow and would cause performance problems if you relied on them exclusively to perform a mapping. The way to work around this is to use the bit-shift operators.

Remember that the bit-shifting operations in C are specified with << for a left bit-shift and a >> for a right bit-shift. Now you may just think of bit-shifting as moving bits around, but when integers are stored as bits, what is happening to the integer? Well, since a bit-shift left moves all the bits up 1 bit to higher-valued bits, it is the same as multiplying by 2. Shift more to multiply by higher multiples of two: so bit-shifting 4 times
would multiply the integer by $2^4$ or 16. A similar process holds for bit-shifting right, just with division instead of multiplication.

Bit-shifting isn’t just used for replacing multiplication and division operations; it can also be used for removing all of the lower-ordered bits from a number. As an example if you right bit-shift 0xA5 by 4 the result would be 0xA, the highest 4 bits in that number.

For example, take an integer in the range of [0,1023] (means from 0 to 1023 and including both). You should first notice that this number is stored entirely in the lowest 10 bits of that integer. Now map that number into the range [0,31], which can be represented in 5 bits. This mapping, therefore, will just remove the lowest 5 bits of the input with a right shift and use the result.

Now when writing a mapping operation from some integer range to another integer range, you will probably need to do something a little more complex than the bit-shift in the example above. Below we list the relevant bitwise and arithmetic operators you will use and how they can be used. They will often need to be combined to get the desired result and sometimes this can take six or more operations!

Here are some common uses of the operators for manipulating bits and integers:

- **bit-shift left** multiplying by powers of 2, moving bits higher
- **bit-shift right** dividing by powers of 2, removing lower-order bits, selecting higher-order bits
- **bitwise and** clearing bits (to 0), selecting specific bits
- **bitwise or** setting bits (to 1), combining numbers
- **bitwise xor** inverting specific bits
- **bitwise not** inverting all bits
- **subtraction** reversing the ordering of a range
- **addition** offsetting a range